Report for the Deep Learning Course Assignment 2

Frederik Harder
frederik.harder@student.uva.nl

Abstract

This assignment explores supervised learning with multilayer-perceptrons in the tensorflow framework. It uses the cifar10 data-set, a 10 class image dataset of 32 by 32 color images for classification.

1 Task 1

Tensorflow is a machine learning framework under continuous development by Google, in which models are constructed by chaining together atomic operations on in graph. This allows a great amount of optimization and automation of steps, which would usually have to be done by the programmer, while still maintaining a great deal of generality. Because parts of the framework are quite unintuitive for beginners and documentation leaves a lot to be desired, we are answering a number of questions related to common pitfalls in this section.

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. [2 points]*

   All three represent tensors of numbers. constants, as the name suggests represent constants, which are assigned a static value in the code, which is not changed at runtime. Placeholders function much like constants, having a specified shape and type etc. but their actual values are assigned not while building the graph but with a feed during the session when executing a run() call. This can be used for supplying new training data batches or 'fresh' random numbers to a model on each iteration. Finally, variables represent the trainable parameters of a model. They must be initialized in a session and will be updated when running an optimizer function.

2. *Give two examples of how to initialize variables in TensorFlow. Provide an example code (snippet). [2 points]*

   Variables can be initialized both directly with a given value as in var1 =tf.Variable(0, dtype=tf.float32), with an initializer such as var2 = tf.get_variable(name='var2', shape=[], dtype=tf.float32, initializer=tf.constant_initializer) and tf.random_normal_initializer or they can be initialized based on another variable, as in var3 = tf.Variable(var1.initialized_value() + 1, name="var2"). Initialization by explicit value is likely a sub-functionality of initialization by another tensor of vice versa, but listed separately here for completeness.

3. *What is the difference between tf.shape(x) and x.get_shape() for a Tensor x? [2 points]*

   tf.shape(x) returns the a tensor containing the actual dimensions of x during a session. outside of a session, this information is unavailable. x.get_shape() returns a TensorShape object indicating the desired shape of x, as set by the declaration of x (or manually via x.set_shape()). Operations while building the graph rely on the latter, while operations during a session require use the former.

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? [2 points]*

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

Unlike <,>,<= and >=, the == operator is not overloaded in tensorflow. thus tf.constant(True) == True compares a tensor with a boolean and returns False. (in the overloaded cases, an argument is cast to a tensor, when compared to another tensor) tf.constant(True) == tf.constant(True) also returns False, as it compares two different tensor objects. During a session, comparison of boolean tensors can be achieved with tf.equal(), which returns a boolean tensor.

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. [2 points]*

   The tf.cond(c, f1, f2) function takes a boolean scalar tensor c and two functions f1, f2 with matching number of outputs and returns the return values of f1 if c is true and the return values of f2 if c is false.

   x = tf.constant(1)
   y = tf.constant(2)
   def f1(): return tf.mul(x, 3)
   def f2(): return tf.add(y, 4)
   r = tf.cond(tf.less(x, y), f1, f2)

6. *Name 3 TensorFlow components that you need in order to run computations in TensorFlow. [2 points]*

   At the very least, every computation in Tensorflow requires a graph, a tensor operation on that graph and a session, in which that operation is evaluated.

7. *What are variable scopes used for? Is there a difference between a variable scope and a name scope? [2 points]*

   The names of tensors defined within a scope are prefixed with the name of that scope, which makes it easier to keep track of large models. Variable scope has the added functionality that variables defined with get_variable() only search for pre-existing instances within the current variable scope. Name scope applies to all non-variable tensors.

8. *Can you freeze a given variable tensor such that it will maintain its value during, for instance, optimization? How? [2 points]*

   Yes. By default, optimizers only change variables in the GraphKeys.TRAINABLE_VARIABLES collection. Removing a variable from that collection thus excludes it from training.

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? [2 points]*

   The automatically computed differentials can be numerically unstable. providing your own differentials, while possible, can be a bit more effort than in other frameworks.

10. *Describe two ways to feed your own data into a TensorFlow graph. Shortly explain the pipelines. [2 points]*

    Data can be loaded and preprocessed in regular python and then passed to placeholders through a feed_dict in the session.run() call. Alternatively tensorflow provides file readers for a number of common formats such as CSV and raw text files, along with pre-processing functionalities, which directly supply the data in tensor form.

2 Task 2

The tensorflow implementation of the MLP we had already used in the previous assignment generated a test accuracy of around 0.44, as was the case last time. The graph of the model is displayed in figure 1. Accuracies and errors are shown in figures 2 and 3.

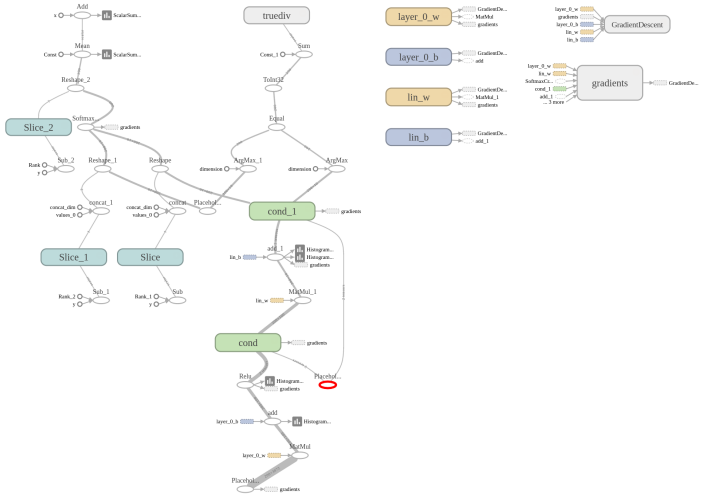

Figure 1: Graph of the model with default settings

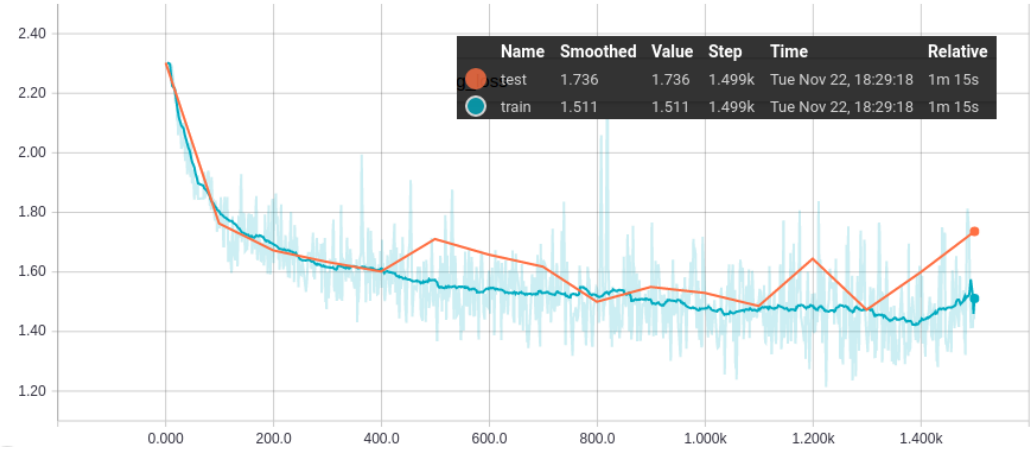

Figure 2: training and test error for default settings

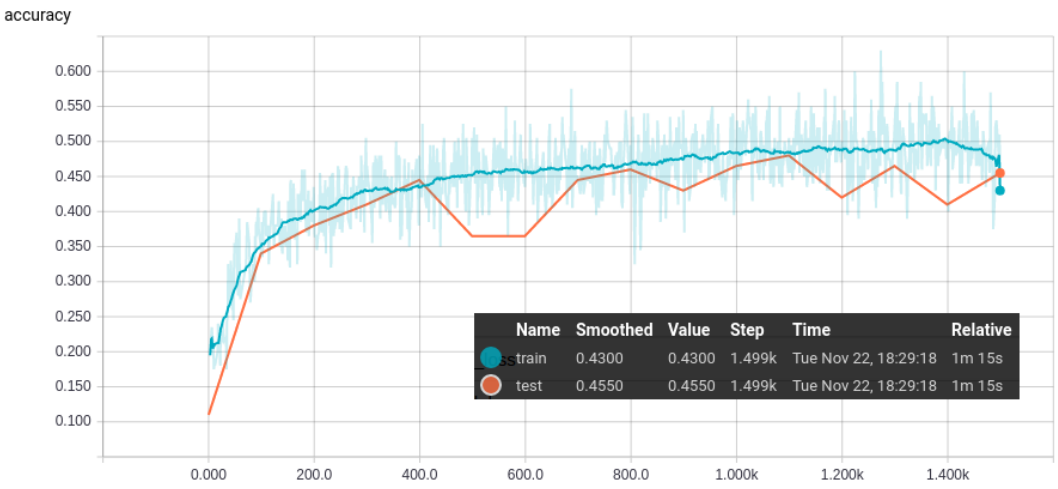

Figure 3: training and test accuracy for default settings

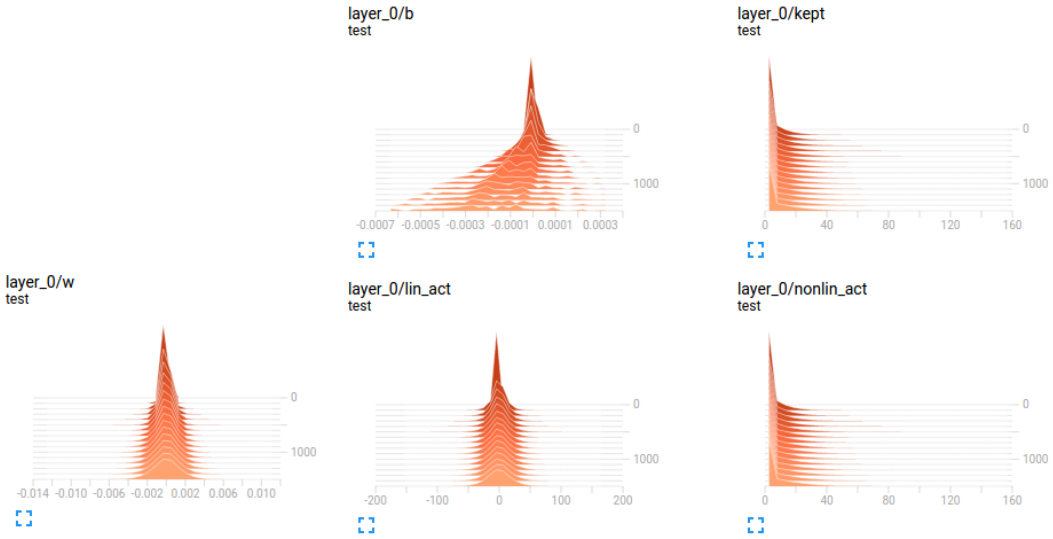

Figure 4: Trained weights of non-linear layer for default settings

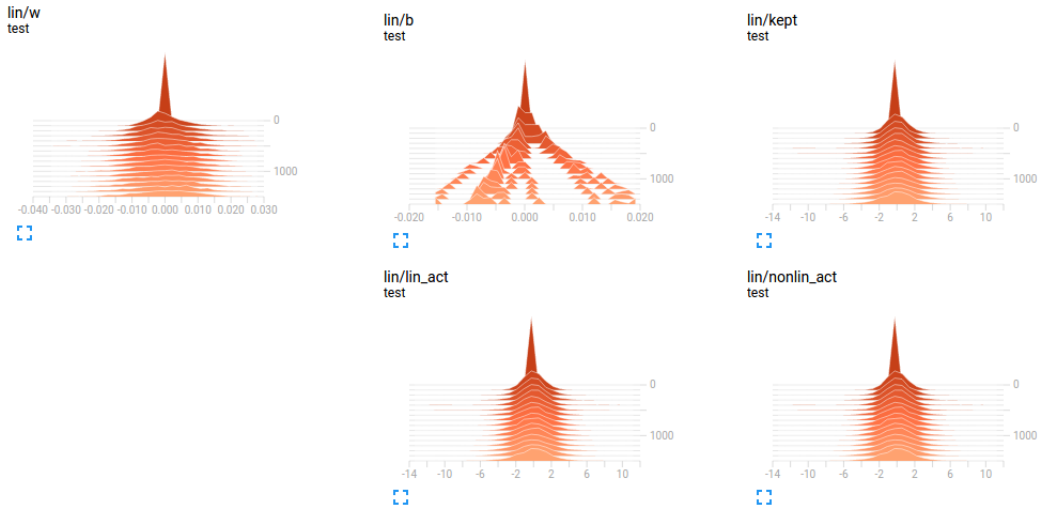

Figure 5: Trained weights of linear layer for default settings

3 Task 3

In this section, several model parameters have been tested over a range of values. unless listed otherwise the model ran with the settings shown in 6

3.1 Experiment-1: Weight initialization

We tested several choices of weight initialization, the final results of whicht are shown in table 7. Figures 8 and 9 show training errors and accuracies over time. What is notable here, is how badly xavier initilation performs. While the other settings only slightly differ in convergence speed, as can be seen in 8 and are quite similar in the end, Xavier initialization is notably worse and begins with an error which is magnitudes bigger (which is why it cannot be plotted here.

Learning rate	1e-3
Initializer	Normal (scale = 1e-4)
Batch size	200
Max Steps	1500
Dropout rate	0
Hidden layers	1 (100 units)
Regularizer	none
Activation function	ReLU
Optimizer	SGD

Figure 6: Default parameter settings

Initializer	Init. scale	Train Error	Train Accuracy	Test Error	Test Accuracy
Normal	1e-5	1.4477	0.490	1.4755	0.488
Normal	1e-4	1.4731	0.485	1.4708	0.490
Normal	1e-3	1.4897	0.440	1.4961	0.485
Normal	1e-2	1.5312	0.470	1.5636	0.456
Uniform	1e-5	1.4733	0.475	1.4828	0.484
Uniform	1e-4	1.4857	0.460	1.4694	0.489
Uniform	1e-3	1.4461	0.505	1.4664	0.495
Uniform	1e-2	1.4499	0.500	1.4865	0.479
Xavier	N/A	1.6280	0.430	1.7877	0.396

Figure 7: Performance achieved with different weight initializations

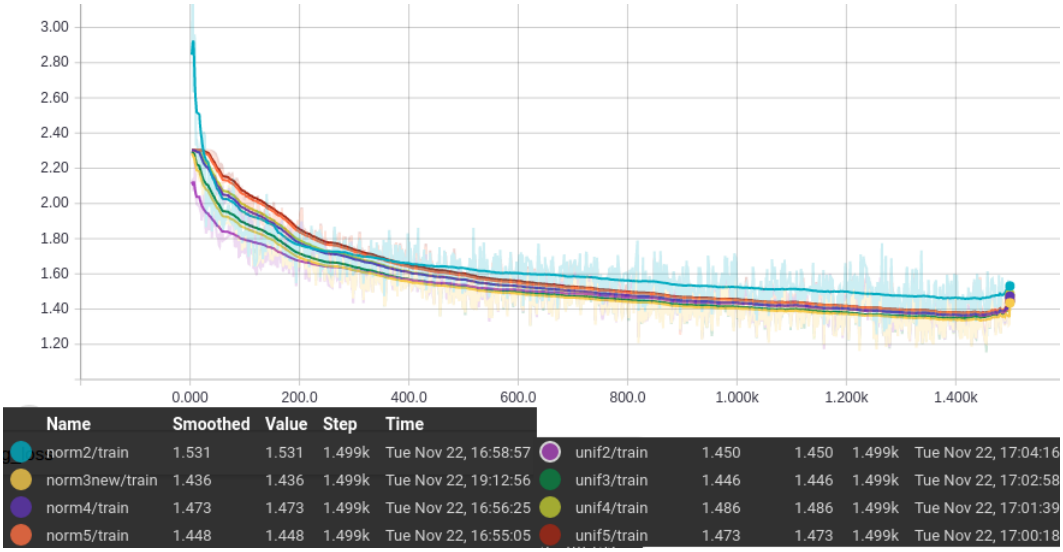

Figure 8: training error for tested initializations (excluding xavier)

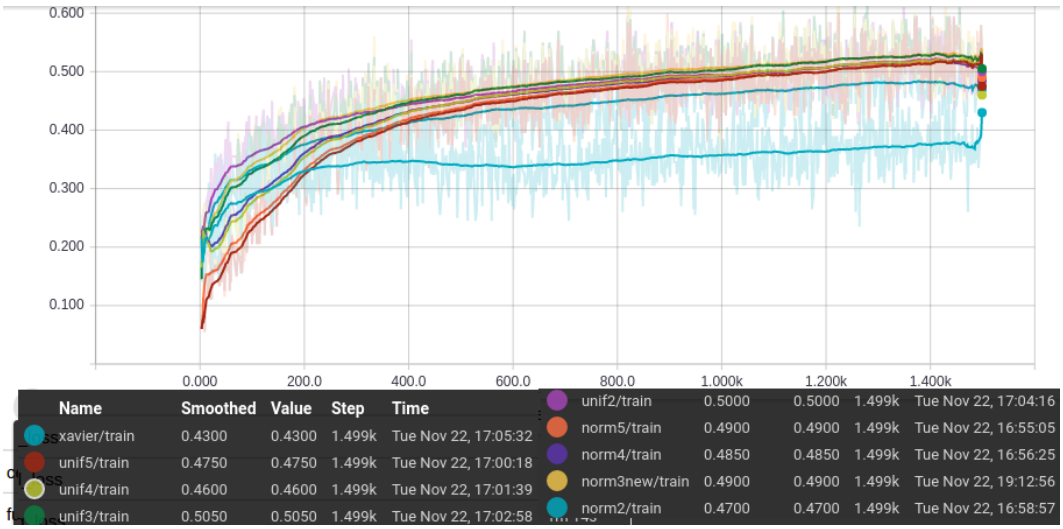

Figure 9: training accuracy for tested initializations

Initializer	Activation	Train Error	Train Accuracy	Test Error	Test Accuracy
Normal (1e-3)	tanh	1.7833	0.400	1.7922	0.380
Normal (1e-3)	ReLU	1.4424	0.505	1.4606	0.493
Xavier	tanh	1.9524	0.310	1.9916	0.311
Xavier	ReLU	2.0172	0.335	2.4649	0.330

Figure 10: Performance achieved with different combinations of nonlinearities and initializers

3.2 Experiment-2: Interaction between initialization and activation

In comparing the difference between Gaussian and Xavier initialization for both ReLUs and tanh activations, most of the results are expected, see figure 10. As established before, Xavier initialization and tanh activation fare worse than their alternatives. The drop-off in performance is smaller for tanh than for ReLUs, which may suggest, that tanh benefits more from the up-side of Xavier initialization. This observation matches that of Glorot and Bengio. In the paper in which introduces the method [1], the authors conclude, that tanh layers in particular benefit from Xavier initialization.

3.3 Experiment-3: Architecture

In this experiment, we drastically increase the size of the model from a single layer of 100 units to two layers of 300 units each. Overall this leads to a steep drop in performance which can be attributed largely to the now insufficient training time, especially given the use of a basic SGD optimizer. What can be observed here, however, is that Xavier initialization starts paying off. This makes sense of course, given that it has been designed to aid gradient propagation and should outperform more basic methods on deeper architectures.

Initializer	Activation	Train Error	Train Accuracy	Test Error	Test Accuracy
Normal (1e-3)	tanh	2.2698	0.135	2.2646	0.174
Normal (1e-3)	ReLU	1.9142	0.270	1.8910	0.287
Xavier	tanh	1.8811	0.360	1.9047	0.331
Xavier	ReLU	1.7431	0.390	2.1525	0.352

Figure 11: training accuracy for tested initializations

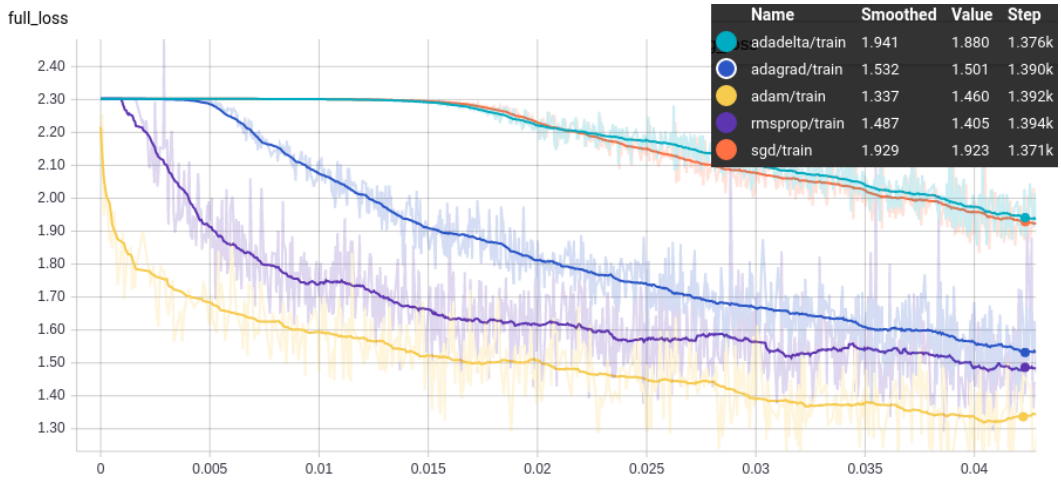

Figure 12: training error for tested optimizers

Optimizer	Train Error	Train Accuracy	Test Error	Test Accuracy
Adam	1.4662	0.505	1.4956	0.490
Adagrad	1.5963	0.425	1.5241	0.460
Adadelta	1.9364	0.245	1.9071	0.279
RMSProp	1.6739	0.410	1.5338	0.473
SGD	1.9183	0.265	1.8928	0.286

Figure 13: Performance for varying Optimizers

3.4 Experiment-4: Optimizers

Table 13 shows a comparison of different optimizers on the architecture introduced in the previous section (now with ReLUs and Gaussian initialization). The training error plotted in figure 12 further highlights the difference in convergence speed between the optimizers. This is not surprising, given that these optimizers were designed as explicit improvements of one another. The only optimizer with slower conversion than its predecessor is Adadelta. This is odd and I cannot explain it at this point.

3.5 Experiment-5: Be creative

I have not been able to find a Test accuracy score over 53% on my own. The closest I have gotten was 51.6% with the parameters specified below.

Learning rate	1e-3
Initializer	Normal (scale = 1e-3)
Batch size	200
Max Steps	3500
Dropout rate	0.5
Hidden layers	1 (1500 units)
Regularizer	l2 (strength = 0.1)
Activation function	ELU
Optimizer	adagrad

4 Conclusion

This assignment gave an overview of the wide array of components involved in training basic neural networks and how these are used within the tensorflow framework. While tensorflow's power as an architecure became very clear, large parts, are still hard to use because of its ongoing and sometimes radical development. Official documentation is sparse, and third-party contributions are often out of date. assets such as tensorboard are in essence great ideas but still lack basic features such as graph export functions. Much of the same could be said for my current understanding of neural networks. While the core concepts are very clear, and many of the available methods make sense, little intuition exists with regards to their interaction and building models ultimately remains a lot of guesswork.

References

[1] Glorot, X., & Bengio, Y. (2010, May). Understanding the difficulty of training deep feedforward neural networks. In Aistats (Vol. 9, pp. 249-256).