ECAP-2018-021

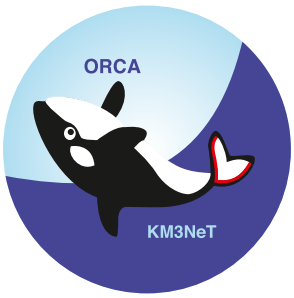
Investigating systematics for KM3NeT/ORCA using unsupervised Deep Learning

Master's Thesis in Physics

Presented by
Stefan Reck
Erlangen, September 25, 2018

Department of Physics
Erlangen Centre for Astroparticle Physics
Friedrich-Alexander-Universität Erlangen-Nürnberg

ERLANGEN CENTRE
FOR ASTROPARTICLE
PHYSICS



Supervisor: PD Dr. Thomas Eberl
Co-Supervisor: Prof. Dr. Gisela Anton

Abstract

KM3NeT/ORCA is a water-Cherenkov neutrino detector, currently under construction in the Mediterranean Sea. Its main goal will be to determine the neutrino mass hierarchy by measuring the energy- and zenith-angle dependency of the oscillation probabilities of atmospheric neutrinos after travelling through the Earth.

Deep Learning provides a promising method to analyse the signatures produced by the particles traversing the detector. A common point of critique of the popular supervised Deep Learning techniques is their dependency on simulated data. If this data contains features that deviate from measured data, networks can become sensitive to them, and their performance on measurements will fall behind expectations. Ultimately, the network might fixate on effects only present in the simulations, or become unaware of properties of measured data.

This thesis will cover an unsupervised learning approach with deep convolutional autoencoders, which tackles the problem of learning unwanted features by making it possible to train large parts of the network on measured data. By evaluating them on simulations, it is found that these networks perform between 3.7% and 5.3% worse than networks trained in a supervised fashion, but are significantly more robust against differences between datasets. If the gap in quality can be closed in the future, the autoencoder-based approach could prove to be a superior method of training networks.

Contents

1. Neutrino physics

As far as we know, neutrinos are elementary particles. They are classified as fermions and, more specifically, as leptons in the Standard Model of particle physics. Similar to the electron, the muon and the tau, neutrinos are also distinguished by their leptonic flavour: the three currently known flavour states are the electron neutrino, the muon neutrino and the tau neutrino. Unlike the three charged leptons, neutrinos have been observed to be electrically neutral. Like all leptons, they carry no strong charge either, meaning that only two possible interactions of neutrinos with other particles remain: gravitation and the weak interaction. As they are neutral to the other two elementary forces, neutrinos are more difficult to measure in a detector as compared to the other leptons.

In fact, it was assumed up until the late 1990s when the Super-Kamiokande experiment was conducted that neutrinos do not have any rest mass at all; a property that was incorporated into the Standard Model as well. As it turns out, they have a non-zero mass, albeit a very small one in comparison with the charged leptons. Only due to this presence of mass, one of the fundamental properties of neutrinos can be explained: the neutrino flavour oscillations.

1.1. Neutrino oscillations in vacuum and the mass hierarchy

The properties of a neutrino can be described by two state vectors, one containing the flavour eigenstates ν_e, ν_μ and ν_τ, and the other containing the mass eigenstates ν_1, ν_2 and ν_3. In the following, both of them are assumed to be orthonormal bases. These state vectors are coupled with the so called PMNS-Matrix $\boldsymbol{U}$, whose elements describe the mixture of mass eigenstates contained in a pure flavour eigenstate:

$$\begin{pmatrix}\nu_e\\ \nu_\mu\\ \nu_\tau\end{pmatrix} = \boldsymbol{U}\cdot\begin{pmatrix}\nu_1\\ \nu_2\\ \nu_3\end{pmatrix} = \begin{pmatrix}U_{e1} & U_{e2} & U_{e3}\\ U_{\mu1} & U_{\mu2} & U_{\mu3}\\ U_{\tau1} & U_{\tau2} & U_{\tau3}\end{pmatrix}\cdot\begin{pmatrix}\nu_1\\ \nu_2\\ \nu_3\end{pmatrix}. \tag{1.1.1}$$

This is comparable to the case of quark flavour mixing, in which the change of quark flavours is described by the CKM matrix. In contrast to the CKM matrix, the PMNS matrix of the neutrino mixing is not dominated by its diagonal elements.

Like in the case of quark mixing, the PMNS matrix is often parametrized by three weak mixing angles θ_{12}, θ_{23} and θ_{13}. In addition, there is a CP violating phase δ_{CP}, and two complex Majorana phases α_1 and α_2:

$$\boldsymbol{U} = \begin{pmatrix}1 & 0 & 0\\ 0 & c_{23} & s_{23}\\ 0 & -s_{23} & c_{23}\end{pmatrix}\begin{pmatrix}c_{13} & 0 & s_{13}e^{-i\delta_{CP}}\\ 0 & 1 & 0\\ -s_{13}e^{i\delta_{CP}} & 0 & c_{13}\end{pmatrix}\begin{pmatrix}c_{12} & s_{12} & 0\\ -s_{12} & c_{12} & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}e^{i\alpha_1/2} & 0 & 0\\ 0 & e^{i\alpha_2/2} & 0\\ 0 & 0 & 1\end{pmatrix}, \tag{1.1.2}$$

with $s_{ij} = \sin(\theta_{ij})$ and $c_{ij} = \cos(\theta_{ij})$. The Majorana phases are only physical if neutrinos are Majorana particles, meaning that they are their own antiparticle [1]. This is currently unknown, and does not affect the oscillation behaviour either way, so the phases will be omitted in the following.

When a neutrino is produced in a weak interaction in one of the three specific flavour states above, its state is, at the same time, a mixture of all the three mass eigenstates. During the propagation of the neutrino, this particular mixture will change, which in turn will result in a non-zero probability of measuring a different flavour eigenstate than the one it started as. The fact that the measurable flavour of a neutrino can change over time is called neutrino oscillation.

This can be understood by calculating the probability amplitude of the mass eigenstates during propagation. Let $|\nu(0,0)\rangle$ be a mass eigenstate at time $t = 0$ and position $\vec{x} = 0$. The mass states are eigenstates of the Hamiltonian [2], and solving the Schrödinger equation for them implies that the state has evolved like a plain wave after propagating for a time t and distance L in vacuum:

$$|\nu(L,t)\rangle = e^{-i(E\cdot t - p\cdot L)} \, |\nu(0,0)\rangle \,, \tag{1.1.3}$$

with the energy E and momentum p of the particle. Natural units with $c = \hbar = 1$ are used for this equation as well as the following. For neutrinos, the ultrarelativistic approximation of the momentum is often appropriate, due to their typically small mass to energy ratio: An upper limit to the summed mass of the three mass states has been determined to be 0.17 eV at a 95% confidence level [3], while the typical energies of the neutrinos measured by the ORCA neutrino telescope are in the GeV range. Then, the following approximation can be made:

$$p = \sqrt{E^2 - m^2} \simeq E - \frac{m^2}{2 \cdot E}, \quad \text{for } \frac{m}{E} \ll 1. \tag{1.1.4}$$

With this, equation 1.1.3 can be written as:

$$|\nu(L,t)\rangle = e^{-iE(t-L)} e^{-i\frac{m^2 L}{2E}} \, |\nu(0,0)\rangle \,. \tag{1.1.5}$$

From this equation, it can be seen that the squared mass m^2 of the eigenstate is a defining factor for the speed at which its probability amplitude oscillates. If a neutrino is produced with a specific energy E, and as a mixture of two mass eigenstates of masses m_1 and m_2, their phase difference γ will similarly amount to:

$$\gamma = \frac{(m_1^2 - m_2^2) \cdot L}{2E} \equiv \frac{\Delta m^2 L}{2E}, \tag{1.1.6}$$

with the squared-mass difference $\Delta m^2 = m_1^2 - m_2^2$. Since the mass eigenstates are coupled to the flavour eigenstates by the PMNS matrix (equation 1.1.2), a change in the probability amplitude of the mass eigenstates implies such a change for the flavour states, too. A direct consequence from the above equation is the fact that non-zero masses of neutrinos are a requirement for the existence of neutrino oscillations, since identical masses would imply a squared-mass difference of zero.

Both the phase difference γ and the neutrino mixing angles define the properties of neutrino oscillations. For this, consider the simple case of the two flavour scenario, in which one of both the neutrino flavours and the mass eigenstates is disregarded. This results in a two by two mixing matrix, which is defined by just one mixing angle θ:

$$\begin{pmatrix} \nu_e \\ \nu_\mu \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} \nu_1 \\ \nu_2 \end{pmatrix}. \tag{1.1.7}$$

In this case, the transition amplitude $A^{(2\nu)}(L,t)$ for a neutrino that was generated as an electron neutrino at time and position zero, and then propagated for a timespan t and distance

L, is given by:

$$\begin{aligned} A^{(2\nu)}(L,t) &= \langle \nu_\mu |\ \nu_e(L,t)\rangle \\ &= (-\sin\theta\cdot\langle\nu_1| + \cos\theta\cdot\langle\nu_2|\ )\cdot(\cos\theta\cdot|\nu_1(L,t)\rangle + \sin\theta\cdot|\nu_2(L,t)\rangle) \\ &= (-\sin\theta\cdot\langle\nu_1| + \cos\theta\cdot\langle\nu_2|\ )\cdot e^{-iE(t-L)}e^{-i\frac{m_1^2 L}{2E}}\cdot\left(\cos\theta\cdot|\nu_1\rangle + \sin\theta\cdot e^{i\gamma}|\nu_2\rangle\right) \\ &= e^{-iE(t-L)}e^{-i\frac{m_1^2 L}{2E}}\cdot\left(-\sin\theta\cos\theta + \sin\theta\cos\theta e^{i\gamma}\right), \end{aligned} \tag{1.1.8}$$

with the use of $\langle\nu_i|\ \nu_j\rangle = \delta_{ij}$, the Kronecker delta. The basis was changed from the flavour basis to the mass basis before the second line using equation 1.1.7. Equations 1.1.5 and 1.1.6 were inserted going from the second line to the third line to express the propagation of the mass eigenstates. The transition probability in the two flavour case is given by the absolute square of the transition amplitude from above:

$$\begin{aligned} P^{(2\nu)}_{\nu_e\to\nu_\mu}(L,t) = |A^{(2\nu)}(L,t)|^2 &= \sin^2(2\theta)\sin^2(\gamma/2) \\ &= \sin^2(2\theta)\sin^2\left(\frac{\Delta m^2 L}{4E}\right) \\ &\equiv \sin^2(2\theta)\sin^2\left(\frac{L}{L_0}\right). \end{aligned} \tag{1.1.9}$$

The length scale of the oscillation is defined by $L_0 = 4E/\Delta m^2$, while the oscillation amplitude $\sin^2(2\theta)$ is defined by the mixing angle. The amplitude can be between zero ($\theta = 0°$), where no mixing occurs, and one ($\theta = 45°$), where the mixing is at its strongest.

In the full three flavour case, the calculations become more lengthy, but are principally done in the same way as in the two flavour model above. If Δm^2_{12} is assumed to be much smaller than Δm^2_{13} and Δm^2_{23}, which is consistent with the current state of observations, the transition probability in vacuum for a muon neutrino to an electron neutrino can be approximated in the three flavour picture by:

$$P^{(3\nu)}_{\nu_\mu\to\nu_e}(L,t) \approx \sin^2(\theta_{23})\sin^2(2\theta_{13})\sin^2\left(\frac{\Delta m^2_{31} L}{4E}\right), \tag{1.1.10}$$

as shown in [4]. Similar to the two flavour case, the amplitude of the oscillation is defined by the mixing angles and its scale by the squared-mass difference and the energy of the neutrino. Since Δm^2_{31} appears as an argument of the $\sin^2$ function, which is symmetric around zero, the transition probability is actually insensitive to the sign of the squared-mass difference. However, the sign does come into effect when matter is present, as will be shown in the next chapter.

Based on this, the sign of Δm^2_{12} has been determined to be positive via measurements of solar neutrinos, for example in the SNO experiment [5]. The sign of Δm^2_{31} is still unknown, so there are two distinct ways of ordering the masses of the mass eigenstates: The case where $m_1 < m_2 < m_3$, which is called the normal hierarchy, and the case where $m_3 < m_1 < m_2$, the inverted hierarchy. A scheme of these orderings is shown in figure 1.1. Determining the correct mass hierarchy is a primary goal of the neutrino detector ORCA.

The currently assumed values for the oscillation parameters as of November 2017 are shown in figure 1.2. Noteably, the squared-mass differences Δm^2_{31} and Δm^2_{32} are larger than Δm^2_{21} by a factor of 30. Since the length scale at which oscillations happen is defined by $2E/\Delta m^2$ (e.g. equation 1.1.6), this means that in cases where a measurement is primarily sensitive to one of the oscillation scales, simplified models can be used to describe the oscillations in total [2]. It is worth noting that, since the actual masses of the eigenstates are not known, there could also be the case where $m_1 \cong m_2 \cong m_3$, which is called the quasi-degenerate ordering.

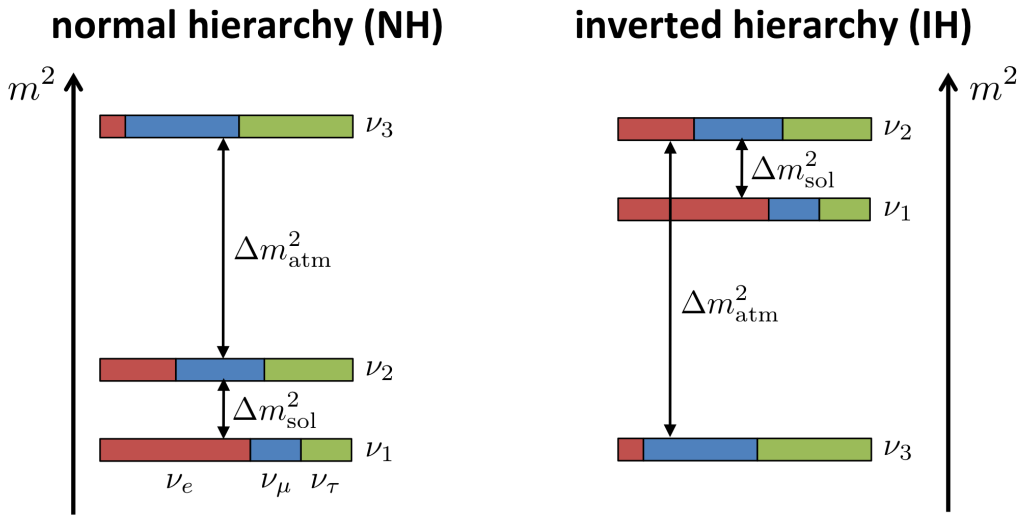


Figure 1.1.: Scheme of the two different ways of ordering the neutrino masses. The size of the colored segments of the bars indicate the fraction of the respective flavours in that mass state. This fraction is given by the corresponding elements of the PMNS matrix, e.g. the fraction of red for ν_1, labelled as ν_e, is given by $|U_{e1}|^2$ [6].

Parameter	best-fit	3σ
Δm^2_{21} [10^{-5} eV ^{2}]	7.37	$6.93 - 7.96$
$\Delta m^2_{31(23)}$ [10^{-3} eV ^{2}]	2.56 (2.54)	$2.45 - 2.69$ $(2.42 - 2.66)$
$\sin^2\theta_{12}$	0.297	$0.250 - 0.354$
$\sin^2\theta_{23}$, $\Delta m^2_{31(32)} > 0$	0.425	$0.381 - 0.615$
$\sin^2\theta_{23}$, $\Delta m^2_{32(31)} < 0$	0.589	$0.384 - 0.636$
$\sin^2\theta_{13}$, $\Delta m^2_{31(32)} > 0$	0.0215	$0.0190 - 0.0240$
$\sin^2\theta_{13}$, $\Delta m^2_{32(31)} < 0$	0.0216	$0.0190 - 0.0242$
δ/π	1.38 (1.31)	2σ: (1.0 - 1.9) (2σ: (0.92-1.88))

Figure 1.2.: The best-fit values and 3σ allowed ranges of the 3-neutrino oscillation parameters, derived from a global fit of the current neutrino oscillation data as of 2017. Δm^2 is defined here as $\Delta m^2 = m_3^2 - (m_1^2 + m_2^2)/2$. The values are given for the case of the normal hierarchy $\Delta m^2_{31(32)} > 0$, and for the inverted hierarchy $\Delta m^2_{31(32)} < 0$ where applicable. The value for $\delta = \delta_{CP}$ is given in brackets for the inverted hierarchy [7].

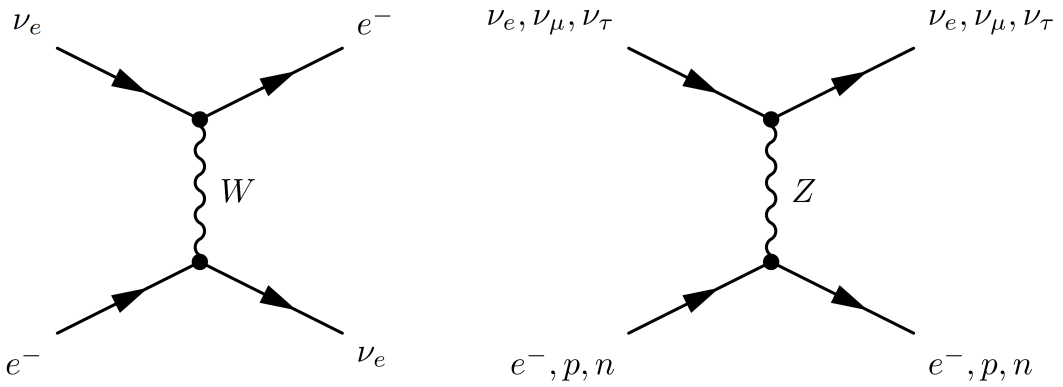


Figure 1.3.: Feynman diagrams of two different neutrino forward scattering channels. The CC interaction on the left is only possible for the electron neutrino, while the NC scattering on the right is possible for neutrinos of all flavours [2].

1.2. Neutrino oscillations in matter

For the calculations above, it was assumed that the neutrinos propagate through vacuum. If matter is present in the path of the neutrino, this can change the properties of the oscillations significantly by amplifying or even suppressing the probability of flavour transitions. In matter, where quarks and electrons are present, different interaction channels for the different neutrino flavours exist: All the three differently flavoured neutrinos can undergo an elastic scattering with electrons, protons or neutrons under the exchange of a Z^0 boson (figure 1.3), but the electron neutrino in particular can also forward scatter with an electron in a CC-interaction by exchanging a W boson, introducing an imbalance in the propagation of the different flavour states. In vacuum, the mixing of flavours was due to the difference in the propagation of the mass eigenstates. The situation in matter is similar, as a disparity in the propagations of the flavour states allows for transitions between mass eigenstates.

The additional CC interaction channel results in an additional potential for the electron neutrinos of $V = +\sqrt{2}G_F N_e$, with the Fermi constant G_F and the electron density of the surrounding matter N_e. For antineutrinos, the sign of the potential V is changed to a minus. The modified Hamiltonian, including both the vacuum Hamiltonian and this additional potential, can be diagonalized to yield the mass eigenstates and the effective mixing angles in matter [4]. For the sake of simplicity, the two flavour case is considered again, in which the third flavour and mass eigenstate are disregarded. Similar to the two flavour mixing matrix in vacuum from equation 1.1.7, one can establish a mixing matrix for the effective mass eigenstates in matter ν_{1m}, ν_{2m} by introducing an effective mixing angle θ_m:

$$\begin{pmatrix} \nu_e \\ \nu_\mu \end{pmatrix} = \begin{pmatrix} \cos\theta_m & \sin\theta_m \\ -\sin\theta_m & \cos\theta_m \end{pmatrix} \cdot \begin{pmatrix} \nu_{1m} \\ \nu_{2m} \end{pmatrix}. \tag{1.2.1}$$

Ultimately, the transition probability for an electron neutrino to a muon neutrino in matter can be written in analogy to the vacuum case of equation 1.1.9 as

$$P_m^{(2\nu)}(\nu_e \to \nu_\mu) = \sin^2(2\theta_m)\sin^2\left(\frac{L}{L_m}\right). \tag{1.2.2}$$

The oscillation amplitude $\sin^2(2\theta_m)$ is defined by the oscillation amplitude in vacuum $\sin^2(2\theta_m)$,

the squared-mass difference D in vacuum and the matter parameter $A = 2Vp$ as

$$\sin 2\theta_m = \sin 2\theta \cdot \frac{1}{\sqrt{\left(\frac{A}{D} - \cos 2\theta\right) + \sin^2 2\theta}}. \tag{1.2.3}$$

The effective oscillation length in matter L_m is given by the oscillation length in vacuum L_0 as

$$L_m = L_0 \cdot \frac{1}{\sqrt{\left(\frac{A}{D} - \cos 2\theta\right) + \sin^2 2\theta}}. \tag{1.2.4}$$

Both the oscillation amplitude and the oscillation length are changed due to the presence of matter as compared to the vacuum case. The amplitude is a Breit-Wigner distribution of A/D, and reaches its maximum of one when $A = D\cos 2\theta$, where the mixing angle $\theta_m = 45°$. In this case of a resonance, the probability to measure an electron neutrino is oscillating between zero and one, independent of the value of the vacuum mixing angle. The matter parameter $A = \pm 2\sqrt{2}G_F N_e p$ depends on both the electron density of the traversed matter, as well as the momentum of the neutrino. Therefore, the mixing angle is maximized only for specific values of electron densities and neutrino energies. In contrast, if A/D is very large, which is the case for a high electron density, the transition amplitude becomes almost zero, so that the transition is suppressed. Finally, the equations of the vacuum case are recovered when $A = 0$.

From the equations above, it can be seen that the sign of the squared-mass difference in vacuum D is actually relevant for the oscillations in matter, since it defines the value of A at which the resonance is present. Taking into account that the sign of A is inverted for antineutrinos, the resonance only appears for neutrinos in the case of the normal hierarchy, and for antineutrinos in the case of the inverted hierarchy.

Similar equations can be deduced for the three flavour case [8]. The resonant neutrino energy for the ν_e and ν_μ transitions when traversing the Earth can be calculated to be at

$$E_{res} \approx 7\text{GeV} \left(\frac{4.5\ \text{g/cm}^3}{\rho}\right) \left(\frac{\Delta m^2_{31}}{2.4 \cdot 10^{-3}\ \text{eV}^2}\right) \cos 2\theta_{13}, \tag{1.2.5}$$

written as a function of the matter density ρ, the squared-mass difference Δm_{31} and the mixing angle θ_{13}. The units in this equation were chosen to reflect reasonable assumptions for the scale of the squared-mass difference and the density in the Earth. Inserting typical values for the density of different regions in the Earth, the resonant energy for neutrinos travelling through the planet can be found to range between 7 GeV when in the mantle and 3 GeV when in the core. Neutrinos in this energy range happen to get produced in the atmosphere of the Earth due to incident cosmic radiation, as explained in more detail in the following chapter. After they have surpassed the Earth, they can then be measured by a neutrino telescope like ORCA to observe how much of an influence the matter had on the transition amplitudes. From this, the sign of the squared-mass difference and consequently the true neutrino mass ordering can be determined.

This effect of the resonance can be seen in figure 1.4, where the transition probabilities are plotted as a function of the neutrino energy for the cases of the normal and the inverted hierarchy, as well as for neutrinos and antineutrinos. The influence of matter on the transition probability of neutrinos in the normal hierarchy is very similar to that of antineutrinos in the inverted hierarchy. One might be worried whether a non-magnetizing neutrino detector like ORCA, which cannot distinguish between single neutrino and antineutrino events, can be used at all to determine the neutrino mass hierarchy. However, the cross section of neutrino-nucleon

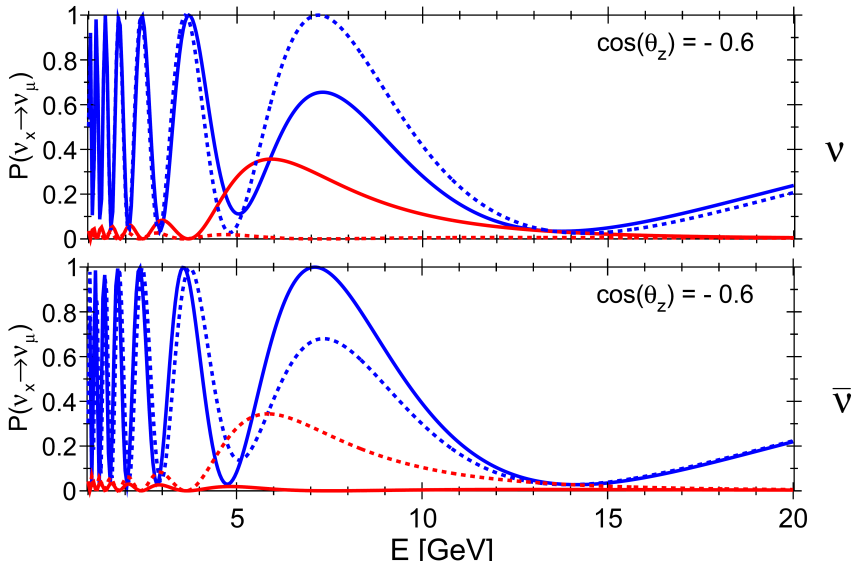


Figure 1.4.: Oscillation probabilities of the $\nu_\mu \to \nu_\mu$ transition in blue, and the $\nu_e \to \nu_\mu$ transition in red for atmosheric neutrinos travelling through the Earth and arriving at a zenith angle of $\cos\theta_z = -0.6$. The solid (dashed) lines are for the normal (inverted) hierarchy. The plots were made for neutrinos (top) and antineutrinos (bottom) (adapted from [8]).

interactions is higher than that of antineutrinos by a factor of about two. Also, the spectrum of atmospheric neutrinos consists of different proportions of electron and muon neutrinos. Due to this, the different mass hierarchies will still result in different net event rates in the detector, as shown in figure 1.5.

1.3. Atmospheric neutrinos

Atmospheric neutrinos are well suited to determine the mass hierarchy, since the energy range in which their flux is large, together with the typical densities of the Earth, are both close to the values at which the resonance of neutrino oscillations in matter occurs (see equation 1.2.5). From the effect of this resonance, the sign of the squared-mass difference Δm_{31} can be determined with the ORCA detector.

The neutrinos produced in the atmosphere of the Earth are a product of interactions between cosmic rays and atmospheric nuclei. Cosmic radiation is made up of highly energetic particles, which get mostly produced outside of the solar system. Their exact origin is unknown, yet there is evidence suggesting that supernova remnants are a major source [5]. The constituents of cosmic rays are energy dependent, but in the GeV range they are mostly composed of protons and to about 5% of Helium and even heavier nuclei [9]. Their flux decreases exponentially towards higher energies.

When a cosmic ray particle interacts with an atmospheric nucleus, a chain reaction of particle interactions is started. The mesons produced in the collision of a cosmic ray particle in the GeV range are mostly pions and, to a lesser extent, kaons. These mesons can then either collide again, or decay into other particles. In turn, these might decay again before hitting Earth's surface if they are unstable. For example, the dominant decay chains of charged pions

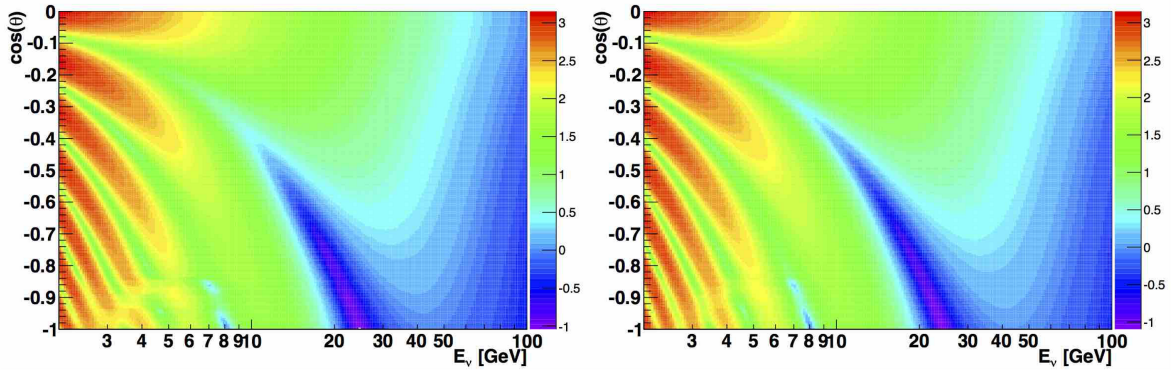


Figure 1.5.: Combined event rate of muon and anti muon neutrinos, in units of $1/(\text{GeV} \cdot \text{y} \cdot \text{sr})$ over the neutrino energy and the zenith angle for the normal hierarchy (left) and the inverted hierarchy (right) [8].

are:

$$\begin{aligned} \pi^+ &\to \mu^+ + \nu_\mu & \qquad \pi^- &\to \mu^- + \bar{\nu}_\mu \\ \mu^+ &\to e^+ + \nu_e + \bar{\nu}_\mu & \qquad \mu^- &\to e^- + \bar{\nu}_e + \nu_\mu. \end{aligned} \tag{1.3.1}$$

The most likely decay channel of the kaon is similar to the one above, but it can also decay into a variety of charged and neutral pions. The particle cascade which is caused by an incident cosmic ray particle is called air shower; an example of this is depicted in figure 1.6. From the decay process above, it can be seen that the ratio of muon to electron neutrinos and antineutrinos is roughly 2:1, if all muons in the shower decay before reaching the ground. If their energy is high enough, they might actually reach Earth's surface before decaying, in which case they quickly loose most of their energy. After that, the products of their decay will have very low energies as well. This leads to an even higher ratio of muon to electron neutrinos in the high GeV range, as can be seen in the simulation studies on this topic by Honda et al. [10]. In contrast, the cross section of interactions of neutrinos in the GeV range travelling through the Earth is negligible.

Neutrinos are often measured by detecting particles that were produced from neutrino-nucleon interactions. A possible particle produced in such an interaction is a muon, which leaves a track-like signal of Cherenkov radiation as it passes through the detector. However, equation 1.3.1 shows that muons also get produced in the air showers above the detector and hence pose a significant background to the measurement of neutrinos from below. To reduce this background, neutrino telescopes are often placed underneath a large mass of matter, which shields the set-up from atmospheric muons. In the case of ORCA, this shielding is achieved by building the detector in the Mediterranean sea at a depth of 2450 meters. Down going muons will still be measurable, however, since on the one hand, the energy loss of muons is only about 0.25 GeV per meter of traversed water, meaning that highly energetic atmospheric muons can still be in the GeV range even at the depth of ORCA. On the other hand, atmospheric neutrinos coming from above can also produce muons in the proximity of the detector.

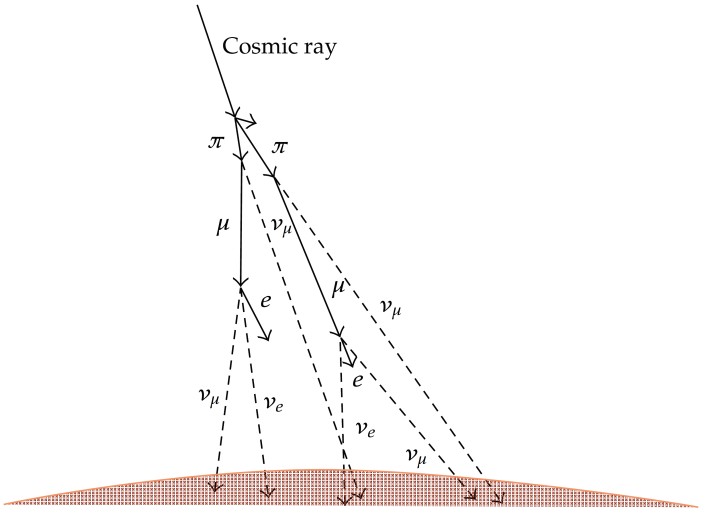


Figure 1.6.: Scheme of an air shower in the atmosphere of the Earth, in which atmospheric neutrinos are produced [9].

1.4. The ORCA neutrino detector

The KM3NeT/ORCA neutrino telescope detects neutrinos indirectly by measuring Cherenkov radiation that is emitted by secondary particles which are created in the interactions of neutrinos with nucleons. This is achieved by instrumenting a large volume of sea water with photomultipliers, which detect the radiation emitted by passing particles. The resulting patterns in the detector depend on the type of interaction, and are shown in figure 1.7. These events can be classified into two categories: track-like and shower-like events.

- Track-like event signatures are caused by muons, which are typically generated in a CC interaction of a muon neutrino, or alternatively by the decay of a tau which was generated by a tau neutrino. If the produced muon travels faster than the local speed of light in water $c_W \approx 0.75 \cdot c$, it will emit electromagnetic radiation known as Cherenkov radiation at roughly a 42° angle relative to its trajectory. This results in a track-like signature in the detector. The energy loss of muons in the GeV range travelling through water is small enough to let them to produce long tracks, which allow for a good estimation of the direction it is travelling in. As the direction of the muon is closely related to that of the muon neutrino from which it was produced, this allows for a good reconstruction of the trajectory of the original neutrino as well.
- Shower- or cascade-like events are produced in the CC interactions of electron or tau neutrinos, or generally in NC neutrino interactions. The resulting hadronic shower is fairly compact, and contains a large part of the energy of the original neutrino, allowing for a good reconstruction of its energy.

To detect the event signatures described above, 31 photomultipliers at a time are arranged in a single digital optical module (DOM), a water-proof 43 centimetres diameter glass sphere containing all the required electronics. 18 of these DOMs are aligned horizontally in a detection line with a 23 metre vertical spacing in between, being kept in place by two strings at the sides, and anchored to the sea floor. While the sea has a depth of 2450 meters at the position of the detector, the first DOM in each line will be positioned at a height of 40 meters above that, resulting in a total length of each string of about 200 meters. A buoy at the top of each line provides additional buoyancy to keep the line in an upright position.

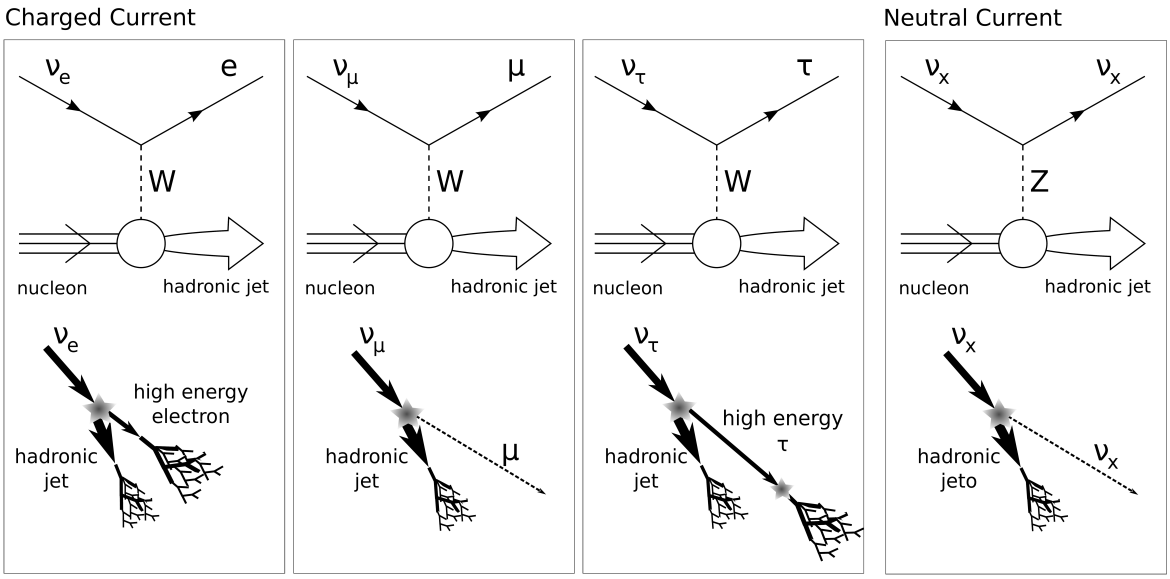


Figure 1.7.: The different event signatures left behind in a water based neutrino detector, depending on the interaction which took place [11].

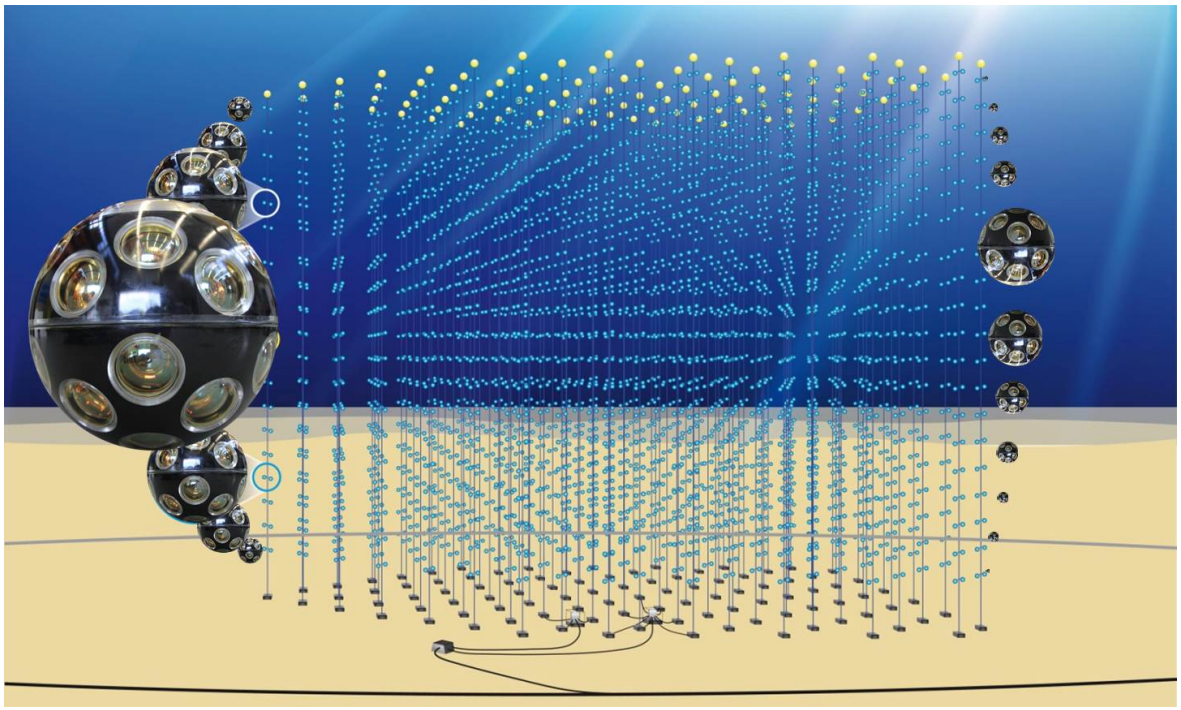


Figure 1.8.: An artists impression of the full KM3NeT/ORCA building block with 115 lines. The individual DOMs, each containing 31 photomultipliers, are shown along the edge of the image [12].

The first phase of ORCA, which is currently under construction, will feature seven of these detection lines, with an average horizontal spacing of 23 meters. The final phase is targeted to be completed in 2020, and will have a total of 115 lines, instrumenting about 8 megatons of sea water. An artists impression of the complete detector is shown in figure 1.8.

Since the neutrino telescope is not completed yet, the data analysis in this thesis relies on simulations, which generate Monte Carlo events for the full detector layout described above.

2. Artificial neural networks

When analysing the data from the ORCA neutrino detector, a prediction of the properties of the original particle is made based on the data measured by the detector. For example, one might be interested in its energy or the direction it is travelling in. Ideally, the prediction should be a good approximation of the real value of the property.

Mathematically speaking, this prediction is a function $\hat{F}(\boldsymbol{x_{in}})$ of the measured data from the detector $\boldsymbol{x_{in}}$, whose function values are as close as possible to the values of this desired property in terms of some specific metric. In other words, a close approximation of the hypothetical ideal prediction function F is sought, which is assumed to always reconstruct the property correctly. In the case of ORCA, the process of finding such an approximative function is difficult, since it is a high-dimensional task. An especially promising way of tackling problems of this kind is the use of artificial neural networks and especially Deep Learning, which has proven to be very successful in the field of image recognition in the past years (e.g. achieving super-human performance in the board game Go [13]; image recognition for the image net challenge [14]).

This chapter will give an overview of the theoretical background of artificial networks, as well as the practical implementation of Deep Learning for analysing ORCA data.

2.1. The basics of Deep Learning

The Deep Learning techniques applied in this thesis are based on artificial neural networks, a term coined by the fact that they were originally inspired by the biological function of brain cells. In this spirit, an artificial neural network is a system of connected nodes named artificial neurons, each of them applying an arbitrary function - the activation function - to its inputs. The output of this operation can then again be fed into other neurons, or can be taken as the output of the network as a whole.

In practice, a neuron will usually apply the following operation to its N inputs $\{x_j | j = 1, ..., N\}$:

$$f(x_1, x_2, ..., x_N) = \Theta\left(\sum_{j=1}^{N} \omega_j x_j + b\right) \tag{2.1.1}$$

That is, a sum over every input to the neuron $x_j \in \mathbb{R}$ is taken, each weighted with a real number ω_j. A constant, the bias b, is added, before applying the activation function Θ.

Often times, the neurons are ordered in layers. If the input to a certain layer consist only of neurons from the previous layer, it is called a feed-forward network. In this case, the output $x_i^{(n)}$ of neuron i in the n-th layer is given as a function from the output $x_j^{(n-1)}$ from the N neurons in the previous $(n-1)$-th layer by the following expression:

$$x_i^{(n)}\left(x_1^{(n-1)}, x_2^{(n-1)}, ..., x_N^{(n-1)}\right) = \Theta\left(\sum_{j=1}^{N} \omega_{i,j}^{(n)} x_j^{(n-1)} + b_i^{(n)}\right), \tag{2.1.2}$$

with $\omega_{i,j}^{(n)}$ denoting the weight between neuron j in the $(n-1)$-th layer, and neuron i in the n-th layer. With this terminology, the output $\vec{x}^{\,(n)}$ of all neurons in layer n can be compactly written in vector notation with the weight matrix $\boldsymbol{\omega}$ as follows:

$$\vec{x}^{\,(n)}\left(\vec{x}^{\,(n-1)}\right) = \Theta\left(\boldsymbol{\omega}^{(n)} \cdot \vec{x}^{\,(n-1)} + \vec{b}^{\,(n)}\right), \tag{2.1.3}$$

where Θ is applied element-wise to a vector. The weights in the matrix $\boldsymbol{\omega}^{(n)}$ and the biases $\vec{b}^{\,(n)}$ are the free parameters of the n-th layer of the network.

Apart from the input and the output, all layers inside the network are called hidden layers. As an example, consider the simple case of a network with just one hidden layer: The input $\vec{x}^{\,(0)}$ is taken as the 0-th layer, and the output $\vec{y}^{\,(2)}$ of the whole network is then given by iteratively inserting equation 2.1.3 :

$$\vec{y}^{\,(2)} = \Theta\left(\boldsymbol{\omega}^{(2)} \cdot \vec{h}^{\,(1)} + \vec{b}^{(2)}\right) = \Theta\left(\boldsymbol{\omega}^{(2)} \cdot \Theta\left(\boldsymbol{\omega}^{(1)} \cdot \vec{x}^{\,(0)} + \vec{b}^{(1)}\right) + \vec{b}^{(2)}\right), \tag{2.1.4}$$

with $\vec{h}^{\,(1)}$ being the output of the hidden layer.

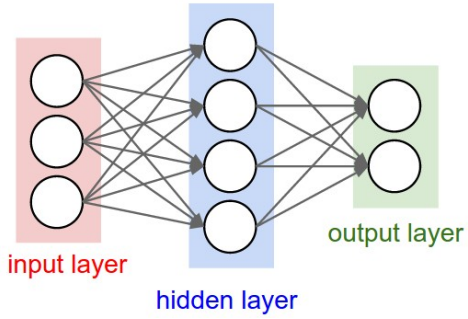


Figure 2.1.: Scheme of a shallow, fully connected feed-forward network. Every circle represents a neuron, and every line connecting two neurons stands for the corresponding weight. Connections only exist between neurons in subsequent layers [15].

A network with only a single hidden layer, like the network defined by the function above, is called shallow. A scheme of this network is shown in figure 2.1. If Θ is chosen to be a non-linear function, a network with this structure can be used to approximate any continuous function to arbitrary precision, as long as there are enough neurons in the network [16]. For this, the free parameters of the network - that is, the weights $\boldsymbol{\omega}$ and the biases $\vec{b}$ - are chosen so as to reproduce the desired function.

When it comes to practical application of networks, stacking multiple hidden layers, and therefore making the network deeper, can improve the ability of the network to reproduce the desired function. This approach is called Deep Learning.

2.2. Back propagation and gradient descent

As mentioned in the introduction to this chapter, we seek to approximate the theoretical, ideal prediction function $F(\boldsymbol{x_{in}})$, which predicts certain properties of the original particle as a function of the measured data $\boldsymbol{x_{in}}$ from the detector. If this approximation is realised with the help of artificial neural networks, the approximation function $\hat{F}_{\boldsymbol{\omega},\vec{b}}(\boldsymbol{x_{in}})$ is defined via its free parameters, the weights $\boldsymbol{\omega}$ and biases $\vec{b}$ of all the layers in the network. These free parameters then have to be chosen so as to closely approximate the desired ideal prediction function.

Usually, this is achieved in an iterative process by assessing the quality of the prediction of the network on a small sample of so called labelled data, for which the desired output is known. In Machine Learning, these sets of samples are refereed to as batches. The distance between the target output and the actual output of the network is measured with a problem-specific metric. Then, the weights and biases of the network are tweaked to reduce the metric distance, and the quality is reassessed. This process of training the network on data samples is repeated over and over until the metric distance over the whole dataset reaches its minimum.

Since neural networks often contain millions of free parameters, many training steps are needed before the network is fully converged, and a large pool of labelled data is required. For the ORCA data analysis in this thesis, it is obtained by simulating neutrino events with the known properties of the detector.

A very common technique to optimize the quality of a network is called back propagation, and is an important concept of Deep Learning. For an input to the network $\vec{x}^{\,(0)}$, the network predicts the desired property of the particle to be $\hat{F}\left(\vec{x}^{\,(0)}\right)$, whereas the target output - the actual property of the particle - is given by $F\left(\vec{x}^{\,(0)}\right)$.

Note, that the input $\vec{x}^{\,(n)}$ is written here as a vector, but this does not imply that the input necessarily has a one-dimensional data structure. It could as well be a two-dimensional image or data of even higher dimension, all of which can be linearised into a 1-D vector. Regardless of the dimension, the connections between a layer and the next can still be described by a two-dimensional weight matrix: each of its elements defines the connection between exactly two neurons, independent of each of their positions in the potentially higher dimensional data structure. For this, all neurons can be assigned a unique number, which is used to identify both their position in the vector $\vec{x}^{\,(n)}$, and their corresponding column or row in the weight matrix.

How well the network managed to approximate the desired output is defined by a cost function C, which gives the metric distance between two output vectors, depending on the weights and biases of the network. A popular choice for this distance is the squared vector norm difference, in which case the cost function for a specific input $\vec{x}^{\,(0)}$ is given by:

$$C(\hat{F}_{\boldsymbol{\omega},\vec{b}}, \vec{x}^{\,(0)}) = \|\hat{F}_{\boldsymbol{\omega},\,\vec{b}}\left(\vec{x}^{\,(0)}\right) - F\left(\vec{x}^{\,(0)}\right)\|^2, \tag{2.2.1}$$

with the vector norm $\|.\|$ and the true output $F\left(\vec{x}^{\,(0)}\right)$. This cost function is also called the mean squared error. In principal though, the cost function can be defined arbitrarily, and the best choice is often problem-specific. The lower the value of the cost function is for a certain set of free parameters, the better is the prediction of the network. Therefore, the goal of the training process is to find a set of parameters which minimizes the cost function for the whole training dataset, which requires calculating the above cost function for all samples in the dataset. Since the datasets used for training networks tend to be quite large, the cost function for all samples in the dataset is usually approximated by the cost function for a small batch of training data, so that the calculation can be achieved in a reasonable time frame.

After the cost function of the network for the dataset was computed (or approximated), the weights and biases of the network will be adjusted slightly to decrease the value of the cost function via the method of gradient descent. For this, the free parameters are changed in the direction of the negative gradient of the cost function:

$$\boldsymbol{\omega} \mapsto \boldsymbol{\omega} - \eta \frac{\partial C(\hat{F}_{\boldsymbol{\omega},\vec{b}})}{\partial \boldsymbol{\omega}}, \tag{2.2.2}$$

and similar for the biases $\vec{b}$. The parameter η defines the step size of the descent, and is also called the learning rate.

By applying the chain rule, the derivative of the cost function becomes

$$\frac{\partial C(\hat{F}_{\boldsymbol{\omega},\vec{b}})}{\partial \boldsymbol{\omega}} = \frac{\partial C(\hat{F}_{\boldsymbol{\omega},\vec{b}})}{\partial \hat{F}_{\boldsymbol{\omega},\vec{b}}} \cdot \frac{\partial \hat{F}_{\boldsymbol{\omega},\vec{b}}}{\partial \boldsymbol{\omega}}. \tag{2.2.3}$$

The first factor depends on the choice of the cost function, and, for the squared vector norm distance described above, amounts to

$$\frac{\partial C(\hat{F}_{\boldsymbol{\omega},\vec{b}})}{\partial \hat{F}_{\boldsymbol{\omega},\vec{b}}} = 2\|\hat{F}_{\boldsymbol{\omega},\,\vec{b}}\left(\vec{x}^{\,(0)}\right) - F\left(\vec{x}^{\,(0)}\right)\|. \tag{2.2.4}$$

The second factor is the derivative of the output of the network with respect to a certain weight inside of the network. The output is a known function of the input (an example was given in equation 2.1.4 for a shallow network), so the derivative can be calculated analytically.

In fact, just like the forward pass through the network, the backpropagation can also be calculated iteratively on a per-layer basis. For this, consider the general case of calculating the derivative of layer n: Let $\omega^\star$ be a weight or a bias in a layer somewhere in the network before the current layer n. The output of a specific neuron in the current layer n is given as a function of the outputs of the previous layer $n-1$ by

$$x_i^{\,(n)}\left(\vec{x}^{\,(n-1)}\right) = \Theta\left(\boldsymbol{\omega}^{(n)} \cdot \vec{x}^{\,(n-1)} + \vec{b}^{\,(n)}\right)_i = \Theta\left(\sum_{j=1}^{N} \omega_{i,j}^{(n)} x_j^{(n-1)} + b_i^{(n)}\right), \tag{2.2.5}$$

similar to equation 2.1.3. Note that the output of the previous layer $\vec{x}^{\,(n-1)}$ depends on all the weights and biases before that layer in the network, including the target $\omega^\star$. The weights $\boldsymbol{\omega}^{(n)}$ and biases $\vec{b}^{\,(n)}$, however, do not. Therefore, the derivative of the above expression is given by

$$\frac{\partial x_i^{\,(n)}}{\partial \omega^\star} = \Theta'\left(\sum_{j=1}^{N} \omega_{i,j}^{(n)} x_j^{(n-1)} + b_i^{(n)}\right) \cdot \sum_{k=1}^{N} \omega_{i,k}^{(n)} \cdot \frac{\partial x_k^{\,(n-1)}}{\partial \omega^\star}, \qquad \omega^\star \text{ not in } \boldsymbol{\omega}^{(n)}. \tag{2.2.6}$$

This equation expresses the derivative of the output of the current layer through the output from the previous one. It can be inserted into itself repeatedly, until the layer m containing the target parameter $\omega^\star$ is reached. Depending on whether it is a weight or a bias, the derivative is then given by one of the following:

$$\frac{\partial x_i^{\,(m)}}{\partial \omega^\star} = \Theta'\left(\sum_{j=1}^{M} \omega_{i,j}^{(m)} x_j^{(m-1)} + b_i^{(m)}\right) \cdot \delta_{i,l}\; x_k^{\,(m-1)}\,, \qquad \omega^\star = \omega_{l,k}^{(m)} \tag{2.2.7}$$

$$\frac{\partial x_i^{\,(m)}}{\partial \omega^\star} = \Theta'\left(\sum_{j=1}^{M} \omega_{i,j}^{(m)} x_j^{(m-1)} + b_i^{(m)}\right) \cdot \delta_{i,l}\,, \qquad \omega^\star = b_l^{(m)}, \tag{2.2.8}$$

with the Kronecker delta δ. By introducing the definition

$$\boldsymbol{M}_{i,k}^{(n)} \equiv \Theta'\left(\sum_{j=1}^{N} \omega_{i,j}^{(n)} x_j^{(n-1)} + b_i^{(n)}\right) \cdot \omega_{i,k}^{(n)}, \tag{2.2.9}$$

equation 2.2.6 can be compactly written in matrix and vector notation again:

$$\frac{\partial \vec{x}^{\,(n)}}{\partial \omega^\star} = \boldsymbol{M}^{(n)} \cdot \frac{\partial \vec{x}^{\,(n-1)}}{\partial \omega^\star}, \qquad \omega^\star \text{ not in } \boldsymbol{\omega}^{(n)}. \tag{2.2.10}$$

The matrix $\boldsymbol{M}^{(n)}$ defines the step between a layer n and the previous one in the network during the process of backpropagation. It is independent of the target parameter $\omega^\star$, provided that this variable is not in the current layer n. This makes the backpropagation algorithm very computationally efficient, since most calculations have to be done only once per input for the whole network, and not specifically for every single weight or bias. Furthermore, since both the forward and the backward pass through the network require the multiplication of large and many matrices, the preferred hardware to perform these calculations on are Graphics Processing Units (GPUs), which are ideal for this task due to their high memory bandwidth.

2.3. Activation functions

In chapter 2.1, it has been established that neural networks can represent functions to arbitrary precision if the activation function Θ of layers is chosen to be non-linear. This does, of course, leave a lot of options open for the specific choice of it. Historically speaking, artificial neural networks were designed to resemble the neural structure of the brain. In a simplified model, the biological neuron receives signals from its dendrites, sums the signal strengths and, if it is above a certain threshold, the neuron fires along its output, the axon. To mimic this behaviour, a frequent choice for the activation function in artificial networks has been the **sigmoid** function:

$$\Theta(x) = \frac{1}{1 + e^{-x}}, \tag{2.3.1}$$

which is also plotted in figure 2.2. Depending on the sum of its inputs, this function can be zero (neuron does not fire) or one (neuron does fire), with a differentiable area in between, which is a requirement for the backpropagation algorithm(see equation 2.2.6). However, this choice of an activation function has become less popular, since it not only seems to be quite far from how actual neurons in the brain work (see e.g. [17] for a recent discussion of the computation in biological neurons), but is also suboptimal for the training process in artificial networks, since it possesses undesirable properties:

As can be seen in figure 2.2, the sigmoid function converges exponentially to zero or one for very small or very large inputs. This in turn means that the gradient of the function is very close to zero in these areas. Since the matrix which defines the back propagation step depends on this gradient (equation 2.2.9), the weight updates in a layer in which the inputs have a large absolute value can get very small, too. Furthermore, since the weight updates in the previous layers depend on the back propagation matrix of that layer as well, this can actually slow the training process in all preceding layers, a phenomenon known as vanishing gradients.

Nowadays, the sigmoid function is often used exclusively in the last layer of classification networks, together with the categorical cross entropy cost function. This function is explained in greater detail in section 2.4. The sigmoid function is often generalised for multiple neurons, as networks are frequently used to classify their inputs into multiple different categories. It is then called a **Softmax** function, whose output is given by:

$$\Theta_j(x_1, ..., x_K) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, \qquad \text{for } j = 1, \ldots, K. \tag{2.3.2}$$

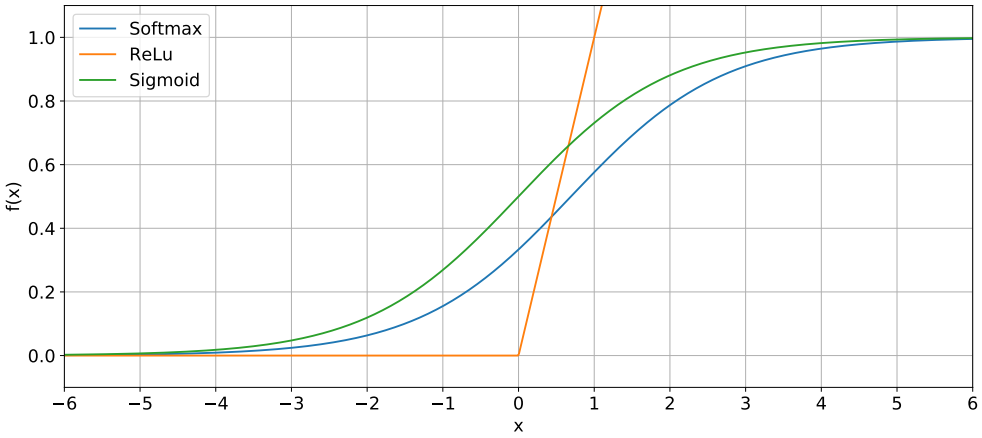


Figure 2.2.: Example of three different, commonly used activation functions: The sigmoid function, the rectified linear unit (ReLu) and the Softmax function. For the first two, x is the output of the neurons prior to applying the activation function. The Softmax function requires a multi-dimensional argument. Here, it was plotted for the case of a layer with three neurons in which two are assumed to output zero, that is, $f(\vec{x}) = f(x, 0, 0)$.

A plot for the specific case of $K = 3$ neurons, in which two neurons x_k are always zero, is plotted in figure 2.2. The curve of the Softmax function is identical to the sigmoid function, shifted in the positive x direction by the natural logarithm of the sum of the other neuron's outputs.

The activation function that has taken the place of the sigmoid function in the more recent past is the rectified linear unit (**ReLu**, see figure 2.2), which is given by:

$$\Theta(x) = \max(0, x). \tag{2.3.3}$$

Both the ReLu itself, as well as its derivative are computationally very inexpensive, which is a big upside compared to the sigmoid function. Also, this function does not saturate in the positive x regime. However, there is the risk of a ReLu becoming inactive, if the weights defining its input are set in such a way, that the ReLu will always output zero for all samples in the batch or even dataset. In this case, the neuron will permanently output zero, and no updates to its weights are applied anymore. Nevertheless, ReLus have shown to yield very good results over the sigmoid function (e.g. in [18], one of the first networks to use the ReLu activation in image classification), and they will be used for most of the neurons in this thesis as well.

2.4. Cost functions

In equation 2.2.3, the first step of the backpropagation has been defined, which depends on the cost function $C(\hat{F}_{\boldsymbol{\omega},\vec{b}})$. In general, this cost function specifies the goal of the network training, as it defines how close the prediction $\hat{F}_{\boldsymbol{\omega},\vec{b}}$ of the network is to the desired output F, given a specific input. Often times, networks will conduct one of the two following tasks:

- Classification: The input can be sorted in different categories, like neutrino events, which are to be classified as either travelling upwards or downwards through the detector.

- Regression: The input has a continuous distribution of a certain property, and the network should approximate this quantity. An example is the reconstruction based on the energy of a neutrino based on the signature its interaction left in the detector.

Depending on the task, different choices of loss functions are popular.

In the case of a regression, a common choice is the mean squared error loss function. For a given input, the network predicts the desired quantities to be $\hat{F}_{\boldsymbol{\omega},\vec{b}} \equiv \boldsymbol{y}^{pred} = (y_1^{pred}, ..., y_n^{pred})$, while the true values are $F \equiv \boldsymbol{y}^{true} = (y_1^{true}, ..., y_n^{true})$. The **mean squared error** then amounts to:

$$C(\boldsymbol{y}^{true}, \boldsymbol{y}^{pred}) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i^{true} - y_i^{pred} \right)^2 . \tag{2.4.1}$$

This function is always positive, and reaches its minimum when the prediction from the network is exactly equal to the true value. Another popular choice for regression tasks is the mean absolute error, which is defined in a similar fashion as above.

In the case of a classification task, a frequently used loss function is the categorical cross entropy: Given the prediction of the network $C(\hat{F}_{\boldsymbol{\omega},\vec{b}}) \equiv \vec{y}^{\,pred} = (y_1^{pred}, y_2^{pred}, ..., y_n^{pred})$, which sorts its input into n different categories, and the desired output $F \equiv \vec{y}^{\,true}$ with the same number of categories, the **categorical cross entropy** loss function is defined as:

$$C(\vec{y}^{\,true}, \vec{y}^{\,pred}) = -\sum_{i=1}^{n} y_i^{true} \ln y_i^{pred}. \tag{2.4.2}$$

This loss function is often used in conjunction with the softmax activation function defined in equation 2.3.2. The outputs from a layer with this activation function will always be between zero and one after passing through this activation function, and the sum over all outputs will be one. This makes it similar to a probability distribution. However, if the softmax function is in the saturated regime, in which the input to one neuron is much higher than those of the others, the gradient of the function becomes exponentially small (see figure 2.2 for very large or small x). In the mean squared error from equation 2.4.1, the activation function itself appears in the definition, since $\vec{y}^{\,pred}$ is the output of neurons which apply this function. If neurons get in the saturated regime, their weights and biases may only receive small updates, thereby slowing the training process. With the cross entropy loss, in contrast, the cost function can be written as:

$$C\left(\vec{y}^{\,true}, \vec{y}^{\,pred}\right) = -\sum_{i=1}^{n} y_i^{true} \left(x_i - \ln \sum_{k=1}^{K} e^{x_k} \right), \qquad \text{with } y_i^{pred} = \theta(x_i), \tag{2.4.3}$$

by inserting equation 2.3.2 in equation 2.4.2. Here, the outputs $x_1, ..., x_K$ of the neurons prior to the activation function appear as a linear factor in the cost function, since the logarithm of the the categorical cross entropy cancels out with the exponentiation in the softmax activation. This prevents the neurons from getting stuck in the saturated regime of the softmax activation during backpropagation.

2.5. Optimizers

After the backpropagation for a batch of training data is completed, that is, the derivative of the cost function $C(\hat{F}_{\boldsymbol{\omega},\vec{b}})$ with respect to all the weights and biases in the network has been calculated, the free parameters are updated in order to minimize the loss. In equation 2.2.2, a

simple way of updating was shown, in which a fraction of the gradient of the cost function is subtracted from every parameter, which is known as gradient descent. In fact, since the derivative of the cost function was not calculated for the entire training dataset, but rather approximated via a small batch of randomly chosen samples from the set, it is more accurately called stochastic gradient descent (SGD). However, there are also other strategies of updating the weights and biases, which for example try to accelerate the learning process by modifying equation 2.2.2 in various ways. The specific strategy after which the free parameters are updated is called optimizer, and a popular one, which was also chosen for most of the networks trained in this thesis, is the adaptive momentum estimation (adam).

The adam optimizer was introduce by Kingma and Ba in 2014 [19]. It attempts to speed up the convergence of network training by setting the step size η from equation 2.2.2 to not be constant for the entire gradient step, but instead makes it adapt to the current situation of convergence, and allows different free parameters to have different effective step sizes. For this, decaying averages over the past gradients and squared gradients are kept track of, which estimate the mean $\vec{m}_t$ and the uncentered variance $\vec{v}_t$ of the gradient updates at the current step t:

$$\vec{m}_t = \beta_1 \cdot \vec{m}_{t-1} + (1-\beta_1) \cdot \vec{\nabla}_\omega C(\hat{F}_{\boldsymbol{\omega},\vec{b}}) \tag{2.5.1}$$

$$\vec{v}_t = \beta_2 \cdot \vec{v}_{t-1} + (1-\beta_2) \cdot \left(\vec{\nabla}_\omega C(\hat{F}_{\boldsymbol{\omega},\vec{b}})\right)^{\circ 2}, \tag{2.5.2}$$

where the square in the last line is to be understood as the Hadamard product, in which the vectors are multiplied elementwise. Every element in the vectors $\vec{m}_t$ and $\vec{v}_t$ are the moving moment estimates of a specific free parameter in the network. Similar equations can be defined, if the parameter with respect to which the differentiation is computed is a bias b instead of a weight ω. The two constant hyperparameters β_1, β_2 introduced in the above equations define the rate at which past gradients and variances are exponentially decayed in the saved momentum estimates, and are usually set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$ as recommended in the original paper.

The above equations will result in estimates that are heavily biased towards zero in the early stages of training, since the original estimates $\vec{m}_0$ and $\vec{v}_0$ are initialized as vectors of zeros. This can be fixed by scaling the contribution of moment estimates in early gradient steps t towards the decaying average:

$$\vec{\widehat{m}}_t = \frac{\vec{m}_t}{1-\beta_1^t} \tag{2.5.3}$$

$$\vec{\widehat{v}}_t = \frac{\vec{v}_t}{1-\beta_2^t} \tag{2.5.4}$$

In total, the gradient update with the adam optimizer for a specific weight or bias in the network is given with the corresponding bias-corrected momentum estimates $\widehat{m}_t$ and $\widehat{v}_t$ by:

$$\omega \mapsto \omega - \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}. \tag{2.5.5}$$

Two new hyperparameters were introduced here: The step size α, which allows to scale the gradient step of all weights simultaneously as with the SGD optimizer, and ϵ, which was originally introduced for numerical stability and is therefore typically set to be very small. The authors of the paper, for example, recommend a value of $\epsilon = 10^{-8}$. However, it has turned out to sometimes be beneficial to the training process of the network to increase ϵ to up to 10^{-1}, and in the case of the autoencoders in this thesis, this increase even turned out to be of major importance.

The fraction $m_t/\sqrt{v_t}$ is compared to the signal-to-noise ratio of the gradient by the authors. If this ratio is small, that is, if there are large changes in the gradient from batch to batch, the adam optimizer automatically adapts to this by decreasing the effective learning rate, since this might indicate a close proximity of the weight to its optimum. In contrast, if the ratio is high, then the direction of the gradient seems to be clear, even within the limited statistics of single batches of samples. Thus, an increase in the effective learning rate could accelerate the learning process.

2.6. Types of layers

In the networks described above so far, all layers featured the same structure, in which all neurons in a specific layer are connected to all neurons in the following one. This type of layer is called a **dense** layer.

In the practical application of Deep Learning, other types of layers are used frequently as well, and are in fact often crucial for the performance of the network. In this section, several different types of layers besides the densely connected ones will be introduced, all of which were used for this work.

2.6.1. Convolutional layers

This layer is especially important for image recognition tasks, and also forms the basis of many networks used in this thesis. The basic idea behind convolutional layers is to make use of the translational invariance that frequently occurs in images. For example, if the energy of a neutrino is to be estimated by analysing the shower it produced in the detector, a shift of the whole event in the horizontal directions has little impact on the reconstruction task, as long as it is still fully contained in the detector.

To utilize the translation invariance, only neurons that are close to one another are connected via weights to the same neuron in the next layer. Since the features appear independent of their position in the image, the weights of nearby neurons should also be the same independent of their position in the image.

In convolutional layers, these two properties are realized by defining a kernel of a specific size, which is sled over the image. Every cell of the kernel stands for a specific weight, which locally connects a neuron inside this kernel to a neuron in the next layer. The process of applying a convolution to an image is best understood by considering a practical example, as shown in figure 2.3 for the case of a simple two dimensional image.

The shape of both the kernel and the pixels of the image, as well as the image as a whole is not necessarily required to be quadratic or rectangular to apply a convolutional layer to it. Nonetheless, it is the default implementation in common machine learning frameworks, since images and videos usually come in this format. By defining the kernel as a N-dimensional shape, the convolution can easily be generalized to inputs with a higher dimension than two, as is required for the analysis of ORCA data.

The mathematical description of a convolutional layer is very similar to that of a dense layer, since the above-mentioned sliding of the kernel across the image is describable by a weight matrix as well. Unlike the weight matrix of a fully connected layer, though, many entries are fixed to be either zero or identical to other weights in the matrix. To be exact, all input neurons that are outside of the kernel area for a specific output neuron, and should therefore not be connected to that output neuron, are given a weight of zero; neurons inside the kernel area are given the respective weight of the kernel. This is repeated for all output neurons, so that all entries in the weight matrix are specified to be either zero or one of the

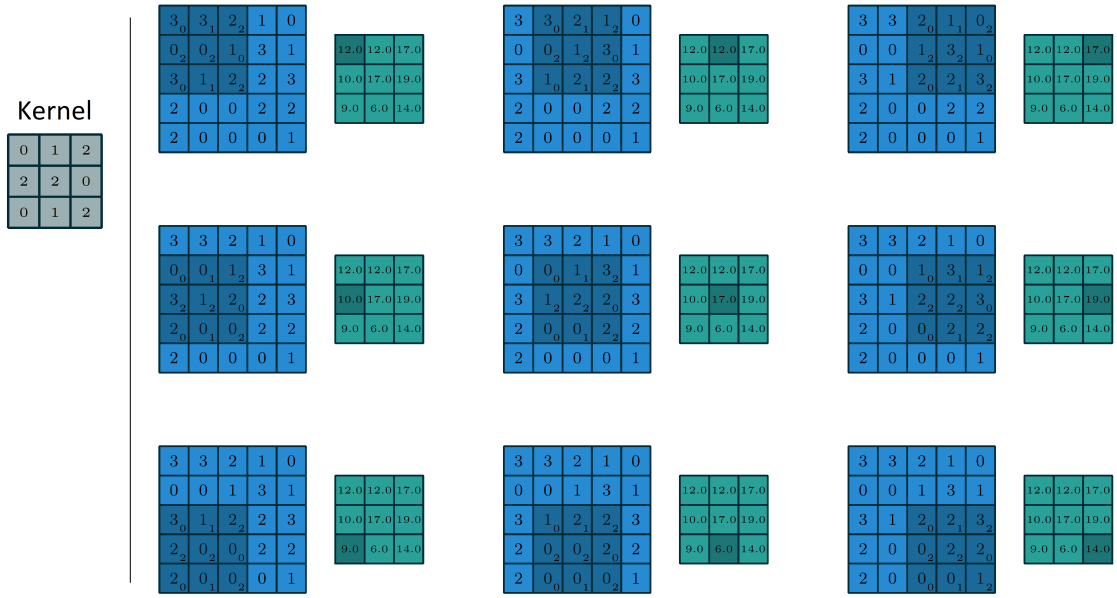


Figure 2.3.: Scheme of a two dimensional convolution with a three by three kernel over a five by five image, resulting in a three by three image. The original image before the convolution is represented by the blue grid, while the result of the convolution is the cyan grid. The big numbers inside the grids are the pixel values of the images, while the small numbers in the bottom right corner indicate the weights of the kernel. To calculate an output pixel value (cyan), all pixel values of the original image inside the highlighted kernel area are multiplied with the respective weight of the kernel, and then summed. In this specific example, no biases are added and no activation function is applied. The kernel is slid over the whole original image from edge to edge, while keepeing the kernel weights the same, independent of its current location [20].

weights defined in the kernel.

The large number of zeros in the weight matrix implies that a convolutional layer is generally much more sparsely connected than a dense layer, heavily reducing the number of free parameters in the network and therefore increasing the speed at which the network can be trained. For example, the convolution that is shown in figure 2.3 involves only nine different weights, in contrast to the $25 \cdot 9 = 225$ different weights a dense layer connecting these two images would contain. This is because the total number of weights of a single convolution depends only on the size of its kernel, and not on the size of the images it connects, as would be the case for a dense layer.

The weights in the convolutional layer define towards which patterns in the image the kernel is sensitive to. One might wonder in which way these kernels manage to identify certain features of the images they are applied to. This is, of course, problem-specific, but a somewhat general example of such kernels in conventional 2-D image processing are the Gabor filters. They are often used for feature extraction in images, with different filters being sensitive to patterns of different frequencies and orientations. Very much like these Gabor filters, the kernel defined in a convolutional layer is also thought of as a way of extracting a specific feature from an image. However, the network is not limited to resemble Gabor filters with its kernel, as the weights can be chosen freely. To allow the convolutional layer to become

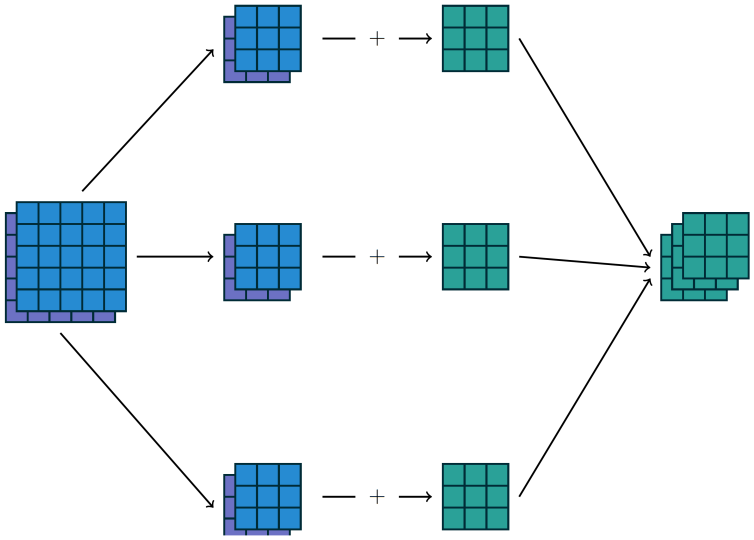


Figure 2.4.: Scheme of a two dimensional convolution over a five by five image with two channels, resulting in a three by three image with 3 channels. First, a different kernel is applied to both input channels for each of the three output channels, resulting in three pairs of convoluted images. Each of those pairs is depicted in its own row in the sketch. In total, six different kernels are used for this. Then, the pixels in the same spot in both channels of each pair are summed to give a single image at a time. Afterwards, the rows are concatenated.[20]

.

sensitive to more than just one feature, there will usually be multiple kernels defined per layer, resulting in multiple output images accordingly. These multiple output images from the same convolutional layer are called feature maps or channels, each of which being produced by convoluting a different kernel with the original image. Just like with the Gabor filters, some of those kernels might have been sensitive to high-frequency components (edges) in the image, while others might have extracted low frequencies. When the input to a convolutional layer already has multiple channels, for example because multiple convolutional layers have been stacked, kernels with different weights are applied to every single one of them, and the resulting images are summed.

More specifically, if the input to a convolutional layer has a total of N channels, and the output has a total of M channels, there will be $N \cdot M$ kernels applied. In doing so, for every of the M channels of the output, a different kernel is applied to each of the N input channels, and the N resulting convoluted images are summed pixel-wise to give a single image. All of these single images are different channels of the output. A practical example of this for a two dimensional convolution is shown in figure 2.4.

In general, the application of a convolutional layer reduces the dimension of the image, since the kernel is is moved within the edges of the image. If one of the original dimensions of the image is N, and the corresponding dimension of the kernel is k, the resulting dimension after the convolution will be $N - k + 1$. This reduction of the dimension if often undesired, as it limits the number of convolutional layers that can be conducted in sequence, and also reduces the amount of information that can be stored per channel due to the smaller image size. To circumvent this, convolutions are frequently preceded by a zero padding operation, which simply adds $k - 1$ zeros to the edges of the image, and therefore keeps the image dimension constant. Of course, different padding sizes can be used to adjust the resulting dimension of

the image as desired, which is especially useful for the design of autoencoders.

2.6.2. Deconvolutional layers

As mentioned in the previous chapter, a convolutional layer reduces the size of the image it got as an input by the dimension of its kernel + 1. For the reconstructing decoder part of an autoencoder, it is convenient to apply a layer that goes in the opposite direction, that is, a layer that has a similar connectivity pattern as a convolutional layer, but increases the size of the input image by its kernel size − 1. For this, the deconvolutional layer, which is also called transposed convolutional layer, or even sometimes falsely called inverted convolutional layer, is defined by transposing the weight matrix which defines the normal convolutional layer. This means that when a convolution with kernel size k produces a quadratic image of edge length $N - k + 1$ from an image of edge length N, the respective deconvolution produces an edge length of $N + k - 1$ from the same image. In this case, the deconvolutional operation is actually equivalent to a convolution on the original image with a zero pad of size $k - 1$ [20]. Despite not introducing new free parameters, the additional zeros will, however, increase the size of the image, and therefore also the number of both the neurons and the size of the weight matrix in the layer. Since the calculation time necessary to perform the matrix multiplications in the forward and backward pass through the net depends on the matrices' size, using an equivalent padded convolution instead of a deconvolution is less efficient.

2.6.3. Pooling and upsampling

Pooling layers are frequently used in networks involving convolutional layers, like residual networks [21], or the VGG network [14] on whose layout the nets in this thesis are based on. A pooling operation with a kernel size n reduces the size of its input, for instance by averaging over every n pixels, and then using this averaged value for a single pixel of the output. The number of pixels to average over is generally given as a kernel size, just like with convolutions and deconvolutions. For example, if the input to a pooling layer is a quadratic image with edge length $2 \cdot N$, and the kernel has a size of 2 by 2, the resulting image will have the edge length N, and therefore only contain one fourth of the pixels of the original image. Unlike the convolutional and deconvolutional layers, the pooling layer has no trainable weights, meaning that its operation cannot be influenced directly through training of the network. If the input to the pooling layer consists of multiple channels, which frequently occurs when using convolutional and pooling operations in the same network, the pooling is applied to every channel independently of the others, such that the pooled image ends up with the same amount of channels as its input.

Apart from averaging over the pixels in the kernel, a different kind of pooling is used commonly as well, in which the maximum value of all pixels inside the kernel volume is used in the output image. Both methods are shown to the left in the scheme in figure 2.5. The result is similar, and the preferred way of pooling is problem-specific.

The main motivation behind the use of pooling layers is the reduction of the number of neurons in the network, and thus a speed-up of the training process. Also, the pooling layers will often reduce the number of free parameters significantly, since the convolutional layers in image recognition networks are usually followed up by fully connected layers, whose number of associated weights increases with the number of neurons in the layer. The pooling operation can be defined for more than two dimensions by simply adding dimensions to the size of the kernel: For the three dimensional case, the kernel could for example be a cube, so that the sizes of all dimensions of the input are halved.

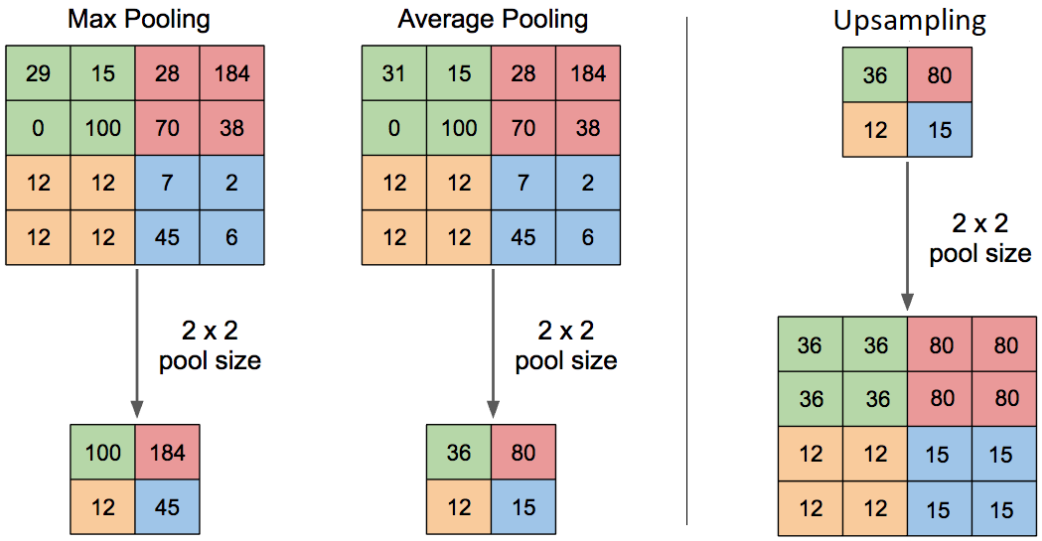


Figure 2.5.: Examples of two different pooling operations (max pooling and average pooling) and an upsampling operation. The numbers inside of each pixel of the images stands for their pixel value, and the colors represent which pixels get pooled into a single output pixel. For the two pooling operations, the input is a two dimensional quadratic image of edge length four, and the kernel size is two by two, so that the resulting image has an edge length of two. In the case of max pooling, the highest pixel value is chosen. In the case of average pooling, the pixel values inside the kernel area are averaged. For the upsamling operation, the value of the original pixel gets copied four times, so that it fills out the kernel volume, resulting in a doubling of the edge length of the original image. [22].

If the size of an input to a pooling layer is not a multiple of the corresponding size of the kernel, the input will usually be padded with zeros accordingly, so that information at the edges of the input does not have to be cut off. For the example of an image of a width of five pixels and a kernel with a width of two, a single pixel with value zero will be added to the image, so that the resulting pooled image would have a width of three pixels.

Similar to the deconvolutional layers, which form a counterpart of a convolution, a layer that has a comparable connectivity pattern to the pooling layer, but increases the size of its input in contrast to decreasing it, can be defined. This layer is called upsampling layer, and adds new pixels to the image in a specific way without raising the information contained. Just as with the pooling operation, this layer possesses no trainable weights. It is particularly useful in the construction of the decoder part of an autoencoder network, which attempts to reconstruct an image from a small set of variables.

In the upsampling operation used in this thesis, the size of the input image is increased by simply filling out the kernel volume with copies each original pixel, as shown to the right of figure 2.5. In the specific example shown there, the size of the kernel is quadratic. This is not a necessity, however, since both the pooling and upsampling operations can be realised with a higher kernel size in some of its dimensions, as it will be done in the autoencoders shown later.

2.6.4. Batch normalization

The concept of batch normalization has been introduced in 2015 by Sergey Ioffe and Christian Szegedy [23], and is nowadays frequently used in deep neural networks, since it can strongly

increase the speed of convergence during training, as well as make the network more robust against initialization problems and vanishing gradients. This layer is usually put right before the activation function is applied by the neurons. In the case of a dense layer, this would mean that the inputs to the neurons have already been multiplied by the weights and the biases have been added, before the Batch Normalization is conducted. Afterwards, the activation function is applied as usual.

The goal of batch normalization is to keep the change of the distribution of the inputs to the neurons during the training process as small as possible. In the standard network set-ups using the layers described above, however, the distribution is likely to change. This is because the training will adjust the weights of each layer, and therefore also its output, which in turn represents the input to the following layer. To reduce this internal covariate shift, the batch normalization attempts to whiten the output of every neuron in the network, that is, linearly transform the output to have a mean of zero and a standard deviation of one. For this, the calculation of the mean and variance of the output of every neuron would have to be calculated for the whole training set after every gradient update step. To reduce the complexity of this computation, the mean μ and the standard deviation σ is instead calculated only for the output $B = \{x_1, x_2, ..., x_m\}$ of the batch that is used for the current gradient step:

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{2.6.1}$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{2.6.2}$$

As a reminder, a batch is a small number of samples from the dataset, which is used to approximate the true value of the cost function (see section 2.2). Then, the output of the neurons is set to have a mean of zero and a variance of one over the current batch:

$$y_i = \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \tag{2.6.3}$$

The parameters β and γ are trainable, but are updated depending on all samples in the batch. The parameter ϵ is a constant introduced for numerical stability, since it limits the value of the denominator and avoids division by zero. For that reason, it is often chosen to be very small (e.g. 10^{-8}).

Note that the parameters β and γ could be learned by the network to exactly cancel out the shift and scale done before. This is a deliberate feature, as it allows the neurons to still represent the same range of outputs as before, if that is helpful to increase the performance, but still gives the network an easy way to normalize the outputs. Ultimately, the Batch Normalization operation is somewhat similar to those of convolutional or fully connected layers, in that they all get their input from neurons in the layer before, apply a mathematical function with trainable parameters to them, and feed the result into subsequent layers. Hence, the Batch Normalization is typically referred to as its own layer, and applying the Normalization to a fully connected layer can be seen as stacking three layers on top of each other: First comes the dense layer with a linear activation function, followed by the Batch Normalization layer and then an activation layer, which merely applies the intended activation function.

After the training of the network is finished, it should be able to reliably predict on input data, even if the input is not a full batch, but rather single samples of data. With the above approach, this is not possible, since batch statistics are necessary for the calculation of the batch mean and variance. To address this issue, the mean μ_B and the standard deviation σ_B are in practice not completely recalculated for every batch, but instead a decaying average

over the last batches of training samples is used: Let μ be the mean output of the current batch of training data. The mean used for the batch normalization for the current batch is then given with the decay factor δ by:

$$\mu_B \leftarrow \delta \cdot \mu + (1 - \delta) \cdot \mu_B, \qquad 0 < \delta < 1, \tag{2.6.4}$$

and similar for the variance σ_B. These decaying averages are updated during every batch gradient step in the training, and, after the training is complete, get saved to allow for predictions on single data samples.

A special case is the use of batch normalizations together with convolutional layers, since convolutions are based on the differences between the outputs of neurons in different regions of the input images. Therefore, normalizing and zero centring all neurons independently of one another would have a very strong negative impact on the performance of convolutional layers. Instead, the different channels of the input images are batch normalized as a whole, meaning that there is only one set of parameters β, γ per channel, and the same linear transformation is applied to all of them.

2.7. Network training in practice

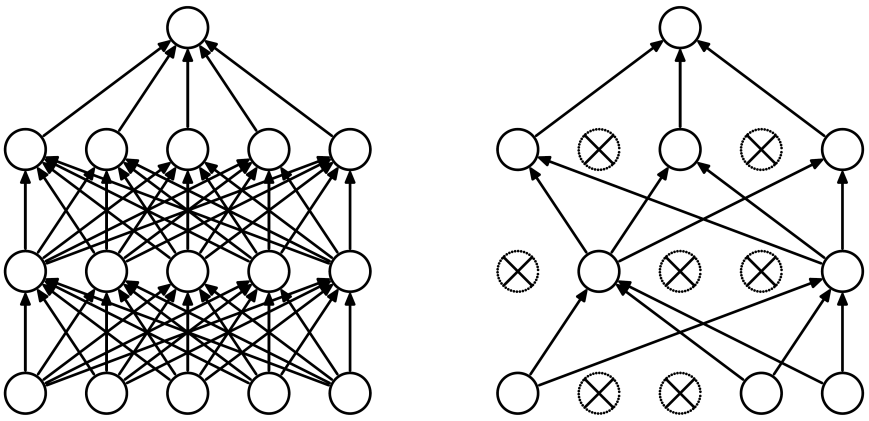
This chapter will address several issues that arise when dealing with networks and their training in practice, like overfitting, weight initialization and data preprocessing.

Train and test datasets: A common problem when training networks in practice is overfitting on the data: The networks used in Deep Learning often contain millions of free parameters, which allow them to adapt to the specific properties of the samples it is presented with during training, without learning general features of the data. To monitor this behaviour, the dataset is usually split in a training set and a test set. The training is exclusively conducted on the samples in the training set. After every iteration on the training data, the performance of the network is evaluated on the test set as well, whose samples have never been seen during training. If the performance of the network is higher on the training set than on the test set, it overfitted on the training samples, which might be undesirable.

Dropout: While different ways to counteract this tendency towards overfitting have been developed, a particularly easy to implement method is the utilization of dropout [24]. For this, a fraction of the neurons in one or more layers is randomly disabled during training, meaning that their output is set to zero. This is shown for the case of a small network in figure 2.6.

The training of a network to which dropout is applied is similar to training exponentially many different networks at the same time and combining their results, since a network with dropped-out neurons can be seen as a different instance of the original model with less neurons [24]. The weights are shared between all the different thinned out networks, so that training remains efficient. When the network training is completed, and its performance is to be evaluated on the test dataset, no dropout is applied anymore to achieve the best performance possible. For this, it is important for every neuron to receive on average the same input it is expecting due to its training. If all neurons are simply reactivated during testing, this is not the case, since the mean input will be increased by a factor of p, where p is the chance of its input neurons to have been dropped out. To fix this issue, the outputs of all neurons are scaled by a constant factor of $1 + p$ during training, so that the expected input to the neurons is identical in both the training and the testing phases.

Weight initialization: While the procedure of updating the free parameters in the network is defined by the backpropagation algorithm, there is still the question of what starting values to pick for them. If one were to, for example, initialize all the weights to be zero before the training, this would imply that all neurons emit the same output value zero, and therefore



(a) Standard Neural Net (b) After applying dropout.

Figure 2.6.: Examplary scheme of dropout being applied to network with two hidden dense layers. The arrows stand for the weights connectig the neurons, and crossed out neurons symbolize the ones that have been randomly deactivated due to dropout (Taken from [24]).

experience the same gradient update as well. Instead, the parameters are usually initialized according to some distribution with a mean of zero and a reasonably small variance, like a unit Gaussian centred on zero. This can, however, lead to the output of the neurons having a high variance: Let $\vec{x}$ be the inputs to a neuron, $\vec{y}$ its outputs, and $\boldsymbol{\omega}$ the randomly initialized weights. Similar to equation 2.1.3, the outputs are given as:

$$\vec{y}(\vec{x}) = \Theta\left(\boldsymbol{\omega} \cdot \vec{x} + \vec{b}\right). \tag{2.7.1}$$

Then, the variance of a specific output $\vec{y}_l$ increases linearly with the number of inputs to the neuron n, assuming both the weights $\boldsymbol{\omega}$ and the inputs $\vec{x}$ have an expectation value of zero:

$$\mathrm{Var}(\vec{y}_l) = n \cdot \mathrm{Var}(\boldsymbol{\omega}_l \vec{x}_l). \tag{2.7.2}$$

To counteract this, a slight modification of the initialization can be applied, according to He et al. [21]. They instead initialize all weights according to a zero centred Gaussian distribution with a variance of $2/n$. The factor 2 in this equation stems from the fact that the distribution of the inputs x_l do not have a mean of zero if ReLu activation functions like described in chapter 2.4 are used, since all outputs that would be negative are instead set to zero. For that reason, the He initialization is used as a standard for most layers in this thesis.

Zero centering: This means calculating the mean input bin-wise over the whole training dataset, and then subtracting it from each input sample bin accordingly, both during testing and training. This has to be done only once before the training, and is a form of preprocessing the data. Zero centering the data is generally considered to be helpful (see e.g. [25]), since it prevents the inputs to neurons to be exclusively positive or negative.

3. Using autoencoders on ORCA data

3.1. The concept of autoencoders

When utilizing Deep Learning for data analysis, a frequent approach is to train a fully supervised network. This means that for every sample in the dataset, the network is provided not only with the sample itself, but also with the correct answer it is supposed to predict during the training. The parameters in the network are then adjusted like it was described in section 2.2 to make future predictions match these labels as accurately as possible. Naturally, this can pose two problems:

- The need for labelled data: A large set of data has to be available, for which the correct predictions (labels) are reliably known. In the case of ORCA, these labels are generated as a part of the simulations which provide the data the network is developed on.
- The accurateness of the labelled data: When using simulations, one has to make sure that they are in close resemblance to the real data, e.g. that all substantial features of the real data are present in the simulations, and vice versa.

The latter is especially important, since networks often contain millions of free parameters, which allow them to become sensitive to even small details of the data. If, for example, the simulations were to include some feature which is helpful for predicting the energy of the particle, the network might heavily rely on that to conduct its prediction. If the feature is not present in the real detector data, the overall performance of the network might drop significantly when it is employed in practice. To make things worse, this drop-off in accuracy might not even be noticeable, since it is difficult to properly assess the performance on measured data due to the lack of labels. Even if a discrepancy between the simulations and the measured data is known, it might still not be realistic to adjust the simulations accordingly if the cause for the disparity is unknown.

A possible way to circumvent the issue of flawed simulations is to simply not train on them, and to use actual measured data instead. Since the labels on measured data are generally not known, this approach is called unsupervised learning. In this thesis, the specific network type to implement this is the deep convolutional autoencoder, which is a popular architecture for unsupervised learning (see e.g. [26, 27, 28, 29, 30] for applications). It is smiliar in parts to the supervised VGG image recognition network [14], which has proven to be successful in the ImageNet classification challenge. This shows its potential in extracting features from data with spatial coherence, like it is the case not only for the photographs of ImageNet, but also for the ORCA detector data.

The first part of the VGG net consists of several blocks of convolutional layers, with pooling layers in between. They reduce the dimensions of the image step by step, while increasing the number of channels at the same time. This part is generally considered to be extracting the features from the input images which are needed for predicting the labels. In the second part, a few small dense layers combine these features to produce the desired output. The VGG network has to be trained fully supervised, which is only viable if labels exist in the first place.

Autoencoders are made up of two parts as well, and the first half, the encoder, is actually similar to that of the VGG type: Blocks of convolutional layers and pooling layers in alternating

succession, which extract features from the original image. The second half, the decoder, is like a mirrored version of the first half. It is built from transposed convolutions and upsampling layers, which gradually increase the dimensions of the image again, while reducing the number of channels. Ultimately, the target of the autoencoder is to reconstruct the original image. The crucial point of autoencoders is a bottleneck in the model structure: If the number of neurons in the innermost layer of the network between the encoder and the decoder is smaller than that in the input layer, the network has the incentive to find a good representation of the input in a lower dimensional space, which conserves as much information about the original image as possible. For example, this encoded, compressed version of the image might contain properties like the energy of the particle, the direction it is travelling in, or whether it produced a track or a shower, because this is information that is important for reconstructing the image. Unlike the first half of the VGG net, which extracts only features relevant for correctly predicting a given label, the encoder tries to extract as many descriptive features as possible, as long as they are helpful for reconstructing the original image. Therefore, no labels are needed to train the autoencoder.

However, the goal of the data analysis is to give predictions on properties of the particle, like its energy or direction, and not an abstract compressed image, in which this info is encoded in. For this reason, the autoencoder training is followed up by a second, supervised stage of training. For this, the encoder part of an autoencoder is taken, and all previously free parameters in it are fixed. This means that the encoder is essentially preprocessing the images into an encoded state, by extracting the features it has learned during the training on measured data. Then, a small dense network is appended to the encoder, which acquires the specific property one is interested in from the encoded representation. This second stage of training has to be performed in a supervised fashion on labelled data. However, since by far the largest part of the network was trained on measured data, it might be much more resistant to the effects of imperfect simulations. The procedure of these two phases of training is shown in figure 3.1.

It is the aim of this thesis to find out how well this unsupervised approach will work on ORCA data in practice, and to which degree the robustness of networks can be increased with it.

3.2. Properties of the dataset

The simulated data, on which the networks are trained on, come in the form of muon- and electron-neutrino charged-current interactions on nucleons which have passed the trigger conditions. Every photomultiplier of the detector has recorded the specific times at which it measured photons, the so called hits. Since the DOMs of the detector are composed of 31 photomultipliers each, the resulting data is five dimensional: The three spatial coordinates x, y, z of the DOM, the ID of the photomultiplier, and the points in time at which the hits were recorded. All the five-dimensional data points which are measured in a short timespan around the moment the interaction happened are called an event snapshot. They can be seen as a multi-dimensional image, similar to the two-dimensional photos used for many conventional image recognition networks.

In order to feed the data into the network, the temporal and positional coordinates have to be binned, so that standard convolutional layers can be applied to them. For the positional coordinates, the detector is overlaid by a three dimensional $11 \times 13 \times 18$ grid, and the exact position of each DOM inside these bins is disregarded. This is straight forward for the vertical z coordinate due to the way the DOMs are spaced in that direction. The horizontal $x - y$ case is more difficult, as its irregular spacing results in some bins containing more than one

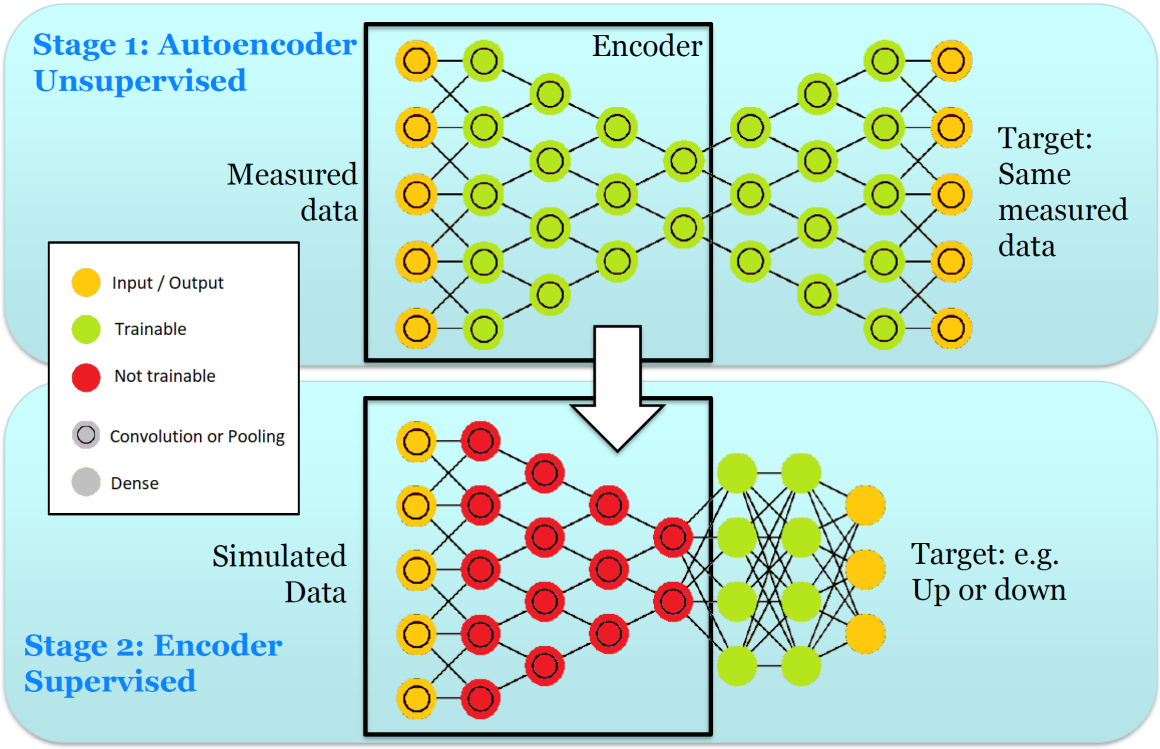


Figure 3.1.: The training principle employing autoencoder networks. The circles represent neurons in the network, but the actual number of neurons in the models is much higher than shown here. In particular, the dense layers of the second stage contain much fewer neurons than the convolutional layers. The first phase (upper half) is unsupervised, meaning that it can be trained on measured data, as no labels are needed. The second phase (lower half) has to be done on simulations. The encoder from the first part of training is reused in the second part, after fixing all its weights and biases (modified from [31]).

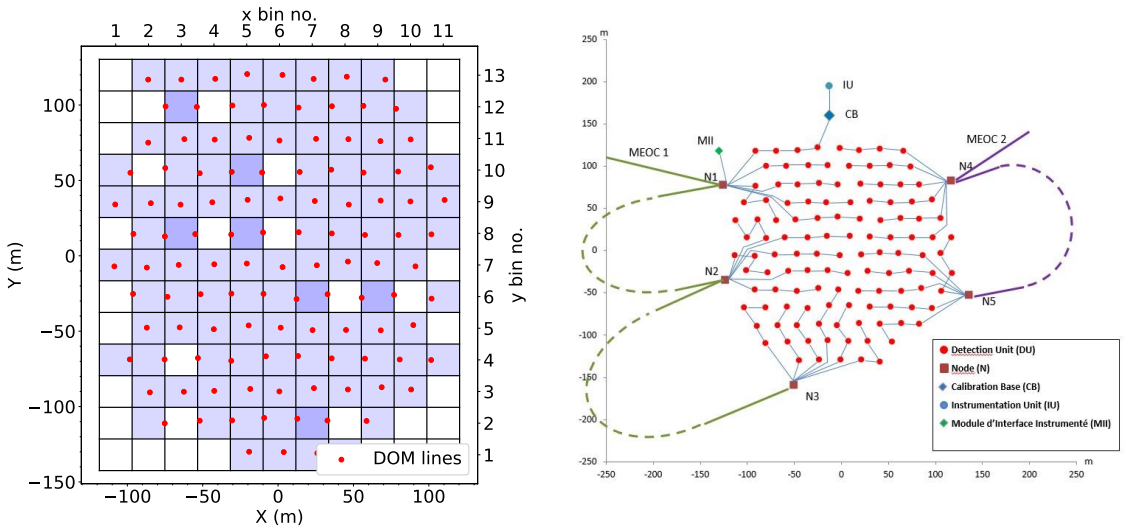


Figure 3.2.: Left: Spatial binning of the detector data as part of the preprocessing, shown as a slice in the horizontal direction. The red dots mark the positions of the DOMs in the simulations according to the real-world detector layout shown on the right (created by Patrick Lamare), and the grid represents the binning. Some bins do not enclose any DOMs, and therefore never contain any hits either.

DOM (figure 3.2).

For the time binning, all hits outside of 30 percent of the timespan of each event snapshot were cut off, with the interval being centred on the middle of the timespan. The hits in the remaining interval are binned into 50 temporal bins, each with a duration of about 19 nanoseconds. This has to be considered slightly suboptimal, since the total duration of each snapshot varies from event to event with a standard deviation of about one nano second per bin. This can be circumvented by cutting out a time interval of a fixed length instead, centred on the mean of all hit times.

At the end of the research period for this thesis, an improved binning was tested, in which no bin contains more than one DOM and the fixed timespan is implemented. This improved the performance of the network slightly (88.16% before to 88.33% accuracy with the new binning in the up-down classification introduced in section 3.4.2).

It is likely that the network performance could be improved a bit by reducing the timespan of the bins to less than 19 nanoseconds. However, this would result in more bins and would therefore lengthen the process of training, so a compromise was made. The information about which photomultiplier measured each hit is processed by assigning each photomultiplier a specific ID – the channel ID – depending on where in the DOM it is located at. Photomultipliers in different DOMs have the same ID if they are in the same spot in the DOM.

The result from the above binning process is data in a five dimensional $11 \times 13 \times 18 \times 50 \times 31$ format, that contains the number of hits in each x-, y-, z-, time-, and channel id-bin. This thesis uses KERAS [32] with the TENSORFLOW [33] backend to construct and train networks, in which convolutional layers are currently only defined for data with up to three dimensions. For that reason, the dimension of the binned data is reduced by summing over some of the dimensions. For example, the xzt-data, which represents the input to many of the networks

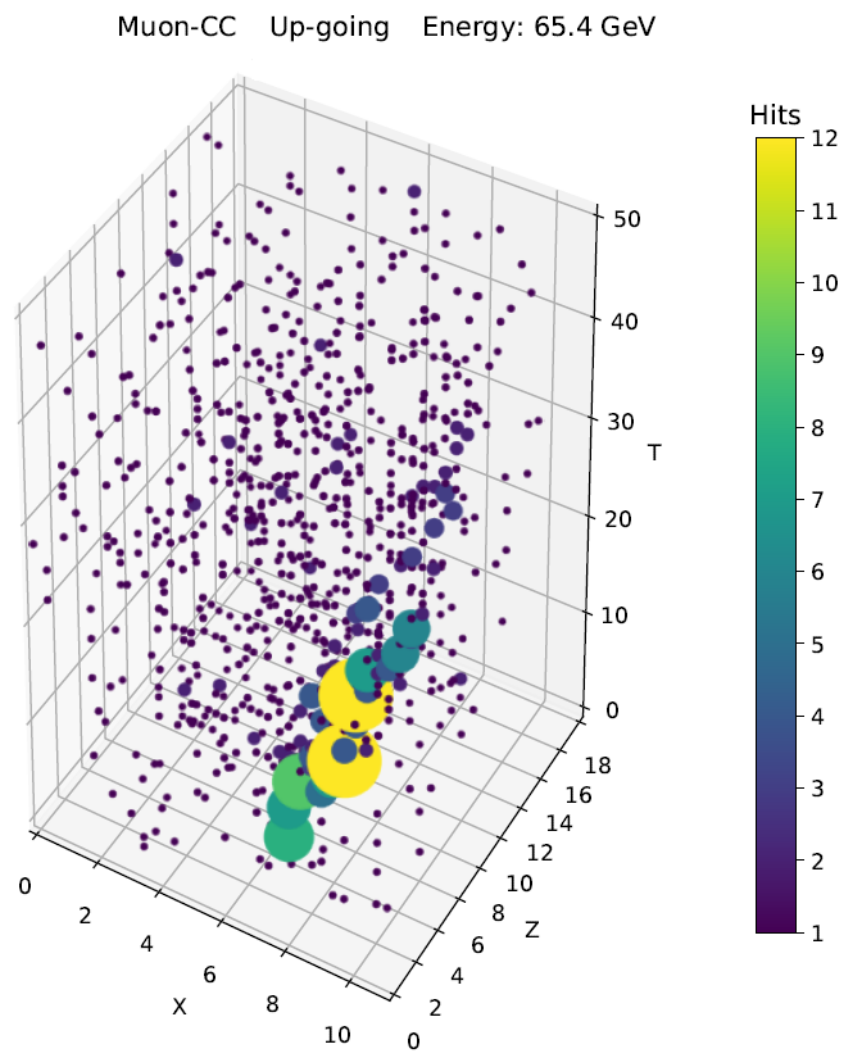
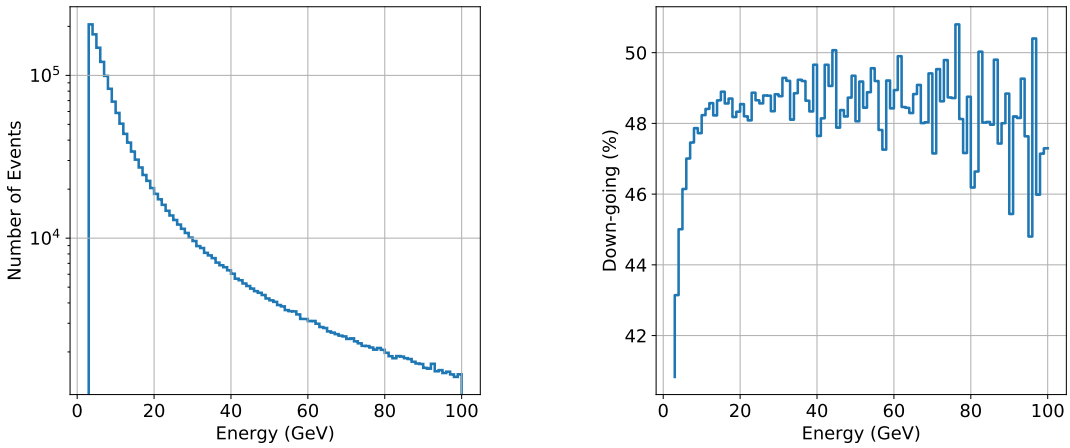


Figure 3.3.: A single sample, containing noise and a highly energetic event binned to three dimensions. Both the size and the color of the circles represent the number of hits measured in each bin.

in the following, is obtained by adding up all the counts of the y- and channel id-axes of the data. The result is a $11 \times 18 \times 50$ histogram, as shown in figure 3.3 for the example of a highly energetic event.

The xzt training dataset used for most networks contains simulated events between 3 and 100 GeV, with a roughly even split between up and down going events and a flux following a E^{-2} power-law over the energy (figure 3.4). This curve is based on the flux of the real world spectrum of atmospheric neutrinos, yet with a softer slope, so that highly energetic neutrinos occur in a higher frequency during the model training. To which degree this may increase the probability for incorrect reconstructions of the energy on measured data will have to be tested in the future. In total, the training set contains about 1.6 million events, and the test set with the same properties as above has 400.000 events. No precuts other than the trigger conditions were applied to the datasets.



(a) Number of events, binned over their energy.

(b) Fraction of down-going events, binned over their energy.

Figure 3.4.: Statistics of the simulated xzt training dataset used for most models in this thesis.

3.3. Network design

The design of the autoencoder is based on the design of deep convolutional networks like the VGG network [14], which has been empirically found to be successful in image recognition tasks. The encoder contains multiple convolutional blocks, with pooling layers in-between, while the decoder has a mirrored structure with transposed convolutional blocks and upsampling layers. Each convolutional or transposed convolutional block is build of a padding layer, followed by a convolution, a batch normalization and an activation function, as shown in table 3.1. Unless otherwise noted, the default values given in the table are used for all layers.
In table 3.2 the model of an autoencoder with 1920 neurons in the bottleneck is summarized. The architecture of this model is exemplary for many of the autoencoders in this thesis. The input image has a size of $11 \times 18 \times 50 = 9900$ bins, so it is compressed to a minimum of 19% of its original size during a forward pass through the net. The size of the bottleneck severely affects the performance of the autoencoder and is investigated further in chapter 5.1. Pooling and upsampling layers are used to reduce or increase the size of the image, as described in section 2.6.3. The paddings of the convolutional blocks in the encoder part were chosen in such a way that the edge lengths of the images are even before a pooling layer is applied. This way, no separate padding has to be done before the pooling operations.

In the decoder, the size of the images ought to resemble those of the encoder part for the sake of symmetry between the two halves of the autoencoder. This can in general not be achieved only with transposed convolutions with the given layers, since they can only be used to increase the size of the images. If the dimension has to be reduced, a convolutional layer is used instead. It is functionally identical to the transposed convolution, albeit less computationally efficient.

It might seem odd that in the decoder part, the last transposed convolutional layer in the middlemost section reduces the number of channels from 64 to 32, while in the encoder part, the channels were the same for all layers in the section. However, this stems only from the fact that the decoder layers are listed exclusively with their output and not their input: If both are taken into regard, the symmetry becomes obvious:

Table 3.1.: Structure of a convolutional building block. The weights of the convolution are He Normal initialized (section 2.7). The transposed convolutional building block is identical to this one, except that the convolutional layer is replaced by a transposed convolutional layer (section 2.6.2).

Convolutional Block:

Layer type	Properties
Padding	Appends zeros to the image to increase the output size of the building block (section 2.6.1).
Convolution	Cubic kernel with an edge length of three bins (section 2.6.1).
Batch Normalization	Improves the convergence of the network (section 2.6.4).
Activation	Rectified linear unit per default (section 2.4). If another function is used, it is given in parentheses.

	Input dimension				Output dimension		
Last encoder section	$5 \times 8 \times 12$	$\times$	32	$\longrightarrow$	$2 \times 3 \times 5$	$\times$	64
First decoder section	$2 \times 3 \times 5$	$\times$	64	$\longrightarrow$	$5 \times 8 \times 12$	$\times$	32

An output-wise symmetric alternative design is tested in section 4.2.

The last layer in the network is a convolution with a $1 \times 1 \times 1$ kernel, and a linear activation function. This is essentially a weighted summation of all filters from the previous layer, and ensures that the output of the autoencoder can be negative as well, which will be the case as the input data is zero centred. This also means that the bin counts will in general not be whole numbers.

After the unsupervised training of the autoencoder is completed, the model is prepared for the supervised phase as described in section 3.1: The encoder part is taken from the autoencoder, its weights are frozen, and some dense layers are added. Similar to the convolutional blocks, the dense layers are also ordered in a block-like structure. Since dense layers are prone to overfitting, they are preceded by a dropout layer, and, similar to the convolutional block, followed by a batch normalization and the activation function. The structure of the building block is shown in table 3.3.

The complete setup of the network for the supervised training in phase two is shown in table 3.4. The first part of this model with a red background is identical to the encoder part of the autoencoder in structure, and also has the same weights and biases. Furthermore, the previously free parameters in these layers are now fixed. During backpropagation, only the parameters of the dense layers at the bottom get updated. The last layer features multiple neurons and a softmax activation function, as this encoder+dense version will be used for a classification task. If a regression is to be performed instead, like a reconstruction of neutrino energies, a single output neuron with a linear activation is used.

3.4. Autoencoder training procedure

In this section, the autoencoder training procedure described in chapter 3.1 is tested on simulated data and arising problems are discussed. For all trainings in this thesis, binned xzt

Table 3.2.: Network structure of the *model-1920* autoencoder. The layers inside each convolutional block are defined in table 3.1. The first half of the network (top three sections) is the encoder, and the second half (bottom three sections) is the decoder.

Building block	Output dimension
Input	11x18x50 x 1
Convolutional block	11x18x50 x 32
Convolutional block	11x18x50 x 32
Average Pooling (1,1,2)	11x18x25 x 32
Convolutional block	11x18x25 x 32
Convolutional block	10x16x24 x 32
Average Pooling (2,2,2)	5x8x12 x 32
Convolutional block	5x8x12 x 64
Convolutional block	5x8x12 x 64
Convolutional block	4x6x10 x 64
Average Pooling (2,2,2)	2x3x5 x 64 (1920)
Upsampling (2,2,2)	4x6x10 x 64
Convolutional block (T)	6x8x12 x 64
Convolutional block	5x8x12 x 64
Convolutional block (T)	5x8x12 x 32
Upsampling (2,2,2)	10x16x24 x 32
Convolutional block (T)	12x18x26 x 32
Convolutional block	11x18x25 x 32
Upsampling (1,1,2)	11x18x50 x 32
Convolutional block (T)	11x18x50 x 32
Convolutional block (T)	11x18x50 x 32
Convolution (kernel size 1, linear)	11x18x50 x 1

Trainable parameters: 748609

Table 3.3.: Structure of a dense building block. The weights of the dense layer are He Normal initialized (section 2.7).

Dense Block:

Layer type	Properties
Dropout	Reduces overfitting of the network (section 2.7).
Dense	Every neuron is connected to all neurons in the previous layer (section 2.1).
Batch Normalization	Improves the convergence of the network (section 2.6.4).
Activation	Rectified linear Unit per default (section 2.4). If another function is used, it is given in parentheses.

Table 3.4.: Architecture of the *model-1920* encoder+dense network. Cells with a red background indicate that the corresponding layer is not trainable (its weights and biases do not get updated during training).

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	32
Convolutional block	11x18x50	x	32
Average Pooling (1,1,2)	11x18x25	x	32
Convolutional block	11x18x25	x	32
Convolutional block	10x16x24	x	32
Average Pooling (2,2,2)	5x8x12	x	32
Convolutional block	5x8x12	x	64
Convolutional block	5x8x12	x	64
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64
Flatten	1920		
Dense block	256		
Dense block	16		
Dense (Softmax)	2		

Trainable parameters: 499762

data is used in batches of 32 samples each. The architecture of the *model-1920* introduced in the previous section is used. Both phases of the training are done on the same simulated dataset, so this can be seen as the "worst case" scenario for the autoencoder in which the simulated data would be identical to the measured data. In this case, the autoencoder training procedure would not be necessary in practice, since its advantage is a potentially higher robustness against differences between simulations and measured data.

The performance of the encoder+dense network is then compared to that of a network with the exact same architecture, but which was trained in a completely supervised fashion. The effects of differences between the simulated and measured datasets on these networks are tested separately in chapter 6.

3.4.1. Phase 1: Unsupervised autoencoder training

The loss function used during the autoencoder training is the mean squared error. This means that the overall loss, which is supposed to be minimized, is computed by taking the difference between counts in every bin of the original and the reconstructed image, squaring each difference, and then averaging over all of them. The loss is zero only if the reconstruction and the original are exactly identical. This loss function might not be an ideal choice, since better reconstructions of bins containing nothing but background noise will also reduce the loss. Another important decision to make for the training is the selection of the optimizer and its settings. Three different setups have been tested:

- Adam optimizer with $\epsilon = 10^{-1}$: Initial learning rate 0.001 with an exponential decay of 5% per epoch up until epoch 40, then constant at 0.001, 0.01 after epoch 80, and halved for the last 10 epochs.
- Adam optimizer with $\epsilon = 10^{-8}$: Initial learning rate 0.001 with an exponential decay of 5% per epoch.
- Stochastic gradient descent: Initial learning rate 0.1 with an exponential decay of 5% per epoch.

The choice of the learning rate and its schedule is a problem-specific hyperparameter. In general, it should be set larger as long as the loss is far from its minimum, and smaller once it gets close. One has to be careful though, as too large of a learning rate might stop the network from optimizing altogether, and a too small one might unnecessarily slow the convergence. Since a single epoch of autoencoder training takes multiple hours, an optimization of the learning rate schedule is waived and it is instead adjusted several times during training with respect to the current progress, as described in the following paragraphs.

The history of the training is shown in figure 3.5. For all models, it is made sure that neither an increase nor a decrease in the learning rate would improve the performance of the fully trained model further. Even though no dropout is used, no overfitting is observable for the converged networks, even with very high learning rates. This suggests that autoencoders might be resistant to overfitting per design, possibly due to the presence of a bottleneck in the model structure. On the other hand, the autoencoder trained with $\epsilon = 10^{-8}$ does experience slight overfitting in the early parts of the training, which disappears after about 25 epochs.

The training progress of the $\epsilon = 10^{-1}$ autoencoder shows unusual properties: It is started with the default learning rate for adam of 0.001. As was found out during the training, it can be increased by a factor of ten to speed up the convergence, without destabilizing the training process. Figure 3.6 shows the reconstructions of this autoencoder at different points during the training. With this insight, the model with the SGD optimizer is trained with a

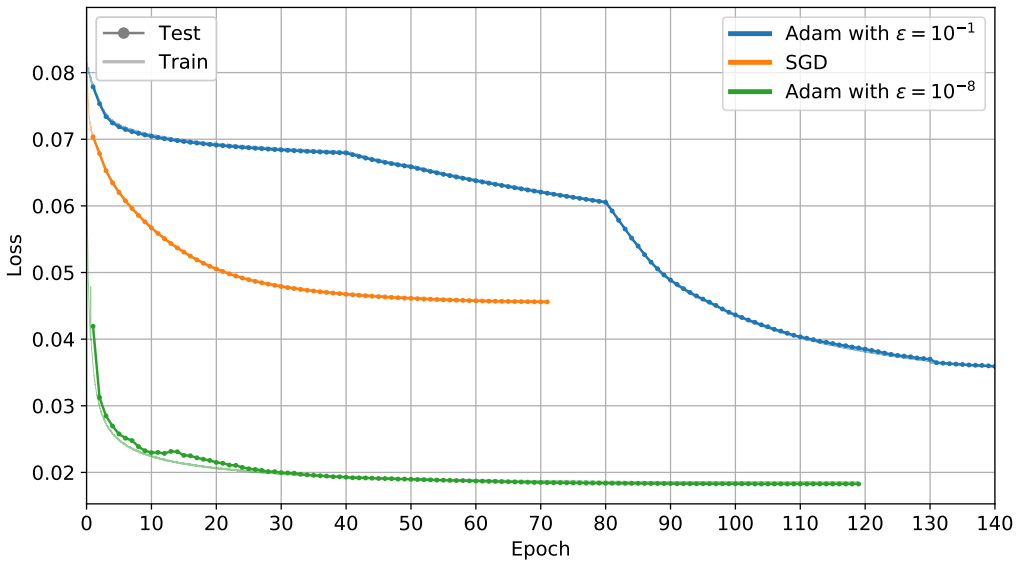


Figure 3.5.: Loss (mean squared error) of three identical *model-1920* autoencoders, each trained with different optimizer settings. An epoch is an iteration over the full dataset. The faint solid line shows the performance during the training: Every 500 batches, the loss of the network on them is calculated. After an epoch is completed, the network is evaluated on the whole of the test dataset, which results in a high-statistics data point for the loss (dotted line). For the blue and orange settings, the faint line is hidden behind the dotted line completely.

learning rate of 0.1 right from the start, which is a factor 100 higher than the default value. In the fully converged state, this model has a distinctively worse performance. The model with the $\epsilon = 10^{-8}$ optimizer shows drastically faster convergence than the other two, while also reaching much lower losses with a smaller learning rate, enabling it to reconstruct its input almost perfectly (figure 3.7).

To understand the importance of the epsilon parameter, one has to take a look at the definition of the effective learning rate of the adam optimizer again (equation 2.5.5): ϵ, originally introduced to avoid divisions by zero, coincidentally also limits the influence of the denominator by establishing a lower limit of $\sqrt{\epsilon}$ for it. If the moving average of the raw second moment estimate is smaller than 10^{-1} for a certain weight in the network, training with $\epsilon = 10^{-8}$ allows for a much higher learning rate for it, up to seven orders of magnitude. Therefore, a lower ϵ can be seen as an ease of the restriction on the effective learning rate of every weight. It was not possible to emulate the behaviour with $\epsilon = 10^{-8}$ by a simple increase in the overall learning rate of an adam $\epsilon = 10^{-1}$ or SGD optimizer. Repeating the training with a different random initialization of the free parameters in the networks did not change this behaviour.

3.4.2. Phase 2: Supervised encoder + dense training

The task to be solved by the networks in this section is to classify events as either up-going or down-going, depending on the direction that the original neutrino travelled in relative to the detector. In practice, this could be used to reject atmospheric neutrinos or muons which are produced in the atmosphere above the detector. Furthermore, since this is one of the easier tasks of ORCA data analysis, it is expected to converge quickly, shortening the time needed to assess the performance of newly designed models.

Following the autoencoder training procedure, the encoder part is extracted from one of the trained autoencoders from section 3.4.1, its weights are frozen and three dense layers are added. This results in the encoder+dense version of the *model-1920*, as was shown in table 3.4. The loss is chosen to be the categorical cross entropy introduced in equation 2.4.2, and the output from the last two neurons of the network is supposed to be either $[0, 1]$ or $[1, 0]$, depending on the direction of the particle. The performance of the network is expressed in terms of its accuracy, i.e. the fraction of all samples in the dataset which were correctly classified as up- or down-going.

To be able to rate the accuracy of the encoder+dense network, another network is trained according to the supervised approach for comparison. For this, the same layer structure involving convolutions, poolings, and dense layers as in the encoder+dense network is used, except that no layers are frozen and all of them are randomly initialized. In the resulting model, all layers are trained according to the supervised loss. This network was trained with the adam optimizer ($\epsilon = 10^{-8}$), a default learning rate of 0.001 with a 5% decay per epoch and 20% dropout on the first two dense layers. It quickly converges to an accuracy of 88.16% on the test set after five epochs. While it is not the goal of this thesis to optimize this fully supervised network by modifying its architecture, it was nevertheless made sure that the performance cannot be increased further by using different learning or dropout rates.

In contrast, the encoder+dense network with the encoder part taken from the $\epsilon = 10^{-8}$ model after ten epochs of training completely failed to converge, and could not improve its performance beyond guessing. To investigate this behaviour, histograms of the outputs of the neurons in the three dense layers – the only trainable layers in the network – were plotted during the start of the training (figure 3.8).

Each subfigure contains four histograms, and shows the network at different points during the training: From the randomly initialized start in the top left, to after it was trained on

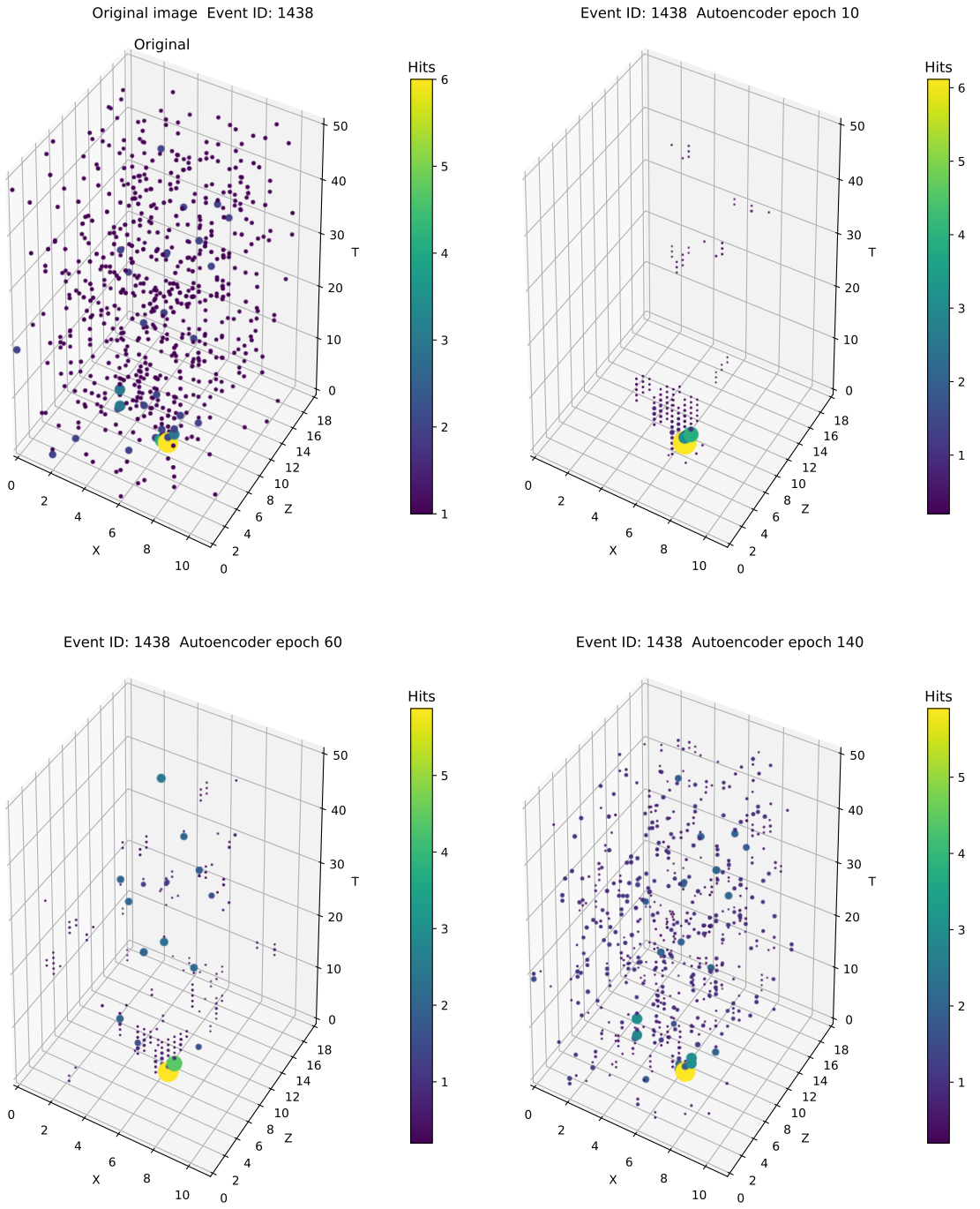


Figure 3.6.: Reconstruction of a 20 GeV CC event by the autoencoder of *model-1920*, trained with adam and $\epsilon = 10^{-1}$. The top left shows the original image, and the other histograms are the attempted reconstructions after different numbers of epochs trained. Bins with less than 0.3 counts are removed in the reconstructions to improve the visibility of clusters. These images show the input before zero centring is applied, and the reconstruction after it is removed again.

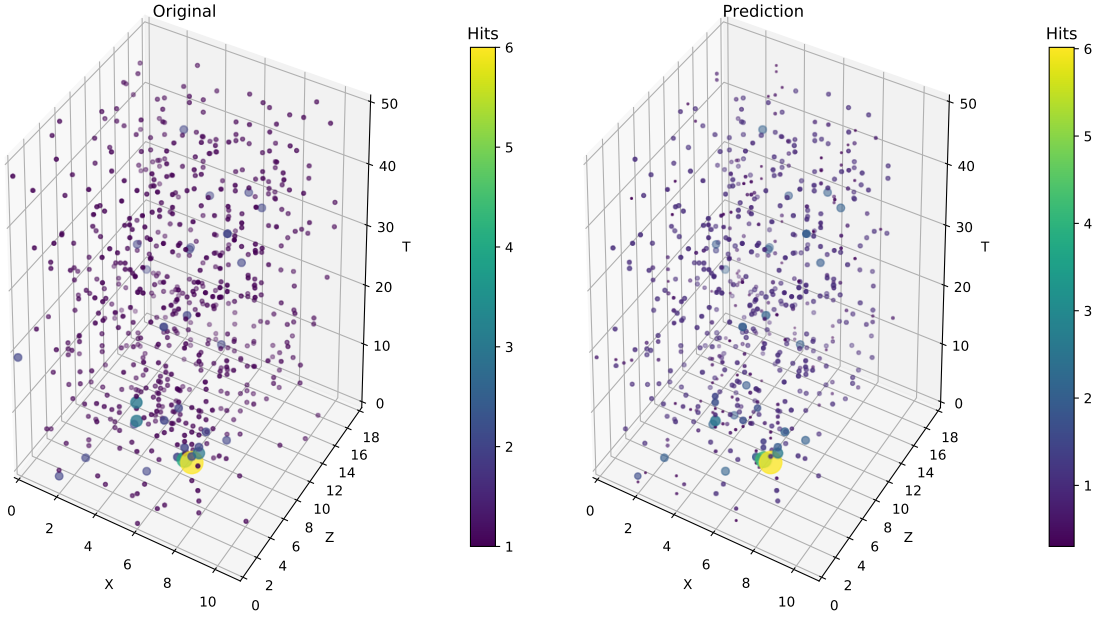


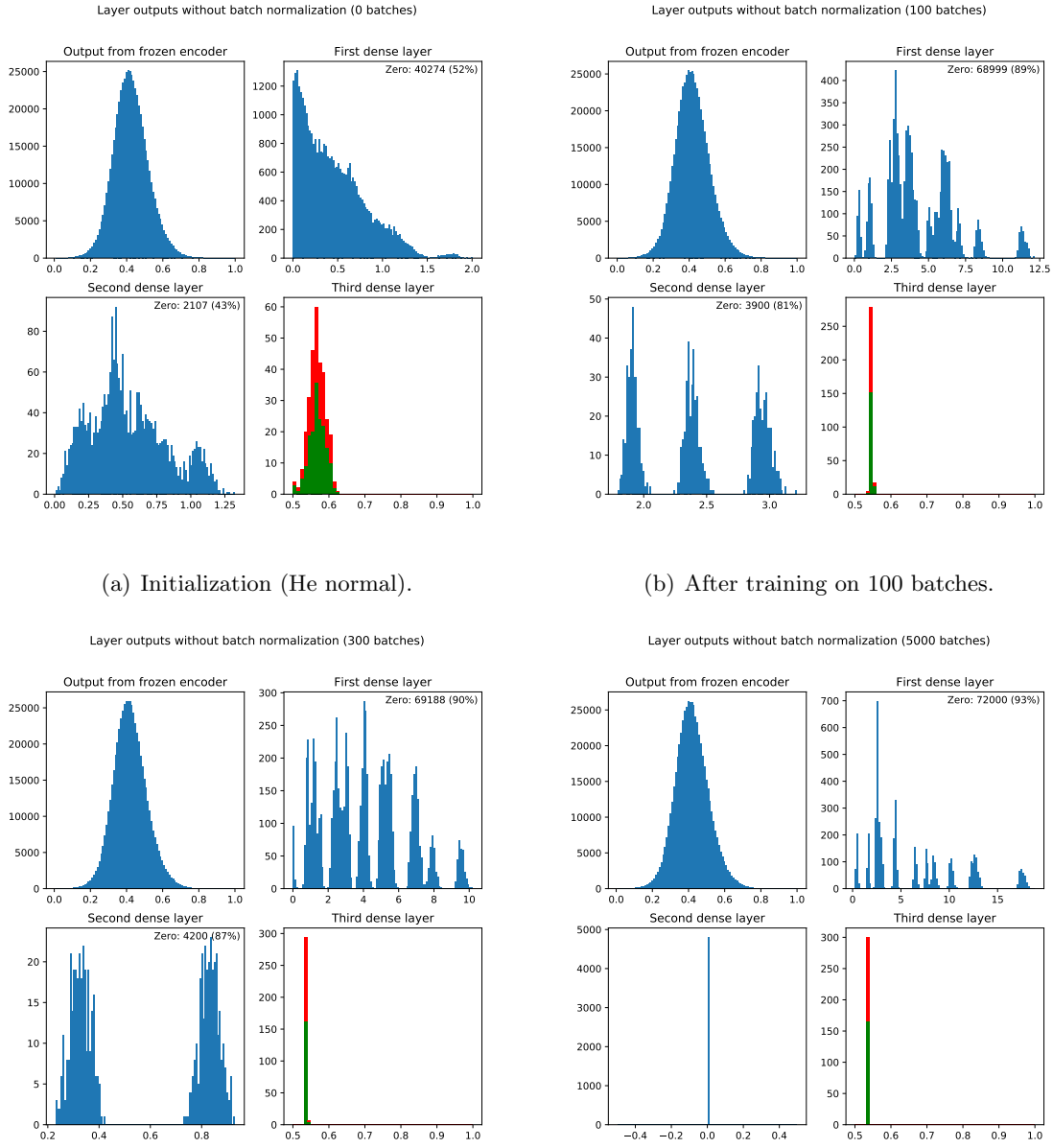
Figure 3.7.: Original and reconstruction of the *model-1920* autoencoder, trained with adam and $\epsilon = 10^{-8}$ for 119 epochs. The same event as in figure 3.6 is used.

5000 batches in the bottom right. For each histogram shown in these four subfigures, the same 300 samples from the test set were fed into the network. The outputs from all the neurons in a layer were calculated for these samples. Due to the last batch normalization layer in the encoder part, the first histogram is shaped like a Gaussian. As the training progresses, the distributions of the outputs in the other three layers change.

After 5000 batches, which is about 10% of an epoch, all neurons in the second to last layer constantly output zero for all samples, so that the classification of events depends exclusively on the two biases in the last layer. In other words, all connections to the input were cut in the network during training, and the same output is produced all the time instead.

To solve this issue, consider the encoder+dense network as two separate networks: The encoder, which preprocesses the data into a condensed representation of its features, and the dense layers, which extract the direction of the particle from that. The last layers in the encoder are a batch normalization, a ReLu activation function and an average pooling. The batch normalization leads to the shape of a Gaussian in the input to the dense network, which can be seen in each top left histogram of figure 3.8. As was explained in chapter 2.6.4, it was designed with the intent to whiten the output of neurons, but still allow the network to change this distribution if need be. This allowed the encoder to produce a Gaussian distribution with a non-zero mean. While it might be advantageous in the innermost layer of the autoencoder, it is suboptimal as an input to the dense network, as inputs should usually be zero-centred.

To fix this, a custom designed layer was added before the first dense layer, which calculates the mean of the output from all its neurons for each batch, and subtracts that value from every output. This is essentially a batch-statistics based zero centring of the input data to the dense layers. With this addition, the encoder+dense network converges and reaches a top accuracy of 70.59% after 18 epochs. As it turns out, this can be improved even further by using a trainable batch normalization layer instead of the custom layer. The outputs of the neurons



(a) Initialization (He normal).

(b) After training on 100 batches.

(c) After training on 300 batches.

(d) After training on 5000 batches.

Figure 3.8.: Output of the last layer from the encoder and from all dense layers of the *model-1920* encoder+dense network during the start of the first epoch of training. For each picture, the same 300 events from the test set were fed to the network. For the first two layers, outputs of exactly zero were removed from the plot, and are shown as a number in the top right instead, together with their proportion of all outputs. For the last layer, the bars were coloured in green or red depending on whether they are a correct up/down prediction or not.

during training with this modified setup is shown in figure 3.9. The Gaussian distribution is shifted towards zero during training, meaning that the network learns to zero-centre the data. The accuracy of the network increased to 74.54% after four epochs and, with the addition of 20% dropout on the first two dense layers to reduce overfitting, even manages to reach 76.09% after 50 epochs.

Successive training of the encoder+dense network

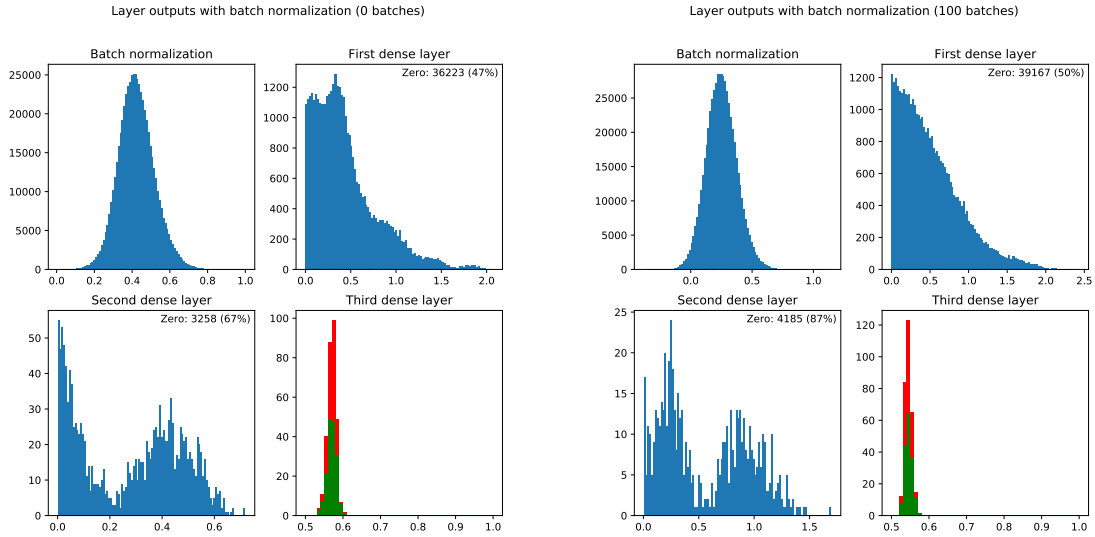
For the above, the encoder used for the encoder+dense network was taken from the autoencoder which was trained with adam and $\epsilon = 10^{-8}$ for ten epochs. This choice was arbitrary, and the state of convergence of the autoencoder naturally has a strong impact on the final classification performance. In the following, the correlation between the loss of an autoencoder and the up/down-accuracy of a fully trained encoder+dense network build from its encoder part will be investigated. For this, some autoencoders at different points in the training with either an adam $\epsilon = 10^{-8}$, adam $\epsilon = 10^{-1}$ or SGD optimizer are used as a basis for the encoder+dense networks, which are then trained until their accuracy is at its maximum on the test set. Similar to the fully supervised network, the encoder+dense phase is trained with a 20% dropout on the first two dense layers, a default learning rate of 0.001 with a 5% decay per epoch and an adam optimizer with $\epsilon = 10^{-8}$. In contrast to the case of the unsupervised learning phase, the choice of epsilon does not influence the outcome of the training in phase two.

The result is shown in figure 3.10. For the model with $\epsilon = 10^{-1}$, and going from high autoencoder losses on the right to smaller ones on the left, the highest achievable accuracy increases steadily at first. This is presumably because more and more information about the original event is encoded and therefore accessible to the dense layers. Counter-intuitively, the performance then begins to decrease again linearly after the autoencoder has reached a mean squared error of about 0.07, losing up to about 4% accuracy (absolute) until epoch 140. The performance of the encoders with the SGD optimizer align with this linear decrease. Such a reduction is also visible for the autoencoder with $\epsilon = 10^{-8}$ as well, albeit the accuracies appear to be shifted down by about one percent as compared to the $\epsilon = 10^{-1}$ case.

To make sure that this behaviour is not exclusive to the up-down-classification task, another short test has been performed with the same encoders: The aim of the dense layers is to predict the particle ID, that is, to classify events as either track- or shower-like (see section 1.4). The resulting accuracies are depicted as crosses in the above plot and show the same behaviour of a decreasing accuracy. Furthermore, for models predicting the energy of neutrinos, which are covered later in the thesis, the same pattern is observable as well, suggesting that this is not problem-specific.

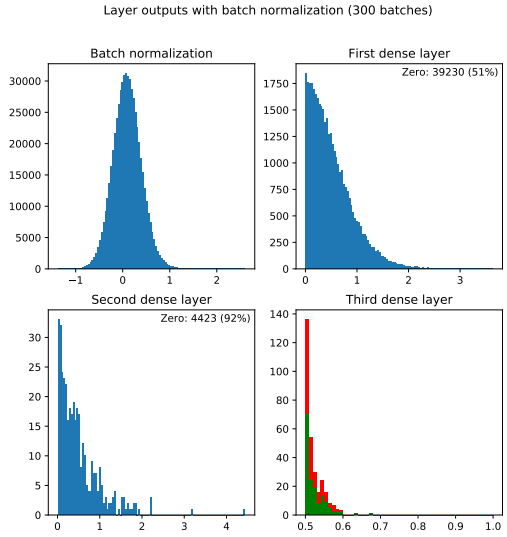
A simple potential explanation for this decrease could be that it is an initialization problem: Maybe the optimum of the $\epsilon = 10^{-1}$ model at epoch ten is not easily accessible anymore as the encoder is trained further - after all, it is supposed to optimize according to a different task during the autoencoder phase than during the supervised one. To test this hypotheses, the encoder+dense networks were trained in a successive fashion: The model is initialized with the encoder of the first epoch of autoencoder training. As the dense layers are trained, the weights of encoders from further trained autoencoders are loaded in after every few epochs. This way, the structure of the output of the encoder presumably changes slowly, allowing the dense layers to adapt to a potential shift of the optimum.

For example, consider the following successive training of the autoencoder with $\epsilon = 10^{-8}$: The encoder+dense network was initialized with the weights of the encoder of autoencoder epoch one and then trained as before. Following its convergence after 26 epochs, the weights of the encoder from autoencoder epoch two are loaded in, and the training is resumed for ten epochs with a constant learning rate of 0.0003. This is repeated once, and then a new encoder

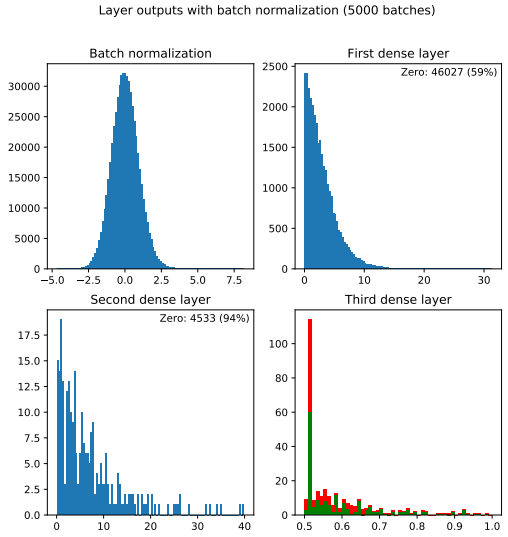


(a) Initialization

(b) After training on 100 batches



(c) After training on 300 batches



(d) After training on 5000 batches

Figure 3.9.: Output of all layers of the *model-1920* encoder+dense network with an additional batch normalization layer in the beginning during the start of the first epoch of the training.

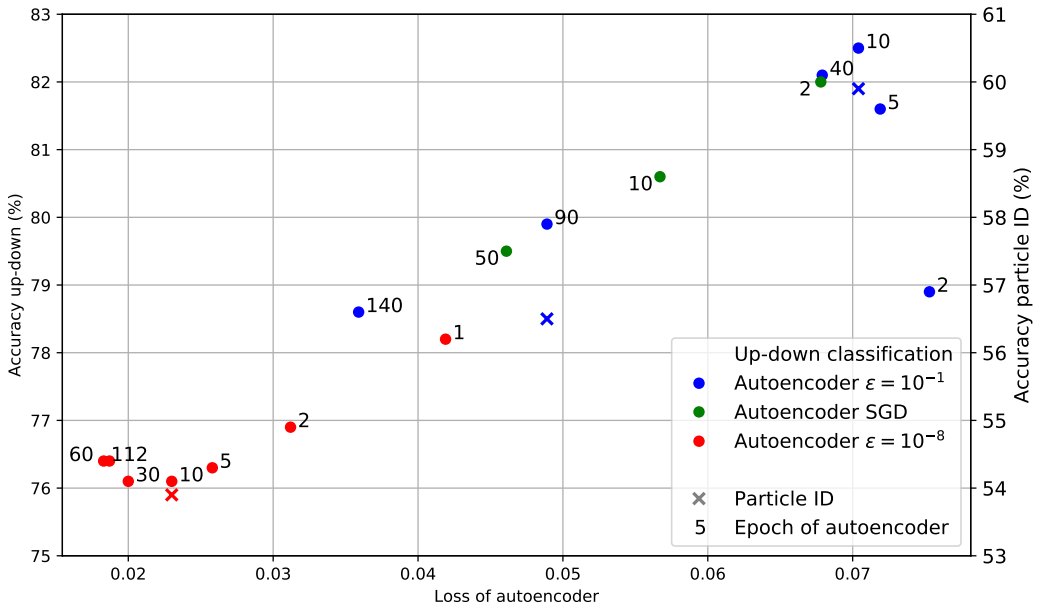
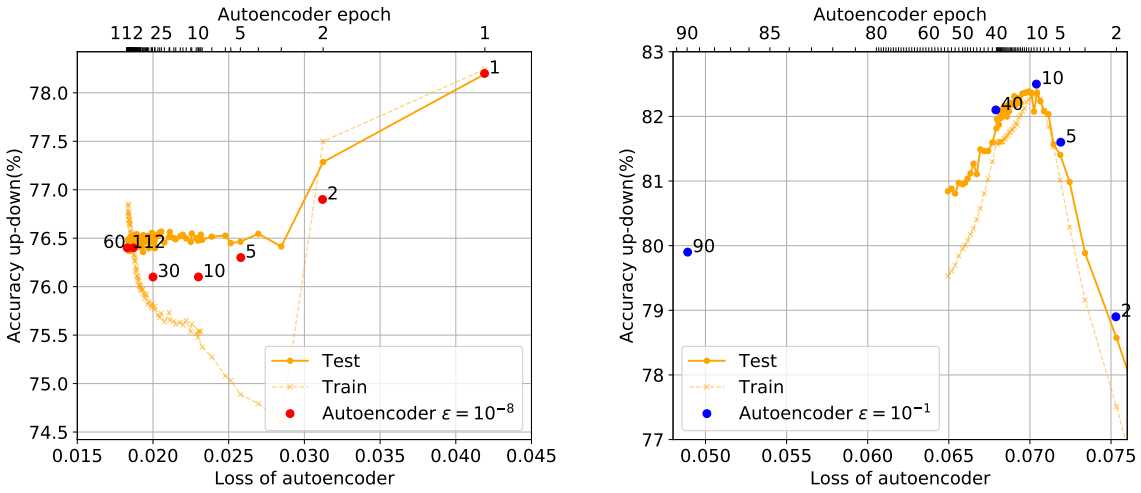


Figure 3.10.: The highest accuracy of encoder+dense networks over the loss of the autoencoder from which the encoder was taken from. Dots are for the up-down classification task and crosses are for a particle ID classification, in which events are classified as either track- or shower-like. The black digits represent the number of passed epochs in the training of the corresponding autoencoder. The colors identify the used optimizer.



(a) Adam with $\epsilon = 10^{-8}$

(b) Adam with $\epsilon = 10^{-1}$

Figure 3.11.: Up/down-accuracy of encoder+dense networks, with the encoder being taken from autoencoders with adam optimizers and $\epsilon = 10^{-8}$ or $\epsilon = 10^{-1}$ after different numbers of epochs are trained. The x-axes show the loss and the epoch of the autoencoder to which the encoder belongs, while the y-axis shows the highest accuracy achieved on the test set with the corresponding encoder+dense network. The red and blue dots are carried over from figure 3.10 and show the accuracy if the encoder+dense net is initialized with the weights of the autoencoder. The orange lines represent the test and train accuracy of the encoder+dense networks as obtained by successive training.

is loaded in after every supervised epoch. The following table shows how many epochs the encoder+dense network was trained for with each encoder:

Autoencoder epoch	Trained supervised epochs	Learning rate
1	26	0.001, -5% per epoch
2	10	0.0003
3	10	0.0003
4	1	0.0003
5	1	0.0003
6	1	0.0003
...	...	...

The result is shown in figure 3.11(a). The decline in accuracy is still visible, but the valley between epochs 5 and 60 does not occur; it seems to have in fact be caused by an increasingly inaccessible minimum.

The successive training procedure is repeated for the autoencoder with the $\epsilon = 10^{-1}$ optimizer, but this time the supervised epochs are spread more evenly between the autoencoder epochs:

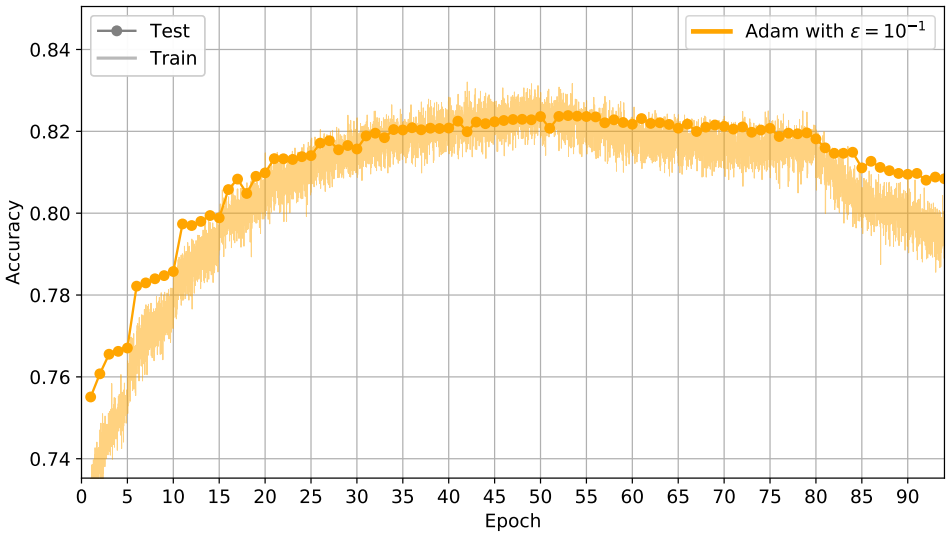


Figure 3.12.: Up/down-accuracy over trained epochs for the successive training with the adam $\epsilon = 10^{-1}$ autoencoder. The jumps at epoch 5, 10, etc. are due to new encoders being loaded in. The performance before these jumps are the ones plotted in figure 3.11(b).

Autoencoder epoch	Trained supervised epochs	Learning rate
1 - 10	5 each	0.001, -2.5% per epoch
11 - 50	1 each	continued -2.5% per epoch
51	1	0.0003
52	1	0.0003
...	...	...

The initial encoder epochs are assigned more supervised epochs than the later ones, since the network is expected to be far from the optimized state after initialization, meaning that some training is necessary to get into the proximity of the optimum. It can be seen from figure 3.11(b) that the drop in performance for the autoencoder with $\epsilon = 10^{-1}$ cannot be fixed by using this training method either.

Figure 3.12 shows the performance of the encoder+dense networks during the process of this successive training. In the early epochs, jumps are clearly visible whenever the weights of a new encoder epoch are loaded in according to the schedule above. This is likely because more information about the event is encoded and therefore accessible by the dense layers. Also, the accuracy increases even without loading in new epochs as the dense layers learn to extract the information better. Later, the accuracy on a single autoencoder epoch is mostly flat (e.g. epochs 20 to 25), meaning that training less than 5 epochs per autoencoder would be reasonable at this point and could be used to shorten the training time.

The lower the loss of the autoencoder, the better it is able to reconstruct the original image. Intuitively, one might think that an encoder which performs better in the reconstruction would encode more information about the image in its bottleneck layer. The dense layers, which extract properties of the particle from this encoded representation, would consequently be expected to perform better, or, if the additional information is not helpful for the up/down classification, at the very least to not perform worse when preceded by a better encoder. On

the contrary, the best performing autoencoder in the training in this chapter (adam with $\epsilon = 10^{-8}$, epoch 112) contained the encoder that was worst in the supervised task. This unexpected behaviour will be investigated to greater detail in chapter 5.3.

As the ideal encoder epoch for the autoencoder is not identical to the one for the encoder+dense phase, the whole training process is somewhat more complex: Instead of just training the autoencoder to the minimum of its loss and then taking the respective encoder part for phase two, one would have to train a new encoder+dense network for every epoch of autoencoder training in order to achieve optimal results. Since every one of these networks requires about a day of computation time to be brought to convergence, this is suboptimal for extensive model architecture studies as performed in this thesis.

Instead, the **successive training** introduced in this section will be used for future models to conduct a scan over all autoencoder epochs and find the ideal encoder for the supervised task. With this, only down to a single epoch of encoder+dense training is needed per autoencoder epoch, reducing the necessary total computation time per autoencoder by an order of magnitude. The learning rate will be kept above a threshold in the process, so that the dense layers can still adjust to the new encoder within the scope of one epoch. Once the best epoch was found, a new encoder+dense network will be initialized with the weights of the ideal encoder. This is then trained as before, reducing the learning rate to achieve the best possible accuracy.

From all autoencoders so far, the one with $\epsilon = 10^{-8}$ trained for 10 epochs was the best performing for the encoder+dense net: It reached an accuracy of 82.51% after 23 epochs over the whole energy spectrum. In figure 3.13, its performance is plotted over the energy of the particles in comparison to the completely supervised approach (88.16%). Both networks show a drop in accuracy towards low energies, as the expected number of hits measured per event in the whole detector depends on the energy of the particle. If only a few photons of a low energy event were detected, the reconstruction of the direction of its trajectory is expected to be more difficult.

So far, the autoencoder strategy is significantly outperformed by the supervised network if they are tested exclusively on simulations. The next two chapters 4 and 5 investigate ways to optimize both the architecture of autoencoders, as well as the performance of the encoder+dense networks. In chapter 6, the influence of differences between simulations and measured data on the autoencoder and the supervised approach are tested.

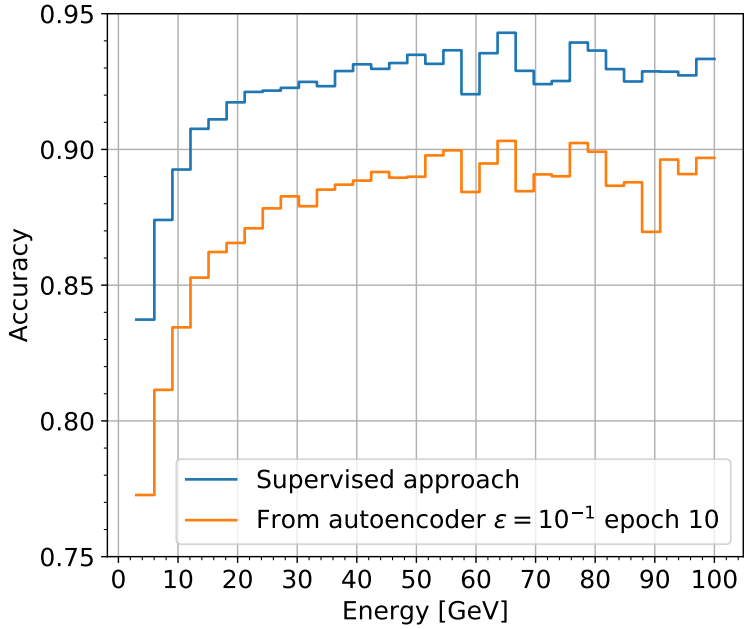


Figure 3.13.: Up/down-accuracy, binned over the energy of the particles with a bin size of about 3 GeV. The network trained completely supervised is compared to the best performing encoder+dense network from the *model-1920* autoencoders from this chapter (Autoencoder with adam and $\epsilon = 10^{-1}$ trained for 10 epochs, encoder+dense trained for 23 epochs).

4. The architecture of autoencoders

The autoencoder network architecture of the *model-1920* (table 3.2) used for the previous chapter is motivated by the structure of popular convolutional-based image recognition networks like VGG [14]. The decoder is essentially a mirrored version of this net, with the bottleneck of the autoencoder as a point of symmetry. In the following, various changes to this layout of autoencoders will be tested by comparing their performances on the xzt dataset.

To make the different models comparable, they have been constructed in such a way that the amount of free parameters is approximately the same for all of them. Also, they are trained with the same optimizer settings, learning rate schedule and size of bottleneck:

Optimizer:	Adam	$\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$
Learning rate:	0.001 (epoch 1)	Reduced by 5% per epoch
Cost function:	Mean squared error	
Bottleneck:	1920 neurons	

4.1. Network depth and width

When constructing deep convolutional networks, an important decision to make is how many convolutional layers are stacked on top of each other, and how many filters each of them possesses. The *model-1920* autoencoder used in the previous chapter features 14 convolutional layers - disregarding the last layer which has a reduced kernel size. It is possible that a reduction of this number would benefit the performance of the network, while the total number of free parameters is kept constant by increasing the amount of filters per layer instead. By this, the autoencoder is made less deep, but the layers are "wider" in exchange. Conversely, a deeper autoencoder with "narrower" layers may prove superior.

To test this, five networks with different depths and widths were designed based on the *model-1920* and trained for 30 epochs. The detailed architectures with 12, 16, 20, 30 and 60 layers can be found in the appendix in tables A.1 to A.5. The basic idea behind their designs is to use three pooling layers in the encoders of each of them, and to double the number of filters after the second pooling. This number is scaled up or down across the whole network depending on the amount of layers to keep the count of parameters constant. The innermost convolution right before the bottleneck always has 64 filters, so that the size of the bottleneck does not change. The symmetry of the encoder and decoder is ensured by always adding pairs of normal and transposed convolutions, each in the corresponding part.

Figure 4.1 shows the test and train losses of the autoencoders during training. The model with 14 layers is the $\epsilon = 10^{-8}$ autoencoder from chapter 3.4.2. For the very deep architectures with 30 and 60 layers, the training appears slightly unstable as jumps in the test loss are visible from epoch to epoch. For both the very deep models and the most shallow one with 12 layers, the test loss is significantly higher than the train loss initially, which can be a sign of overfitting. However, this difference decrease over time, and is mostly vanished at epoch 30, while classic overfitting is expected to grow worse as training progresses. Similarly, the losses

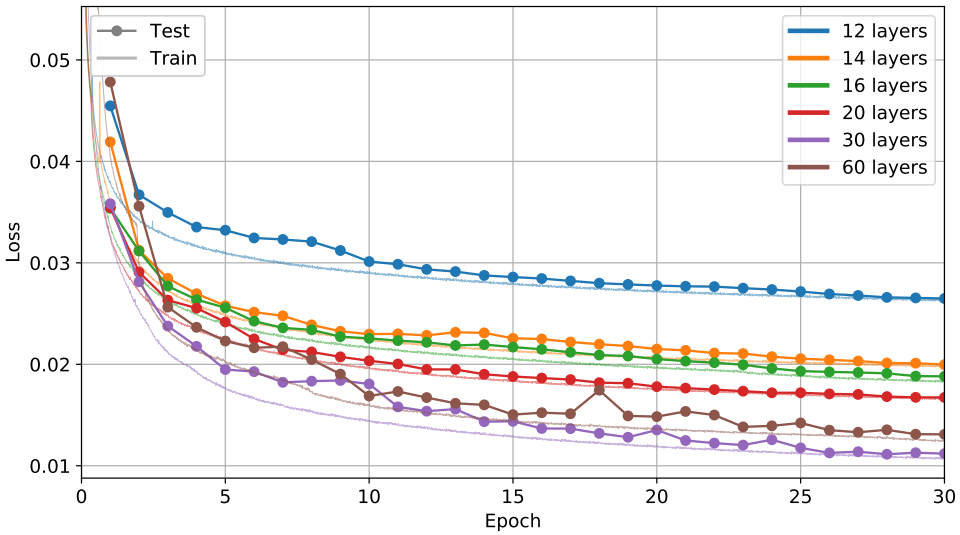


Figure 4.1.: Mean squared error loss of autoencoders with different numbers of convolutional layers during training. All networks were trained with the same learning rate schedule for 30 epochs and have a comparable amount of free parameters.

Table 4.1.: Mean squared error loss on the test set of autoencoders with different numbers of convolutional layers after 30 epochs of unsupervised training. The number of free parameters in the networks and the average time in minutes it took to complete one epoch are listed as well. The network with 14 layers (*) is the *model-1920* autoencoder from the previous chapter.

Convolutional layers	Free parameters	Loss (MSE, epoch 30)	Time per epoch (min)
12	745,065	0.0265	184
14*	748,609	0.0200	124
16	750,193	0.0188	**117**
20	747,681	0.0167	164
30	756,040	**0.0112**	201
60	758,324	0.0131	287

are more stable in higher epochs, suggesting that both might be caused by an underlying effect which is limited to very early parts of the training. The train loss is very stable for all networks throughout.

The test losses of all autoencoders after epoch 30 are listed in table 4.1. Since the time needed to train the models is important in practice, the average passed time per epoch is shown, too. The best performing model is the one with 30 convolutional layers, more than double the amount of the original *model-1920*. It has a 44% lower mean squared error on the test set, but takes 62% longer to train. In contrast, removing layers from the initial model and making them wider instead lead to a higher loss and longer train time. Likewise, doubling the depth of the 30 layer network also proved to be worse in both aspects.

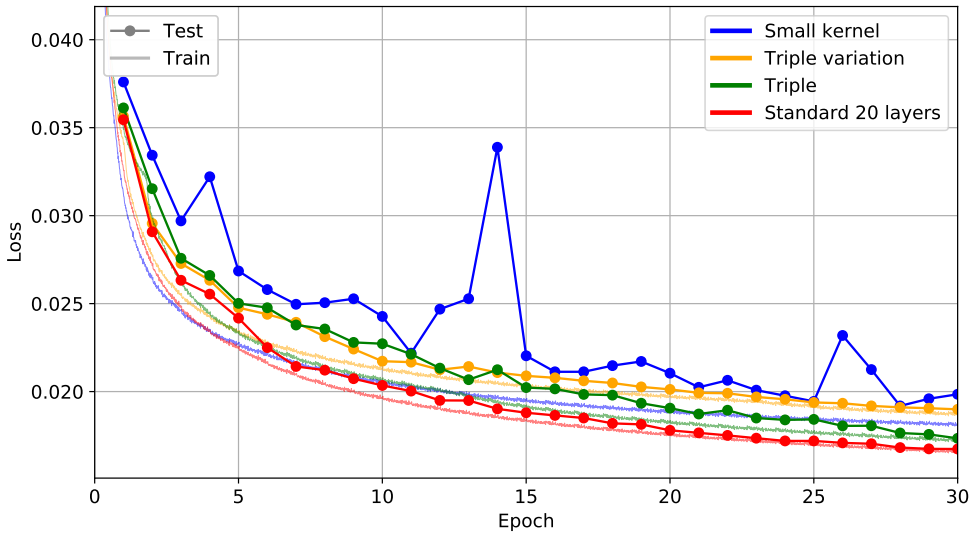


Figure 4.2.: Mean squared error loss of autoencoders with different properties during training. All networks were trained with the same learning rate schedule for 30 epochs and have a comparable amount of free parameters.

4.2. Design

In this section, some essential changes to the design of autoencoders will be tested for their effect on the performance.

Reduced kernel size: The kernel size of the convolutional layers of image recognition networks are often chosen to have an edge length of three. Heuristically, it is sometimes argued that this results in the smallest quadratic kernel that can distinguish between the up-down and right-left directions independently. However, the input images to these networks are much larger than it is the case for binned ORCA data: The VGG net for example is trained on 224 by 224 pixel pictures, whereas the size of the binned xzt data is $11 \times 18 \times 50$. Here, a size three kernel covers more than a third of the bins in the x direction of the input and even more for the smaller images deeper in the network. This might be more than necessary.

In figure 4.2, the training of a 20 layer model with a kernel size of $2 \times 2 \times 2$ (table A.6) is compared to the 20 layer model from the previous section with a $3 \times 3 \times 3$ kernel. Since a smaller kernel has fewer weights, additional filters were added, so that the number of free parameters is similar and the networks remain comparable in that regard. This alternative design turns out to be inferior to the original one, both in terms of the loss after 30 epochs (table 4.2) as well as the instability in the training, as the test loss varies greatly between the epochs.

Arrangement of filters: In the VGG network, the number of filters gets increased not only once, like in the above models, but multiple times, always following a pooling operation. The model *triple* is designed to resemble this (table A.7). The number of convolutional layers is kept constant, but the amount of filters per layer is scaled up or down in each section, so that the overall count of free parameters remains almost unchanged. The loss after 30 epochs is slightly higher with this change than for the standard architecture. At the same time, the average duration to train one epoch was reduced. This could be because the number of filters in the first section, where the image size is still large and before any pooling layers, has a

Table 4.2.: Mean squared error loss on the test set of autoencoders with different properties in their structure after 30 epochs of unsupervised training. All models have 20 convolutional layers. The number of free parameters and the average time it took to complete one epoch are listed as well.

Model	Free parameters	Loss (MSE, epoch 30)	Time per epoch (min)
Small kernel	752,634	0.0198	267
Triple variation	745,957	0.0190	126
Triple	750,921	0.0173	**122**
Standard	747,681	**0.0167**	164

stronger impact on the runtime than that of deeper sections. The number of calculations necessary for a convolution depends, among others, on the size of the image. Since the standard 20 layer model has 26 filters in the first section as compared to the 14 of the *triple* model, more calculations are necessary and the run time is expected to increase.

Decoder symmetry: It was mentioned in section 3.3 that the structure of the autoencoders has been constructed to be symmetric around the bottleneck in terms of both the input and output of layers. Another way of implementing a symmetry is to design the decoder only with respect to the output dimensions of the layers, which might seem more logical when considering the model structure as shown in table A.8. Each section of the decoder now has the same amount of output filters, separated by pooling or upsampling layers, but an otherwise identical structure to the *triple* model. However, this architecture turned out to be worse in both the performance as well as the training time.

4.3. Conclusion and discussion

The ideal depth for autoencoders used on xzt data was found to be between 20 and 60 convolutional layers. Changes to their design did not improve the performance. From the tested models with a comparable number of free parameters, the network with 30 layers had the lowest mean squared error loss on the test set after 30 epochs of training. As the autoencoders are optimized according to this loss, bins containing nothing but background noise are reconstructed as closely as possible as well. Since these bins are likely irrelevant for predicting properties of the neutrinos, the mean squared error function might not be an ideal choice for acquiring good encoder+dense networks. The autoencoders might show different performances when provided with other losses.

The encoder+dense network in the previous chapter 3.4.2 showed a decrease in performance as the loss of the autoencoder fell below a critical value of about 0.07. A similar behaviour is observed for many autoencoders in the following chapter as well. This questions the importance of an optimized autoencoder design for the quality of the encoder+dense network, which is ultimately the goal of the unsupervised autoencoder approach. For this reason, further optimization of the autoencoder design with respect to its loss are omitted. Instead, the focus will be placed on its impact on the supervised encoder+dense phase.

5. Optimisation of the encoder+dense network

The goal of the unsupervised approach is to train the encoder+dense network which predicts properties of the neutrinos. Most of the neurons in this network are located in the encoder and are trained as a part of the autoencoder, so its design is closely connected to the performance of the encoder+dense network. For this reason, most networks in the following chapter differ not in the architecture of the dense layers, but in that of the encoder part, even though this section of the net is not trainable in the encoder+dense phase.

5.1. Size of the bottleneck

Autoencoders are based on the idea of introducing a bottleneck in the model architecture. This forces the network to find a compact representation of the input image which contains as much of its information as possible, so that a detailed reconstruction can be made. The choice of the size of the bottleneck is crucial for the performance: If it is too big, there might be no incentive to abstract features; If it is too small, it is impossible to funnel all relevant information through the constriction, even if the network could learn to extract them. In both cases, the resulting encoder+dense network is expected to suffer in performance. It is noteworthy that the qualities of the autoencoder and the encoder+dense phase are not necessarily correlated; an autoencoder with no bottleneck can generate perfect reconstructions by simply applying identity transformations to its input without learning to abstract a single feature, likely leaving the small dense network overtaxed with the challenge.

The size of the bottleneck is an important hyperparameter exclusive to autoencoders, and this section will investigate its influence on the resulting networks for two different tasks: An up-down classification, as in the previous chapters, and an energy regression. In contrast to the supervised approach, which requires the training of a new network for each of those tasks, the first phase of the unsupervised training procedure is independent of the goal of the second phase. The same autoencoders trained for the up-down classification can be reused for the encoder+dense nets predicting the energy of the particle, and, in principle, for any other property, too.

However, there is no guarantee that an autoencoder, whose encoder part is suited well to reconstruct a specific property, will also be ideal for another one. Once a set of autoencoders has been trained, one has to separately find out which of them is ideal for each reconstruction. This can be done fairly quickly, since the encoder+dense networks are much faster to train than the autoencoders.

5.1.1. Up-down classification

The goal of the networks in this section is to classify events according to the direction in which the neutrino travelled through the detector. In practice, this could be used to separate background events in the form of atmospheric neutrinos or muons, that have not propagated through the Earth and are therefore not used to determine the mass hierarchy. All networks feature two neurons in the final layer, which are used for categorizing the events as up- or

down-going neutrinos. Their activation is the softmax function (section 2.2), and the loss of the models is the categorical cross entropy (section 2.4). The performance of the networks is assessed in terms of their accuracy, that is, the fraction of events correctly classified as up- or down-going in the whole training or test dataset. To make networks comparable, they have a similar amount of free parameters.

In section 3.4.2, it was found that the autoencoder trained with the adam optimizer and $\epsilon = 10^{-8}$ has a lower loss than the one with $\epsilon = 10^{-1}$. However, the goal is to develop a model with a high accuracy in the encoder+dense phase, and the $\epsilon = 10^{-1}$ has proven to be superior in that regard. For this reason, all following models will be trained with this value of epsilon. A dropout of 20% is used by default on the dense layers, while none is used on convolutional layers.

Size 600

The first set of models will test a reduction of the size of the bottleneck from previously 1920 neurons to 600 neurons. In the bottleneck itself, the neurons still possess a spatial structure: For the *model-1920* for example, the bottleneck had the dimension $2 \times 3 \times 5 \times 64$, with the last dimension being the filters. This can be thought of as 64 different $2 \times 3 \times 5$ images. Convolutions will slide their kernel over these images when applied to them, utilizing possible translational invariances. If the size of the bottleneck is to be reduced, it raises the question of whether the size of the images or the number of filters should be lessened. Three different models with 600 neurons in the bottleneck were designed to test this:

- Model *600-20 filters*, dimension of bottleneck: $2 \times 3 \times 5 \times 20$
  16 convolutional layers, 3 pooling layers (table A.9)
- Model *600-50 filters*, dimension of bottleneck: $2 \times 2 \times 3 \times 50$
  16 convolutional layers, 4 pooling layers (table A.10)
- Model *600-75 filters*, dimension of bottleneck: $2 \times 2 \times 2 \times 75$
  16 convolutional layers, 4 pooling layers (table A.11)

All models are trained according to the autoencoder training procedure developed in section 3.4: The autoencoder is trained first, then a supervised network is trained successively by loading in the weights of new encoder epochs periodically to determine the autoencoder epoch that is best for the encoder+dense phase. Based on the experience with *model-1920*, the weights are loaded in according to a new schedule which is shown in table 5.1.

Finally, an encoder+dense network is trained with that encoder until the maximum of its accuracy is reached. The latter is trained with adam, $\epsilon = 10^{-8}$, the default initial learning rate of 0.001, together with a 5% decay per epoch. These settings have worked well for the models in the previous chapter, not showing signs of overfitting or requiring further readjustments of the learning rate.

The model *600-20 filters* has shown a strong tendency to overfit in the encoder+dense phase, even when the dropout of dense layers is increased to 60%. The autoencoder phase did not suffer from this issue. The highest achieved accuracy was 72.42%, which puts it far behind the other architectures. The successive trainings of the models *600-50 filters* and *600-75 filters* are shown in figure 5.1. For both models, the initial learning rate was the default value of 0.001, and it was increased by a factor of ten at epochs 22 and 53. They show a similar behaviour during training, and both experience a drop-off in accuracy shortly after epoch 50, at an autoencoder loss of about 0.068. The drop-off coincidentally occurs at roughly the same epoch at which the learning rate was increased for the second time, as this is where

Table 5.1.: Schedule of successive training used for all models in this chapter. It shows the number of epochs the encoder+dense network is trained for with the weights of the encoder of a certain autoencoder epoch. The decay of the learning rate is stopped at 0.0005, so that the dense layers can still adapt to the new encoder during a single epoch.

Autoencoder epoch	Trained supervised epochs	Supervised learning rate
1	10	0.001 - 5% per epoch
2-6	2	*continued*
7-15	1	*continued*
16	1	0.0005
17	1	0.0005
...	...	

Table 5.2.: Highest accuracy of phase two of the autoencoder training for three different models with a bottleneck size of 600. The average time per epoch of autoencoder training is also shown.

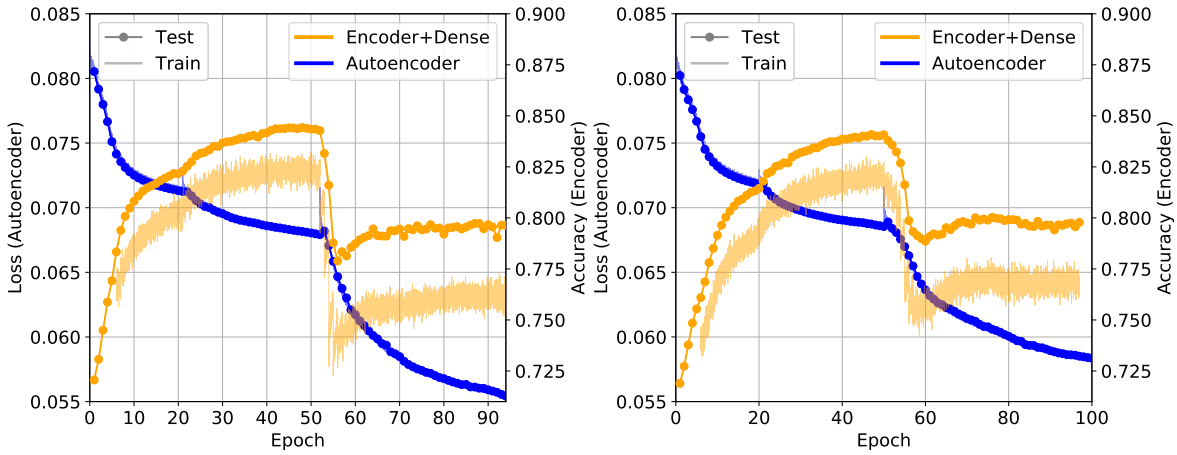
Model	Highest accuracy	Time per epoch (min)
600-20 filters	72.42%	158
600-50 filters	**84.93**%	137
600-75 filters	84.38%	117
600-75 filters, 15 layers	84.43%	96

the autoencoder reaches its critical loss. The drop-off also occurs when the same model is retrained with a constant learning rate (this is shown in figure 5.13 in section 5.3).

The accuracy is consistently higher on the test set than it is on the train set. This is the intended behaviour of the dropout that was applied to the dense layers (see section 2.6). The best encoders for phase two were found to be after 48 (*600-50 filters*) or 47 epochs (*600-75 filters*) respectively. By letting the encoder+dense network converge with these encoders, a maximum accuracy of 84.93% or 84.38% was reached (table 5.2). In figure 5.2, the performance of these two networks is evaluated on the test set, binned over the energy of the events.

A reduction of the filters down to 20 had a heavy impact on the performance of the encoder+dense network. This could be because the autoencoder makes use of the spatial ordering of neurons in the individual images with its convolutional layers if the images are still large in the bottleneck. The dense layers might not be able to utilize this to the same degree as the convolutions, which would explain the worse result. Both the models with 50 and 75 filters had a very similar accuracy, with the 50 filter one being slightly superior. This indicates the presence of an optimum for the image size-to-filter ratio, which is at $2 \cdot 2 \cdot 3/50 = 0.24$ for the model *600-50 filters*.

Table 5.2 also lists the model *600-75 filters, 15 layers*, which only has one instead of two convolutional layers before the first pooling layer, but the same number of free parameters. The structure is otherwise identical to the one of model *600-75 filters*, and is shown in table A.12. This model did not only reach a slightly higher accuracy, but also was 18% quicker to train. This is because convolutions applied to large images as the input require more calculations than those applied to small ones deeper in the net, even if the number of free parameters is the same in the model. With respect to this behaviour, the following architectures have a small number of layers before the first pooling as well.



(a) Model *600-50 filters*

(b) Model *600-75 filters*

Figure 5.1.: Successive training of the two different models with 600 neurons in the bottleneck. The loss of the autoencoder is shown together with the up-down accuracy of the encoder+dense model. The learning rate was increased by a factor of 10 at epochs 22 and 53. The training accuracy is omitted for the first six epochs, since multiple supervised epochs were trained per autoencoder epoch in that interval according to the schedule in table 5.1.

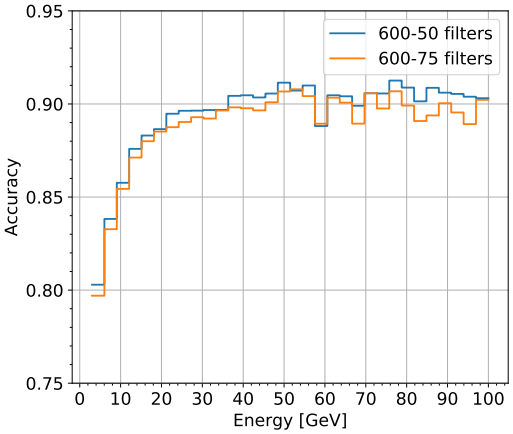


Figure 5.2.: Up-down accuracy of the models *600-50 filters* and *600-75 filters* over the energy of the particles.

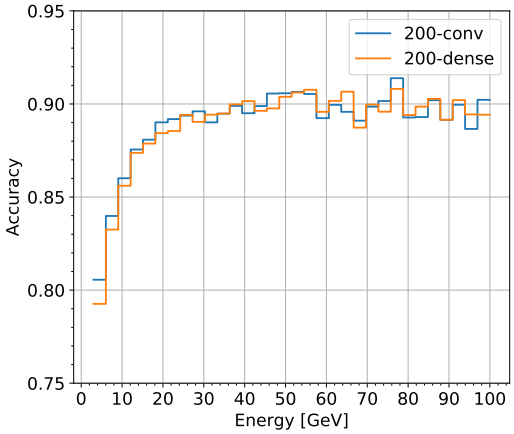


Figure 5.3.: Up-down accuracy of the models *200-conv* and *200-dense* over the energy of the particles.

Table 5.3.: Learning rate schedule of the autoencoder training for the models with 200 neurons in the bottleneck.

Autoencoder epoch	1-20	21-54	55-97	98+
Learning rate	0.001	0.01	5% increase per epoch	0.2

Table 5.4.: Highest accuracy on the test set of the encoder+dense models with a bottleneck size of 200, 64 and 32.

Model	Dimension in bottleneck	Up-down accuracy
200-conv	$2 \times 2 \times 2 \times 25$	84.98%
200-dense	200	84.32%
model-64	$1 \times 1 \times 1 \times 64$	84.72%
model-32	$1 \times 1 \times 1 \times 32$	82.67%

Size 200

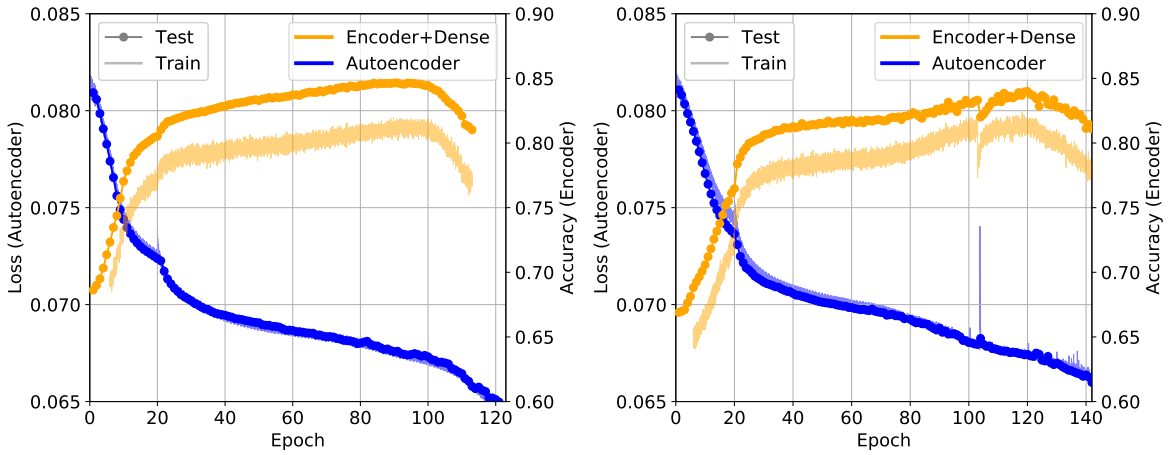
The networks in this subsection have 200 neurons in their bottleneck. The model *200-conv* is constructed similar to the model *600-75 filters, 15 layers* of the previous subsection, but the number of filters of the convolutional layers is reduced gradually towards the bottleneck (table A.13). It also has an additional convolutional layer each in the decoder and the encoder which are used to reduce the spatial dimensions. It was shown in chapter 4 that additional convolutions do not impair the ability of the network autoencoder to converge, but instead even slightly improve its performance. To compensate for the additional weights in the new layers, the number of filters in the previous ones were scaled down accordingly.

A different approach to the reduction of bottleneck's size is tested in the model *200-dense* (table A.14). Here, dense layers are used as the innermost layers of the autoencoders, which connect the 528 neurons from the last pooling layer to the 200 neurons in the bottleneck, and vice versa for the decoder part. The size of the bottleneck is particularly adaptable with this architecture, as one can simply change the output number of neurons in the dense layer. Completely convolutional based networks in contrast require the addition of new pooling layers or a change in the paddings for this, complicating the design process.

As before, successive training is used to acquire the best performing encoder networks. The learning rate schedule is shown in table 5.3 and plots of the training histories are shown in figure 5.4. As with previous autoencoders, the default learning rate can be increased greatly without destabilizing the training or causing overfitting. The time necessary for training could likely be shortened severely if a high learning rate was chosen from the start. Once again, the accuracy of the encoder+dense networks increase in the beginning, but experience a drop-off once encoders belonging to an autoencoder with a loss of about 0.068 are loaded in. The model *200-dense* performs slightly worse than fully convolutional based *200-conv*, as can be seen in the performance comparison in figure 5.3 and table 5.4. The following models will therefore not make use of dense layers in the bottleneck.

Size 64 and Size 32

For the models with 64 or 32 neurons in the bottleneck, a fourth pooling layer and an additional convolution is added to the architecture of the encoder, and a transposed convolution and an upsampling layer accordingly for the decoder. The spatial dimensions in the bottleneck are reduced to $1 \times 1 \times 1$ with 64 or 32 filters (tables A.15 and A.16). The learning rate schedule is shown in table 5.5 and the result of the successive trainings in figure 5.5. The dropout rate in the dense layers of the encoder+dense networks have been set to zero for the



(a) Model *200-conv* (b) Model *200-dense*

Figure 5.4.: Successive training of the two different models with 200 neurons in the bottleneck. The loss of the autoencoder is shown together with the up-down accuracy of the encoder+dense model.

Table 5.5.: Learning rate schedule of the autoencoder training for the models with 64 and 32 neurons in the bottleneck. The learning rate was reduced again as the losses approached their minimum.

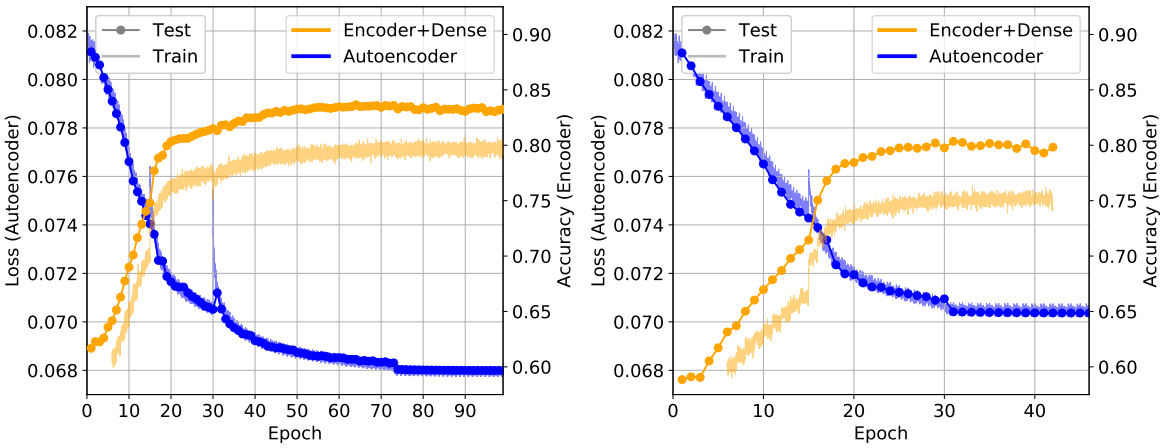
Autoencoder epoch	1-15	16-30	31-73	74+
Learning rate *model-64*	0.001	0.01	0.1	0.01
Learning rate *model-32*	0.001	0.01	0.001	/

continued training after the best epoch has been found. In contrast to the previous models, this improved the performance of the network, possibly due to the small number of neurons in the bottleneck. However, dropout was still used during the successive training to prevent the network from overfitting, as many epochs are trained without lowering the learning rate. Since the successive training is only used to identify the best epoch in relative terms, this does not affect the highest absolute performance.

The autoencoders are fully converged at the end of the trainings. The accuracy increases in the beginning, and then remains constant. In the previous autoencoders, a drop-off was visible during training at an autoencoder loss of about 0.068. For the *model-64* and *model-32*, the autoencoder is not able to achieve this loss because the bottleneck is too small, so no drop-off is occurring either. The encoder+dense network with 32 neurons performs significantly worse than the one with 64 neurons, which is shown in table 5.4. A summary of the results will be given in section 5.1.3.

5.1.2. Energy regression

In this section, the networks are used to reconstruct the energy of neutrinos. Since the first phase of training of a particular autoencoder – the unsupervised training – is independent of the task of the second phase, the autoencoders from the previous section can be reused for the following encoder+dense networks. However, the autoencoder from which the best encoder+dense network for the up-down classification was built from, might not be ideal for



(a) Model *model-64* (b) Model *model-32*

Figure 5.5.: Successive training of the two different models with 64 or 32 neurons in the bottleneck. The loss of the autoencoder is shown together with the up-down accuracy of the encoder+dense model. The spike of the autoencoder loss at epoch 31 of *model-64* is caused by an increase of the learning rate.

the energy reconstruction. Therefore, the process of finding the ideal autoencoder has to be repeated for the energy reconstruction.

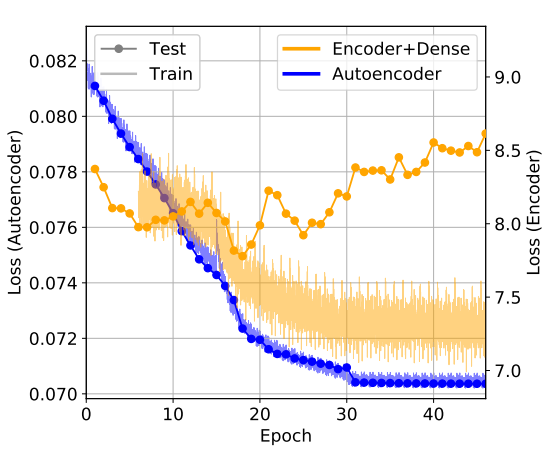
The encoder+dense networks have the same architecture as before, except that the final layer now has a single neuron with a linear activation instead of two with a softmax function. This neuron is supposed to output the energy of the neutrino, and the loss function of the training is the mean absolute error (MAE). Occasionally, the median relative error (MRE) is used for comparing the performances of the trained models:

$$\mathrm{MRE} = \left\langle \frac{|E_{true} - E_{reco}|}{E_{true}} \right\rangle_{\mathrm{median}} \tag{5.1.1}$$

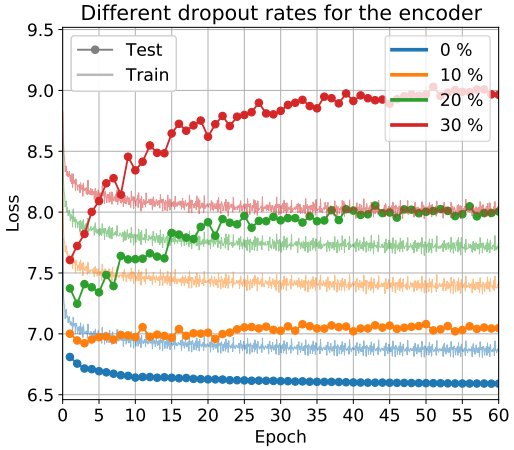
If not specified otherwise, the median is calculated over the whole test or train dataset, consisting of electron and muon neutrino events in a 1:1 ratio.

If a dropout of 20% is used for the dense layers as before, the networks showed a tendency to overfit during the successive training phase, especially for small bottleneck sizes (figure 5.6(a)). Note that the encoder+dense networks for energy regression are plotted with their loss, which they are supposed to minimize. In contrast, the up-down networks from the previous section were plotted with their accuracy, which is to be maximized. The change of the MAE due to new weights being loaded in is dominated with this dropout rate by the overfitting, making the identification of the optimal autoencoder epoch for phase two unreliable. A reduction of the dropout reduces the overfitting (figure 5.6(b)), even though dropout is usually intended to combat it.

The network does not overfit even with 0% dropout, so it is possible that most neurons in the dense layers are needed to predict the energy, preventing them from overfitting to the samples of the training set. On the contrary, an increase of the number of neurons in the first dense layer from 256 to 1000 or 4000 neurons did not increase the performance, neither did a reduction to 64 or 32 neurons. The following encoder+dense networks for the energy regression are trained without dropout.



(a) Successive training with 20% dropout. Strong overfitting occurs after about 10 epochs.



(b) Encoder+dense with different dropout rates. In this application, higher dropout leads to more overfitting and a worse performance.

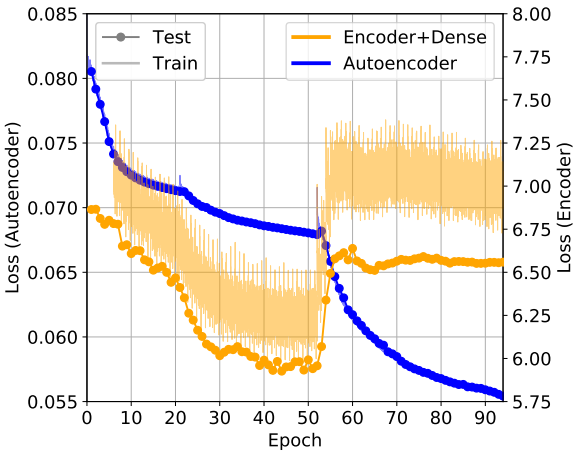
Figure 5.6.: Energy regression training with *model-32*.

The successive energy regression training of the models with the autoencoders from the previous section are shown in figure 5.7. In general, the behaviour of the models during the training of the energy regressions is similar to that of the up-down classification. The drop-off in the performance of the encoder+dense networks happen at roughly the same autoencoder epochs or losses as with the up-down classification. The fluctuations in the test loss from epoch to epoch are larger, as the output of the network is no longer a discrete classification, but a continuous value instead. All networks show an inferior loss compared to the one from the supervised approach. For the *model-32* in particular, the training loss shows a distinct periodicity from epoch to epoch, likely due to the batches in the dataset being in the same order for every iteration. This could be fixed by shuffling the samples prior to every epoch.

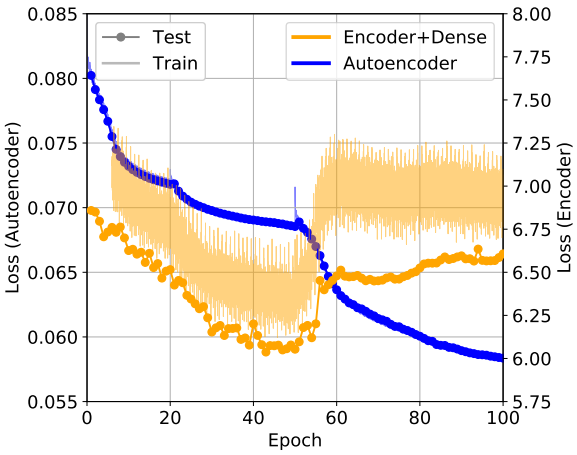
The peak performances of all models are shown in the summary table in section 5.6. The best network has 200 neurons in the bottleneck, as is the case for the up-down classification, too. However, for the energy reconstruction this model is not the *200-conv* network, but the *200-dense* one instead, with a MAE of 5.77 GeV and a MRE of 26.08% in the continued training.

In order to be able to compare the autoencoder procedure to the supervised approach, a network with the architecture of the *model-1920* encoder+dense is trained with no frozen layers and with a random initialization. It achieved a mean absolute error of 5.32 GeV and a median relative error of 23.70%.

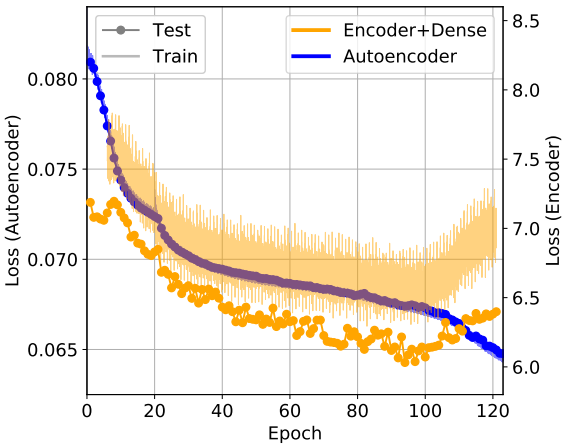
In 5.8(a), the median relative error of the model *200-dense* is compared to the supervised approach, separately for track- and shower-like events. The error increases for track-like events of higher energies, as they are less likely to be fully contained in the detector. It is more difficult to correctly predict their energy, since only a fraction of the emitted photons along the track are measured. Shower events are not spread out as much, so this effect is not visible for them as strongly. Figures 5.8(b) and 5.8(c) show the reconstruction in the form of histograms. For a theoretical ideal reconstruction, this would be a line through the origin with a gradient of one. As can be seen from these histograms, the energy of many events are underestimated. This is in part due to the fact that no cuts have been applied to the data, which are among others used to reject partially contained events. Therefore, many of those



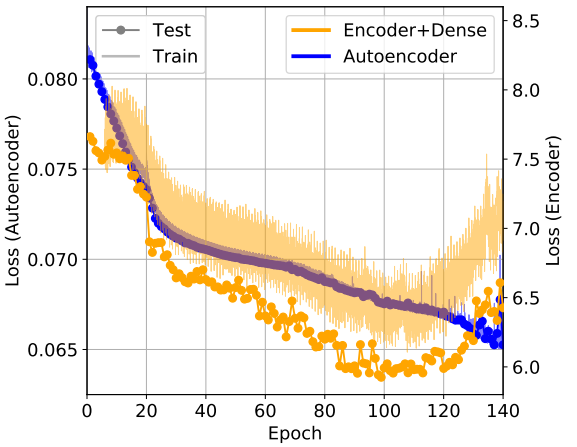
(a) Model *600-50 filters*.



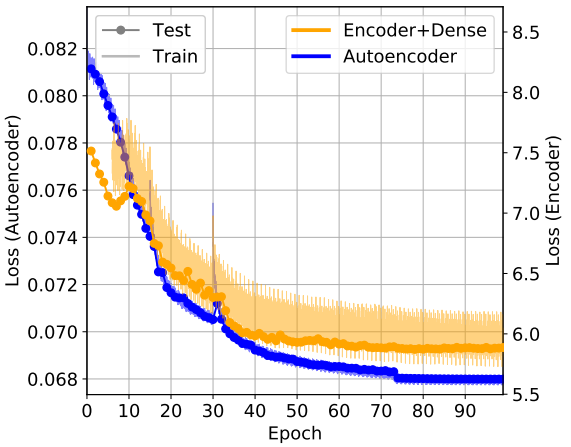
(b) Model *600-75 filters, 15 layers*.



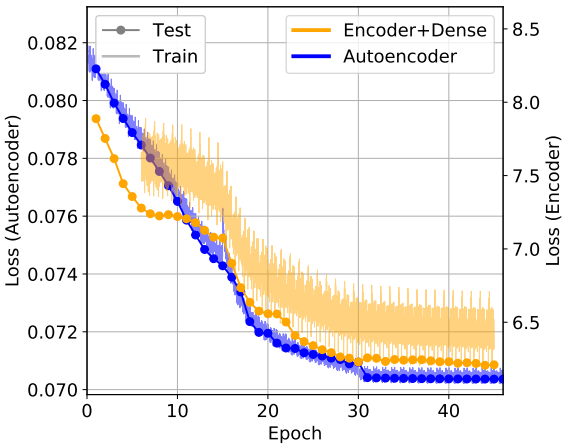
(c) Model *200-pool*.



(d) Model *200-dense*.



(e) *Model-64*.



(f) *Model-32*.

Figure 5.7.: Successive training of models with different numbers of neurons in the bottleneck. The loss of the autoencoder (MSE) and of the encoder+dense model (MAE) in GeV for the energy reconstruction is shown.

events are still present and are likely to have their energy underestimated, as they are located mostly outside of the detector.

The lowest rows in the histograms are mostly empty, i.e. the networks very rarely predict the energy to be between three and four GeV. This edge effect is likely caused by the composition of the training set, which only contains events between three and 100 GeV. The network can learn that it is never presented with energies outside of this range and use this fact to its advantage. The real spectrum of neutrinos is not cut off at these energies, so it will be necessary to test how the networks perform on a dataset with a wider energy range before using them in practice.

5.1.3. Summary of the bottleneck study

In table 5.6, the performance of the networks with different sizes of the bottleneck are shown. The results of the supervised approach are listed as well. These values are also plotted in figure 5.9. For both the up-down classification as well as the energy reconstruction, the models with 200 neurons in the bottleneck achieved the best performance. However, the architecture *200-pool*, which excelled in the up-down task, showed a much higher median relative error for the energy regression than the alternative *200-dense* design.

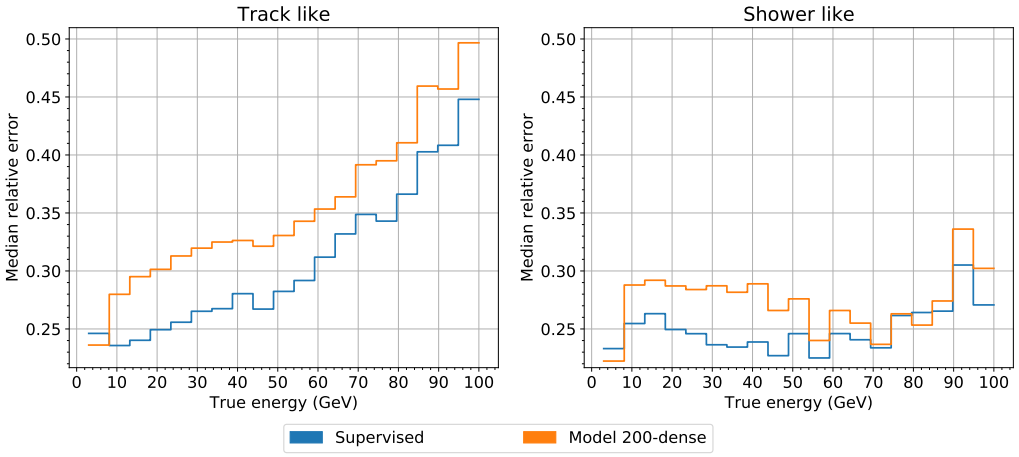
If the bottleneck is too large or small, the quality of the encoder+dense networks is reduced by several percentage points. It is therefore advisable to perform a scan over the number of neurons in the bottleneck for every new supervised task one is interested in. Since the autoencoders are trained independently of the second phase follow-up, the necessary training time is kept within reasonable limits. The typical training times of the autoenocoders in this chapter amounted to weeks, while the encoder+dense networks were often fully converged in less than a day.

Even the best encoder+dense networks were inferior in both tasks compared to the supervised approach, which is shown in figures 5.10 (up-down) and 5.8(a) (energy). For the up-down classification, this difference is 3.7% relative to supervised (88.16% vs. 84.98% accuracy) and for the energy reconstruction, it is 5.3% relative to supervised (24.70% vs. 26.08% median relative error). Based on the summary plot 5.9, it seems unlikely that this gap can be closed by a continued optimization of the size of the bottleneck alone. Different autoencoder architectures with the same bottleneck have shown to result in different encoder+dense performances though, so a variation of the design could be a promising method of further improving the networks. However, the same could probably be done to the supervised network as well, so whether or not the gap can be closed through means of redesigning the architecture remains unclear.

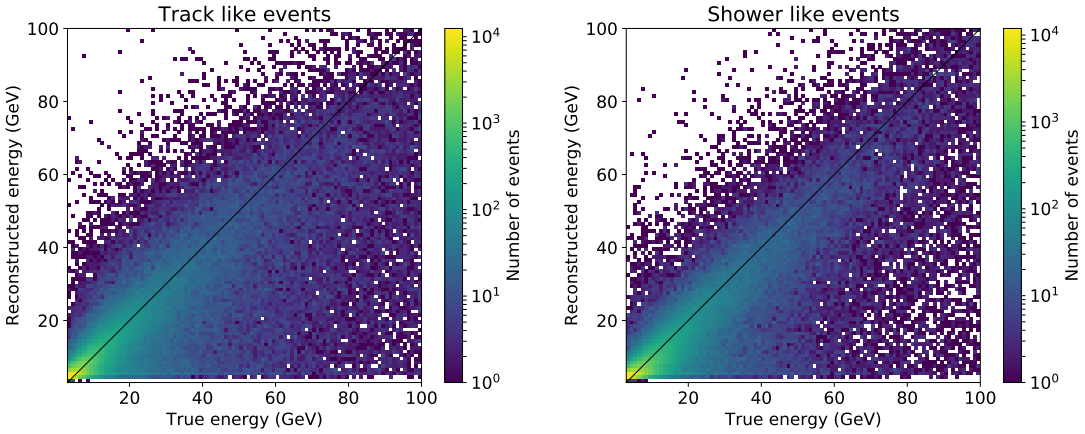
5.2. Number of free parameters

In the previous studies, the number of free parameters is almost constant for all models, so that advantages of different designs become apparent while keeping the expenditure of training time to a reasonable measure. For the practical application of the network, however, one is interested in the highest possible performance achievable. For that reason, four different models were designed to test the influence of the number of variables in the model on the performance of the encoder+dense network. These architectures are based on the *200-pool* model, which was found in the previous section to have the highest accuracy in the up-down classification task.

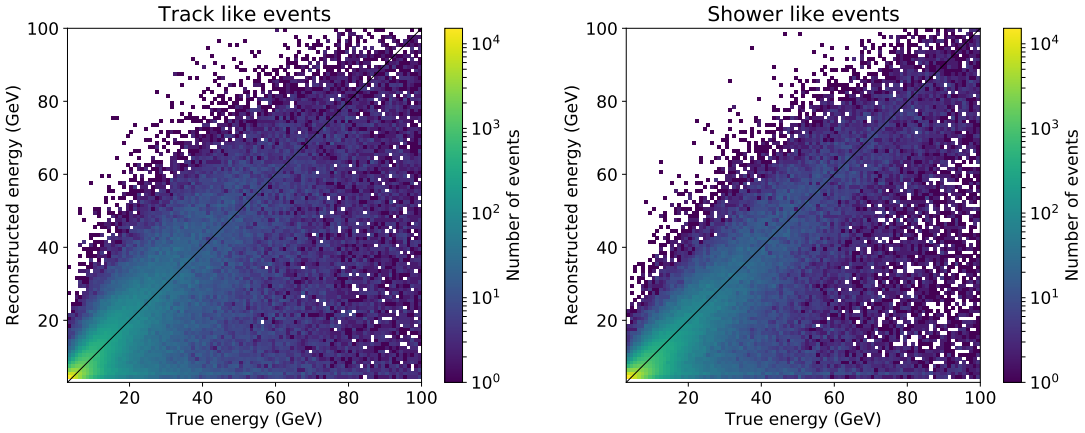
- *200-wide* (A.17): Increased number of filters.



(a) Encoder+dense



(b) Reconstruction of the supervised approach.



(c) Reconstruction of the *200-dense* encoder+dense network.

Figure 5.8.: Energy regression performance of the network trained with the supervised approach and of the *200-dense* encoder+dense network, which was best among all autoencoders. Figure (a) shows the median relative error over the true energy, separate for track- and shower-like events. Figures (b) and (c) show the energy as reconstructed by the networks over the true Monte Carlo energy.

Table 5.6.: Summary of the bottleneck parameter study. Up-down accuracy and mean relative error of the energy reconstruction are shown for the best encoder+dense network of the corresponding autoencoder model, in comparison to the result from the supervised approach. All models have a similar amount of free parameters.

Model	Neurons in bottleneck	Accuracy up-down	MRE energy
Supervised approach		88.16%	24.70%
model-1920	1920	82.51%	28.01%
600-50 filters	600	84.93%	26.50%
600-75 filter, 15 layers	600	84.38%	26.61%
200-pool	200	**84.98%**	26.92%
200-dense	200	84.32%	**26.08%**
model-64	64	84.71%	26.24%
model-32	32	82.67%	28.25%

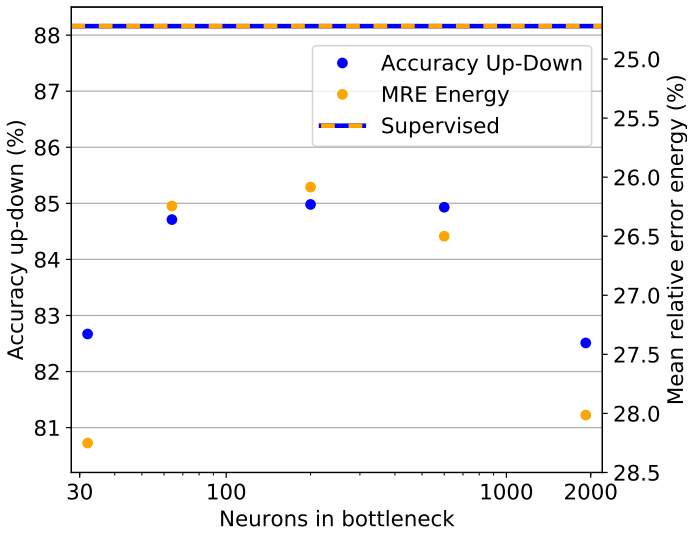


Figure 5.9.: Up-down accuracy and mean relative error of the energy reconstruction of models with different sizes of the bottleneck, compared to the results of the supervised approach with a fixed architecture. For each bottleneck size, the models with the best performance in the corresponding task were chosen. All models have a similar amount of free parameters.

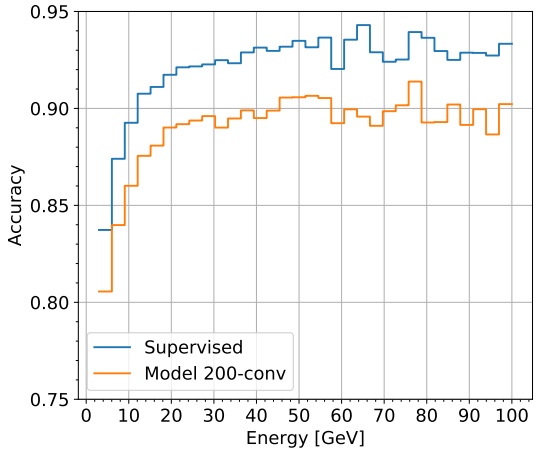


Figure 5.10.: Up-down accuracy of the *200-conv* model and the supervised approach over the neutrino energy.

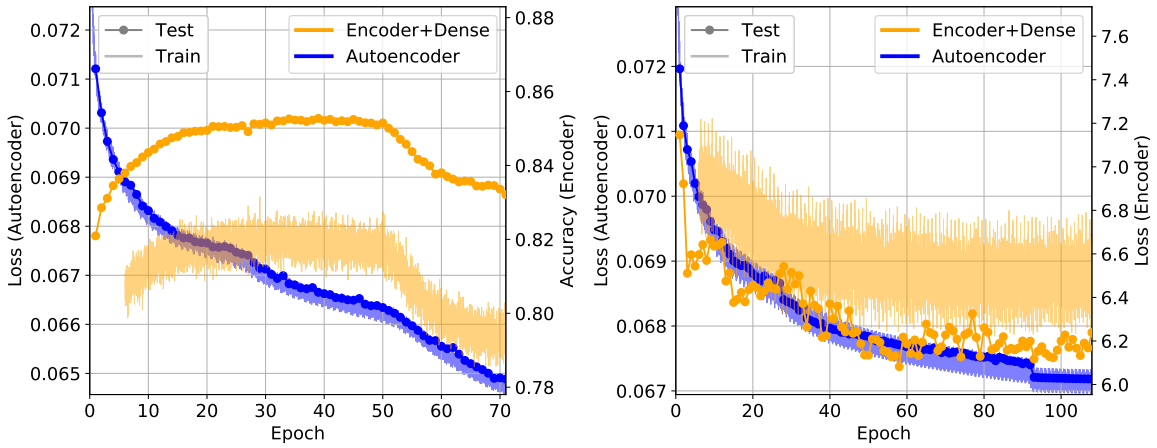
Table 5.7.: Influence of the number of free parameters on the performance. For every model, the number of free parameters and the average training time per epoch of the autoencoder are shown. The results of the best encoder+dense network for the up-down classification and the energy reconstruction are also listed.

Model	Free parameters	Minutes/Epoch	Acc. up-down	MRE energy
Supervised approach	860,673	42	88.16%	24.70%
200-wide	1,699,239	202	**85.46%**	27.08%
200-deep	1,334,779	173	85.22%	27.02%
200-pool	744,464	96	84.98%	26.92%
200-small	491,569	76	84.74%	**26.65%**
200-shallower	345,947	54	83.94%	28.36%

- *200-deep* (A.18): Increased number of convolutional layers.
- *200-small* (A.19): Reduced number of filters.
- *200-shallower* (A.20): Reduced number of convolutional layers.

All of these autoencoders have 200 neurons in their bottleneck. Once their training is finished, the encoder+dense networks are trained successively as before, one each for the up-down classification as well as for the energy reconstruction. All autoencoders are trained with a high learning rate of 0.1 for the first 15 epochs, which is then linearly increased to 0.3 in the next 15 epochs and kept constant after that. As with previous autoencoders, the high learning rates sped up the training process without causing instability or overfitting. At the end of the training, the learning rate was reduced again until the loss did not decrease further. The resulting performances are shown in table 5.7, together with the ones of the *200-pool* model from the previous section and of the model of the supervised approach.

Figure 5.11 shows the successive training of the two best architectures. Like before, the loss of the continuous energy regression is less stable than the categorical accuracy. At an autoencoder loss of about 0.066, the accuracy drops by several percentage points over the course of ten epochs. The *200-small* autoencoder has too few parameters to reach this loss even in the fully converged state after 105 epochs, so no drop-off occurs for the encoder+dense



(a) Model *200-wide*, up-down classification.

(b) Model *200-small*, energy regression.

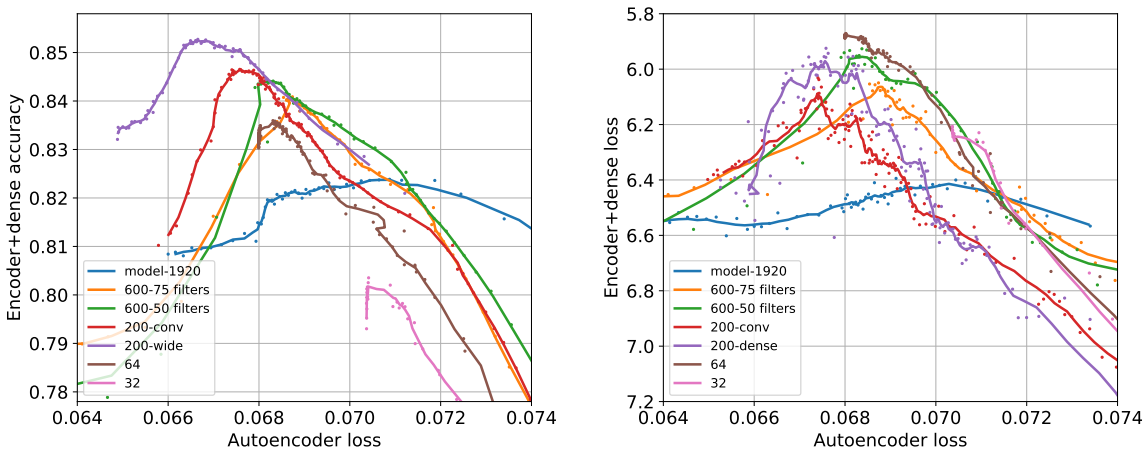
Figure 5.11.: Successive training of models with different numbers of neurons in the bottleneck. The loss of the autoencoder (MSE) is shown, together with either the accuracy (left) or the MAE in GeV of the encoder+dense model. The learning rate for the autoencoder of *200-small* was reduced by a factor of ten for the last few epochs.

network.

For the up-down classification task, the *200-wide* network with a higher number of filters improved upon the accuracy the original *200-pool* network by 0.48% (absolute). For that, the autoencoder required 2.28 times as many free parameters, and took 2.1 times longer to complete one epoch of training. In general, it can be observed that a higher number of free parameters in the given autoencoders lead to a higher accuracy for the encoder+dense network as well. This is despite the dense layers having the same number of free parameters for all the autoencoders.

For the energy regression in contrast, the median relative error is generally higher for autoencoders with more parameters. The *200-shallower* model is the exception to this, as it performs worse than all other networks. The best architecture from the five tested variants, the *200-small* model, has a lower MRE than the original *200-pool* model, while also having less parameters and a lower training time per autoencoder epoch. It still performs worse than the *200-dense* model from the previous section, which has a MRE of 26.08%. Due to time constraints, no autoencoders based on the dense design with additional parameters have been tested, but it is likely that the performance of the energy reconstruction task could also be improved slightly by this.

For the up-down classification task, it seems plausible that a further increase in the number of free parameters could improve the performance even further. If such an autoencoder was to be trained, it might be a good idea to test out first how high the learning rate can be set without destabilizing the training. This way, the elapsed time to train the autoencoder can be kept within reasonable limits.



(a) Up-down classification.

(b) Energy reconstruction.

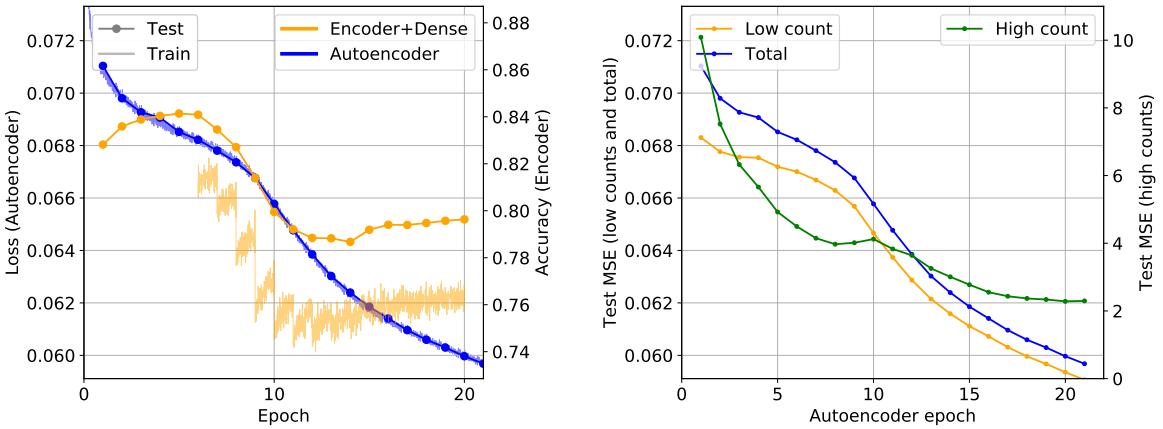
Figure 5.12.: The accuracy or the mean absolute error of the encoder+dense network over the mean squared error of the autoencoder from which the encoder was taken. Several different networks from this chapter are shown, including the best performing one for each task. The points are taken from the successive training of the encoder+dense networks. The solid lines are acquired by smoothing with a moving average over three (up-down) or five (energy) data points.

5.3. Investigating the encoder+dense performance drop-off

In many of the encoder+dense trainings in this chapter, a significant reduction of the accuracy or increase in the mean absolute error was noticeable when training with the encoder of an autoencoder with a low enough loss. For many networks, the critical mean squared error loss was found to be at roughly 0.068. As the autoencoder undercuts this value, encoder+dense networks performed worse - an unintuitive behaviour, as one might expect that a better encoder, which manages to encode more information of the original image in the bottleneck, would allow the dense layers to extract properties of the event to at least the same precision from its output. This drop-off might be an inherent property of the autoencoder training procedure. If it can be fixed, however, this could allow for a reduction of the gap in performance between the encoder+dense network and the supervised approach.

In figure 5.12, the drop-off is depicted for several networks of this chapter. For autoencoder losses higher than the critical value, the encoder+dense network and the autoencoder performance increase approximately linearly at the same time. In this range, an encoder that is better for the autoencoder is also better for the encoder+dense network. This behaviour inverts after the autoencoder goes below the critical loss. Some autoencoders are not able to reach the loss necessary for the reduction to occur. For those that do, the critical loss varies between 0.066 and 0.071 depending on the autoencoder model. There seems to be a tendency for the encoder+dense performance to be better the lower the critical loss is, especially for the up-down classification.

To be able to investigate this behaviour, the model *600-50 filters* from chapter 5.1.1 is trained again, but this time with a high learning rate of 0.1 from the beginning. This way, the critical loss is reached after only a few epochs (figure 5.13(a)). The accuracy of the up-down classification drops between epochs 6 and 14 from 84.09% to 78.67%. In figure 5.14, the



(a) Successive training.

(b) MSE constituents.

Figure 5.13.: On the left, the successive training of the *600-50 filters* model with a high learning rate of 0.1 is shown. On the right, the mean squared error on the test set of bins that contain three or more hits are plotted, as well as the MSE for bins with less than three hits. The total MSE is also shown, which is the same as in the left plot.

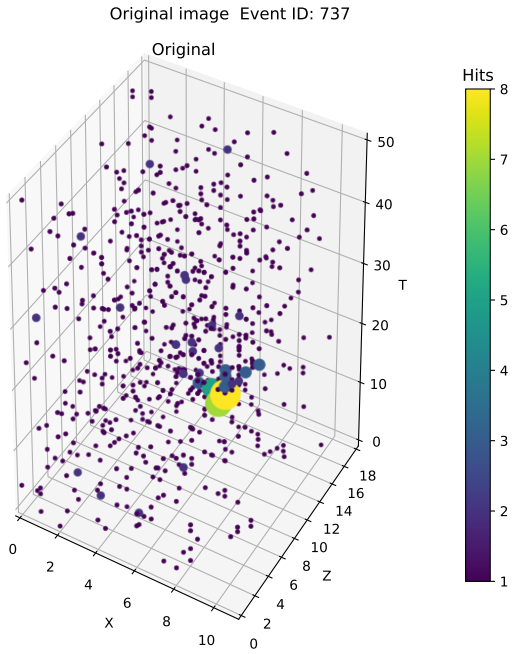
reconstructions of the autoencoders after 5, 9 and 13 epochs are shown for a random event from the test set. As the training progresses, the autoencoder starts to reconstruct more and more of the noise surrounding the event.

The network does this not only for the specific event shown, but across the whole dataset, as can be seen in figure 5.13(b). The information about which bins contain the event is not accessible anymore on the binned data. Instead of generating another set of simulations and storing this info as well, an approximation on the given samples is used. For this, the mean squared error is calculated separately for bins containing less than three hits. Since the background noise rarely amounts to more than two hits per bin, this quantity can be seen as an indicator of how well the autoencoder manages to reconstruct noise. The figure also shows the MSE for bins with three or more hits, which are approximately the bins containing the counts from the event.

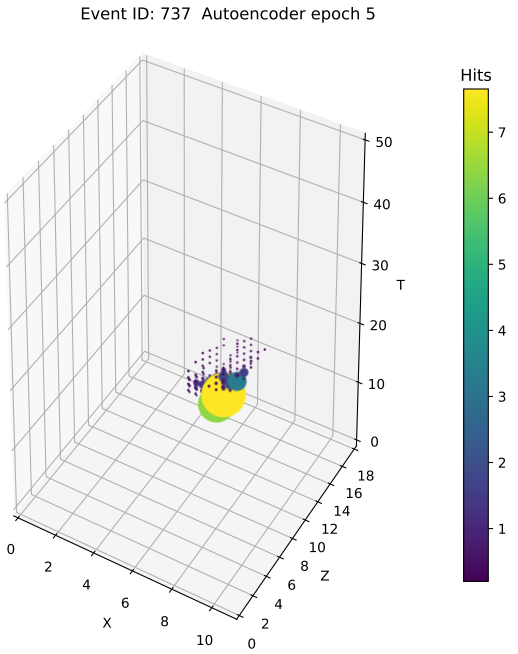
Before the critical loss is undercut, the MSE of the event bins falls more steeply than the one of the noisy bins. Afterwards, it is the other way around, as the autoencoder seems to focus more on the reconstruction of noise than of the event. It is possible that this behaviour is the cause of the drop-off, as detailed properties of the noise are neither helpful for reconstructing the up-down direction nor the energy of the neutrino. However, the MSE of event bins is still decreasing even below the critical loss, so it is unclear why this would lead to a worse encoder+dense performance.

A possible explanation is that the information in the bottleneck is stored in an increasingly compact way as the autoencoder learns to reconstruct noise as well. This could make it more difficult for the small dense network to extract the desired properties from the encoded representation of the image - after all, it takes the autoencoder a large deconvolutional network to reconstruct the image from this info. This hypotheses is tested by modifying the dense network of the encoder+dense model in various ways:

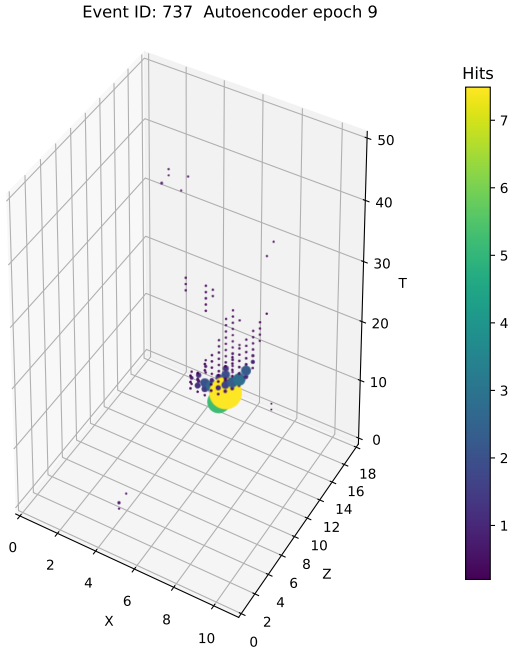
- Standard dense network: three dense layer with 256, 16 and 2 neurons. This is the



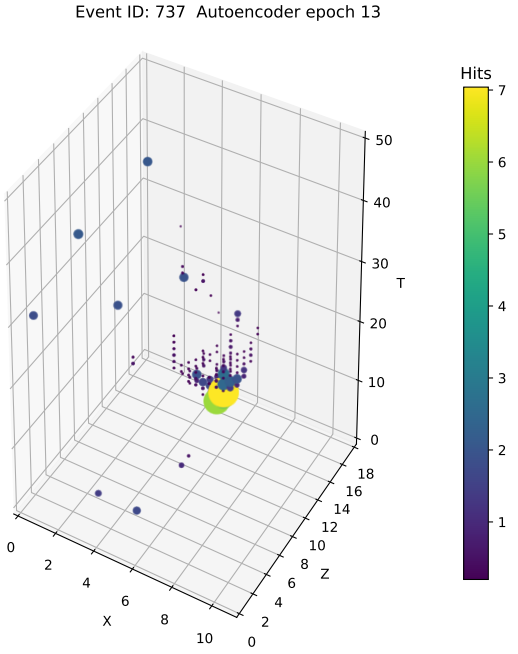
(a) Original image



(b) Epoch 5, before the drop-off.



(c) Epoch 9, during the drop-off.



(d) Epoch 13, after the drop-off.

Figure 5.14.: Event reconstructions of the *600-50 filters* autoencoder trained with a high learning rate, after different numbers of epochs. Bins with less than 0.3 counts are not displayed.

default layout used for all previous networks. Highest accuracy for the *600-50 filters* with a high learning rate: 84.83%.

- Additional convolution: A convolutional layer was added before the three dense layers. Highest accuracy: 84.83%.
- Additional dense layer: The network has four dense layers in total with 256, 256, 16 and 2 neurons. Highest accuracy: 84.16%.
- Removed dense layer: Smaller network with only two dense layers with 256 and 2 neurons. Highest accuracy: 83.86%.

None of the variants above managed to achieve a higher performance than the default dense network. The drop-off still occurred for all of them, so an undersized dense network seems to not be the cause of it.

Another idea to counteract the drop-off is to prevent the autoencoder from focussing too much on the noise bins by modifying the cost function. So far, the mean squared error was used, for which the squared difference between every bin of the original image and the reconstruction is calculated and then averaged. Since most bins contain only background noise, this gives the autoencoder a strong incentive to focus on them - possibly a suboptimal behaviour, as detailed information about noise is usually not relevant for the encoder+dense phase. To combat this, one can weight the contribution of every bin to the overall loss according to the likeliness of it containing noise.

The background in the given datasets is Poisson distributed. The probability of a bin containing k hits according to a Poisson distribution with an expectation value λ is given by:

$$P_{poisson}(k) = \frac{\lambda^k}{k!}\,\mathrm{e}^{-\lambda}. \tag{5.3.1}$$

The expectation value λ can be approximated by calculating the mean number of counts per bin in the dataset. This slightly overestimates the real expectation value as it contains the hits of the event as well, but could in practice also be used on measured data. With this, the mean squared error Poisson (MSEP) can be introduced, in which the contribution to the loss in every bin is weighted with the probability of it not being Poisson distributed:

$$C_{MSEP}(\boldsymbol{y}^{true}, \boldsymbol{y}^{pred}) = \frac{1}{n}\sum_{i=1}^{n}\left(1 - P_{poisson}(y_i^{true})\right)\cdot\left(y_i^{true} - y_i^{pred}\right)^2. \tag{5.3.2}$$

With this cost function, bins of original images which are likely to contain Poisson noise have a lower influence on the total loss. In contrast, bins that are unlikely to contain Poisson noise are more important. The factor $\left(1 - P_{poisson}(y_i^{true})\right)$ can be seen as a sensitivity mask with the same dimension as the input. It is calculated for every input sample and contains higher values for the bins of the event than for the others. The squared difference of every bin is multiplied with the corresponding entry of this mask before taking the mean. Unlike the second factor in the sum, this mask does not depend on the output of the autoencoder y_i^{pred}, and can therefore not be influenced by the network. In the top row of figure 5.15, an arbitrary event is shown in a 2-D plot, together with its MSEP sensitivity mask. Bright areas in the mask identify bins which more likely belong to the event, and are therefore more important for the reconstruction of the autoencoder.

The figure also shows the masks of two other modified loss functions, the squared-MSEP, and the log-MSEP. For the first one, the factor $\left(1 - P_{poisson}(y_i^{true})\right)$ in equation 5.3.2 is simply replaced by its square, so that noisy bins have even less of an impact on the overall loss. The most extreme weighting is the log-MSEP, in which the factor is given by $-\log(P_{poisson})/100$.

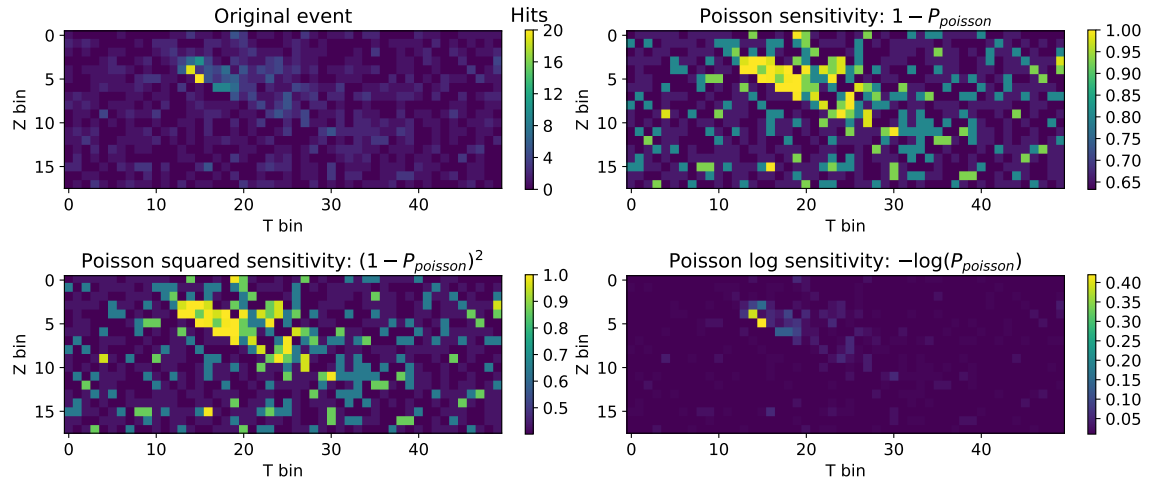


Figure 5.15.: Top left: One event depicted in a two-dimensional z-t-histogram by summing over the x-, y- and channel id dimensions. The event is an up-going anti-electron CC event with an energy of 44 GeV. The other subplots show three different sensitivity masks calculated for that event.

For each of these three modified cost functions, a new instance of the *600-50 filters* model is trained. The autoencoder with the log-MSEP loss failed to converge, as some bins in the dataset are found to contain a very high amount of hits, some even more than 30. This leads to an enormous loss, as the logarithm becomes very large in this case.

The successive training of the model with the MSEP cost function is shown in figure 5.16. The drop-off is still occurring, this time after around 20 epochs. At that point the loss of low count bins is also starting to decrease more quickly, as was the case with the standard MSE cost function. The accuracy of the encoder+dense network is lower for this model compared to the using the standard MSE function.

Figure 5.17 shows the training of the model with the log-MSE cost function. There is a drop-off visible in the first few epochs of the encoder+dense network, and a steady climb afterwards. Previous networks did not show such an increase, even after long training times. However, the accuracy of this model is much lower than it was for the one with MSE, even though it has a lower loss on high count bins.

None of the modified cost functions managed to resolve the drop-off, nor did they lead to an increase in accuracy of the encoder+dense network. This is despite letting the autoencoder achieve a more exact reconstruction of high count bins. It remains unclear whether the presence of a critical loss is an intrinsic property of the autoencoder training procedure, or whether it can be fixed by an adequate modification of the cost function.

In this regard, the generative adversarial network archetype (GAN) is an interesting approach. For these networks, the quality of the reconstruction of the autoencoder would not assessed by a user-defined cost function, but by a separate network instead. The two parts of the GAN-based network – the autoencoder and the discriminator – are trained to outperform each other: The autoencoder tries to reconstruct its input in such a way that the discriminator cannot distinguish it from the original. In contrast, the discriminator tries to develop a way of differentiating them as reliably as possible. This could prevent the autoencoder from focussing too much on bins containing noise: As long as their counts are Poisson distributed and the expectation value of the distribution is the same in both the original and the reconstruction, the discriminator will not be able to make use of the noise for its decision. Therefore, it

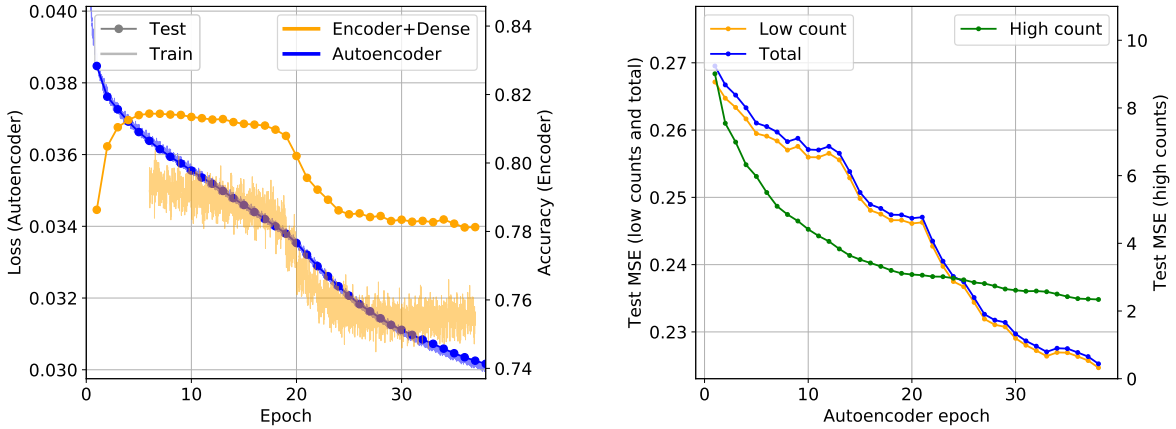


Figure 5.16.: Successive training (left) and mean squared error for bins with three or more, two or less and any number of hits (right). The architecture of *600-50 filters* was trained with the MSEP loss and a learning rate of 0.01.

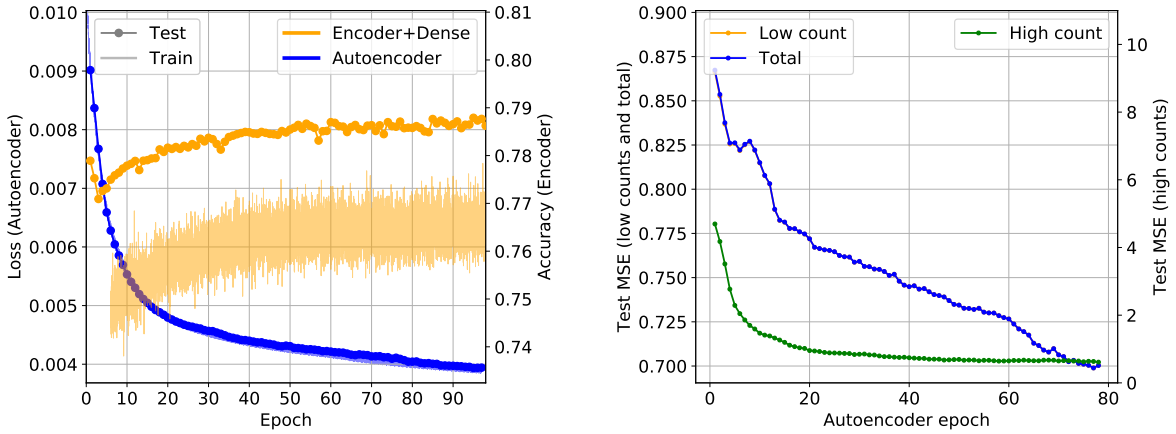


Figure 5.17.: Successive training (left) and mean squared error for bins with three or more, two or less and any number of hits (right). The architecture of *600-50 filters* was trained with the MSEP^2 loss and a learning rate of 0.1. The orange line is concealed by the blue line.

is plausible that the autoencoder might only learn these two properties of the noise, and otherwise exclusively improve the event reconstruction. Due to time constrains, however, the design and training of such a network is not a part of this thesis.

Before this is implemented, it might be advisable to check whether the drop-off also occurs if the networks are trained on a set of simulations in which no noise is present. If the noise is the cause for the reduction, it should not be appearing anymore then. In this form, it could not be used in practice on measured data, though, as it is unknown which bins contain the event in this case.

5.4. Conclusion and discussion

The size of the bottleneck of the autoencoder has a strong impact not only on the quality of its reconstructions, but also on the performance of the encoder+dense network built from it. For both the up-down classification as well as the energy reconstruction, a bottleneck between 64 and 600 neurons seems to be ideal for samples from the xzt-dataset with 9900 bins. This corresponds to a compression to between 0.6% and 6.1% of the original size. However, it is possible that this fraction is not applicable for other datasets, since the the neurons in the bottleneck are expected to contain a compressed representation of the features in the input. If another dataset has a different density of information in the bins of its samples, the encoder might require more or less bins in the bottleneck, so the ideal compression rate would change as well.

The changes to the size of the bottleneck were conducted by modifying the architecture of the networks. For example, some models have additional convolutional or pooling layers to reduce the dimension further. At the same time, the design of autoencoders with the same bottleneck had a significant influence on the encodcer+dense performance as well. It is therefore not possible to attribute a different performance exclusively to the sizes of the bottlenecks. The impact of the design was distinctly smaller than the one from the bottleneck, though.

A possible way of testing almost identical designs could be to use the model *200-dense*, which features a pair of dense layers in the middle. The size of its bottleneck can be easily adjusted without changing the architecture. Dense layers do, however, contain a very large number of free parameters compared to their amount of neurons, so this technique is ideally used only for small bottlenecks.

The best performing networks for both tasks have 200 neurons in the bottleneck, albeit they have a different design. For the up-down classification, an increase in the number of free parameters leads to an improved performance, while the opposite is true for the energy reconstruction. This goes to show that different autoencoder models are ideal for different reconstruction tasks. Since the time-consuming autoencoder trainings are independent of this task, they have to be done only once for a given input dataset, and the resulting autoencoders can be used as the basis of any encoder+dense network.

Even the encoder+dense networks with the best bottleneck size are several percent worse on the simulations than the ones trained according to the supervised approach in both the up-down classification as well as the energy reconstruction. While it is plausible that an optimization of the autoencoder architecture would improve the performance, the same is true for the supervised network, which was not altered in any way in this chapter. This might therefore not reduce the gap between the supervised and unsupervised approaches.

A possible explanation for the inferior performance of the encoder+dense networks is the fact that the autoencoder attempts to reconstruct the background noise of its input in order to lower its mean squared error loss. This could be the reason for the observed drop-off in the

performances of the encoder+dense networks once the autoencoder reaches a certain loss. By using a generative adversarial autoencoder, it might be possible to circumvent this problem in the future.

6. Autoencoder robustness

The big advantage of the encoder+dense networks is that large parts of them can be trained in an unsupervised fashion on measured data – unlike the networks of the supervised approach, which have to be trained on simulations. This might make the former more robust against differences between the simulations and the measured data.

Deep neural networks have the potential to make use of any and all features present in a dataset due to their often very large number of free parameters. If a supervised network is trained on simulations containing features which are systematically different or not present on real data, it might learn to rely on them in order to solve its reconstruction task. This could consequently lead to a reduction of performance once the network is used on measured data where these features are not present. To make things worse, this might even go unnoticed, as an assessment of the performance on measured data is often difficult.

Autoencoders in contrast can be trained directly on the measured data, so they will only adapt to features that are present there. The small dense network in the encoder+dense phase, however, has to be trained on simulations, so unwanted features may still play a role for the prediction. In this chapter, the robustness of the autoencoder training procedure will be investigated and compared to the supervised approach for several systematic differences for both the up-down classification, as well as the energy regression.

In general, it is important to know the change of the performance when switching from simulations to measured data. Since the ORCA detector is still under construction, no actual measured data can be used. But even if measured data was available, the lack of labels there would still not allow for an exact quantification of the performance.

Instead, **both datasets** in this chapter – the "simulation" dataset, as well as the "measured data" dataset – are generated by simulations.

This way, the performance of the networks on either one can be easily assessed. The potential differences between the two datasets are implemented by artificially manipulating one of them: For example, an additional feature might be added to the "simulation" dataset, but not to the "measured data" dataset, to investigate how much of an impact this feature has on the encoder+dense network and the one from the supervised approach.

The manipulations are kept simple, so that they can be easily added to the existing data. Not all of them are designed to be highly realistic or strongly physically motivated by the currently known circumstances of the detector. Instead, the following tests should be seen as spot tests of the networks' robustness, which show the influence of several specific cases on the performance. In practice, the cause and nature of the differences are often unknown, so testing with a variety of manipulations can give a general understanding of how the networks might behave in the real world scenario.

6.1. Up-down classification

As in the previous chapters, the networks in this section are classifying neutrinos as either up- or down-going depending on the direction they travelled through the detector. In practice, networks of the supervised approach would be trained and optimized on simulations, and then applied to measured data. Based on this, their robustness is tested according to the

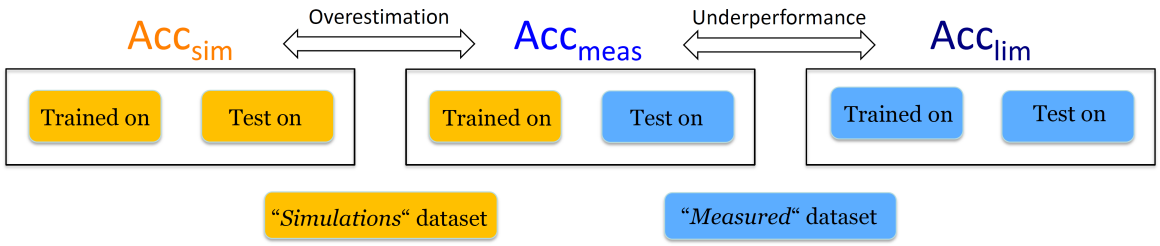


Figure 6.1.: Scheme of how to determine the overestimation and underperformance of a network. It shows on which datasets the supervised or encoder+dense network is trained and tested on, together with the resulting accuracy $\mathrm{Acc}_{\mathrm{sim}}$, $\mathrm{Acc}_{\mathrm{meas}}$ or $\mathrm{Acc}_{\mathrm{lim}}$. The autoencoder is always trained on the "measured" dataset.

strategy shown in figure 6.1:

First, the supervised network is trained and tested on the "simulation" dataset, until it is in its best-performing state ($\mathrm{Acc}_{\mathrm{sim}}$). Then, the same network is tested on the "measured data" dataset ($\mathrm{Acc}_{\mathrm{meas}}$). The potential reduction of accuracy between these two evaluations represents the degree to which the performance of the network would be **overestimated** in practice. This is because no labels exist for the actual measured data, so the true accuracy of the network on it is unknown. If it is assumed that the performance on "measured" data is the same as on "simulated" data, this leads to the described overestimation. It is calculated as the relative reduction of accuracy in the datasets:

$$\text{Overestimation} = \frac{\mathrm{Acc}_{\mathrm{sim}} - \mathrm{Acc}_{\mathrm{meas}}}{\mathrm{Acc}_{\mathrm{meas}}} \tag{6.1.1}$$

Additionally, an instance of the supervised network is trained and tested on the "measured data" dataset. This gives the best performance the network could achieve if the simulations would be perfectly identical to the measured data ($\mathrm{Acc}_{\mathrm{lim}}$). Accordingly, the difference between this accuracy and the accuracy from the network that was trained on the "simulation" set and evaluated on the "measured data" dataset $\mathrm{Acc}_{\mathrm{meas}}$ describes the **underperformance** of the network. This is how much accuracy is lost due to suboptimal simulations. It is calculated as a fraction relative to $\mathrm{Acc}_{\mathrm{meas}}$:

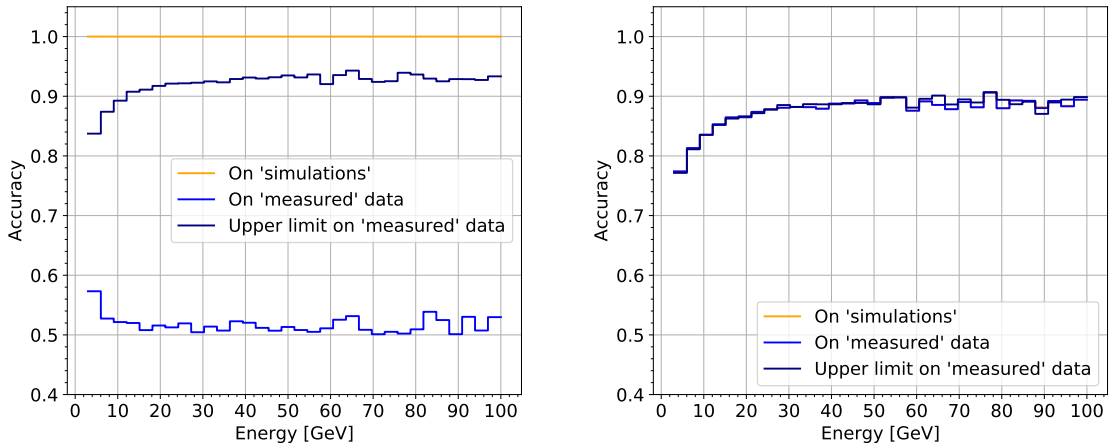
$$\text{Underperformance} = \frac{\mathrm{Acc}_{\mathrm{lim}} - \mathrm{Acc}_{\mathrm{meas}}}{\mathrm{Acc}_{\mathrm{meas}}} \tag{6.1.2}$$

These two quantities – the overestimation and the underperformance – are also determined for the unsupervised approach. As in practice, the autoencoder is always trained on the "measured data" dataset first for this. Then, the encoder+dense network is built from it and the same three evaluations as for the supervised approach are performed:

Training and testing the dense layers on the "simulation" dataset to highest performance, training on "simulations" and testing on "measured data", as well as training and testing on "measured data". From this, the two robustness indicators can be calculated. The closer they are to zero, the more robust the network is against a given manipulation. The successive training method introduced in section 3.4.2 will be used to find the best autoencoder epoch for the encoder+dense network on the "simulation" dataset.

6.1.1. Proof of concept: Defect bin

For a first assessment of the autoencoder robustness, a highly correlated feature is added to the "simulation" dataset: One specific bin of each event contains the information about



(a) Supervised approach.

(b) *model-1920* encoder+dense network.

Figure 6.2.: Up-down accuracy on the test set over the neutrino energy. "On simulations" is the performance of a network that is trained and tested on the "simulation" dataset ($\mathrm{Acc}_{\mathrm{sim}}$). "On measured data" is for a network that is trained on "simulations" and tested on "measured data" ($\mathrm{Acc}_{\mathrm{meas}}$). "Upper limit" is for a network that is trained and tested on the "measured data" dataset ($\mathrm{Acc}_{\mathrm{lim}}$).

whether the particle was up- or down-going. This bin is always found at the coordinates $(x, z, t) = (0, 0, 0)$, and has exactly one count if the event is up-going, and no counts for those that are down-going. This could resemble an unintentional artefact in the simulations. In practice, such artefacts are expected to be – if at all present – less drastic, so this case should be seen as a proof of concept of the autoencoder training procedure rather than a realistic case.

The accuracies from the three evaluations described above are shown in figure 6.2. The supervised network learns to reproduce the content of the manipulated bin, since this allows it to always predict correctly on the "simulations" dataset (orange line). When this network is moved over to the "measured data", the performance is equivalent to guessing since it still only reproduces the content of that one bin, even though it does not contain the up-down information anymore (light blue line). The large reduction in performance results in a high overestimation of 88.7%. The network is also far below the highest potential accuracy on the "measured" dataset (dark blue line), which is reflected in an underperformance of 66.4%.

In contrast, the encoder+dense network performs virtually the same, independent on the dataset it was trained or tested on. This is because the dense layers do not have access to the information in the defect bin: The encoder had no incentive to include this bin in the bottleneck, as it was trained on the "measured data" dataset, where the bin was not manipulated. The overestimation and underperformance consequently amount to only 0.0% and 0.1% for the encoder+dense network.

6.1.2. Additional uncorrelated noise

The background noise in the ORCA simulations is generated by assuming a frequency of 10 kHz of noise per photomultiplier. For xzt-data, this results in a Poisson distribution with an expectation value of about 0.1 counts of noise per bin. It is possible, however, that there is more or less than that on the measured data. This is tested in this section by adding an

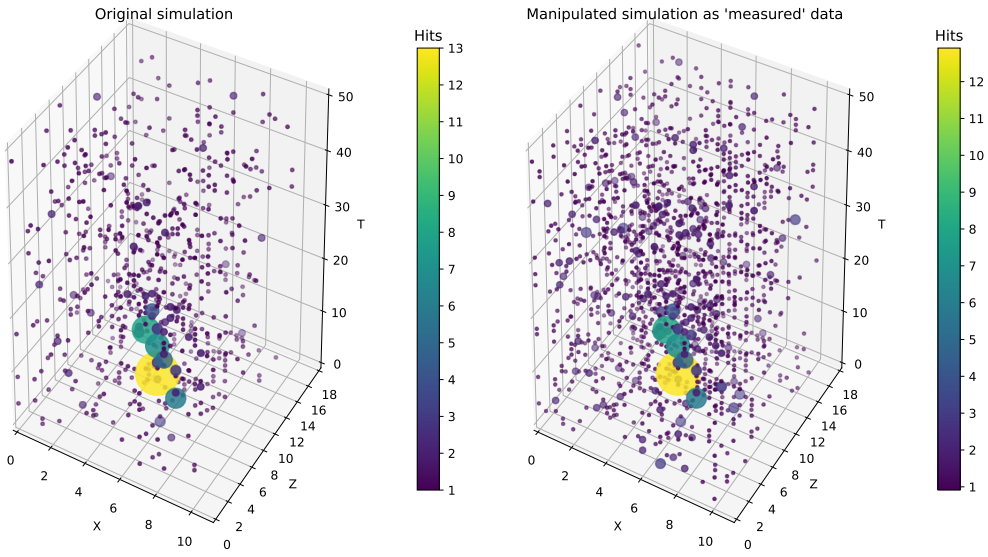


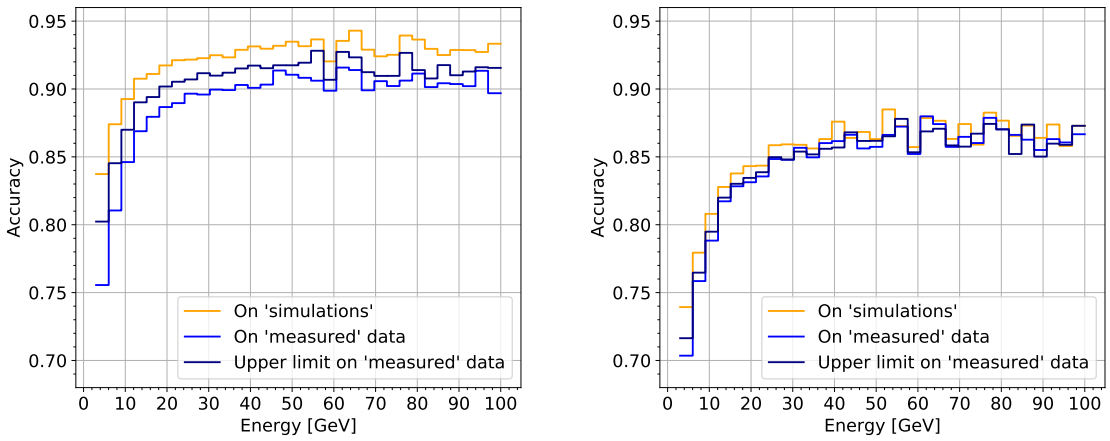
Figure 6.3.: An example for the datasets of section 6.1.2. The left histogram shows an event from the "simulation" dataset, and the right one shows the same event from the "measured data", to which additional Poisson noise has been added. This event is a down-going electron neutrino interaction with an energy of 75.1 GeV.

additional 10 kHz of noise to the "measured data" dataset (figure 6.3). It should be noted that the true background noise rate can be measured in practice, so the simulations could be adjusted to it.

In contrast to the manipulation of the previous section, this is completely uncorrelated to the up-down direction the network is supposed to predict. This can be thought of as a particularly adverse case for the autoencoder, as the features which the supervised network learns to extract from the "simulation" set are present in the "measured data" set as well. The only advantage the encoder+dense network has over the supervised approach is that it is used to the higher average amount of noise.

As before, the robustness indicators are calculated by performing three evaluations of the networks on the different datasets, which are shown in figure 6.4. The upper limit on the "measured data" is lower than the accuracy on the "simulations", as the additional noise makes it harder for the networks to identify the direction the particles are travelling in. The overestimation and underperformance amount to 6.7% and 3.5% for the supervised approach, and 2.8% and 1.0% for the encoder+dense network. Even though the robustness of the encoder+dense network is better, the overall accuracy that would be achieved in practice is still higher for the supervised approach: The encoder+dense net achieved an accuracy on the measured set of $\mathrm{Acc}_{\mathrm{meas}} = 77.55\%$, while the supervised approach has $\mathrm{Acc}_{\mathrm{meas}} = 82.68\%$.

This is a prime example for why it is difficult to judge whether the autoencoder training procedure can be reasonably used in practice: If the difference between simulations and measured data has little correlation to the task and therefore a small enough impact on the performance, the encoder+dense network can be outperformed on measured data despite its higher robustness. This is possible because the autoencoder training procedure results in networks with a lower overall accuracy than those from the supervised approach, as was shown in chapter 5. Despite that, it is still beneficial if the network is robust, as it allows for a better assessment of the performance which the network has on measured data.



(a) Supervised approach.

(b) *model-1920* encoder+dense network.

Figure 6.4.: Up-down accuracy on the test set over the neutrino energy for the three different evaluations. Additional noise was added to the "measured data" dataset for these networks.

6.1.3. Brighter down-going events

In this section, the "simulation" dataset is manipulated by increasing the brightness of down-going events. Bins with three or more counts, which are likely to be containing the counts from the event, get an additional two counts added to them on average if the event is down-going. The exact number of counts is randomly calculated for every bin according to a binomial distribution with $n = 3$ and $p = 2/3$. This is shown in figure 6.5.

In practice, a dependence of the brightness of an event on the angle of incidence could be a consequence of a reduced efficiency of the individual photomultipliers. For example, biological structures might form on top of DOMs, so that the photomultipliers which are facing upwards measure less photons. In the datasets of this section, down-going events are brighter in the "simulations" than in the "measured data", so this would compare to the case where the reduction of the efficiency is underestimated or not accounted for in the simulations.

The three different evaluations are shown in figure 6.6. The networks make use of the correlation, as the performance on the manipulated "simulation" set is much higher than in the other two cases. The network trained according to the supervised approach shows a severe reduction when moving to "measured data". The accuracy does also not increase towards high energies, suggesting that the network does not make full use of the higher count number of highly energetic events. Instead, it heavily relies on the additional counts, which are more frequent at higher energies as more bins are expected to have three or more counts.

The encoder+dense network also experiences a drop in accuracy. While it is not as strong as with the supervised approach, this still goes to show that the dense layers can become sensitive to correlations during their training on "simulations". In the given case, it may not be surprising that the encoder extracts the number of counts of the event in some way, as variations in the brightness of events are seen in the "measured data" as well. Through this, the correlation between the count number and the direction appears to still be visible to the dense network, as it uses this information for its classification.

The encoder+dense network has shown to be much more robust for this manipulation. The overestimation and underperformance are 39.7% and 28.6% for the supervised approach, and

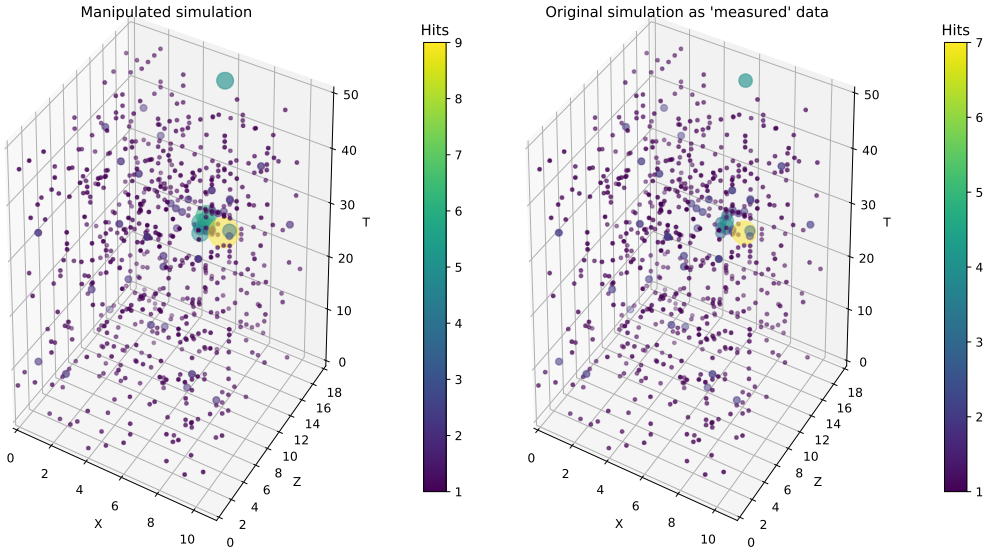
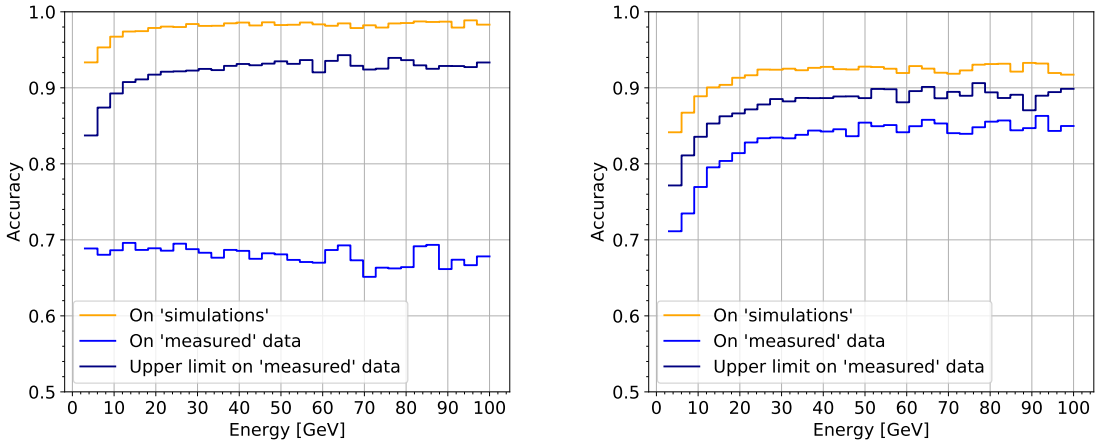


Figure 6.5.: An example for the datasets of section 6.1.3. The left histogram shows an event from the "simulation" dataset, and the right one shows the same event from the "measured data". Additional counts are added to the bins containing down-going events in the "simulations". This event is a down-going anti-electron neutrino with an energy of 23.7 GeV.



(a) Supervised approach.

(b) *model-1920* encoder+dense network.

Figure 6.6.: Up-down accuracy on the test set over the neutrino energy for the three different evaluations. Down-going events have been made brighter in the "simulation" dataset.

Table 6.1.: Comparison of the robustness of different model architectures. $\mathrm{Acc}_{\mathrm{sim}}$ is the accuracy of a model that was trained and tested on the manipulated "simulations". $\mathrm{Acc}_{\mathrm{meas}}$ is the mean accuracy on the "measured data" test dataset of a model that was trained on the manipulated "simulation" dataset in which down-going events are brighter. The overestimation and underperformance are calculated as shown in equation 6.1.1 and 6.1.2; The closer to zero they are, the more robust the network is.

Model	$\mathrm{Acc}_{\mathrm{sim}}$	$\mathrm{Acc}_{\mathrm{meas}}$	Overestimation	Underperformance
Supervised approach	95.80%	68.62%	39.7%	28.6%
model-1920	88.00%	76.51%	15.0%	7.8%
600-50 filters	**89.80%**	79.02%	13.7%	7.5%
200-conv	88.64%	**79.63%**	**11.3%**	6.7%
model-64	88.84%	**79.63%**	11.6%	**6.4%**
model-32	85.96%	75.86%	13.3%	9.0%
200-wide	**88.64%**	**81.71%**	**8.5%**	**4.6%**
200-small	88.15%	78.64%	12.1%	7.8%

15.0% and 7.8% for the encoder+dense network. This example shows that the unsupervised approach can significantly remedy potential issues with systematic discrepancies between datasets.

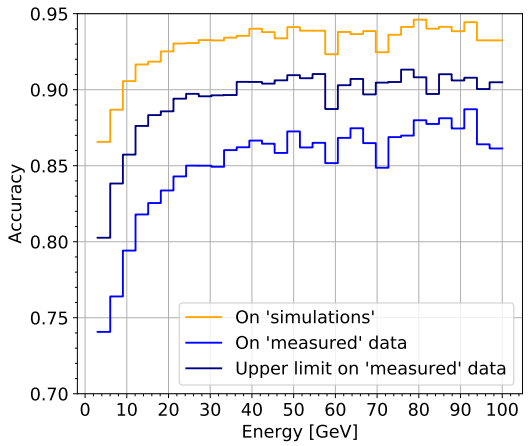
6.1.4. Dependency on the bottleneck size

The size of the bottleneck of the autoencoder does not only have an impact on the performance of the encoder+dense network as shown in section 5.1, but also on its robustness. To investigate this behaviour, the study from the previous section 6.1.3, in which down-going events are brighter in the "simulation" set, is repeated for autoencoders with differently sized bottlenecks. The best-performing architecture for each bottleneck with a comparable number of parameters was used for this. The resulting evaluations for size 600, 200, 64 and 32 are shown in figure 6.7, and summarized in the middle section of table 6.1.

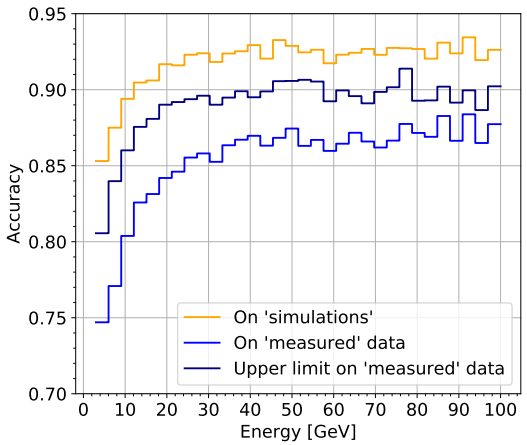
Similar to the change in the accuracy observed in the bottleneck study of chapter 5.1, the robustness has an optimum depending on the bottleneck. Both the networks with 200 and with 64 neurons in the bottleneck have a low overestimation and underperformance. They achieved the same accuracy on the measured dataset $\mathrm{Acc}_{\mathrm{meas}}$. The robustness indicators of all the tested encoder+dense models are also shown graphically in figure 6.8.

The table lists $\mathrm{Acc}_{\mathrm{sim}}$ as well, which is the only of the four quantities from the chart that is also obtainable in the practical application where no labels are available for the measured data. For this, the autoencoder would be trained on the measured data and the encoder+dense network trained and evaluated on the simulations. This could be done without artificially manipulating the datasets. In this case, if one were to simply pick the encoder+dense network that performs best on the "simulations", it would be the model *600-50 filters*. However, this architecture is performing worse on the "measured" set than the ones with 200 or 64 neurons in the bottleneck. This goes to show that robustness scans, like the ones in this section, might be necessary in order to find the ideal bottleneck size in practice when real measured data is used to train the autoencoders.

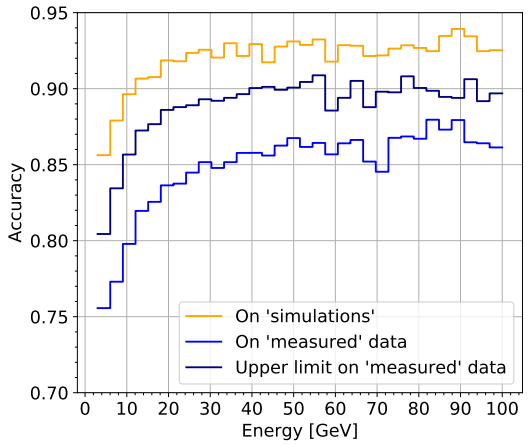
Table 6.1 also shows the results for some of the networks with a higher number of free parameters and a bottleneck of size 200. The larger model *200-wide* with a higher $\mathrm{Acc}_{\mathrm{lim}}$ is also much more robust. In contrast, the *200-small* network is worse in both regards than the medium-sized *200-conv* model.



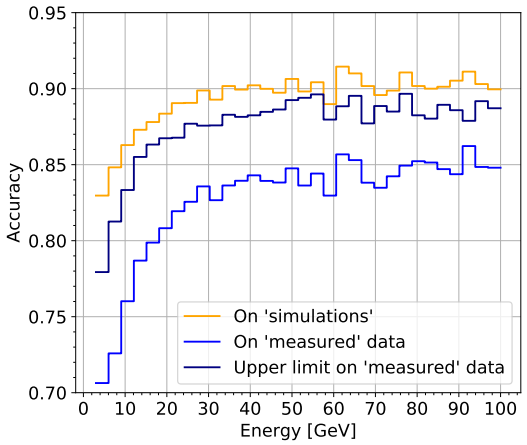
(a) *600-50 filters* encoder+dense network.



(b) *200-conv* encoder+dense network.



(c) *model-64* encoder+dense network.



(d) *model-32* encoder+dense network.

Figure 6.7.: For four different encoder+dense architectures, the up-down accuracy on the test set over the neutrino energy for the three different evaluations are shown. Down-going events have been made brighter in the "simulation" dataset.

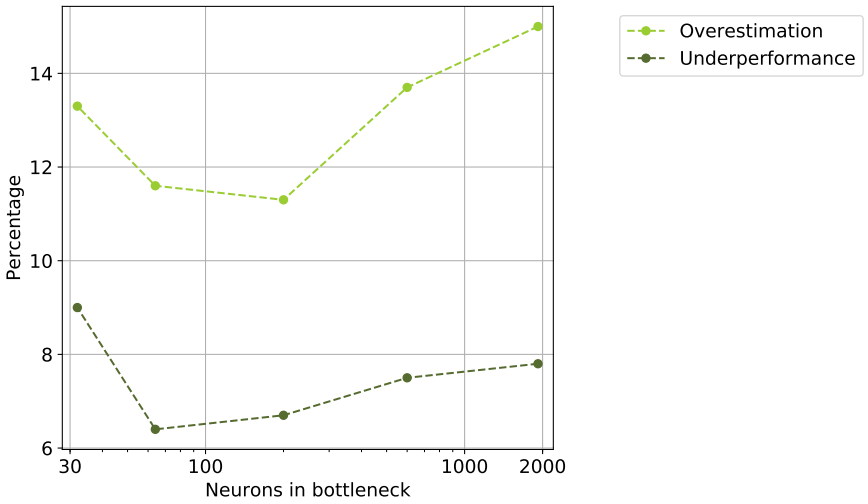


Figure 6.8.: Plot of the overestimation and underperformance of encoder+dense networks with different sizes of bottlenecks. The closer to zero the values are, the more robust the network is. The "simulations" have been manipulated by making down-going events brighter (section 6.1.4).

6.2. Energy regression

The following networks are used to predict the energy of a neutrino, with the mean absolute error of the reconstructed energy as the quantity to minimize. Instead of an accuracy, their performances are compared with each other by calculating the median relative error over all samples in the dataset: The smaller it is the better the network. This is unlike the accuracy of the up-down classification, which was supposed to be maximized. Therefore, the definitions of the overestimation and underperformance from the previous section are altered by flipping the quantities in the numerator:

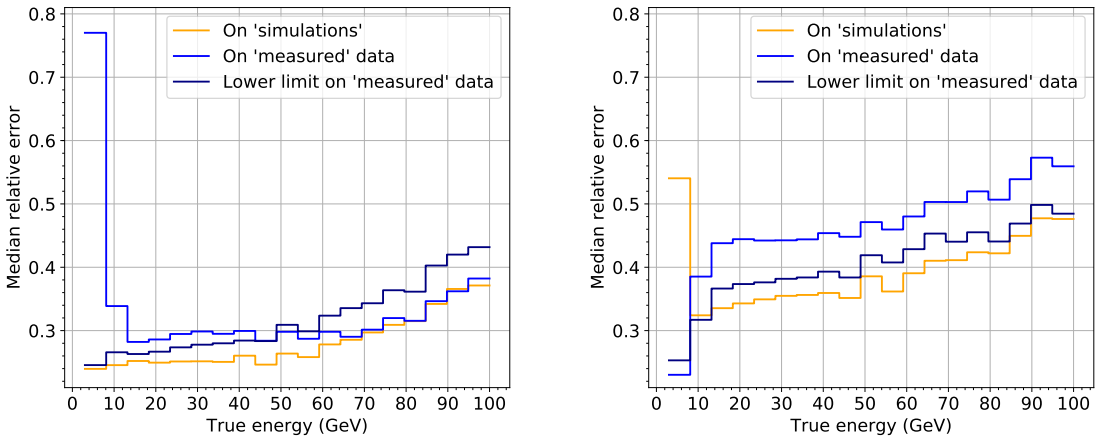
$$\text{Overestimation} = \frac{\text{MRE}_{\text{meas}} - \text{MRE}_{\text{sim}}}{\text{MRE}_{\text{meas}}} \tag{6.2.1}$$

$$\text{Underperformance} = \frac{\text{MRE}_{\text{meas}} - \text{MRE}_{\text{lim}}}{\text{MRE}_{\text{meas}}}. \tag{6.2.2}$$

The closer to zero these values are, the more robust the network is against a specific manipulation. As in the previous section, the encoder+dense network and the one from the supervised approach are first trained and tested on the "simulation" set, until the best performance is reached (MRE_{sim}). Then, these networks are tested on the "measured" dataset (MRE_{meas}). Finally, they are trained and tested on "measured" data (MRE_{lim}). The autoencoder of the unsupervised approach is always trained on "measured" data.

6.2.1. Additional uncorrelated noise

This is the same manipulation as in section 6.1.2, in which the expectation value of the uncorrelated background Poisson noise is doubled in the "measured" dataset. The three evaluations for the supervised and unsupervised approach are shown in figure 6.9, and the results are listed in the following table.



(a) Supervised approach.

(b) *model-1920* encoder+dense network.

Figure 6.9.: Median relative error of the energy reconstruction on the test set over the neutrino energy for the three different evaluations. Additional uncorrelated noise was added to the "simulations". As before, "On simulations" is trained and tested on "simulations"; "On measured data" is trained on "simulations" and tested on "measured data"; "Upper limit" is trained and tested on "measured data".

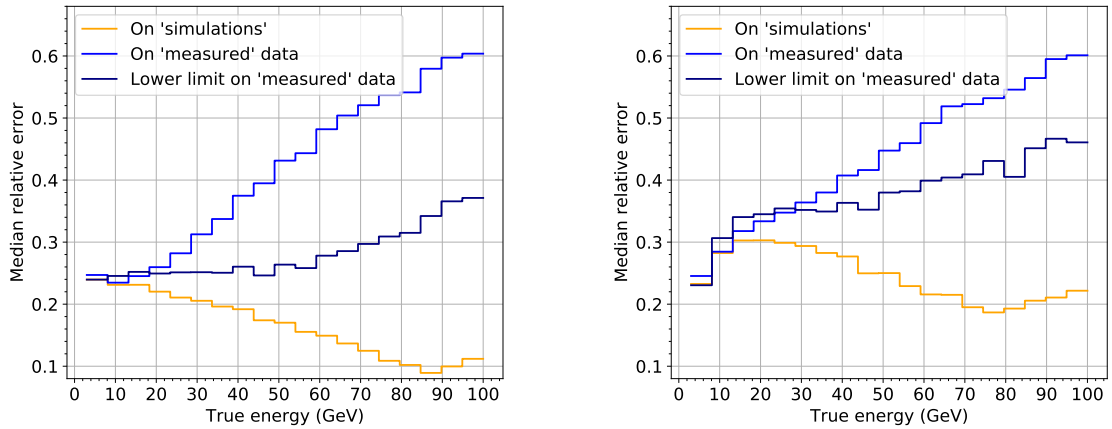
	MRE_{meas}	Overestimation	Underperformance
Supervised approach	44.48%	44.5%	41.3%
model-1920 encoder+dense	**32.06%**	**−26.7%**	**5.5%**

The impact of the additional noise on the energy reconstruction of the supervised approach is quite large. This is likely because the networks rely to some degree on the number of counts belonging to the event for their reconstruction. Since the average number of counts per bin is increased in the manipulated "simulations", the networks adapt to that during training. This leads to especially large errors for the lower energies, where fewer counts are expected to be measured per event (light blue curve). The performance on "simulations" (orange) is better than the limit on "measured" data (dark blue), as the signal-to-noise ration is higher there.

For high energies, the error on the "measured" set is higher if the network was also trained on this set (dark blue) instead of on the "simulations" (light blue). The mean absolute error, which the network is trained to minimize, is averaged over events and not energy bins. Since events with low energies appear much more often in the dataset, they have a larger influence on the loss. Because of this, the dark blue curve corresponds to a lower overall mean absolute error, despite it being inferior for higher energies.

The encoder+dense network shows an interesting property: When applying the network that was trained on "simulations" to the manipulated "measured" dataset, the performance *increases* by 26.7%, even though the signal-to-noise ratio is smaller there. This is caused by the events with very low energy in the first bin, which appear more often in the dataset than the high energy ones, and therefore have a larger influence on the median relative error. Apparently, the encoder adapted to the low signal-to-noise ratio data in a way that worsens the performance on the "simulations" set. When the fully trained network is applied to "measured" data again, this effect is not visible anymore, and the resulting performance is only 5% worse than if it had been trained on "measured" data.

In the first bin for very low energies, the MRE on "measured" data is below the lower limit.



(a) Supervised approach.

(b) *model-1920* encoder+dense network.

Figure 6.10.: Median relative error of the energy reconstruction on the test set over the neutrino energy for the three different evaluations. Correlated noise was added to the "simulations".

This is likely caused by an underestimation of the true energy, and events never being below 3 GeV in the dataset. It is explained in more detail in section 6.2.4, where this effect is visible even stronger.

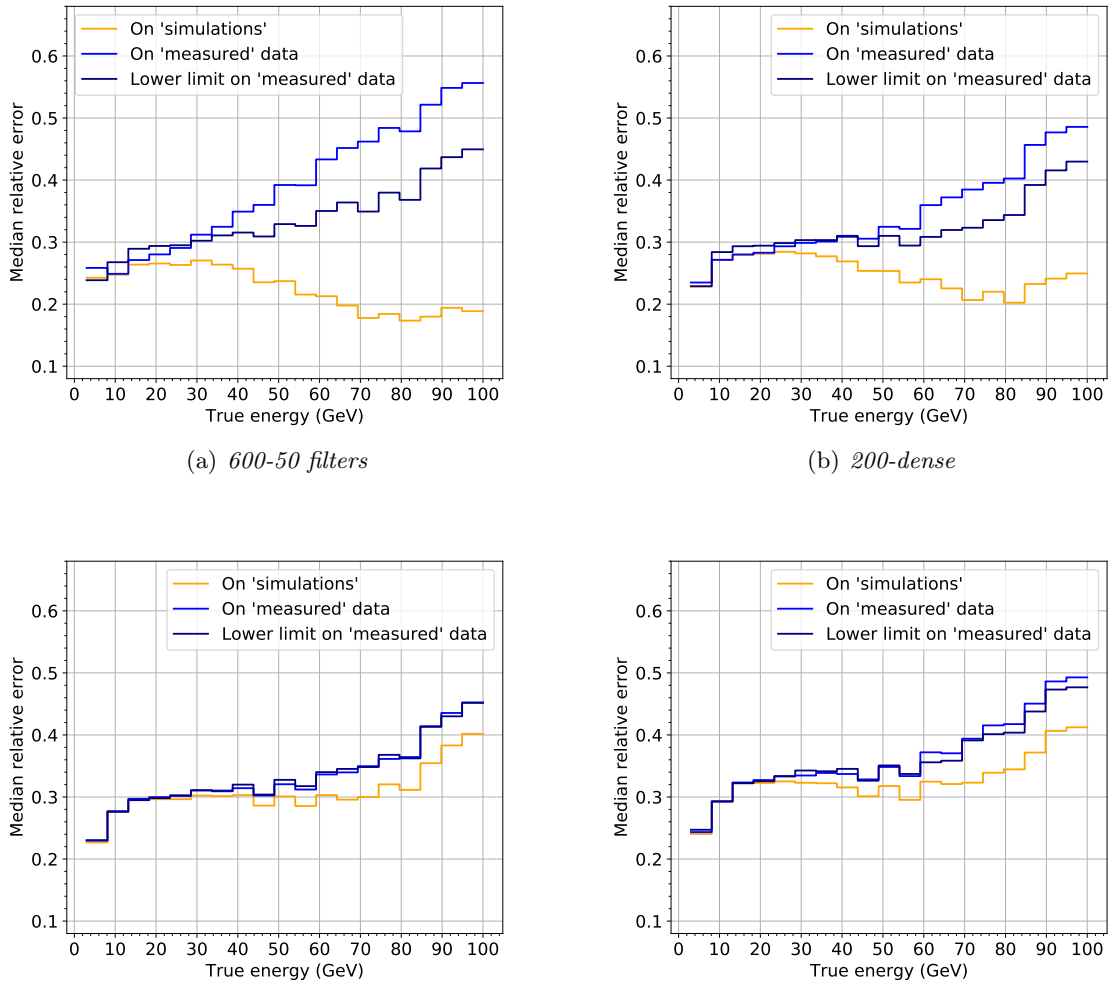
6.2.2. Additional correlated noise

For the manipulation in this section, additional Poisson noise which is correlated to the energy of the neutrino is added to the "simulations" dataset: Events with an energy of 3 GeV have the standard background noise frequency of 10 kHz per photomultiplier. This frequency is increased linearly up to 12 kHz at 100 GeV. The resulting evaluations are shown in figure 6.10.

Both networks show an increasingly better performance on the "simulations" for higher energies, as the correlation is stronger there. By taking into account the background noise rate, the networks improve the quality of their prediction. If these networks are then applied to the "measured" dataset, they are presented with the real 10 kHz noise rate for all samples, independent of the neutrino energy. This leads to an underestimation of the true energy: When the supervised approach is tested on the "measured" data, the true energy is 0.51 GeV larger then the reconstructed energy on median average, as compared to -0.14 GeV on the "simulated" data. For the encoder+dense network, the true energy is 0.74 GeV larger on the "measured" data, and 0.37 GeV on the "simulations".

Even though the encoder+dense network is more robust, it still performs worse in this scenario than the network trained according to the supervised approach, as it has an inherently better performance. The results are shown in the following table.

	MRE_{meas}	Overestimation	Underperformance
Supervised approach	**27.05%**	18.8%	8.7%
model-1920 encoder+dense	29.23%	**12.6%**	**4.2%**



(a) *600-50 filters* (b) *200-dense*

(c) *model-64* (d) *model-32*

Figure 6.11.: For four different encoder+dense architectures, the median relative error of the energy reconstruction on the test set over the neutrino energy is shown. Correlated noise was added to the "simulations".

6.2.3. Dependency on the bottleneck size

The robustness of the encoder+dense network depends on the size of the bottleneck of the autoencoder. In this section, models with different bottlenecks are trained with the manipulation from the previous section 6.2.2, in which correlated Poisson noise is added to the "simulations". The resulting evaluations for the best performing models with 600, 200, 64 and 32 neurons in the bottleneck are shown in figure 6.11. For smaller bottlenecks the three lines approach each other, indicating higher robustness. For the *model-64*, the light and dark blue lines are almost on top of each other, even though the network still makes use of the correlation at higher energies on the "simulations" (orange line).

The results of the evaluations are shown in the middle section of table 6.2. The overestimations and underperformances are also plotted in figure 6.12. The robustness increases if the bottleneck is tightened up to 64 neurons. If it is reduced further to 32 neurons, the robustness decreases again very slightly. The network with 64 neurons has almost the same performance

Table 6.2.: Comparison of the robustness of different model architectures. MRE_{sim} is the median relative error of a model that was trained and tested on the manipulated "simulations". MRE_{meas} is the MRE on the "measured data" test dataset of a model that was trained on the "simulation" dataset. The overestimation and underperformance are calculated as shown in equation 6.2.1 and 6.2.2; The closer to zero they are, the more robust the network is against the given manipulation.

Model	MRE_{sim}	MRE_{meas}	Overestimation	Underperformance
Supervised approach	21.98%	27.06%	18.8%	8.7%
model-1920	25.55%	29.24%	12.6%	4.2%
600-50 filters	**24.51%**	27.53%	11.0%	3.8%
200-dense	24.86%	**26.20%**	5.1%	0.5%
model-64	25.83%	26.26%	**1.6%**	**0.1%**
model-32	27.54%	28.12%	2.1%	-0.5%
200-pool	25.76%	28.49%	9.6%	5.5%
200-wide	27.76%	28.39%	**2.2%**	4.6%
200-small	**25.41%**	**26.53%**	4.2%	**-0.5%**

in all three evaluations, resulting in a very good overestimation and underperformance close to zero. Both the networks with 200 and 64 neurons manage to outperform the supervised approach on the "measured" dataset, due to their much higher robustness.

This suggests that for smaller bottlenecks, the encoder provides less information about the noise in its input, so that the dense layers cannot make use of the artificially added correlation. The noise is not correlated with the neutrino energy in the "measured" dataset on which the autoencoder is trained, so its encoder cannot learn to relate these two features.

One might wonder how a network like the *model-32* can manage to achieve a negative underperformance, i.e. have a slightly lower median relative error on the "measured" set if the encoder+dense network was trained on "simulations" instead of the "measured" dataset. This comes from the fact that the metric used for training the models is the mean absolute error, while the results in table 6.2 are calculated from the median relative error. The latter is more commonly used for assessing the quality of energy reconstructions. When the underperformance for *model-32* is calculated with the MAE, it is not negative and amounts to 0.26% instead.

As with the robustness test of the up-down classification, the model *600-50 filters* has the highest performance on the "simulations" set MRE_{sim}. In the practical application, this would be the only obtainable quantity from the ones listed in the table. However, this architecture is performing distinctively worse on the "measured" set than the ones with 200 or 64 neurons in the bottleneck. For real measured data, tests with multiple models and manipulations like the one in this section might be necessary to determine the best encoder.

For comparison, architectures with an increased or reduced number of free parameters are also listed in the lower section of table 6.12, together with the model *200-pool* on which their design is based on. The model *200-small* with fewer parameters has a higher robustness and lower MRE on the "measured set" than the model *200-pool*. This is similar to its behaviour in section 5.2, in which the datasets were not manipulated.

6.2.4. Reduced photomultiplier efficiency

The photomultipliers of the ORCA detector might measure less photons the longer they are in the water, perhaps due to opaque organic structures forming on the hull of the DOMs. If this effect is not properly accounted for in the simulations, it would lead to a difference in the

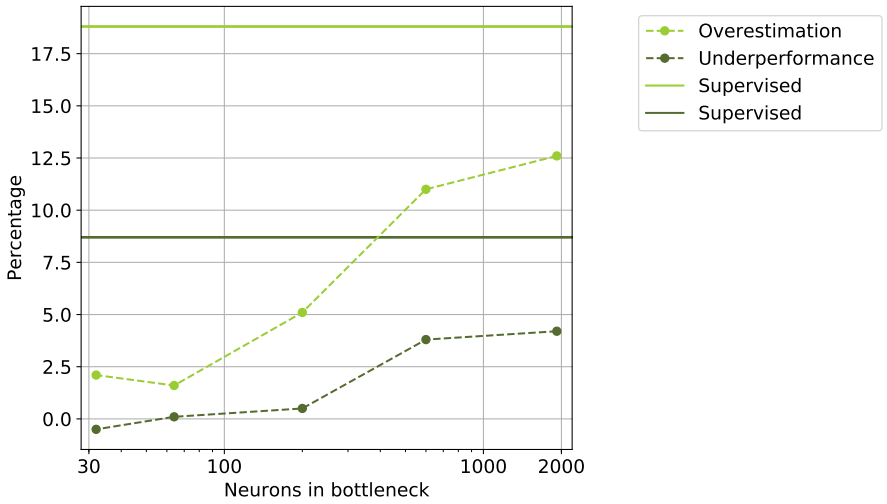


Figure 6.12.: Plot of the overestimation and underperformance of encoder+dense networks with different sizes of the bottleneck, but comparable numbers of free parameters. The "simulations" have been manipulated by adding energy-correlated noise (section 6.2.3). The values of the supervised approach are shown as solid lines.

datasets.

To test the impact of this, the efficiency of the photomultipliers in the "measured" dataset is reduced by up to 55% (figure 6.13). It is reduced more strongly for bins with a high x coordinate to represent the different points in time at which the lines might be placed in the water during construction - the sooner they got placed, the lower the efficiency. This is likely not the order after which the lines will be anchored in reality.

If a bin has n counts and an efficiency η, the manipulated number of counts n' is calculated by sampling from a binomial distribution:

$$n' = n - B(n, 1 - \eta). \tag{6.2.3}$$

This means that every count has a chance of $1-\eta$ to be removed in the "measured" dataset. The efficiencies are the same for all samples in the dataset, but the reduced counts are recalculated randomly for every one of them. The evaluations of the networks trained according to the supervised and unsupervised approach are compared in figure 6.14. The results are also listed in the following table.

	MRE_{meas}	Overestimation	Underperformance
Supervised approach	28.99%	**14.8%**	10.6%
model-1920 encoder+dense	**28.41%**	-30.2%	**1.2%**

For the supervised approach, the MRE of a network which was trained and tested on manipulated "measured data" (dark blue) is higher than that of a network which was trained and tested on "simulations" (orange). It is likely that the network takes into account the number of counts belonging to an event for its reconstruction, as it is strongly correlated with the energy of the event. The random removal of counts on the "measured" dataset adds additional variance and therefore makes it more difficult to accurately predict the energy with this strategy.

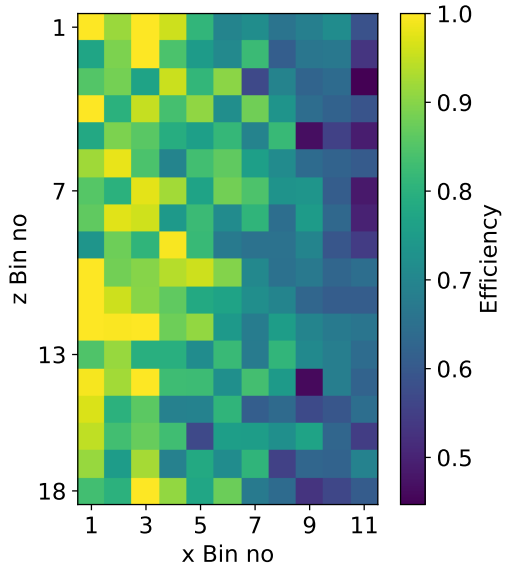


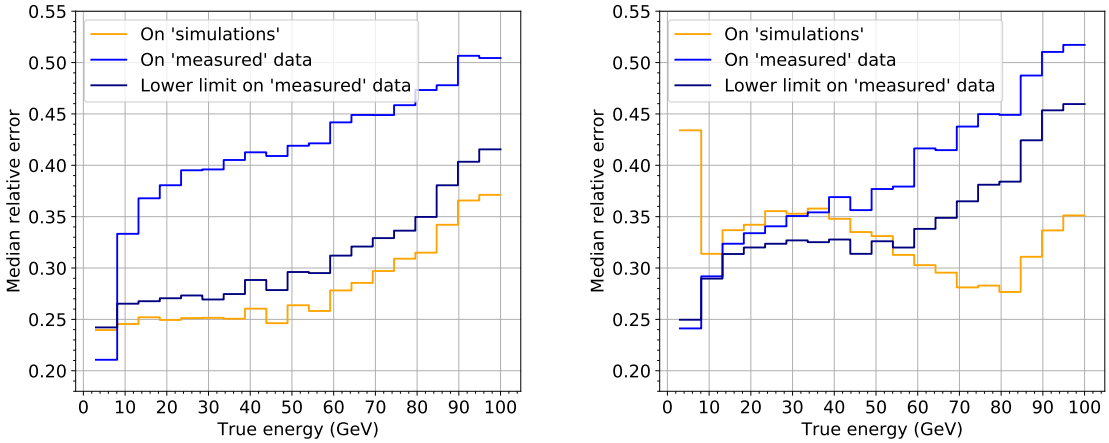
Figure 6.13.: Efficiency of every bin in the xz-plane for the manipulation in section 6.2.4, which is applied to the "measured" dataset. The efficiency decreases linearly in the x-direction. Additionally, a Gaussian uncertainty with a standard deviation of 0.1 is incorporated, while keeping the efficiency below 1. The efficiency is constant over time.

When the network that was trained on "simulations" is applied to "measured data" (light blue), the performance drops severely, but not in the first bin where it actually manages to improve its performance. This can be explained with the histograms in figure 6.15. The network learns to refrain from predicting energies close to 3 GeV, since this was the lowest energy it encountered during training. This means that very low energies are slightly overestimated. On the "measured" dataset, the network generally tends to underestimate the true energy, as less counts are present due to the lower efficiencies. For low energies these effects counteract each other to some degree, thus resulting in a better reconstruction of very low energetic events, but a worse one for all others.

The encoder+dense network in figure 6.14(b) has a bad performance during the supervised training on the "simulations" (orange), but improves drastically when moving back to the "measured" data after the training (light blue). Apparently, the encoder adapted to the lower number of counts during the autoencoder training, making it difficult for the dense layers to properly reconstruct the energy on the non-manipulated dataset. In the practical application, an encoder+dense performance that is far below expectations on the simulations might be taken as an evidence that they differ from the measured data significantly.

6.3. Conclusion and discussion

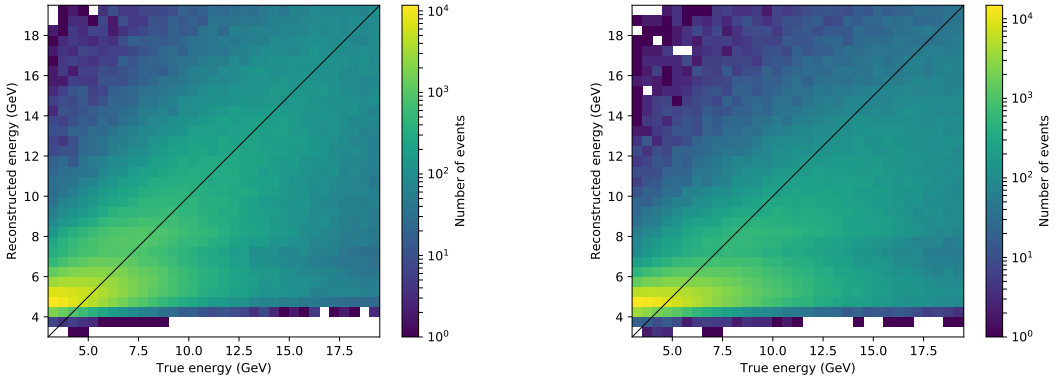
Networks trained according to the unsupervised autoencoder approach have shown a higher robustness than the ones from the supervised approach on almost all of the tested manipulations (see table 6.3 for a summary of the results of this chapter). As the performance of encoder+dense networks is generally worse, a higher robustness does not necessarily imply an actual superior quality of the reconstructions on the "measured" dataset. This becomes



(a) Supervised approach.

(b) *model-64* encoder+dense network.

Figure 6.14.: Median relative error of the energy reconstruction on the test set over the neutrino energy for the three different evaluations. The efficiency of the photomultipliers in the "measured data" were reduced.



(a) On "simulations".

(b) On "measured data".

Figure 6.15.: Reconstructed energy over true energy for the supervised approach at low neutrino energies. The network is trained and tested on the non-manipulated "simulated" data (left), and then tested on the manipulated "measured data" (right).

Table 6.3.: Overestimation and underperformance of the tested manipulations of chapter 6 for the networks trained according to the supervised approach, and the encoder+dense networks. For each manipulation, the overestimation is shown first in the cell, and the underperformance below it. The upper half of the table lists the robustness indicators for the up-down classification, while the lower half is for the energy reconstruction. The values of the most robust networks, i.e. the ones that are closest to zero, are printed in bold.

Manipulation	Supervised approach	Encoder+Dense
Defect bin	Overest.: 88.7% Underperf.: 66.4%	**0.0%** **0.1%**
Additional uncorrelated noise	6.7% 3.5%	**2.8%** **1.0%**
Brighter down-going events	39.7% 28.6%	**8.5%** **4.6%**
Additional uncorrelated noise	44.5% 41.3%	**-26.7%** **5.5%**
Additional correlated noise	18.8% 8.7%	**1.6%** **0.1%**
Reduced photomultiplier efficiency	**14.8%** 10.6%	-30.2% **1.2%**

especially apparent for manipulations that are not correlated with the property that the network is supposed to reconstruct. For example, the addition of Poisson noise does not strongly impair the way in which the network classifies events as up- or down-going, so the loss of performance when switching datasets is marginal. In this case, the network from the supervised approach managed to perform better on the noisy data, despite its lower robustness.

On the other hand, added deviating features which strongly correlate with the target property can render the supervised approach anywhere from significantly impaired up to almost useless in extreme cases, while the encoder+dense networks retain much of their quality thanks to the superior robustness. The usefulness of the autoencoder-based training does therefore strongly depend on the nature of the true deviations, that may or may not be found between actual measured detector data and simulations.

A possible way of testing whether those two datasets differ is to train a neural network which tells apart samples from them. This would essentially be a classification into two categories – simulations and measured data – which could be implemented in a similar way as the up-down classification network in this thesis. If the datasets can be distinguished, then there is a difference between them to which a network can become sensitive to. However, this technique cannot establish how strong the impact of the difference is on the reconstruction tasks of the networks.

The manipulations in this chapter were chosen so that they can be easily implemented by changing existing simulations afterwards. This way, it was possible to quickly conduct many different robustness tests. However, changing parameters of the simulations, like the properties of the water, might lead to differences which are closer to the possibly existing real ones. In general, there is a trade-off between realistic manipulations and the use of autoencoders: If the differences seem plausible, their cause and implication might be understood well, so that the simulations could be adjusted accordingly. This would eliminate the need for an autoencoder altogether, as the supervised approach has proven to perform better on identical datasets. The autoencoder is therefore especially useful when the nature of the differences is

unknown, which in turn might make them seem unrealistic when tested as a manipulation. In any case, the more manipulations are tested, the more convincing the advantage of the autoencoder training becomes.

The tested autoencoder architectures with the highest robustness in both the up-down classification and the energy reconstruction are not the ones which lead to the highest encoder+dense performance on the "simulations". This means that in the practical application where the autoencoder is trained on real measured data, one should not simply choose the encoder+dense network which performs best on the simulations, as it might not have the best performance on measured data. Instead, robustness tests like the ones in this chapter for different bottleneck sizes and manipulations are advised.

It is conspicuous that networks which perform well when there are no differences between the datasets often also exhibit the highest robustness or best performance on the "measured" dataset when manipulations are introduced. If this relation would be accurate, it would allow for identifying a universally robust architecture independent of the properties of the differences. This would be especially useful in the practical application with real measured data.

The most robust encoder+dense network for the up-down classification was built from the autoencoder of the model *200-wide*, while it was the one of *model-64* for the energy reconstruction. This could mean that either different supervised tasks or different manipulations require varying autoencoder architectures for a maximized robustness. Tests with other manipulations for the given tasks and models are required to clarify this.

7. Summary and Outlook

An unsupervised approach based on deep convolutional autoencoders has been successfully used to train networks with a comparable, yet slightly inferior performance as compared to networks trained in a supervised fashion. For this, an autoencoder was trained unsupervised to reconstruct its input by encoding relevant features in its bottleneck. Then, another small network was trained supervised to extract the desired property of the neutrino from the output of the encoder part.

The best autoencoder based network was able to classify 85.5% of the neutrinos on an unseen simulated dataset without precuts correctly as up- or down-going, as opposed to the 88.2% of a comparable network trained according to a supervised approach. For a reconstruction of the energy of electron and muon neutrinos, the median relative error in the range from 3 to 100 GeV is 26.1% (unsupervised) and 24.7% (supervised). The size of the bottleneck plays an important role in the quality of the resulting reconstructions. Different autoencoder architectures are used for these two tasks, but both have 200 neurons in their bottleneck.

It is unclear why the autoencoder based networks show a worse performance. There is evidence suggesting that this might not be an intrinsic property of autoencoders, but could instead be caused by the loss function used to train the autoencoder, the mean squared error. Slight modifications to the function did not resolve the issue, so more sophisticated methods such as the use of a generative adversarial autoencoder might be required.

The main advantage of the unsupervised approach is its higher robustness against potential differences between the simulations and the measured data. This was investigated by evaluating networks on a different simulated dataset than the one they were trained on. These differences were introduced, for example, by adding specific correlations. In general, the autoencoder based networks showed a smaller reduction in performance when switching the datasets. The robustness is found to be dependent on the size of the autoencoder's bottleneck, with 200 neurons being the best size from the tested ones.

Despite its worse performance on identical datasets, the network of the unsupervised approach is superior for multiple of the tested manipulations. Whether or not this will also be the case in the practical application with real measured data depends on the nature of the true differences to the simulations, if there are any to be observed. The stronger the impact of the disparity is on the performance of the networks, the more likely it is that the autoencoder based approach works better on measured data. If the gap in performance between the supervised and unsupervised approach could be removed, it could even be considered a strictly superior method of training the network.

All networks in this thesis were trained on three-dimensional xzt data. It is likely that incorporating the y- and photomultiplier-ID-dimension of the ORCA data will increase the overall performance of many reconstructions. This will require a reiteration of some of the studies in this thesis, as different architectures and sizes of bottlenecks might be necessary.

Once the construction of the ORCA detector is finished, the autoencoder approach can be tested on actual measured data. So far, simulations which are composed of very large quantities of neutrino events were used for training. The measured data is expected to contain relatively few of those, and mostly atmospheric muons otherwise. This might have a negative impact on the performance of the autoencoder when it comes to reconstructing neutrino events, so it is vital to test this scenario beforehand, for example by using a simulated dataset

with the corresponding particle fluxes.

Unsupervised Deep Learning on the basis of autoencoders is a promising and interesting approach to tackle the problem of simulations deviating from measured data. With additional research answering some of the left-open questions, this technique has the potential to boost the confidence in modern Machine Learning algorithms in the physics community.

8. Bibliography

[1] A. de Gouvea et al. "Neutrinos". In: (Oct. 16, 2013). arXiv: 1310.4340v1 [hep-ex].

[2] C. Giunti and C.W. Kim. *Fundamentals of Neutrino Physics and Astrophysics.* OUP Oxford, 2007. ISBN: 9780191523229. URL: https://books.google.de/books?id=2faTXKIDnfgC.

[3] F. Couchot et al. "Cosmological constraints on the neutrino mass including systematic uncertainties". In: *A&A 606, A104 (2017)* (Mar. 31, 2017). DOI: 10.1051/0004-6361/201730927. arXiv: 1703.10829v2 [astro-ph.CO].

[4] N. Schmitz. *Neutrinophysik.* Teubner Studienbücher Physik. Vieweg+Teubner Verlag, 2013. ISBN: 9783322801142. URL: https://books.google.de/books?id=Ud2SxVnGIgIC.

[5] A. Bellerive et al. "The Sudbury Neutrino Observatory". In: (Feb. 8, 2016). DOI: 10.1016/j.nuclphysb.2016.04.035. arXiv: 1602.02469v2 [nucl-ex].

[6] Michael Wurm. *Neutrino mass hierarchy.* URL: http://www.staff.uni-mainz.de/wurmm/juno.html (visited on 07/02/2018).

[7] K.A. Olive. "Review of Particle Physics". In: *Chinese Physics C* 40.10 (2016), p. 100001. DOI: 10.1088/1674-1137/40/10/100001.

[8] S. Adrián-Martínez et al. "Letter of Intent for KM3NeT 2.0". In: *Journal of Physics G: Nuclear and Particle Physics, 43 (8), 084001, 2016* (Jan. 27, 2016). DOI: 10.1088/0954-3899/43/8/084001. arXiv: 1601.07459v2 [astro-ph.IM].

[9] Takaaki Kajita. *Atmospheric Neutrinos.* 2012. DOI: 10.1155/2012/504715. URL: http://dx.doi.org/10.1155/2012/504715.

[10] M. Honda et al. "Atmospheric neutrino flux calculation using the NRLMSISE00 atmospheric model". In: (Feb. 13, 2015). DOI: 10.1103/PhysRevD.92.023004. arXiv: 1502.03916v2 [astro-ph.HE].

[11] Javier Tiffenberg. *UHE Neutrino searches with the Pierre Auger Observatory.* At the NUSKY 2011. URL: http://users.ictp.it/~smr2246/monday/tiffenberg-NUSKY.pdf (visited on 07/05/2018).

[12] Edewolf. *Artists impression of the KM3NeT detector.* URL: https://en.wikipedia.org/wiki/File:Artist%27s_expression_of_KM3NeT.jpg (visited on 07/05/2018).

[13] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), p. 354. URL: http://dx.doi.org/10.1038/nature24270.

[14] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (Sept. 4, 2014). arXiv: 1409.1556v6 [cs.CV].

[15] Stanford University. *CS231n syllabus.*
URL: http://cs231n.github.io/neural-networks-1/.

[16] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks".
In: *Neural Networks* 4.2 (1991), pp. 251 –257. ISSN: 0893-6080.
DOI: https://doi.org/10.1016/0893-6080(91)90009-T.
URL: http://www.sciencedirect.com/science/article/pii/089360809190009T.

[17] Nicolas Brunel, Vincent Hakim, and Magnus JE Richardson.
"Single neuron dynamics and computation". In: *Current Opinion in Neurobiology* 25 (2014). Theoretical and computational neuroscience, pp. 149 –155. ISSN: 0959-4388.
DOI: https://doi.org/10.1016/j.conb.2014.01.005.
URL: http://www.sciencedirect.com/science/article/pii/S0959438814000130.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton.
"ImageNet Classification with Deep Convolutional Neural Networks".
In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1.* NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[19] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization".
In: (Dec. 22, 2014). arXiv: 1412.6980v9 [cs.LG].

[20] Vincent Dumoulin and Francesco Visin.
"A guide to convolution arithmetic for deep learning". In: (Mar. 23, 2016).
arXiv: 1603.07285v2 [stat.ML].

[21] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (Dec. 10, 2015).
arXiv: 1512.03385v1 [cs.CV].

[22] Mohit Deshpande.
Introduction to Convolutional Neural Networks for Vision Tasks (modified).
URL: https://pythonmachinelearning.pro/introduction-to-convolutional-neural-networks-for-vision-tasks/.

[23] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (Feb. 11, 2015).
arXiv: 1502.03167v3 [cs.LG].

[24] Nitish Srivastava et al.
"Dropout: A Simple Way to Prevent Neural Networks from Overfitting".
In: 15 (June 2014), pp. 1929–1958.

[25] Yann LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade.*
Ed. by Genevieve B. Orr and Klaus-Robert Müller.
Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. ISBN: 978-3-540-49430-0.
DOI: 10.1007/3-540-49430-8_2.
URL: https://doi.org/10.1007/3-540-49430-8_2.

[26] Adrian Alan Pol et al. "Detector monitoring with artificial neural networks at the CMS experiment at the CERN Large Hadron Collider". In: (July 27, 2018).
arXiv: 1808.00911v1 [physics.data-an].

[27] Qinxue Meng et al. "Relational Autoencoder for Feature Extraction".
In: *2017 International Joint Conference on Neural Networks (IJCNN), Anchorage, AK, 2017, pp. 364-371* (Feb. 9, 2018). DOI: 10.1109/IJCNN.2017.7965877.
arXiv: 1802.03145v1 [cs.LG].

[28] Marco Farina, Yuichiro Nakai, and David Shih.
"Searching for New Physics with Deep Autoencoders". In: (Aug. 27, 2018).
arXiv: 1808.08992v1 [hep-ph].

[29] Theo Heimel et al. "QCD or What?" In: (Aug. 27, 2018).
arXiv: 1808.08979v1 [hep-ph].

[30] Zhengxue Cheng et al.
"Deep Convolutional AutoEncoder-based Lossy Image Compression". In: (Apr. 25, 2018).
arXiv: 1804.09535v1 [cs.CV].

[31] Fjodor van Veen. *The neural network zoo*. URL:
http://www.asimovinstitute.org/neural-network-zoo/ (visited on 07/11/2018).

[32] François Chollet et al. *Keras*. https://keras.io. 2015.

[33] Martín Abadi et al.
TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.
Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

A. Network architectures

This chapter contains the explicit architecture of the models used in this thesis. Motivations to their design are given in the sections in which they first appear. The building blocks as well as the default properties of the layers are defined in section 3.3.

A.1. For chapter 4

Table A.1.: Network architecture of the *model-1920 12 layers* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	40
Convolutional block	11x18x50	x	40
Average Pooling (1,1,2)	11x18x25	x	40
Convolutional block	11x18x25	x	40
Convolutional block	10x16x24	x	40
Average Pooling (2,2,2)	5x8x12	x	40
Convolutional block	5x8x12	x	80
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	6x8x12	x	80
Convolutional block	5x8x12	x	40
Upsampling (2,2,2)	10x16x24	x	40
Convolutional block (T)	12x18x26	x	40
Convolutional block	11x18x25	x	40
Upsampling (1,1,2)	11x18x50	x	40
Convolutional block (T)	11x18x50	x	40
Convolutional block (T)	11x18x50	x	40
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 753969

Table A.2.: Network architecture of the *model-1920 16 layers* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	28
Convolutional block	11x18x50	x	28
Average Pooling (1,1,2)	11x18x25	x	28
Convolutional block	11x18x25	x	28
Convolutional block	10x16x24	x	28
Average Pooling (2,2,2)	5x8x12	x	28
Convolutional block	5x8x12	x	55
Convolutional block	6x8x12	x	55
Convolutional block	6x8x12	x	55
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	6x8x12	x	55
Convolutional block	5x8x12	x	55
Convolutional block (T)	5x8x12	x	55
Convolutional block (T)	5x8x12	x	28
Upsampling (2,2,2)	10x16x24	x	28
Convolutional block (T)	12x18x26	x	28
Convolutional block	11x18x25	x	28
Upsampling (1,1,2)	11x18x50	x	28
Convolutional block (T)	11x18x50	x	28
Convolutional block (T)	11x18x50	x	28
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 750193

Table A.3.: Network architecture of the *model-1920 20 layers* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	26
Convolutional block	11x18x50	x	26
Convolutional block	11x18x50	x	26
Average Pooling (1,1,2)	11x18x25	x	26
Convolutional block	11x18x25	x	26
Convolutional block	10x16x24	x	26
Convolutional block	10x16x24	x	26
Average Pooling (2,2,2)	5x8x12	x	26
Convolutional block	5x8x12	x	52
Convolutional block	6x8x12	x	52
Convolutional block	6x8x12	x	52
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	6x8x12	x	52
Convolutional block	5x8x12	x	52
Convolutional block (T)	5x8x12	x	52
Convolutional block (T)	5x8x12	x	26
Upsampling (2,2,2)	10x16x24	x	26
Convolutional block (T)	12x18x26	x	26
Convolutional block	11x18x25	x	26
Convolutional block (T)	11x18x25	x	26
Upsampling (1,1,2)	11x18x50	x	26
Convolutional block (T)	11x18x50	x	26
Convolutional block (T)	11x18x50	x	26
Convolutional block (T)	11x18x50	x	26
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 747681

Table A.4.: Network architecture of the *model-1920 30 layers* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	21
Convolutional block	11x18x50	x	21
Convolutional block	11x18x50	x	21
Convolutional block	11x18x50	x	21
Average Pooling (1,1,2)	11x18x25	x	21
Convolutional block	11x18x25	x	21
Convolutional block	10x16x24	x	21
Convolutional block	10x16x24	x	21
Convolutional block	10x16x24	x	21
Convolutional block	10x16x24	x	21
Average Pooling (2,2,2)	5x8x12	x	21
Convolutional block	5x8x12	x	41
Convolutional block	6x8x12	x	41
Convolutional block	6x8x12	x	41
Convolutional block	6x8x12	x	41
Convolutional block	6x8x12	x	41
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	6x8x12	x	41
Convolutional block	5x8x12	x	41
Convolutional block (T)	5x8x12	x	41
Convolutional block (T)	5x8x12	x	41
Convolutional block (T)	5x8x12	x	41
Convolutional block (T)	5x8x12	x	21
Upsampling (2,2,2)	10x16x24	x	21
Convolutional block (T)	12x18x26	x	21
Convolutional block	11x18x25	x	21
Convolutional block (T)	11x18x25	x	21
Convolutional block (T)	11x18x25	x	21
Convolutional block (T)	11x18x25	x	21
Upsampling (1,1,2)	11x18x50	x	21
Convolutional block (T)	11x18x50	x	21
Convolutional block (T)	11x18x50	x	21
Convolutional block (T)	11x18x50	x	21
Convolutional block (T)	11x18x50	x	21
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 756040

Table A.5.: Network architecture of the *model-1920 60 layers* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
8 × Convolutional block	11x18x50	x	15
Average Pooling (1,1,2)	11x18x25	x	15
Convolutional block	11x18x25	x	15
9 × Convolutional block	10x16x24	x	15
Average Pooling (2,2,2)	5x8x12	x	15
Convolutional block	5x8x12	x	28
10 × Convolutional block	6x8x12	x	28
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	6x8x12	x	28
Convolutional block	5x8x12	x	28
10 × Convolutional block (T)	5x8x12	x	15
Upsampling (2,2,2)	10x16x24	x	15
Convolutional block (T)	12x18x26	x	15
Convolutional block	11x18x25	x	15
8 × Convolutional block (T)	11x18x25	x	15
Upsampling (1,1,2)	11x18x50	x	15
8 × Convolutional block (T)	11x18x50	x	15
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 758324

Table A.6.: Network architecture of the *small kernel* autoencoder. The standard kernel size for convolutions and transposed convolutions is $2 \times 2 \times 2$ for this model.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	51
Convolutional block	11x18x50	x	51
Convolutional block	11x18x50	x	51
Average Pooling (1,1,2)	11x18x25	x	51
Convolutional block	10x17x24	x	51
Convolutional block	10x16x24	x	51
Convolutional block	10x16x24	x	51
Average Pooling (2,2,2)	5x8x12	x	51
Convolutional block	5x8x12	x	102
Convolutional block	6x8x12	x	102
Convolutional block	5x7x11	x	102
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	5x7x11	x	102
Convolutional block	5x8x12	x	102
Convolutional block (T)	5x8x12	x	102
Convolutional block (T)	5x8x12	x	51
Upsampling (2,2,2)	10x16x24	x	51
Convolutional block (T)	11x17x25	x	51
Convolutional block	11x18x25	x	51
Convolutional block (T)	11x18x25	x	51
Upsampling (1,1,2)	11x18x50	x	51
Convolutional block (T)	11x18x50	x	51
Convolutional block (T)	11x18x50	x	51
Convolutional block (T)	11x18x50	x	51
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 752634

Table A.7.: Network architecture of the *triple* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	14
Convolutional block	11x18x50	x	14
Convolutional block	11x18x50	x	14
Average Pooling (1,1,2)	11x18x25	x	14
Convolutional block	11x18x25	x	28
Convolutional block	10x16x24	x	28
Convolutional block	10x16x24	x	28
Average Pooling (2,2,2)	5x8x12	x	28
Convolutional block	5x8x12	x	56
Convolutional block	6x8x12	x	56
Convolutional block	6x8x12	x	56
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	6x8x12	x	56
Convolutional block	5x8x12	x	56
Convolutional block (T)	5x8x12	x	56
Convolutional block (T)	5x8x12	x	28
Upsampling (2,2,2)	10x16x24	x	28
Convolutional block (T)	12x18x26	x	28
Convolutional block	11x18x25	x	28
Convolutional block (T)	11x18x25	x	14
Upsampling (1,1,2)	11x18x50	x	14
Convolutional block (T)	11x18x50	x	14
Convolutional block (T)	11x18x50	x	14
Convolutional block (T)	11x18x50	x	14
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 750921

Table A.8.: Network architecture of the *triple variation* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	14
Convolutional block	11x18x50	x	14
Convolutional block	11x18x50	x	14
Average Pooling (1,1,2)	11x18x25	x	14
Convolutional block	11x18x25	x	28
Convolutional block	10x16x24	x	28
Convolutional block	10x16x24	x	28
Average Pooling (2,2,2)	5x8x12	x	28
Convolutional block	5x8x12	x	56
Convolutional block	6x8x12	x	56
Convolutional block	6x8x12	x	56
Convolutional block	4x6x10	x	64
Average Pooling (2,2,2)	2x3x5	x	64 (1920)
Upsampling (2,2,2)	4x6x10	x	64
Convolutional block (T)	6x8x12	x	50
Convolutional block	5x8x12	x	50
Convolutional block (T)	5x8x12	x	50
Convolutional block (T)	5x8x12	x	50
Upsampling (2,2,2)	10x16x24	x	50
Convolutional block (T)	12x18x26	x	25
Convolutional block	11x18x25	x	25
Convolutional block (T)	11x18x25	x	25
Upsampling (1,1,2)	11x18x50	x	25
Convolutional block (T)	11x18x50	x	12
Convolutional block (T)	11x18x50	x	12
Convolutional block (T)	11x18x50	x	12
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 745957

A.2. For chapter 5

Table A.9.: Network architecture of the *600-20 filters* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	32
Convolutional block	11x18x50	x	32
Average Pooling (1,1,2)	11x18x25	x	32
Convolutional block	11x18x25	x	60
Convolutional block	10x16x24	x	60
Average Pooling (2,2,2)	5x8x12	x	60
Convolutional block	5x8x12	x	50
Convolutional block	4x6x10	x	50
Average Pooling (2,2,2)	2x3x5	x	50
Convolutional block	2x3x5	x	20
Convolutional block	2x3x5	x	20 (600)
Convolutional block (T)	2x3x5	x	20
Convolutional block (T)	2x3x5	x	50
Upsampling (2,2,2)	4x6x10	x	50
Convolutional block (T)	6x8x12	x	50
Convolutional block	5x8x12	x	60
Upsampling (2,2,2)	10x16x24	x	60
Convolutional block (T)	12x18x26	x	60
Convolutional block	11x18x25	x	32
Upsampling (1,1,2)	11x18x50	x	32
Convolutional block (T)	11x18x50	x	32
Convolutional block (T)	11x18x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 755841

Table A.10.: Network architecture of the *600-50 filters* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	32
Convolutional block	11x18x50	x	32
Average Pooling (1,1,2)	11x18x25	x	32
Convolutional block	11x18x25	x	39
Convolutional block	10x16x24	x	39
Average Pooling (2,2,2)	5x8x12	x	39
Convolutional block	5x8x12	x	50
Convolutional block	4x6x10	x	50
Average Pooling (2,2,2)	2x3x5	x	50
Convolutional block	2x4x6	x	50
Convolutional block	2x4x6	x	50
Average Pooling (1,2,2)	2x2x3	x	50 (600)
Upsampling (1,2,2)	2x4x6	x	50
Convolutional block (T)	2x4x6	x	50
Convolutional block	2x3x5	x	50
Upsampling (2,2,2)	4x6x10	x	50
Convolutional block (T)	6x8x12	x	50
Convolutional block	5x8x12	x	39
Upsampling (2,2,2)	10x16x24	x	39
Convolutional block (T)	12x18x26	x	39
Convolutional block	11x18x25	x	32
Upsampling (1,1,2)	11x18x50	x	32
Convolutional block (T)	11x18x50	x	32
Convolutional block (T)	11x18x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 744999

Table A.11.: Network architecture of the *600-75 filters* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	28
Convolutional block	11x18x46	x	28
Average Pooling (1,1,2)	11x18x23	x	28
Convolutional block	11x18x23	x	29
Convolutional block	10x16x22	x	29
Average Pooling (2,2,2)	5x8x11	x	29
Convolutional block	5x8x9	x	38
Convolutional block	4x6x8	x	38
Average Pooling (2,2,2)	2x3x4	x	38
Convolutional block	2x4x4	x	75
Convolutional block	2x4x4	x	75
Average Pooling (1,2,2)	2x2x2	x	75 (600)
Upsampling (1,2,2)	2x4x4	x	75
Convolutional block (T)	2x4x4	x	75
Convolutional block	2x3x4	x	38
Upsampling (2,2,2)	4x6x8	x	38
Convolutional block (T)	6x8x10	x	38
Convolutional block	5x8x11	x	29
Upsampling (2,2,2)	10x16x22	x	29
Convolutional block (T)	12x18x24	x	29
Convolutional block	11x18x23	x	28
Upsampling (1,1,2)	11x18x46	x	28
Convolutional block (T)	13x20x48	x	28
Convolutional block (T)	13x20x50	x	28
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 749951

Table A.12.: Network architecture of the *600-75 filters, 15 layers* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	28
Average Pooling (1,1,2)	11x18x24	x	28
Convolutional block	11x18x24	x	29
Convolutional block	10x16x23	x	29
Average Pooling (2,2,2)	5x8x11	x	29
Convolutional block	5x8x9	x	40
Convolutional block	4x6x8	x	40
Average Pooling (2,2,2)	2x3x4	x	40
Convolutional block	2x4x4	x	75
Convolutional block	2x4x4	x	75
Average Pooling (1,2,2)	2x2x2	x	75 (600)
Upsampling (1,2,2)	2x4x4	x	75
Convolutional block (T)	2x4x4	x	75
Convolutional block	2x3x4	x	40
Upsampling (2,2,2)	4x6x8	x	40
Convolutional block (T)	6x8x10	x	40
Convolutional block	5x8x11	x	29
Upsampling (2,2,2)	10x16x22	x	29
Convolutional block (T)	12x18x24	x	29
Convolutional block	11x18x23	x	28
Upsampling (1,1,2)	11x18x46	x	28
Convolutional block (T)	13x20x48	x	28
Convolutional block (T)	13x20x50	x	28
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 748399

Table A.13.: Network architecture of the *model-200* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	32
Average Pooling (1,1,2)	11x18x24	x	32
Convolutional block	11x18x24	x	51
Convolutional block	10x16x23	x	51
Average Pooling (2,2,2)	5x8x11	x	51
Convolutional block	5x8x9	x	50
Convolutional block	4x6x8	x	50
Average Pooling (2,2,2)	2x3x4	x	50
Convolutional block	2x4x4	x	50
Convolutional block	2x4x4	x	25
Convolutional block	2x4x4	x	25
Average Pooling (1,2,2)	2x2x2	x	25 (200)
Upsampling (1,2,2)	2x4x4	x	25
Convolutional block (T)	2x4x4	x	25
Convolutional block	2x3x4	x	25
Convolutional block (T)	2x3x4	x	50
Upsampling (2,2,2)	4x6x8	x	50
Convolutional block (T)	6x8x10	x	50
Convolutional block	5x8x11	x	51
Upsampling (2,2,2)	10x16x22	x	51
Convolutional block (T)	12x18x24	x	51
Convolutional block	11x18x23	x	32
Upsampling (1,1,2)	11x18x46	x	32
Convolutional block (T)	13x20x48	x	32
Convolutional block (T)	13x20x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 744464

Table A.14.: Network architecture of the *model-200 dense* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x50	x	32
Average Pooling (1,1,2)	11x18x25	x	32
Convolutional block	11x18x25	x	32
Convolutional block	10x16x24	x	32
Average Pooling (2,2,2)	5x8x12	x	32
Convolutional block	5x8x12	x	42
Convolutional block	4x6x10	x	42
Average Pooling (2,2,2)	2x3x5	x	42
Convolutional block	2x4x6	x	44
Convolutional block	2x4x6	x	44
Average Pooling (1,2,2)	2x2x3	x	44
Flatten	528		
Dense Block	200		
Dense Block	528		
Reshape	2x2x3	x	44
Upsampling (1,2,2)	2x4x6	x	44
Convolutional block (T)	2x4x6	x	44
Convolutional block	2x3x5	x	42
Upsampling (2,2,2)	4x6x10	x	42
Convolutional block (T)	6x8x12	x	42
Convolutional block	5x8x12	x	32
Upsampling (2,2,2)	10x16x24	x	32
Convolutional block (T)	12x18x26	x	32
Convolutional block	11x18x25	x	32
Upsampling (1,1,2)	11x18x50	x	32
Convolutional block (T)	11x18x50	x	32
Convolutional block (T)	11x18x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 752721

Table A.15.: Network architecture of the *model-64* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	32
Average Pooling (1,1,2)	11x18x24	x	32
Convolutional block	11x18x24	x	40
Convolutional block	10x16x23	x	40
Average Pooling (2,2,2)	5x8x11	x	40
Convolutional block	5x8x9	x	44
Convolutional block	4x6x8	x	44
Average Pooling (2,2,2)	2x3x4	x	44
Convolutional block	2x4x4	x	46
Convolutional block	2x4x4	x	46
Average Pooling (1,2,2)	2x2x2	x	46
Convolutional block	2x2x2	x	64
Convolutional block	2x2x2	x	64
Average Pooling (2,2,2)	1x1x1	x	64 (64)
Upsampling (2,2,2)	2x2x2	x	64
Convolutional block (T)	2x2x2	x	64
Convolutional block (T)	2x2x2	x	46
Upsampling (1,2,2)	2x4x4	x	46
Convolutional block	2x3x4	x	46
Convolutional block (T)	2x3x4	x	44
Upsampling (2,2,2)	4x6x8	x	44
Convolutional block (T)	6x8x10	x	44
Convolutional block	5x8x11	x	40
Upsampling (2,2,2)	10x16x22	x	40
Convolutional block (T)	12x18x24	x	40
Convolutional block	11x18x23	x	32
Upsampling (1,1,2)	11x18x46	x	32
Convolutional block (T)	13x20x48	x	32
Convolutional block (T)	13x20x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 749177

Table A.16.: Network architecture of the *model-32* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	32
Average Pooling (1,1,2)	11x18x24	x	32
Convolutional block	11x18x24	x	32
Convolutional block	10x16x23	x	32
Average Pooling (2,2,2)	5x8x11	x	32
Convolutional block	5x8x9	x	60
Convolutional block	4x6x8	x	60
Average Pooling (2,2,2)	2x3x4	x	60
Convolutional block	2x4x4	x	44
Convolutional block	2x4x4	x	44
Average Pooling (1,2,2)	2x2x2	x	44
Convolutional block	2x2x2	x	32
Convolutional block	2x2x2	x	32
Average Pooling (2,2,2)	1x1x1	x	32 (32)
Upsampling (2,2,2)	2x2x2	x	32
Convolutional block (T)	2x2x2	x	32
Convolutional block (T)	2x2x2	x	44
Upsampling (1,2,2)	2x4x4	x	44
Convolutional block	2x3x4	x	44
Convolutional block (T)	2x3x4	x	60
Upsampling (2,2,2)	4x6x8	x	60
Convolutional block (T)	6x8x10	x	60
Convolutional block	5x8x11	x	32
Upsampling (2,2,2)	10x16x22	x	32
Convolutional block (T)	12x18x24	x	32
Convolutional block	11x18x23	x	32
Upsampling (1,1,2)	11x18x46	x	32
Convolutional block (T)	13x20x48	x	32
Convolutional block (T)	13x20x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 752417

Table A.17.: Network architecture of the *model-200 wide* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	64
Average Pooling (1,1,2)	11x18x24	x	64
Convolutional block	11x18x24	x	75
Convolutional block	10x16x23	x	75
Average Pooling (2,2,2)	5x8x11	x	75
Convolutional block	5x8x9	x	75
Convolutional block	4x6x8	x	75
Average Pooling (2,2,2)	2x3x4	x	75
Convolutional block	2x4x4	x	75
Convolutional block	2x4x4	x	25
Convolutional block	2x4x4	x	25
Average Pooling (1,2,2)	2x2x2	x	25 (200)
Upsampling (1,2,2)	2x4x4	x	25
Convolutional block (T)	2x4x4	x	25
Convolutional block	2x3x4	x	25
Convolutional block (T)	2x3x4	x	75
Upsampling (2,2,2)	4x6x8	x	75
Convolutional block (T)	6x8x10	x	75
Convolutional block	5x8x11	x	75
Upsampling (2,2,2)	10x16x22	x	75
Convolutional block (T)	12x18x24	x	75
Convolutional block	11x18x23	x	64
Upsampling (1,1,2)	11x18x46	x	64
Convolutional block (T)	13x20x48	x	64
Convolutional block (T)	13x20x50	x	64
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 1699239

Table A.18.: Network architecture of the *model-200 deep* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	32
Convolutional block	11x18x48	x	32
Average Pooling (1,1,2)	11x18x24	x	32
Convolutional block	11x18x24	x	51
Convolutional block	10x16x23	x	51
Convolutional block	10x16x23	x	51
Convolutional block	10x16x23	x	51
Average Pooling (2,2,2)	5x8x11	x	51
Convolutional block	5x8x9	x	50
Convolutional block	4x6x8	x	50
Convolutional block	4x6x8	x	50
Convolutional block	4x6x8	x	50
Average Pooling (2,2,2)	2x3x4	x	50
Convolutional block	2x4x4	x	50
Convolutional block	2x4x4	x	25
Convolutional block	2x4x4	x	25
Convolutional block	2x4x4	x	25
Average Pooling (1,2,2)	2x2x2	x	25 (200)
Upsampling (1,2,2)	2x4x4	x	25
Convolutional block (T)	2x4x4	x	25
Convolutional block	2x3x4	x	25
Convolutional block (T)	2x3x4	x	25
Convolutional block (T)	2x3x4	x	25
Upsampling (2,2,2)	4x6x8	x	25
Convolutional block (T)	6x8x10	x	50
Convolutional block	5x8x11	x	50
Convolutional block (T)	5x8x11	x	50
Convolutional block (T)	5x8x11	x	51
Upsampling (2,2,2)	10x16x22	x	51
Convolutional block (T)	12x18x24	x	51
Convolutional block	11x18x23	x	51
Convolutional block (T)	11x18x23	x	51
Convolutional block (T)	11x18x23	x	32
Upsampling (1,1,2)	11x18x46	x	32
Convolutional block (T)	13x20x48	x	32
Convolutional block (T)	13x20x50	x	32
Convolutional block (T)	11x18x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 1334779

Table A.19.: Network architecture of the *model-200 small* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	32
Average Pooling (1,1,2)	11x18x24	x	32
Convolutional block	11x18x24	x	32
Convolutional block	10x16x23	x	32
Average Pooling (2,2,2)	5x8x11	x	32
Convolutional block	5x8x9	x	25
Convolutional block	4x6x8	x	25
Average Pooling (2,2,2)	2x3x4	x	25
Convolutional block	2x4x4	x	25
Convolutional block	2x4x4	x	25
Convolutional block	2x4x4	x	25
Average Pooling (1,2,2)	2x2x2	x	25 (200)
Upsampling (1,2,2)	2x4x4	x	25
Convolutional block (T)	2x4x4	x	25
Convolutional block	2x3x4	x	25
Convolutional block (T)	2x3x4	x	25
Upsampling (2,2,2)	4x6x8	x	25
Convolutional block (T)	6x8x10	x	25
Convolutional block	5x8x11	x	32
Upsampling (2,2,2)	10x16x22	x	32
Convolutional block (T)	12x18x24	x	32
Convolutional block	11x18x23	x	32
Upsampling (1,1,2)	11x18x46	x	32
Convolutional block (T)	13x20x48	x	32
Convolutional block (T)	13x20x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 345947

Table A.20.: Network architecture of the *model-200 shallower* autoencoder.

Building block	Output dimension		
Input	11x18x50	x	1
Convolutional block	11x18x48	x	32
Average Pooling (1,1,2)	11x18x24	x	32
Convolutional block	10x16x22	x	51
Average Pooling (2,2,2)	5x8x11	x	51
Convolutional block	5x8x9	x	50
Convolutional block	4x6x8	x	50
Average Pooling (2,2,2)	2x3x4	x	50
Convolutional block	2x4x4	x	25
Convolutional block	2x4x4	x	25
Average Pooling (1,2,2)	2x2x2	x	25 (200)
Upsampling (1,2,2)	2x4x4	x	25
Convolutional block	2x3x4	x	25
Convolutional block (T)	2x3x4	x	50
Upsampling (2,2,2)	4x6x8	x	50
Convolutional block (T)	6x8x10	x	50
Convolutional block	5x8x11	x	51
Upsampling (2,2,2)	10x16x22	x	51
Convolutional block (T)	12x18x24	x	32
Upsampling (1,1,2)	11x18x48	x	32
Convolutional block (T)	13x20x50	x	32
Convolution (kernel size 1, linear)	11x18x50	x	1

Trainable parameters: 491569

Statutory Declaration

I declare that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such. This thesis was not submitted in the same or in a substantially similar version, not even partially, to any other authority to achieve an academic grading and was not published elsewhere.

.....

Stefan Reck Erlangen, September 25, 2018