Background optimization with neural networks, in the decay $B \to K^* \nu \bar{\nu}$

Lorenz Gärtner

August 10, 2018

1 Overview

The goal is to preselect events prior to Monte Carlo detector simulations, as these are computationally expensive. The Monte Carlo simulations are useful to find signals, which differ from standard model predictions. From simulated data, a neural network selects useful events according to particles of interest. We feed the neural network degrees of freedom, available directly after simulation (generator level variables).
SuperKEKB has a centre of mass energy of $\sqrt{s} = 10.58$ GeV, where the electron beam has 7 GeV and the positron beam has 4 GeV. This c.o.m. energy is just large enough to produce $\Upsilon(4S)$, which almost exclusively decays into $B^0\bar{B}^0$ or B^+B^- at a threshold energy of 10.55 GeV. Though, the decay branching ratio is 10^{-5}.
The Belle II detector consists of a pixel detector (identify decay vertices and detect low momentum tracks), a silicon vertex detector (reconstruct vertices), a Central Drift Chamber (charged particle trajectory and momentum), particle identification (Cherenkov radiation - mainly distinguish K and π), an EM calorimeter (photons, electrons) and a muon detector. Collision simulation is done by the EvtGen software. Detector simulation is done by Geant4. It simulates particle-detector interactions.
The background can be divided into combinatorial ($e^+e^- \to q\bar{q}$) and $B\bar{B}$ (from $\Upsilon(4S)$).

2 Thesis overview on neural networks

Training of neural networks is done by first calculating the element-wise neural output, then training it with b elements, known as the batch size, and then looping over the data e times, the epochs.
Neuron inputs $x_1, \ldots, x_m$, weighted by $w_1, \ldots, w_m$, plus a bias b are summed to

$$z = \boldsymbol{w}.\boldsymbol{x} + b.$$

The activation function maps z to the network output y, it comes in many forms. The classifier, Θ matches the output to a binary value. Each neuron has a target output r^t. For gradient descend learning, the network learns by updating weights and bias according to

$$\begin{aligned} \delta\boldsymbol{w}^h = \eta\nabla_{\boldsymbol{w}}E^h \qquad &\Rightarrow \qquad \boldsymbol{w}^h \to \boldsymbol{w}^h + \delta\boldsymbol{w}^h, \\ \delta\boldsymbol{b}^h = \eta\nabla_{\boldsymbol{b}}E^h \qquad &\Rightarrow \qquad \boldsymbol{b}^h \to \boldsymbol{b}^h + \delta\boldsymbol{b}^h, \end{aligned}$$

where η is the learning rate and the superscript t specifies the neuron. Small learning rates lead to a long runtime and slow learning. Large learning rates lead to makes the neural network very sensitive to noise and it might not converge. Therefore, decaying learning rates, which decrease after an update of weights and bias, are common.
Online learning updates weights and biases after each input data element, whereas batch learning calculates the mean error and updates the weight and biases after each batch. For a multilayer perceptron weights and bias of the output neuron are updated first, then the layer above etc; which is known as backpropagation.
Outputs can be divided in four groups: **true positive (tp), true negative(tn)**, in accordance with the target values. On the contrary, **false positive(fp), false negative(fn)** results are falsely labelled. We define $tp - \backslash tn - rates$ as tp/t_{target} and tn/t_{target}, respectively, where the $tn - rate$ classifies background rejection. Accuracy is calculated with the G mean, the geometric mean of the product of precision and recall, $tn/f_{target} = 1 - fp - rate$.
We use TensorFlow and Keras. TensorFlow has a feature called Tensorboard which visualizes the code as data flow graph.

3 Artificial neural networks - a deeper insight

Machine learning problems can be broadly classified into supervised learning, unsupervised learning and reinforcement learning. In supervised learning, we have set of feature vectors and their corresponding target

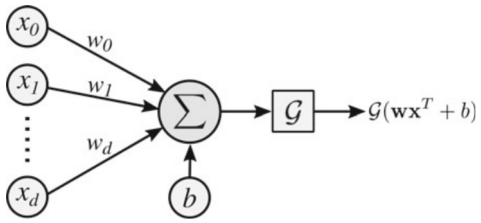

Figure 1: A schematic representation of a single neuron.

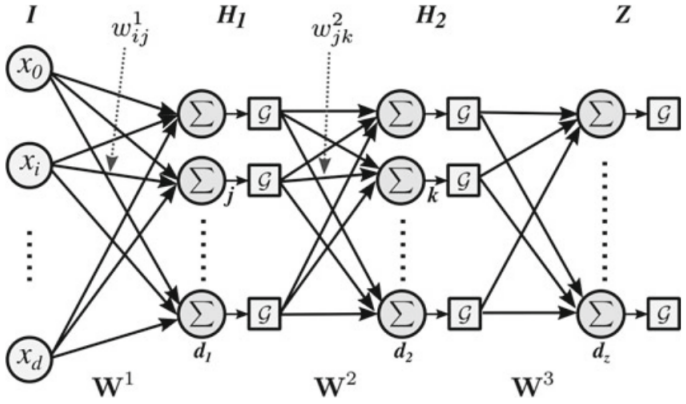

Figure 2: The general architecture of a feedforward neural network. Note that the bias connection is missing here.

values.
We represent the output of a single neuron as

$$\boldsymbol{x}^h = G(\boldsymbol{w}.\boldsymbol{x}^{h-1} + b),$$

where $\boldsymbol{w}$ is the weight vector, $\boldsymbol{x}$ is the input vector and b is the bias. The bias term shifts the activation function to left or right. It gives a neuron more freedom to be fitted on data.The activation function $G(x)$ performs a non-linear transformation on a real number and returns a real number - it acts as a threshold function, depending on the input it maps to the output correctly. A schematic is shown in Fig. 1. An artificial neural network is created by connecting one or more neurons to the input. Each pair of neurons may or may not have a connection between them. The logistic regression model can be formulated using only one neuron where $G(x)$ is the sigmoid function. Depending on how neurons are connected, a network act differently.

3.1 Feedforward neural networks

In a feedforward neural network the connections between neurons do not form a cycle - there are no loops in the network. The input layer is denoted I, the hidden layers are $\{H_i\}$ and the outer layer is labelled Z. Every neuron in a hidden layer or the output layer is connected to all the neurons in the previous layer. The weights connecting the input layer, I, to H_1 can be represented using a weight matrix $\boldsymbol{W}^1$ where W^1_{ij} shows the connection from the i^{th} input to the j^{th} neuron. All the neuron in the same layer usually have the same activation function. Each layer also has a bias vector, $\boldsymbol{b}^h$, for the h^{th} layer. As the output of one neuron is the input of the next we can write

$$\boldsymbol{x}^h = G(\boldsymbol{W}^h\boldsymbol{x}^{h-1} + \boldsymbol{b}^h),$$

where $G(\boldsymbol{x})_i = G(x_i)$ is a vectorising function. We define the weighted input, $\boldsymbol{z}^h = \boldsymbol{W}^h\boldsymbol{x}^{h-1} + \boldsymbol{b}^h$. The output of the network in Fig. 2 can be represented as

$$\boldsymbol{x}^Z = G(\boldsymbol{W}^3G(\boldsymbol{W}^2G(\boldsymbol{W}^1\boldsymbol{x}^0 + \boldsymbol{b}^1) + b^2) + b^3).$$

The input vector $\boldsymbol{x}^0$ is transformed into a d_1-dimensional space using the first hidden layer. Then, the transformed vectors are transformed into a d_2-dimensional space using the second hidden layer. Finally, the output layer classifies the transformed d_2-dimensional vectors. A feedforward network with one hidden layer can approximate any continuous function. A neural network can imitate a classification function. Only the *hyperparameters*, the number of hidden layers, number of neurons in each layer, and the type of activation functions need to be determined.
The *loss* or *cost function*, L, is used to determine the goodness of the model - it quantises the difference between network and true output. For neural networks, we try to minimise this cost function. For example, the mean squared error is defined as

$$L \equiv \frac{1}{2n} \sum_i \left| \boldsymbol{y}(\boldsymbol{x}_i) - \boldsymbol{x}^{i,Z} \right|^2 \ ,$$

where n is the total number of training inputs and $\boldsymbol{y}(\boldsymbol{x}_i)$ is the desired output for a specific training sample $\boldsymbol{x}_i$. We must assume that we can write the cost function as an average $L = \frac{1}{n}\sum_i L_i$. This must hold, as we require $\frac{\partial L_i}{\partial w}$ and $\frac{\partial L_i}{\partial b}$. We will write L_i as L from now on. More carefully, we use the gradient of loss function, computed by *backpropagation*.

3.1.1 Backpropagation 1

We define the error of neuron j in layer l by

$$\delta_i^h \equiv \frac{\partial L}{\partial z_i^l} \ , \tag{3.1}$$

and $\boldsymbol{\delta}^h$ as the vector of errors.
The *error in the output layer* is given by

$$\boxed{\delta_i^Z = \frac{\partial L}{\partial x_i^Z}\frac{\partial G}{\partial z_i^Z} \qquad \text{or} \qquad \boldsymbol{\delta}^Z = \nabla_{x^Z} L \odot \nabla_{z^Z} G} \ , \tag{3.2}$$

where $\boldsymbol{a} \odot \boldsymbol{b}$ represents the element wise product of the two vectors or Hadamard product. Which is just the chain rule of Eq. 3.1, as $G(z_i^Z) \equiv x_i^Z$.
The *recursive error* $\boldsymbol{\delta}^h$ *in terms of the error in the next layer,* $\boldsymbol{\delta}^{h+1}$,

$$\boxed{\boldsymbol{\delta}^h = \left[(\boldsymbol{W}^{h+1})^T \boldsymbol{\delta}^{h+1} \right] \odot \nabla_{z^h} . G} \ . \tag{3.3}$$

We prove this by writing

$$\begin{aligned} \boldsymbol{\delta}^h &= \nabla_{z^h} L \\ &= (\nabla_{z^{h+1}} L)(\nabla_{z^h} . \boldsymbol{z}^{h+1}) \\ &= \boldsymbol{\delta}^{h+1} (\nabla_{z^h} . \boldsymbol{z}^{h+1}) \ , \end{aligned}$$

but $\boldsymbol{z}^{h+1} = \boldsymbol{W}^{h+1}\boldsymbol{x}^h + \boldsymbol{b}^{h+1}$, so that

$$\nabla_{z^h} . \boldsymbol{z}^{h+1} = \boldsymbol{W}^{h+1} (\nabla_{z^h} . \boldsymbol{x}^h) = \boldsymbol{W}^{h+1} (\nabla_{z^h} . G(\boldsymbol{z}^h)) \ .$$

Which gives Eq. 3.3 upon substitution. We can use this to move the error backward through the network. Combined with Eq. 3.2, we can compute the error for each layer.
The *rate of change of the cost with respect to any bias in the network* is

$$\boxed{\frac{\partial L}{\partial b_i^h} \equiv \delta_i^h \qquad \text{or} \qquad \nabla_b L \equiv \boldsymbol{\delta}^h} \ . \tag{3.4}$$

The *the rate of change of the cost with respect to any weight in the network* is

$$\boxed{\frac{\partial L}{\partial W_{ij}^h} = \delta_i^h x_j^{h-1}} \,. \tag{3.5}$$

Hence, if the input/activation of the neuron, x_j^{h-1} is small, $\frac{\partial L}{\partial W_{ij}^h}$ is also small and the neuron learns slowly - does not change much during gradient descend.
The activation function $G(z_j^h)$, taken as a sigmoid, flattens as $z_j^h \to 0, 1$. Hence, the neuron in the following layer, $h+1$ will learn slowly in those limits. The output neuron has saturated and the weight has stopped adjusting. Of course the first term in Eq. 3.3 can compensate.
We can use the above 4 equations to design activation functions with desired properties.
We split the complete set training samples into batches. For a set of batches $\{\boldsymbol{x}^{i,0}\}$, the *backpropagation algorithm* works as follows:

1. For each training sample i:
   (a) **Input:** Set the input $\boldsymbol{x}^{i,0}$.
   (b) **Feedforward:** Compute $\boldsymbol{z}^{i,h}$ and $\boldsymbol{x}^{i,h} = G(\boldsymbol{z}^{i,h})$ for all h.
   (c) **Output error:** Compute $\boldsymbol{\delta}^{i,Z}$ - for the output layer.
   (d) **Backpropagate the error:** Compute $\boldsymbol{\delta}^{i,Z-1}, \boldsymbol{\delta}^{i,Z-2}, \ldots$.
   (e) **Output:** The gradient of cost is given by

$$\frac{\partial L}{\partial W_{jk}^{i,h}} = \delta_j^{i,h} x_k^{i,h-1} \qquad \text{and} \qquad \frac{\partial L}{\partial b_j^{i,h}} \equiv \delta_j^{i,h} \,.$$

2. **Gradient descend:** For each layer, update the weights and biases according to

$$\boldsymbol{W}^h \to \boldsymbol{W}^h - \frac{\eta}{m} \sum_i \boldsymbol{\delta}^{i,h} (\boldsymbol{x}^{i,h})^T \qquad \text{and} \qquad \boldsymbol{b}^h \to \boldsymbol{b}^h - \frac{\eta}{m} \sum_i \boldsymbol{\delta}^{i,h} \,,$$

   where m is the number of training samples in the batch and η is the learning rate.

3. Loop over many **epochs**.

3.1.2 Backpropagation 2

Taking a slightly different approach. Consider the neural network in Fig. 3. We take the activation function of the output layer is the identity function, $G_i^3(x) = x$, so we can ignore it. Computing $\frac{\partial L}{\partial \boldsymbol{w}_i^j}$ is equal to adding all paths starting from $\boldsymbol{w}_i^j$ and ending at L, according to the multivariate chain rule. For efficient computation we can factorize the equation as shown in Fig. 4.5 We see, that the gradient computations moves in the reverse direction. We can reuse parts of the equations for the computation of alternative paths. The backpropagation algorithm has been developed based on this factorization.
It is a method for efficiently computing the gradient of leaf nodes with respect to each node on the graph using only one backward pass from the leaf nodes to input nodes. Let $G = \langle \boldsymbol{V}, \boldsymbol{E} \rangle$ denote a directed acyclic graph where $\boldsymbol{V} = \{v_1, \ldots, v_K\}$ is set of nodes in the computational graph and $\boldsymbol{E} = (v_i, v_j) | (v_i, v_j \in \boldsymbol{V})$ is the set of ordered pairs (v_i, v_j) showing a directed edge from v_i to v_j. Number of edges going into a node is called *indegree* and the number of edges coming out of a node is called *outdegree*. The indegree of v_a is $|in(v_a)|$, where $in(v_a) = \{(v_i, v_j) | (v_i, v_j) \in \boldsymbol{E} \wedge v_j = v_a\}$. Similarly, The outdegree of v_a is $|out(v_a)|$, where $out(v_a) = \{(v_i, v_j) | (v_i, v_j) \in \boldsymbol{E} \wedge v_i = v_a\}$. A node is called input if $in(v_a) = 0$ and $out(v_a) > 0$, and leaf if $out(v_a) = 0$ and $in(va) > 0$. There must be only one leaf node in a computational graph which is typically the

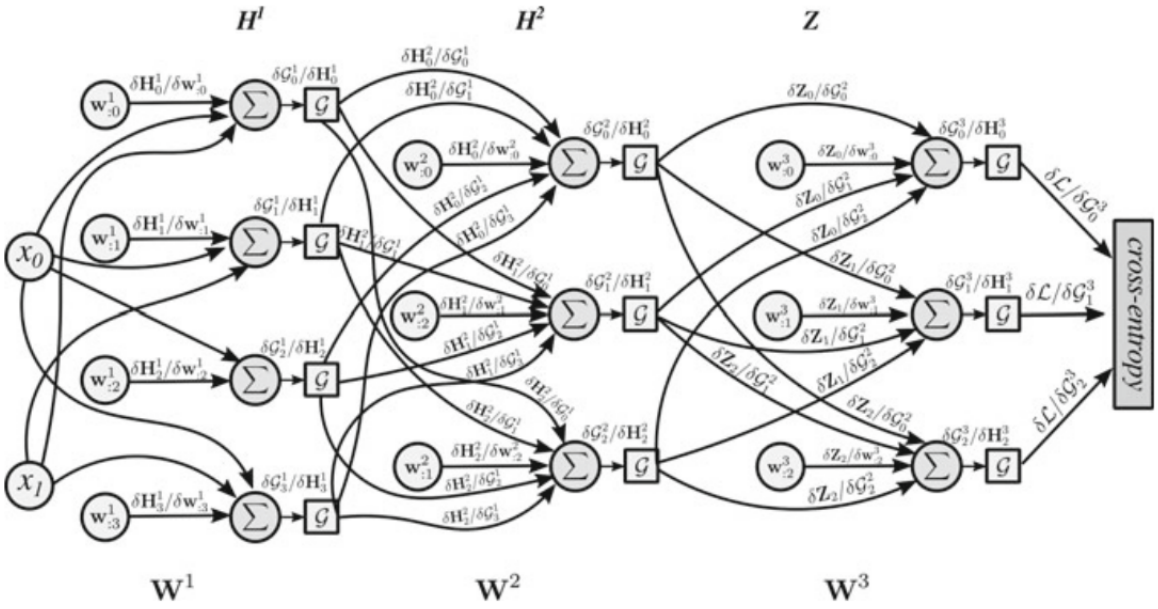

Figure 3: The network accepts two-dimensional inputs and it has two hidden layers. Each neuron has two inputs including the weights and inputs from previous layer. The gradient of each node with respect to each input is shown on the edges.

loss. If there are more than one leaf node in the graph, the gradient of the leaf node of interest with respect to all other leaf nodes will be equal to zero, i.e. all outpute nodes must be in the same layer in a neural network. We denote the leaf node as v_{leaf} and defining the child and parent nodes as $child(v_a) = \{v_j | (v_i, v_j) \in \boldsymbol{E} \wedge v_i = v_a\}$ and $parent(v_a) = \{v_i | (v_i, v_j) \wedge \boldsymbol{E} \wedge v_j = v_a\}$, respectively. Also, the depth of v_a, $dep(v_a)$, is the number of edges of the longest path from input nodes to v_a. For any node v_i, where $dep(v_i) \geq dep(v_{leaf})$, the gradient of v_{leaf} with respect to v_i will be 0.
The backpropagation algorithm is displayed in Fig. 5.
Given an input $\boldsymbol{x}$, the data is forwarded throughout the network until it reaches to the leaf node. Then, the backpropagation algorithm is executed and the gradient of loss with respect to every node given the input $\boldsymbol{x}$ is computed. Using this gradient, the parameters vectors are updated.

3.1.3 Cross-entropy cost function

The cross-entropy cost function is defined as

$$L = -\frac{1}{n} \sum_i \sum_j \left[y_j \ln x_j^{i,Z} + (1 - y_j) \ln(1 - x_j^{i,Z}) \right], \tag{3.6}$$

where n is the number of training samples, y_j is the desired output of neuron j, in the final layer and x_i^Z is the output of the i^{th}, last layer neuron. Note that L is always positive as $x_i^Z \in \{0, 1\}$. Also, the cost entropy is close to 0 when the network output is close to the desired output.

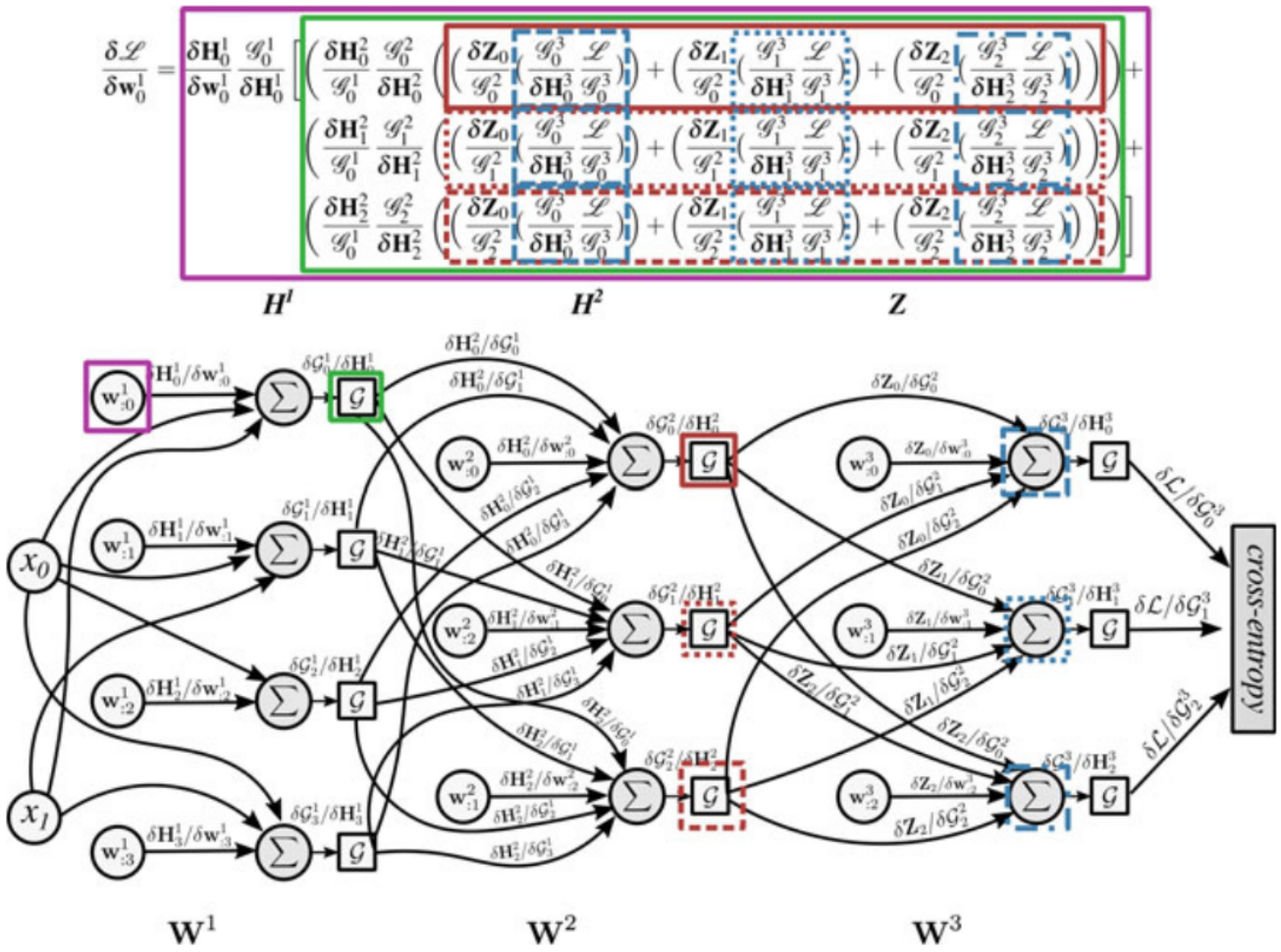

Figure 4: The network accepts two-dimensional inputs and it has two hidden layers. Each neuron has two inputs including the weights and inputs from previous layer. The gradient of each node with respect to each input is shown on the edges. The outputs of the intermediate layers is here denoted by $\boldsymbol{H}_j^i$.

Algorithm 1 The backpropagation algorithm

$G :< \mathbf{V}, \mathbf{E} >$ is a directed graph.
$\mathbf{V}$ is set of vertices
$\mathbf{E}$ is set of edges
v_{leaf} is the leaf node in $\mathbf{V}$
$d_{leaf} \leftarrow dep(v_{leaf})$
$v_{leaf}.d = 1$
for $d = d_{leaf} - 1$ to 0 **do**
  for $v_a \in \{v_i | v_i \in \mathbf{V} \wedge dep(v_i) == d\}$ **do**
    $v_a.d \leftarrow 0$
    for $v_c \in child(v_a)$ **do**
      $v_a.d \leftarrow v_a.d + \frac{\delta v_c}{\delta v_a} \times v_c.d$

Figure 5: The network accepts two-dimensional inputs and it has two hidden layers. Each neuron has two inputs including the weights and inputs from previous layer. The gradient of each node with respect to each input is shown on the edges.

The partial derivatives are given by

$$\begin{aligned}\frac{\partial L}{\partial W_{op}^{Z}} &= -\frac{1}{n}\sum_i\sum_j\left[\frac{y_j}{x_j^{i,Z}} - \frac{(1-y_j)}{(1-x_j^{i,Z})}\right]\frac{\partial x_j^{i,Z}}{\partial W_{op}^{Z}} \\ &= -\frac{1}{n}\sum_i\sum_j\left[\frac{y_j}{x_j^{i,Z}} - \frac{(1-y_j)}{(1-x_j^{i,Z})}\right]\frac{\partial x_j^{i,Z}}{\partial z_j^{i,Z}}\frac{\partial z_j^{i,Z}}{\partial W_{op}^{Z}} \\ &= \frac{1}{n}\sum_i\sum_j\frac{x_j^{i,Z}-y_j}{x_j^{i,Z}(1-x_j^{i,Z})}\frac{\partial x_j^{i,Z}}{\partial z_j^{i,Z}}x_p^{i,Z-1}\ ,\end{aligned}$$

and similarly for $\frac{\partial L}{\partial b_o^Z}$. The dependence on $x_j^{i,Z} - y_j$ tells us that the neuron will learn faster, the faster the error.

3.1.4 Activation function

We are mainly interested in activation functions that are nonlinear and continuously differentiable. A nonlinear activation function makes it possible that a neural network learns any nonlinear functions provided that the network has enough neurons and layers. Differentiability property is also important since we mainly train a neural network using gradient descend method (amoeba algorithm).
It is desirable that the activation function approximates the identity mapping near origin. activation of a neuron is given by $G(\boldsymbol{w}.\boldsymbol{x}^T + b)$ where G is the activation function. Usually, the weight vector w and bias b are initialized with values close to zero by the gradient descend method. Consequently, $\boldsymbol{w}.\boldsymbol{x}^T + b$ will be close to zero. If G approximates the identity function near zero, its gradient will be approximately equal to its input. In other words, $\delta G \approx \boldsymbol{w}.\boldsymbol{x}^T + b \approx 0$. In terms of the gradient descend, it is a strong gradient which helps the training algorithm to converge faster.

3.1.4.1 The Softmax activation function ,

$$G(z_i^h) = \frac{e^{z_i^h}}{\sum_j e^{z_j^h}} \tag{3.7}$$

is useful as the output from the softmax layer can be thought of as a probability distribution, as

$$\sum_i G(z_i^h) = \frac{\sum_i e^{z_i^h}}{\sum_j e^{z_j^h}} = 1\ .$$

Hence, it is especially useful in the outer layer. By differentiating, one can also show that the slow learning problem is avoided with the softmax function.

3.1.5 The rectified linear unit (ReLU) activation function

,

$$G(z_i^h) = max(0, z_i^h) = \begin{cases} 0 & z_i^h < 0 \\ z_i^h & z_i^h \geq 0 \end{cases} \tag{3.8}$$

is useful for shallow (few hidden layers) networks. For the above examples, neurons stop learning when the weights saturate towards 0 or 1. But, increasing the weighted input to a rectified linear unit will never cause it to saturate, and so there is no corresponding learning slowdown. On the other hand, when the weighted input to a rectified linear unit is negative, the gradient vanishes, and so the neuron stops learning entirely.

This introduces dead neurons, which can be removed for an increase in efficiency.
Additionally, there is the parametric ReLU (PReLU), which is defined as

$$G(z_i^h, \alpha) = \begin{cases} \alpha z_i^h & z_i^h < 0 \\ z_i^h & z_i^h \geq 0 \end{cases} \tag{3.9}$$

allowing for a gradient when $z_i^h < 0$.

3.1.6 Overfitting and regularization

Overfitting is the saturation of accuracy. When a neural network learns with a training sample, the cost continuously improves with the number of epochs. But, after a certain number of epochs, the accuracy of classification of the training sample stops improving. At that point the network only learns about details of the training sample, which is useless learning.
A sign of overfitting is the rise in cost after a certain number of epochs, when using the test sample.
One method to prevent overfitting is *early stopping*, i.e. we keep track of the accuracy and stop training when the accuracy has saturated.
Another method, *held out*, uses part of the training sample as validation. We tweak our hyperparameters with the training sample and then use the validation sample to check for signs of overfitting.
Still, one of the best ways to reduce overfitting, is to use a very large training sample.

3.1.7 Regularization

Another, very effective measure against overfitting is regularization. For *L2 renormalization*, we add an additional term to the cost function,

$$C = C_0 + \frac{\lambda}{2n} \sum_{h,\{ij\}} (W_{ij}^h)^2 \ ,$$

which is proportional to the squared sum of all weights. Here, $\lambda > 0$ is the regularization parameter. This has the effect, that the network prefers to learn small weights. Regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function.
For one batch, the weights and biases are updated according to

$$\begin{aligned} \boldsymbol{b}^h &\to \boldsymbol{b}^h - \eta \nabla_{b^h} C = \boldsymbol{b}^h - \eta \nabla_{b^h} C_0 \\ W_{ij}^h &\to W_{ij}^h - \eta \frac{\partial C}{\partial W_{ij}^h} = \left(1 - \frac{\eta\lambda}{n}\right) W_{ij}^h - \eta \frac{\partial C_0}{\partial W_{ij}^h} \end{aligned}$$

This weight rescaling is referred to as weight decay.
For networks with small weights, as will tend to happen in a regularized network,the behaviour of the network won't change much if we change a few random inputs here and there. So, single pieces of evidence don't matter too much to the output of the network. Hence, regularized networks are constrained to build relatively simple models based on patterns seen often in the training data, and are resistant to learning peculiarities of the noise in the training data.
Alternatively, *L1 regularization* is based on the same idea, but uses $|W_{ij}^h|$ instead of $(W_{ij}^h)^2$.
The *dropout* method modifies the network itself. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. We forward-propagate one batch $\boldsymbol{x}_i^0$ through the modified network, and then backpropagate the result, also through the modified network. Then we update the appropriate weights and biases. We then restore the dropout neurons and repeat for the next batch.
When we actually run the full network that means that twice as many hidden neurons will be active. To compensate for that, we halve the weights outgoing from the hidden neurons.

3.1.8 Weight initialization

One way to initialize weights is to use random numbers out of a Gaussian distribution. As $\boldsymbol{z}^h = \boldsymbol{W}^h \boldsymbol{x}^h + \boldsymbol{b}^h$, z_i^h will also follow a Gaussian distribution. We want to avoid that z_i^h becomes too large, as saturation would occur. To avoid saturation, we choose mean 0 and standard deviation $1/\sqrt{n_{in}}$, where n_{in} is the number of input weights of a neuron. This ensures a sharply peaked distribution. For the biases we choose mean 0 and standard deviation 1.

3.1.9 Hyperparameter optimization

First, one should try to get a signal at all - to guess hyperparameters so the neural network is better than chance.
One can estimate an order of magnitude learning rate by starting with $\eta = 0.01$. If the cost decreases during the first few epochs, then we successively try $\eta = 0.1, 1.0, \ldots$ until we find a value where the cost oscillates or increases during the first few epochs. Alternately, if the cost oscillates or increases during the first few epochs when $\eta = 0.01$, then we try $\eta = 0.001, 0.0001, \ldots$ until we find a value where the cost decreases during the first few epochs.
It's often advantageous to vary the learning rate. A larger learning rate is better for the beginning, when the weights are wrong. The a smaller learning rate is more advantageous. We can hold the learning rate constant until the validation accuracy starts to get worse - then decrease it by a certain factor. This process can be repeated many times.
The number of epochs is determined by the early stopping method. Once the accuracy stops increasing, we terminate. One can play around with the threshold number of epochs, i.e. termination after ca certain number of epochs of non-improvement.
The regularization parameter is best set to $\lambda = 0$ until the above parameters have been found. Then we can use a similar approach as for the learning rate, starting at $\lambda = 1$.
The choice of batch size at which the speed is maximized is relatively independent of the other hyper-parameters (apart from the overall architecture). One can plot the validation accuracy versus time, and choose whichever batch size gives the most rapid improvement in performance.
However, the above process is usually automated. A common technique is grid search, which systematically searches through a grid in hyper-parameter space.

3.2 Convolutional neural network

Convolutional neural networks are a specific type of feedforward networks. They use three basic ideas: local receptive fields, shared weights, and pooling. The general architecture consists of a *Convolutional Layer*, a *Pooling Layer*, and a *Fully-Connected Layer*.

3.2.1 Convolutional layer

In convolutional neural networks, each layer is a 2D array of neurons. Connections to the next layer are made only in localized regions, the *local receptive field* (filter size). This is shown in Fig. 6. Each connection learns a weight. And the hidden neuron learns an overall bias. Hence, each hidden neuron learns to analyse its particular receptive field.
Three hyperparameters control the size of the output volume: the *depth*, *stride* and *zero-padding*.

- The *depth* of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input.
- The *stride length* tells us how far we shift the local receptive field - in Fig. 6 the stride length is 1.
- *Zero-padding* means, to pad the input volume with zeros around the border. This allows us to control the spatial size of the output volumes.

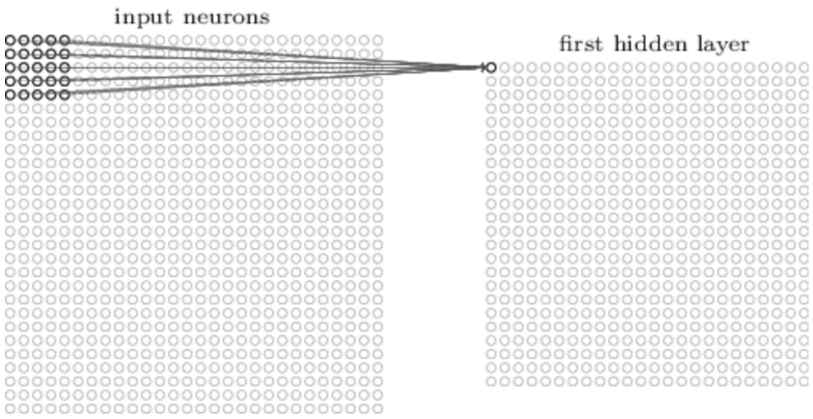

Figure 6: The general architecture of a convolutional neural network.

The number of output neurons is given by

$$N_{out} = \frac{D_{in} - D_{rec} + 2p}{s+1}, \tag{3.10}$$

where D_{in} is the input dimension, D_{rec} is the receptive field dimension, p is the padding and s is the stride. *Parameter sharing* scheme is used in Convolutional Layers to control the number of parameters. Each individual layer of neurons detects one specific feature of an image. We assume that the spatial position of these features is unimportant. Hence, we use the same weights and biases for all neurons in a particular hidden layer. So, the output of the i, j hidden neuron in layer h is

$$\boldsymbol{x}_{ij}^{h+1} = G\left(\sum_{l=0,m=0}^{r,r} \boldsymbol{W}_{lm}^{h} \boldsymbol{x}_{(i+l)(j+m)}^{h} + \boldsymbol{b}^{h}\right),$$

where r is the size of the receptive field and we have to add an additional dimension to the input vector and weight matrix.
We call the map from the input layer to the hidden layer a *feature map*. We call the weights defining the feature map the *shared weights*. And we call the bias defining the feature map in this way the *shared bias*. The shared weights and bias are often said to define a *kernel* or *filter*.
Usually, multiple features need to be recognised by the network, so we require multiple feature maps. This block of feature maps is called the *convolutional layer*. Hence, the layers of a convolutional neural network have neurons arranged in 3 dimensions: width, height, depth.

3.2.2 Pooling layer

The Pooling Layer operates independently on each feature map. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. A pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map - it summarizes feature map regions.
One specific procedure is *max-pooling*, where a pooling unit simply outputs the maximum activation in the 2×2 input region (stride 2). Pooling is useful, as it tells us if a specific feature has been found, discarding positional information.
In *L2 pooling*, we take the square root of the sum of the squares of the activations in the 2×2 region. L2 pooling is a way of condensing information from the convolutional layer.

3.2.3 Fully connected layer

Finally, the pooling layers are then connected to the output layer, which is fully connected - i.e. every neuron from the pooling layers is connected to every output neuron.

3.2.4 General architecture

One can treat the input volume of a fully connected layer, as input to another convolutional layer. So we can produce different stacks of layers. Generally, the above layers are usually stacked as

$$INPUT \rightarrow [[CONV \rightarrow RELU] * N \rightarrow POOL] * M \rightarrow [FC \rightarrow RELU] * K \rightarrow FC,$$

where N, M, K represent repetitions. Multiple convolution layers are advantageous, as deeper layers "see" a greater effective receptive field with respect to the input image. This has the advantage of non-linear connections between increasingly deeper layers. Additionally, the number of parameters is reduced, compared with one convolutional layer with a large receptive field.

3.2.5 Layer sizing

The input layer should dimensions should be divisible by 2 many times.
The convolution layer should be using small filters, i.e. maximally 5×5. A padding of $p = (D_{rec} - 1)/2$ preserves the input size.
The pooling is usually done by 2×2 max-pooling, with stride 2. This discards exactly 75%of the data.

3.3 Recurrent neural network

4 Code: Training a 1D convolutional neuroal network: `conv1D_FSP.py`

For my analysis, the most important files are `conv1D_FSP.py` and `smrt_expert.py`.

4.1 Data read in and preprocessing

The first part of the script is for data read in and preprocessing. Event data is stored in `*.pickle` files. First, flags are created, as shown in List. 1:

- `-i` : Path to training data pickle.
- `-o` : Training file output directory.
- `-f` : Training file output name.

Listing 1: Creating flags.

```
parser = argparse.ArgumentParser(
description='''1D CNN training. Input format: (labels, [FSPs,[FSP vars]], [event vars])''',
)
parser.add_argument('-i', type=str, nargs='+', required=True,
help="Path to training data pickle.", metavar="INPUT_FILE",
dest='in_file')
parser.add_argument('-o', type=str, required=True,
help="Training file output directory.", metavar="OUTPUT_DIR",
dest='out_dir')
parser.add_argument('-f', type=str, required=False,     default='conv1D_training.h5',
help="Training file output name.", metavar="OUTPUT_FILE",
dest='out_file')
args = parser.parse_args()
```

Next, output directories are created. Then, the `*.pickle` file is read in, stored in `in_data`. The data is stored in the format shown in List. 2.

Listing 2: Data storage format

```
[useful(1/0), [[particle 1 info], ...], [total Monte Carlo (?), total FPS, charged FPS]]
where particle info = [PDG index, Mass, Charge, Energy, Production time, x, y, z, px, py,
    pz, mother PDG index]
```

`in_data` is then shuffled. See List. 3.

Listing 3: Loading data and shuffling.

```
# Load training data
in_data = []
for f in args.in_file:
        in_data += pickle.load(open(f, 'rb'))

# Mix
shuffle(in_data)
```

Next, the set for training is prepared. Events without final state particles (FSP) are sorted out and we take about 50/50 of useful and useless events.

Listing 4: Preparing trianing set. Sorting out zero FSP events. Here, `el[0]` is the target value, i.e. is this an event of interest or not. `y_count` is a temporary index, so that not too many useless events are carried through.

```
# Crop to balance pass and fail event numbers
norm_data = []
y_count = 0

for el in in_data:
        # Sort out events with no FSPs
        if len(el[1]) == 0:
                print('Event with 0 FSPs and label {}, skipping'.format(el[0]))
                continue
        # Fail events
        if y_count < 1 and el[0] == 0:
                norm_data.append(el)
                y_count += 1
        elif el[0] == 1:
                norm_data.append(el)
                y_count = 0
in_data = norm_data
```

Then we find the maximum FSP event, as shown in List. 5. This is useful to specify input dimension and padding.

Listing 5: Find maximum FSP.

```
max_FSPs = max(map(len, [e[1] for e in in_data]))
print('Max FSPs:', max_FSPs)
```

Now, the events and target values are separated into `x_list` and `y_list`, respectively, as seen in List. 6. Here, we only take some of the particle information to be fed to the neural network and pad the rest with 0, to the dimension of maximum FPS.

Listing 6: Separation of target values and event information.

```
y_list = []
x_list = []
# evt_list = []

# Collect our data
```

```
for el in in_data:
        y_list.append(el[0])
        # evt_list.append(el[2])
        # PDG_p = [[a[0]] + a[8:11] for a in el[1]],
        x_list.append(
                np.pad(
                # Try just PDG, px, py, pz
                # PDG_p,
                [[a[0]] + a[8:11] for a in el[1]],
                # sorted(PDG_p, key=lambda a: (a[0])),
                # Sort by |p|
                # sorted(el[1], key=lambda a: (a[8]**2 + a[9]**2 + a[10]**2)),
                # Sort by mother PDG
                # sorted(el[1], key=lambda a: (a[-1])),
                # el[1],
                ((0, max_FSPs - len(el[1])), (0, 0)),
                mode='constant',
                constant_values=0
        )
)
```

The last step before starting the neural network is outputting the number of parameters, used for each particle and the total number of events.

4.2 Neural network initialisation

First, we separate the input data into training and test events, as shown in List 7. Initially a training fraction of `t_frac = 0.9` was implemented.

Listing 7: Separation of trianing and test events.

```
#keras.utils.to_categorical() converts a class vector (integers) to binary class matrix.
y_train = keras.utils.to_categorical(np.array(y_list[:int(t_frac * num_data)]), num_classes
    =2)
y_test = keras.utils.to_categorical(np.array(y_list[int(t_frac * num_data):]), num_classes
    =2)

x_train = np.array(x_list[:int(t_frac * num_data)])
x_test = np.array(x_list[int(t_frac * num_data):])
```

Next, the training and test data particle info is normalized according to its mean and standard deviation, in each respective category, as shown in List. 8. Normalization is necessary so gradients for backpropagation don't become too small.

Listing 8: Normalization of training an test data.

```
x_mean = np.mean(x_train, axis=(0, 1), keepdims=True)
x_std = np.std(x_train, axis=(0, 1), keepdims=True)
x_train = (x_train - x_mean) / x_std
x_test = (x_test - x_mean) / x_std
```

Now we start the network configuration, shown in List. 9.

Listing 9: Network architecture.

```
# ## The Network stuff
batch_size = 64
epochs = 100

# Convolutional model
convnet = Sequential()
convnet.add(Conv1D(32, 3, padding='same', input_shape=(None, num_FSP_vars)))
convnet.add(LeakyReLU())
convnet.add(Conv1D(32, 3))
```

```
convnet.add(LeakyReLU())
convnet.add(GlobalAveragePooling1D())
convnet.add(Dropout(0.25))
convnet.add(Dense(128))
convnet.add(LeakyReLU())
convnet.add(Dropout(0.5))
convnet.add(Dense(2, activation='sigmoid'))
```

We first add a 1D convolutional neural network with 32 filters, a kernel size 3 - the length of the convolution window. The padding is so that the output has the same length as the input. The input shape takes the number of parameters, included in the training/test data. ??? We add a LeakyReLU activation layer, which behaves similar to the PReLU function, Eq. 3.9. These two layers are repeated once more. Then we have a global average pooling layer, which outputs the average activation if its input region. Next, we perform the dropout procedure, dropping 25% of the neurons - randomly setting a fraction of input units to 0 at each update during training time, which helps prevent overfitting. Next, a regular, densely connected layer with 132 outputs. Then another LeakyReLU activation layer and a 50% dropout. Lastly, we have another dense layer with 2 outputs and sigmoid activation.
Next, we set up the compilation of the network. As an optimiser (like stochastic gradient descend) we use Adagrad, with a learning rate of 0.01, as shown in List. 10.

Listing 10: Network architecture.

```
adagrad = keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)

convnet.compile(loss='categorical_crossentropy',
        optimizer=adagrad,
        metrics=['accuracy'])
```

We also want an output to `TensorBoard`, shown in List. 11.

Listing 11: Output so we can examine performance in `TensorBoard`.

```
# Want tensorboard output to assess training
# Make separate subdirs for logs, needed for run separation
tbCallBack = keras.callbacks.TensorBoard(
        log_dir=os.path.join(args.out_dir, 'logs', now),
        histogram_freq=1,
        write_graph=True,
        write_grads=True,
        batch_size=32,
        write_images=True,
)
```

Before starting the training, we implement a few additional features, shown in List. 12.

Listing 12: Additional featrues: stop trianing early, reduce learning rate when training plateaus, save the model after every epoch

```
# Stop training if it's not improving
earlyStop = keras.callbacks.EarlyStopping(monitor='val_acc', min_delta=0, patience=17,
    verbose=0, mode='auto')

# Reduce learning rate when training plateaus
rlrop = keras.callbacks.ReduceLROnPlateau(monitor='loss', factor=0.1, patience=8, verbose
    =0, mode='auto', min_lr=0.)

# Save the model after every epoch, allows us to kill training and resume it later
modelCheckpoint = keras.callbacks.ModelCheckpoint(os.path.join(out_dir, 'train_checkpoint.
    h5'), monitor='val_loss', verbose=0, save_best_only=False, save_weights_only=False,
    mode='auto', period=1)
```

`EarlyStopping` monitores the accuracy in this case and stopt training if there was a `delta=0` change in the last 17 epochs. `ReduceLROnPlateau` monitors the loss and decreases the learning rate when no improvement

has been observed within the last 8 epochs. `ModelCheckpoint` monitors the loss and saves the model after every epoch.
Now we move on to training the network, as shown in List. 13.

Listing 13: Network training.

```
try:
        convnet.fit(
                x_train,
                y_train,
                batch_size=batch_size,
                epochs=epochs,
                validation_data=(x_test, y_test),
                callbacks=[tbCallBack, rlrop, modelCheckpoint],
        )
```

The batch size is 64 and the number of epochs is 100, as shown in List. 9.
Finally we save everything and evaluate the network on the test data, as displayed in List. 14.

Listing 14: Saving, evaluating on test data and outputting the result.

```
finally:
        convnet.save(os.path.join(out_dir, args.out_file))

        # Also save our normalisation values used
        xmeanstd = np.array([x_mean, x_std])
        xmeanstd.dump(os.path.join(out_dir, args.out_file + '.norm.pickle'))
        score = convnet.evaluate(x_test, y_test, batch_size=batch_size)
        print('\n', score)
```

5 Training data

I wrote a small script, as shown in List. 15, to combine specific training data sets.

Listing 15: Combining `EvtGen_CNN.pickle` files.

```
# code for combining all the trianing data files of a certain type

import os
import pickle

try:
os.remove("EvtGen_CNN_total00.pickle")
except OSError:
pass

total = []

for root, dirs, files in os.walk("/srv/data/jkahn/output/smrt_gen/gen_init_rec/mixed/sub00"
    ):
for file in files:
if file.endswith("EvtGen_CNN.pickle"):
print(os.path.join(root, file))
total += pickle.load(open(os.path.join(root, file), 'rb'))

pickle.dump(total, open("EvtGen_CNN_total00.pickle", "wb"))
```

The reasoning behind this, is that larger training sets will result in more consistent results - before we had a training event size of 1200, of which 10% was validation data. The spread of results, event with the same network architecture and hyperparameters was very large. This makes comparisons and effects of small adjustments to the network impossible to see. The combined data of one `subXX` folder contains 25000 events. Now, this was shown to be a bit inconvenient for adjusting, as evaluation takes quite long. So will mainly

work with half of that.

6 Improvements

6.1 Preprocessing

The balancing of useful and useless data, as shown in List. 4, was improved to ensure an exact balance, with a better algorithm for selection. The improved code is shown in List. **??**.

Listing 16: The improved selection algorithm. The data shuffeling occurs in two stages now, on both the separated and the combined data set.

```
# Crop to balance pass and fail event numbers
data_0 = []
data_1 = []

for el in in_data:
        # Not sure why but there's an event with no FSPs?
        if len(el[1]) == 0:
                print('Event with 0 FSPs and label {}, skipping'.format(el[0]))
                continue
        # Fail events
        if el[0]==0:
                data_0.append(el)
        elif el[0]==1:
                data_1.append(el)

#mix data
shuffle(data_0)
shuffle(data_1)

#ensure data is balanced
data_diff = len(data_0) - len(data_1)
if data_diff > 0:
        data_0 = data_0[:-abs(data_diff)]
elif data_diff < 0:
        data_0 = data_0[:-abs(data_diff)]
in_data = data_0 + data_1

# Mix that shit up
shuffle(in_data)
```

6.2 Input

Here, we have the problem that the PDG index is not easily normalizable, as it is not a continuous parameter. But, if it is not normalized, we have the problem that handing large numbers to the network results in shallow gradients and the learning saturates. Now, even if we decrease the numbers, the network will still try to learn something continuous.
We have two possible solutions for this problem: one-hot vectors or embedding. One-hot vectors is basically a vector of dimension corresponding to the number of possible particles - all entries are 0, except for one, which is 1. The disadvantage is, that we would have to feed the network a massive amount of inputs. The second possible solution, embedding, is a mapping from discrete objects, such as words, to vectors of real numbers. But I am also not sure if embedding helps, because the PDG index is already numerical.
What I also tried was to abandon the PDG index and use the mass to classify the particles. Basted on a few tries, this works as well. As the mass is not highly correlated with any other variables (baste on Nico Reebs thesis), it could be are useful to include. Also, the charge should be included. Maybe as a one-hot vector.
Completely abandoning the PDG index seems not good though, as it is also a very uncorrelated variable.

Now, I have added a 3 category one-hot vector for charge. For some reason, directly adding the charged did not improve the accuracy. Even the one-hot vector did not improve the accuracy, but also didn't make it worse.
I also tried to input the total momentum, instead of each component separately. This also did not seem to make a difference.

6.2.1 PDG index

The current neural network, and everything done by James, was limited to a validation accuracy of 0.68. This might be an indication for a physical limitation, or just, that we do not supply enough useful information to the network.
Out of this reasoning, I will write a script, to decode the PDG index - as this contains information about particle spins, their quark content etc.
The first run, replacing the PDG index input just by the information if it is a particle or antiparticle, and its spin, the network showed no signs of improvement.
Additionally, including the quark content made no difference either.

6.3 Padding and normalization

The original code padded the input to the number of maximum final state particle with vectors of entries 0. This is slow and messes up the normalization. It would be good if we could change that to just handing a list of Numpy arrays of different lengths to the network.

6.4 Optimizers

Initially we had the Adagrad optimizer implemented. Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates.
We tried the Adadelta optimizer next. Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done. The learning rate had to be left at the default 1.0, otherwise results were disastrous. Although the performance on the training data was better than before, the validation results showed no significant improvement.
The Adam optimizer, proposed in the paper Adam: A method for stochastic optimization, showed an improvement of about 2%, compare to the best run before that. This was only true, if the learning rate was set to the default value. This optimizer performed best in all four categories - (validation-)accuracy and (validation-)loss. There is a bit more fluctuation in the validation curves visible, compared to other optimisers. I suspect, that a run with even more data will result in smoother curves. The validation accuracy, running trough half the data of sune `subXX` folder (12000 events), was at 0.6541 and similar with the full dataset. The loss was 0.5876 and 0.6048 respectively.
The Adamax optimizer, also in the above paper, a variant of Adam based on the infinity norm, also had a good performance, but not quite as good as Adam.
Lastly I tried the Nadam optimizer. It is much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum. This performed exceptionally well in all four categories, with similar results to the Adam optimizer. It had a faster increase in validation accuracy, but seemed to hit the same barrier as Adam, with a final validation accuracy of 0.6519 for half the data and 0.6481 for the full set. The loss was 0.5815 and 0.5926.

6.5 General architecture

I tried increasing the input size from 32 to 64 - this had a minimal negative effect and was abandoned.
Next, I added another 1D convolutional layer. This increased the training accuracy to over 0.75 and the loss

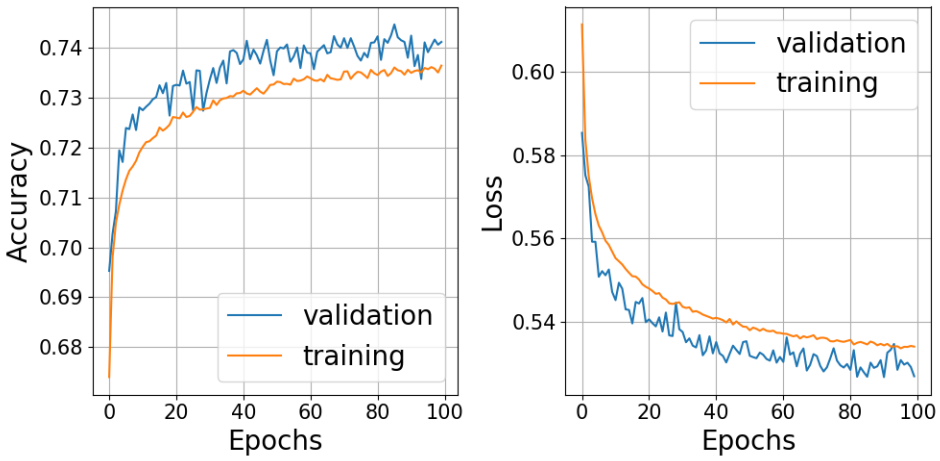

Figure 7: The training results of the network described above, with the tag-side reconstructed data.

decreased to 0.4972. But, the validation accuracy still saturated at 0.65 - this is a clear sign of overfitting.
Next, I decreased the input size of the second 1D convolutional layer from 32 to 16. This Worked surprisingly well, leading to a validation accuracy of 0.6683 and loss of 0.5849.
Adding another, 8 input layer decreased the accuracy again.
For a new set of training data: `mdst_*.pickle` and `udst_*.pickle`, which includes not the reconstruction of he signal side,a s used before, but only the reconstruction of the tag-side, the 32-32 architecture worked better. The results are shown in Fig. 7. We see, that this data works much better with the network, leading to a validation accuracy of 0.735.

6.5.1 Masking and Normalization

As the above idea of excluding the padding did not work, I adapted a new method. `Keras` provides additional layers which I will implement to avoid the normalization problem and maybe even improve normalization overall. I included the following layers as the initial layers of the network:

- `Masking`: This masks out the undesired gaps in the data, which arise from the difference in final state particles. *Turns out the 1D convolutional layer do not support masking...*
- `BatchNormalization`: This layer replaces the part of the code responsible for the data normalization, List. 8. The method draws its strength from making normalization a part of the model architecture and performing the normalization for each training batch.

Turns out, the normalization layer does not help at all either.
But the general above approach would be very good, if it was implemented and worked. I also applied the approach to the recurrent neural network which is built of `LSTM` layers. But the performance was still poor.

6.6 Tweaks

Changed the mean and standard deviation calculation to include the complete data. Probably makes a marginal difference, but still - to be exact.
Tried not normalizing the PDG index as it not a continuous variable. This only seemed to decreased the accuracy though.
Tried to change the loss function from `categorical_crossentropy` to `binary_crossentropy`, as we are looking for a binary output. No real improvement could be seen. In this case we want hot-one vectors output - so we have two outputs. This is why we use `categorical_crossentropy`, but it does not make much difference.

6.7 API model

For this, the above code was completely restructured. I decided to split everything in classes, to keep things organized. The complete code is shown in A new approach is to take the above model, but add additional first layers. The *PDG index* will be sent through an embedding layer, the mass and momenta will be sent through a normalization layer, and the charge will just be left as a one hot vector. The outputs of these layers will then be combined and input to the above network.

- The embedding should be done with a `input_dim` that corresponds to the number of different particle types in the data. The `input_length` is the maximum number of final state particles and the `output_dim` will have to be tested.
- The normalization layer is pretty straight forward, as described above. But I have found out that it is best to include the normalization layer between the non-linear and linear layers in the network, and not before. So I continued to normalize the normalizable data (mass, momenta) and moved the normalization layer in front of the first fully connected layer.

The combination of data is done with `keras.layers.Concatenate(axis=-1)`. Anyhow, this idea seems not to be the best.
Next, I want to try separating all non-correlated variables and building a sub-network for each. First, I'll try finding a good way to embed the PDG index. I'll try embedding it to a high dimension – probably 10 – and then running it through a 1D convolutional network, before combining the output with the rest of the data. The plan is to do something similar for the mother PDG and maybe combine the two PDG `conv1D` outputs, and running them through another one or two layers, before merging with the rest of the data. The charge one-hot vector might also run through two dense layers before merger.
In the end, we have three subnetwork architectures to figure out - the single PDG, after embedding, the network, after the merger of the two embedding layers and the last network, which gets fed the complete data/outputs.
I will focus on the embedding technique for now. The newest set of data, I am looking at, contains the total decay tree - i.e. all particles in the collision. They are sorted according to their parent index - which describes which particle they come from.

6.7.1 Decay string embedding

For this I will use the same technique, usually used for word embedding in neural networks. The decay string represents the complete complete decay, an example is shown in List. 17.

Listing 17: A decay string example. The numbers represent PDG indices.

```
300553 (--> -511 (--> 423 (--> 421 (--> 310 (--> 211 -211 <--) 211 -211 <--) 111 (--> 22 22
     <--) <--) 3212 (--> 22 3122 (--> 2212 -211 <--) <--) -2114 (--> -2112 111 (--> 22 22
    <--) <--) <--) 511 (--> 211 -211 -411 (--> 323 (--> 311 (--> 310 (--> 111 (--> 22 22
    <--) 111 (--> 22 22 <--) <--) <--) 211 <--) -211 -211 <--) 211 111 (--> 22 22 <--) <--)
     <--)
```

First I tokenize the decay string, consisting of PDG indices and the separators ($(--)$ and $(<--)$), which represent decays. For this, I use a list of all PDG indices, plus the separators, saved in `evt.pdl`. In the decay string, each PDG index and separator is replaced with a 1D vector, containing the line number in the list file. The complete decay string vector is then flattened, giving a variable length list of indices. This list is then padded with 0 from above using `keras.preprocessing.sequence.pad_sequences()`.
For now, the other particle information (PDG, mass, momenta, charge) run through the usual convolutional network. The decay string is embedded and then the two are concatenated. What is also worth trying is adding a few layers after the embedding, before the concatenation.
The first network, I tried is shown in Fig. 8. The training is very slow, due to the LSTM layer. But the validation accuracy is better than anything we have achieved before. With about 300000 training events, the network reached a validation accuracy of 0.78 and loss of 0.477 after only 20 epochs. From there on

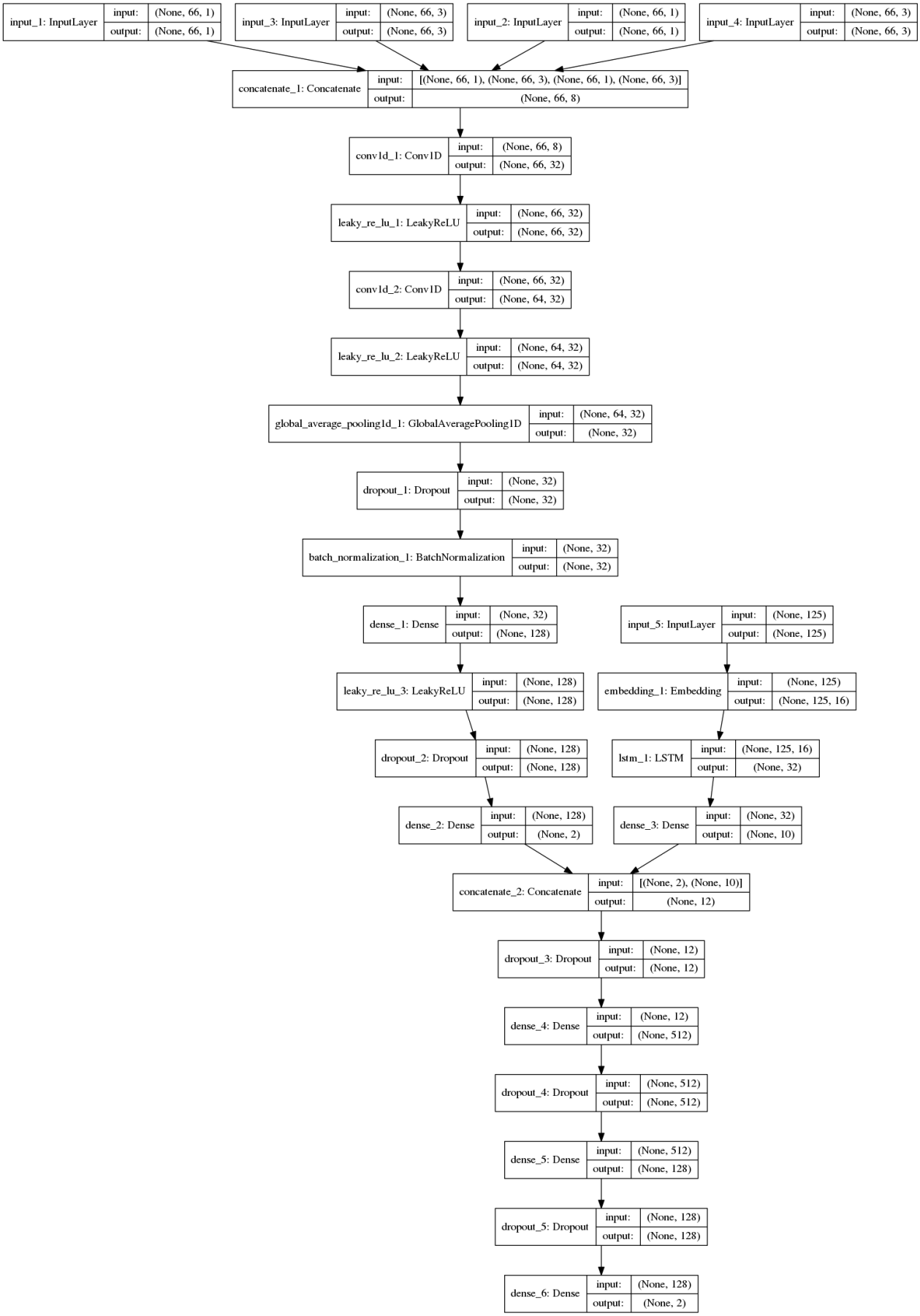

Figure 8: The first API neural network including the decay string. The inputs 1-4 are the usual inputs, PDG (unchanged), momenta (x,y,z), mass and charge (one-hot), respectively.

the validation accuracy flattened off and the network stopped training after 44 epochs. Still, this is a great improvement upon earlier work. But there seems to be a great dependence on the amount of training data, as there is a 3% difference in validation accuracy when the training data is reduced to 50000 events.
Next, I am trying to find a suitable 1D convolutional network to replace the LSTM. If successful, this will improve the speed drastically. A pure convolutional network did not seem to work that well. The next approach is something proposed in the literature - a combination of convolutional and LSTM layer. This reduces the input to LSTM and increases training speed. I have tried different architectures with 50000 events, to see which one might be best to test with the full data set. The problem with so little data was, that the network started to show signs of overfitting. This was reflected in fluctuating validation accuracy, while the training accuracy constantly inclined. The network I settled for now is displayed in Fig. 9 The above model performed quite well, achieving a validation accuracy of 0.7853 and a validation loss of 0.4781. The results are shown in Fig. 10. The network performed very good, right form the start. Unfortunately, the validation accuracy did not improve much over the training period and fluctuations are high. Still, this is the best model so far.
The fluctuations are an indication for overfitting. To minimise this, I tried changing the architecture slightly, as shown in Fig. 11. The smaller LSTM layer has improved speed by a factor of 60%, so that each epoch only takes about 10min, instead of 30min, for around 500000 events. The fluctuations slightly decreased, leading to a more continuous rise in validation accuracy. The validation accuracy has also improved slightly to 0.7901 and a validation loss of 0.4611, which is a slight improvement to the model above. The training also only ran over 50 epochs, but with a batch size of 128, instead of 64. The results are shown in Fig. 12. The second plot shows the results for running over a maximum of 100 epochs and the two, second to last dropout layers increased to a rate of 0.5 instead of 0.25. Here, we see an early stopping at epoch 51, but much less fluctuation. The final validation results are 0.7876 and 0.4568 for accuracy and loss, respectively.

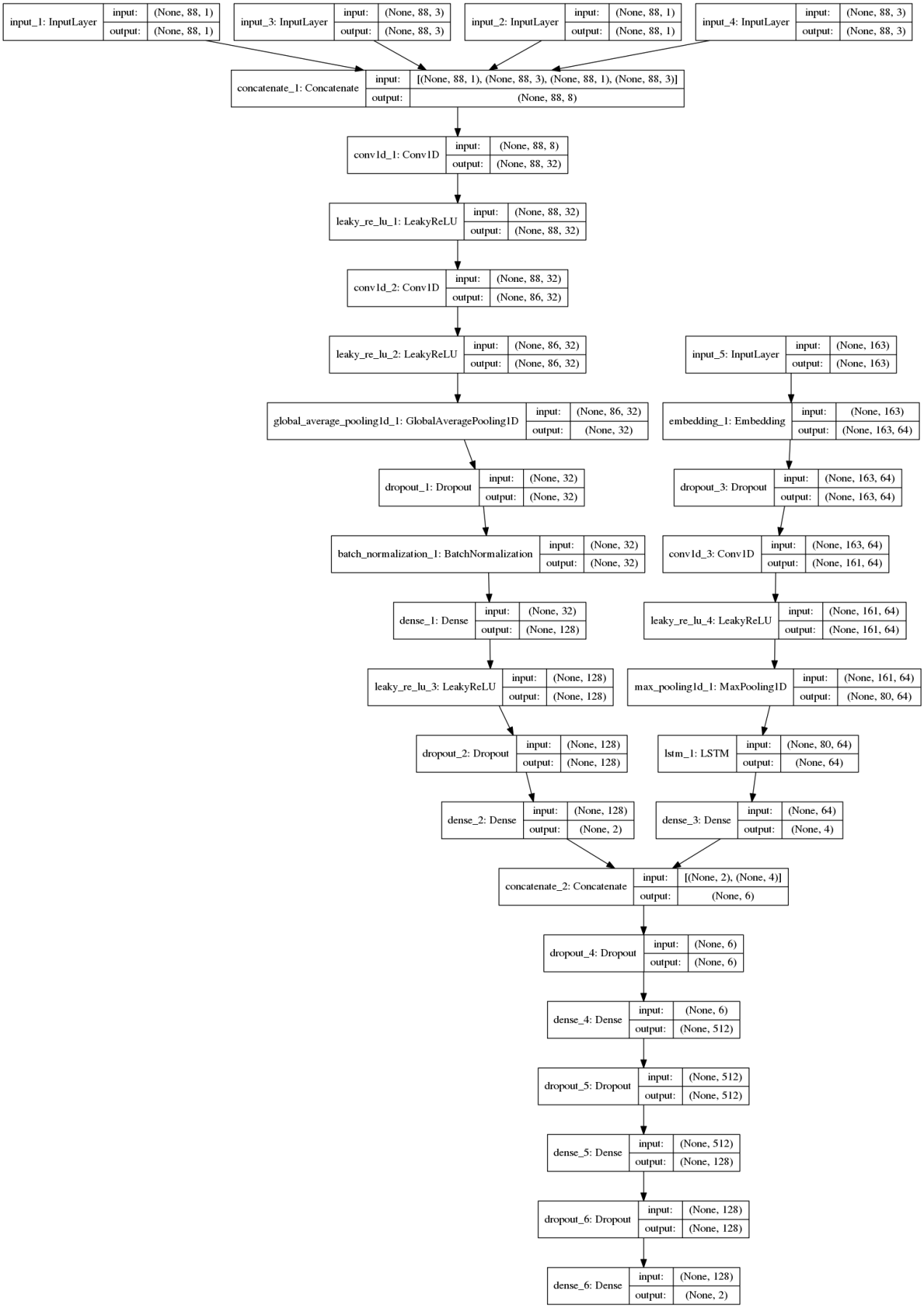

Figure 9: The second API neural network including the decay string. The decay string is now processed by a convolutional layer, before the LSTM layer.

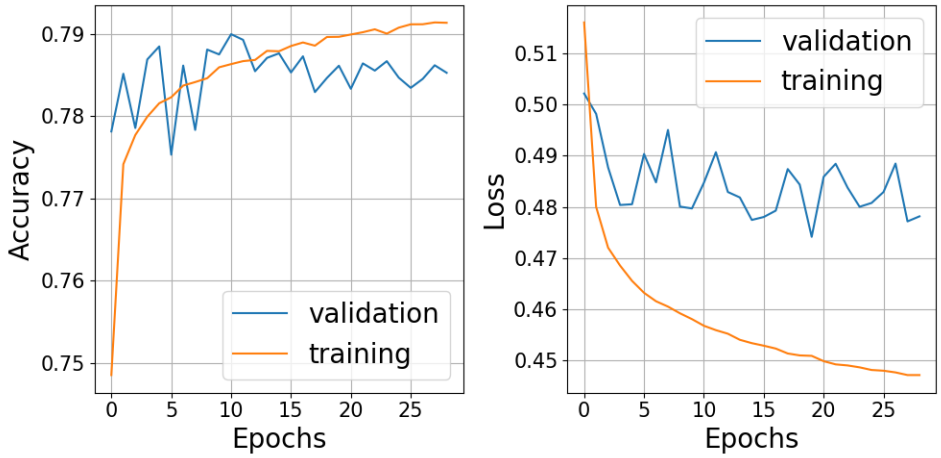

Figure 10: The training results of the network described above. The network stopped improving quite quickly, leading to an early stop, after 29 epochs.

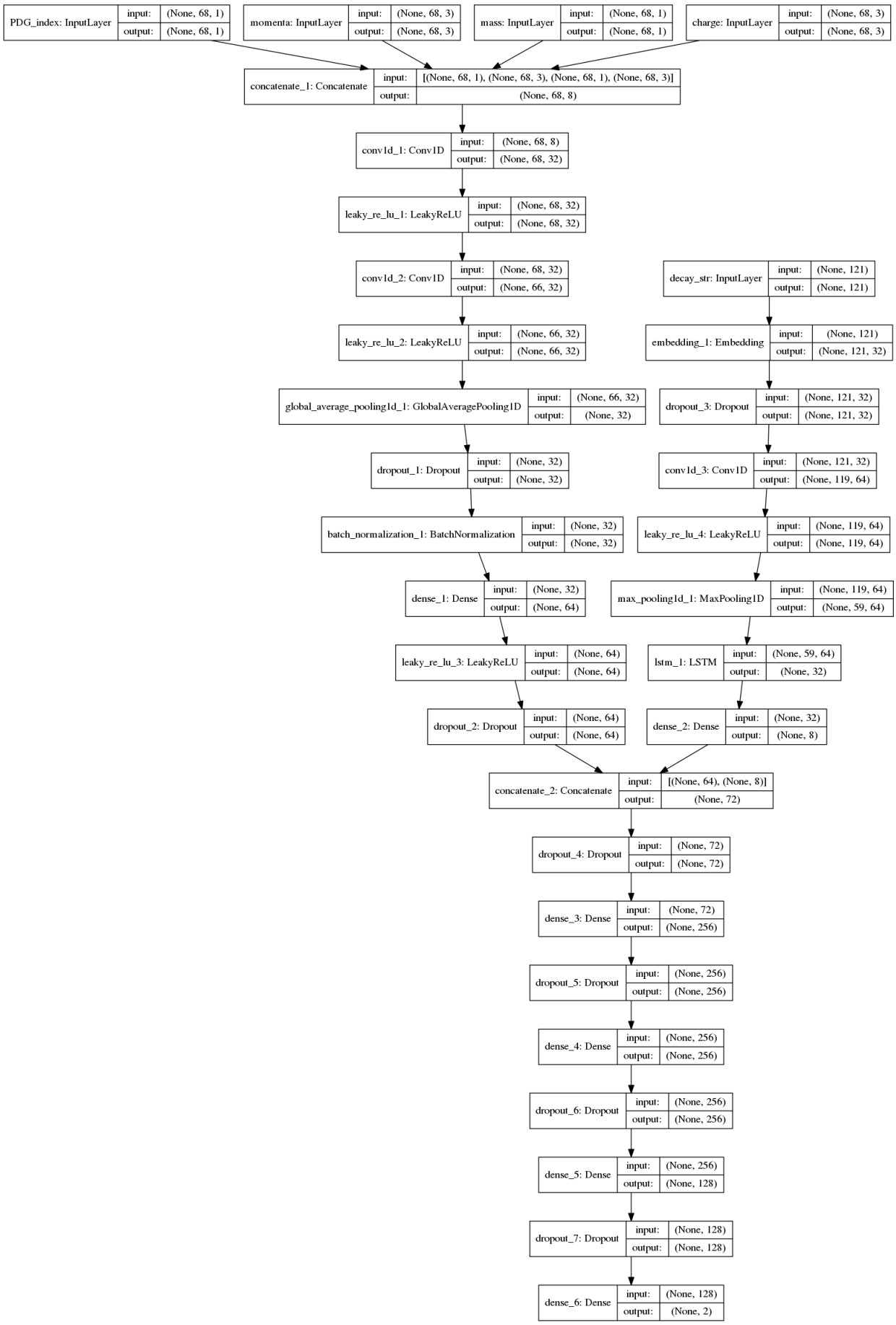

Figure 11: The third API model, which has a reduced embedding space and the final, fully-connected network has been modified slightly.

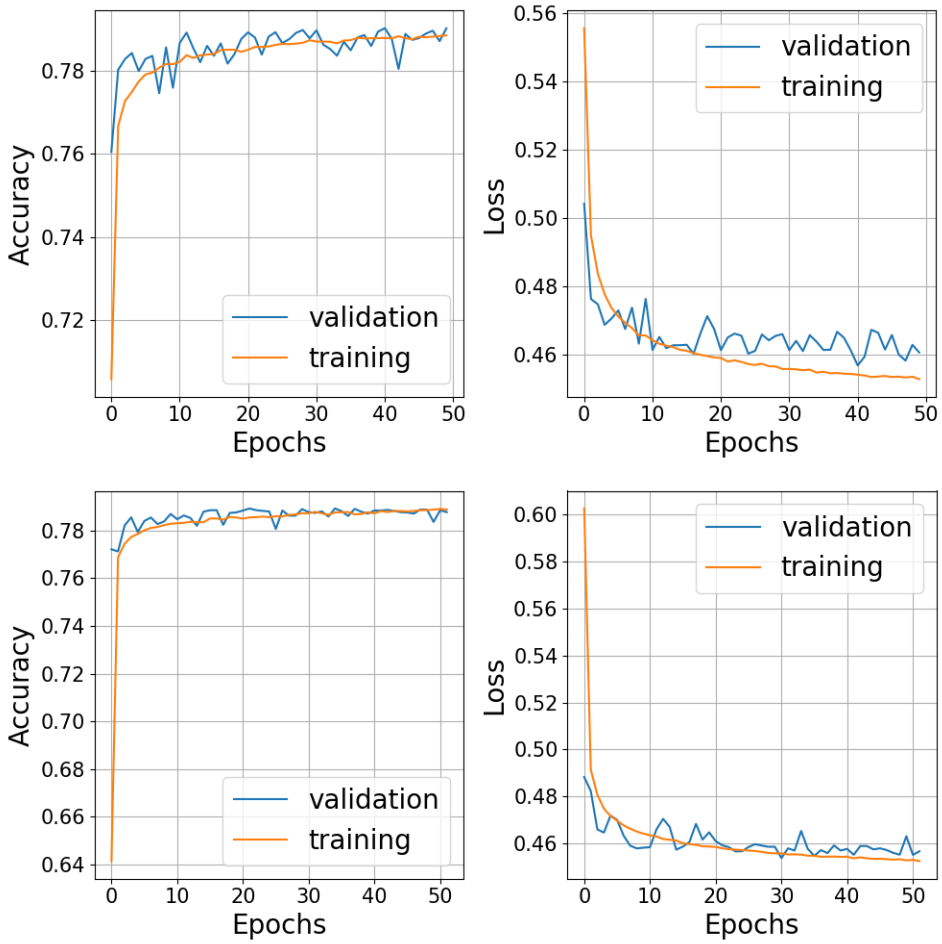

Figure 12: The results of the third API model. The fluctuations have been reduced slightly, which is good. Also the performance was better and the validation data matches the training data very well, which is a sign, that little overfitting is taking place. The second plot shows the result with an epoch limit of 100. Here, the training automatically stopped after 51 epochs.

Appendices

A

The full API_FSP.py script

Below is the full, restructured script for the API network model.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Train convolutional 1D network on dataaaa
# James Kahn

import pickle
import argparse
import os
import time
from random import shuffle
import numpy as np
import keras

from keras.models import Model, Input
from keras.layers import Dense, Dropout
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D
from keras.layers import LeakyReLU, LSTM
from keras.layers import BatchNormalization, Embedding
from keras.layers import concatenate
from keras.preprocessing import text, sequence
from keras.utils import plot_model
# from keras.layers import *
# from keras.callbacks import Tensorboard

# DON'T USE GPU - only while other trianing is running
# os.environ["CUDA_VISIBLE_DEVICES"] = "-1"

# ------------------------------------------------------------------------------


class prep_input:
    def __init__(self, args):
        self.in_data = []
        self.load_data(args.in_file)
        # self.exclude_statusbit()
        self.extract_data()
        self.tokenize_decay_string()
        self.separate_data()
        self.train_mass = self.normalize_data(self.train_mass)
        self.test_mass = self.normalize_data(self.test_mass)
        self.train_momenta = self.normalize_data(self.train_momenta)
        self.test_momenta = self.normalize_data(self.test_momenta)

    def load_data(self, files):
        ''' Load training data, balance and shuffle'''
        for file in files:
            self.in_data += pickle.load(open(file, 'rb'))

        # Crop to balance pass and fail event numbers
        data_0 = []
        data_1 = []

        for event in self.in_data:
            # Not sure why but there's an event with no FSPs?
```

```
            if not event[1]:
                print('Event with 0 FSPs and label {}, skipping'.format(event[0]))
                continue
            # Fail events
            if event[0] == 0:
                data_0.append(event)
            elif event[0] == 1:
                data_1.append(event)

        # mix data
        shuffle(data_0)
        shuffle(data_1)

        # ensure data is balanced
        data_diff = len(data_0) - len(data_1)
        if data_diff > 0:
            data_0 = data_0[:-abs(data_diff)]
        elif data_diff < 0:
            data_0 = data_0[:-abs(data_diff)]
        self.in_data = data_0 + data_1

        # Mix that shit up
        shuffle(self.in_data)
        self.in_data = self.in_data[:10]

    def sort_data(self):
        '''sort data according to key'''
        for event in self.in_data:
            event[1].sort(key=event[1][13])

    def exclude_statusbit(self):
        '''exclude particles with certain status bits'''
        self.in_data = [[el[0]] +
                        [[a for a in el[1]
                          if not ((a[13] & 16 == 16) |
                                  (a[13] & 32 == 32) |
                                  (a[13] & 64 == 64) |
                                  (a[13] & 128 == 128) |
                                  (a[13] & 256 == 256))]] +
                        [el[2]] for el in self.in_data]

    def extract_data(self):
        '''extract the different particle properties from the
        input and put them in seperate arrays'''
        self.max_fsps = max(map(len, [e[1] for e in self.in_data]))
        print('Max FSPs:', self.max_fsps)

        self.labels = []    # event labels
        self.list = []      # list of full data
        self.pdg = []       # PDG indices
        self.mass = []
        self.momenta = []   # x,y,z
        self.charge = []
        self.mother_ind = []
        self.decay_str = []

        # Collect our data
        for event in self.in_data:
            # Pad data equally
            pad_bef = (self.max_fsps - len(event[1]))
            pad_aft = 0
            self.labels.append(event[0])
            self.pdg.append(np.pad(
```

```
                [[a[0]] for a in event[1]],
                ((pad_bef, pad_aft), (0, 0)),
                mode='constant',
                constant_values=0))
            self.mass.append(np.pad(
                [[a[1]] for a in event[1]],
                ((pad_bef, pad_aft), (0, 0)),
                mode='constant',
                constant_values=0))
            self.momenta.append(np.pad(
                [a[8:11] for a in event[1]],
                ((pad_bef, pad_aft), (0, 0)),
                mode='constant',
                constant_values=0))
            self.charge.append(np.pad(
                [(keras.utils.to_categorical(a[2], num_classes=3)).flatten()
                 for a in event[1]],
                ((pad_bef, pad_aft), (0, 0)),
                mode='constant',
                constant_values=0))
            self.mother_ind.append(np.pad(
                [[a[13]] for a in event[1]],
                ((pad_bef, pad_aft), (0, 0)),
                mode='constant',
                constant_values=0))
            self.list.append(np.pad(
                # Try just pdg, px, py, pz
                # pdg_p,
                [np.concatenate(
                    (a[0:2], a[8:11], (keras.utils.to_categorical(a[2], num_classes=3)).
                        flatten())
                ) for a in event[1]],
                ((pad_bef, pad_aft), (0, 0)),
                mode='constant',
                constant_values=0))
            self.decay_str.append(event[2][3])

    def normalize_data(self, data):
        # Normalize data
        # Eventually move to gen_var_saver when I've decided on best scheme
        # [0,1] norm across each channel individually
        # Not sure if the padding will mess this up, probably?
        # Need to figure out how to avoid that

        mean = np.mean(data, axis=(0, 1), keepdims=True)
        std = np.std(data, axis=(0, 1), keepdims=True)
        norm = (data - mean) / std
        return norm

    def separate_data(self):
        '''separate the extracted data into test and training sets'''
        # How many events we'll process
        num_data = len(self.in_data)
        print('Num data:', num_data)

        # Training fraction
        t_frac = 0.9

        # This should (I hope) give us the correct shape
        # ONLY USING END OF LAST INPUT FILE AS TEST,
        # SHOULD BE END OF EVERY FILE
        self.train_labels = keras.utils.to_categorical(
            self.labels[:int(t_frac * num_data)], num_classes=2)
```

```
        self.test_labels = keras.utils.to_categorical(
            self.labels[int(t_frac * num_data):], num_classes=2)

        self.train_list = np.array(self.list[:int(t_frac * num_data)])
        self.test_list = np.array(self.list[int(t_frac * num_data):])

        self.train_pdg = np.array(self.pdg[:int(t_frac * num_data)])
        self.test_pdg = np.array(self.pdg[int(t_frac * num_data):])

        self.train_mass = np.array(self.mass[:int(t_frac * num_data)])
        self.test_mass = np.array(self.mass[int(t_frac * num_data):])

        self.train_momenta = np.array(self.momenta[:int(t_frac * num_data)])
        self.test_momenta = np.array(self.momenta[int(t_frac * num_data):])

        self.train_charge = np.array(self.charge[:int(t_frac * num_data)])
        self.test_charge = np.array(self.charge[int(t_frac * num_data):])

        self.train_decay_str = self.decay_str[:int(t_frac * num_data)]
        self.test_decay_str = self.decay_str[int(t_frac * num_data):]

    def PDGdecompose(self, index):
        '''decomposition of pdg index'''
        # ANTI-/PARTICLE
        if np.sign(index) > 0:
            decompose = '01'
        elif np.sign(index) < 0:
            decompose = '10'
        else:
            print('FormatError: PDG index error')
            exit()
        for i in range(0, 7 - len(str(np.absolute(index)))):
            decompose += '0'
        for i in str(np.absolute(index)):
            decompose += i
        print(index, decompose)
        return decompose

    def tokenize_decay_string(self):
        '''replace the decay string with its corresponding tokens,
        specified by the evt.pdl file'''
        with open('evt.pdl', 'r') as file:
            evt_file = file.read().splitlines()

        self.num_pdg_codes = len(evt_file)
        tokenize = text.Tokenizer(num_words=self.num_pdg_codes,
                                  filters='!"#$%&*+,./:;=?@[\]^_`{|}~')
        tokenize.fit_on_texts(evt_file)
        decay_str_tok = []
        for decay in self.decay_str:
            tok = tokenize.texts_to_sequences(decay.split())
            decay_str_tok.append(
                [i for sub in tok for i in sub]
            )
        self.decay_str = decay_str_tok
        self.decay_str = sequence.pad_sequences(self.decay_str)
        print(self.decay_str[3])

# ------------------------------------------------------------------------


class conv_net_1D():
    # NETWORK CLASS
```

```
    def __init__(self, data, args, batch_size=128, epochs=100):
        self.num_pdg_codes = data.num_pdg_codes
        self.batch_size = batch_size
        self.epochs = epochs

        self.output_dir(args.out_dir)

        self.net_inputs(data)
        self.net_layers()
        callbacks = self.callbacks(args)

        model = Model(inputs=[self.input_pdg, self.input_mass,
                              self.input_momenta, self.input_charge, self.input_decay_str],
                      outputs=[self.net_output])
        adam = keras.optimizers.Adam(lr=0.001)  # best so far

        model.compile(loss='categorical_crossentropy',
                      optimizer=adam,
                      metrics=['accuracy'])

        # visualize model
        plot_model(model, to_file=os.path.join(args.out_dir, 'model.png'),
                   show_shapes=True, show_layer_names=True)

        try:
            model.fit(
                [data.train_pdg, data.train_mass, data.train_momenta,
                 data.train_charge, data.train_decay_str],
                data.train_labels,
                batch_size=self.batch_size,
                epochs=epochs,
                # validation_split=1.0-t_frac,
                validation_data=([data.test_pdg, data.test_mass,
                                  data.test_momenta, data.test_charge,
                                  data.test_decay_str],
                                 data.test_labels),
                callbacks=callbacks,
                verbose=1
            )
        finally:
            model.save(os.path.join(args.out_dir, args.out_file))
            # convnet.save(os.path.join(out_dir, args.out_file))

            # Also save our normalisation values used
            # xmeanstd = np.array([x_mean, x_std])
            # xmeanstd.dump(os.path.join(
            #           out_dir, args.out_file + '.norm.pickle'))
            self.score = model.evaluate([data.test_pdg, data.test_mass,
                                         data.test_momenta, data.test_charge,
                                         data.test_decay_str],
                                        data.test_labels, batch_size=self.batch_size)

    def output_dir(self, dire):
        '''Creates output directory'''
        # Create output dir
        self.now = time.strftime("%Y.%m.%d.%H.%M")
        self.out_dir = os.path.join(dire, self.now)
        if not os.path.exists(self.out_dir):
            # Try creating the output directory
            try:
                os.makedirs(self.out_dir)
            except OSError as err:
                print("OS error: {0}".format(err))
```

```
    def net_inputs(self, data):
        '''define the network inputs'''
        num_fsp_vars = len(data.list[0][0])
        print('Num FSP vars:', num_fsp_vars)
        num_fsp_pdg = len(data.pdg[0][0])
        num_fsp_mass = len(data.mass[0][0])
        num_fsp_p = len(data.momenta[0][0])
        num_fsp_q = len(data.charge[0][0])
        self.len_decay_str = len(data.decay_str[0])
        print(data.decay_str.shape)

        self.input_pdg = Input(shape=(data.max_fsps, num_fsp_pdg),
                               name='PDG_index')
        self.input_mass = Input(shape=(data.max_fsps, num_fsp_mass),
                                name='mass')
        self.input_momenta = Input(shape=(data.max_fsps, num_fsp_p),
                                   name='momenta')
        self.input_charge = Input(shape=(data.max_fsps, num_fsp_q),
                                  name='charge')
        self.input_decay_str = Input(shape=(self.len_decay_str,),
                                     name='decay_str')

    def net_layers(self):
        '''define the network layers'''
        x_particle = concatenate([self.input_pdg, self.input_momenta,
                                  self.input_mass, self.input_charge], axis=-1)

        x_particle = Conv1D(32, 3, padding='same')(x_particle)
        x_particle = LeakyReLU()(x_particle)
        x_particle = Conv1D(32, 3)(x_particle)
        x_particle = LeakyReLU()(x_particle)
        # x_particle = MaxPooling1D()(x_particle)
        # x_particle = Conv1D(16, 2)(x_particle)
        x_particle = GlobalAveragePooling1D()(x_particle)
        x_particle = Dropout(0.25)(x_particle)
        x_particle = BatchNormalization(axis=1)(x_particle)

        x_particle = Dense(64)(x_particle)
        x_particle = LeakyReLU()(x_particle)
        x_particle = Dropout(0.5)(x_particle)
        # x_particle = Dense(2, activation='sigmoid')(x_particle)

        x_decay_str = Embedding(self.num_pdg_codes, 32,
                                input_length=self.len_decay_str)(self.input_decay_str)
        x_decay_str = Dropout(0.25)(x_decay_str)
        x_decay_str = Conv1D(64, 3, padding='valid')(x_decay_str)
        x_decay_str = LeakyReLU()(x_decay_str)
        x_decay_str = MaxPooling1D()(x_decay_str)  # global max pooling
        x_decay_str = LSTM(32)(x_decay_str)
        x_decay_str = Dense(16, activation='softmax')(x_decay_str)

        decay = concatenate([x_particle, x_decay_str], axis=-1)
        decay = Dropout(0.25)(decay)
        decay = Dense(256, activation='relu')(decay)
        decay = Dropout(0.5)(decay)
        decay = Dense(256, activation='relu')(decay)
        decay = Dropout(0.5)(decay)
        decay = Dense(128, activation='relu')(decay)
        decay = Dropout(0.25)(decay)
        self.net_output = Dense(2, activation='sigmoid')(decay)

    def callbacks(self, args):
```

```
        '''define callbacks for network, details below'''
        # Want tensorboard output to assess training
        # Make separate subdirs for logs, needed for run separation
        tensorboard = keras.callbacks.TensorBoard(
            log_dir=os.path.join(args.out_dir, 'logs', self.now),
            histogram_freq=0,
            write_graph=True,
            write_grads=True,
            batch_size=self.batch_size,
            write_images=True,
        )

        # Stop training if it's not improving
        earlystop = keras.callbacks.EarlyStopping(monitor='val_acc',
                                                  min_delta=0, patience=17,
                                                  verbose=0, mode='auto')

        # Reduce learning rate when training plateaus
        rlrop = keras.callbacks.ReduceLROnPlateau(monitor='loss', factor=0.1,
                                                  patience=8, verbose=0,
                                                  mode='auto', min_lr=0.)

        # Save the model after every epoch, allows us to kill training and
        # resume it later
        modelcheckpoint = keras.callbacks.ModelCheckpoint(
            os.path.join(self.out_dir, 'train_checkpoint.h5'),
            monitor='val_loss', verbose=0, save_best_only=False,
            save_weights_only=False, mode='auto', period=1)

        return [tensorboard, earlystop, rlrop, modelcheckpoint]
# ------------------------------------------------------------------------------


if __name__ == '__main__':

    # Read args
    PARSER = argparse.ArgumentParser(
        description='''1D CNN training. Input format: (labels, [FSPs,[FSP vars]], [event
            vars])''',)
    PARSER.add_argument('-i', type=str, nargs='+', required=True,
                        help="Path to training data pickle.",
                        metavar="INPUT_FILE", dest='in_file')
    PARSER.add_argument('-o', type=str, required=True,
                        help="Training file output directory.",
                        metavar="OUTPUT_DIR", dest='out_dir')
    PARSER.add_argument('-f', type=str, required=False,
                        default='conv1D_training.h5',
                        help="Training file output name.",
                        metavar="OUTPUT_FILE", dest='out_file')
    ARGS = PARSER.parse_args()

    INPUT_DATA = prep_input(ARGS)

    NETWORK = conv_net_1D(INPUT_DATA, ARGS)
    print(NETWORK.score)
```