Unit 14 Case Study: Modeling High-Energy Physics Data with a Hybrid Convolutional Neural Network

Damon Resnick
Collaborators: Rajni Goyal and Jim Hosker

April 19, 2018

Abstract

In this study we create a hybrid convolutional neural network to model the 11.1 million rows and 28 columns of data from a simulation of the output of high-energy particles. This data was taken from a paper by Baldi, Sadowski, and Whiteson [1]. They outline how using neural networks and deep learning can be used to improve the performance of signal-versus-background classification problems. In this study we try to replicate the general technique they use and provide our own model to approximate the accuracy they obtain. Our best model produced n ROC AUC score of 0.8727.

1 Introduction

This case study focuses on generating a hybrid convolutional neural network (CNN) to model the 11.1 million rows and 28 columns of data from a simulation of the output of high-energy particles. Since new particle detection in high-energy physics depends a lot on separating signal from background noise the study follows the blueprint that the paper by Baldi et. al. [1]. That paper details how deep learning models can be used to improve the classification of signal and noise in the dense data generated by high-energy particles detection.

This study focusses on modeling the data and answering these questions:

Q1: What was the effect of adding more layers/neurons?
Q2: Which parameters gave you the best result and why (in your opinion) did they work?
Q3: How did you decide that your model was 'done'?

For this analysis, the Keras and TensorFlow Python library packages were used [2, 3].

2 Data

The data has been produced by Monte Carlo simulations and used to show the efficacy of using complex Deep Learning neural network approaches to classify high-energy particle data. It was provided by Daniel Whiteson from UC Irvine, posted to the Machine Learning Repository (MLR), and analyzed in the paper by Baldi, Sadowski, and Whiteson [1].

The data set has 29 columns and is broken up into three classes. The first column is the target class label (1 for signal, 0 for background), the next 21 columns are kinematic properties measured by

the particle detectors in the accelerator, and the last 7 columns are features that are functions of the previous 21 columns that were derived to help discriminate between the two class labels. More details about each feature can be obtained from the paper and the MLR website.

3 Modeling the Data

A hybrid complex convolutional neural network algorithm was used to model the data. The hyper parameters for the models were chosen using a subset of the data consisting of 2.6 million training examples and 100,000 test examples. The optimization for the model included combinations of the network architectures, initial learning rates, batch size, kernel initializers, optimizers, and regularization methods. We selected a three-layer neural network with 512 hidden units for our input layer and first hidden layer with 256 inputs for second hidden layer, a learning rate of 0.01, and a weight decay coefficient of 10^{-6}. It should be noted that extra hidden units, and additional hidden layers significantly increased training time with noticeably increasing performance. To facilitate comparison, deep neural networks were trained with the different hyper-parameters and the different number of units per hidden layer. Additional details are provided below. The Keras model library of python is used to create our neural network models.

3.1 Model Steps and Details

To initialize the model an input shape is specified. The first layer in a *Sequential()* model (and only the first, because the following layers can do automatic shape inference) needs to receive information about its input shape. In our analysis, the *Dense()* function is used which has 2D layers to support the specification of their input shape via the argument *input_dim*, and some 3D temporal layers that support the arguments *input_dim* and *input*_length. In addition, additional models, optimizers, *Dense*, *Dropout*, *Activation* and *ROC* functions are loaded as libraries to perform our analysis.

Before training a model, the learning process needs to be configured which is done via the compile method. It receives three arguments:

- An optimizer: In this analysis, SGD, rmsprop, Adam and Adagrad optimizers are used.
- A loss function: In our analysis, "binary_crossentropy" is used.
- A list of metrics: In our analysis, the metric is: accuracy and the loss is an output.

The base code is shown below. This code establishes a base set of inputs for the models that will be used on the data. In addition, four different models are examined and modified in order to identify a better model. Each input will be addressed. Inputs that are modified to determine a better model are the following:

1) 3-layer architecture with 512 neurons in the input layer and 2nd layer, and a 3rd layer with 256 neurons, a learning rate of 0.01, and "relu" as the activation function in all layers except for the sigmoid in output layer.

2) 4-layer architecture with 300 neurons in all layers and a learning rate of 0.01 and "relu" as the activation function in all layers except for the sigmoid in output layer.

3) 4-layer architecture with 300 neurons in all layers and a learning rate of 0.05 and "relu" as the activation function in all layers except for the sigmoid in output layer.

4) 4-layer architecture with 300 neurons in all layers and a learning rate of 0.05 and “tanh” as the activation function in all layers except for the sigmoid in output layer.

3.1.1 Base Inputs

Here are the base settings for our models before the analysis:

read.csv: Training data set of 2.6 million entries and test data set of 1 million entries. This will not vary in our analysis until we run on 80% of the data in a later model.

```
data=pd.read_csv("./HIGGS/HIGGS.csv",nrows=2600000,header=None)
test_data=pd.read_csv("./HIGGS/HIGGS.csv",nrows=100000,header=None,skiprows=2600000)
```

model.add and *density* functions: Input dimensions using 28 input variables, kernel initialization of 'uniform', activation is 'sigmoid'. This will vary in a later model.

```
model = Sequential()
model.add(Dense(600, input_dim=x.shape[1], kernel_initializer='uniform'))
model.add(Activation('sigmoid'))
```

The final layer, output layer, uses the 'sigmoid' activation function. This was never changed since this is common practice for classification of two outcomes. A ‘softmax’ function could have also been used but was unnecessary.

```
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))
```

```
model.dropout: 10%
model.add(Dropout(0.10))
```

Optimizer: Initally use stochastic gradient descent (SGD), learning start (lr) of 0.01, learning decay is 10^{-6}, momentum of 0.9 and nesterov = TRUE. Some of these values are changed for later models.

```
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile: Optimizer is SGD, loss calculation is 'binary_crossentropy', and metrics will be ‘accuracy'
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)
```

model.fit: Number of epochs is 40 and batch size is 100.

```
model.fit(x, y, epochs=40, batch_size=100)
```

4 Results

The code and results of the 4 different architectures detailed above are presented below:

1) 3-layer architecture with 512 neurons in the input layer and 2nd layer, and a 3rd layer with 256 neurons, a learning rate of 0.01, and “relu” as the activation function in all layers except for the sigmoid in output layer.

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15 here
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)
```

```
model.fit(x, y, epochs=40, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 292s 112us/step - loss: 0.4578 - acc: 0.7779
Epoch 36/40
2600000/2600000 [==============================] - 277s 107us/step - loss: 0.4572 - acc: 0.7782
Epoch 37/40
2600000/2600000 [==============================] - 276s 106us/step - loss: 0.4565 - acc: 0.7786
Epoch 38/40
2600000/2600000 [==============================] - 285s 110us/step - loss: 0.4559 - acc: 0.7789
Epoch 39/40
2600000/2600000 [==============================] - 335s 129us/step - loss: 0.4554 - acc: 0.7792
Epoch 40/40
2600000/2600000 [==============================] - 317s 122us/step - loss: 0.4547 - acc: 0.7797
100000/100000 [==============================] - 3s 33us/step
0.8579306989674672
```

2) 4-layer architecture with 300 neurons in all layers and a learning rate of 0.01 and “relu” as the activation function in all layers except for the sigmoid in output layer.

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] ==
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)
```

```
model.fit(x, y, epochs=40, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 191s 74us/step - loss: 0.5208 - acc: 0.7365
Epoch 36/40
2600000/2600000 [==============================] - 190s 73us/step - loss: 0.5201 - acc: 0.7368
Epoch 37/40
2600000/2600000 [==============================] - 191s 73us/step - loss: 0.5201 - acc: 0.7369
Epoch 38/40
2600000/2600000 [==============================] - 189s 73us/step - loss: 0.5195 - acc: 0.7375
Epoch 39/40
2600000/2600000 [==============================] - 191s 73us/step - loss: 0.5194 - acc: 0.7374
Epoch 40/40
2600000/2600000 [==============================] - 203s 78us/step - loss: 0.5189 - acc: 0.73770s
100000/100000 [==============================] - 3s 26us/step
0.835923240159103
```

3) 4-layer architecture with 300 neurons in all layers and a learning rate of 0.05 and “relu” as the activation function in all layers except for the sigmoid in output layer.

```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD
from sklearn.metrics import roc_auc_score
```

```
model = Sequential()
model.add(Dense(300, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.05, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
Epoch 35/40
2600000/2600000 [==============================] - 199s 77us/step - loss: 0.5298 - acc: 0.7317
Epoch 36/40
2600000/2600000 [==============================] - 223s 86us/step - loss: 0.5294 - acc: 0.7316
Epoch 37/40
2600000/2600000 [==============================] - 233s 90us/step - loss: 0.5290 - acc: 0.7321
Epoch 38/40
2600000/2600000 [==============================] - 231s 89us/step - loss: 0.5288 - acc: 0.7323
Epoch 39/40
2600000/2600000 [==============================] - 230s 88us/step - loss: 0.5281 - acc: 0.7326
Epoch 40/40
2600000/2600000 [==============================] - 229s 88us/step - loss: 0.5280 - acc: 0.7330
100000/100000 [==============================] - 2s 24us/step
0.8308408932271054
```

4) 4-layer architecture with 300 neurons in all layers and a learning rate of 0.05 and “tanh” as the activation function in all layers except for the sigmoid in output layer.

```
model = Sequential()
model.add(Dense(300, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15 here
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('tanh'))
#model.add(Dropout(0.5)
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.05, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
model.fit(x, y, epochs=40, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 190s 73us/step - loss: 0.5559 - acc: 0.7134
Epoch 36/40
2600000/2600000 [==============================] - 200s 77us/step - loss: 0.5552 - acc: 0.7133
Epoch 37/40
2600000/2600000 [==============================] - 200s 77us/step - loss: 0.5545 - acc: 0.7139
Epoch 38/40
2600000/2600000 [==============================] - 197s 76us/step - loss: 0.5538 - acc: 0.7142
Epoch 39/40
2600000/2600000 [==============================] - 217s 83us/step - loss: 0.5533 - acc: 0.7148
Epoch 40/40
2600000/2600000 [==============================] - 218s 84us/step - loss: 0.5525 - acc: 0.7151
500000/500000 [==============================] - 13s 25us/step
0.8207052044706236
```

Now using those architectures, we ran the code using different activation functions.

2 different activation functions used are: - 1st model with “softplus” and “sigmoid”

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] ==
model.add(Activation('softplus'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('softplus'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('softplus'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)
```

```
model.fit(x, y, epochs=5, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
WARNING:tensorflow:Variable *= will be deprecated. Use variable.assign_mul if you want assignment to
x * y' if you want a new python Tensor object.
Epoch 1/5
2600000/2600000 [==============================] - 336s 129us/step - loss: 0.6580 - acc: 0.5980
Epoch 2/5
2600000/2600000 [==============================] - 342s 132us/step - loss: 0.6270 - acc: 0.6426
Epoch 3/5
2600000/2600000 [==============================] - 334s 128us/step - loss: 0.6073 - acc: 0.6661
Epoch 4/5
2600000/2600000 [==============================] - 339s 130us/step - loss: 0.5977 - acc: 0.6750
Epoch 5/5
2600000/2600000 [==============================] - 365s 140us/step - loss: 0.5911 - acc: 0.6807
100000/100000 [==============================] - 4s 43us/step
0.759464139786276
```

2nd model with “softplus” and “sigmoid”

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1]
model.add(Activation('softplus'))
model.add(Dropout(0.5))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('softplus'))
model.add(Dropout(0.5))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('softplus'))
model.add(Dropout(0.5))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('softplus'))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)
```

```
model.fit(x, y, epochs=40, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 36/40
2600000/2600000 [==============================] - 276s 106us/step - loss: 0.5900 - acc: 0.6831
Epoch 37/40
2600000/2600000 [==============================] - 254s 98us/step - loss: 0.5892 - acc: 0.6834
Epoch 38/40
2600000/2600000 [==============================] - 242s 93us/step - loss: 0.5887 - acc: 0.6842
Epoch 39/40
2600000/2600000 [==============================] - 260s 100us/step - loss: 0.5883 - acc: 0.6847
Epoch 40/40
2600000/2600000 [==============================] - 251s 97us/step - loss: 0.5878 - acc: 0.6851
100000/100000 [==============================] - 3s 33us/step
0.7679694192477359
```

3rd model with “softsign” and “sigmoid”

```
model = Sequential()
model.add(Dense(300, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15 here
model.add(Activation('softsign'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('softsign'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('softsign'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('softsign'))
#model.add(Dropout(0.5)
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.05, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)
```

```
model.fit(x, y, epochs=5, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
WARNING:tensorflow:Variable *= will be deprecated. Use variable.assign_mul if you want assignment to the var
x * y' if you want a new python Tensor object.
Epoch 1/5
2600000/2600000 [==============================] - 215s 83us/step - loss: 0.6131 - acc: 0.6609
Epoch 2/5
2600000/2600000 [==============================] - 218s 84us/step - loss: 0.5885 - acc: 0.6856
Epoch 3/5
2600000/2600000 [==============================] - 215s 83us/step - loss: 0.5808 - acc: 0.6921
Epoch 4/5
2600000/2600000 [==============================] - 214s 82us/step - loss: 0.5758 - acc: 0.6964
Epoch 5/5
2600000/2600000 [==============================] - 208s 80us/step - loss: 0.5723 - acc: 0.6993
500000/500000 [==============================] - 11s 22us/step

0.7933619486818193
```

4th model with “linear” and “sigmoid”

```
model = Sequential()
model.add(Dense(300, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15
model.add(Activation('linear'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('linear'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('linear'))
model.add(Dropout(0.5))
model.add(Dense(300, kernel_initializer='uniform'))
model.add(Activation('linear'))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.05, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)
```

```
model.fit(x, y, epochs=5, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
WARNING:tensorflow:Variable *= will be deprecated. Use variable.assign_mul if you want assignment to th
x * y' if you want a new python Tensor object.
Epoch 1/5
2600000/2600000 [==============================] - 203s 78us/step - loss: 0.6476 - acc: 0.6262
Epoch 2/5
2600000/2600000 [==============================] - 210s 81us/step - loss: 0.6449 - acc: 0.63270s - loss
Epoch 3/5
2600000/2600000 [==============================] - 211s 81us/step - loss: 0.6444 - acc: 0.6340
Epoch 4/5
2600000/2600000 [==============================] - 210s 81us/step - loss: 0.6440 - acc: 0.6349
Epoch 5/5
2600000/2600000 [==============================] - 210s 81us/step - loss: 0.6438 - acc: 0.6355
100000/100000 [==============================] - 2s 24us/step

0.6826638809046681
```

We took our best model from parts 1 & 2 and vary the batch_size = 10000

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
model.fit(x, y, epochs=40, batch_size=10000)
score = model.evaluate(x_test, y_test, batch_size=10000)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 174s 67us/step - loss: 0.5263 - acc: 0.7325
Epoch 36/40
2600000/2600000 [==============================] - 174s 67us/step - loss: 0.5251 - acc: 0.7333
Epoch 37/40
2600000/2600000 [==============================] - 173s 67us/step - loss: 0.5240 - acc: 0.7341
Epoch 38/40
2600000/2600000 [==============================] - 175s 67us/step - loss: 0.5230 - acc: 0.7347
Epoch 39/40
2600000/2600000 [==============================] - 174s 67us/step - loss: 0.5220 - acc: 0.7353
Epoch 40/40
2600000/2600000 [==============================] - 172s 66us/step - loss: 0.5208 - acc: 0.7362
100000/100000 [==============================] - 2s 16us/step

0.823501124005926
```

We took our best model from parts 1 & 2 and vary the Batch_size = 100000

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15 here
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
model.fit(x, y, epochs=5, batch_size=100000)
score = model.evaluate(x_test, y_test, batch_size=100000)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 1/5
2600000/2600000 [==============================] - 224s 86us/step - loss: 0.6925 - acc: 0.5280
Epoch 2/5
2600000/2600000 [==============================] - 203s 78us/step - loss: 0.6916 - acc: 0.5296
Epoch 3/5
2600000/2600000 [==============================] - 209s 80us/step - loss: 0.6914 - acc: 0.5296
Epoch 4/5
2600000/2600000 [==============================] - 199s 76us/step - loss: 0.6913 - acc: 0.5296
Epoch 5/5
2600000/2600000 [==============================] - 194s 75us/step - loss: 0.6912 - acc: 0.5296
100000/100000 [==============================] - 2s 19us/step

0.5512528713897988
```

We took our best model and then used 3 different kernel initializers.
kernel_initiliazers: Normal

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='normal')) # X_train.shape[1] == 15 here
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='normal'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='normal'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='normal'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
model.fit(x, y, epochs=40, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 278s 107us/step - loss: 0.4609 - acc: 0.7758
Epoch 36/40
2600000/2600000 [==============================] - 277s 107us/step - loss: 0.4602 - acc: 0.7763
Epoch 37/40
2600000/2600000 [==============================] - 276s 106us/step - loss: 0.4597 - acc: 0.7766
Epoch 38/40
2600000/2600000 [==============================] - 275s 106us/step - loss: 0.4590 - acc: 0.7773
Epoch 39/40
2600000/2600000 [==============================] - 277s 106us/step - loss: 0.4584 - acc: 0.7773
Epoch 40/40
2600000/2600000 [==============================] - 277s 107us/step - loss: 0.4578 - acc: 0.7778
100000/100000 [==============================] - 3s 30us/step
0.8575451272240283
```

Random normal

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='random_normal')) # X_train.shape[1] == 15 here
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='random_normal'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='random_normal'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='random_normal'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
model.fit(x, y, epochs=40, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 275s 106us/step - loss: 0.4613 - acc: 0.7755
Epoch 36/40
2600000/2600000 [==============================] - 276s 106us/step - loss: 0.4608 - acc: 0.7759
Epoch 37/40
2600000/2600000 [==============================] - 275s 106us/step - loss: 0.4599 - acc: 0.7765
Epoch 38/40
2600000/2600000 [==============================] - 277s 106us/step - loss: 0.4594 - acc: 0.7768
Epoch 39/40
2600000/2600000 [==============================] - 276s 106us/step - loss: 0.4588 - acc: 0.7771
Epoch 40/40
2600000/2600000 [==============================] - 275s 106us/step - loss: 0.4581 - acc: 0.7776
100000/100000 [==============================] - 3s 30us/step
0.8576858114535069
```

Random uniform

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='random_uniform')) # X_train.shape[1] == 15 here
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='random_uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='random_uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='random_uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
model.fit(x, y, epochs=40, batch_size=100)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 36/40
2600000/2600000 [==============================] - 311s 120us/step - loss: 0.4575 - acc: 0.7780
Epoch 37/40
2600000/2600000 [==============================] - 320s 123us/step - loss: 0.4568 - acc: 0.7783
Epoch 38/40
2600000/2600000 [==============================] - 331s 127us/step - loss: 0.4565 - acc: 0.7786
Epoch 39/40
2600000/2600000 [==============================] - 322s 124us/step - loss: 0.4557 - acc: 0.7792
Epoch 40/40
2600000/2600000 [==============================] - 319s 123us/step - loss: 0.4552 - acc: 0.7794
100000/100000 [==============================] - 4s 35us/step

0.8576136837807264
```

Took our best results from and tried 3 different optimizers. (LMGTFY)

Three different optimizers used are: Rmsprop

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15 here
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

rmsprop = RMSprop(lr=0.01, decay=1e-6)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=rmsprop)


```

```
model.fit(x, y, epochs=5, batch_size=10000)
score = model.evaluate(x_test, y_test, batch_size=10000)
roc_auc_score(y_test,model.predict(x_test))
```

```
WARNING:tensorflow:Variable *= will be deprecated. Use variable.assign_mul if you want assignment to the variable value or 'x =
x * y' if you want a new python Tensor object.
Epoch 1/5
2600000/2600000 [==============================] - 213s 82us/step - loss: 7.4730 - acc: 0.5294
Epoch 2/5
2600000/2600000 [==============================] - 324s 125us/step - loss: 7.4991 - acc: 0.5296
Epoch 3/5
2600000/2600000 [==============================] - 209s 81us/step - loss: 7.4991 - acc: 0.5296
Epoch 4/5
2600000/2600000 [==============================] - 205s 79us/step - loss: 7.4991 - acc: 0.5296
Epoch 5/5
2600000/2600000 [==============================] - 195s 75us/step - loss: 7.4991 - acc: 0.5296
100000/100000 [==============================] - 2s 19us/step

0.5
```

Adagrad

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

adagrad = Adagrad(lr=0.01, decay=1e-6)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=adagrad)
```

```
model.fit(x, y, epochs=40, batch_size=10000)
score = model.evaluate(x_test, y_test, batch_size=10000)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 545s 210us/step - loss: 0.4843 - acc: 0.7615
Epoch 36/40
2600000/2600000 [==============================] - 542s 209us/step - loss: 0.4837 - acc: 0.7620
Epoch 37/40
2600000/2600000 [==============================] - 496s 191us/step - loss: 0.4830 - acc: 0.7625
Epoch 38/40
2600000/2600000 [==============================] - 185s 71us/step - loss: 0.4827 - acc: 0.7625
Epoch 39/40
2600000/2600000 [==============================] - 187s 72us/step - loss: 0.4821 - acc: 0.7627
Epoch 40/40
2600000/2600000 [==============================] - 198s 76us/step - loss: 0.4814 - acc: 0.7634
100000/100000 [==============================] - 2s 17us/step

0.8475752688448333
```

Adam

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

adam = Adam(lr=0.01, decay=1e-6)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=adam)
```

```
model.fit(x, y, epochs=40, batch_size=10000)
score = model.evaluate(x_test, y_test, batch_size=10000)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 35/40
2600000/2600000 [==============================] - 237s 91us/step - loss: 0.5100 - acc: 0.7435
Epoch 36/40
2600000/2600000 [==============================] - 194s 75us/step - loss: 0.5093 - acc: 0.7437
Epoch 37/40
2600000/2600000 [==============================] - 192s 74us/step - loss: 0.5094 - acc: 0.7439
Epoch 38/40
2600000/2600000 [==============================] - 194s 75us/step - loss: 0.5089 - acc: 0.7441
Epoch 39/40
2600000/2600000 [==============================] - 186s 71us/step - loss: 0.5089 - acc: 0.7442
Epoch 40/40
2600000/2600000 [==============================] - 182s 70us/step - loss: 0.5090 - acc: 0.7441
100000/100000 [==============================] - 2s 17us/step

0.8371529877342359
```

The best model was a 3-layer hybrid CNN model with some dropouts between layers and a smaller output layer. The relu activations function was a big part of the algorithm as well. This gave our best score of **87.27**.

```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD
from sklearn.metrics import roc_auc_score
from keras.callbacks import EarlyStopping
es=EarlyStopping(monitor='loss', min_delta=0,patience=0,verbose=0,mode='auto')
```

```
model = Sequential()
model.add(Dense(512, input_dim=x.shape[1], kernel_initializer='uniform')) # X_train.shape[1] == 15 here
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(512, kernel_initializer='uniform'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(256, kernel_initializer='uniform'))
model.add(Activation('relu'))
#model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='uniform'))
model.add(Activation('sigmoid'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=sgd)


```

```
model.fit(x, y, epochs=60, batch_size=100,callbacks=[es],shuffle=True)
score = model.evaluate(x_test, y_test, batch_size=100)
roc_auc_score(y_test,model.predict(x_test))
```

```
Epoch 45/60
8500000/8500000 [==============================] - 1850s 218us/step - loss: 0.4469 - acc: 0.7848
Epoch 46/60
8500000/8500000 [==============================] - 1248s 147us/step - loss: 0.4465 - acc: 0.7851
Epoch 47/60
8500000/8500000 [==============================] - 1287s 151us/step - loss: 0.4464 - acc: 0.7850
Epoch 48/60
8500000/8500000 [==============================] - 1190s 140us/step - loss: 0.4460 - acc: 0.7854
Epoch 49/60
8500000/8500000 [==============================] - 1006s 118us/step - loss: 0.4456 - acc: 0.7855
Epoch 50/60
8500000/8500000 [==============================] - 915s 108us/step - loss: 0.4456 - acc: 0.7855
100000/100000 [==============================] - 3s 30us/step

0.8727399045530653
```

5 Discussion

We sought to answer the questions:

Q1: What was the effect of adding more layers/neurons?

In general, adding layers increased the depth of the network and adding neurons introduced more parameters to the model making it more complex and capable of modeling the variation in the data. Both helped to model the data better but adding too many slowed things down and ended up not helping the score. Essentially adding too many layers and neurons tended to ‘overfit’ the data and found patterns that was not reflected in the test data.

We started with 1 layer and stopped at 4 layers. Adding more layers and neurons helped us in improving the score of the model. To avoid overfitting, we used dropout rate of 0.1 in between the hidden layers. The model with 3 layers gave us the best score and after that adding more layers started dropping the score. I believe with further adjustments this model could be further enhanced.

Q2: Which parameters gave you the best result and why (in your opinion) did they work.

Parameters such as number of layers, 3, activation, 'Relu' in 3 layers and 'Sigmoid' in output layer with a dropout rate of 0.1 gave us the best result.

As stated above 3 layers worked the best and with dropout minimized overfitting.

The purpose of activation functions is to introduce non-linearities into the network. The "Relu" function worked best for us. It was better in this case because of better gradient propagation, sparse activation, efficient computation, and scale invariance. It is also one sided as compared to "tanh" which is antisymmetric.

The dropout rate helped in avoiding the problem of overfitting. In this case it may be a dropout rate of 0.1 worked best because a high value of dropout value results in under-learning by the model. This is pretty typical for large data sets like this.

Also, some of the hyper-parameters such as a learning rate of 0.01, *kernel_initializer* = 'uniform', *optimizer* = *sgd*, and *batch_size* =100 helped to improve the results.

The learning rate is one of the most important hyper-parameter. Small learning rates converge slowly and get stuck in false local minima, while large learning rates tend to overshoot, become unstable, and diverge. Stable learning rates should converge smoothly and move through local minima. Setting the learning rate to 0.001 took ages to converge, and with a learning rate of 0.0 the scores were not good. Therefore, we chose a learning rate of 0.01 which performed best for our model with this data.

The kernel initializers set the initial random weights of the Keras layers. Using the 'uniform' *kernel_initializer* makes the initial weights uniformly initialized between ±1. This was best for our analysis.

Using the SGD optimizer worked best in our case. We provided parameters such as *momentum* and *nesterov* where *momentum* is a parameter that accelerates SGD in the relevant direction and dampens oscillations and *nesterov* is the parameter which when given the value 'True' applies Nesterov momentum.

The batch size is the number of training examples in one forward/backward pass. Mini-batches lead to fast training, the higher the batch size, the more memory space you'll need. With batch size of 10000 and 100000, the model accuracy and ROC AUC score decreased and took longer to converge. Again, the *batch_size* of 100 worked best for our model.

Q3: How did you decide that your model was 'done?'

We decided that our model was "done" because the loss stopped decreasing for a number of epochs. This is a standard stopping procedure. We used this early stopping to tune the number of epochs, which is the simplest use of the test set, where the model simply stopped training once the loss hasn't decreased for a fixed number of epochs (a parameter known as patience). As this is a relatively small benchmark which saturates quickly, we had a patience of zero epochs, and increased the upper bound on epochs to 60 (which will likely never be reached). And as can be seen in the last screenshot, the model stopped at 50 epochs as the loss sopped decreasing at the 50^{th} epoch. We can't say for sure that our model is "done" as we've trained on only 8.5 million

entries, however this makes up a large portion of the total. If the training set was increased to encompass the entire data set except for the test portion a slightly better score is likely.

6 Conclusion

In this study we created a hybrid convolutional neural network to model the 11.1 million rows and 28 columns of data from a simulation of the output of high-energy particles. We found that a complex hybrid CNN model with dropouts and parameters as detailed worked the best. Our best model produced a ROC AUC score of 0.8727.

References

[1] Baldi, P., P. Sadowski, and D. Whiteson. "Searching for Exotic Particles in High-energy Physics with Deep Learning." Nature Communications 5 (July 2, 2014). https://arxiv.org/pdf/1402.4735.pdf, https://archive.ics.uci.edu/ml/datasets/HIGGS

[2]. Keras Model Documentation: https://keras.io/

[3]. Tensorflow Documentation: https://www.tensorflow.org/