Recreating the HIGGS Experiment Deep Neural Network

Gupta, Moro, Kannan

1. Business Understanding

Introduction: The objective of this case study is to replicate a deep neural network model for the HIGGS dataset [1] which is a fine example of classification problem in the field of high-energy physics. Here, the underlying machine learning problem is to distinguish between a signal process which produces Higgs bosons (God particle) and a background process (noise). The following sections give a high-level overview about Higgs Boson particles.

Brout-Englert-Higgs mechanism: The Brout-Englert-Higgs mechanism (BEH mechanism) describes how fundamental particles obtain mass. As per this theory, which was developed independently by Robert Brout and François Englert in Belgium and Peter Higgs in the United Kingdom in 1964, fundamental particles acquire mass by interacting with a "field" that permeates the entire Universe. The more strongly the particles interact with the field, the more massive they are. This field is named as "***Higgs Field*"**. The ***Higgs boson*** is the quantum particle associated with the Higgs field.

Higgs Boson: The Higgs Boson which is a crucial sub-atomic particle also known as the "God Particle" could unlock the greatest mysteries of universe and help us understand its basic nature better. Physicists claim that this long-theorized subatomic particle would explain why matter has mass. The existence of this particle has been considered a missing cornerstone of physics.

Experiment at CERN: The ATLAS experiment at CERN is designed as a general-purpose particle physics experiment and contains the Large Hadron Collider (LHC) which is the world's largest and most powerful particle accelerator. Inside the LHC, two particle beams travel at close to the speed of light before they are made to collide. The beams travel in opposite directions and there are detectors installed to analyze the myriad of particles produced by this collision.

Need for Machine Learning Algorithm: A Higgs boson particle emanating from the above experiment cannot be detected directly as it decays too rapidly. Instead, one must infer its brief existence by examining the energy properties of the particles produced from its decay. But the chances of producing a Higgs boson decay is only one in a trillion, so the scientists must carefully comb through all the particle data, which is often obscured by high background signals from other particles generated by the collision. LHC produces over 30 petabytes of such collision data a year which is why scientists prefer using machine learning algorithms to identify potential particles from the datasets. Machine learning provides an efficient way to mine the Higgs boson data for precious signals with limited waste.

Baseline model referred for replication: In this case study, we referred to the paper published on the topic *"Searching for exotic particles in high-energy physics with deep learning"* [2] authored by *D. Whiteson, P. Baldi & P. Sadowski* from UC - Irvine. This paper demonstrates how deep learning methods can overcome the shortcomings of this high-energy physics experiment, providing significant boosts in the Higgs boson discovery results. This paper leverages the advancement in the field of deep learning which make it possible to learn more complex functions and better discriminate between signal (Higgs Boson) and background classes (other particles). Using the benchmark datasets, deep-learning method used in this study was able to improve the classification metric by 8% when compared to other existing approaches.

Proposed Improvement in Model Replication: The above-mentioned research study which was published in July 2014 used pylearn2 and Theano python libraries. While Theano helps with evaluating complex mathematical operations, pylearn2 is a machine learning library that has been designed to facilitate research projects. From a technology perspective, switching to advance framework like TensorFlow and Keras API makes more sense as there is not much support available for pylearn2 anymore [3]. Moreover, TensorFlow, which is built and maintained by Google, allows researchers and developers to work together on developing an AI model and uses full computational power of modern hardware including the latest GPUs. Keras on the other hand provides another layer of API over TensorFlow, encapsulating the model implementation details and enables quicker implementations.

Assumptions: In order to replicate the deep neural network architecture using TensorFlow and Keras, we assumed that the features in pylearn2 and Theano (especially the key features used for building the final neural network model in this paper) are also available in the new framework under consideration (TensorFlow, Keras API). In certain situations where some features or functions are not available, we considered using other alternate approaches available in the latest framework.

2. Data Quality

The data used for the model developed by the researchers has been produced using Monte Carlo simulations of collisions between particles. The data contains a target column representing if the record is about a signal (value of 1) or a background process (value of 0). Then there are 28 total features; 21 of these represent the kinematic properties measured by the particle detectors in the accelerator, and seven are functions of the previous 21 features that help discriminate between the two classes.

The dataset has total of 11 million records [1]. The original dataset was too large to be analyzed using the resources available, so for the purpose of this case study, we only used 20,000 random records (from the 11 million the original data set). Since these were picked randomly, they would still be representative of the entire dataset. We utilized the Google Colab online service to perform the analysis, which allows the use of a cloud based virtual machine with the most recent version of TensorFlow library and advanced GPU capabilities. [4].

An initial analysis of the data shows that (1) there are no missing values in any of the available features, and (2) all the features contains numeric value, so there is no need to perform one hot encoding. The target variable contains two class levels: 1 for a signal (Higgs boson) and 0 for a background process (other particles). There are 10,457 observations for class level 1 and 9,543 observations for class level 0. This indicates we have a fair balanced dataset with 52.3% positive class and 47.7% negative class.

The 28 features, representing the experiment parameters, have numeric values ranging from -2.96 to 16.13. Each feature has a different range and distribution. Fig. 1 shows the distribution and the key statistical values for each feature. We notice that five of the features have a normal distribution, six have a uniform distribution, and 13 have a right skewed distribution. However, all features have been standardized to some extent with values ranging from approximately -3 to 3 in most cases (representing a standard deviation of 1). This ensures that the neural network can train efficiently as we discuss later in 4.5. If they were not on the same scale, then these features would have had to be standardized before training to help the optimizer but that was not deemed necessary in this case.

Four features (number 8, 12, 16, and 20) have only three values presents in the data set and their representation is very similar. Even if the model can still use these features, there is a risk that these are cofounding or highly correlated variables. We would recommend that this be validated with subject matter experts to check if these features are measuring the same parameter or highly correlated parameters. However, for this purpose of this case study we will assume that there are no cofounding variables.

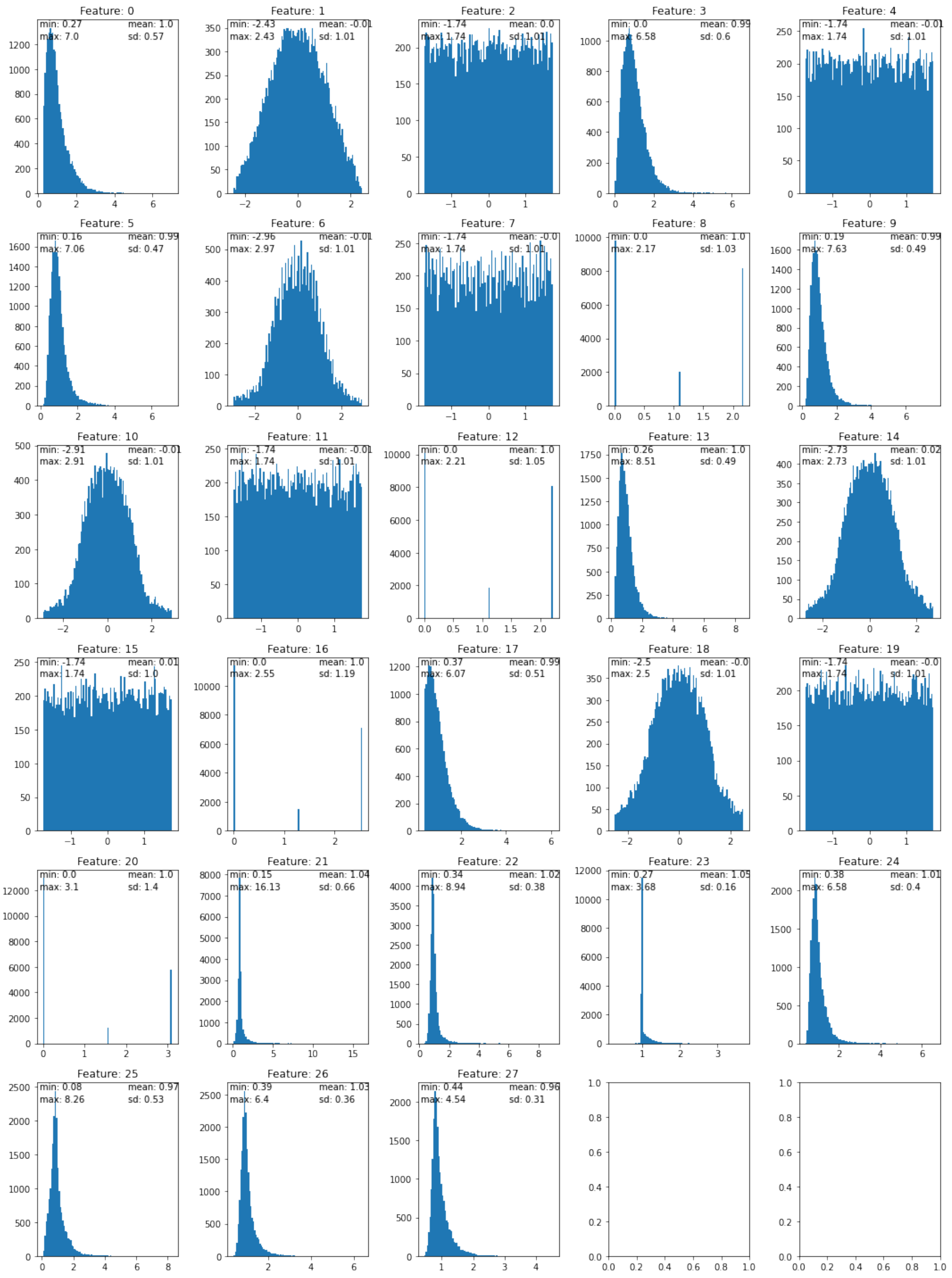


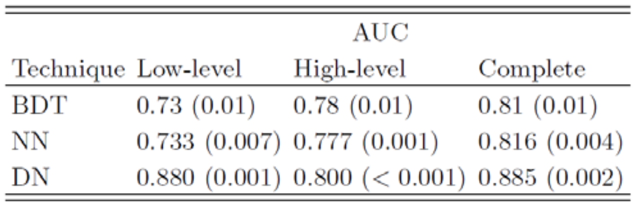
Fig. 1. Distribution of the 28 Available Features

3. Architecture

The objective of this section of the case study is to replicate the paper's deep neural network (DN) based using the latest framework TensorFlow 2.3. We start off by describing how the DN was originally developed in Pylearn2 and conclude this section with how we replicated the same in TensorFlow.

3.1 Original Implementation in Pylearn2

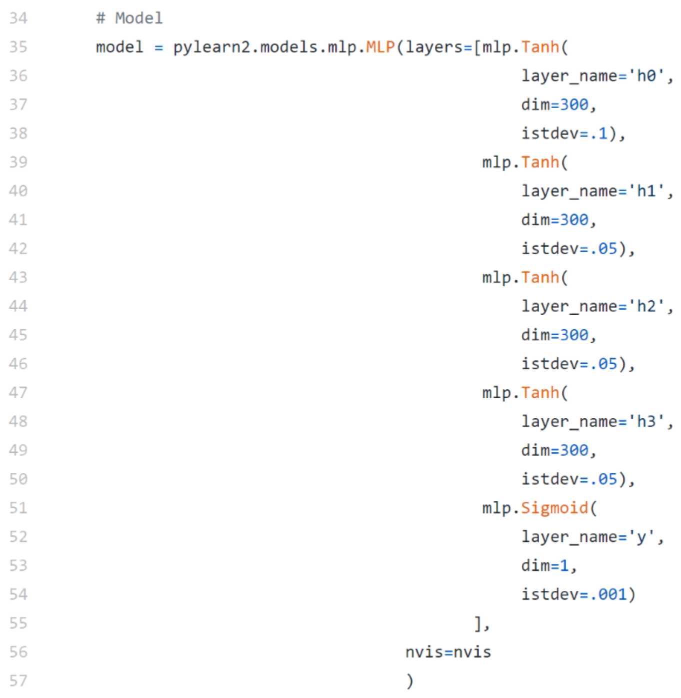
Evaluation Metric: The training of the original model was based on 2.6 million records randomly selected from the entire dataset and the test dataset was based on 100,000 records. Area Under the ROC Curve (AUC) was used to tune and measure the performance of the model. The higher the AUC, the better the model. Fig. 2 shows the results published in the original paper. We can notice that amongst all the methods tried, the Deep Neural Network (DN) model obtained a maximum AUC of 0.885 with the complete dataset.



	AUC		
Technique	Low-level	High-level	Complete
BDT	0.73 (0.01)	0.78 (0.01)	0.81 (0.01)
NN	0.733 (0.007)	0.777 (0.001)	0.816 (0.004)
DN	0.880 (0.001)	0.800 ($<$ 0.001)	0.885 (0.002)

Fig. 2. Original paper model performance

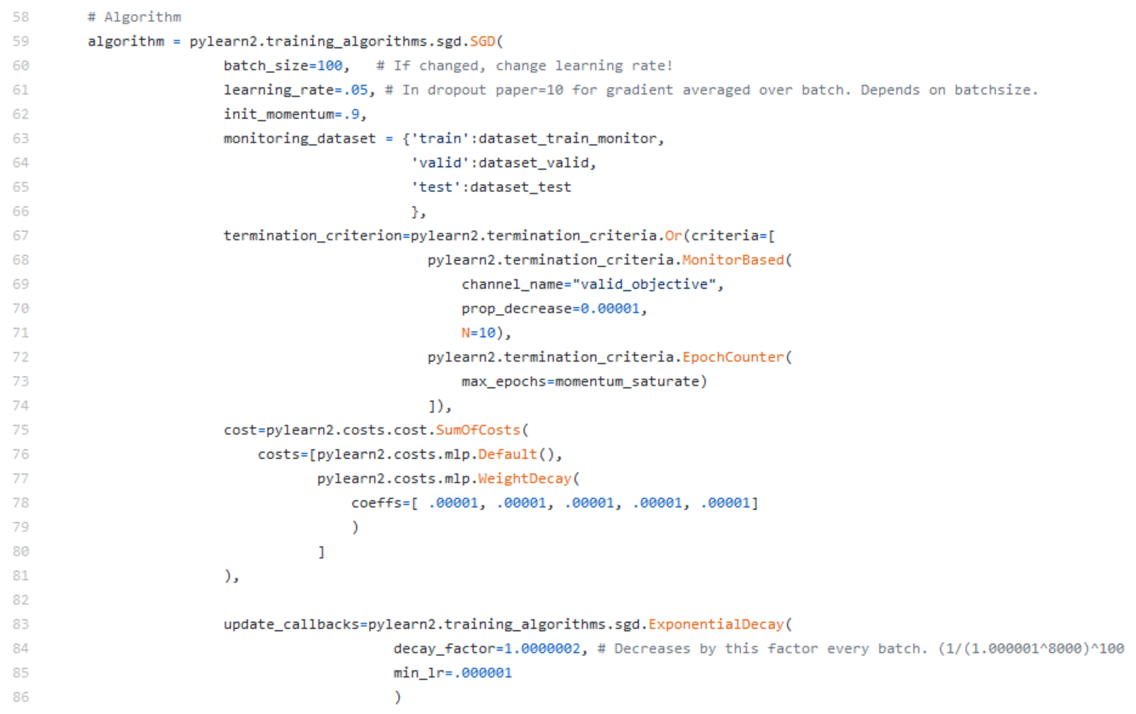
Architecture Implementation: The DN model used in the original paper had five layers. The five-layered structure was composed of four hidden dense layers with *tanh* activation function and 300 neurons each and an output dense layer with a sigmoid function to generate the likelihood of the output being 1 or 0. Fig. 3 shows the original model implementation in Pylearn2 [5]. We can clearly notice the four hidden layers named 'h0' to 'h3' with the '*tanh*' activation function, and the output layer using the sigmoid activation function.



```
# Model
model = pylearn2.models.mlp.MLP(layers=[mlp.Tanh(
                                             layer_name='h0',
                                             dim=300,
                                             istdev=.1),
                                        mlp.Tanh(
                                             layer_name='h1',
                                             dim=300,
                                             istdev=.05),
                                        mlp.Tanh(
                                             layer_name='h2',
                                             dim=300,
                                             istdev=.05),
                                        mlp.Tanh(
                                             layer_name='h3',
                                             dim=300,
                                             istdev=.05),
                                        mlp.Sigmoid(
                                             layer_name='y',
                                             dim=1,
                                             istdev=.001)
                                       ],
                                 nvis=nvis
                                 )
```

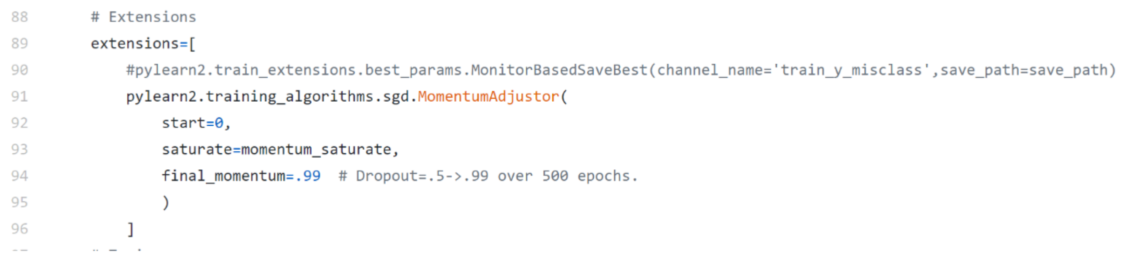
Fig. 3. Original Model in Pylearn2

Fig. 4 shows the algorithm parameters set by the authors which indicate the additional behavior the model training should follow such as early stopping (termination criteria), learning rate decay and penalty weights (Weight Decay). Fig. 5 show the extensions applied to the model to control for momentum parameter.



```
58  # Algorithm
59  algorithm = pylearn2.training_algorithms.sgd.SGD(
60                  batch_size=100,   # If changed, change learning rate!
61                  learning_rate=.05, # In dropout paper=10 for gradient averaged over batch. Depends on batchsize.
62                  init_momentum=.9,
63                  monitoring_dataset = {'train':dataset_train_monitor,
64                                        'valid':dataset_valid,
65                                        'test':dataset_test
66                                        },
67                  termination_criterion=pylearn2.termination_criteria.Or(criteria=[
68                                          pylearn2.termination_criteria.MonitorBased(
69                                              channel_name="valid_objective",
70                                              prop_decrease=0.00001,
71                                              N=10),
72                                          pylearn2.termination_criteria.EpochCounter(
73                                              max_epochs=momentum_saturate)
74                                          ]),
75                  cost=pylearn2.costs.cost.SumOfCosts(
76                      costs=[pylearn2.costs.mlp.Default(),
77                             pylearn2.costs.mlp.WeightDecay(
78                                 coeffs=[ .00001, .00001, .00001, .00001, .00001]
79                                 )
80                             ]
81                  ),
82
83                  update_callbacks=pylearn2.training_algorithms.sgd.ExponentialDecay(
84                                      decay_factor=1.0000002, # Decreases by this factor every batch. (1/(1.000001^8000)^100
85                                      min_lr=.000001
86                                      )
```

Fig. 4 Original Algorithm of the Pylearn2 model



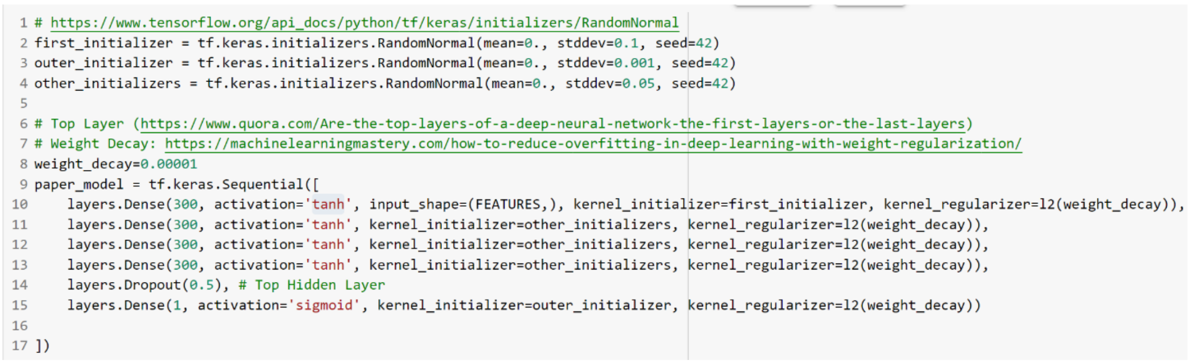
```
88  # Extensions
89  extensions=[
90      #pylearn2.train_extensions.best_params.MonitorBasedSaveBest(channel_name='train_y_misclass',save_path=save_path)
91      pylearn2.training_algorithms.sgd.MomentumAdjustor(
92          start=0,
93          saturate=momentum_saturate,
94          final_momentum=.99  # Dropout=.5->.99 over 500 epochs.
95          )
96      ]
```

Fig. 5. Original Extensions of the Pylearn2 model

3.2 New Implementation in TensorFlow

Layers: The first task was to upgrade the original model to the TensorFlow library. TensorFlow has a significantly different coding structure as indicated in Fig. 6. In TensorFlow, the layers of the neural network are added in sequential format as indicated. In the new code we included four hidden layers with the $tanh$ activation function and 300 neurons each (rows 10-13 in Fig. 6) to match the original structure visible starting from lines 36-50 of Fig. 3.

Weight Decay: As suggested by the paper, to control for overfitting of the neural network, we utilized a weight regulation technique. During each update cycle of the network, the weights are penalized based on the $“L2”$ norm. L2 norm updates the loss function based on the sum of squares of the weight values ($loss = weight\_decay * reduce\_sum(x^2)$) [6]. This ensures that the weights do not get too big and hence helps prevent overfitting. In the TensorFlow model, this is performed using the parameter `kernel_regularizer=l2(weight_decay)` in lines 10-15 of Fig. 6. The `weight_decay` value was set to 10^{-5} on row 8. The original model performed this operation by using the algorithm parameters ‘`cost`’ indicated in lines 75-81 in Fig. 4, and manually setting the coefficients of 10^{-5} per each layer (line 78).



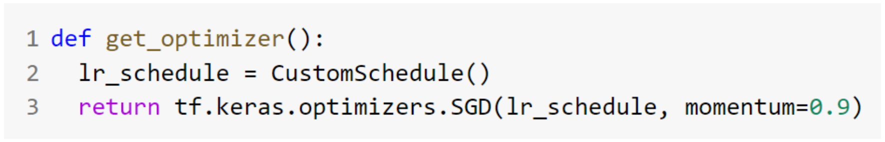
```
 1 # https://www.tensorflow.org/api_docs/python/tf/keras/initializers/RandomNormal
 2 first_initializer = tf.keras.initializers.RandomNormal(mean=0., stddev=0.1, seed=42)
 3 outer_initializer = tf.keras.initializers.RandomNormal(mean=0., stddev=0.001, seed=42)
 4 other_initializers = tf.keras.initializers.RandomNormal(mean=0., stddev=0.05, seed=42)
 5
 6 # Top Layer (https://www.quora.com/Are-the-top-layers-of-a-deep-neural-network-the-first-layers-or-the-last-layers)
 7 # Weight Decay: https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-learning-with-weight-regularization/
 8 weight_decay=0.00001
 9 paper_model = tf.keras.Sequential([
10     layers.Dense(300, activation='tanh', input_shape=(FEATURES,), kernel_initializer=first_initializer, kernel_regularizer=l2(weight_decay)),
11     layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_regularizer=l2(weight_decay)),
12     layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_regularizer=l2(weight_decay)),
13     layers.Dense(300, activation='tanh', kernel_initializer=other_initializers, kernel_regularizer=l2(weight_decay)),
14     layers.Dropout(0.5), # Top Hidden Layer
15     layers.Dense(1, activation='sigmoid', kernel_initializer=outer_initializer, kernel_regularizer=l2(weight_decay))
16
17 ])
```

Fig. 6. TensorFlow code for the Deep Neural Network

Weight Initialization: The initial weights of the first hidden layer are set at random numbers based on a normal distribution with mean 0 and standard deviation of 0.1. The initial weights of the other hidden layers are similarly selected but using a random normal distribution with standard deviation of 0.001. Finally, the initial weights of the output layer are set using a random normal distribution with a standard deviation of 0.05. This has been done from line 2 to line 4 of Fig. 6, while the original model used the parameter '`istdev`' of the '`Tanh`' and '`Sigmoid`' functions (line 38-54 in Fig. 3)

Dropout: We included a "dropout" layer to limit the overfitting in the model. Dropout technique randomly removes interconnections between neurons in the network. In this way, each neuron has a chance to learn independently and become stronger. Consecutively, it removes the likelihood that the increased cooperation between neurons lead to overfitting [7]. This is visible in the parameter `Dropout` in line 14 of Fig. 6. While the original paper mentions using the dropout step in the top layer, there is no evidence of it in the code published in GitHub. We decided to keep the dropout layer for consistency with the published paper. Also, we made an assumption that the "top" layer is the last (output) layer in the network since more specifics were not provided in the paper or the original GitHub code.

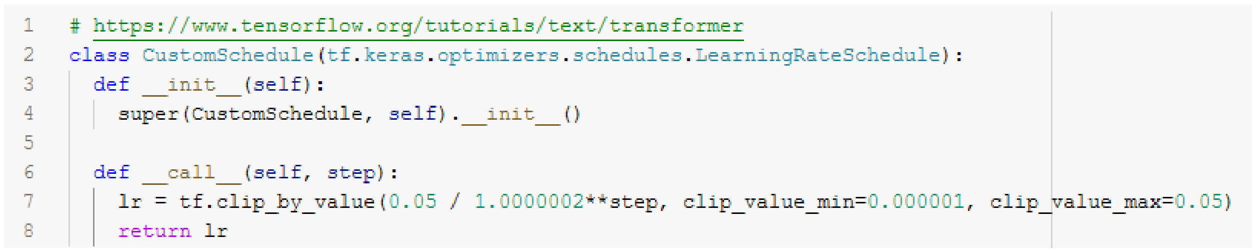
Optimizer: The original paper utilized the Stochastic Gradient Descent with 'Momentum' techniques to improve training speed and accuracy. This technique allows the process to find new values for the weights that are closer to the known values from the previous iterations (detailed description in 4.4). The momentum uses a coefficient that ranges from 0 to 1. The original code uses a momentum that increases linearly from 0.90 to 0.99 for the first 200 iterations and is then kept stationary for all future iterations. This has been coded by utilizing the '`MomentumAdjustor`' function (line 62 in Fig. 4 and lines 91-95 in Fig. 5). We have been unable to replicate such behavior using the TensorFlow 2 library, so we relied on the fixed momentum rate of 0.9 throughout all the iterations (see Fig. 7, line 3). As we will see in 4.4, this approach is standard in the latest methodology to train neural networks.



```
1 def get_optimizer():
2   lr_schedule = CustomSchedule()
3   return tf.keras.optimizers.SGD(lr_schedule, momentum=0.9)
```

Fig. 7. TensorFlow Optimization Parameters

Learning Rate Decay: Learning Rate Decay is also an important criterion to consider in training neural networks (as will be discussed later in detail in 4.2). In the original paper, the authors start with a learning rate of 0.05 (line 61 in Fig. 4), but then gradually reduce the learning rate exponentially at each mini-batch update (lines 83-86 in Fig. 4). In TensorFlow, the implementation is done using a `CustomSchedule` Class as indicated in Fig. 8 and line 2 in Fig. 7) and can be visualized in Fig. 9.



```python
# https://www.tensorflow.org/tutorials/text/transformer
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self):
    super(CustomSchedule, self).__init__()

  def __call__(self, step):
    lr = tf.clip_by_value(0.05 / 1.0000002**step, clip_value_min=0.000001, clip_value_max=0.05)
    return lr
```

Fig. 8 Learning Rate Decay Implementation in TensorFlow

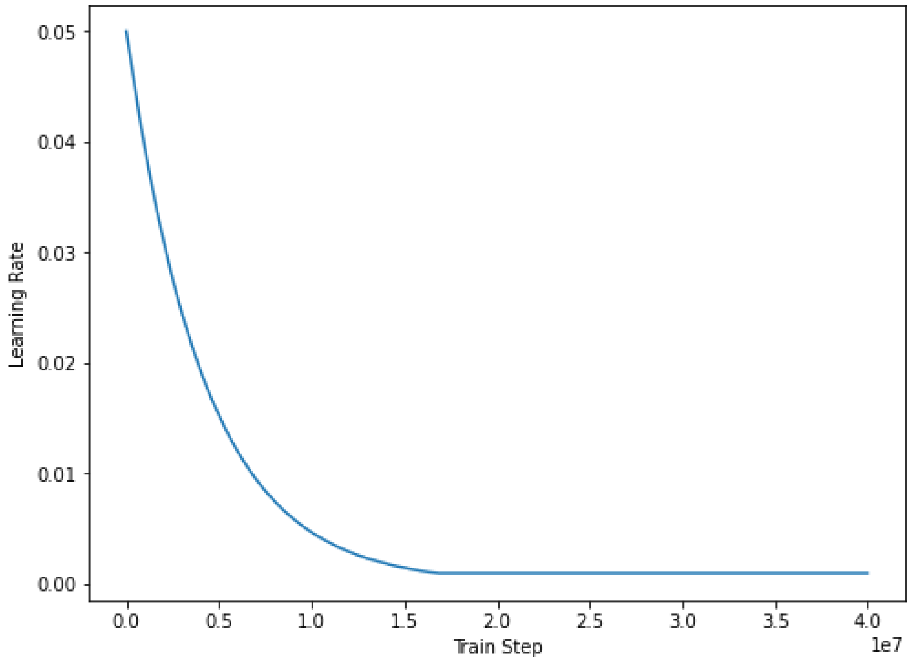
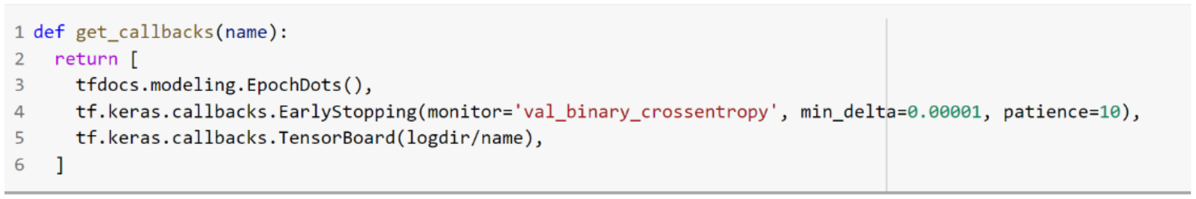


Fig. 9. Learning Rate Decay Visualization

Early Stopping: The original paper also used an early termination parameter, where the model stops the optimization process if there is not enough improvement in successive iterations. In line 69 of Fig. 4 the authors set an acceptable improvement of 0.00001 in the `valid_objective`, with a waiting period of 10 iterations. If there is no improvement beyond 0.00001 for at least 10 iterations the process will stop. Since not much information was provided about `valid_objective`, we assumed this to be the standard `binary_crossentropy` which is common for binary classification problems and applied the same criteria using the `EarlyStopping` function in line 4 of Fig. 10.

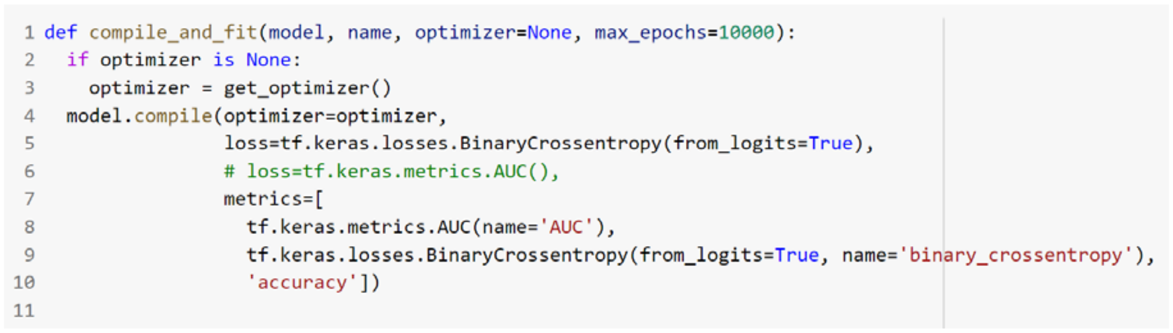


```python
1 def get_callbacks(name):
2   return [
3     tfdocs.modeling.EpochDots(),
4     tf.keras.callbacks.EarlyStopping(monitor='val_binary_crossentropy', min_delta=0.00001, patience=10),
5     tf.keras.callbacks.TensorBoard(logdir/name),
6   ]
```

.

Fig. 10. TensorFlow Early Stopping criteria

Metric: Lastly, to ensure we are using the AUC as measure of performance (as indicated in the original paper), we specified the `metrics.AUC` parameter in the compile section of the TensorFlow model (line 8 in Fig. 11).



```
def compile_and_fit(model, name, optimizer=None, max_epochs=10000):
  if optimizer is None:
    optimizer = get_optimizer()
  model.compile(optimizer=optimizer,
                loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                # loss=tf.keras.metrics.AUC(),
                metrics=[
                  tf.keras.metrics.AUC(name='AUC'),
                  tf.keras.losses.BinaryCrossentropy(from_logits=True, name='binary_crossentropy'),
                  'accuracy'])

```

Fig. 11. TensorFlow Compilation Parameters

Sample Training: Due to the limited amount of resources available, we only trained the model on 10,000 observations with a validation set of 1,000 observations. Fig. 12 show the performance of the model across the various epochs. We can see that the overall AUC increases from 0.54 up to 0.69 as we increase to 55 epochs. The solid line shows the model training, while the dotted line shows the validation step. There does not seem to be much improvement in the model's performance for the last 10 epochs so early stopping likely kicked in at this point and terminated the training.

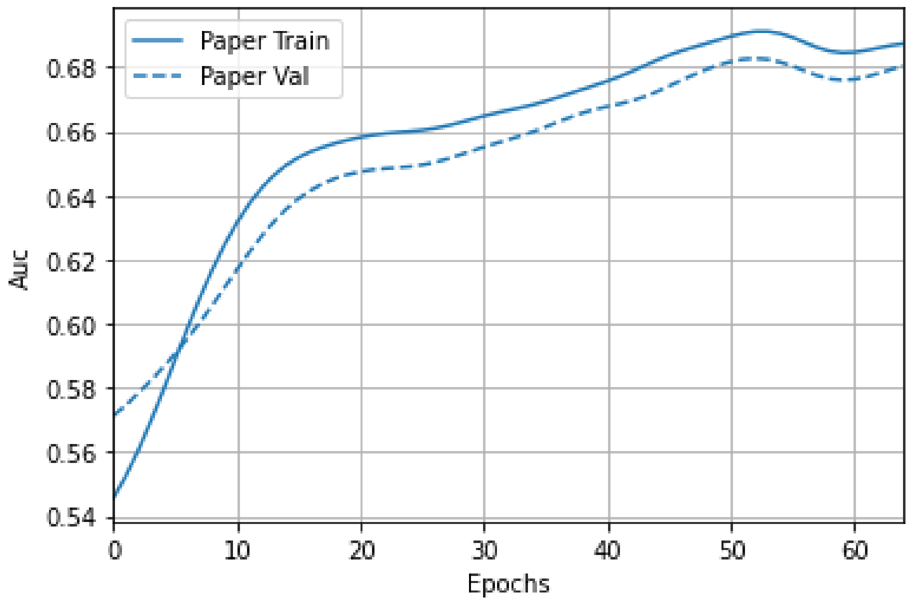


Fig. 12. TensorFlow Train and Test AUC output across the Training Epochs

4. Recommendations & Evaluations

Since the original paper was released in 2014, a lot of improvements have been introduced in the way neural network architectures are developed and trained. In this section, we will make some recommendation in the way the architecture and training of the neural network in this paper can be improved.

4.1 Activation Function

The original paper used a *tanh* activation function which (along with the Sigmoid activation) was the state of the art when the paper was released in 2014. The issue with this activation function (tanh) is that the derivative of this activation function has a peak of 1 at an input of 0, and its value falls sharply to less than 1 as the input diverges away from 0 (Fig. 13). This is an issue since the derivative of the activation function shows up in the backpropagation equation and is repeated as many times as there are hidden layers. Hence, for deep neural networks, this derivative (smaller than 1) is multiplied multiple times and results in a very small number. This means that any update to weights due to backpropagation will be very small and that the network will learn very slowly. Hence these activation functions are not suitable for deep neural networks.

Since 2014, better activation functions such as the Rectified Linear Unit (ReLU) and Leaky ReLU have been released (Fig. 14). As can be seen the derivative of the ReLU is always 1 when the input is greater than 0. This ensures that no matter how many layers there are, the derivative will not shrink to a small value when it has been multiplies multiple times. One of the disadvantages of ReLU is that if the input is negative, the derivative is 0 and hence the neuron stops learning. To avoid this, Leaky ReLU was introduced which has a small negative slope when the input is negative. This helps the neuron recover and possibly start learning again even if the input becomes negative. Other variants of ReLU such as Exponential Linear Unit have also been introduced recently (Fig. 15) which offer similar benefits but differ in how the function behaves when the input is negative. If this study were to be recreated today, we would recommend using one of these newer activation functions, especially since the authors used a deep neural network with multiple hidden layers.

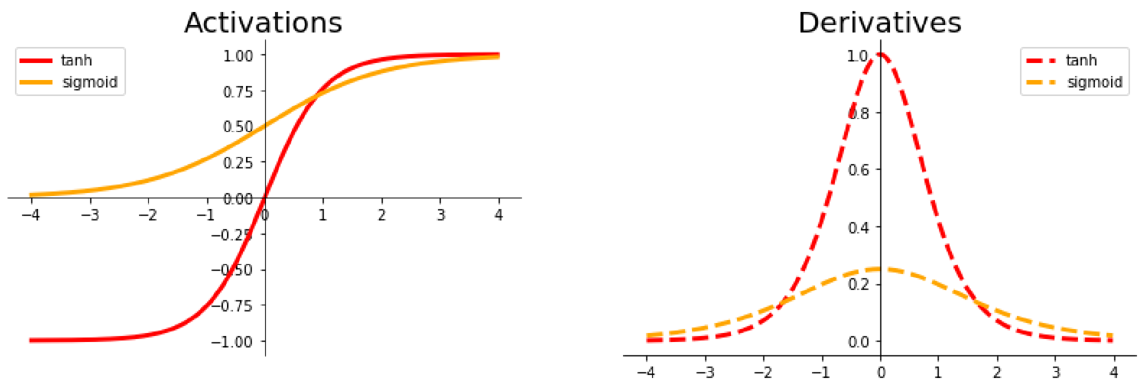


Fig. 13. Sigmoid and Tanh Activation Functions and their Derivatives

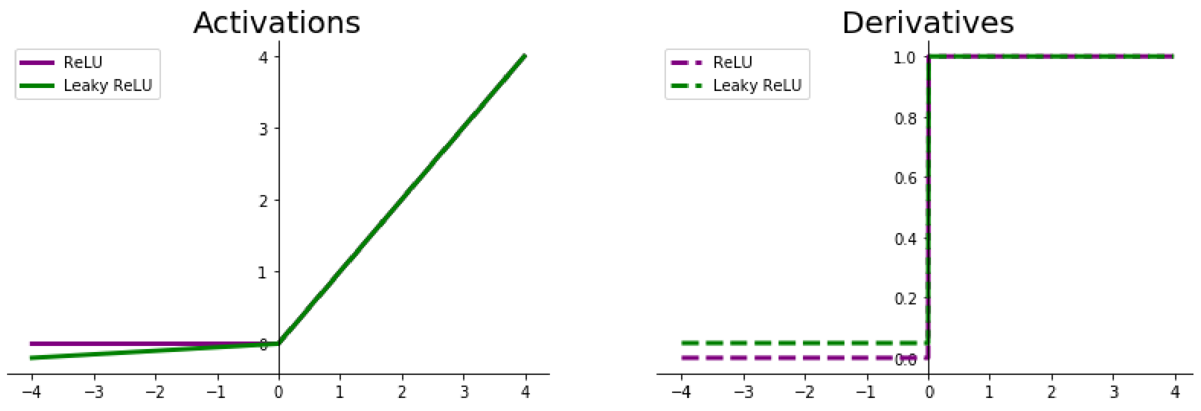


Fig. 14. ReLU and Leaky ReLU Activation Functions and their Derivatives

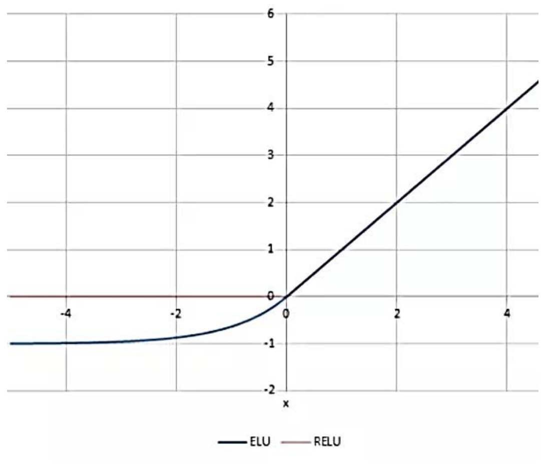


Fig. 15. ELU Activation Function compared to ReLU [8]

4.2 Learning Rate

The learning rate determines how large of a weight update (learning) is made after each mini-batch is processed. A larger learning rate provides the ability to reach the global minima quicker (and conversely a smaller learning rate takes more updates to reach the global minima). However, in reality, we want to start with a larger learning rate so that we can move quickly towards the global minima, but as we get closer to the global minima, we want to make sure that we take smaller updates (smaller learning rate) since we don't want to overshoot the global minima as indicated by the blue updates in Fig. 16. This process is captured efficiently by learning rate decay which essentially starts with a larger learning rate and gradually reduces the learning rate at the end of each mini-batch update (as shown by the green curve in Fig. 16). This paper also uses this method as discussed in the section above.

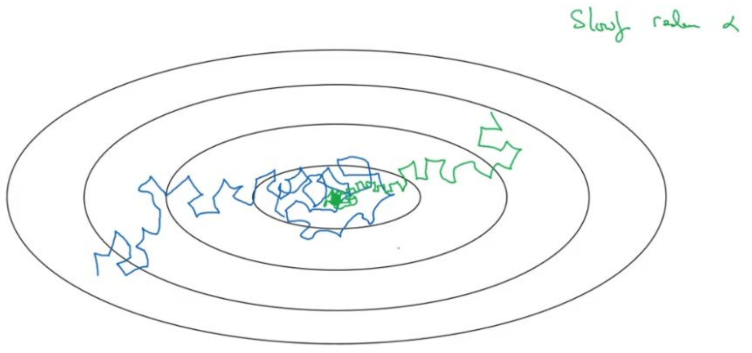


Fig. 16. Effect of Large Learning Rate as updates approach the Global Minima [9]

The issue with the previous approach is that the learning process is quite susceptible to saddle points which are regions where the cost function is flat (red line in Fig. 17). In these regions the gradients are quite small, and hence the updates made to the neural network weights will also be very small. If we were to use a learning rate decay and the network gets stuck in these regions when the learning rate is small, it can be quite difficult to get out of this and hence difficult to reach the global minima.

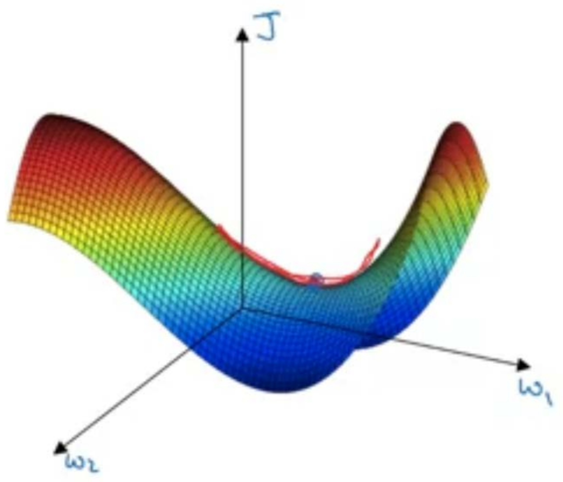


Fig. 17. Saddle Points in Neural Network Learning [10]

More recently, newer learning rate scheduling methods have been devised to avoid this problem. In particular, the “Cyclical Learning Rate” has been proven to be quite effective [11]. In this method, the learning rate decays as before, but after every few iterations, the learning rate is increased back again (Fig. 18). This ensures that if the network does get stuck in a saddle point, it is able to get out when the learning rate is increased after a few iterations. Another variant of this approach is called “fit one cycle” which essentially repeats the above process only once [11]. If this study were to be implemented again, we would recommend applying one of these newer learning rate scheduling methods in the training process.

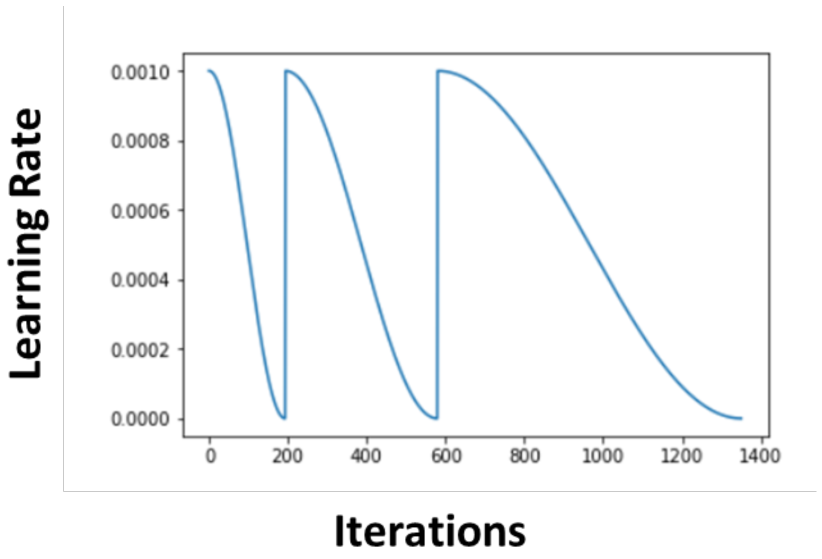


Fig. 18. Cyclical Learning Rate Scheduling Example [11]

4.3 Dropout

The original paper only added dropout to the “top” layer of the neural network. With the limited training that we performed, we found that the early stopping was triggered after only a few epochs (< 100). This may be happening since the model is overfitting (training error is reducing, but validation error is not reducing). This can in turn be due to a couple of reasons.

1. We only used a small subset of the data to perform a trail training. Since we used a big network (the one in the original paper) on a small subset of the data, it may have led to overfitting, or
2. The model could have been overfitting even when the full dataset was used by the authors.

If the actual reason is the second one (which can be confirmed after running the full training), we can potentially benefit from adding dropout in between more layers in the model architecture. Adding more dropout acts as a regularizer and prevents overfitting in the model. Essentially, the output of a certain percentage of neurons in the layer is blanked (made 0) out at random during the training. This prevents the future layers from depending on these blanked neuron outputs for making their decisions. This in turn prevents the network from depending on only a few "strong" neurons and hence, the other neurons must become stronger in turn (i.e. they must learn some latent feature representation as well). By spreading the latent feature learning across more neurons in the network, dropout helps the network to generalize better and not be overly reliant on a few "strong" neurons (overfitting).

4.4 Optimizer

Gradient Descent with Momentum (used in the paper) works better than just Stochastic Gradient Descent (SGD). The problem with plain SGD is described in a video by Andrew Ng [12]. We will use the example from this video as shown in Fig. 19. In this example, a typical SGD update might move along the blue line (direction of negative gradient) after each mini-batch update. However, since the contour of the cost function is quite steep on one side (vertical), the updated value after each mini-batch update may end up bouncing off the walls of the contour and this may result in the algorithm taking many iteration (mini-batch updates) to reach the global minima marked by the red dot. The other drawback is that due to the difference in the contours, we must use a small learning rate (to cater to the steeper gradient along the vertical direction). If we were to use a larger gradient, then the update can overshoot as shown in the purple line and end up diverging away from the global minima. This smaller learning rate in turn, slows down the training process.

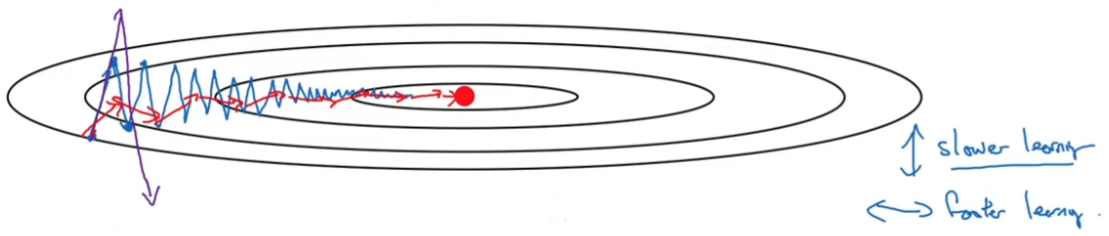


Fig. 19. Stochastic Gradient Descent Example [12]

Some of the challenges above can be avoided using Momentum. In this method, the update is determined by not just the gradient of the last mini-batch, but by an exponentially weighted moving average (EWMA) of the last few mini-batch gradients. In the example above, it would mean that the alternating "vertical" gradients would get cancelled out and the updates would take a much smoother and faster path towards the global minima (red updates in Fig. 19). SGD with Momentum is associated with an additional hyperparameter called β which corresponds to averaging out the last $\frac{1}{1-\beta}$ gradients. So, a β of 0.9 corresponds to taking the EWMA of the last 10 gradients whereas a β of 0.99 corresponds to taking the EWMA of the last 100 gradients. In this paper the author changes the momentum hyperparameter from 0.9 to 0.99 for the first 200 epochs. In today's world of neural network training, this is not needed as such and keeping it fixed to 0.9 should be reasonable [13]. In fact, we could not find an option in TensorFlow to change this momentum hyperparameter during the training process which points to the fact that keeping it constant would be enough.

More importantly, since the paper was published in 2014, other better optimizers have been introduced which tend to perform even better than SGD with Momentum. The Adam (Adaptive Moment Estimation) optimizer is the most widely used optimizer today. This builds upon the concept of Momentum. Intuitively, in addition to taking the EWMA of the last few gradients, this method also penalizes updates made for larger EWMA gradients, so they don't end up making larger updates just because of the larger EWMA values. This again helps to solve the issue explained above related to uneven contours. We would hence recommend using the Adam optimizer if this research were to be implemented again.

4.5 Batch Normalization

The challenge of uneven contours from an optimization perspective (as discussed in 4.4) can also be alleviated by making sure that the inputs being fed into the network are on the same order. This in turn ensures that the gradients are roughly on the same order of magnitude and hence the common learning rate used will learn across all parameters effectively. However, this can only be ensured for the input layer. The hidden layers will get their inputs from previous layers and if their output keeps changing during the training process as the weights are updated, this can cause difficulty in learning. After this paper was released in 2014, a new concept called Batch Normalization was introduced in 2015 [14]. This technique ensures that the inputs to the hidden layers of the neural network are also normalized and hence remain roughly around the same order of magnitude which in turn helps the learning process [15]. If this study were to be implemented again, we would recommend adding “Batch Normalization” to the hidden layers of this network.

4.6 Mini Batch Sizes

In neural network training, the hardware memory required to train a large neural network is large. This is due to two reasons, (1) The number of parameters in the neural network can be very large), and (2) The number of training examples need to be large for the neural network to learn effectively. Because of these reasons, it is often not possible to train using all the training data at once. What is often done is to divide the training data into mini batches and train with each mini-batch in succession. This paper also used the same approach with a mini-batch size of size 100.

Using a larger mini-batch means that we can go over entire data quicker (one epoch) and can in turn go over more epochs if necessary, leading to faster and better training (learning). Since the paper was released in 2014, several hardware advances have been made and it may be possible to use a larger batch size while training which could lead to faster training. It is also recommended to use batch sizes which are powers of 2 since they can fit into the RAM memory more efficiently without wasting space. We would recommend trying a mini-batch size of 256 or even 512 if this study were to be recreated. Often the breaking point can be found when the training returns an “out of memory” error at which point, the mini-batch size can be reduced by a power of 2 and the network retrained.

4.7 Number of Layers and Neurons

The number of layers and neurons in a network architecture is hard to determine upfront. The best way to determine these would be to start with a simple network and then diagnose the output of the network. For example, if the network is not able to learn well i.e. it is underfitting and the desired accuracy is not met, then the number of layers and neurons can be increased in order to increase the learning capacity of the network. On the other hand, if the network is overfitting, i.e. the training error is continuing to reduce while the validation error is not reducing, then the network size may be too large. In this case, either the number of layers and/or the number of neurons in each layer can be reduced.

The exact amount of change in the number of layers and neurons is hard to quantify and is best determined using a grid search approach. The paper uses a fixed grid search approach (Supplementary Table 2 in [16]). However, this kind of grid search can be very costly because if one of the hyperparameters (number of layers or neurons) is not helpful in improving the performance of the network, then we would waste a lot of iterations in the grid search (Fig. 20). A Random Grid search is more efficient instead since it does not just try the same values of any hyperparameter again and again. Alternately, Bayesian Grid Search could also be used which is even more efficient in the hyperparameter search process [17]. If we were to change the network architecture in the future (example adding dropout to more layers), we would recommend using these more efficient grid search approaches instead of a fixed grid search used by this paper.

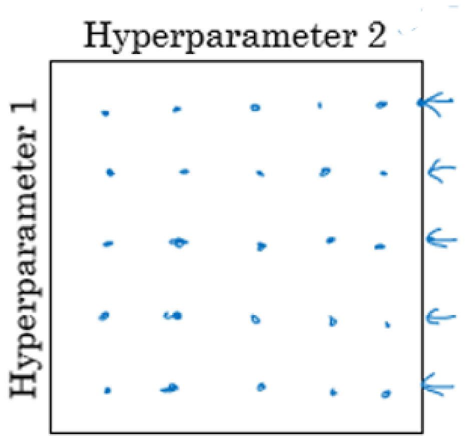


Fig. 20. Fixed Grid Search Example [13]

4.8 Evaluation Metric

The evaluation metric used by this paper was the Area under the ROC Curve. This is a valid evaluation metric for a binary classification problem. Intuitively, this metric captures how well the model is doing with respect to a random guess. AUC value for a random guess would be 0.5. Any value above 0.5 means that the model is doing better than a random guess and a value equal to 1 means that the model is perfect.

In this case, we saw that the data was roughly balanced with equal number of positive and negative examples. Hence accuracy could have also been used as an additional metric to evaluate the model. Another metric that could have been used is the F1 Score which balances the Precision (what percentage of positive predictions were actual positives) and Recall (what percentage of actual positive observations were correctly predicted as positive by the model) of the positive class. This is an especially useful metric to track (instead of accuracy), especially when the datasets are not balanced, since accuracy can be a misleading metric in these cases. All these are readily available in TensorFlow and can be included in the metric section in lines 7-10 of Fig. 11 .

5. Conclusion

The original paper by Valdi et.al. provided breakthroughs in the way deep learning can be used in the field of particle physics. It offered considerable improvements over conventional methods in detecting the Higgs boson particles from millions of records. However, this paper and neural network architecture were implemented in 2014. Although the techniques used in the paper were state of the art at that time, several enhancements have been made since then in the field of deep learning since then. In this case study, we show that the original paper (implemented in Pylearn2 which is not supported today) can be reimplemented in the latest TensorFlow framework. More importantly, we also recommend improvements that can be made to the network and training process which can potentially improve the performance of the model. The actual implementation of these enhancements and quantification of any improvements will be left as a post case study exercise.

6. References

[1] "UCI HIGGS Dataset," [Online]. Available: https://archive.ics.uci.edu/ml/datasets/HIGGS. [Accessed 16 11 2020].

[2] P. Valdi, P. Sadowski and D. Whiteson, "Searching for Exotic Particles in High-Energy Physics with Deep Learning," 2014.

[3] "Pylearn2 GitHub," [Online]. Available: https://github.com/lisa-lab/pylearn2. [Accessed 16 11 2020].

[4] Google Inc., "Collaboratory FAQ," Google, 2020. [Online]. Available: https://research.google.com/colaboratory/faq.html. [Accessed 14 11 2020].

[5] "GitHub Higgs Susy," [Online]. Available: https://github.com/uci-igb/higgs-susy/blob/master/higgs/layers4_width300_lr005_m200_wd000001_all.py. [Accessed 16 11 2020].

[6] Machine Learning Mastery Pty. Ltd., "How to Use Weight Decay to Reduce Overfitting of Neural Network in Keras," Machine Learning Mastery Pty. Ltd., 21 November 2018 . [Online]. Available: https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-learning-with-weight-regularization/. [Accessed 15 November 2020].

[7] Towards Data Sciecne, "Understanding And Implementing Dropout In TensorFlow And Keras," 18 May 2020. [Online]. Available: https://towardsdatascience.com/understanding-and-implementing-dropout-in-tensorflow-and-keras-a8a3a02c1bfa. [Accessed 15 November 2020].

[8] Read The Docs, "Activation Functions," 2020. [Online]. Available: https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#elu. [Accessed 15 11 2020].

[9] "Learning Rate Decay (Andrew Ng)," [Online]. Available: https://www.youtube.com/watch?v=QzulmoOg2JE&list=PLOZ51y9Re6jX0SVmY_WXgpTrOlyol5XgD&index=693. [Accessed 16 11 2020].

[10] "Saddle Point (Andrew Ng)," [Online]. Available: https://www.coursera.org/lecture/deep-neural-network/the-problem-of-local-optima-RFANA. [Accessed 16 11 2020].

[11] "1-Cycle Learning Rate," [Online]. Available: https://iconof.com/1cycle-learning-rate-policy/. [Accessed 16 11 2020].

[12] "Gradient Descent with Momentum (Andrew Ng)," [Online]. Available: https://www.youtube.com/watch?v=k8fTYJPd3_I. [Accessed 16 11 2020].

[13] "Hyperparameter Tuning (Andrew Ng)," [Online]. Available: https://www.coursera.org/lecture/deep-neural-network/tuning-process-dknSn. [Accessed 16 11 2020].

[14] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv,* 2015.

[15] A. Ng, "Why Does Batch Norm Work?," [Online]. Available: https://www.youtube.com/watch?v=nUUqwaxLnWs.

[16] B. Pierre, P. Sadowski and D. Whiteson, "Searching for Exotic Particles in High-Energy Physics with Deep Learning," *arXiv,* 2014.

[17] scikit-optimize, "Comparing Surogate Models," [Online]. Available: https://scikit-optimize.github.io/stable/auto_examples/strategy-comparison.html. [Accessed 16 11 2020].

7. Appendix

The entire code used for this case study is available in the following GitHub repository: Repository Link