Report for the Deep Learning Course Assignment 2

Arthur Bražinskas

Abstract

In this document a study of the effect of different neural network architectures, configurations, and optimization methods on image classification(CIFAR10) is described. It was verified that tanh activation function is indeed prone to saturation that causes sub-optimal results, and that initialization plays an important role and has a direct affect to the performance. It has been discovered that even shallow neural networks are prone to falling into saturated regime. Finally, the best two hidden layer network configuration was ReLu with normal initialization.

1 Task 1

1. *Constants* are entities(symbolic) that can take a value only once, during the run-time their value can't be changed. *Variables* are entities that have to be initialised explicitly before we run operations that use their values. However, their values can be changed during the execution of the program. *Placeholders* - are entities that allow passing data via dictionary that can be used during the execution of the program.

   In the context of convolutional neural networks, constants can be the dimensions of matrices, or vocabulary size or a constant learning rate (if no learning rate optimisation are used). Variables are weights that are updated, and placeholders are data batches.

2. I will give two version depending on what the author means by initialization

```
with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())
  # or
  tf.initialize_all_variables().eval()
```

   here initialization = creation.

```
v = tf.get_variable("v", ...)
# or
v = tf.Variable(name="v", ...)
```

3. $tf.shape(x)$ returns the 1d integer tensor representing the shape of x. And $x.get_shape()$ returns the TensorShape of the input x. Technically, the former runs the graph in order to infer the dynamic shape of the tensor while the latter returns a static shape without execution of the graph.

4. Both evaluate to false, and therefore one should instead use $tf.equal()$ that performs evaluation during the execution of the graph.

5. The equivalences are : $tf.select$ and $tf.cond()$

```
pred = tf.placeholder(tf.bool)  # Can be any computed boolean exp
val_if_true = tf.constant(28.0)
val_if_false = tf.constant(12.0)
result = tf.select(pred, val_if_true, val_if_false)
```

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

```
sess = tf.Session()
print sess.run(result, feed_dict={pred: True})
# ==> 28.0
print sess.run(result, feed_dict={pred: False})
# ==> 12.0
```

or

```
a = tf.constant(5.0)
b = tf.constant(6.0)
c = tf.constant(7.0)
pred = tf.placeholder(tf.bool)
def if_true():
return tf.mul(a, b)
def if_false():
return tf.mul(b, c)
# Will be `tf.cond()` in the next release.
from tensorflow.python.ops import control_flow_ops
result = tf.cond(pred, if_true, if_false)
sess = tf.Session()
print sess.run(result, feed_dict={pred: True})
# ==> executes only (a x b)
print sess.run(result, feed_dict={pred: False})
# ==> executes only (b x c)
```

6. Tensors Operations(Nodes in a graph), sessions
7. The main difference between $name_scope$ and $variable_scope$ is that the former does not affect the prefixes of created variables using $tf.get_variable()$, which can be used for reusing variables that were created before. Therefore, is one want's to have a name prefix when variables are created/retieved via $tf.get_variable()$, he should use $variable_scope$ instead.
8. Yes there is, we can filter a collection of variable that we would like to optimise w.r.t. for example by scope. The following code does just that:

```
optimizer = tf.train.AdagradOptimzer(0.01)

first_train_vars = tf.get_collection(
tf.GraphKeys.TRAINABLE_VARIABLES,
                                     "scope/prefix/for/first/vars"
first_train_op = optimizer.minimize(cost, var_list=first_train_va

second_train_vars = tf.get_collection(
tf.GraphKeys.TRAINABLE_VARIABLES,
                                      "scope/prefix/for/second/va
second_train_op = optimizer.minimize(cost, var_list=second_train_

#This setup is beneficial for example for adversarial neural netw
```

   Alternatively one can use $stop_gradient()$ method.

9. Yes, TensorFlow performs automatic differentiation. It's adventigious because one can define an objective and the metchanism of the model and let Tensorflow produce gradients of the objective with respect to parameters.

   The numerical problems can be caused when we deal with matrix calculus. For example in objectives involving matrix inverses the error can grow disproportionately large, leading to e.g. scipy optimization routines failing to converge. The problem is likely to be caused the optimizer not exploiting known matrix factorizations such as Cholensky, LU, and other forms or reframing multiplications as more stable linear system solves[1].

[1] https://github.com/tensorflow/tensorflow/issues/4101

Tensorflow is not saving precomputed gradients of neural network parts (deltas) that can be reused during the backpropogation, which means that the proper implementation in the same environment would be faster and more efficient.

10. One way is by using placeholders and feed dictionaries, and second using transformation from numpy array like so :$a = tf.convert\_to\_tensor(a)$ the second method is less desired because it does not scale. The pipiline is relatively simple, tensorflow makes a dummy tensor for placeholder and when the graph is executed we pass our data into it via feed dictionary, which makes the value of the tensor that flow around the graph.

2 Task 2

In the following and the next tasks I've been using batch training accuracy and full test accuracy(as we did in the previous lab). In addition, for this task I've executed the default setup.

2.1 Plots of accuracy and loss curves for the train and test data

See fig. 1

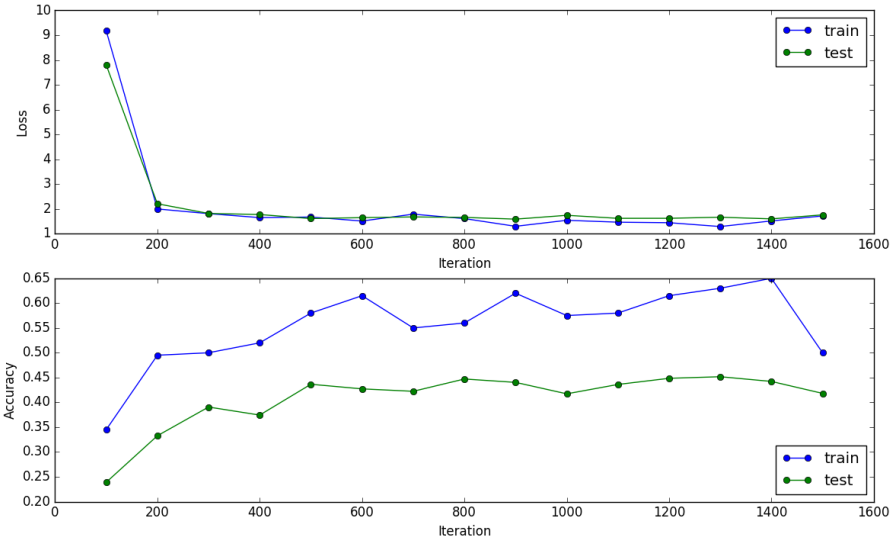


Figure 1: Accuracy and losses over iterations.

2.2 Graph of your model

See fig. 2

2.3 Histograms of the weights and biases of each layer

See fig. 3.

2.4 Histograms of logits

See fig. 4.

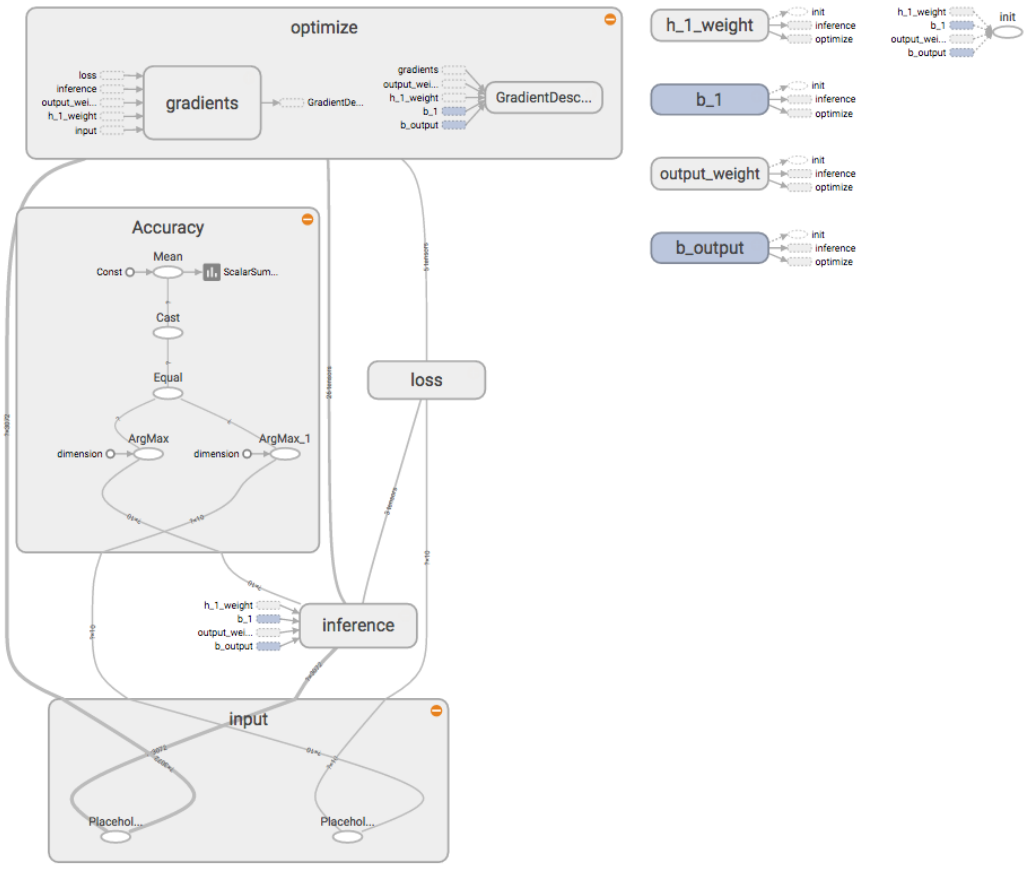


Figure 2: Graph of the model.

3 Task 3

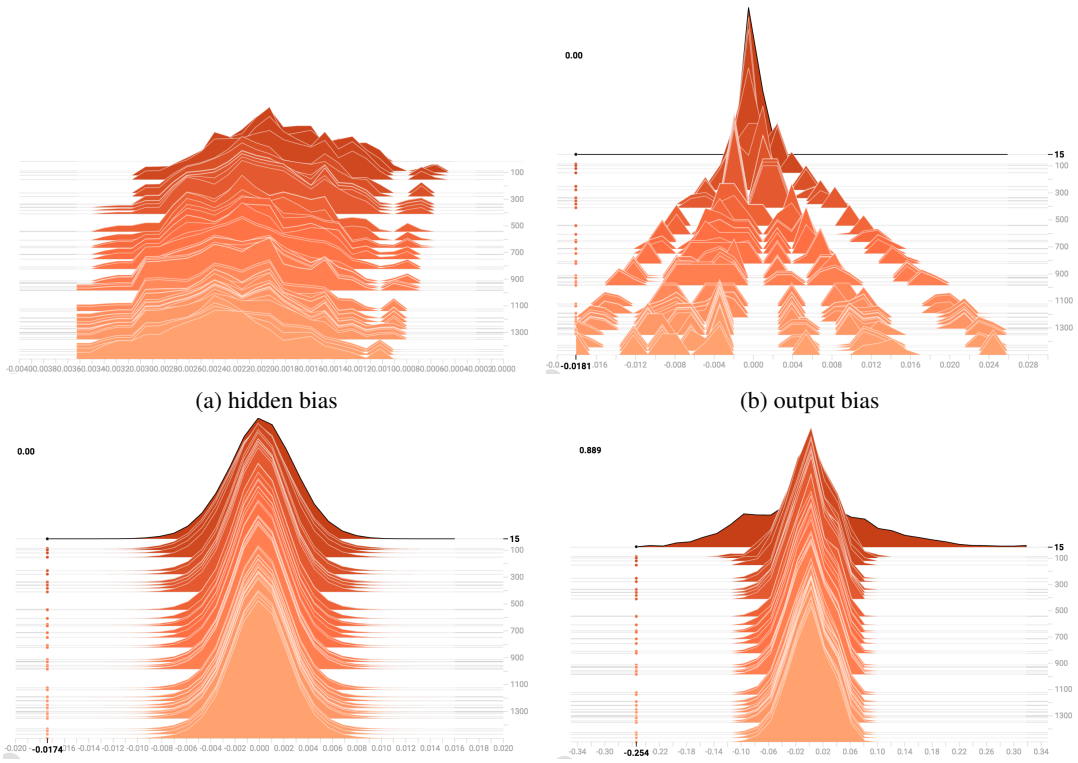
3.1 Experiment-1: Weight initialization

3.1.1 Gaussian initialization

The test accuracy after the experiments with different initialization noise is presented in fig. 7. From the plot one can see that the test accuracy is better for smaller initial noise. In addition, the noise has been observed to affect the convergence (how fast the training and test losses flatten out). Noises 1e-2, 1e-3 seem to converge at around iteration 500, while 1e-4 is closer to 600, and 1e-6 does not flatten-out completely as there are small fluctuations, see fig. **??**. For example, one can observe that training loss is still decreasing at iteration 1300, while test loss changes insignificantly after iteration 600. Loss and accuracy over iterations plots are provided in fig 5. In addition, one can observer that smaller noise leads not only to better test accuracy but also to better training accuracy, for example, 1e-2 noise produces training accuracy of around 0.7 while 1e-5 around 0.6.

3.1.2 Uniform initialization

When weights are initialized with uniform distribution it has been observed that for interval values ($[-val : +val]$) 1e-5, 1e-4, and 1e-3 noise does not affect the final test accuracy that much, it stays around 0.49, and drops to around 0.48 when 1e-2 is used. In addition, the convergence flattening seem very similar for the first 3 intervals - around iteration 500 - 600, and around 700 for the last interval. Also minor training loss fluctuations have been observed for all cases, while more variant ones were in the latter case(1e-2). The loss and accuracy plots over iterations are presented in fig. 6. One explanation for decrease when stronger noise is used the model converges to a point which is less optimal.



(a) hidden bias

(b) output bias

(c) hidden weights

(d) output weights

Figure 3: Histograms of parameters

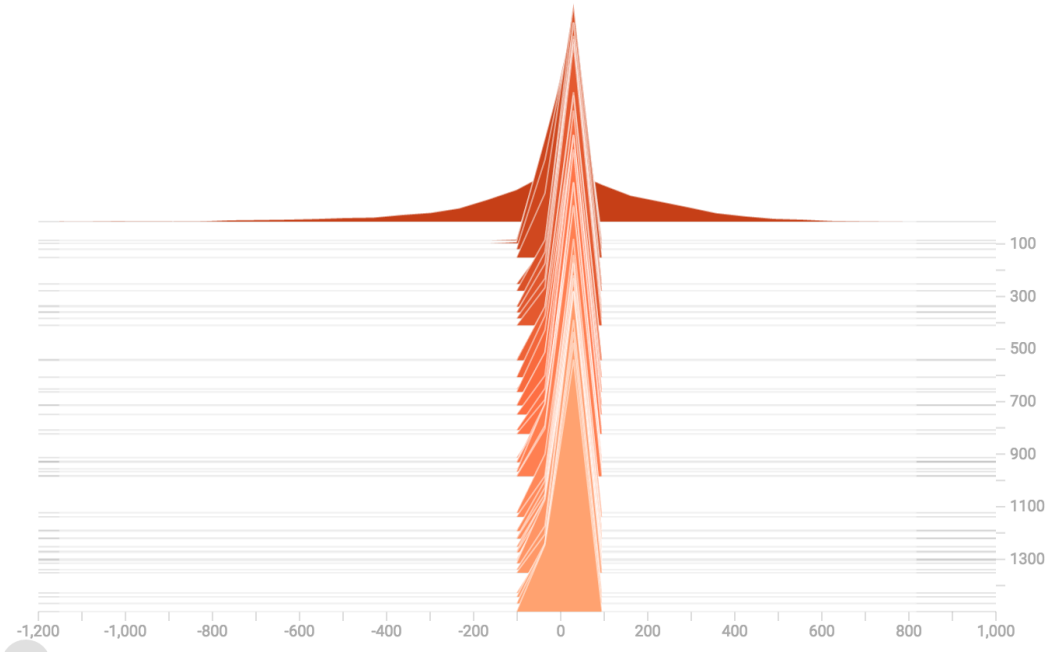
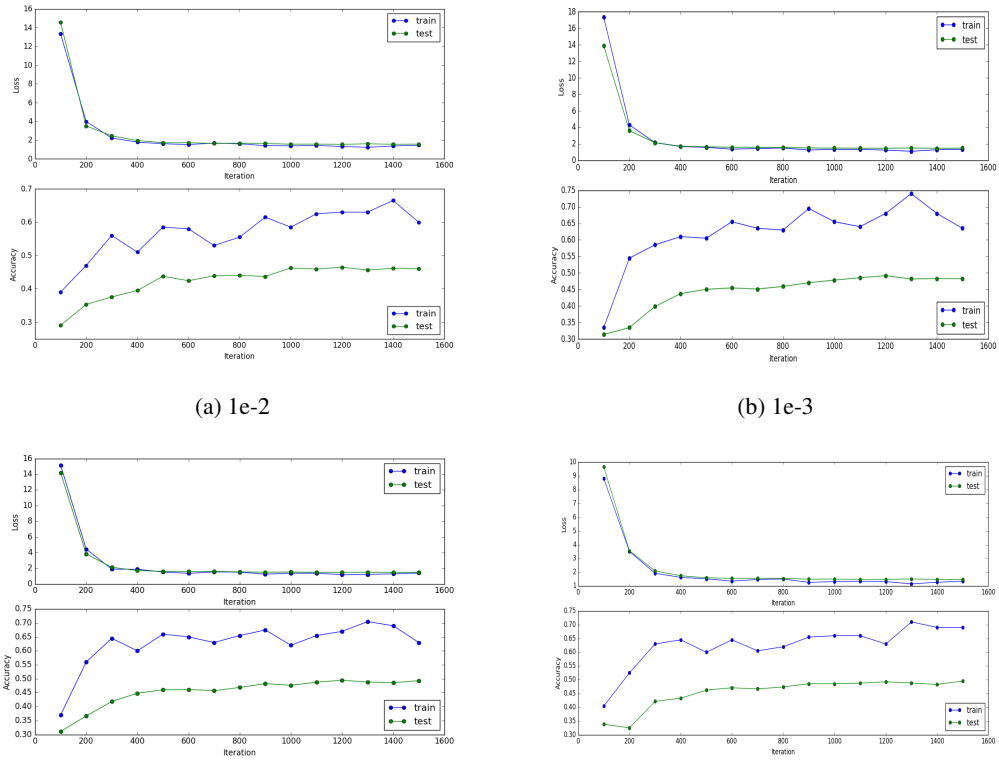


Figure 4: Logits



(a) 1e-2

(b) 1e-3

(c) 1e-5

(d) 1e-4

Figure 5: Loss and accuracy curves for different normal initializations.

3.1.3 Xavier

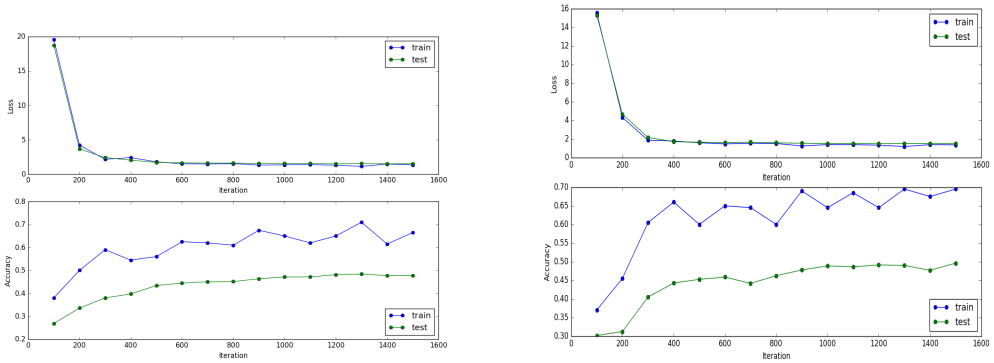
The Xavier weight initialization [1] has been theoretically derived for multilayer neural networks for maintaining activation variances and back-propogation variances as one moves up or down the network. However, our setup has only 1 hidden layer, which makes it questionable usage of the initialization method at the first place. Practically it has been observed a significant drop in performance when Xavier is used, it dropped to 0.34 test accuracy, in addition it produced higher loss. Both loss curves do not seem to flatten out, and some fluctuations are observed for both curved, see fig. 8. One reason of such poor performance can be explained by the fact that theoretical foundation of the method has can derived for multi-layer neural networks, and the method is not optimal for one hidden layer networks.

3.2 Experiment-2: Interaction between initialization and activation

First of all, as we observe from fig. 9, NN with tanh activation functions seem to have very stochastic training loss curve and it does not seem to converge. Second, the experiments with tanh activation produce worse results than with ReLu. Third, Xavier seems to produce worse results then normal initialization.

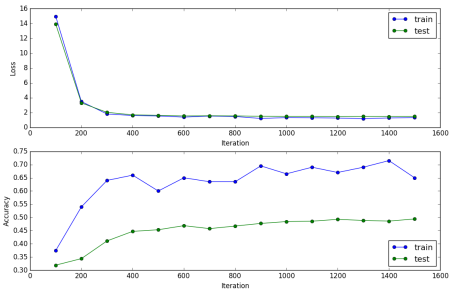
From the loss curve it does seem that the tanh NN does not converge and the test loss is continuously decreasing, so I've explored what will happen if the network will run longer (3k iterations). Unfortunately, it seems that the model could not converge even after 3k iterations, and the newly loss curve looked as chaotic as 9.

An explanation for tanh under-performance can be derived from the activation values of the hidden layer in fig. 10. One clearly can see that the network operates in the saturated regions where gradient is minimal, which makes training inefficient. Another important aspect that one could consider is the

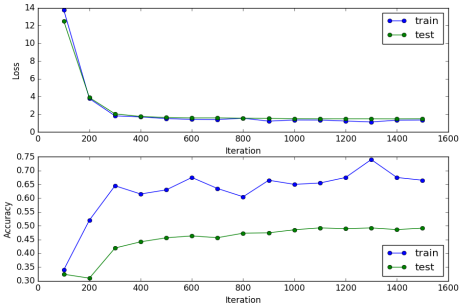


(a) 1e-2

(b) 1e-3



(c) 1e-5



(d) 1e-4

Figure 6: Loss and accuracy curves for different uniform initializations.

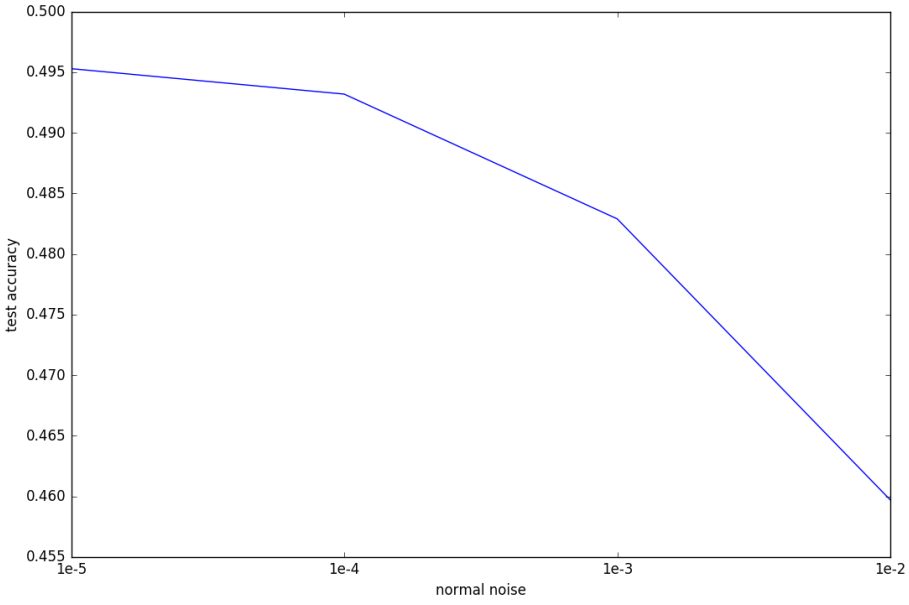


Figure 7: Test accuracy for different noise initializations

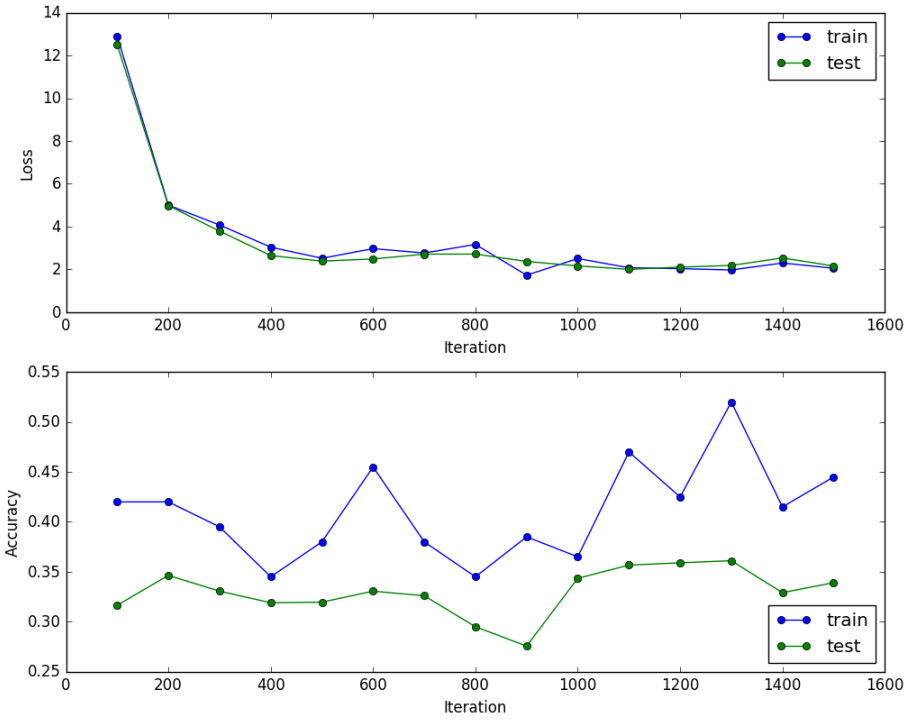


Figure 8: Training and test loss and accuracy under Xavier initialization.

flow of the gradient magnitudes that I was not able to obtain to better understand the procedure of learning. In addition, it's a strong argument for saturation in general.

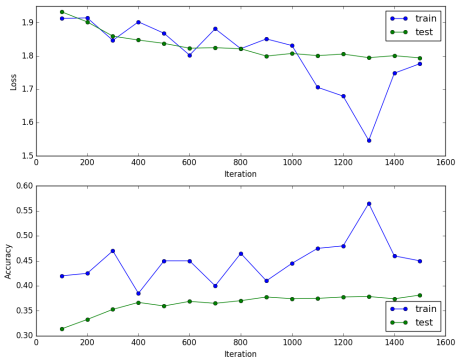
Finally, I note that Xavier initialization as explained in [1] has been researched and derived based on multilayer neural networks, and it's expected to work better for deeper ones, which is a subject of the following experiments.

3.3 Experiment-3: Architecture

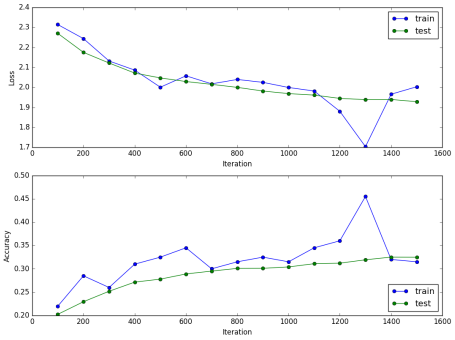
The accuracy and loss plots are presented in fig. 11. All setups produce better performance in comparison to 1 hidden layer setups, see table below. The reason is that neural networks can learn more complex features. However, One can observer that the similar trend persists: tanh is underperforming and xavier initialization. Again, using visualisation of the hidden layer's activations(in this case first) one in fig. 12, we can see that again our model operates in saturation regions. It's important to note that ReLU is less pron to saturation due to its shape, and this is what we observe.

It's unclear why Xavier initialization does not improve the performance in our situation, but a few extra experiments with deeper NN(more than 3 hidden layers) showed that the result is better with Xavier than with normal initializations.

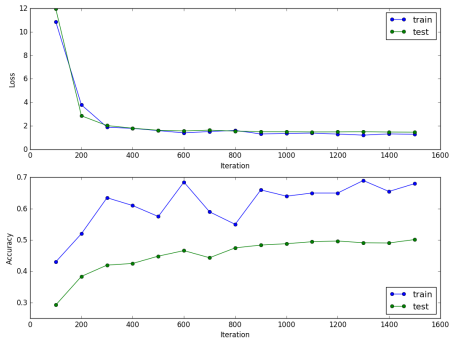
activation	setup	accuracy
tanh	normal, 1 hidden	0.3815
tanh	xavier, 1 hidden	0.3248
relu	normal, 1 hidden	0.5
relu	xavier, 1 hidden	0.3246
tanh	normal, 2 hidden	0.4218
tanh	xavier, 2 hidden	0.3289
relu	normal, 2 hidden	0.515
relu	xavier, 2 hidden	0.3927



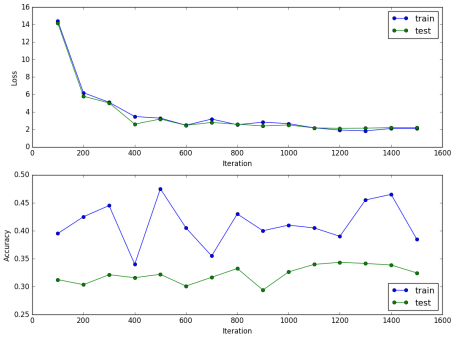
(a) tanh + normal init



(b) tanh + xavier



(c) relu + normal



(d) relu + xavier

Figure 9: Loss and accuracy over iterations for different 1 hidden layer setups.

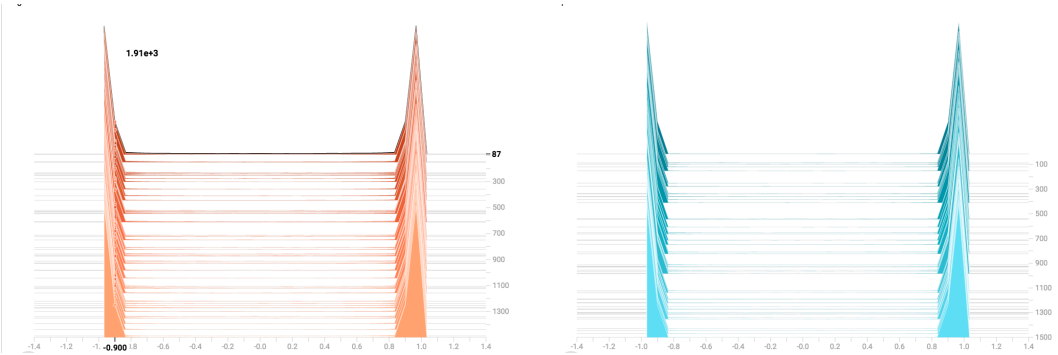
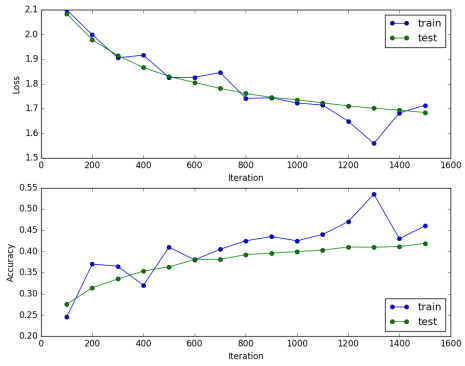
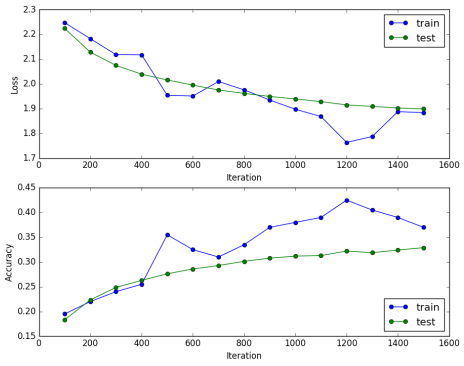


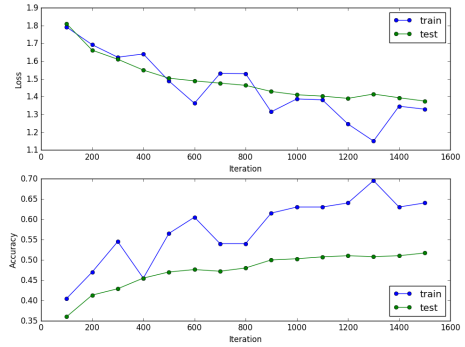
Figure 10: Orange normal and blue Xavier initializations of tanh NN.



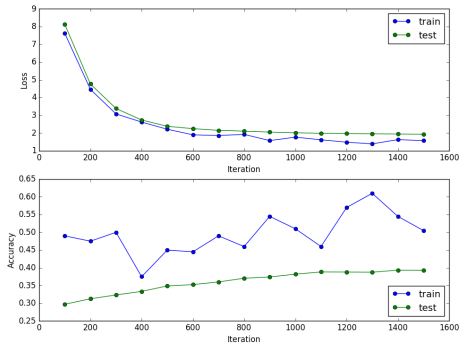
(a) tanh + normal init



(b) tanh + xavier



(c) relu + normal



(d) relu + xavier

Figure 11: Loss and accuracy over iterations for different 2 hidden layer setups

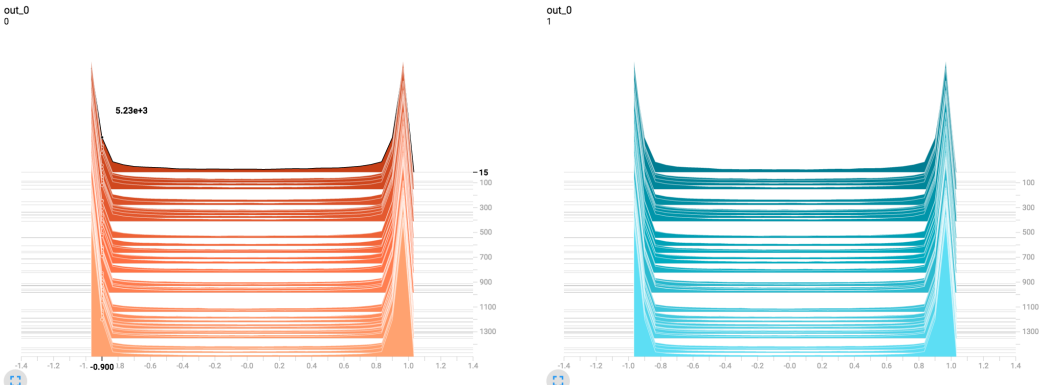
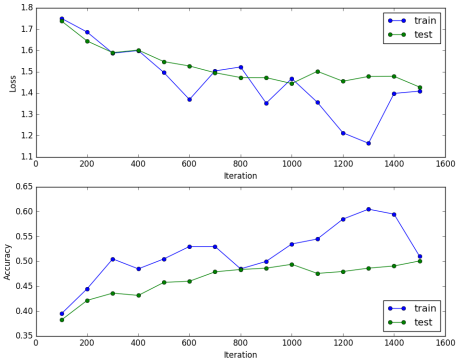


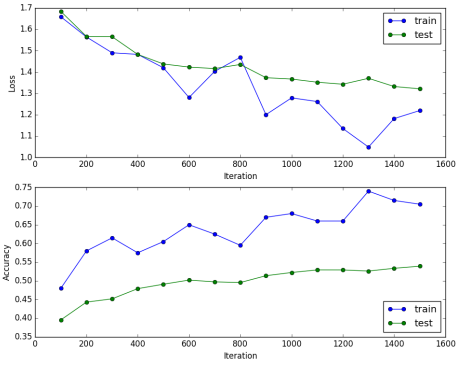
Figure 12: Orange normal and blue Xavier initializations of tanh NN.

3.4 Experiment-4: Optimizers

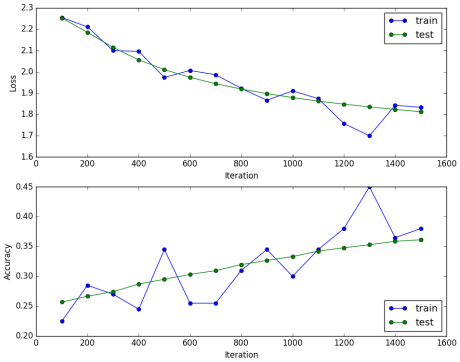
The experiments with different optimizers show that adadelta is the best optimizer for our setup, it yields around 0.54 accuracy, see fig. 13. Convergence-rate-wise adagrad, adam and pure sgd are similar and dominant methods. One explanation why adagrad improves the accuracy can be in the nature of the method itself, it remember previous gradients and reduces the learning rate proportionally such that it becomes smaller while we iterate, and that let us find better local points. However, some hyper-parameters tuning can produce very different results, as for example adaGrad



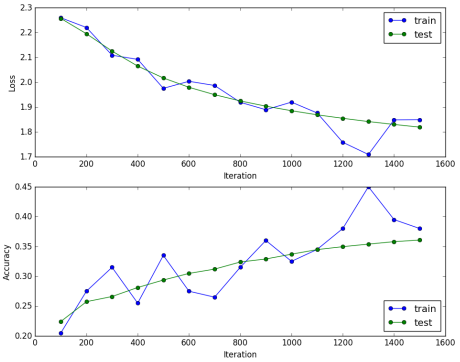
(a) Adam



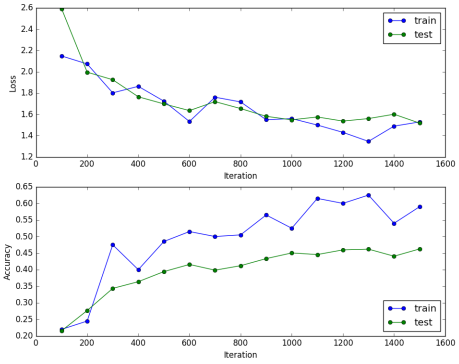
(b) adagrad



(c) adadelta



(d) rmsprop



(e) sgd

Figure 13: Loss and accuracy over iterations for 2 hidden layer ReLU and normal noise with different optimizer setups

has 1 hyper-parameters and Adadelta 2, therefore our results might not generalize to a different architecture or a setup.

3.5 Experiment-5: Be creative

It has been discovered that ELU activation function[2] with 2 hidden layers (both 400), normal initialization (1e-6), dropout rate 0.001, adagrad optimizer, and the rest default result produces around 0.55 accuracy(1 percent higher). ELU is a similar activation function to ReLU and it's know to

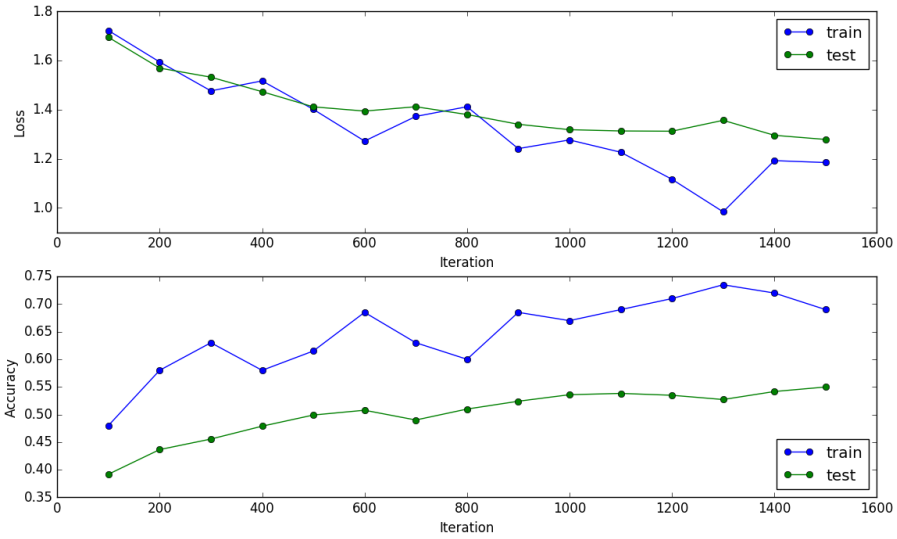


Figure 14: Loss and accuracy for the best setup.

```
[[683  49 104  48  54  26   9  34 123  53]
 [ 27 653  17  16  13  12  19  17  62 147]
 [ 35  18 411  65 146  70  94  43  11  11]
 [ 19  30 111 419  81 222 126  76  33  43]
 [ 32   6 125  59 453  63  74  95  20  14]
 [ 23  20  74 208  50 462  58  98  18  27]
 [ 19  12  71  78  95  46 583  13   4  15]
 [ 24  16  55  41  66  54   8 579  10  45]
 [ 95  50  15  28  26  22  10   7 670  58]
 [ 43 146  17  38  16  23  19  38  49 587]]
```

Figure 15: Confusion matrix for the best setup.

produce slightly better results and speedup up convergence, while preserve the benefits that ReLU has. The result is shown in fig. 14, and confusion matrix in 15.

The search for the setup was conduced in the following way: I've searched for similar to ReLu functions, and regularized the setup by including dropout to reduce the gap between training and test accuracy. In addition, I've followed the previously obtained insights that small normal noise produces better results. Finally, I've used the best optimization method from the previous setup.

Please note that the final sub-task has been assumed to be not mandatory :"You can also visualize 4-5 examples per class for which your model makes confident (wrong) decisions.", and thus omitted.

4 Conclusion

It was observed that initialization method has affect on the performance, and that even shallow neural networks can operate in saturated mode when tanh is used. In addition, it was observed that ReLU is more suitable as an activation function for the task. And that optimization methods plays an important role too. Unfortunately, it was not possible to confidently verify if Xavier initialization is also suitable for shallow (1-2 hidden layers) networks because we never used really for deep neural networks as in the original study. And the obtained results were suboptimal in comparison to normal initialization.

The TensorBoard has very neat features that allow visualisation of parameters that plays a key role in my research.

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[2] Anish Shah, Eashan Kadam, Hena Shah, and Sameer Shinde. Deep residual networks with exponential linear unit. *arXiv preprint arXiv:1604.04112*, 2016.