Deep Learning 2019 - Practical I

Yoni Schirris
MSc Artifical Intelligence
University of Amsterdam
yschirris@gmail.com

1 MLP backprop and NumPy implementation

1.1 Question 1.1 a) Analytical derivation of gradients

1.1.1 a

1
$x^{(N)} \in R^{1*n}, t \in R^{1*n}$
$L = -\sum_i t_i log(x_i^{(N)}) \in R^{1*1}$ From this we know that $\frac{\delta L}{\delta x^{(N)}} \in R^{1*n}$ where $\frac{\delta L}{\delta x_i^{(N)}} = -\frac{t_i}{x_i^{(N)}}$

Which is $-\frac{1}{x_i^{(N)}}$ for the class where $t_i = 1$ as the targets are one-hot encoded

2

As we will use the quotient rule, we can write our softmax as

$f = \frac{g}{h}$

so that

$f' = \frac{g' * h - h' * g}{h^2}$

where

$g = exp(\tilde{x}_i)$, $h = \sum_k exp(\tilde{x}_k)$, $g' = \frac{\delta exp(\tilde{x}_i)}{\delta \tilde{x}_j}$, $h' = \frac{\delta \sum_k exp(\tilde{x}_k)}{\delta \tilde{x}_j} = exp(\tilde{x}_j)$

For the softmax there are 2 cases:

Case 1: $i = j$

$g' = exp(\tilde{x}_j) = exp(\tilde{x}_i$

$f' = \frac{exp(\tilde{x}_j) * \sum_k exp(\tilde{x}_k) - exp(\tilde{x}_j) * exp(\tilde{x}_j)}{[\sum_k exp(\tilde{x}_k)]^2} = \frac{exp(\tilde{x}_j)}{\sum_k exp(\tilde{x}_k)} * \frac{\sum_k exp(\tilde{x}_k) - exp(\tilde{x}_j)}{\sum_k exp(\tilde{x}_k)} = x_j(1 - x_j)$

Case 2: $i \neq j$

$g' = 0$

$f' = \frac{0 - exp(\tilde{x}_i) * exp(\tilde{x}_j)}{[\sum_k exp(\tilde{x}_k)]^2} = -\frac{exp(\tilde{x}_j)}{\sum_k exp(\tilde{x}_k)} * \frac{exp(\tilde{x}_i)}{\sum_k exp(\tilde{x}_k)} = -x_i * x_j$

Putting this together we get

$\frac{\delta x}{\delta \tilde{x}} \in R^{n*n}$ where $\frac{\delta x_i}{\delta \tilde{x}_j} = x_i(\mathbb{1}(i = j) - x_j)$

3
$x \in R^n, \tilde{x} \in R^n, \frac{\delta x}{\delta \tilde{x}} \in R^(n * n)$

We know that a single output is computed by looking at the single input in a ReLu, so x_i is only dependent on $\tilde{x}_i$.

For each x_i we know that the derivative is $\frac{\delta max(0, \tilde{x}_i}{\delta \tilde{x}_i} = 1$ if $\tilde{x}_i > 0$ and 0 otherwise.

So we know that the resulting derivative looks like $\begin{bmatrix} \mathbb{1}(\tilde{x}_1 > 0) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbb{1}(\tilde{x}_n > 0) \end{bmatrix}$

4
$\tilde{x} \in R^{n_l}, x \in R^{n_{l-1}}, \frac{\delta \tilde{x}}{\delta x} \in R^{n_l * n_{l-1}}$

$\tilde{x} = W * x + b$ where $b \in R^{n_l}$ and $W \in R^{n_l * n_{l-1}}$

We know that

$\frac{\delta \tilde{x}_i}{\delta x_j} = \frac{\delta(W_{ij} * x_j + b_j)}{\delta x_j} = W_{ij}$

So we know that

$\frac{\delta \tilde{x}}{\delta x} = W$

5

We can solve this derivative by looking at several small steps and then putting those together.

We start with noting that, for a specific layer l:

$$\frac{\delta \tilde{x}^{(l)}}{\delta W^{(l)}} = \begin{bmatrix} \frac{\delta \tilde{x}_1^{(l)}}{\delta W^{(l)}} \\ \cdots \\ \frac{\delta \tilde{x}_m^{(l)}}{\delta W^{(l)}}) \end{bmatrix} \in R^{m*1}$$

where the derivative of each element q is given by

$$\frac{\delta \tilde{x}_q^{(l)}}{\delta W^{(l)}} = \begin{bmatrix} \frac{\delta \tilde{x}_q^{(l)}}{\delta W_{11}^{(l)}} & \cdots & \frac{\delta \tilde{x}_q^{(l)}}{\delta W_{1n}^{(l)}} \\ \vdots & \ddots & \vdots \\ \frac{\delta \tilde{x}_q^{(l)}}{\delta W_{m1}^{(l)}} & \cdots & \frac{\delta \tilde{x}_q^{(l)}}{\delta W_{mn}^{(l)}} \end{bmatrix}$$

Where each element of the matrix is simply given by

$$\frac{\delta \tilde{x}_q^{(l)}}{\delta W_{(ij)}^{(l)}} = \frac{\delta \sum_p W_{(kp)}^{(l-1)} * x_p^{(l-1)} + b_k^{(l)}}{\delta W_{ij}^{(l-1)}} = x_j$$

if $i = k$ and 0 otherwise

So that

$$\frac{\delta \tilde{x}_q^{(l)}}{\delta W^{(l)}} = \begin{bmatrix} 0 & \cdots & 0 \\ x_1^{(l-1)} & \cdots & x_n^{(l-1)} \\ 0 & \cdots & 0 \end{bmatrix} \in R^{m*n}$$

Where the non-zero values are on the q^{th} row

Putting this together gives us that

$$\frac{\delta \tilde{x}^{(l)}}{\delta W^{(l)}} = \begin{bmatrix} \begin{bmatrix} x_{11}^{(l-1)} & \cdots & x_{1n}^{(l-1)} \\ 0 & \cdots & 0 \\ 0 & \cdots & 0 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 0 & \cdots & 0 \\ x_{q1}^{(l-1)} & \cdots & x_{qn}^{(l-1)} \\ 0 & \cdots & 0 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 0 & \cdots & 0 \\ 0 & \cdots & 0 \\ x_{m1}^{(l-1)} & \cdots & x_{mn}^{(l-1)} \end{bmatrix} \end{bmatrix}$$

6

$\tilde{x} \in R^{n_l}, b \in R^{n_l}, \frac{\delta \tilde{x}}{\delta b} \in R^{n_l * n_l}$

where $\frac{\delta \tilde{x}}{\delta b} = \mathbf{1} \in R^{n_l * n_l}$

1.2 Question 1.1 b) Using the gradients of the modules calculate the gradients by propagating back

1

$$\frac{\delta L}{\delta x^{(n)}} \frac{\delta x^{(n)}}{\delta \tilde{x}^{(n)}} = \frac{\delta L}{\delta x^{(n)}} x_i(\mathbb{1}(i = j) - x_j)$$

$$= -t^T \cdot [ID - \mathbf{1}(x^{(n)})^T]$$
$$= ((x^{(n)})^T - t^T)$$

Where Id is the identity matrix and **1** is a column vector.

2

Assuming we already calculated the loss gradient from the layer before...

$$\frac{\delta L}{\delta x^{(l)}} \frac{\delta x^{(l)}}{\delta \tilde{x}^{(l)}} = \frac{\delta L}{\delta x^{(l)}} \begin{bmatrix} \mathbb{1}(\tilde{x}_1^{(l)} > 0) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbb{1}(\tilde{x}_n^{(l)} > 0) \end{bmatrix}$$

This simply sets the gradients to 0 if the output of the layer is smaller than 0, and sets the gradient to 1 if the output of the layer is greater than 0. This is exactly what the relu does!

3

From section 4 from the last question we get....

$$\frac{\delta L}{\delta \tilde{x}^{(l+1)}} \frac{\delta \tilde{x}^{(l+1)}}{\delta x^{(l+1)}} = \frac{\delta L}{\delta \tilde{x}^{(l+1)}} W^{(l+1)}$$

which we can't simplify any further. The previous loss gradient is simply multiplied with the weights.

4

$$\frac{\delta L}{\delta \tilde{x}^{(l)}} \frac{\delta \tilde{x}^{(l)}}{\delta W^{(l)}} =$$

mathematically very cumbersome to write, because of the many dimensionalities. But it makes a lot of sense when implementing it in Python.

It ends up being the previous gradient being $1 * d_l$ times a three-dimensional tensor of size $d_l * (d_l * d_{l-1}$. Essentially it creates a weighted sum of all the matrices, ending up being another $(d_l * d_{l-1}$ matrix holding this product. Although there are other methods, numpy provides a function called `einsum` that beautifully performs this operation.

5

From section 6 from the last question we get

$$\frac{\delta L}{\delta \tilde{x}^{(l)}} \frac{\delta \tilde{x}^{(l)}}{\delta b^{(l)}} = \frac{\delta L}{\delta \tilde{x}^{(l)}} \mathbf{1} = \frac{\delta L}{\delta \tilde{x}^{(l)}}$$

which remains unchanged

1.2.1 Question 1.1 c) Argue how the backpropagation equations derived above change if a batchsize of B != 1 is used.

When a larger batch size is used, we average the gradients of each input at the end.

In intermediate steps, the size of the gradients change, as they now hold the gradients for all input. The gradients will have an extra dimension, where each entry of this dimension holds all gradients of a single input.

1.3 Question 1.2 Implement a multi-layer perceptron using purely NumPy routines

See `train_mlp_numpy.py`, `mlp_numpy.py` and `modules.py` for the implementation. All code passes the provded unit tests. The final accuracy is around $45 - 46\%$. See Figure 1 for the accuracy over time and Figure 2 for the loss over time. We can see that with the initial parameters the curves

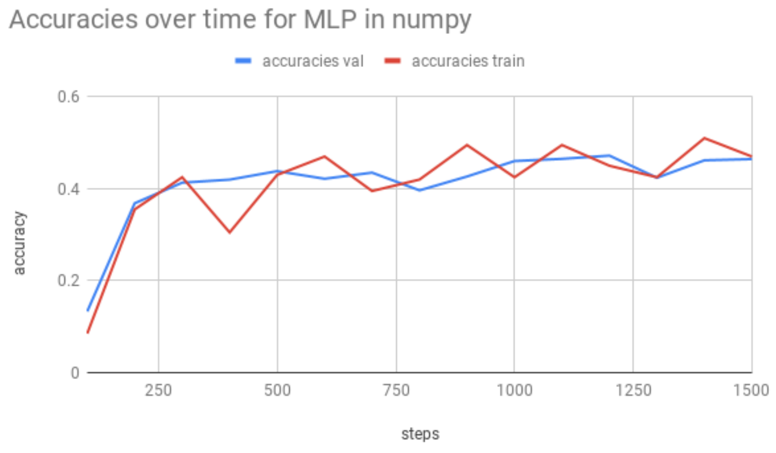

Figure 1: Accuracy of the multilayer perceptron implemented using only numpy methods over time with interpolated lines between the evaluations every 100 steps

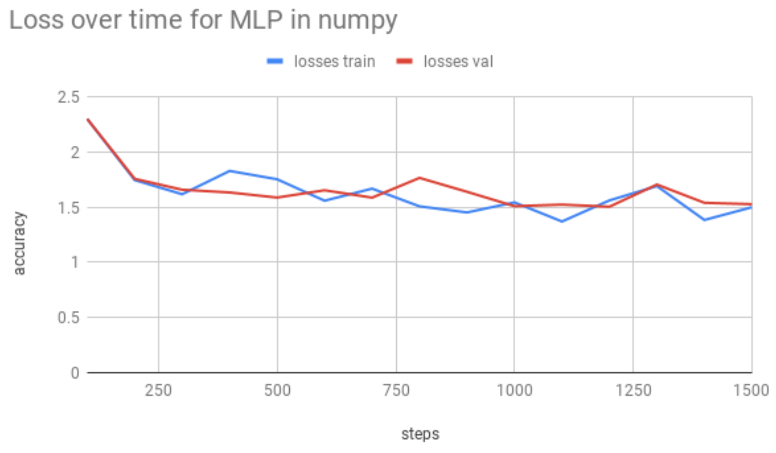

Figure 2: Loss of the multilayer perceptron using only numpy methods over time with interpolated lines between the evaluations every 100 steps

flatten out fairly quickly. It's interesting to note that the two curves are very close to each other, and that there's barely any overfitting happening in 1500 runs.

Note that if you want to use several layers, one has to increase the standard deviation in the `__init__` as due to a numerical instability the network won't train if you don't, and it will stick at an accuracy of 0.1.

2 Question 2 Pytorch MLP

See `mlp_pytorch.py` and `train_mlp_pytorch.py` for the implementation. It passes all unit tests.

I performed several experiments:

- Adam vs SGD
- Different values of momentum for SGD
- Different learning rates
- Different values of weight decay (regularization)

- Different number of steps
- Different number and size of linear layers
- Different batch sizes

Doing a grid search over all these parameters without deploying it on SurfSara and using the GPU power would require a very long time, so I went through trial and errors:

Firstly, I tried to find the best possible result using SGD by testing several combinations of weight decay and momentum, while keeping the rest of the parameters equal.

I then compared this to the best result I got with the Adam optimizer. The results were rooting for the adam optimizer.

Next became an interplay of number and size of layers, batch size, learning rate, and weight decay, as these all influence the amount of training, but often lead to overfitting. More layers requires a higher learning rate to reduce the vanishing gradient problem, but a higher learning rate would often overfit very quickly and would converge too quickly at a low accuracy. More stable learning happens when increasing batch size, but takes longer to train. When it would convert too quickly at a low accuracy and overfitting was clear, I would increase the weight decay and decrease the learning rate.

Going through a near infinite loop here, I ended up with the following parameters:

- optimizer=adam
- lr=3e-4
- weight decay = 2e-3
- batch size = 400
- layers = 600, 600, 200
- steps = 3000
- eval frequency = 100

Which can be run with `python train_mlp_pytorch.py --optimizer="adam" --regularizer=2e-3 --learning_rate=3e-4 --dnn_hidden_units="600,600,200" --batch_size=400 --max_steps=3000 --eval_freq=100`

The results are shown in Figure 3 and Figure 4 for the accuracy and loss, respectively.

In these graphs we can see that in the first 200 steps the loss decreases dramatically for both the training and test set. The accuracy increases accordingly.

After this point, we notice that the accuracy on the test set slowly flattens, while the accuracy on the train set continues to increase.

The pytorch implementation with deeper layers is clearly much better at memorizing the training set than the numpy implementation, and increases its accuracy while increasing the loss on this set as well – therefore it's not actually becoming (much) better, as we can see in the training accuracy and loss.

We notice that the training loss gradually continues to decrease, while the accuracy increases slightly. After 3000 iterations I noticed that it wouldn't increase much further as the loss curve flattens out.

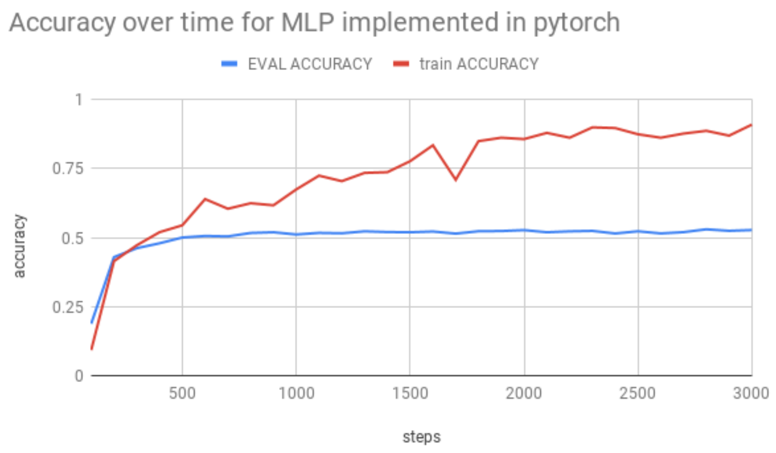

Figure 3: Accuracy of the multilayer perceptron implemented in pytorch over time with interpolated lines between the evaluations every 100 steps

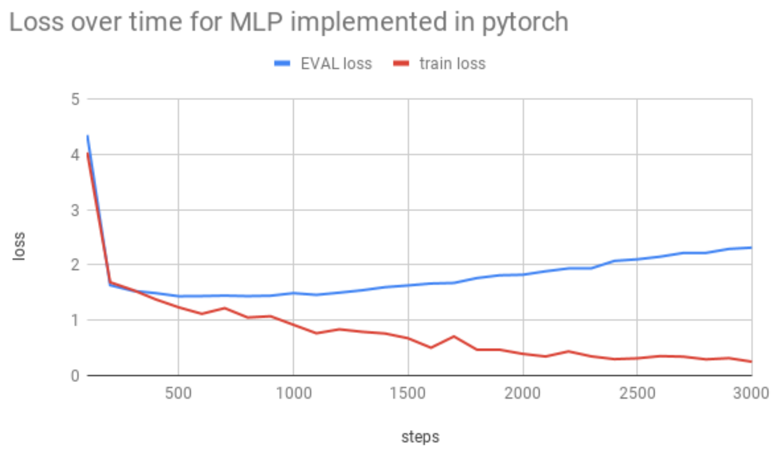

Figure 4: Loss of the multilayer perceptron implemented in pytorch over time with interpolated lines between the evaluations every 100 steps

3 Custom Module: Batch Normalization

3.1 Question 3.1 Automatic Differentiation

See custom_batchnorm.py > class CustomBatchNormAutograd(nn.Module) for the implementation of the custom batch normalization algorithm.

- For the check I have done a simple if-statement with an exception raise might it not be correct. This I do for all further batch normalization implementations
- The code passes the unit tests in `unittest.py`

3.2 Manual implementation of backward pass

3.2.1 Question 3.2 a) Backprop calculation

$\frac{\delta L}{\delta \gamma} = \frac{\delta L}{\delta y}\frac{\delta y}{\delta \gamma}$, where the first term is given by the backpropagation, and the second term is given by $\frac{\delta[\gamma \hat{x_i} + \beta]}{\delta \gamma} = \hat{x_i}$ for each y_i.

The total gradient for gamma for a specific batch is then $\sum_{i=1}^{M} \frac{\delta L}{\delta y_i} \hat{x_i}$

$\frac{\delta L}{\delta \beta} = \frac{\delta L}{\delta y} \frac{\delta y}{\delta \beta}$ where the first term is given by the backpropagation and the second term is given by $\frac{\delta[\gamma \hat{x_i} + \beta]}{\delta \beta} = 1$, so that the total gradient of β for a specific batch becomes $\sum_{i=1}^{m} \frac{\delta L}{\delta y_i} * 1 = \sum_{i=1}^{m} \frac{\delta L}{\delta y_i}$

$\frac{\delta L}{\delta x} = \frac{\delta L}{\delta y} \frac{\delta y}{\delta x}$, where the first term is given by the backpropagation, and the second term is given by $\frac{\delta[\gamma \hat{x}_i + \beta]}{\delta x_i}$.

We can utilize the calculated derivatives from the original paper, namely that

$$\frac{\delta y}{\delta x_i} = \frac{\delta l}{\delta \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\delta l}{\delta \sigma_B^2} \cdot \frac{2(x_i - \mu_B)}{m} + \frac{\delta l}{\delta \mu_B} \cdot \frac{1}{m}$$

where

$$\frac{\delta l}{\delta \hat{x}_i} = \frac{\delta l}{\delta y_i} \cdot \gamma$$

and

$$\frac{\delta l}{\delta \mu_B} = \left(\sum_{i}^{m} \frac{\delta l}{\delta \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\delta l}{\delta \sigma_B^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_B)}{m}$$

where

$$\begin{aligned} \frac{\sum_{i=1}^{m} -2(x_i - \mu_B)}{m} &= -2\frac{\sum_{i=1}^{m}(x_i - \mu_B)}{m} \\ &= -2(\frac{1}{m}\sum_{i=1}^{m} x_i - \frac{1}{m}\sum_{i=1}^{m} \mu_B) \\ &= -2(\mu - \frac{m\mu}{m}) \\ &= 0 \end{aligned} \tag{1}$$

so that

$$\frac{\delta l}{\delta \mu_B} = \left(\sum_{i}^{m} \frac{\delta l}{\delta \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) \tag{2}$$

Plugging all this into the equation for $\frac{\delta y}{\delta x_i}$ we get

$$\begin{aligned} \frac{\delta y}{\delta x_i} = \left[\frac{\delta l}{\delta \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} \right] + \left[\frac{2(x_i - \mu_B)}{m} \sum_{j=1}^{m} \frac{\delta l}{\delta \hat{x}_j} \cdot (x_j - \mu_B) \right. \\ \left. \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{\frac{-3}{2}} \right] + \left[\frac{1}{m} \sum_{j=1}^{m} \frac{\delta l}{\delta \hat{x}_j} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right] \end{aligned} \tag{3}$$

which we can then rewrite to become

$$\begin{aligned} \frac{\delta y}{\delta x_i} = \left[\frac{\delta l}{\delta \hat{x}_i} \cdot (\sigma_B^2 + \epsilon)^{-0.5} \right] - \left[\frac{(\sigma_B^2 + \epsilon)^{-0.5}}{m} \frac{(x_i - \mu_B)}{\sqrt{(\sigma_B^2 + \epsilon)}} \sum_{j=1}^{m} \frac{\delta l}{\delta \hat{x}_j} \right. \\ \left. \cdot \frac{(x_j - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}} \right] - \left[\frac{(\sigma_B^2 + \epsilon)^{-0.5}}{m} \sum_{j=1}^{m} \frac{\delta l}{\delta \hat{x}_j} \cdot \right] \end{aligned} \tag{4}$$

which we simplify by factoring out the sigma and epsilon term, m, and plug in $\hat{x}$ to become the final result, for one specific batch:

$$\frac{\delta y}{\delta x_i} = (\frac{(\sigma_B^2 + \epsilon)^{-0.5}}{m})(\left[\frac{\delta l}{\delta \hat{x}_i} \cdot m\right] - \left[\hat{x}_i \sum_{j=1}^{m} \frac{\delta l}{\delta \hat{x}_j} \cdot \hat{x}_j\right] - \left[\sum_{j=1}^{m} \frac{\delta l}{\delta \hat{x}_j}\right]) \tag{5}$$

3.2.2 Question 3.2 b) Implement the batch norm operation

See `custom_batchnorm.CustomBatchNormManualFunction` for the implementation.

The forward is similar to the one of the previous question, except that it now saves variables using the `save_for_backward()` function.

The backward gets the tensors from the forward pass and, if the parameter requires an input gradient, calculates the required gradients by following the formulas as written above.

3.2.3 Question 3.2 c) create a nn.Module

See `custom_batchnorm.CustomBatchNormManualModule` for the implementation. The initialization is similar to the one from question *a)*, while the forward checks the input, initializes the manual function and calls it with `apply()` while passing it the parameters saved during initialization.

4 Question 4 Pytorch CNN

See `convnet_pytorch.py` and `train_convnet_pytorch.py` for the implementation of the CNN. See `pytorch.job` for the job script that was used to run the script on SurfSara. One can find the raw results of this run in `results/cnn/slurm-2105150.out`.

The raw results are summarized in Figure 5 and Figure 6 for the acccuracy and loss, respectively. We see that when using the standard parameters (i.e. `lr=1e-4, batch size = 32, steps = 5000, evaluation frequency = 500, optimizer = adam`), the expected accuracy of *0.75* is successfully obtained. This is, clearly, a lot higher than what the multilayer perceptron can achieve. This is because a convolutional net has many features that are optimal for image recognition, like the fact that it is translation invariant. It doesn't always matter where it finds certain features, certain features are indicative of a specific figure (e.g. the two circles of an 8, or the distinctive corner of 2 where it goes from 'smooth line' to 'base line'.

The figures are also as expected. We see that the loss quickly decreases at the beginning of training, and this reduction slows down over time. At approximately 3500 steps we see that the validation loss becomes stably higher than the training loss, meaning we're reaching a point of overfitting (which is said to happen when the validation loss will actually start increasing, which is not yet happening here). Additionally, the accuracy behaves similarly to the loss in the opposite direction. At first it strongly increases after which it flattens out, after which the accuracy on both the training and test set seem to be stably similar, indicating we've not reached a point of overfitting (which would be the case when the accuracy on the training test is (much) higher than the validation set.)

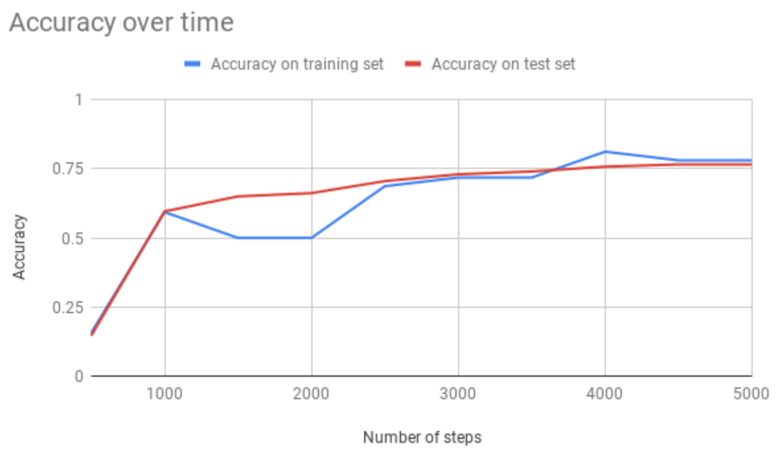

Figure 5: Accuracy of the convolutional neural net over time with interpolated lines between the evaluations every 500 steps

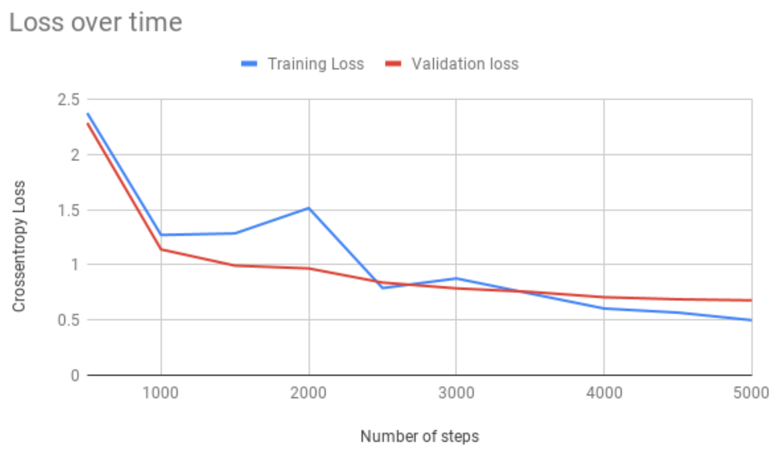

Figure 6: Loss of the convolutional neural net over time with interpolated lines between the evaluations every 500 steps