Topics in Particle Physics and Astroparticles II

Guilherme Soares

Laboratory of Instrumentation and Experimental Particle Physics

6 April 2022

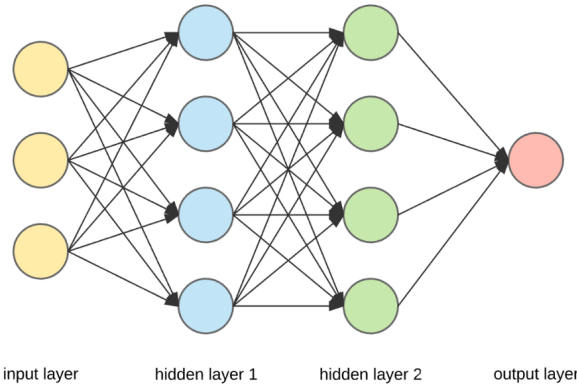
Index

Machine Learning

Machine Learning algorithms have an advantage over traditional selection methods : **They take into account multi-variable correlations** in a non-trivial way.

We will be utilizing **Multi-layer Perceptrons**, which are feedforward Neural Networks.

Neural Networks are interconnected groups of nodes (neurons) arranged in layers, where each node processes information and passes the result on to the next layer
Input layer is the first layer, that receives the data, and has to have 1 node for each feature of the data



Output layer is the last layer, and usually has as many neurons as the classes that we are trying to classify our data into. The one exception is for binary problems, where just 1 neuron is enough (the chance of not being from a set is the maximum score possible minus the score attained, hence low scores are a measure of being of the other set)

The connections between nodes are defined by **Weights** that are adapted through the training phase, and each of the nodes might have a threshold or a bias. Alongside the structure of the network, the weights are what define the network

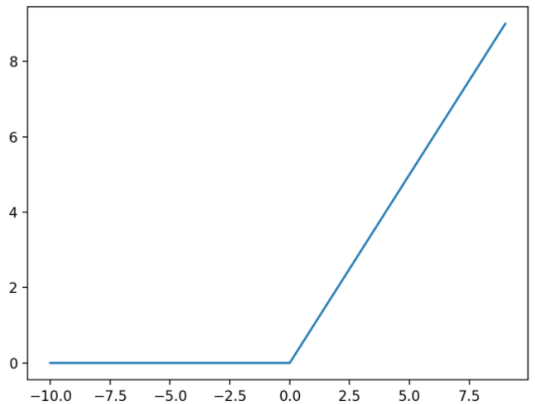
Machine Learning - Activation Functions

Each node processes the received information before transmitting it to the next layer. The biggest difference between MLPs and simple NNs is that the **Activation Functions** for MLPs are non-linear for all layers besides the first, which allows for backpropagation.

This implies that MLPs always have at least 3 layers, since all linear functions can be simplified through linear algebra in a way such that we would only need 2 layers : input and output. The layers in between these are called **Hidden Layers**

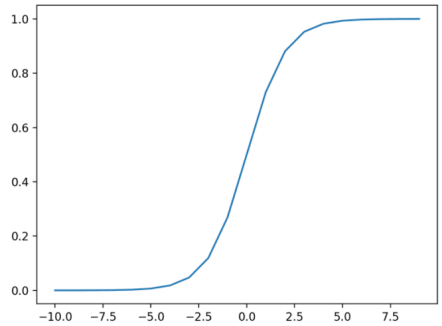
The more complex and bigger these hidden layers are, the longer it takes to process information. As such the activation functions need to be simple, fast to calculate and limited output ranges are preferred.

- Rectified Linear Unit (ReLU)

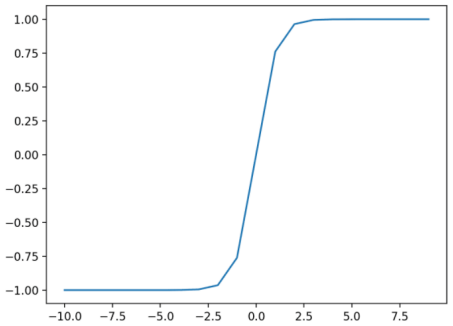


Machine Learning - Activation Functions

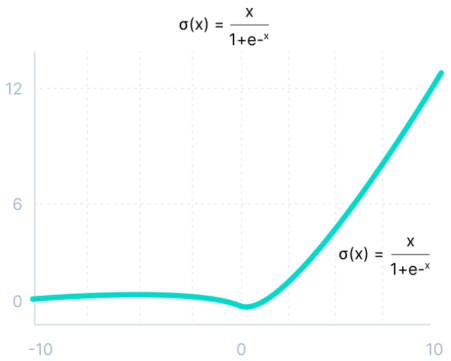
- Sigmoid



- Hiperbolic Tangent (Tanh)



- Swish ($x \times$ Sigmoid)



Machine Learning - More Basic Concepts

MLPs are NNs with **Supervised Learning**, which means that they train by processing examples with clearly defined inputs and results (features of the data, and we know it what is signal/noise à priori), forming a probability weighted association between the two.

This probability (weights) is updated according to a **Loss-Function**, which is commonly the **Cross-Entropy**. This is just a logarithmic function that is skewed towards correct predictions by heavily punishing incorrect predictions

$$\sum_i t_i log\left(p(x_i)\right) + (1 - t_i) log\left(1 - p(x_i)\right)$$

Here t_i is the true value of the event i (usually 1 for signal 0 for background) and $p(x_i)$ is the probability that the MLP attributed to said event.

Since it is impossible to find the optimal value for the weights of NNs, we try to find close to optimal solutions with **Optimization Algorithms**. These are usually stochastic gradient descent algorithms.

Each time the algorithm sees the whole dataset once is called an **Epoch**, but this does not necessarily define the rate at which the **weights** are updated. That rate is known as a **Batch**.

Machine Learning - Last Basic Concepts

The last thing to mention is that **there is** such a thing as **Overfitting**. This happens when the NN becomes so optimized for our data that it is turned obsolete when faced with new data.

In order to circumvent this there are several tools. The first one is setting aside a sample of our data that we do not use in training, but rather use to validate our training after each **epoch**, called **Validation Sample**. This way, we can let our NN run freely for as long as we want, and we can save the weights that give the best performance for the validation sample, that the NN is blind to, and as such should have no biases towards.

The second tool is varying the structure of our MLP. If the MLP is too complex for a problem, there is no avoiding overfitting our data, as it is akin to trying to use a pneumatic hammer to drive a nail.

The third tool is to try and mask information as we progress deeper into the MLP. This can be done through a function that we have already seen in Pandas : **drop()**. This ensures that random fractions of our data are removed, in order to try and avoid biases stemming from a high degree of omniscience.

The last thing to note is that due to all the randomness involved in ML algorithms, which can be as simple as being lucky in the early guesses of the optimization algorithms regarding the optimal solution, even while maintaining everything equal in one NN, training it several times will always yield different results, although they will never be too dissimilar provided we are constructing the Network correctly.

Machine Learning Template

The relevant template can be found under Template_ML.ipynb, under the usual github page.

We will be utilizing the same SHiP dataset utilized for Template_Data_Analysis.ipynb, which is named Background.dat and Signal.dat under the Data directory.

We start by importing the data, and altering slightly the dataframes to make them compatible with each other.

Machine Learning Template - Training and Validation Samples

The first thing to do in ML is to define the datasets that we want to use when training the Neural Networks.

Our **datasets should always be normalized**. While we are able to identify that a mass of 1 GeV is relatively big but a total momentum of 1 GeV is rather small, the algorithm might not be able to do that. Hence, we start by creating what is known as a transformation pipeline, which is an object that transforms all data applied to it according to specific definitions.

In our case we are going to perform 2 transformations : **Standard Scaler** and **Principal Component Analysis**. While the first re-scales all feaatures to have a mean of 0 and a standard deviation of 1, the second performs a basis rotation to the features in order to leave them as independent of each other as possible, while maximizing each the variance within each new feature.

While the second transformation might not be intuitive to us, it is effective and, once again, the algorithm does not understand units or physical features, but is vastly better than us at performing multi-dimensional analysis regardless of the data format.

Machine Learning Tutorial - Training and Validation Samples

Now we need to **define** what **features** we will give as **input to the NN**.

This can be done in several ways, but the important idea here is that we should not be lazy and feed everything we have to the NN. Some features are highly correlated, and others provide no discernible difference between signal and background.

Choose wisely as providing too much data in the best of cases increases the necessary computing power requirements, and in the worst case is like pilling extra straws on a haystack when we are trying to find a needle, "confusing" our algorithm.

After we selected the variables to use, we feed the pipeline our **full dataset** (signal+background), as this allows the **normalization** to **always** be **the same**.

Once we normalize the features that we want, we then separate the data back into the Signal and Background for ease of processing.

Machine Learning Tutorial - Training and Validation Samples

Defining our Training and Validation Samples

Now that our data is ready to be processed, we still need to define the actual samples.

Since the algorithm defines that optimization of a certain set of weights through minimizing a loss function, providing samples with unbalanced sizes might skew the NN's evaluation of its solution towards one. To avoid this we define two **training samples with similar dimensions** : one of signal and one of background.

When training a NN one should reserve a significant amount of event to validate the training, in order to avoid **overfitting**. In our case, we save 30% of the smallest sample for validation and define the other one accordingly.

All that is left to do is to transform our samples into arrays : 1 for development (training) and 1 for validation (X_class), and 1 with the score for the development set and another for the validation set (Y_class).

Machine Learning Tutorial - Training Options

Now we need to define some options regarding our MLP. We present here some basic options, where we define our **loss function** as being the aforementioned **binary cross-entropy**, while utilizing the standard **Adam** tensorflow-keras optimizer.

As for the remaining primary features, we set as default **500 epochs**, which is a small number that allows for very fast, but not optimal results, with **batch sizes of 100**.

We will define a NN for binary classification, since we want to separate between a signal sample and a background sample. Here we define the **depth of the hidden layers** as being 2 layers, with the first one having a **width of 25 neurons**, which **decrease by a step of 5** at each subsequent layer.

These are all default values that work but can be tweaked to provide more optimal results.

Machine Learning Tutorial - Training Options

After giving the general structure of the network, we need to **define the shape of the input and output layers**, as well as **initiate every layer** and **specify the activation functions** for each neuron.

We add layers through the tensorflow : :keras : :Sequential.add(Dense()) function, where the arguments for Dense() must include the number of neurons (here given by width), the kernel_initializer, that sets the default values for the neuron's weights, and the activation functions, which we set as the commonly seen ReLU. Notice that when we create the first layer we need to add the number of inputs through input_shape.

The last thing to note is that there are some Sequential.add(Dropout(x)) functions spread throughout the network. This function makes it so that a fraction x of the information is lost between layers. This is helpful to avoid early overfitting.

While reducing the dimension and complexity of the NN can also avoid overfitting, it can make it so that the near-optimal solution found is not as good as it could be, and the Dropout function allows for more complex NNs to have similar training loss scores to the validation ones for longer, hopefully providing a better final result.

Machine Learning Tutorial - Training the Model

The rest of the training code involves the Callbacks section, that defines some criteria on how and when we want to stop the training, and what data we want to save, both for the weights as well as for the history of the training.

Here we are choosing to save both the values of the loss function on the training sample, as well as for the validation sample.

Training the Neural Network

We start by defining a folder where we want to save our relevant data, such as the weights. **Always remember to alter this and keep a log of your runs**, so that you do not accidentally overwrite previous weights with good results!!!

Additionally, notice that we are choosing to save the set of weights that provides the best loss function result with the validation sample, and not the training one. Recall that overfitting can and always happens, given enough time.

When training your NN, always try to avoid early stops in the first iterations so you know what to expect.

Always check the loss function evolution plot and try to create NNs that maintain close training and validation loss values close for as long as possible.

Machine Learning Tutorial - Results

If you close your notebook after training a NN and want to revisit it later there is no problem. You just need to initiate your NN again (with the same configurations, so keep a good log of your work), open the files you saved from your training, and load the model again.
A standard check of your ML results is the **ROC curve** (Receiver Operating Characteristic). This is simply a plot of the true positive rate (TPR) as a function of the false positive rate (FPR), and gives a visual representation of the rate of true positives versus the rate of false positives as you increase the score threshold to determine what is or is not a signal event.
The closer the **area under the ROC curve** (AUC) is to 1, the better the separation between your samples.
In order to obtain the NN scores for a given dataset, you just need to use the function model.predict(data). This will return a pointer with an array for each output (which is usually on for each class unless you are doing a binary classification, where there is only 1 output), with each entry corresponding to the score attributed to each event.
Before applying the data to your NN always remember to select the correct features and normalize your data properly.
The only thing left is to establish a threshold to determine what we consider signal and what we consider background, and repeat the whole process again and again until we are happy with the selection efficiencies for both signal (as high as possible) and background (as low as possible) that our NNs provide.