Development of an intelligent trigger system based on deep neural networks

Luca Rigon, Advanced Physics Project Course (10 ECTS), Uppsala University

Spring 2021

Abstract

The ARIANNA experiment is project in the Antarctic with the goal of detecting Askaryan radio signals induced by the interaction between high-energetic neutrinos and the Antarctic ice. Because of the low flux of high-energy neutrinos, and the fact that they barely interact with matter, the number of detectable events is small. To maximise sensitivity, the trigger threshold is reduced as much as possible until triggers on mostly thermal noise saturate the detector readout. Hence, by raising the sensitivity of the detector, in order to register more neutrino signals, we will also unavoidably trigger many more thermal noise events. Our goal is to use deep learning techniques to create neural networks capable of rejecting a remarkable fraction of the noise in real time, while retaining most of the neutrino signals. This would allow us to lower the event trigger thresholds even more and double the sensitivity compared to the current ARIANNA capabilities. Here we describe different types of generated networks, and try to find the most suited one in terms of size, efficiency, and speed. We report on the efficiency of the networks, and on their event prediction rate, by running inference tests on an external coprocessor. We manage to find two particularly efficient networks, a Fully Connected Network (FCN) and a Convolutional Network (CNN), capable of retaining a signal efficiency of 98.4% and 99.5% at a thermal rejection factor of 10^4, respectively, while maintaining an event prediction rate in the kHz-range.

Contents

1 Introduction

1.1 Neutrino astronomy

Extreme-high-energy neutrino astronomy offers us a very powerful tool to help us solve the mysteries and challenges of extremely energetic cosmic rays. It is still unknown how these particles can be accelerated up to energies of 10^{20} eV, and since they are charged, they get deflected by all kinds of intergalactic magnetic fields, making it extremely hard to identify the regions where they have been accelerated up to those high energies, once they reach us on Earth [1]. The observation of neutrinos could then be very helpful in identifying those regions, since the charged cosmic rays produce high energy neutrinos (astrophysical neutrinos) and γ-rays by interacting with gas, dust or radiation.

Since neutrinos have negligible mass, are neutral in charge, and have a low interaction probability, they are capable of escaping the dense environments in which the cosmic rays get accelerated, and travel through space at the speed of light, without being hindered by objects such as the Cosmic Microwave Background (CMB) or electromagnetic fields. This makes them perfect source-detection candidates. But for the same reasons, neutrinos also hardly interact with Earth, making them very challenging to detect; detectors need to cover large volumes in order to increase the target material.

The most efficient way to measure high-energy neutrinos is made by using radio techniques in ice. When they interact with the ice, the neutrinos scatter and emit Cherenkov radiation through the Askaryan effect. More precisely, high energy processes such as Compton, Bhabha, and Moller scattering, along with positron annihilation, rapidly lead to a $\sim$20% negative charge asymmetry in the electron-photon part of the shower. This then creates a changing current that produces radio emissions in the MHz-GHz range. These radio emissions then propagate through the ice without interacting with it and can be detected. The medium in which this happens is very important, and ice is best suited for that [2]. Since we need large volumes of ice, the Antarctic and Greenland are the most suited places to perform those experiments.

1.2 The ARIANNA experiment

The primary mission of the ARIANNA ultra-high energy neutrino telescope is to uncover astrophysical sources of neutrinos with energies greater than 10^{16} eV. It is an array of Askaryan detectors designed to record radio signals induced by the neutrino interactions in the ice, and is located in Moore's Bay on the Ross Ice Shelf, with two additional demonstrator stations located at the South Pole. Each ARIANNA station is designed to operate autonomously, with self-contained (solar) power; for the radio detection, each station is composed by four log-periodic dipole antennas (LPDA's), buried face-down just below the snow level. The data acquisition (DAq) is made with a Synchronous Sampling plus Triggering (SST) chip, which is equipped of a coincidence trigger helping to reduce the acquisition of thermal noise. In order for an event to be triggered and saved onto an SD card, the high-low threshold must be met by every channel within 30 ns of one another.

Since all the data gets transferred through satellite communication, which is slow and unreliable, we have a constraint on the amount of the data we want to collect and transfer. Hence, we want to

collect as least thermal noise as possible, so we need to keep a trigger threshold which is high enough in order to be able to collect a good amount of useful data. For now, the trigger rate is hold at about 10^{-3} Hz, corresponding to a trigger threshold at about four times the RMS noise. Unfortunately, by keeping a high threshold, we also limit the observations of actual neutrino signals, that are lower than the threshold [1].

Our goal is to lower the trigger threshold even more, in order to increase the trigger rates and the detection of neutrino signals, but by still trying to keep the rejection of the thermal noise as high as possible (in order to increase the sensibility). We want to achieve this by creating small neural networks, trained with simulated data, which have to distinguish between noise and actual neutrino signals. We will then investigate how efficient, and how fast, they are when computing their predictions, and if there are any ways to optimize these networks.

1.3 Deep learning in neural networks

As already mentioned, one way to solve this problem is by using deep learning techniques: deep learning is a branch of machine learning, in which we create neural networks, with the analogy of a neural system (brain). These networks are made out of many nodes connected to each other, through different layers, capable of learning and recognizing features within a data set. These networks firstly have to be trained with some representative data, and they will create an associated matrix of weights and biases with every learned feature in the data set. The weights of the nodes are calculated based on all the weights of the previous nodes and an activation function, which will output a new weight in the network. We will be using two types of activation functions: ReLU, which is a semi-linear function that gets activated if the weights are higher than a certain threshold; and the 'softmax'-function, which is a sigmoid function ($f(x) = \frac{1}{1+e^{-x}}$) that outputs a value between 0 and 1 [3] (See Fig.1). We can choose the size of the neural networks by choosing the amount of input nodes, layers, and layer node size, according to which the weight matrix will scale. Consequently, "bigger" networks will require more computations.

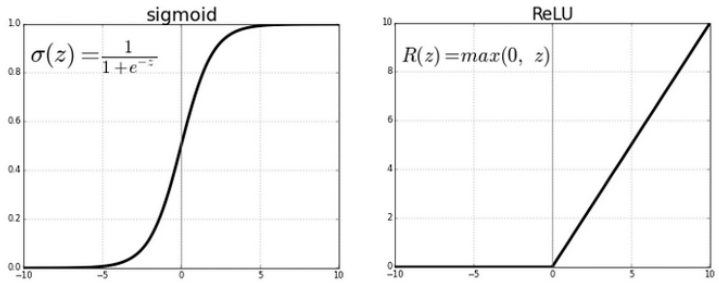


Figure 1: The two types of used activation functions: softmax (left) and ReLU (right) [4]

After the training, we can test and classify incoming data (which has to be different to the training data), and see if the network is capable of recognizing the previously found features in the data, and make the correct predictions. We can do this by interpreting the output of the activation functions as a probability distribution for the given classification problem. In our case, we want to classify between signal and noise; given a certain event (signal or noise), we want our network to predict its nature by outputting a probability: a probability close to 0 means that our event is noise-like, a probability close to 1 gives us a signal-like event. We will test the efficiency of the predictions of such networks with the help of some simulated data (more about it in Section 2), by comparing the predicted probabilities for the generated events, with their actual nature (by labeling the data with a 0 or a 1, depending if we are testing noise or signals).

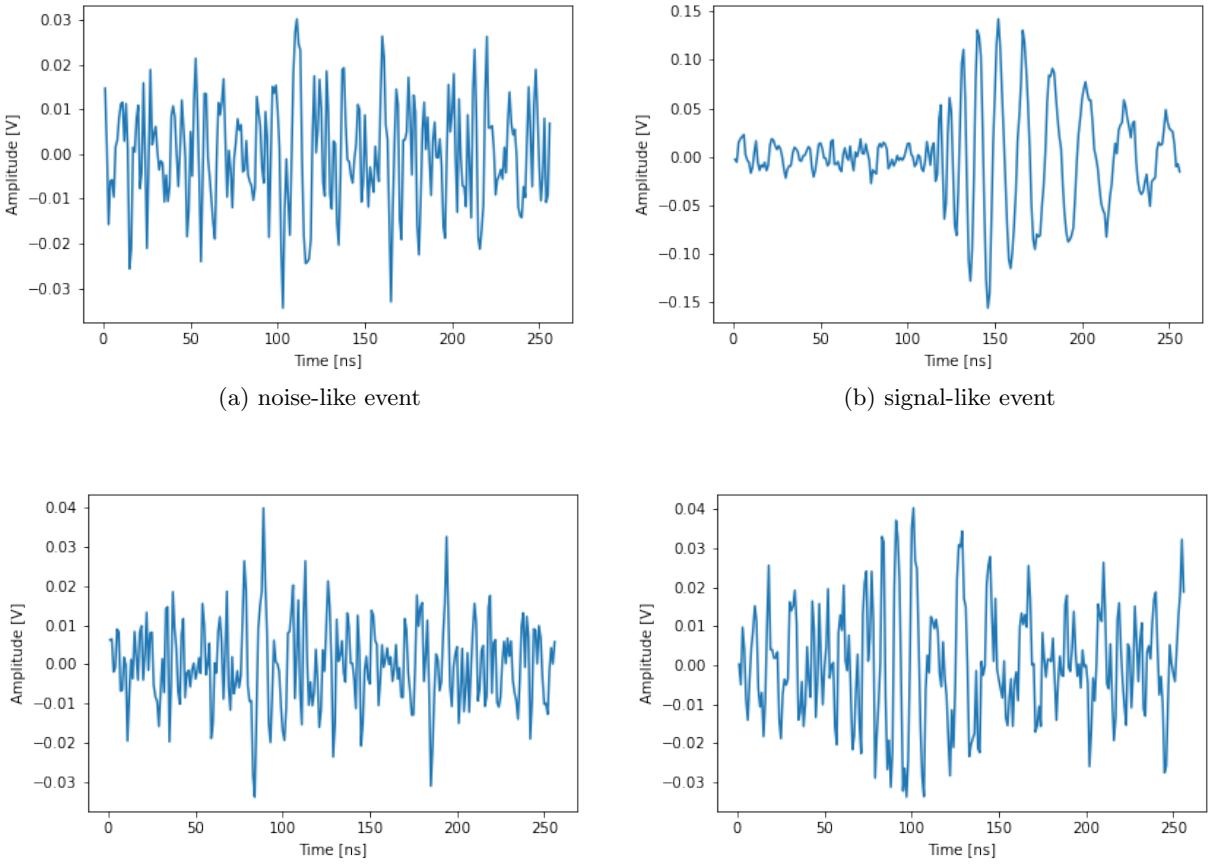
Implementing such a deep learning filter into the ARIANNA detectors could be very useful, since we would be able to filter out most thermal noise events and highly increase the sensitivity to neutrino candidates.

2 Data Processing

In this section we will introduce the data sets we are working with and how they look like. We will also discuss more in detail the type of neural networks we are generating, their structural differences and how they process the data.

2.1 Data-set

We are training and testing our neural networks on a total of 1.1 million events, all generated through some Monte Carlo simulations using NuRadioMC [5], with varying signal-to-noise ratio, up to 3.0: we have 100'000 neutrino signal-like events, and one million of noise-like events. Every event has been recorded in one channel (simulating the antenna/channel at the station which registered the largest signal, out of four), and has a trace of 256 samples, in steps of 1 ns, meaning that every signal has been registered for 256 seconds. In the end, the data we are feeding to the networks has a shape of (*batch_size*, 1, 1), where *batch_size* is the number of events we feed at once; in reality, *batch_size* = 1, since the signals that are registered arrive one after the other, and not at the exact same time. In Fig. 2 we can see some representative examples of the traces of our simulated data, and their different features.



(a) noise-like event

(b) signal-like event

(c) noise-like event

(d) signal-like event

Figure 2: Four examples of voltage vs. time traces (256 ns) for the simulated data: two are noise-like, were (c) is one with the largest amplitude, and two are signals. For certain signal events, which have a higher SNR-ratio, the different features between random thermal noise and a neutrino signal can easily be found (e.g. by comparing subfigure a with subfigure b); other signals, instead, get quite mixed up with noise at similar amplitudes, making it hard to detect the signal.

2.2 Neural Networks and their architectures

We create all our networks with Keras [6], a library of the machine-learning platform Tensorflow. We train them all with the same data-subset of 75'000 signal- and noise-events; the remaining 9.5×10^5 events are used for testing/validation.

We consider and train two different types of networks. The first type are fully connected neural networks (FCN - Fig.s 3,4); they are made out of so-called 'Dense' layers, the nodes of which are connected to every single node in the following layer. We test different networks, going from one to four dense layers, with varying dimension (between 16 and 256). With dimension we mean the node size, i.e. the number of nodes for each layer. The output layer is always a two-dimensional dense layer with a sigmoid activation function, since we want a binary output: x and $1 - x$, where $x \in [0, 1]$.

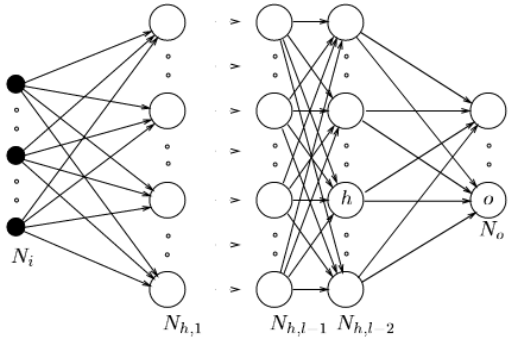


Figure 3: Example of FCN: it has N_i input nodes, and l layers of units [3].

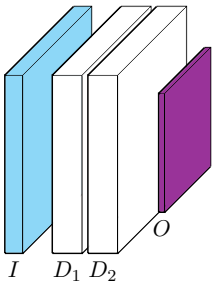


Figure 4: Baseline architecture for our FCNs: input layer I, variable number of hidden dense layers D_i (in this case we have two) of variable dimension with ReLU activation functions , and output layer O, of dimension 2, with a 'softmax' activation function (sigmoid output)

The other type of network we want to use are convolutional neural networks (CNN - Fig.5). These are non-linear networks mostly used for image classification tasks, since they are able to recognize features and visual patterns directly from raw data with little-to-none preprocessing [7]. They are made out of convolutional filters, which can be one- or two-dimensional; in our case, since we want to find features in a time series, a 1D-convolutional layer is sufficient. They have a kernel of a certain size, and a certain number of filters through which the features are passed. The structure of our CNN's consists in either one or two 1D-convolutional layers, with varying number of filters and kernel size; then we have a Dropout filter of 0.5, which helps to prevent overfitting in our model [7], and then a 'Flatten'-layer which reshapes the dimensions of our output tensor in order to make it compatible with the final output Dense-layer.

The exact architectures of all the trained networks can be seen in the Appendix B.

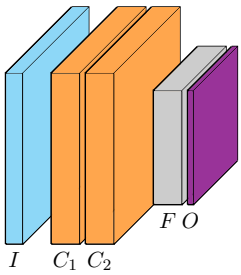


Figure 5: Baseline architecture for our CNNs: input layer I, one or two convolutional layers C_i (in this case we have two) with a variable number of filters and kernel size, and ReLU activation functions; the layers have then to be 'Flattened and made compatible with the dense output layer O (dimension 2, 'softmax' activation function)

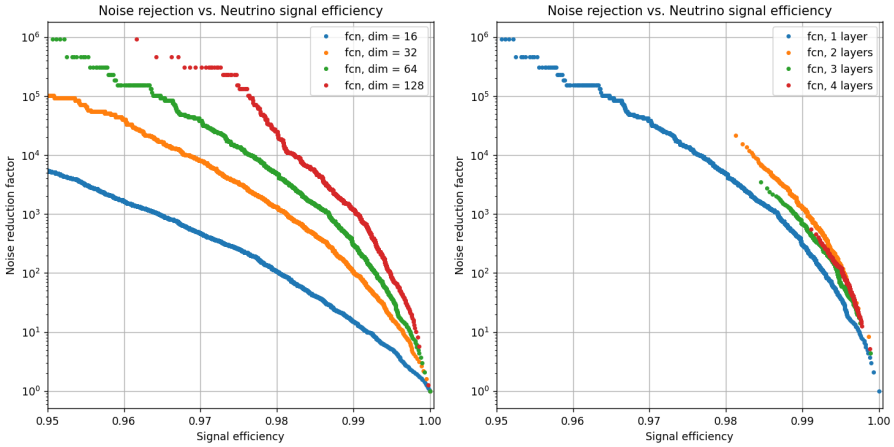
3 Background rejection vs. Signal efficiency

In order for it to be worth to implement a deep learning filter, the network we are looking for must be optimized for fast and accurate classification, meaning that it must be able to reject as much noise as possible, without sacrificing a significant amount of signal efficiency (i.e. the ratio of signal-like events that have correctly been classified as signal, w.r.t. the total number of signal-events). Ideally, it would be able to reach 4 orders of magnitude background rejection while providing a signal efficiency at above 95%. This would allow us to increase the actual processing rate from mHz to 100 Hz at the ARIANNA station. Additionally, we would also be able to reduce the trigger threshold significantly, which would increase the neutrino sensitivity. The noise rejection is calculated by finding the ratio of noise-like events that have been correctly classified as noise, w.r.t. the total number of noise-events; we then take the value $\frac{1}{1-\text{ratio}_{\text{noise}}}$ as order of magnitude for the noise-rejection.

In order to find the efficiency of a model, we can set a threshold value going from 0 to 1, in 10'000 steps, and calculate the noise-rejection and the signal efficiency for every threshold (meaning that all the predicted probabilities above the threshold value are considered as signal, and all below are considered as noise). Logically, for threshold = 0, all the outputs will be considered as signals, giving us a signal efficiency of 100%, meaning that we are also taking in all the noise (no noise rejection). For threshold = 1, all the events will be considered noise and be rejected (noise rejection of 9.5×10^5; all the input noise).

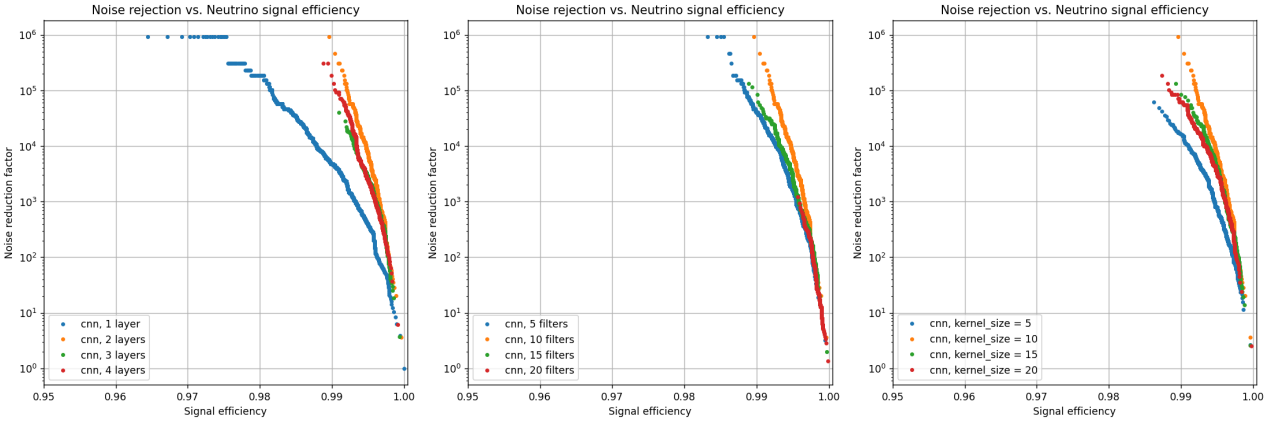
By comparing the efficiencies of the FCNs in Fig. 6, we find that networks with more parameters work better, but there is a limit in the improvement. For example, we can see that by adding more hidden layers we do not get better results. From this we find that the best networks are the following two: 1 hidden layer, 128 nodes (signal efficiency of 98.2% at a 10^4-noise rejection), and 2 hidden layers, 64 nodes each (signal efficiency of 98.4% at a 10^4-noise rejection).

We repeat the same process for the CNNs, where the changeable parameters are the number of convolutional layers, the filters for each layer, and the dimension of the kernel (Fig. 7). In this case, the network performing best is the network with two convolutional layers, with 10 filters and kernel dimension = 10 for each layer (yellow curve in the three plots). For this network, we get a signal efficiency of 99.5% at a 10^4-noise rejection. From this, we can already see that we are efficiently able to distinguish thermal noise from neutrino signals, even with small networks.



(a) FCN, 1 hidden layer, different node size(b) FCN, dim = 64, different nr. of hidden layers

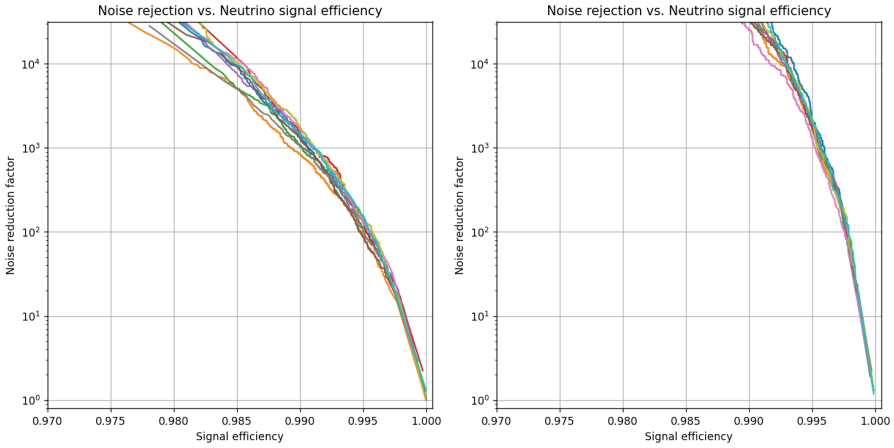
Figure 6: Compare different Fully Connected Networks by plotting their efficiency curve with respect to some variable parameters; number of nodes/dimension per layer (a. - Network structures in Fig. 14), and number of hidden layers, for a fixed dimension (b. - See the Network structures of Fig.s 14a, 16a, 16b, 15c, in the same order).



(a) CNN, 10 filters, kernel size = 10, different nr. of layers (b) CNN, 2 layers, kernel size = 10, different number of filters (c) CNN, 2 layers, 10 filters, different kernel sizes

Figure 7: Compare Convolutional Neural Networks by plotting their efficiency curve with respect to some variable parameters; number of convolutional layers ((a)), number of filters per layer ((b)), dimension of the kernel for each layer ((c)). See architectures in Fig.s 18, 19.

In Fig. 8 we can see the stability plots for the two chosen networks; we trained the same type of network 10 times with the same training data, in order to see by how much the prediction accuracy varies. At a noise rejection factor of 10^4, we get an average signal efficiency of $(98.3 \pm 0.2)\%$ for the fully connected network, and of $(99.3 \pm 0.1)\%$ for the convolutional network. The errors have been calculated by taking the standard deviation of our measurements.



(a) Stability FCN (b) Stability CNN

Figure 8: Stability-plots of the predictions of the chosen fully connected network (a. 2 layers, 64 nodes) and the convolutional network (b. 2 layers, 10 filters, kernel dim = 10). The same network has been created ten times and trained with the same set of data, in order to compare by how much the efficiencies of the same network vary. The points have been drawn as lines in order to prevent confusion, allowing to track the single efficiency curves easier.

4 Runtime Optimizations

Unfortunately, the efficiency of a network is not the only parameter we are trying to optimize; if it was, we could just create some very large networks in order to obtain extremely good results. But the device installed in ARIANNA (a MBED LPC 1768 ARM microprocessor) only has very limited power (the whole station only uses 5 W) and computing resources; because of this, we have to adapt our deep learning for small/mobile devices, which have these limitations. We cannot only consider how good our model's predictions are anymore, but we also have to take into account ([8]):

- The amount of **space** the model takes up on the device: by keeping the generated networks small, they only take up around a few hundred's kb of space;
- The amount of **memory** the models take up at runtime, since we only have a few GB of RAM available;
- How **fast** our models run, since we want to make real-time predictions and immediately reject the noise in order not to save useless data. Also, we want to increase the processing rate of the station, so slow networks would not be useful for our final goal;
- How quickly the models drain the **battery** or heat up the device - our device at the South Pole is run by solar panels, and wind generators in the winter, which can't always be reliable.

In this section we will discuss a way of measuring the speed, and accordingly the size, of the generated neural networks, giving us a way to directly compare them between each other; we then want to discuss a way of optimizing the models in order to make them run faster without loosing too much accuracy; this method is called *model quantization*.

4.1 Speed of the models: Floating Point Operations Per Second

One way to get an indication on how fast a model is, is to simply count the number of computations it does. We count them as FLOPS, or FLoating point Operations Per Second: they are simply the number of operations between floating points (which are the weights and bias values of the networks)

that our model has to compute in one second. When calculating FLOPS, we usually count addition, subtraction, multiplication, division, exponentiation, square root, etc as a single FLOP. Logically, bigger networks have more weight and bias points, hence they have to make more computations and take more computation time in general.

It is important to note that counting the number of computations in a network is useful only to get a very rough idea of what its computational cost is, but other factors such as memory bandwidth are often more important when considering the speed of the models [8].

The best way to measure the speed of a model is to run it a number of times in a row and take the average elapsed time; we will go more into time and inference measurements in section 5.

4.1.1 Fully connected layer

Here we describe the mathematical way of calculation the flops of neural netwrok layers; there is an implemented TensorFlow - keras function which calculates them for us. We consider a fully connected layer with I input values and J output values; its weights W are stored in an I $\times$ J matrix. The computation performed by this layer is a simple matrix product, meaning that we have J dot-products between the input vector x of length I and the weight matrix, with the vector of bias values b (length J) which is added in the end:

$$y[j] = w[1,j] \cdot x[1] + w[2,j] \cdot x[2] + w[3,j] \cdot x[3] + ... + w[I,j] \cdot x[I] + b[j]; \tag{1}$$

y is the output vector of length J ($i \in [1, \mathrm{I}]$, $j \in [1, \mathrm{J}]$). In one dot product we have I multiplications and I - 1 additions, giving us 2I - 1 FLOPS. By also considering the bias vector (+J operations), one fully connected layer takes up 2I $\times$ J FLOPS.

Knowing the architecture of the generated fully connected networks, it is easy to calculate their number of FLOPS; they can be seen in Table 1.

Also the implemented activation functions take up some computation costs; but it is common not to count these operations, as they only take up a small fraction of the overall time.

network	1l, 16 nodes	4l, 16 nodes	1l, 32 nodes	4l, 32 nodes	1l, 64 nodes	4l, 64 nodes
FLOPS	8259	18057	34572	57234	90261	147867
network	1l, 128 nodes	4l, 128 nodes	1l, 256 nodes	4l, 200 nodes	4l, 256 nodes	4l, 350 nodes
FLOPS	213918	378276	510375	853581	1378899	2294505

Table 1: Calculated total number of floating point operations for the generated fully connected networks, increasing in size. See architectures in Appendix B.

4.1.2 Convolutional layer

The input and output to convolutional layers are not vectors but three-dimensional feature maps of size H $\times$ W $\times$ C, where H is the height of the feature map, W the width, and C the number of channels at each location. In the model architecture, the output shape (H, W, C) is shown. For a convolutional layer with a $k_1 \times k_2$ kernel, the number of FLOPS is roughly

$$2 \times k_1 \times k_2 \times C_{in} \times H_{out} \times W_{out} \times C_{out}, \tag{2}$$

where we have an output feature map of size $H_{out} \times W_{out}$, and a dot product of the weights with a $k_1 \times k_2$ window of input values across all input channels C_{in}; because the layer has C_{out} different convolution kernels, we repeat this C_{out} times to create all the output channels. We multiply all by a factor of two in order to consider approximately all the additions that are present inside the dot products.

Since the convolutional networks we trained are always followed by a fully-connected output layer, we also have to consider the final operation as a dense layer operation for the computation costs; the

output feature map of the final convolutional layer is "flattened" into a vector, and then treated as a normal matrix operation, with I = H × W × C. The FLOPS of the trained convolutional networks can be seen in Table 3.

network, 1 layer	$k = 1 \times 1$, 5 filters	$k = 5 \times 1$, 5 filters	$k = 5 \times 1$, 10 filters	$k = 5 \times 1$, 15 filters	$k = 10 \times 1$, 15 filters
FLOPS	8972	18912	37812	56712	92637
network, 2 layers	$k = 10 \times 1$, 5 filters	$k = 5 \times 1$, 10 filters	$k = 10 \times 1$, 10 filters	$k = 20 \times 1$, 10 filters	$k = 10 \times 1$, 20 filters
FLOPS	150897	288132	539782	980082	2031552

Table 2: Calculated total number of floating point operations for the generated convolutional networks, increasing in size (the changing parameters are the number of convolutional layers, the kernel dimension k and the number of filters. See architectures in Appendix Figures 17 and 18, in the same order.

4.2 Model Quantization

Until now, the trained neural networks perform computations at floating point precision, meaning that the values of the input data, the weights and biases if the models, and of the outputs are floating points, more precisely of type 'float32' (32-bit single-precision floating-points). In this section we want to discuss the so-called **model quantization** and its mathematical background: it is a technique which allows to reduce the floating point operations onto operations between integers on the tensors of our neural networks. This allows us to store the tensors at lower bitwidths than the precision we had previously. This technique allows for a more compact model representation, and is in particular useful at the inference time since it saves a lot of inference computation costs without sacrificing too much inference accuracies [9]. For this reason, we will also investigate into the quantization of the neural networks, and verify how much inference time we gain, versus how much accuracy we lose, by checking the efficiency plots of the quantized models.

4.2.1 Quantization Mapping

Uniform integer quantization can be divided in two steps: firstly, we choose the range of the real numbers to be quantized, and then we map the real values to integers representable by the bit-width of the quantized representation (every mapped real value is rounded to the closest integer value): for this project, we will consider 8-bit quantization ('int8'), meaning that we will map floating points onto signed integers with a bit-width of 8 (we can distinguish only between $2^8 = 256$ values).

In order to allow integer operations on networks which are pre-trained on floating-points, we need to define two mapping functions: the *quantization* map (convert real numbers to quantized integer representation) and the *dequantization* map (convert a number from to quantized integer representation to a real number) [10].

After choosing the real-valued range $[\beta, \alpha]$ we want to quantize, the uniform quantization maps the input values $x \in [\beta, \alpha]$ into the range $[-2^{b-1}, 2^{b-1} - 1]$, where inputs outside the range are clipped to the nearest bound, and b is the the bit-width of the signed integer representation. In our case, $b = 8$.

For uniform transformations we only have two possibilities: the **affine quantization**, where $f : x \in [\beta, \alpha] \subset \mathbb{R} \longmapsto x_q \in \{-128, -127, ..., 127\}$, $f(x) = s \cdot x + z$, and the **scale quantization**, which is the special case $z = 0$: $f(x) = s \cdot x$. We call s the scale factor, and z the zero-point, and they can be found as follows:

$$s = \frac{2^b - 1}{\alpha - \beta} = \frac{255}{\alpha - \beta} \tag{3}$$

$$z = -\text{round}(\beta \cdot s) - 2^{b-1} = -\text{round}(\beta \cdot s) - 128, \tag{4}$$

where round() rounds to the nearest integer. We need z like this in order to ensure that 0 in floating point is represented exactly with no error after quantization.

In order to make sure to obtain values that are integers inside the defined range, we define the clipping function and quantize operation in the next two equations:

$$\text{clip}(x, l, u) = \begin{cases} l, & x < 0 \\ x, & l \leq x \geq u \\ u, & x > u \end{cases} \tag{5}$$

$$x_q = \text{quantize}(x, s, z) = \text{clip}(\text{round}(s \cdot x + z), -128, 127). \tag{6}$$

Equation 7 shows the corresponding dequantize operation, which computes an approximation of the original real valued input, $\hat{x} \approx x$.

$$\hat{x} = \text{dequantize}(x_q, s, z) = \frac{1}{s}(x_q - z). \tag{7}$$

4.2.2 Quantized matrix multiplication

Now that we know how to quantize single points x, it is possible to expand the idea onto matrices and matrix-multiplication: consider a fully connected layer performing the matrix multiplication $Y = XW + b$, where $X = (x_{ik}) \in \mathbb{R}^{m \times p}$ is the input activation tensor, $W = (w_{kj}) \in \mathbb{R}^{p \times n}$ is the weight tensor, $Y = (y_{ij}) \in \mathbb{R}^{m \times n}$ is the output tensor and $b \in \mathbb{R}^n$ the bias vector. Denote the quantized version of X, W, b with the subscript "q". We can then approximate the output tensor Y by dequantizing the other tensors first, and then performing the multiplication:

$$\begin{aligned} y_{ij} = b_j + \sum_{k=1}^{p} x_{ik} \cdot w_{kj} &\approx \text{dequantize}(b_{q,j}, s_{b,j}, z_b) + \sum_{k=1}^{p} \text{dequantize}(x_{q,ik}, s_{x,ik}, z_x) \cdot \text{dequantize}(w_{q,kj}, s_{w,kj}, z_s) \\ &= \frac{1}{s_{b,j}}(b_{q,j} - z_b) + \sum_{k=1}^{p} \frac{1}{s_{x,ik}}(x_{q,ik} - z_x) \cdot \frac{1}{s_{w,kj}}(w_{q,kj} - z_w) \\ &= \frac{1}{s_{b,j}}(b_{q,j} - z_b) + \sum_{k=1}^{p} \frac{1}{s_{x,ik}} \frac{1}{s_{w,kj}}(x_{q,ik} \cdot w_{q,kj} - z_x w_{q,kj} - z_w x_{q,ik} + z_x z_w). \end{aligned} \tag{8}$$

In order for Y to be quantized onto integer operations, equation 8 must be equivalent to the dequantization operation $y_{ij} = \frac{1}{s_{y,ij}}(y_{q,ij} - z_y)$, which is only possible if the scale factors are independent of the variable k, in order for them to be factored out of the summation:

$$y_{ij} = \frac{1}{s_{b,j}}(b_{q,j} - z_b) + \frac{1}{s_{x,i} \cdot s_{w,j}}(\sum_{k=1}^{p}(x_{q,ik} \cdot w_{q,kj} - z_x w_{q,kj} - z_w x_{q,ik}) + p z_x z_w). \tag{9}$$

There are different choices for sharing quantization parameters among tensor elements. We call this choice *quantization granularity*. The finest granularity has individual quantization parameters per element, then there is the option to reuse parameters over various dimensions of the tensor giving us per row or per column granularity (or per channel for 3D tensors), and finally the per-tensor granularity, meaning that all the elements in the tensor share the same quantization parameters. The granularity can have an impact on the performance on the models, and as already mentioned, in order for the output tensor to be quantizeable, the granularity must be per-row or per-tensor for the input tensors and per-column or per-tensor for weights [10]. Using eq.s 7 and 9 we can compute Y_q (in per-tensor granularity):

$$y_{q,ij} = z_y + \frac{s_y}{s_b}(b_{q,j} - z_b) + \frac{s_y}{s_x \cdot s_w}(\sum_{k=1}^{p}(x_{q,ik} \cdot w_{q,kj} - z_x w_{q,kj} - z_w x_{q,ik}) + p z_x z_w). \tag{10}$$

4.2.3 Post-training Quantization

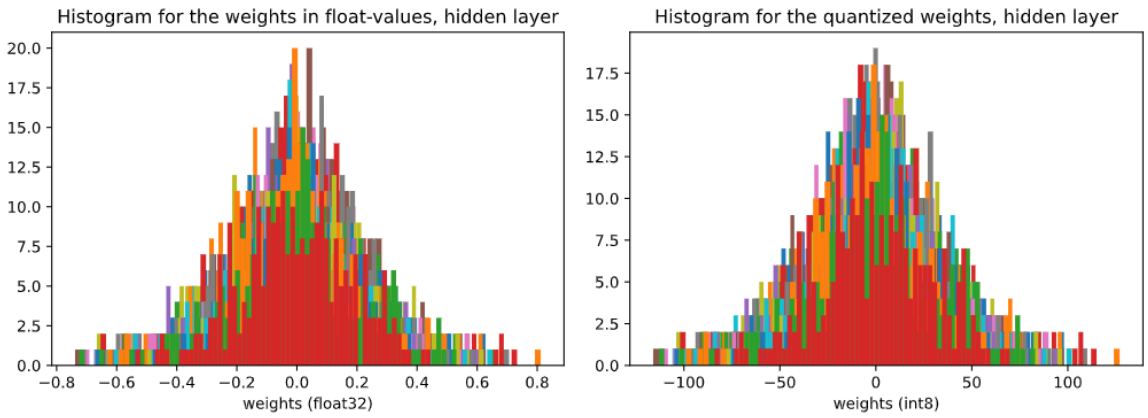
One conversion techinque allows us to quantize already-trained float TensorFlow models by converting them into the TensorFlow Lite - format, using the TensorFlow Lite Converter [11]. Additionally, when converting the models, the range $[\beta, \alpha]$, i.e (min, max), of all floating-point tensors in the model has to be calibrated or estimated, thus the converter needs a representative dataset in order to do the calibration; it has to have the same shape as the data used to train/test the networks ((1,256) for FCN, (1,256,1) for CNN, where the first dimension is the batch size of 1). The mapping range should more or less correspond to the value range of the data; since the amplitudes are in the mV range, we create a repesentative dataset of events samples (random values) between -0.1 and 0.1 (V), in the same shape as the input dimensions for the tensors.

In the TensorFlow Lite library we can find the specifications of the quantization [12]: the mapping is affine, and it has been done using per-tensor granularity for all the tensors.

This procedure has the advantage that all the quantization parameters are calculated offline, without having to involve further training or computing time. The activation tensors can be stored as integer tensors in the memory without having to be converted from floating point tensors. So we get the best inference performance, but also a higher accuracy loss with respect to other techniques (i.e. Quantization Aware Training, [10]); If the data is not representative, the scales and zero points computed might not reflect the true scenario during inference, harming the inference accuracy even more. In Figures 9 and 10 we can see some examples of the models weights before and after converting them. We can also see the value range that changes.

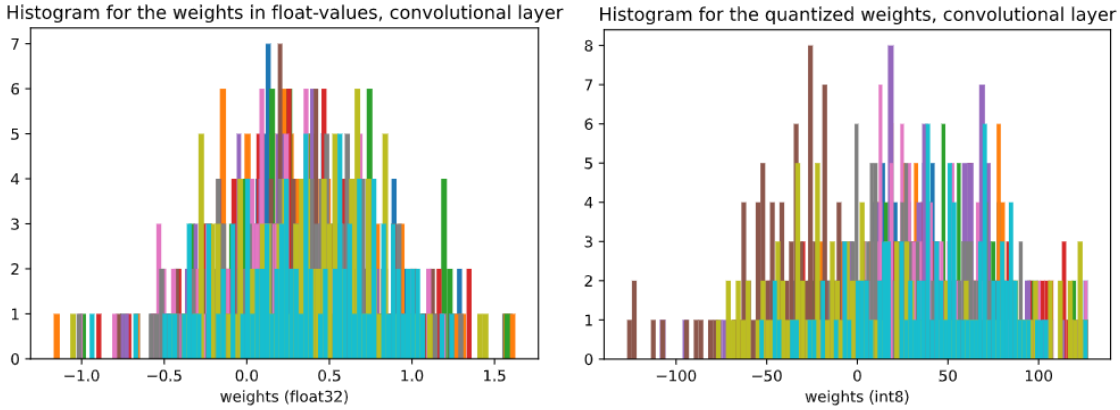
Tensor index	Operation	scaling factor s	zeropoint z
0	Input Tensor (input_int8)	$7.842\,421 \times 10^{-4}$	-1
1	ReadVariableOp, dense 1	$1.889\,821 \times 10^{-5}$	0
2	ReadVariableOp, dense 2	$3.883\,743 \times 10^{-4}$	0
3	MatMul, dense 1	$2.409\,741 \times 10^{-2}$	0
4	MatMul, dense 2	$3.606\,472 \times 10^{-2}$	0
5	Relu;sequential_1, BiasAdd, dense 1	$1.076\,882 \times 10^{-2}$	-128
6	BiasAdd, dense 2	$1.092\,524 \times 10^{-1}$	-8
7	Output Tensor (Identity_int8)	$3.906\,25 \times 10^{-3}$	-128

Table 3: Resulting quantization parameters (scaling factor and zeropoint) for every tensor of the smallest generated FCN (1 hidden layer, 16 parameters - Fig. 14a).



(a) Median = 0.261 (b) Median = 0

Figure 9: Example of quantization: histograms of the weights for the first hidden layer of an FCN (256 × 64 weights), in floating points on the left, and integers on the right. We can see that the quantization maps the weights from a range $[-0.74, 0.81]$ onto the range $[-116, 127]$.



(a) Median = 0.268 (b) Median = 28

Figure 10: Example of quantization: histograms of the weights for a convolutional layer (10 × 1 × 10 × 10 weights), in floating points on the left, and integers on the right. We can see that the quantization maps the weights from a range $[-1.17, 1.63]$ onto the range $[-127, 127]$.

5 Inferences with the Coral EdgeTPU Development Board

After defining and calculating the size of our networks in terms of operations per second in Section 4.1, we want to measure the the speed of the neural networks (also called inference time; the time needed to do one inference), and see by how much it can be optimized through post-training quantization (Section 4.2.3). By taking the inverse value of the average inference time of a network, we can obtain its event processing rate.

We perform all of our measurements on the Coral EdgeTPU DevBoard; it a single-board computer that contains an Edge TPU coprocessor (Tensor Processing Unit). It's ideal for prototyping new projects that demand fast on-device inferencing for machine learning models. In addition to a classic CPU, we can also run inferences on the TPU, which is an accelerator particularly suited for linear algebra calculations and tensor operations. The DevBoard is also equipped with a GPU, which unfortunately does not support model inferences with TensorFlow Lite, though. All documentation for the Edge TPU and how to work with it can be found at Coral.ai [13]. More information about the

hardware can be found in the Dev Board datasheet. We are very interested in testing the speed of our networks on the TPU, with the hope of being able to obtain even faster predictions. If we can find a clear advantage in using the TPU to make our predictions, it could be very interesting to implement the processor onto the ARIANNA station. This is also the reason for us to compare the speed of our quantized models on the CPU with respect to their speed on the TPU.

In order to be able to run the models on the EdgeTPU, they have to be convertet onto a '.tflite' - model, since the DevBoard only runs on TensorFlow Lite [14]; the package is a lighter version of TensorFlow, with some less functionalities, and is suited for neural networks on small devices. The same applies for the quantized models which are automatically converted into a '.tflite' - format.

We use these converted models in order to run inference on the created models on the DevBoard's CPU; in order to see the performance on the TPU coprocessor, we have to firstly convert the models using the Edge TPU Compiler [15], and then call the 'libedgetpu.so.1' - delegate in the python-scripts (Linux) [16].

6 Results

6.1 Inference Results

Knowing the approximate size of a model in terms of FLOPS, we can then test the speed of each model. In Figure 11 we can see the inference measurements (expressed as event prediction rates) as a function of the number of FLOPS for each network. We firstly compare the performance of the FCNs vs. the performance of the CNNs, and we observe by how much the speed of those networks increases when we quantize them. This has been done on the CPU of the Coral DevBoard. In Fig. 11b we then compare the speed of quantized networks by running them firstly on the CPU and then on the TPU.

The inference evaluations have been made by measuring the time necessary to infere over 10,000 events (then divided by the same number in order to find the time over a single inference), in order to reduce the systematic errors; we then repeated this for 30 events in the dataset and took the average of it. The errors have been calulated by taking the standard deviation of the 30 measurements. We the found the average prediction rate by taking the inverse value of the time per inference; we then used the standard gaussian error propagation formula to find the error of the rates: $\Delta\text{rate} = \frac{1}{t^2} \cdot \Delta t$.

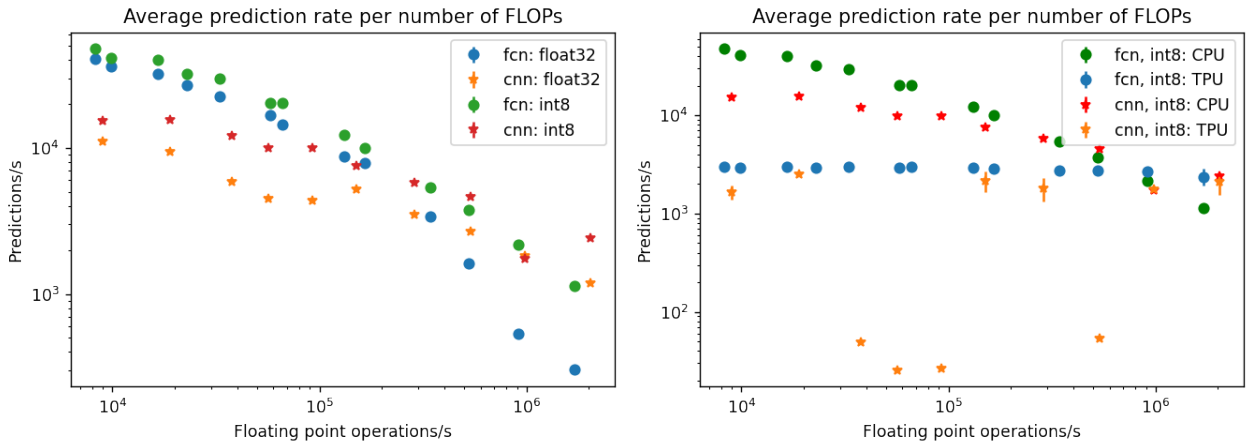
First of all, we can confirm that the quantized networks are actually faster than the unquantized ones, even though the gain in performance is not very remarkable, especially for smaller FCNs. For less FLOPS, FCNs are clearly much faster than the CNNs, but after passing a certain size it is actually worth it to switch to the fully connected networks, which get perform better at a certain point.

Our fastest FCN is also the smallest one (at about 8.3k FLOPS; Fig. 14a): we get (40.56 ± 0.06) kHz for the unquantized network, and (47.83 ± 0.07) kHz for the quantized model, on the CPU.

Our fastest CNN is also the smallest, and has a similar number of FLOPS, but is much slower (Fig. 17a): we get (11.13 ± 0.003) kHz for the unquantized network, and (15.24 ± 0.006) kHz for the quantized model, on the CPU.

Unexpectedly, from the runs on the TPU we observe that none of the networks is able to make more than 3000 predictions per second; the fully connected networks all yield the same results; we have an average rate of (2.9 ± 0.03) kHz, whereas the convolutional networks fluctuate a lot in the daHz to kHz - range.

A possible explanation for such a bad performance could be that networks with less parameters have some performance penalties when they are too small; hey might have some overhead time which takes much longer than the actual computations, making that inference time neglectable, explaining why we would always get the same timing for the FCNs. Some inference investigations about this can be found on Github.

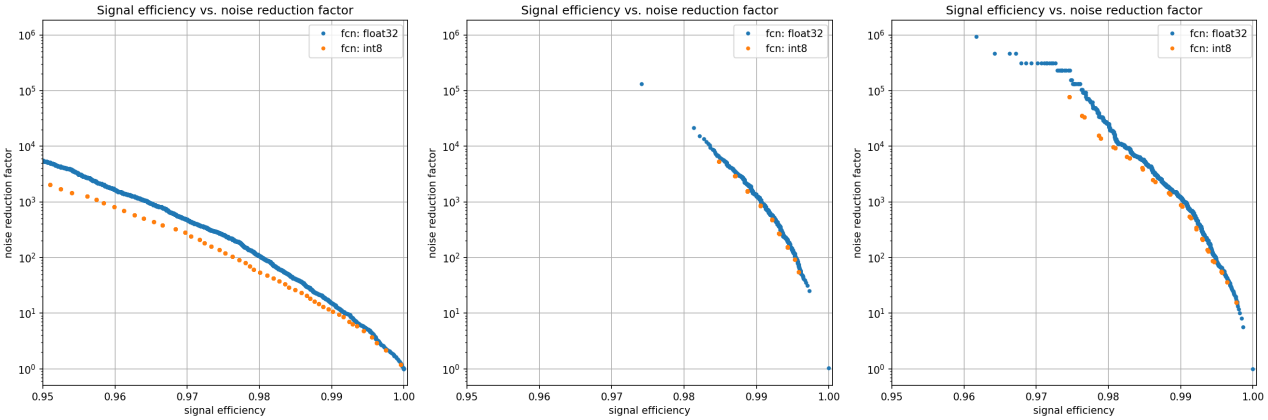


(a) float32 vs. int8 on CPU

(b) int8 models on CPU vs. TPU

Figure 11: Event prediction rate plots for all our generated and converted models, in function of their size (in terms of FLOPS). On the left, we plotted the speed results on the CPU, whereas in the right picture we compare the performance of all the quantized models between the CPU and the TPU runs (once the models are quantized, they do not have FLOPS anymore, but we label them onto the same value as their float32 - version in order to compare them better). We can observe the trend that bigger networks are generally slower than the smaller ones, also that the quantized networks run faster than the unquantized ones. For less FLOPS, the convolutional networks are less efficient in terms of speed, but they become of advantage for bigger sizes. Unfortunately, the speed results on the TPU are inconclusive.

6.2 Efficiency of the quantized models



(a) 1 layer, 16 nodes

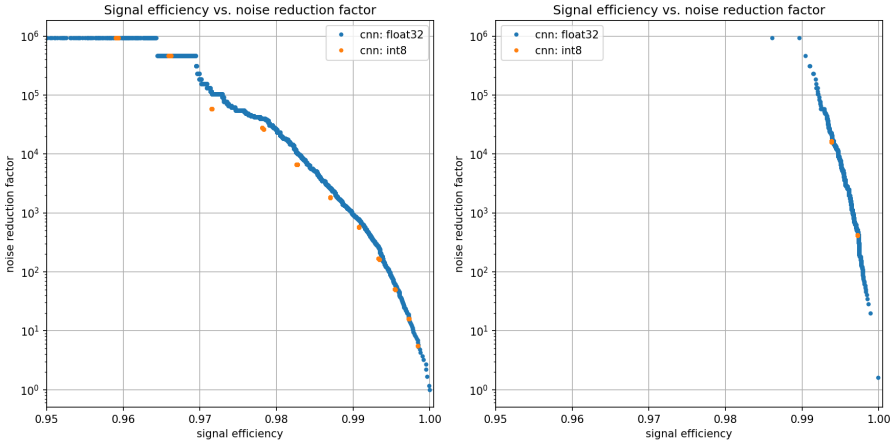
(b) 2 layers, 64 nodes

(c) 1 layer, 128 nodes

Figure 12: Efficiency curves for some fully connected networks in floating-point representation (blue curves) and in integer representation (yellow curves). The models from left to right are: the smallest one (∼8'300 FLOPs), the most efficient one (∼40'000 FLOPs) and the one comparable to the smallest convolutional network (∼123'000 FLOPs). We can see that there is a loss in signal efficiency for the quantized models, especially in the smaller one, but it is very small. NB: since the networks have been quantized, the outputs are only integer values, and this is why the curves of the int8-operating networks are not continuous, but jump whenever the threshold passes an integer value.

The Figures 12 and 13 show us the efficiency plots of the same models shown previously vs. the efficiency of their quantized version. Since the outputs of the quantized models are quantized as well (integers between -128 and 127), they have to be interpreted a bit differently than the floating point values obtained previously. Because of this, in order to calculate the efficiencies, we iterate the threshold value over the range $[-128, 127]$ in 10'000 steps. Another consequence of the quantization is that more outputs get approximated onto the same value, and we have a less continuous range of predicions. This explains why the curves in the figures jump from one point to another; there is always a certain number of predictions that gets added as soon as a certain integer threshold value is passed. For the convolutional networks, we get even less datapoints, and only two for the most efficient one.

In general, from these plots we can say that the quantized version of the models follow the same trend as their floating-point version in terms of accuracy, even though they are a bit less accurate. But they still fulfill the conditions we were looking for, meaning that we still get signal efficiencies that are higher than 95% for a noise rejection of four orders of magnitude.



(a) 1 layer, 5 filters, kernel dim = 5 (b) 2 layer, 10 filters, kernel dim = 10

Figure 13: Efficiency curves for some convolutional networks in floating-point representation (blue curves) and in integer representation (yellow curves). The models from left to right are: the second smallest one (∼18'000 FLOPs) and the most efficient one (∼540'000 FLOPs). We can see that there is a loss in signal efficiency for the quantized models. For these networks, we get even less distinct points to show, our network on the right only predicts 2 distinct values for the signal. But the points still seem to fit the efficiency curve of the floating-point model, telling us that the int8-model is making the same correct predictions aswell.

7 Conclusions and Future plans

The low flux of high-energy neutrinos and their small interaction cross section make it challenging to detect them on Earth and requires large arrays of very sensitive detectors. Even then, most analyses will be limited by the statistical uncertainty of the number of events. Thus, increasing the sensitivity of the detector has huge potential. In this work, the usage of neural networks to reject thermal noise fluctuations in real time was studied that allows to further decrease the detection threshold and thus increase the sensitivity.

From our experiments with deep neural networks we found a way of rejecting efficiently a lot of noise while still keeping a very high signal efficiency: a fully connected network with an efficiency of 98.35%, and a convolutional network with an efficiency of 99.47% at a 10^4-noise rejection. We also found a very efficient way of optimizing those networks in order to reduce them in size and increase their computing speed, using integer quantization techniques; we convert their number operations,

which were based on floating point operations, onto integer operation, reducing their bitwidths. This network quantization would make calculations on the MBED processor (which is the one at the ARIANNA detector) faster, since floating point operations have no hardware acceleration. The processor also has very limited power ressources, since the whole detector only takes up 5W of power.

For the optimization of our network, we also have to take into account their computation speed; we manage to obtain event prediction rates in the 10^3 - 10^4 Hz range, which is what we were looking for. We then tested the networks on a dedicated deep learning chip (TPU) with the hope to increase the computation speed significantly. But unfortunately, those speed measurements were inconclusive. We should have seen a remarkable increase in the speed of every single network, but instead we found that they all performed badly, with many fluctuations in the speeds. A likely explanation for the slow performance on the TPU is that some operations are still performed on the CPU which introduces a significant overhead due to transferring data back and forth the CPU and TPU, slowing down the calculations by a notable factor. Another reason could be that the networks are so small in terms of parameters, that there are some performance penalties on the TPU with respect to the CPU, making it still run faster on the latter. Additionally, the speed comparison between the CPU and the TPU is not everything; if the TPU had the same processing speed as the CPU but at a lower power consumption, this would also be an interesting advantage for our devices.

The next step would be to make lab experiments of the obtained quantized models on the same microprocessor that is used at the ARIANNA station, in order to see how well they perform under the actual conditions of the South Pole. If they perform well, there might be a chance that those neural network actually get implemented onto the station. Adding a TPU chip to the station hardware might be an interesting option for future designs as it only consumes 2W of power on full load, even though the total power consumption of the station would increase by a big fraction.

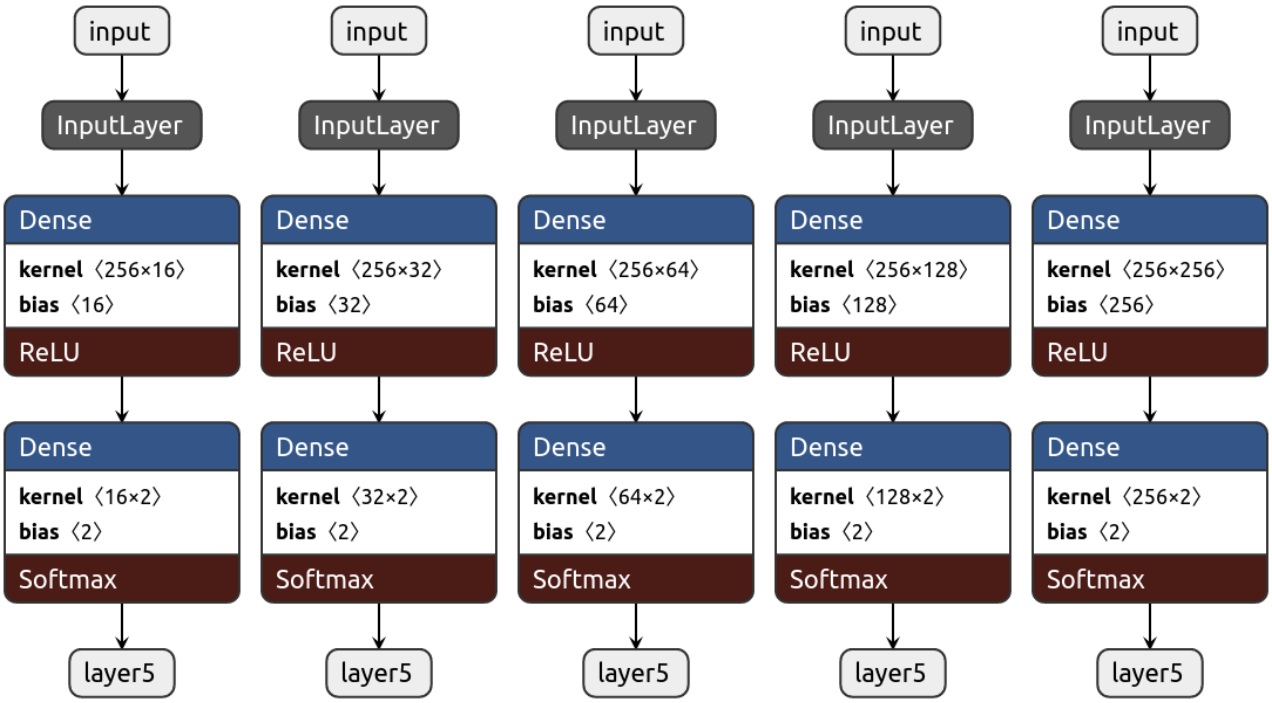
A Code repository

All the python-scripts that have been used for this research project are uploaded on the following Github - Code repository: https://github.com/luca-rigon/Intelligent-trigger.

B Network architectures

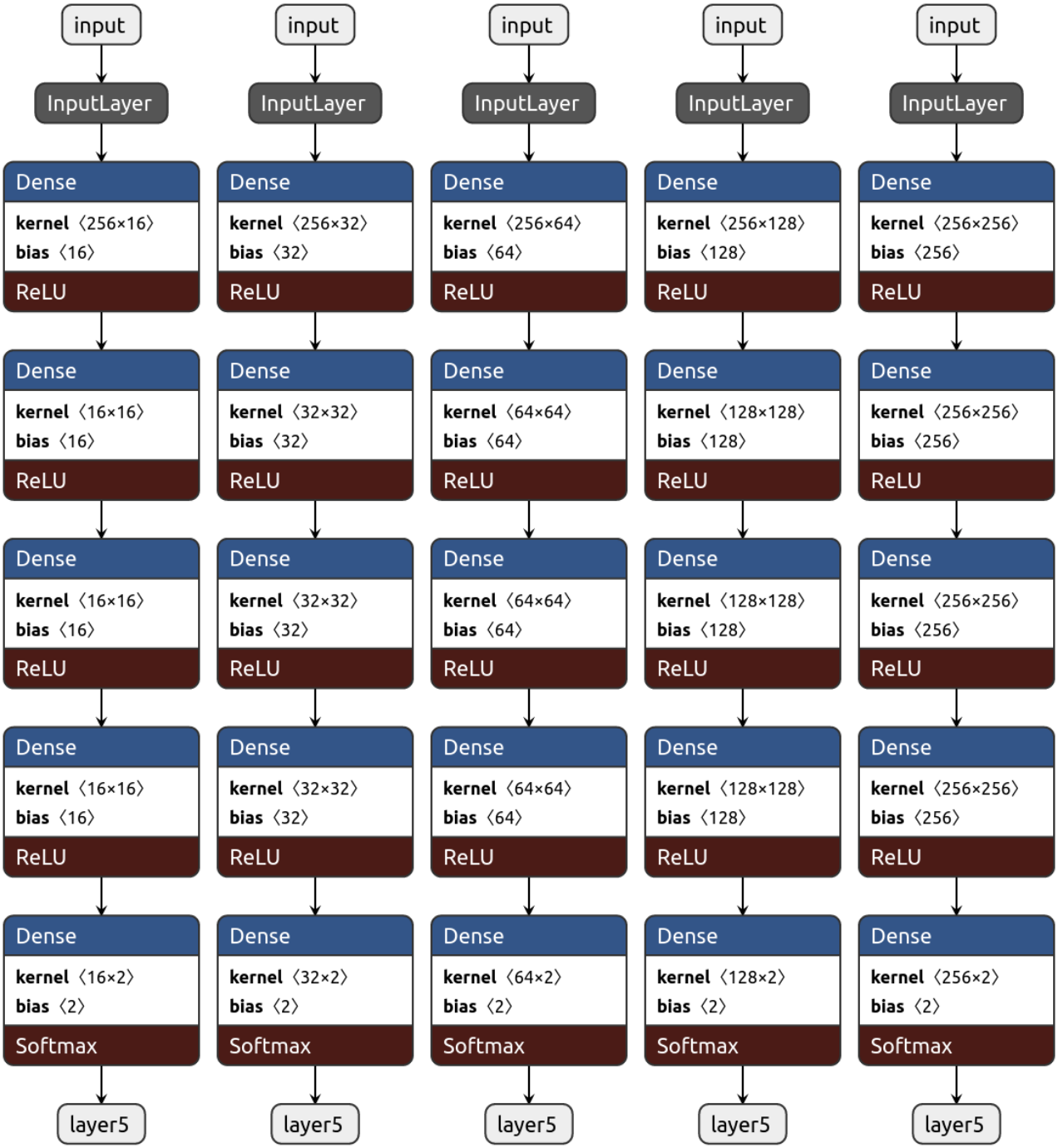
The visualization for the structure of each single trained network has created by using the Netron tool: https://netron.app/.

B.1 FCNs



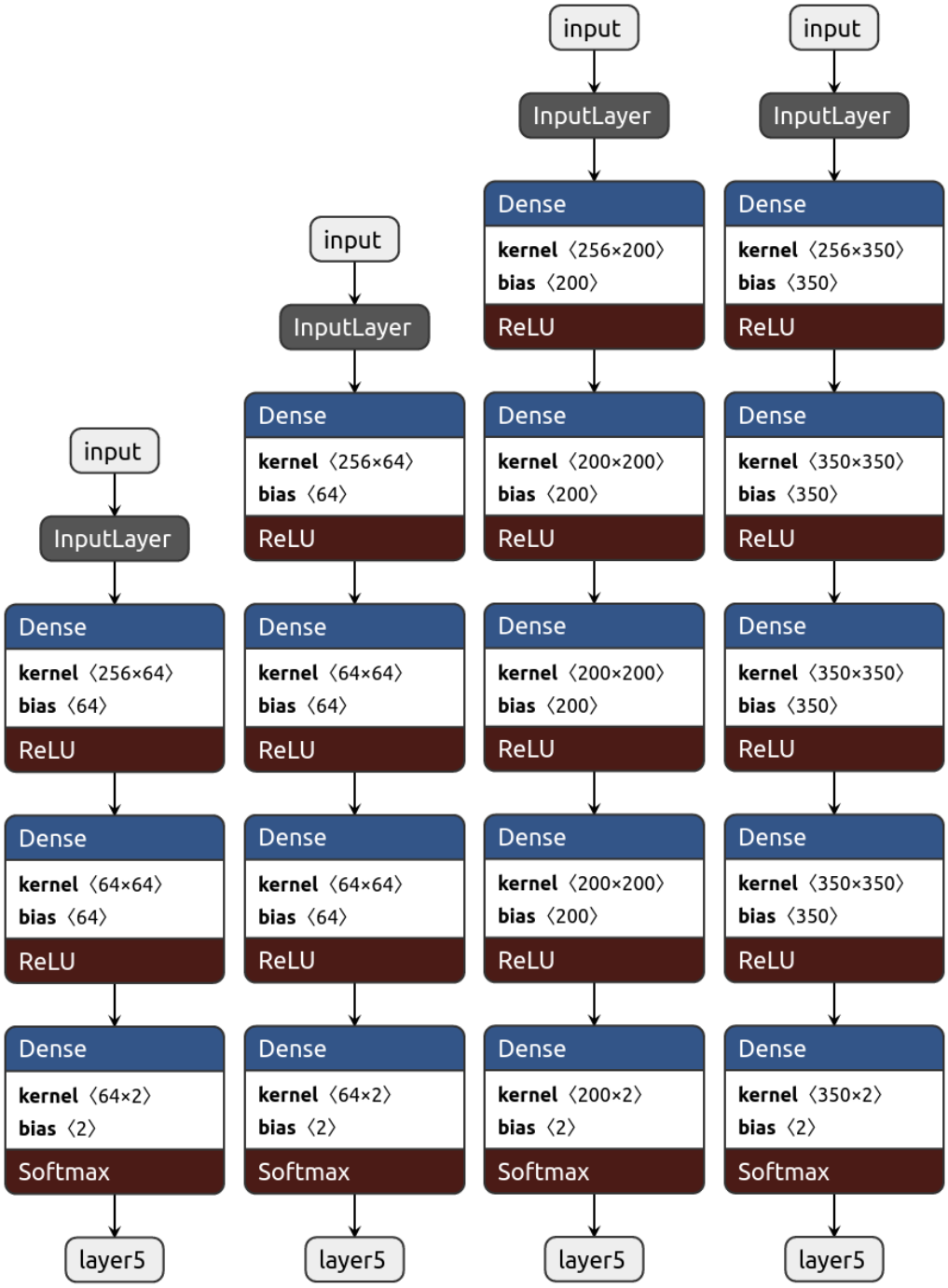
(a) 1 layer, 16 nodes (b) 1 layer, 32 nodes (c) 1 layer, 64 nodes (d) 1 layer, 128 nodes (e) 1 layer, 256 nodes

Figure 14: Fully connected networks with one hidden layer, varying node size.



(a) 4 layers, 16 nodes (b) 4 layers, 32 nodes (c) 4 layers, 64 nodes (d) 4 layers, 128 nodes (e) 4 layers, 256 nodes

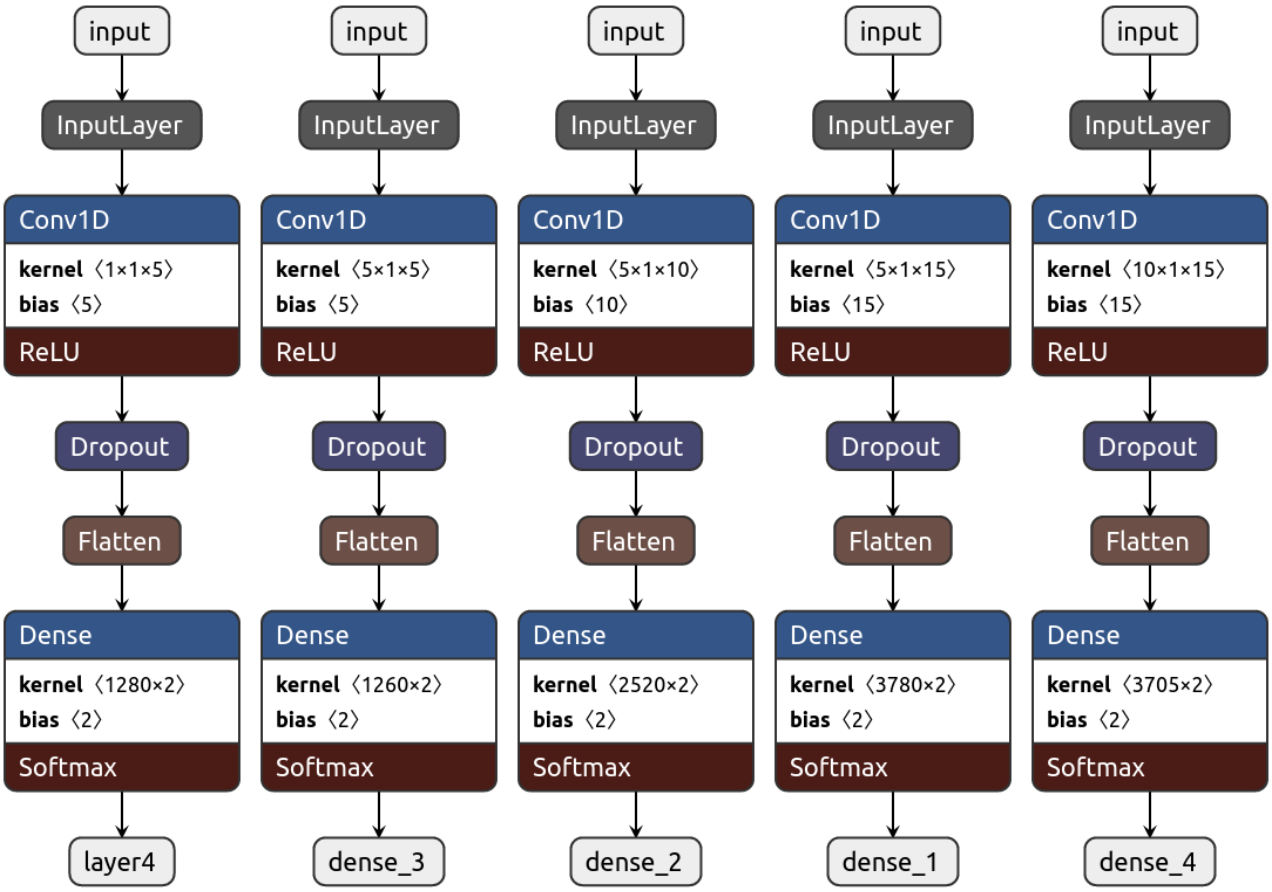
Figure 15: Fully connected networks with four hidden layers, varying node size.



(a) 2 layers, 64 nodes (b) 3 layers, 64 nodes (c) 4 layers, 200 nodes (d) 4 layers, 350 nodes

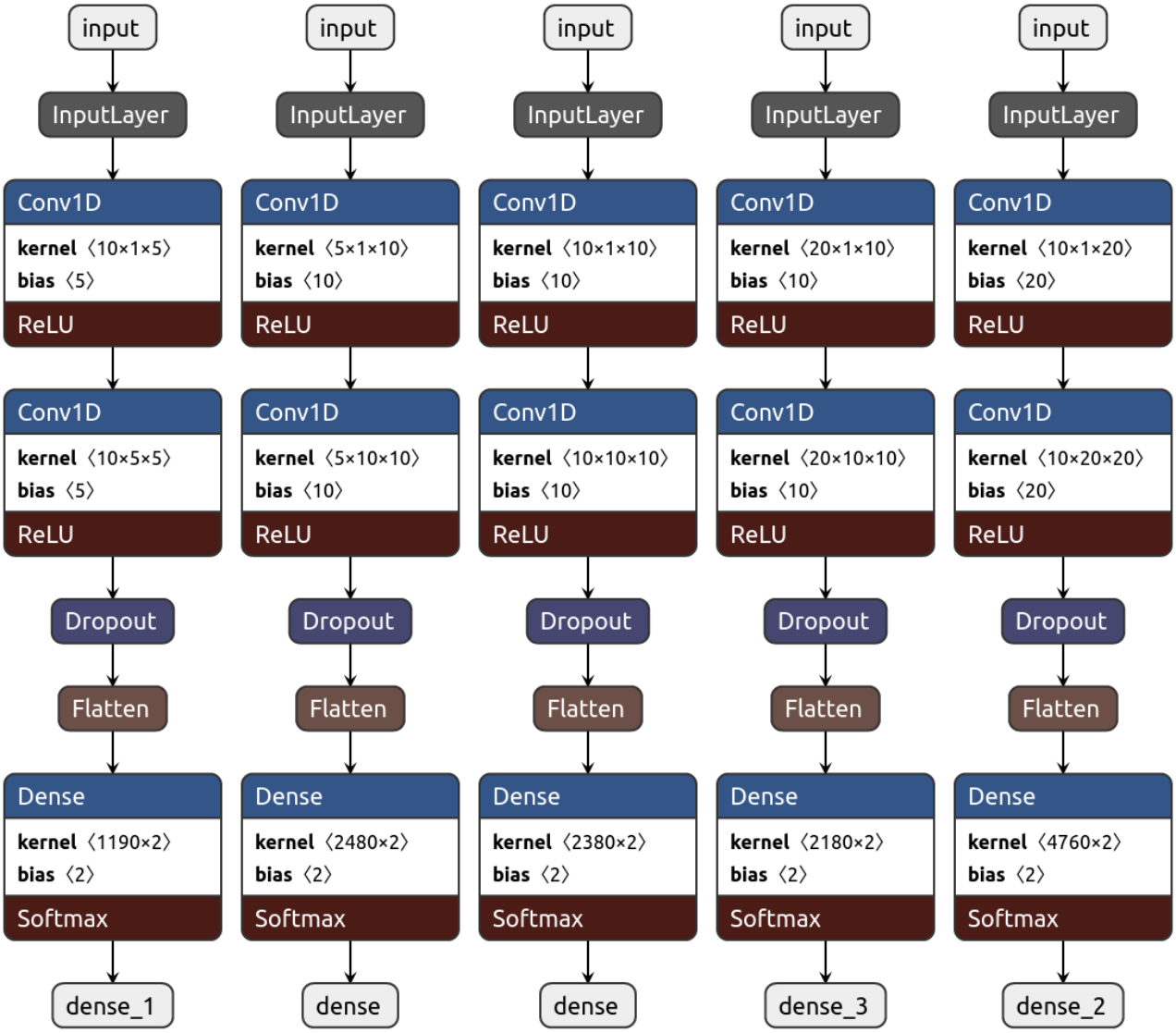
Figure 16: The remaining fully connected networks that have been used.

B.2 CNNs



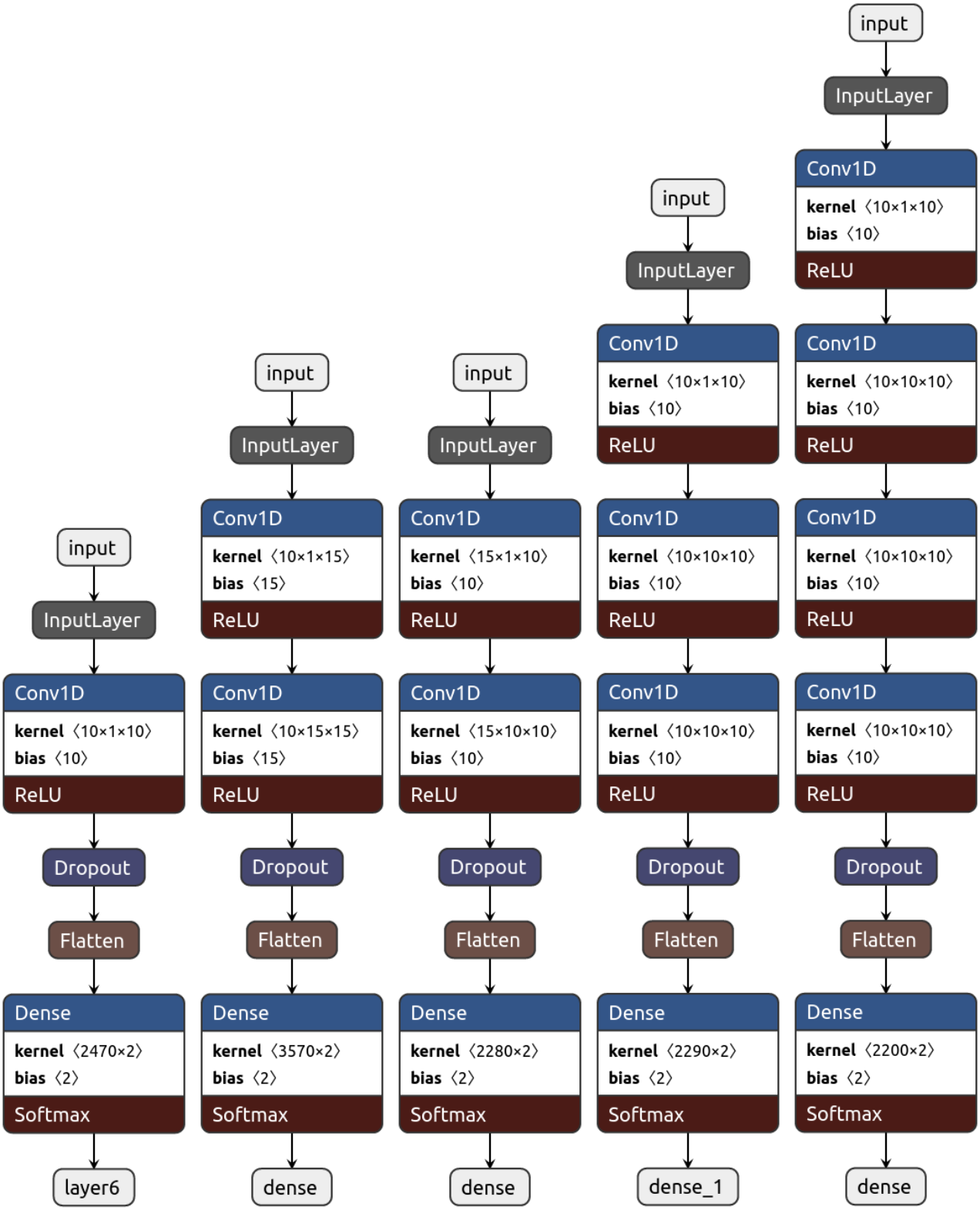
(a) $k = 1 \times 1$, 5 f (b) $k = 5 \times 1$, 5 f (c) $k = 5 \times 1$, 10 f (d) $k = 5 \times 1$, 15 f (e) $k = 10 \times 1$, 15 f

Figure 17: Convolutional networks with one convolutional layer, varying kernel size and number of filters, in increasing number of FLOPS order.



(a) $k = 10 \times 1$, 5 f (b) $k = 5 \times 1$, 10 f (c) $k = 10 \times 1$, 10 f (d) $k = 20 \times 1$, 10 f (e) $k = 10 \times 1$, 20 f

Figure 18: Convolutional networks with two convolutional layers, varying kernel size and number of filters, in increasing number of FLOPS order. They are also all used for the efficiency plots in Fig,7.



(a) 1l, $k = 10 \times 1$, 10 f (b) 2l, $k = 10 \times 1$, 15 f (c) 2l, $k = 15 \times 1$, 10 f (d) 3l, $k = 10 \times 1$, 10 f (e) 4l, $k = 10 \times 1$, 10 f

Figure 19: Remaining Convolutional networks used in Figure 7, with varying number of convolutional layers, kernel size and number of filters.

References

[1] A. Anker, S.W. Barwick, H. Bernhoff, D.Z. Besson, N. Bingefors, D. García-Fernández, G. Gaswint, C. Glaser, A. Hallgren, J.C. Hanson, and et al. A search for cosmogenic neutrinos with the arianna test bed using 4.5 years of data. *Journal of Cosmology and Astroparticle Physics*, 2020(03):053–053, Mar 2020.

[2] P. W. Gorham, S. W. Barwick, J. J. Beatty, D. Z. Besson, W. R. Binns, C. Chen, P. Chen, J. M. Clem, A. Connolly, P. F. Dowkontt, and et al. Observations of the askaryan effect in ice. *Physical Review Letters*, 99(17), Oct 2007.

[3] Patrick van der Smagt Ben Kröse. An introduction to neural networks. University of Amsterdam, Nov 1996.

[4] Activation functions in neural networks. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[5] García-Fernández D. Nelles A. et al. Glaser, C. Nuradiomc: simulating the radio emission of neutrinos from interaction to detector. *Eur. Phys. J. C*, 80(77), 2020.

[6] Keras functional api. https://keras.io/guides/functional_api/.

[7] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Li Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks, 2017.

[8] Matthijs Hollemans. How fast is my model? https://machinethink.net/blog/how-fast-is-my-model/.

[9] Lei Mao. Quantization for neural networks. https://leimao.github.io/article/Neural-Networks-Quantization/.

[10] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation, 2020.

[11] TensorFlow Lite Guide. Post-training quantization. https://www.tensorflow.org/lite/performance/post_training_quantization.

[12] TensorFlow Lite Guide. Tensorflow lite 8-bit quantization specification. https://www.tensorflow.org/lite/performance/quantization_spec.

[13] Coral documentation. https://coral.ai/docs/.

[14] TensorFlow Lite Guide. Tensorflow lite converter. https://www.tensorflow.org/lite/convert.

[15] Edge tpu compiler. https://coral.ai/docs/edgetpu/compiler/.

[16] Run inference on the edge tpu with python. https://coral.ai/docs/edgetpu/compiler/.