TDT4265 - Computer Vision and Deep Learning

Assignment 1

Thomas Aven, Lasse Eggen

January 31, 2019

Preface

For the mathematical proofs, we have included camera scans of our work, as we find them to be fairly readable. As such we got to spend the time we saved from not having to TeXify the work on the programming tasks instead. If the work is expected to be in LaTeX, please let us know for the next assignment.

Task 1.1

Over a single data point we get:

$$-E^n(w) = t^n \ln(y^n) + (1-t^n)\ln(1-y^n)$$

using $y^n = g_w^n$ and $\frac{\partial g_w^n}{\partial w_j} = x_j^n g_w^n (1-g_w^n)$

allows the usage of the chain rule to find $\frac{\partial -E^n(w)}{\partial w_j}$

$$\frac{\partial (t^n \ln(g_w^n))}{\partial w_j} = \frac{t^n x^n g_w^n (1-g_w^n)}{g_w^n}$$ → example chain rule

$$\frac{\partial ((1-t^n)\ln(1-g_w^n))}{\partial w_j} = \frac{-(1-t^n)}{1-g_w^n}(x^n g_w^n)(1-g_w^n)$$

$$\Rightarrow \frac{-\partial E^n(w)}{\partial w_j} = t^n x_j^n (1-g_w^n) - (1-t^n)(x_j^n g_w^n)$$

$$= t^n x_j^n - t^n x_j^n g_w^n - (x_j^n g_w^n - t^n x_j^n g_w^n)$$

$$= t^n x_j^n - t^n x_j^n g_w^n - x_j^n g_w^n + t^n x_j^n g_w^n$$

$$= t^n x_j^n - x_j^n g_w^n = (t^n - g_w^n) x_j^n$$

$$= \underline{(t^n - y^n) x_j^n} \quad \blacksquare$$

Task 1.2

$$y_k^n = \frac{e^{w_k^T x^n}}{\sum_{k'} e^{w_{k'}^T x^n}}$$

Using the quotient rule:

$$\frac{\partial y_k^n}{\partial w_j} = \frac{\frac{\partial}{\partial w_j} e^{w_k^T x^n}}{\left(\sum_{k'} e^{w_{k'}^T x^n}\right)^2} - \frac{e^{w_k^T x^n}}{\left(\sum_{k'} e^{w_{k'}^T x^n}\right)^2} \frac{\partial}{\partial w_j} \sum e^{w_k^T x^n}$$

Now: we split this into parts of $k=j$ and $k \neq j$, such that we differ between whether the weights are inputs to the wanted output or not:

$k=j$: $\Rightarrow \frac{x^n e^{w_k^T x^n}}{\sum_{k'} e^{w_{k'}^T x^n}} - \frac{e^{w_k^T x^n}}{\left(\sum_{k'} e^{w_{k'}^T x^n}\right)^2} \cdot x^n e^{w_j^T x^n}$ (mind this j!)

$$\Rightarrow \frac{\partial y_k^n}{\partial w_j} = x^n y_k^n - x^n y_k^n y_j = \underline{x^n y_k^n (1 - y_j^n)}$$

$k \neq j$: $0 - x^n y_k y_j = \frac{\partial y_k^n}{\partial w_j}$ (by the same method as above for $k=j$)

With the derivative of softmax, we may obtain the gradient of the loss function:

$$\frac{\partial E}{\partial w_{kj}} = -\sum_k^C t_k \ln(y_k^n) = -\sum_{k=1}^C \frac{t_k}{y_k^n} \frac{\partial y_k^n}{\partial w_{kj}}$$ (move the minus sign)

$$\frac{\partial y_k^n}{\partial w_{kj}} = \begin{cases} x^n y_k^n (1 - y_j^n), & k=j \\ -x^n y_k^n y_j^n, & k \neq j \end{cases} \Rightarrow y_k^n x^n (\delta_{kj} - y_j^n)$$ ⇐ Kronecker delta.

Thus: $-\frac{\partial E}{\partial w_{kj}} = \sum_{k=1}^C \frac{t_k}{y_k^n} y_k^n (\delta_{kj} - y_j^n)$

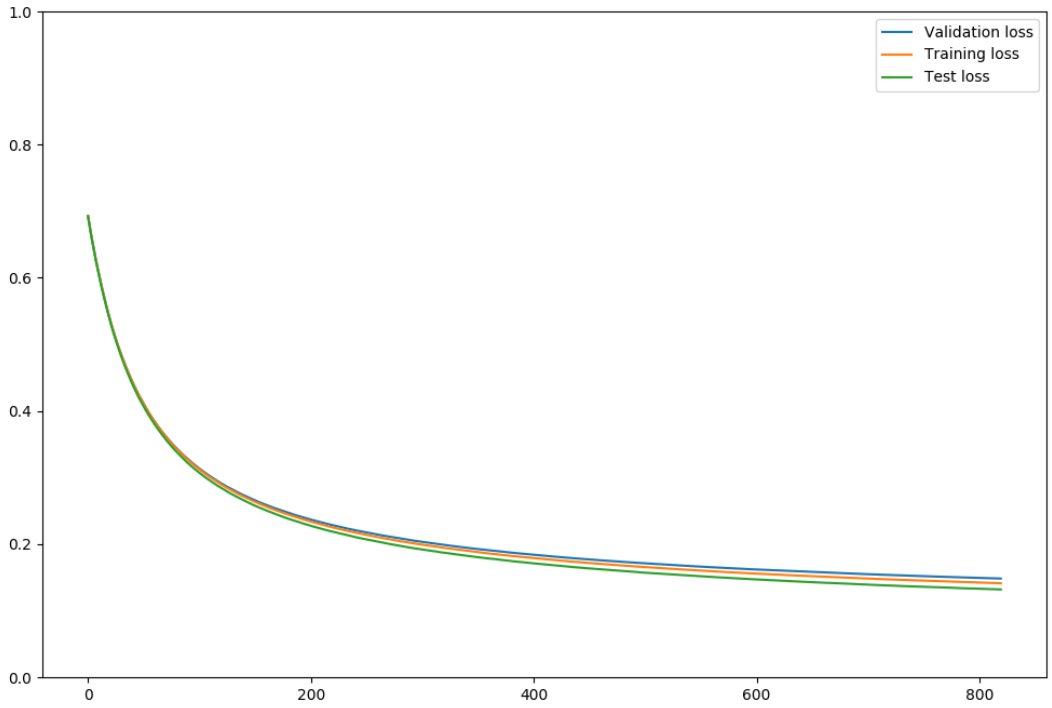
$$= \sum_{k=1}^C t_k x^n \delta_{ij} - \sum_{k=1}^C t_k x^n y_j = x^n (t_k - y_k^n)$$ ∎

⇐ we are using the weights w_{kj}, making $j=k$ here, not to be confused with the $j=k$ from the derivation)

2 - Logistic Regression through Gradient Descent

2.1a) For this task we concluded that a learning rate of 0.0005 worked well. We are also normalizing the input from the images with `X_train / 255`, as otherwise we would need a much lower learning rate to avoid the `z` in the sigmoid function to explode and cause floating point overflows in the exponential function.

As the training happens extremely fast, we have used the full training set and test set, i.e. all 70,000 images, which after pruning away unwanted categories amounts to about 12,000 images of 2s and 3s. Since the gradient descent makes for pretty quick convergence for this task, we computed the losses every 1/20th epoch for a total of 40 epochs.



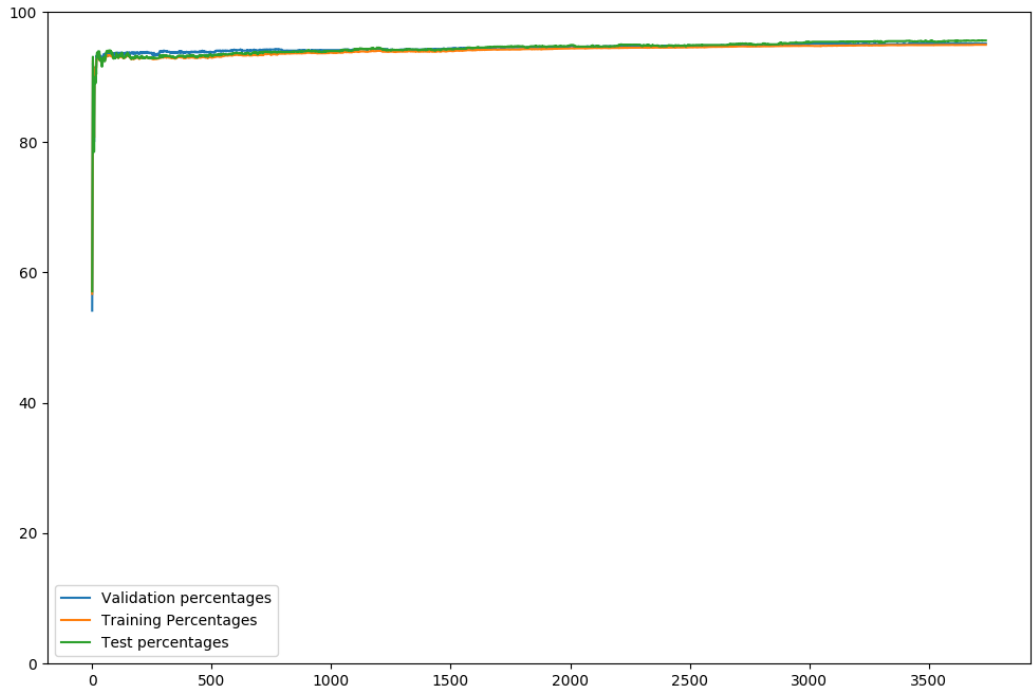
The loss decreases in a much expected manner for binary classification with logistic regression for this task. However, there is one point in particular that is of interest: the model seems to be underfitting on the training set. As we are using a *single-layered* network, we are lacking the needed complexity to perfectly classify the example training set as it grows bigger. A smaller training set, however, would be less likely to see the same underfitting.

For this particular run, there is a gap between the loss of the validation set and the test set. Some runs there were less of a difference, but it's still an interesting result. Having a validation set that is a good stand-in for would see similar loss performances

between the two. In this case, divergence may result between the two, given that the test set consists of digits written by a *different set of people* than the training set. We would still like to argue that this is not necessarily likely to cause problems, as people tend to write digits in pretty similar manners.

We have also implemented annealment of the learning rate and early stopping, as can be seen in the code. Our annealing method of exponentially decaying the learning rate did not provide any extraordinary results that were easy to spot. This is perhaps due to the fact that our network is so simple that there is little room to inch even further along basins during the gradient descent, even if the learning rate decays. We decided that early stopping should occur if we see an increase in validation loss four epochs in a row, which seemeed to work well.

2.1b) As the convergence happens rapidly, we plotted the percentages classified correctly over the sets after every single minibatch for ten epochs, giving exactly what one would expect.



Task 2.2 a)

$$J(w) = \xi(w) + \lambda C(w)$$

$$\frac{J(w)}{\partial w} = \frac{\xi(w)}{\partial(w)} + \frac{\lambda C(w)}{\partial w}$$

↳ given from previous task

$$\frac{\lambda C(w)}{\partial w} = \frac{\partial\left(\lambda \sum_{i,j} w_{i,j}^2\right)}{\partial w} = \underline{\lambda 2w} \quad \square$$

(The summation disappears, as we are differentiating over one specific weight, whilst the others remain constant).

I've also seen $\frac{\lambda}{2n} C(w)$ used, which gives:

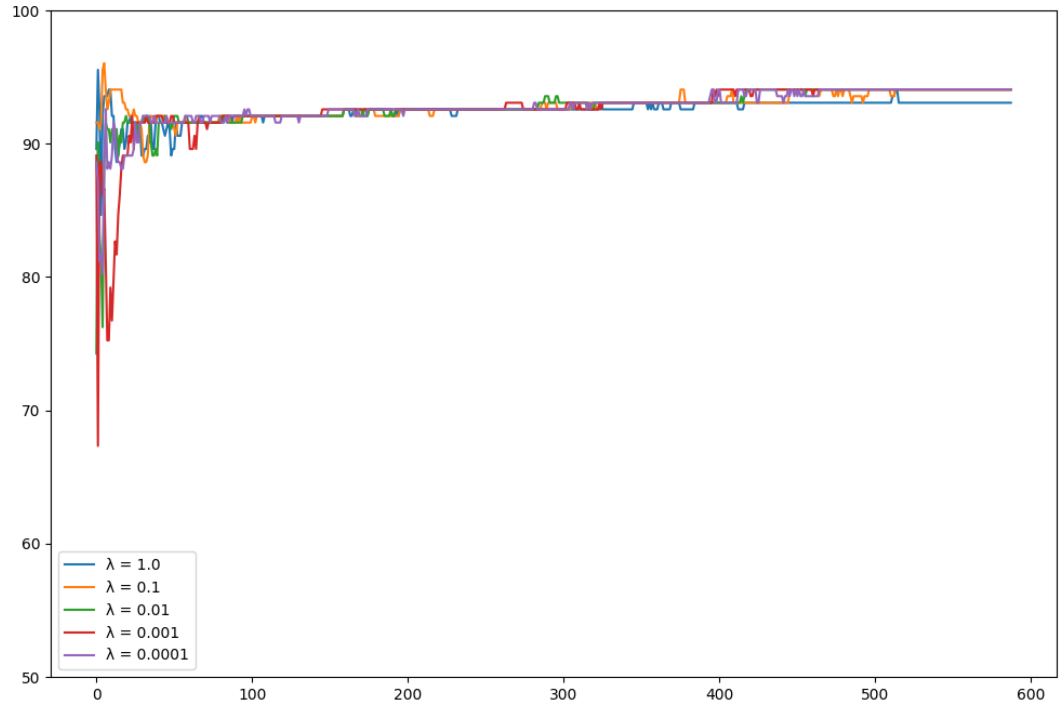
$$\frac{\partial \frac{\lambda}{2n} \xi w^2}{\partial w} = \underline{\frac{\lambda}{n} w}.$$

2.2b) As regularization is a method of preventing overfitting on training data, we decided to decrease the size of the training set for this task. However, we still did not see any occurrences of definite overfitting – we would see it from time to time, but not in general for every training session.

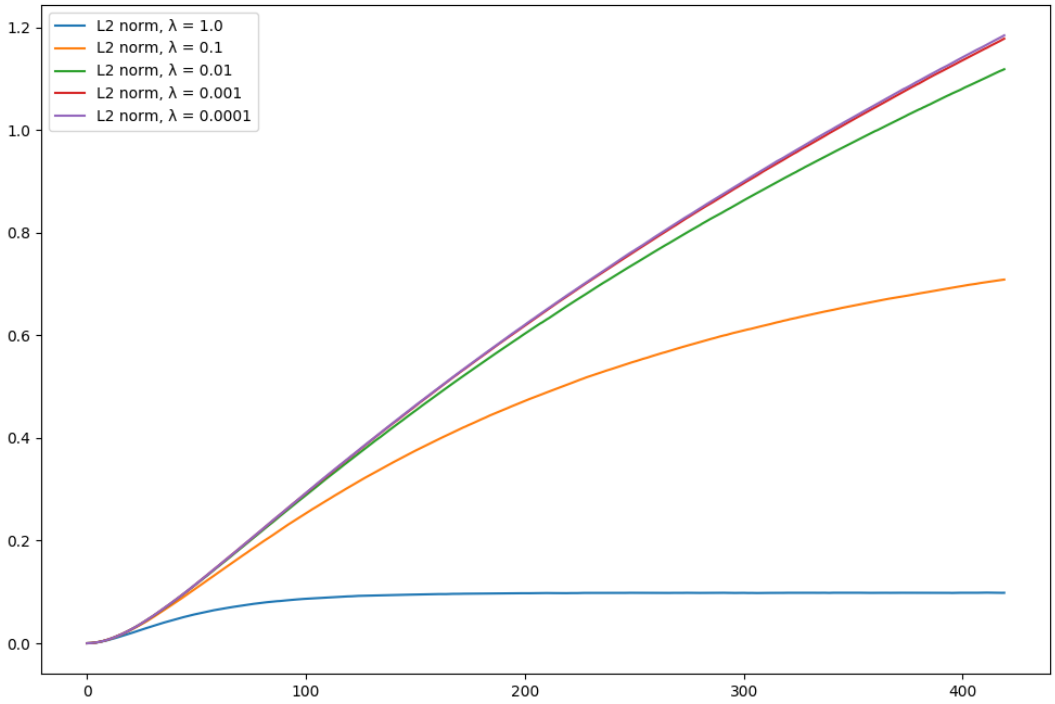
Even though we are pretty certain our method of punishing bigger weights is correct (mostly due to the fact that we studied the derivation of regularization in *Neural Networks and Deep Learning* by Nielsen):

```
reg = 2*lmbd * w
return w - lr*Ewdw - lr*reg
```

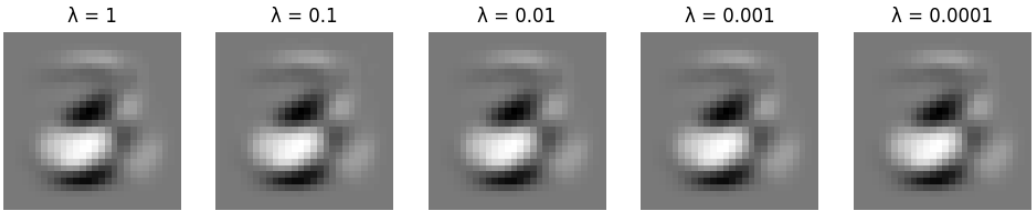
We are still struggling to see any major differences in the binary classifications of our network. We are attributing this to the fact that we did not find any consistently overfitting results. The only thing we definitely found to be prevalent across multiple runs was the fact that our network seems to perform better with less L2-regularization. Our reasoning for this is that, as mentioned, we are often seeing underfitting during training, and that enforcing weight penalties will thus impair training instead of improving results. The graph below consists of verifications every 1/20th epoch for 20 epochs.



2.2c) For a while we were thinking that we must have done something awfully wrong, but the following image very clearly illustrates that bigger weights are being punished quite hard with higher lambda values, which indicates that our previous analysis is correct, and that we are L2-regularizing as intended.



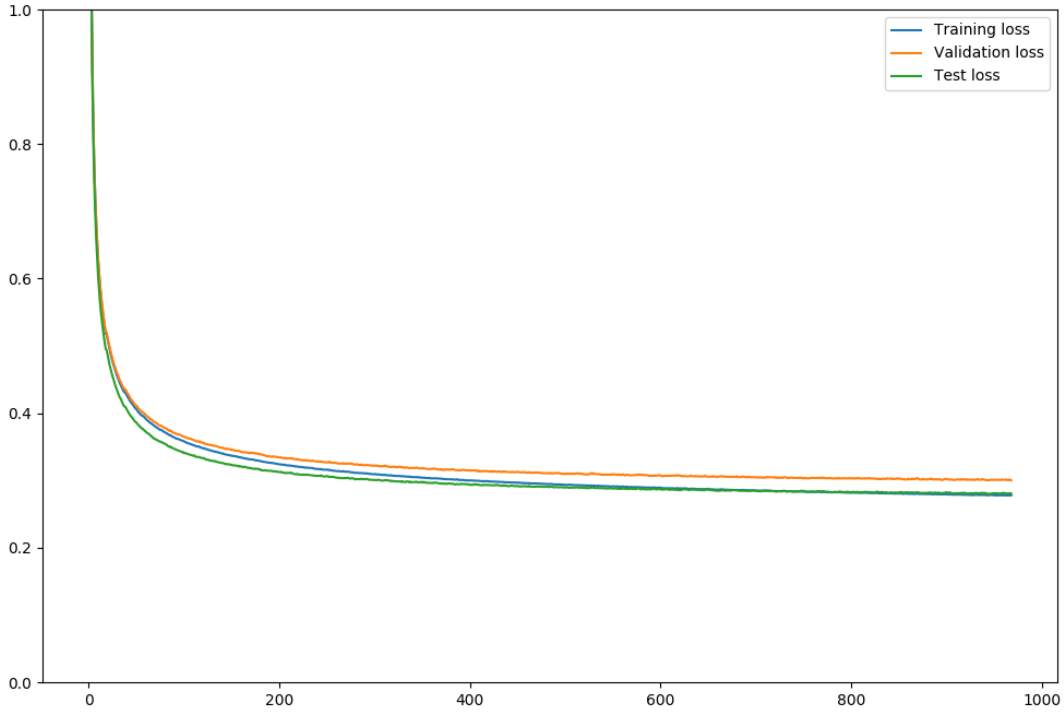
2.2d) As for observing differences between lambda usages, it is quite hard to see the difference (there are slight differences, but there is nothing that immediately screams «look at this difference – very interesting!»). Something that is interesting, however, is their appearances in general. As Matplotlib is heat mapping the weights for us, we can see that bigger weights are used to search for 2, whilst lower (e.g. negative) weights are used to find 3. This is very clear from the image below.



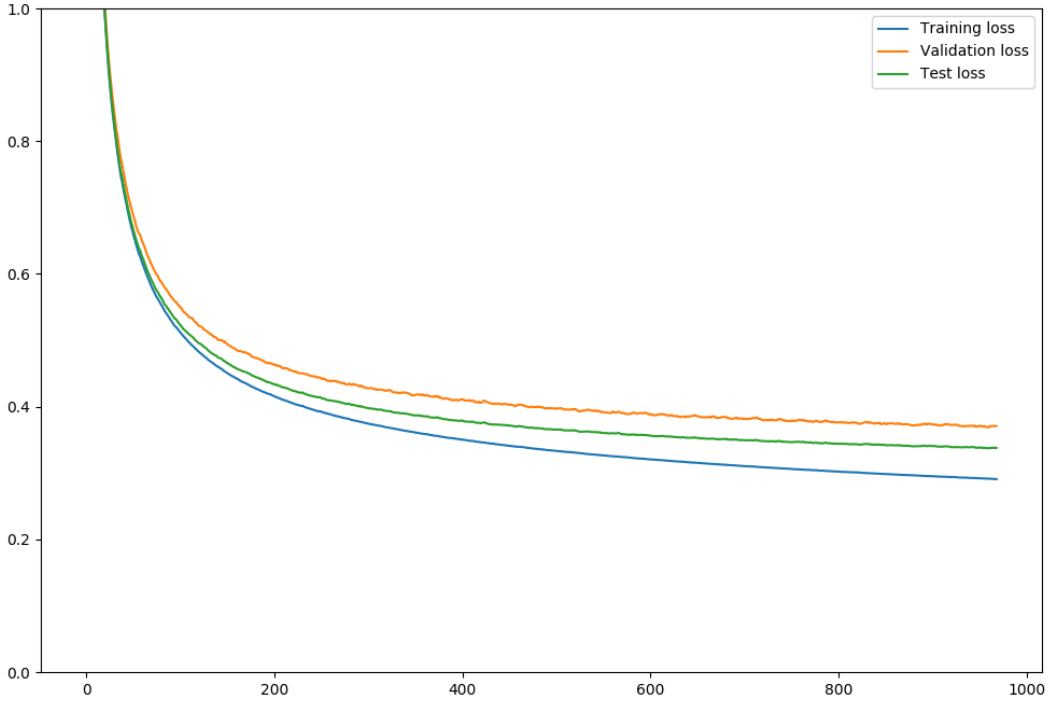
3 - Softmax Regression through Gradient Descent

Note: We didn't notice that softmax_loss was producing loss values that were quite a lot bigger than what can be seen in the lecture notes; we changed this later to give similar values, but did not bother re-doing the screen captures.

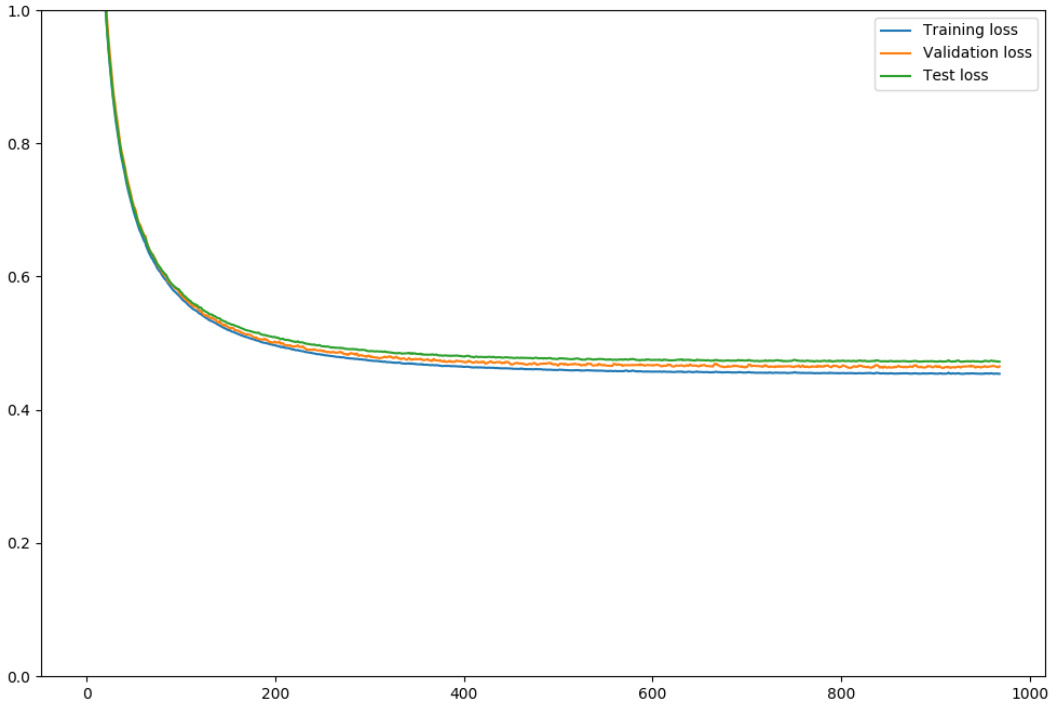
3a) As we don't need to make different plots for different lambda values, we decided to set the lambda value to 0 for now to focus on softmax regression by itself. For the task we used all 70,000 images, as a single-layered network like this is more than fast enough for it to become a nuisance (except perhaps when you are also evaluating the performance of the network 20 times per epoch).



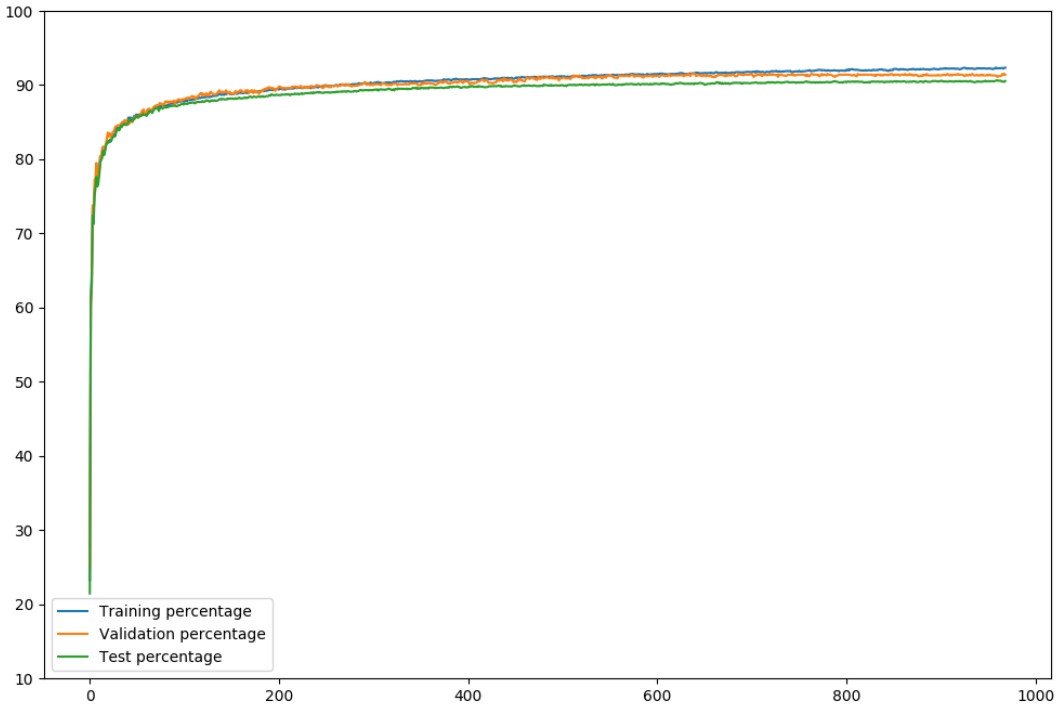
For the above image we are definitely seeing overfitting. If we limit the training set to only 10,000 images, it gets a lot worse:



Adding a regularization lambda of 0.01 produces the following output (note that the difference in the actual accuracy of on the validation set is not that different, but the training set is not generalizing as much):



Percentages for correct classifications are shown below (again without regularization here):


100
90
80
70
60
50
40
30
20
10
0
200
400
600
800
1000
Training percentage
Validation percentage
Test percentage