Report for the Deep Learning Course Assignment 2

Daniel Sanchez Santolaya
daniel.sanchezsantolaya@student.uva.nl

Abstract

In this task, I have implemented a Multilayer perceptron using TensorFlow. Results of different experiments are shown using the CIFAR10 dataset. The experiments were useful to learn how different hyperparameters and methods can vary the performance and how they interact. The best found model is shown and analyzed.

1 Task 1

In this section, we can learn about some concepts of TensorFlow.

1.1 Describe TensorFlow constants, placeholders and variables. Point out the differences and provide explanation of how to use them in the context of convolutional neural networks.

Constants are elements that doesn't change during the operations in the graph. In the context of convolutional neural networks(CNN) we cannot use them for the input of the network (X) and the weights (W), but we could use them to represent the learning rate if it doesn't change during the training, for example.

The placeholder operation is used to feed data on execution. It generates and error if not data is supplied. In the context of CNN it can be used to feed the network with the input data samples (X).

Variables are in-memory buffers containing tensors that can change during the operations in the graph. They must be explicitly initialized. In the context of CNN they can be used to represent the parameters of the model, which change during the training.

1.2 Give two examples of how to initialize variables in TensorFlow. Provide an example code (snippet).

Example 1: We can use *tf.initialize_all_variables()*, which adds an operation to run all the variable initializers:

```
# Create two variables.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name="weights")
biases = tf.Variable(tf.zeros([200]), name="biases")

# Add an op to initialize the variables.
init_op = tf.initialize_all_variables()

# Later, when launching the model
with tf.Session() as sess:
  # Run the init operation.
  sess.run(init_op)
  ...
  # Use the model
  ...
```

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Example 2: Initialize from another variable, using *weights.initialized_value()*. For example, in the next code snippet, we create the variable *weights* with random values. Then, we create the variable *w2* using the value of assigned to weights in the initialization.

```
# Create a variable with a random value.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name="weights")

# Create another variable with the same value as 'weights'.
w2 = tf.Variable(weights.initialized_value(), name="w2")
```

1.3 What is the difference between *tf.shape(x)* and *x.get_shape()* for a Tensor x?

The difference is that *tf.shape(x)* returns the dynamic (true) shape of the tensor *x* and *x.get_shape()* returns the static (inferred) shape.

The static shape is inferred from the operations that were used to create the tensor. However, during the graph construction, sometimes this shape cannot be inferred completely (it could depend of the values of other elements), and as a result when we use *x.get_shape()* we can obtain an *'?'* as a tensor shape. In this case, at runtime we can access to the dynamic shape (which is always fully defined-it) via *tf.shape(x)*.

1.4 What do tf.constant(True) == True and tf.constant(True) == tf.constant(True) evaluate to? What consequence does that have on the use of conditionals in TensorFlow?

They both look if the constant in the left side is the same element than the element in the right side, so in both cases is False, as they are not the same element (even if the have the same value). If we want to do the comparison of the value in an IF statement we can use *tf.constant(True).eval() == True* or *tf.constant(True).eval() == tf.constant(True).eval()*

1.5 What is the TensorFlow equivalent of if ... else ... when using Bool Tensors? Write down a short example code for such an if ... else ... statement in TensorFlow and report the results.

We can use tf.cond(pred, fn1, fn2) as equivalent of if..else. In the next example, the comparison x < y is made, as it is True, it applys fn1, which adds x and y. The final result is 15:

```
x = tf.constant(3)
y = tf.constant(4)
z = tf.mul(x,y)
result = tf.cond(x < y, lambda: tf.add(x , z), lambda: tf.square(y))
with tf.Session() as sess:
  print(sess.run(result))
```

1.6 Name 3 TensorFlow components that you need in order to run computations in TensorFlow.

Tensors. The data is represented in Tensors, which are typed multi-dimensional arrays. For example, we can represent a dataset with a 2-D tensor, where each sample in the dataset is a vector of one dimension.

Operations. The operations are the computations in the graph. Nodes in the graph are operations, which takes zero or more Tensors, performs some computation, and produces zero or more Tensor. An example is the product of two tensors.

Session. The sessions are in charge to place the graph of operations onto devices, such as CPUs or GPUs. It provides the methods to execute the operations in the graph.

1.7 What are variable scopes used for? Is there a difference between a variable scope and a name scope?

The scopes provide a simple namespacing that add prefixes to variable names within scope and are used to avoid clashes between variables. Also is useful when we want to share variables (using the reuse-flag).

The differences between a variable scope and a name scope is that the former creates the namespace for both variables and operators in the graph, while the latter creates the namespace only for operators. The difference can be seen in the next example, where the name scope is ignored by *tf.get_variable()*, but not for the operator:

```
with tf.name_scope("my_scope"):
    v1 = tf.get_variable("var1", [1], dtype=tf.float32)
    v2 = tf.Variable(1, name="var2", dtype=tf.float32)
    a = tf.add(v1, v2)
print(v1.name) # var1:0
print(a.name)  # my_scope/Add:0

with tf.variable_scope("my_scope"):
    v1 = tf.get_variable("var1", [1], dtype=tf.float32)
    v2 = tf.Variable(1, name="var2", dtype=tf.float32)
    a = tf.add(v1, v2)
print(v1.name) # my_scope/var1:0
print(a.name)  # my_scope/Add:0
```

1.8 Can you freeze a given variable tensor such that it will maintain its value during, for instance, optimization? How?

One way to do this is use the argument trainable=False when creating the variable:

```
tf.Variable(my_weights, trainable=False)
```

If this value is false, then the variable is not added to the collection used as a list of variables by the Optimizer classes.

Another way is to specify it in the minimize function of an optimizer, where you can specify a scope. For example:

```
optimizer = tf.train.AdagradOptimzer(0.01)
opt = tf.train.GradientDescentOptimizer(learning_rate=0.1)
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               "scope/model1/vars")
first_train_op = optimizer.minimize(cost, var_list=first_train_vars)
```

1.9 Does TensorFlow perform automatic differentiation? Name two occasions in which TensorFlow mechanism for differentiation can make your life more difficult. What are the advantages?

Yes, TensorFlow performs automatic differentiation. Given a graph of operations, TensorFlow uses automatic differentation (backpropagation) to add new ops representing gradients with respect to the existing operations.

This might be a problem for debugging. The operations are not computed when the line of code is interpreted. Instead, they need to be compiled to C as a computational flow graph and then it can be executed. This makes the debugging hard and slow.

The main advantage is that we do not need to know the derivative of the function to compute it. Also, knowing the structure of the graph before the compilation, TensorFlow can optimize it to compute the gradients faster, saving the necessary values in the forward pass for example.

1.10 Describe two ways to feed your own data into a TensorFlow graph. Shortly explain the pipelines.

We can use tf.placeholder and feed_dict to feed data into a graph, for example:

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.mul(input1, input2)
with tf.Session() as sess:
   sess.run([output], feed_dict={input1:[7.], input2:[2.]})
```

A second way is use preloaded data, which can be used when the whole dataset fits in memory. For example, the data can be stored in a constant:

```
training_data = ...
training_labels = ...
with tf.Session():
  input_data = tf.constant(training_data)
  input_labels = tf.constant(training_labels)
  ...
```

Another way is reading the data from files using a pipeline for reading records. This pipeline has several stages:

- We start with a list of filenames
- Optionally, we shuffle the filenames
- Create a filename queue,
- A reader for the file format is used to read the different files
- A decoder for a record is used to obtain the records obtained in the reader
- Optionally, the decoded records are preprocessed
- Create a queue with the samples.

2 Task 2

In this section results are shown for the cifar10 dataset when using the next settings:

- Learning rate: 0.002
- No regularization
- Weight initialization: Gaussian normal distribution with standar deviation 0.0001
- Batch size: 200
- Number of iterations: 1500
- Dropout rate: 0
- Architecture: 1 hidden layer with 100 units.
- Activation function: ReLu
- Optimizer: Stochastic Gradient Descent

In figure 1 we can see the progress of the training and test loss with the number of steps. Figure 2 shows the accuracy of training and test with the number of steps.

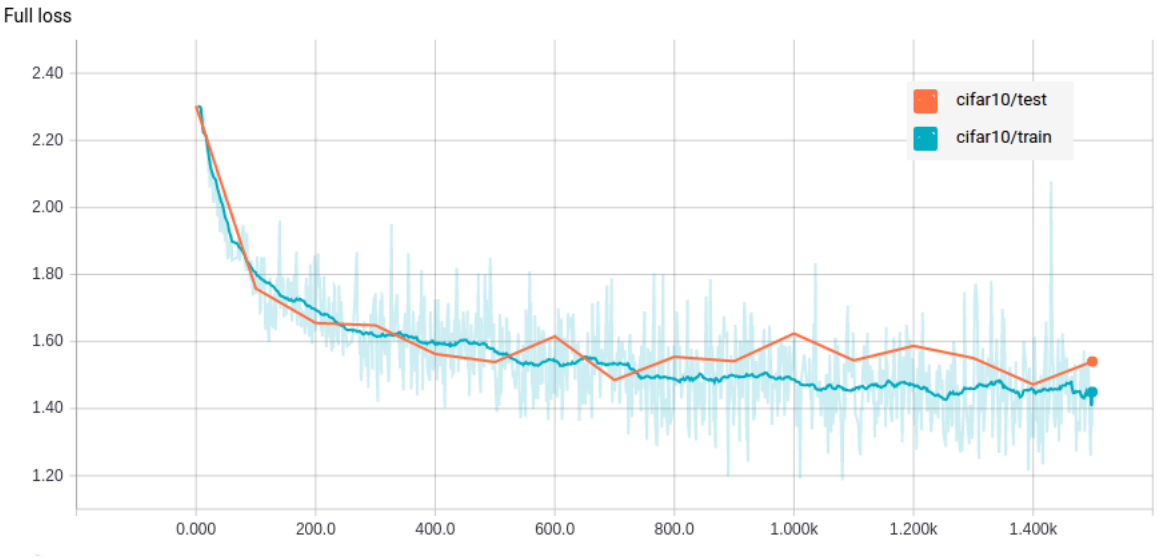

Figure 1: Training and test loss with respect the number of steps.

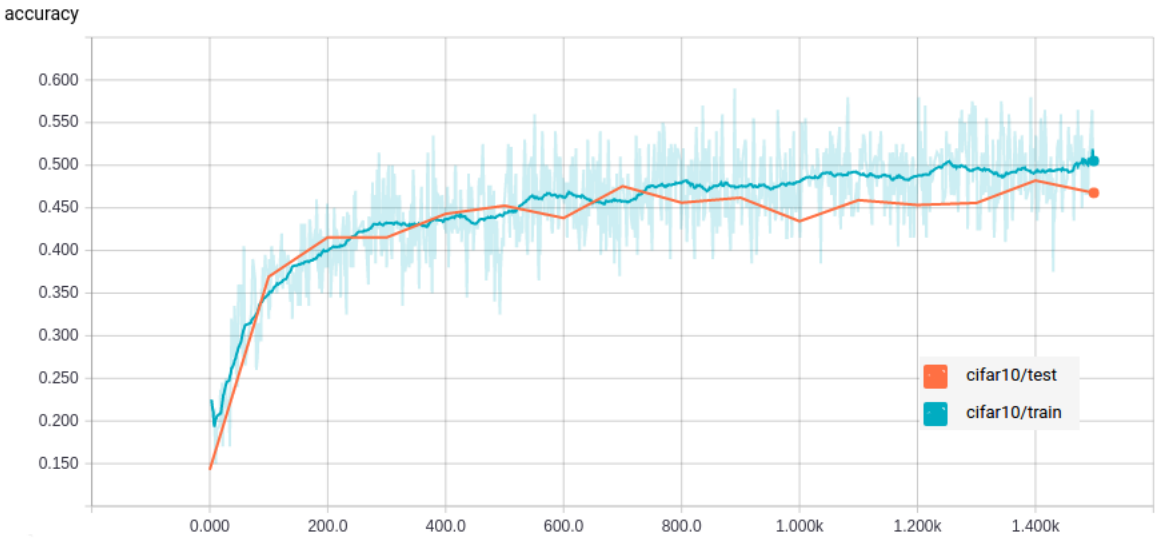

Figure 2: Training and test accuracy with respect the number of steps.

Figure 3 shows the graph of the model visible in the TensorBoard, which represents our model with 1 hidden layer, and the ouput weights connected to the softmax loss.

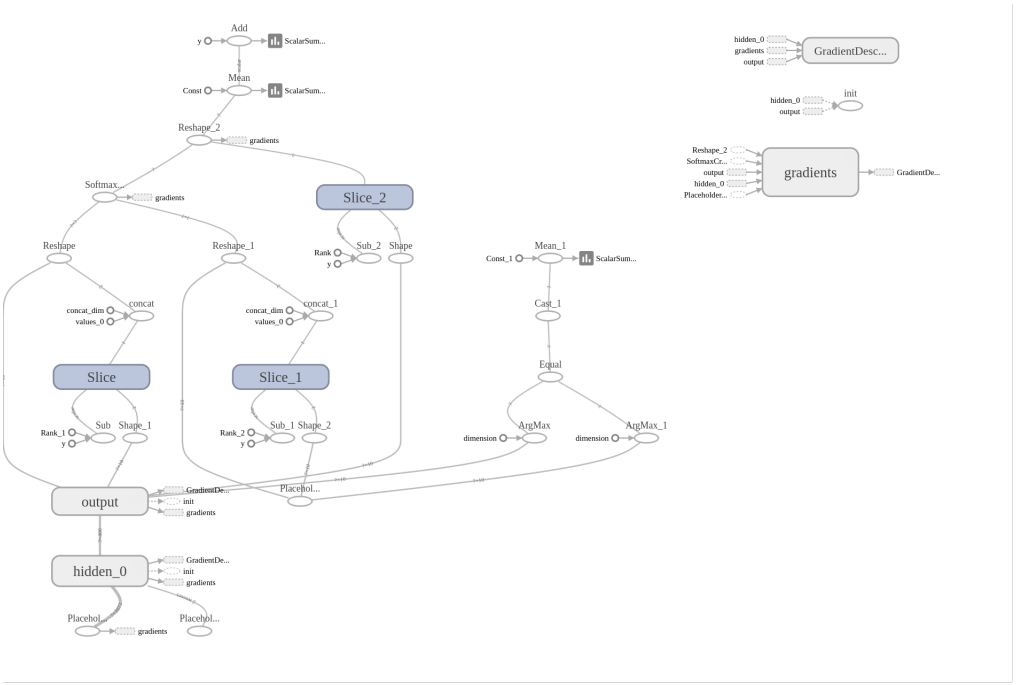

Figure 3: Graph of the model.

Figure 4 shows the histograms of the values of the weights and bias for the hidden layer and the connections from the hidden layer to the logits. We see that the values are more sparse when we advance in the training.

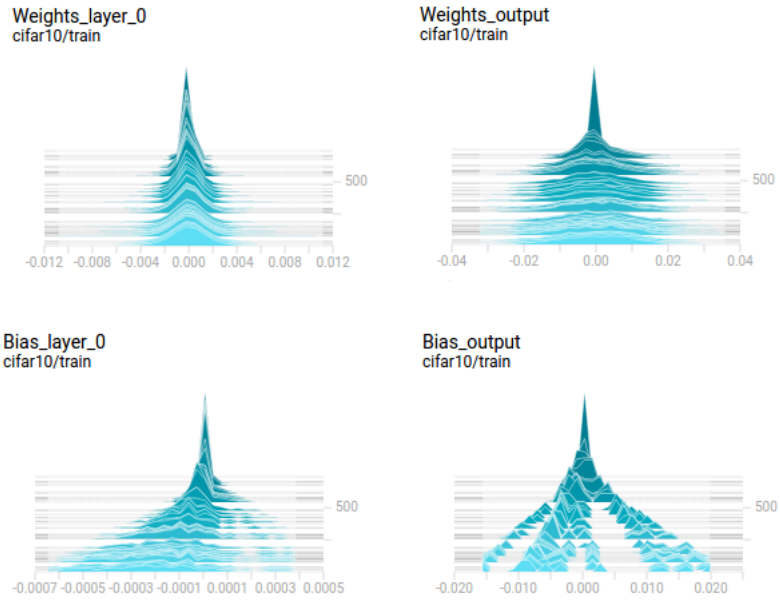

Figure 4: Histograms of the values of the weights(top) and bias(bottom) for the hidden layer(left) and the weights and bias connected to the output(right).

In figure 5 we can see the histogram of the values of the logits. As in the case of the weights, the values of the logits are more sparse when we advance in the training.

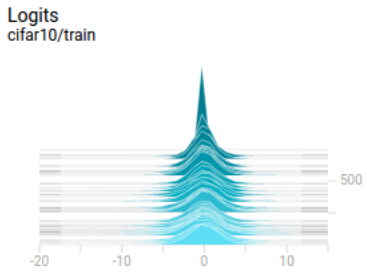

Figure 5: Histograms of the values of the logits

3 Task 3

In the next subsections we can observe some experiment performed to see the impact of the different hyperparameters in the network. The start default settings are:

- Learning rate: 0.002
- Weight initialization: Gaussian with standard deviation 0.0001
- Batch size: 200
- Number of iterations: 1500
- Network architecture: 1 hidden layer with 100 hidden units
- Dropout rate: 0
- Weight regularization: No regularization
- Activation function: ReLu
- Optimizer: SGD

In the next section some of these hyperparameters are varied and the results are shown.

3.1 Experiment-1: Weight initialization

Figures 6 and 7 shows the accuracy and lost of train and test sets, using normal Gaussian initialization, for a different values the standard deviation. We can observe that when the values the standard deviation is higher (0.01), the loss and accuracy are worse, both during training and testing. The reason is that the weights are initialized with values that are too high so we start with a worse solution, the loss is much higher at the beginning, and then the whole process is affected.

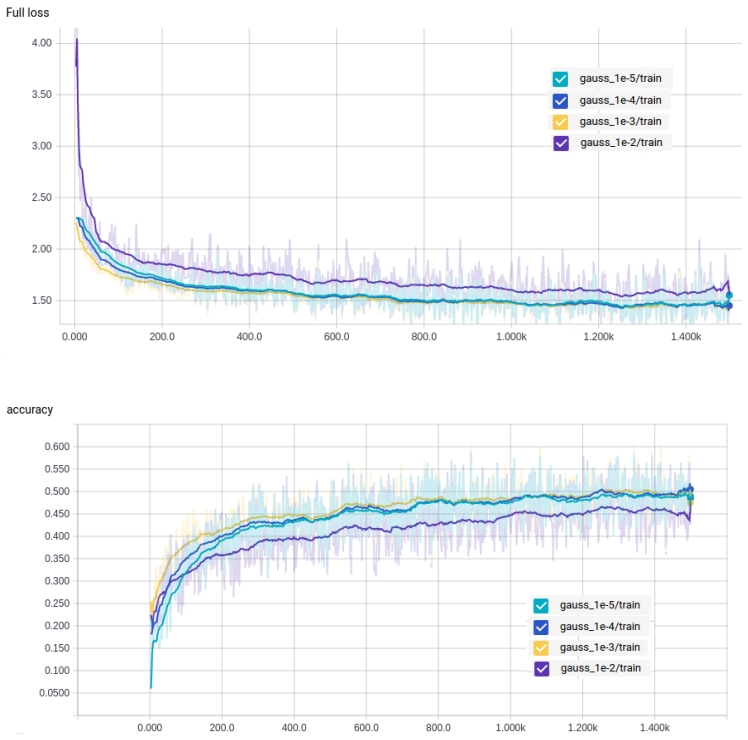

Figure 6: Loss(top) and accuracy(bottom) with normal Gaussian initialization in train set.

Figure 7: Loss(top) and accuracy(bottom) with normal Gaussian initialization in test set.

Figures 8 and 9 shows the accuracy and lost of train and test sets, using uniform initialization in the range [-scale, scale]. We see that in this case all the scale values converge to a similar loss and accuracy, however, smaller scale values (1e-5 and 1e-4) seems to converge slower, as they have worse performance and the beginning.

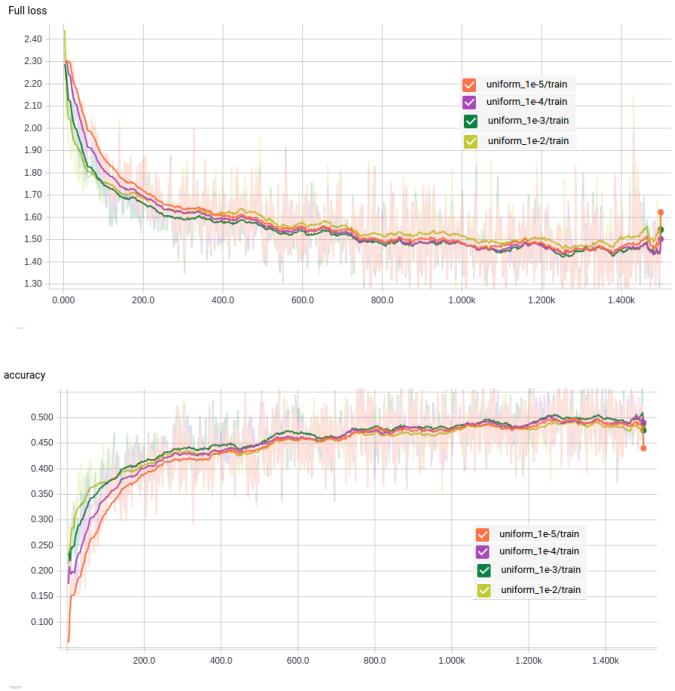

Figure 8: Loss(top) and accuracy(bottom) with uniform initialization in train set.

Figure 9: Loss(top) and accuracy(bottom) with uniform initialization in test set.

Figures 10 and 11 compare the Xavier initialization with uniform and Gaussian initialization, in train and test sets. As we see, this initialization works worse than than uniform and Gaussian. It starts with a much worse loss and this affects the whole process of training. One possible reason is that using Xavier initialization for this small network the weights start with too higher values.

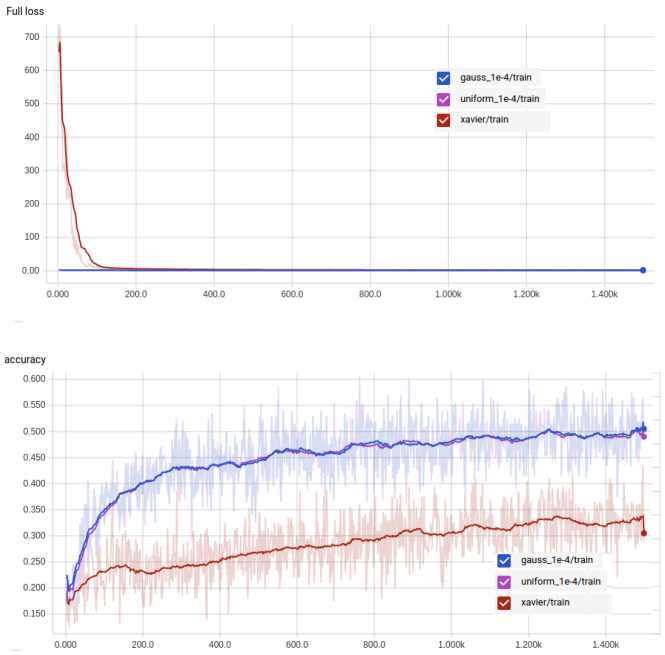

Figure 10: Loss(top) and accuracy(bottom) with Xavier initialization in train set(top). They are compared with Gaussian and Uniform initialization.

Figure 11: Loss(top) and accuracy(bottom) with Xavier initialization in test set(top). They are compared with Gaussian and Uniform initialization.

3.2 Experiment-2: Interaction between initialization and activation

Figures 12 and 13 show the loss and accuracy for the train and test sets respectively, using Xavier initialization and Gaussian normal initialization with standard deviation 0.001. The experiments were performed using two activation functions: ReLu and tanh. If we observe the loss in the training set, we see that Xavier initialization when using ReLu activations is the worse combination. It starts with a high loss compared with the other combinations, and it ends with a worse loss (2.440). Using tanh activation improves the performance of the Xavier initialization (the result at the end of the training is 1.949), but it is still worse than the Gaussian normal initialization with tanh (1.747 at the end of the training) and ReLu (1.451). We can observe this pattern in the loss in the test set as well. If we look at the accuracy, we see that in the train set, the Xavier initialization with using both activations get similar accuracy, but in the test set using tanh activation gives a better end solution. Also, the Gaussian normal initialization outperforms in accuracy the Xavier initialization.

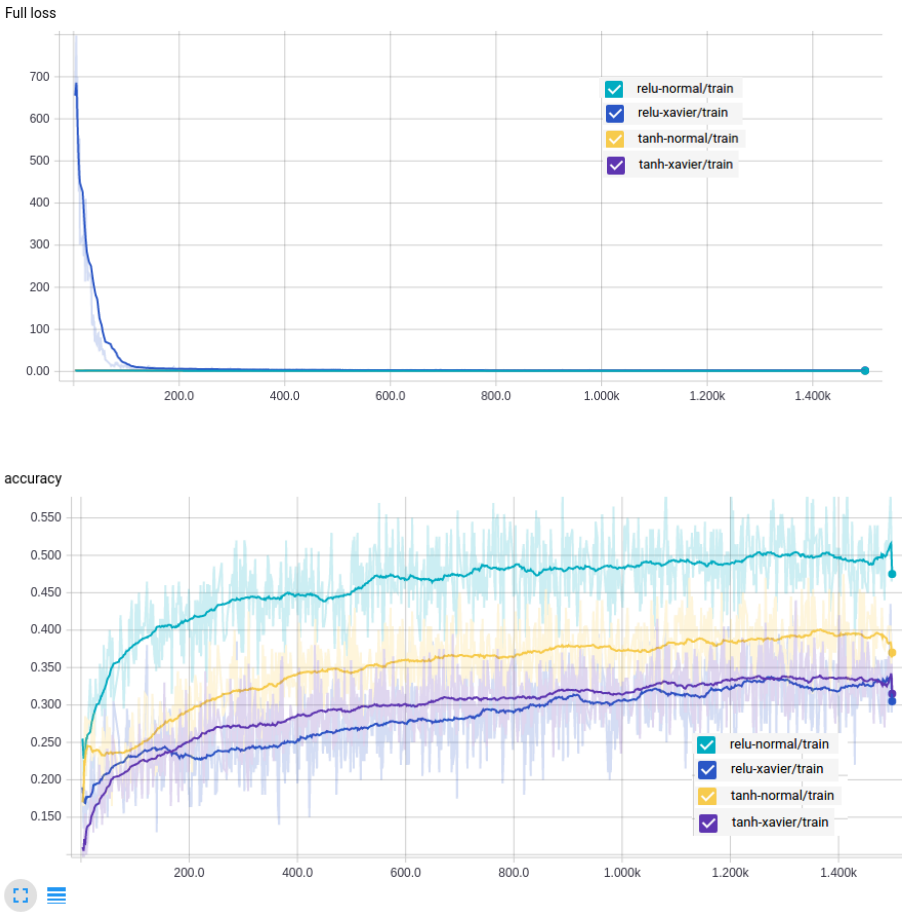

Figure 12: Loss(top) and accuracy(bottom) with Xavier initialization and Gaussian normal initialization with standard deviation 0.001, in train set. Only one hidden layer with 100 neurons was used.

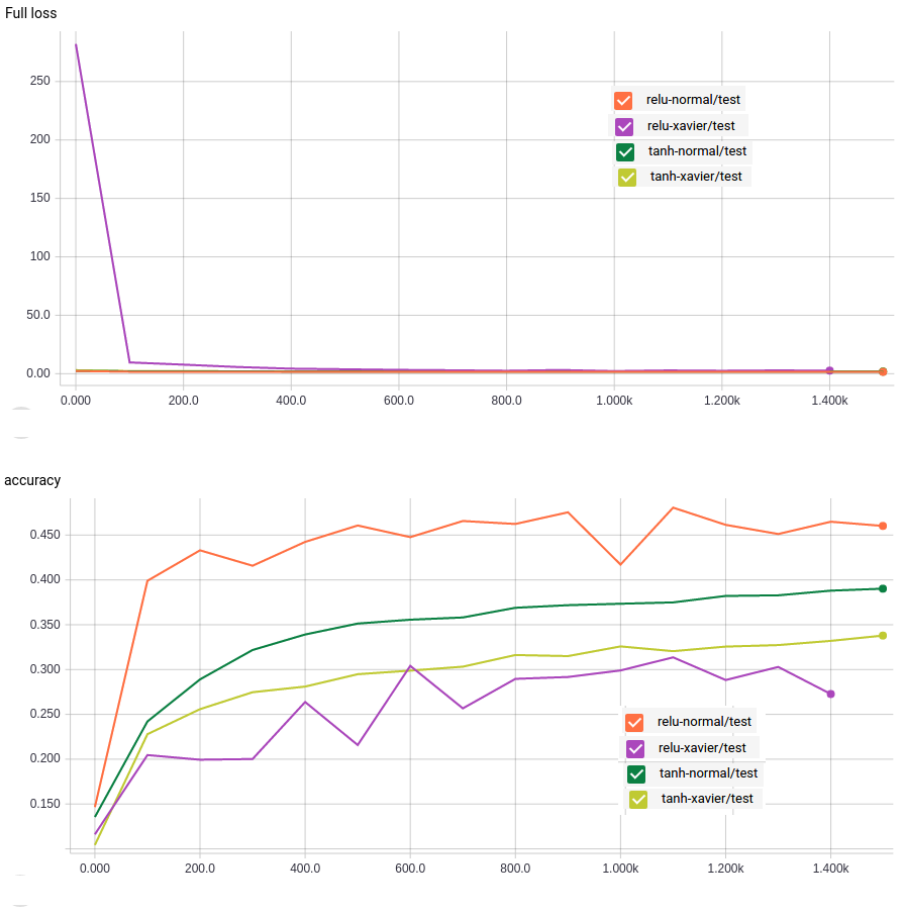

Figure 13: Loss(top) and accuracy(bottom) with Xavier initialization and Gaussian normal initialization with standard deviation 0.001, in test set. Only one hidden layer with 100 neurons was used.

Clearly, we see that there is an interaction between the activations and the initializations. While the Gaussian normal distribution seems work better with ReLu activations, Xavier seems to work better with tanh activations. In fact, Xavier initialization was a great solution for weight initialization when using activation functions like sigmoid or tanh which can suffer from vanishing or exploding gradient when dealing with deep networks. However, when dealing with ReLu layers, the benefits of Xavier initialization can be reduced, specially if we deal with networks with just 1 hidden layer. To make stronger conclusion can perform the experiment with deeper networks.

3.3 Experiment-3: Architecture

Figures 14 and 15 show the loss and accuracy for the train and test sets respectively, using Xavier initialization and Gaussian normal initialization with standard deviation 0.001, and using tanh and ReLu activations. The difference with the experiment 2 is that in this case the architecture of the network is formed by two hidden layers with 300 units in each. We observe again that Xavier initialization with ReLu activations starts with a worse loss compared with the other cases. However, in this case at the end the 1500 iterations both Xavier initializations have lower loss than the Gaussian normal initialization with tanh activations, as we see in table 1. The Gaussian normal initialization is still giving the best results. This is also observable in the accuracy plots. Looking at the accuracy, we can appreciate an interesting fact: the Xavier initializations converge faster than the Gaussian initialization with ReLu activation.

If we compare the test accuracy with experiment 2, we observe that the networks with Xavier initializations have improved their accuracy, while the networks with Gaussian initializations have obtained a worse accuracy.

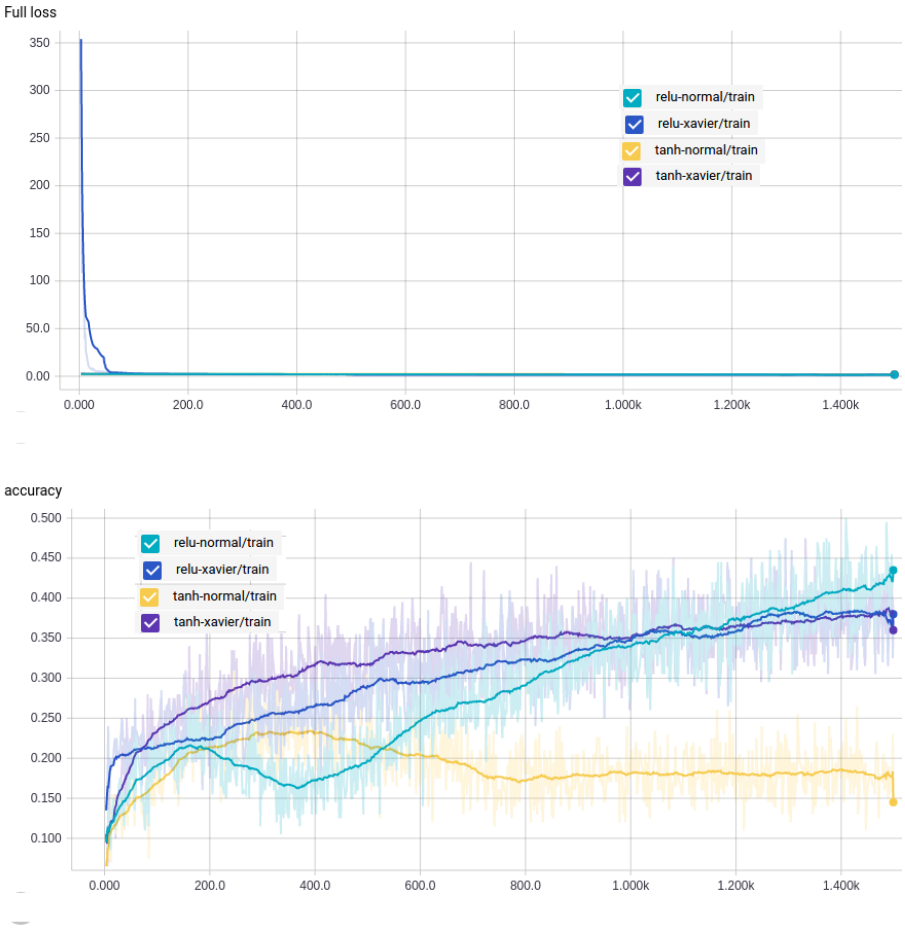

Figure 14: Loss(top) and accuracy(bottom) with Xavier initialization and Gaussian normal initialization with standard deviation 0.001, in train set. Two hidden layers with 300 units were used.

	Loss (Train)	Loss (Test)
relu-normal	1.615	1.613
relu-xavier	1.979	1.720
tanh-normal	2.043	2.068
tanh-xavier	1.851	1.874

Table 1: Loss after 1500 iterations in train and test sets for experiment 3.

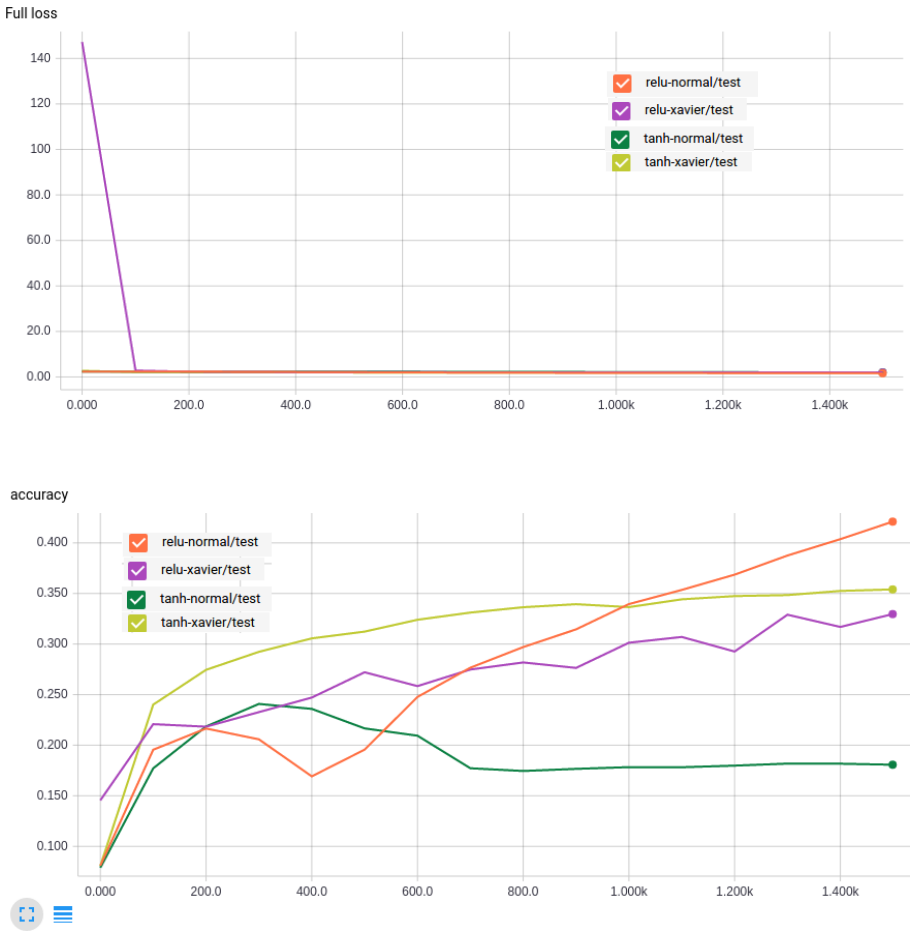

Figure 15: Loss(top) and accuracy(bottom) with Xavier initialization and Gaussian normal initialization with standard deviation 0.001, in test set. Two hidden layers with 300 units were used.

Therefore, Xavier initialization seems to adapt better than Gaussian normal initialization when using deeper neural networks with respect models with only one hidden layer. As the architecture of the network grows, the weight initialization becomes more important. If the weights are too small, then the variance of the signals diminish in every layer, dropping to a really low value, which is a problem (specially with functions like sigmoid as it losses the non-linearity at 0). If the weights are too large, then the variance of the signal increases at eatch layer, and eventually it becomes too large. This is again a problem, specially with functions like sigmoid and tanh were the gradients approach to zero with larger values, which leads to no-learning. Xavier initialization becomes very useful in this circumstances trying to keep the same variance of the input signal in the different layers.

3.4 Experiment-4: Optimizers

In this experiment, we compare the next different optimizers: Stochastic Gradient Descent (SGD), Adaptative Gradient Descent (AdaGrad)[1], AdaDelta[2], RMSprop and Adaptive Moment Estimation (Adam)[3]. Every network used in this experiment is using a learning rate (initial for some algorithms) of 0.001, 2 hidden layers of 300 hidden units each, ReLu activations, Gaussian initialization with standard deviation 0.001.

Figures 16 and 17 show the loss and accuracy for the train and test sets respectively, using the different optimizers. We can observe that the optimizer Adam has the best performance with the used settings. It has the best accuracy and lowest loss in the train and test sets. It also converges faster than the other algorithms. Therefore, it seems the best choice with this architecture and number of steps. However, we notice that there are three algorithms that could need more iterations to converge.

Adadelta seems to be in a very in a early stage of the optimization, as we see that the loss starts to decrease after 1000 iterations, and although the accuracy seems that does not improve, it could be higher if we execute the optimization more iterations. In a similar situation seems to be the SGD optimizer, although in this case it seems to be in a more advanced stage of the optimization, as it has started to get better accuracy. Finally, Adagrad seems to be close to the converge, however in the last steps its performance is still improving. Therefore, if we execute more iterations for the Adagrad algorithm we could check if it gets the same accuracy or even better than the Adam optimizer.

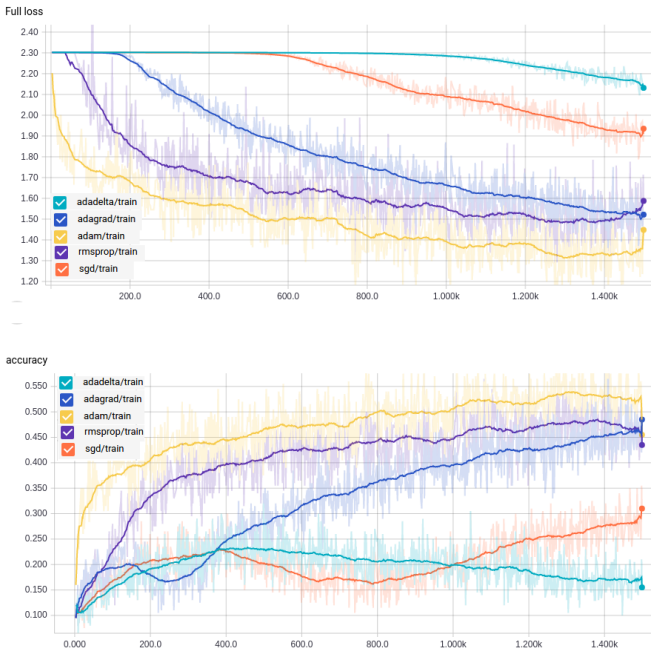

Figure 16: Loss(top) and accuracy(bottom) using different optimizers.

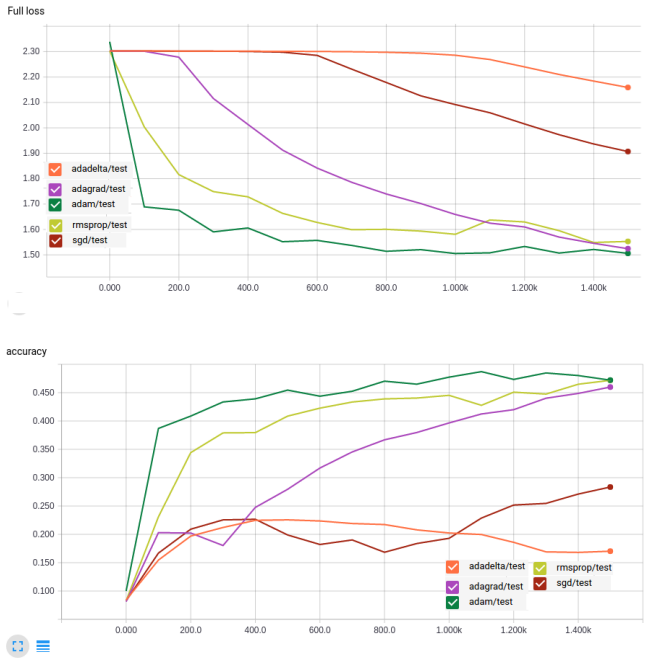

Figure 17: Loss(top) and accuracy(bottom) using different optimizers

Although this methods (except SGD) try to adapt the learning rate for the different dimensions during the optimization, they are still influenced by the initial value. Therefore, the used learning rate (0.001) could be more beneficial for some algorithms. For example, changing the learning rate to 0.1 in the AdaDelta algorithm improves the test accuracy to 0.5359, while with 0.001 we observed that it does not even reach the 0.25 of accuracy. Figure 18 shows the test accuracy, using more suitable learning rates for each algorithm. We observe that all the optimizers, except SGD, obtain similar performance, although Adam is still the faster in convergence rate.

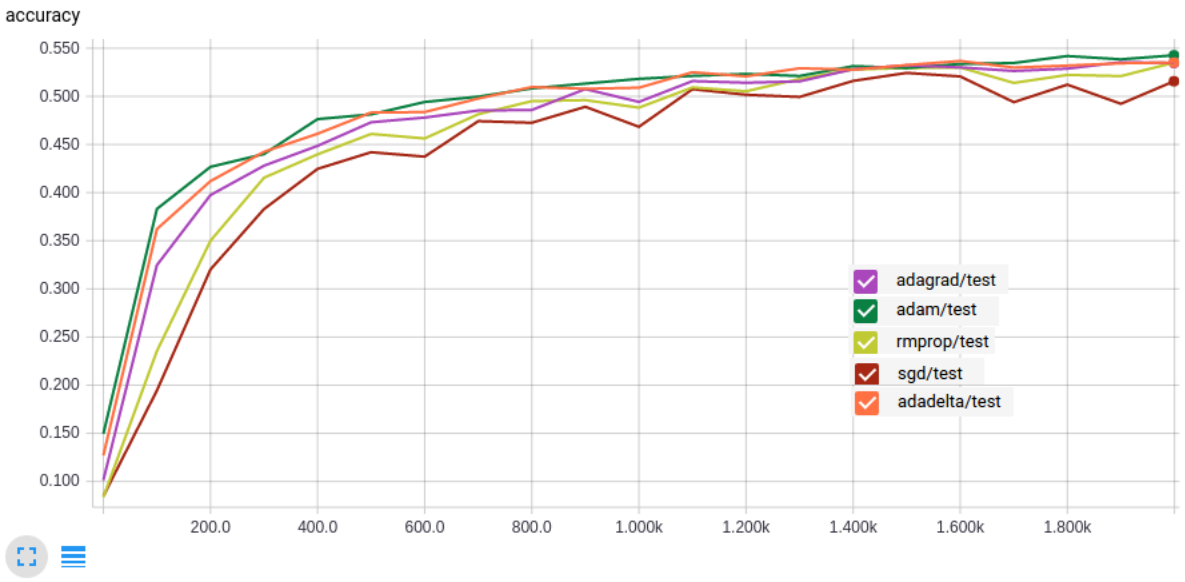

Figure 18: Test accuracy for different methods, with the best suitable learning rates. The network architecture was 2 hidden layers with 300 units each.

3.5 Experiment-5: Be creative

Although the execution of the experiments was considerably fast, there were many combinations with the hyperparameters and methods to run a complete grid search. Therefore, I followed the next strategy:

First, I tried how different architectures can impact in the performance. I first tried architectures with architectures with only 1 hidden layer, from 100 to 1000 nodes. Then I tried architectures with 2 hidden layers, using always more nodes in the first layer. I also tried architectures with 3 layers, but it did not improve the results. Then, selecting only some of the best architectures, I experimented using different activation functions. In this stage, I obtained the best results with ELU, obtaining an accuracy in the test set around 0.55.Finally, I tried to improve the performance trying different dropout probabilities and batch size, which lead to a final accuracy of 0.5726. I used the Adam as optimizer with learning rate 0.001, as it generally worked better in experiment 4. The parameters of the final model were the next:

- Learning rate: 0.0001
- L2 regularization with 0.01 regularization strength
- Weight initialization: Gaussian normal distribution with standard deviation 0.001
- Batch size: 400
- Number of iterations: 3000
- Dropout rate: 0.12
- Architecture: 2 hidden layers. First layer with 1500 units, second layer with 400 units.
- Activation function: ELU
- Optimizer: Adam

Ideally, the process of finding the best parameters should have been done in a validation set, and then, evaluate the final model in the test set. However, the code provided to read the dataset did not allow load a validation set without reduce the number of samples in the training set. Therefore, the test set was used to find the correct parameters, but another set would be necessary to evaluate the real accuracy of the model.

Figure 19 shows the loss and accuracy plots of the train and test sets. Observing the loss, the model seems to converge around the iteration 1000, actually the loss in the test set has a slight increase after some iterations. However, the accuracy still experiments a small improvement in the next iterations.

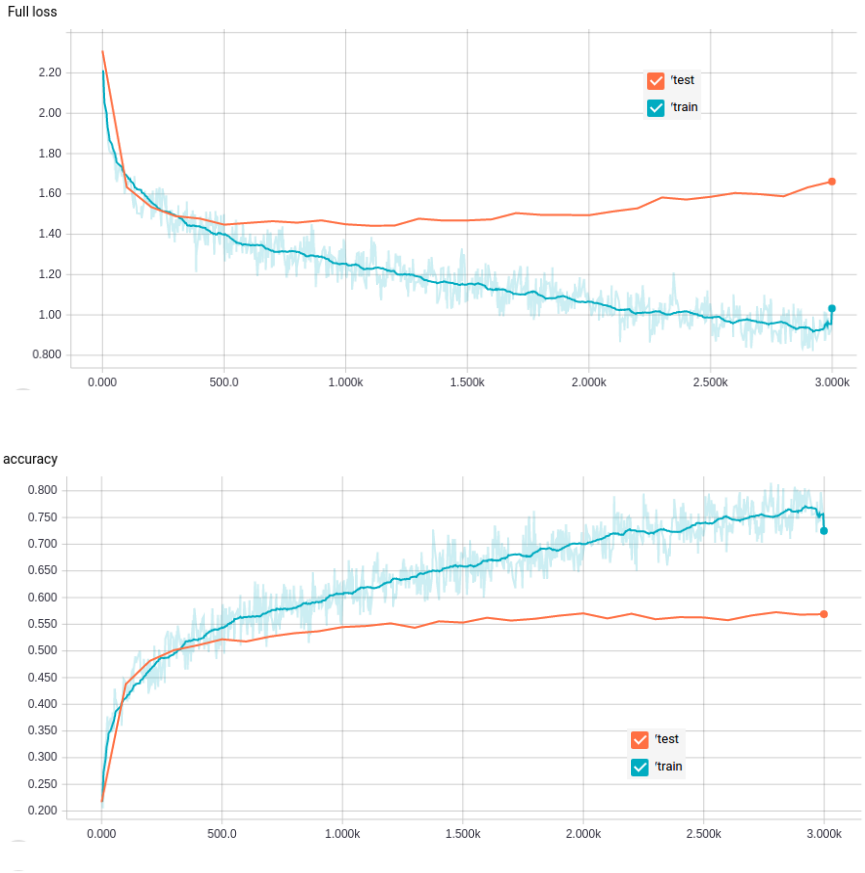

Figure 19: Loss(top) and accuracy(bottom) using different optimizers.

Figure 20 shows the confusion matrix for the test set. Some of the most common confusions that the model does is predict deer instead of birds, predict dogs instead of cats (and vice versa), and predict trucks instead of cars (and vice versa).

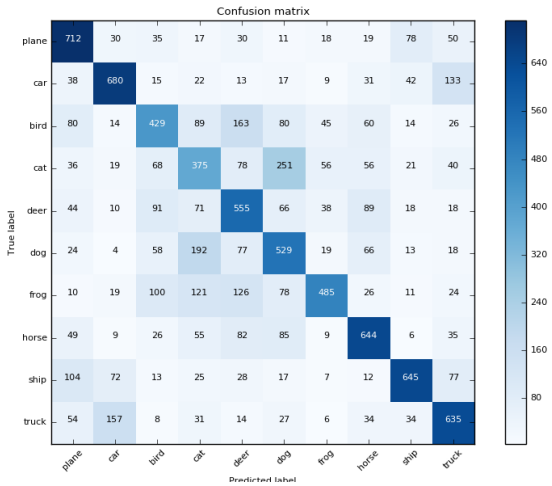

Figure 20: Confusion matrix for the best model found.

In figure 21 we can observe the top 5 wrong classifications for every class, where the model is more confident about the decision. We can observe how the model is prone to confuse (with a high confidence in its decision) trucks and cars, or horse and dogs.

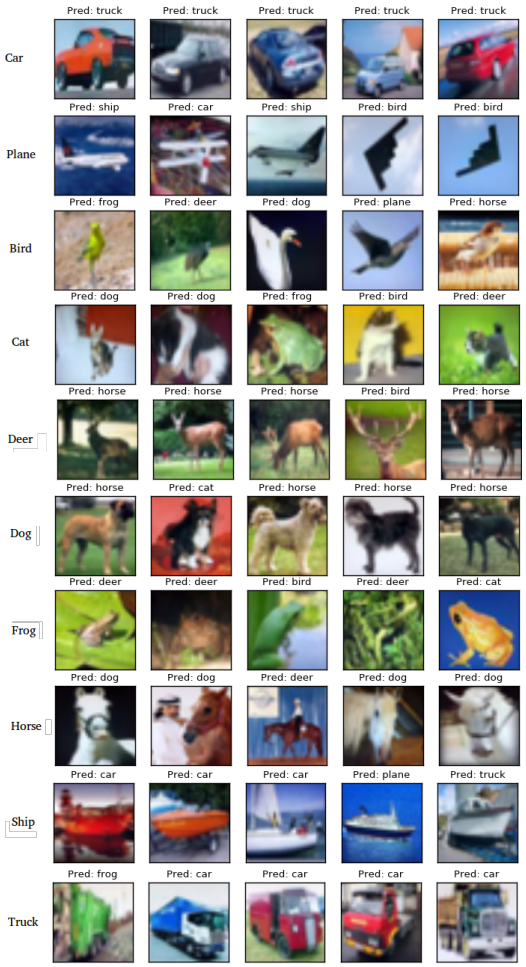

Figure 21: Wrong classifications when the model is more confident about the decision.

4 Conclusion

This assignment has been very useful as a introduction to TensorFlow. I have been able to experiment with some of its power features, as the automatic differentiation, or the capture of the data and visualization with TensorBoard, which has been very useful to have for insight how the different hyperparameters and methods affect to the performance in neural networks. The use of the TensorBoard has been important in order to improve the performance of the model in the CIFAR10 dataset.

References

[1] John Duchi , Elad Hazan , Yoram Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, The Journal of Machine Learning Research, 12, p.2121-2159, 2/1/2011

[2] Zeiler, M. D. (2012). *ADADELTA: An Adaptive Learning Rate Method.* CoRR, abs/1212.5701

[3] Kingma, D. P., & Ba, J. L. (2015). *Adam: a Method for Stochastic Optimization*. International Conference on Learning Representations, 1–13.