Case Study 6, Unit 12 - Neural Networks
Daniel Davieau, Lei Jiang, Kristen Rollins
31 March 2020

Introduction

There have been many advancements in deep learning over the last several years that have exciting applications in many domains. One study in 2014 applied deep learning tools to the high-energy physics domain, using a deep neural network to detect new exotic Higgs boson particles, i.e. classify signal versus background particles. The focus of this case study will be on replicating and attempting to improve the neural architecture used in the research paper. We are not physicists, and thus we will primarily be exploring neural architectures purely from the perspective of implementing a classification task, without substantial knowledge of the topic at hand. We recognize that real advances for this application would benefit much more from research by real physicists who understand the mechanics surrounding the task. However, we hope that we can still provide some insights for best practices for neural networks that have changed in the 6 years since the research was published, to build upon and possibly improve the results presented.

Replication

For this case study, we went down two paths for replicating and tuning the deep learning architecture presented in the paper. In one case, we used a personal machine with only a CPU, in which we used only 2% of the 11 million point dataset (about 22,000 points) to accelerate training time. On the other hand, we also had a powerful machine with a GPU available, with which we attempted to run the model with the original training/test data (2.7 million points), as well as incorporating even more data (up to 6 million training points). We hoped to use this additional data to achieve comparable results to the original paper and potentially improve the performance. Unfortunately, we discovered some errors in our GPU setting that we ran out of time to debug, as we ran into issues with the SGD function in that TensorFlow environment. Thus our analysis was primarily performed on the CPU with only 2% of the dataset, but we achieved satisfactory results even with this small percentage.

We first attempted to replicate the neural network created by the researchers, based on the documentation found in their paper. Our replication of the code can be found in our file submitted called lJiang_kRollins_dDavieauCaseStudy6_Update3.ipynb. More details can be found there, but we will highlight some key decisions made here as well. Below is the main code (pulled out of a function and an unrolled loop) for how we replicated the architecture using TensorFlow and Keras. For the replicated model, `units` are defined as 300, `activation_func` is “tanh”, `weight_decay` is 0.00001, and `learning_rate` is 0.05. For the sake of running time, we only allowed the model to run for 100 epochs with a batch size of 1000, while in the original paper the epochs were upwards of 200 and batch size was 100.

```
model= tf.keras.Sequential()
```

```
model.add(layers.Dense(units,activation=activation_func,kernel_regularizer=tf.
      keras.regularizers.l2(weight_decay)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(units,activation=activation_func,kernel_regularizer=tf.
      keras.regularizers.l2(weight_decay)))
model.add(layers.Dense(units,activation=activation_func,kernel_regularizer=tf.
      keras.regularizers.l2(weight_decay)))
model.add(layers.Dense(units,activation=activation_func,kernel_regularizer=tf.
      keras.regularizers.l2(weight_decay)))
model.add(layers.Dense(1 ,activation='sigmoid'))

# compile the model
model.compile(tf.keras.optimizers.SGD(lr=learning_rate, momentum=0.9),
                  loss = 'binary_crossentropy',
                  metrics=metrics)
# fit the model
results = model.fit(X_train, y_train, validation_data=(X_test,y_test),
      epochs=epochs, batch_size=batch_size, callbacks=[tb])
```

The paper reports that they selected a “five-layer neural network with 300 hidden units in each layer.” This initially led us to believe that there were five hidden layers each with 300 units, and after adding the input and output layers the network should have seven layers in total. However, we delved into the code provided in their Github repository, where we saw a maximum of only four hidden layers being used (example). We chose to trust the code in this case, and believe that their wording includes either the input or the output layer in the “five-layer” network. Thus our replicated model has 6 layers in total: one input layer of size 28 (which is implicitly included based on the input data having 28 columns), four dense hidden layers (MLPs, or Multilayer Perceptrons) each with 300 nodes, and one output layer with 1 node (for binary classification).

In addition, the paper says that they used “weight decay 10^-5”, which is equivalent to L2 regularization in TensorFlow. Thus we included `kernel_regularizer=regularizers.l2(0.00001))` in our code. Also note that the final output layer has a sigmoid activation function, which is necessary for a binary classification task. At first we overlooked this, and did not specify an activation function since the paper did not make this distinction. This is one of the issues that remained in the GPU code, but we were able to correct it for the CPU code.

We did notice that in the researchers’ code (Theano and Pylearn2 framework) they were able to specify a batch size of 100 within their SGD function to get mini-batches. In TensorFlow, it seems that using the SGD optimizer and then including `batch_size=100` later in the model fit will take care of this. Finally, a dropout layer of 0.5 was included after the first hidden layer, which we believe is equivalent to their statement that they “stochastically drop neurons in the top hidden layer with 50% probability during training.” This resulting architecture is our best attempt at reproducing the researchers’ model, given the information available in the paper and the time that we had.

Comparison of Results

The Higgs research paper chose to produce ROC (receiver operator characteristic) curves and report AUC (area under the curve) values as their performance metric. A higher AUC value indicates higher classification accuracy across a range of thresholds, and the paper expressed that small increases in AUC can “represent significant enhancement in discovery significance”, which is another metric relevant to high-energy physics. Thus we also created ROC curves and calculated the AUC for our networks, so we could compare our results directly to the paper’s and quantify whether we duplicated their results. With our replication of the architecture, when we used just 2% of the data as well as 100 epochs and batch sizes of 1000, we produced the ROC curve found in Figure 1, which had an AUC of 0.823. This can be compared to the researchers’ result of 0.885. With more time, we would have liked to increase the epochs (also see Figure 1) and decrease the batch size, to see if we could reach results even closer to the original results even with only 2% of the data. However we were pleasantly surprised how close we could get even with several time-saving shortcuts, which makes us more confident that we have replicated the original model closely.

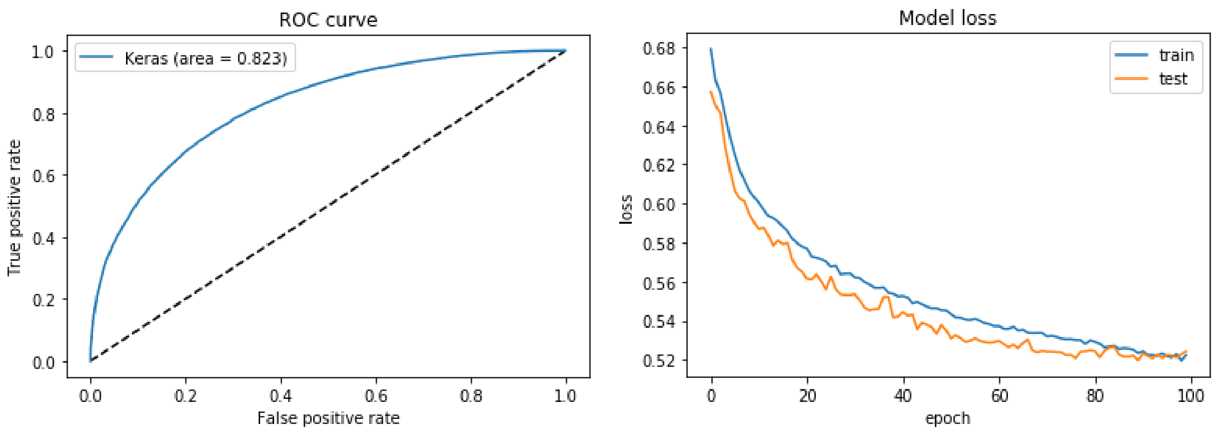

Figure 1. Replicated Model - ROC Curve (left) and Training and Test Loss (right)

Recommendations

In addition to replicating the original study’s neural architecture, we tried out additional architectures and parameters to try to improve the results. Overall, our recommendation to the paper’s approach would be to experiment more with optimization, rather than take the architecture or hyperparameters for granted. The primary things we tried experimenting with were the neural architectures themselves, different activation functions, and dropout rates. Due to time and computation constraints, we allowed each network to train for only 5 epochs, and kept batch size at 1000. To continue the research in earnest, we would utilize GPUs so that we could increase the data size and number of epochs greatly, as well as reduce the batch size back to 100.

Architectures

We first experimented with different neural architectures. The original researchers cited some architecture tuning in the supplementary tables at the end of the paper, but we believe this should be tested more thoroughly. We tried various architectures, ranging from 4 hidden MLP layers up to 6 (note that in our code, there is one more hidden layer than the num_layers parameter specifies; we ran out of time to fix this). With increased computing power, even more layers could be attempted, as well as attempting layers with varying numbers of nodes.

Activation Functions

We also experimented with different activation functions within the hidden layers. Two widely used nonlinear activation functions are the sigmoid and hyperbolic tangent (tanh) activation functions, seen in Figure 2. However, the limitation is that they are not ideal in networks with many layers due to the vanishing gradient problem. The rectified linear activation function, or ReL for short, overcomes the vanishing gradient problem, allowing models to learn faster and perform better. The rectified linear activation has become the default activation when developing multilayer perceptron and convolutional neural networks. A node or unit that implements this activation function is referred to as a rectified linear activation unit, or ReLU for short. The ReLU function as well as other variants such as ELU (Exponential Linear Unit) or Leaky ReLU that have different behavior for negative values, are shown in Figure 3. These have generally become more favorable than the tanh function that the original paper used, so we believe any of these would be good alternative choices for the hidden layers in the network.

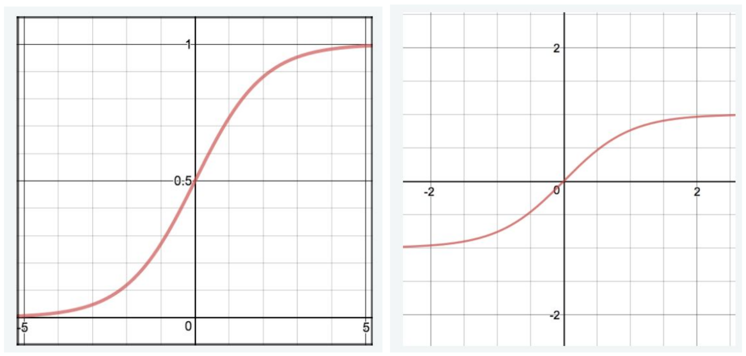

Figure 2. Sigmoid (left) and Hyperbolic Tangent (right) Activation Functions

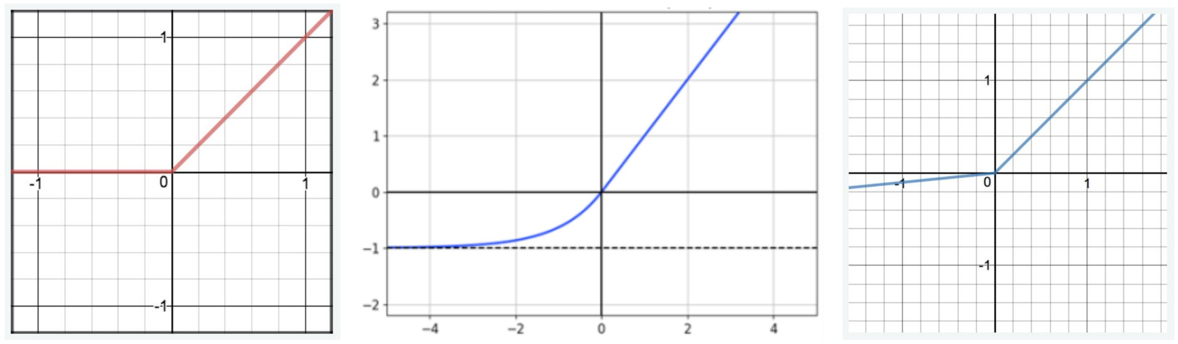

Figure 3. ReLU (left), ELU (middle), and Leaky ReLU (right) Activation Functions

Dropout

We also experimented with different dropout rates (primarily in our GPU attempt). Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or “dropped out.” In this way we can combat overfitting in the network. Research indicates that dropout is more effective on smaller datasets, but in this case we have a huge dataset available with 11 million data points. Thus we primarily experimented with decreasing the dropout rate to 0.1 rather than the 0.5 used in the original research. We would like to do more investigation on dropout rates, as well as explore which layers should include dropout.

Our Results

Overall, we believed that using more recently popular activation functions such as ReLU or ELU instead of tanh would improve the performance of the network, both in accuracy and AUC. However, in our small-scale grid search we actually found that the results were varied when switching between ReLU and tanh. For instance, with 5 hidden layers, tanh was better with an AUC of 0.742 while ReLU had an AUC of 0.737. However for 6 hidden layers, ReLU was better as it had an AUC of 0.740 while tanh had an AUC of 0.738. Thus we found that ReLU does not always outperform the hyperbolic tangent for this dataset. We also found in our small grid search that with all else held constant, 5 hidden MLP layers performed better than the network with just 4 hidden layers. The experimentation we did with dropout was in the GPU attempt, which cannot be compared directly to our CPU results, so more experimentation is needed to determine the best dropout rates. Furthermore, training the neural networks on even more training data is possible with today’s computation capabilities, which could improve the accuracy of the classifier even more.

Standard Practices

The major changes in deep learning practices since this paper was written in 2014 seem to be in regards to hyperparameter tuning as well as computation power. When describing their methods, the paper states, “In training the neural networks, the following hyperparameters were predetermined without optimization.” Today, the standard practice is to not take hyperparameters for granted, but rather take much more time to optimize parameters. In particular, performing a grid search is often accepted as a way to optimize parameters. At the time when the paper was written, computational costs were a significant limiting factor to optimization. The authors even wrote, “Due to computational costs, this optimization was not thorough.” While computing power should be considered, it is not nearly as much of a limiting factor today as it was 6 years ago. Therefore, if this study were to be repeated today, a more thorough grid search would be done to optimize the network much more robustly, since more resources are available at a lower cost.

In addition, distributed deep learning with data parallelism has become increasingly popular and available since the paper was written in 2014. Single-machine parallelism with GPUs and multi-machine parallelism with HPC clusters, or even cloud computing are ubiquitous. With today’s computing power, we can perform this thorough hyper-parameter search. The last step of distributed deep learning are techniques for consolidating models such as Ensemble Learning and Model Averaging, which could improve performance of the networks even more.

Note that performing this thorough grid search today would include more recent developments for neural networks, such as ReLU activation functions and newer optimization methods such as Adam. Finally, the original study only used 2.6 million data points for training and 100,000 points for testing, out of the 11 million data points available. Current trends in machine learning seem to throw as much data as possible at networks to improve performance, so continuing the research today would likely incorporate more training and validation data.

Conclusions

We were glad to find that we could replicate the neural architecture of the Higgs particle research paper fairly closely, using TensorFlow and Keras. While we had some computing and time limitations, and could only run our networks with 2% of the full dataset, we were able to achieve an AUC of 0.823 compared to the paper’s AUC of 0.885. We were very satisfied with this, as our result included other shortcuts such as decreased epochs and increased batch sizes. Since we were able to get fairly close to the original results even with these shortcuts, we believe we accurately replicated the paper, and with additional resources we could be well on our way to improving the original results as well. This is because there are many more resources available today compared to 2014 when the research was concluded. We could utilize distributed deep learning methods to expand the hyperparameter search greatly, as well as incorporate more recent developments in deep learning.

References

- Higgs research paper: https://arxiv.org/pdf/1402.4735.pdf
- Github repository for the paper: https://github.com/uci-igb/higgs-susy
- Tensorflow and Keras documentation

- Adam optimizer paper: https://arxiv.org/abs/1412.6980
- Resource on activation functions: https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html
- Another resource on activation functions: https://towardsdatascience.com/exploring-activation-functions-for-neural-networks-73498da59b02
- Resource on dropout: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/
- Resource on grid search for deep learning models: https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/
- Resource on distributed deep learning: https://medium.com/@Petuum/intro-to-distributed-deep-learning-systems-a2e45c6b8e7

Appendix

Codebase

Our codebase has been included in a file called lJiang_kRollins_dDavieauCase6_Code.zip. The files in this zip folder include:

- lJiang_kRollins_dDavieauCaseStudy6Update3.ipynb - models run on CPU with 2% of dataset
- GPU_lJiang_kRollins_dDavieauCaseStudy6.ipynb - attempt to run models on GPU
- Images to render in GPU notebook:
  - GPU_Activity.png
  - CPU_Activity.png
  - OriginalHardware.png
  - OurHardware.png

Assignment

- Given the following paper: https://arxiv.org/pdf/1402.4735.pdf
- Build a replica Neural Network with the paper's architecture using TensorFlow.
- If possible begin to train on the data located here: https://archive.ics.uci.edu/ml/datasets/HIGGS
  - How close can you get to the original results?
- To facilitate quicker training, you may increase the batch size temporarily (this has a small impact on final result, but can speed up your calculations significantly). You do not need to train a final result using the paper's parameters, only the code for your model is required in your final submission.
- Include in your report:
  1. Your replica neural network based on the paper's architecture

2. Any recommendations or modifications you'd make to the approach taken by the researchers (i.e., state any proposed changes along with the expected improvements or impact)
3. What are standard practices now versus when this paper was written?
4. How would you quantify if your result duplicated the paper's?