Assignment 1: MLPs, CNNs and Backpropagation

Henning Bartsch
12307912
University of Amsterdam
Amsterdam, 1012 WX Amsterdam
henning.bartsch@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of module gradients

For the multiplication of matrices, we assume vectors to be column vectors.

1.1.1 Gradients of all the individual modules

(i) Starting with the loss:

$$L(x^{(N)}, t) = -\sum_i t_i \log x_i^{(N)} = -t^T \cdot \log x^{(N)}, \tag{1}$$

where we apply the logarithm element-wise. Since t is a one-hot vector it follows:

$$\frac{\partial L}{\partial x_i^{(N)}} = -\frac{t_i}{x_i^{(N)}} = \begin{cases} -\frac{1}{x^{(N)}_{argmax(t)}} & \\ 0 & \text{if } i \neq argmax(t) \end{cases} \tag{2}$$

and therefore the vector notation:

$$\frac{\partial L}{\partial x^{(N)}} = -t^T \cdot \frac{\partial \log x^{(N)}}{\partial x^{(N)}} = -t^T \cdot \text{diag}\left(\frac{1}{x^{(N)}}\right)^{d_N, d_N} \tag{3}$$

Note that each element of $x^{(N)}$ non-zero because these are the softmax probabilities.

(ii) For the output we have a softmax:

$$x^{(N)} = \frac{\exp \tilde{x}^N}{\sum_i \exp \tilde{x}_i^N} \tag{4}$$

We first look at it component-wise where we distinguish the cases where $i = j$ and $i \neq j$.

$$x_i^{(N)} = \frac{\exp \tilde{x}_i^N}{\sum_{j'} \exp \tilde{x}_{j'}^N}$$

In the case $i = j$:

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{\exp \tilde{x}_i^N \cdot \left(\sum_{j'} \exp \tilde{x}_{j'}^N\right) - \left(\exp \tilde{x}_i^N\right)^2}{\left(\sum_{j'} \exp \tilde{x}_{j'}^N\right)^2} = x_i^{(N)} - x_i^{(N)} x_i^{(N)} = x_i^{(N)} \cdot \left(1 - x_i^{(N)}\right) \tag{5}$$

Submitted to the Spring 2019 Amsterdam Deep Learning course

For the case $i \neq j$ we compute

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{-\left(\exp x_i^{(N)}\right) \cdot \left(\exp x_j^{(N)}\right)}{\left(\sum_{j'} \exp \tilde{x}_{j'}^{N}\right)^2} = -x_i^{(N)} \cdot x_j^{(N)}. \tag{6}$$

In order to obtain the gradient for all cases, we use the indicator function $\mathbb{1}$ and combine equation 5 and 6:

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \left(x_i^{(N)}(\mathbb{1}\{i=j\} - x_j^{(N)})\right)_{i,j=1}^{d_N,d_N} \tag{7}$$

For the matrix notation we then get

$$\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \operatorname{diag}\left(x^{(N)}\right) - x^{(N)} \cdot \left(x^{(N)}\right)^T \in \mathbb{R}^{d_N \times d_N} \tag{8}$$

(iii) For the non-linearities in the hidden layers ($l < N$) we use

$$x^{(l)} = \operatorname{ReLU}(\tilde{x}^{(l)}) := \max(0, \tilde{x}^{(l)}), l = 1, ..., N-1 \tag{9}$$

Where we define the derivative at the point 0 to be 1 and therefore obtain:

$$\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial}{\partial \tilde{x}^{(l)}} ReLU(\tilde{x}^{(l)}) = \begin{cases} 1, & \text{if } \tilde{x}_i^{(l)} \geq 0 \\ 0 & \text{otherwise} \end{cases} = \operatorname{diag}(\mathbb{1}\{\tilde{x}^{(l)} \geq 0\}) \in \mathbb{R}^{d_l \times d_l} \tag{10}$$

where the resulting matrix is a diagonal matrix.

(iv) The linear modules ($l-1$) simply follow:

$$\tilde{x}^{(l)} := W^{(l)} x^{(l-1)} + b^{(l)}, l = 1, ..., N \tag{11}$$

Consequently, the derivative only depends on the weight matrix W:

$$\frac{\partial \tilde{x}^{(l)}}{\partial x^{(l-1)}} = W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}} \tag{12}$$

Or for element-wise notation:

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial x_j^{(l-1)}} = \frac{\partial}{\partial x_j^{(l-1)}} \sum_j W_{ij}^{(l)} x_j^{(l-1)} + b_i^{(l)} = W_{ij}^{(l)}$$

(v) For the $\frac{\partial \tilde{x}^{(l)}}{\partial W^l}$, we denote $\tilde{x} \in \mathbb{R}^{d_l}$ with index i and $W \in \mathbb{R}^{d_l \times d_{l-1}}$ with indices j, k.

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \begin{cases} 0, \ i \neq j \\ x_k^{(l-1)}, i = j. \end{cases} \qquad \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \begin{cases} x_k^{(l-1)}, & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

For the whole matrix W this results in a matrix where each element is a partial derivative:

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial W^{(l)}} = \begin{bmatrix} \frac{\partial \tilde{x}_i}{\partial W_{11}} & \cdots & \frac{\partial \tilde{x}_i}{\partial W_{1k}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \tilde{x}_i}{\partial W_{j1}} & \cdots & \frac{\partial \tilde{x}_i}{\partial W_{jk}} \end{bmatrix} \in \mathbb{R}^{d_l \times d_{l-1}}$$

For a fixed $\tilde{x}_i^{(l)}$ the resulting matrix contains rows where all elements are zero, except for the i-th row, which would contain $\left(x^{(l-1)}\right)^T$. This matrix can be written as

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial W^{(l)}} = \begin{bmatrix} \mathbf{0} \\ \left(x^{(l-1)}\right)^T \\ \mathbf{0} \end{bmatrix} = e_i \cdot \left(x^{(l-1)}\right)^T \in \mathbb{R}^{d_l \times d_{l-1}}, \tag{13}$$

where $\mathbf{0}$ is a row vector of zeros and e_i the standard basis vector of size d_l that has one non-zero element at position i. With this in mind, we can calculate the whole gradient as a three-dimensional tensor. For each element in $\tilde{x}^(l)$ we would compute the gradient matrix according to 13 and stack these matrices row-wise to obtain

$$\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \left(e_i \cdot \left(x^{(l-1)} \right)^T \right)_{i=1} \in \mathbb{R}^{d_l \times (d_l \times d_{l-1})} \tag{14}$$

(vi) As the bias term $b^{(l)}$ is simply added to $\tilde{x}^{(l)}$, we can compute

$$\frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \mathbf{1} \in \mathbb{R}^{d_l \times d_l} \tag{15}$$

where $\mathbf{1}$ is the identity matrix.

1.1.2 Backpropagation using module gradients

With the module gradient from the previous section we are now going to use the chain rule to compute the gradients of the loss L also with respect to the weights W and biases b. In general, we are using these gradients to update the parameter of the network and propagate the error back through all layers. From the module perspective, we basically pass the error $\frac{\partial L}{\partial x^{(l)}}$ (output of the previous module) as an input into a function and use its output $\frac{\partial L}{\partial \tilde{x}^{(l)}}$ again for the following module. Additionally, the error for each layers is saved to also compute the errors for the weights and biases.

(i) For the softmax module we compute

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \cdot \left[\operatorname{diag}\left(x^{(N)} \right) - x^{(N)} \cdot \left(x^{(N)} \right)^T \right] \in \mathbb{R}^{1 \times d_N} \tag{16}$$

where we obtain a matrix of size $d_N \times d_N$ for $x^{(N)} \cdot \left(x^{(N)} \right)^T$ and a diagonal matrix of the same size when we multiply out the brackets. Note that we assume single input vectors. In the case of batches comprising multiple inputs we would have to deal with three dimensional tensors.

(ii) For the backward pass of linear modules, for the layers $l = 1, ..., N-1$, we receive the error of the subsequent layer's module $\frac{\partial L}{\partial \tilde{x}^{(l+1)}}$ and simply multiply it by that layer's weight matrix

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \cdot W^{(l+1)} \in \mathbb{R}^{1 \times d_l} \tag{17}$$

(iii) Similarly, the backward pass through the ReLu modules is computed by

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \cdot \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} \cdot \operatorname{diag}(\mathbb{1}\{\tilde{x}^{(l)} \geq 0\}) \in \mathbb{R}^{1 \times d_l} \tag{18}$$

(iv) In order to compute the gradients with respect to the weights, we need the previous backpropagated errors.

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \cdot \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \cdot \left(e_i \cdot \left(x^{(l-1)} \right)^T \right)_{i=1} \in \mathbb{R}^{1 \times (d_l \times d_{l-1})} \tag{19}$$

When looking at 14 we can see that the tensor consists of d_l matrices of size $d_l \times d_{l-1}$, where each matrix only has one non-zero row that contains $\left(x^{(l-1)} \right)^T$. When we multiply this tensor with $\frac{\partial L}{\partial \tilde{x}^{(l)}} \in \mathbb{R}^{1 \times d_l}$ we effectively compute the weighted sum of the sub-matrices (in NumPy we would use the built-in function tensordot[1]). Given only one non-zero row in each sub-matrix, we can compute

(v) For the gradients with respect to the biases we compute

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \cdot \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \cdot \mathbf{1} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \in \mathbb{R}^{1 \times d_l} \tag{20}$$

where $\mathbf{1}$ is the identity matrix of size $d_l \times d_l$.

[1] https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.tensordot.html

1.1.3 Batches instead of single input

The previous equation assume the input to a single vector but for the training of the MLP we are going to use mini-batches. Consequently, when a batch size $B \neq 1$ is used, the dimensionality of the input changes. In order to process a batch of samples, they concatenated along an axis and create a matrix. In particular, the dimensions of each gradients are extended by the batch size B.

$$\begin{aligned}
\frac{\partial L}{\partial \tilde{x}^{(N)}} &\in \mathbb{R}^{B \times d_N} \\
\frac{\partial L}{\partial \tilde{x}^{(l<N)}} &\in \mathbb{R}^{B \times d_l} \\
\frac{\partial L}{\partial x^{(l<N)}} &\in \mathbb{R}^{B \times d_l} \\
\frac{\partial L}{\partial W^{(l)}} &\in \mathbb{R}^{B \times (d_l \times d_{l-1})} \\
\frac{\partial L}{\partial b^{(l)}} &\in \mathbb{R}^{B \times d_l}
\end{aligned}$$

In the end the loss for each input vector is computed and the resulting total loss is averaged over the mini-batch:

$$L_{total} = \frac{1}{B} \sum_{s=1}^{B} L_{individual}(x^s, t^s) \tag{21}$$

1.2 NumPy Implementation

For training and testing of all models, starting with the MLP NumPy implementation, we are going to use the CIFAR-10 [2] data set, which consists of 60000 colour images of size 32 x 32 pixels, evenly divided in 10 classes. The class labels are encoded as one-hot vectors. Using the provided `cifar10_utils.py`, we can sample mini-batches from the data and feed images to the MLP. Note that the input images have to be reshape to fit the input layer of size $3 \times 32 \times 32 = 3072$.

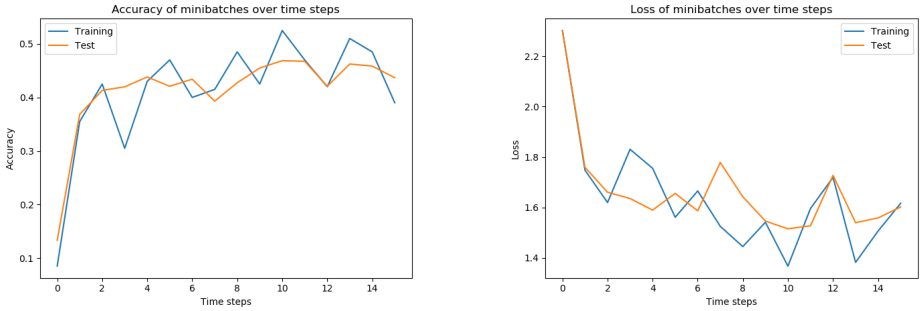
The modules forward and backward pass are implemented in Python according to the formulas computed in section 1.1 and 1.1.2, using matrix multiplication and also incorporating mini-batches. In principal, the MLP architecture and its training parameters can be defined freely by passing the designated arguments to the function. Figure 1 shows the results for the default parameters:

```
# Default constants
DNN_HIDDEN_UNITS_DEFAULT = '100'
LEARNING_RATE_DEFAULT = 2e-3
MAX_STEPS_DEFAULT = 1500
BATCH_SIZE_DEFAULT = 200
EVAL_FREQ_DEFAULT = 100
```

Metric	Best	Last
Train Acc	1.3585	1.5531
Train Loss	0.52	0.455
Test Acc	1.5127	1.5199
Test Loss	0.4672	0.4629

Table 1: Training and test results for the Numpy MLP

[2] https://www.cs.toronto.edu/~kriz/cifar.html



(a) Accuracy (b) Loss

Figure 1: Results for the default MLP NumPy implementation

2 PyTorch MLP

Following the NumPy implementation, we used PyTorch modules to implement and experiment with different architectures and parameter settings. After implementing the model, it was tested we the following default parameters:

```
#default parameters
DNN_HIDDEN_UNITS_DEFAULT = '100'
LEARNING_RATE_DEFAULT = 2e-3
MAX_STEPS_DEFAULT = 1500
BATCH_SIZE_DEFAULT = 200
EVAL_FREQ_DEFAULT = 100
OPTIMIZER_DEFAULT = 'SGD'
```

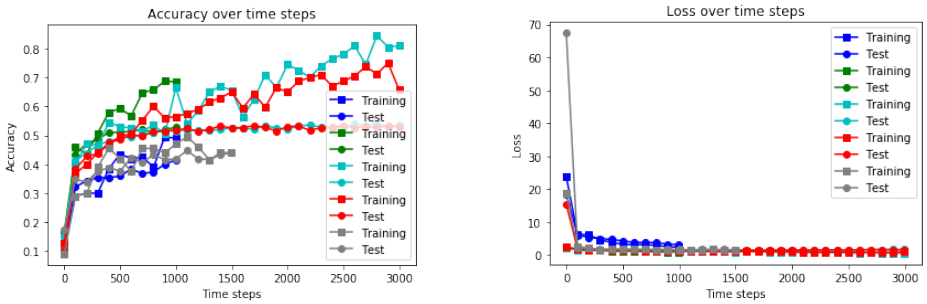
In order to achieve an accuracy of at least 52% a grid search was run to test out different parameter settings and their effect on the model's performance.

```
#parameters for grid search
learning_rates = [0.0002, 0.002 , 0.02]
max_steps = [1000, 3000]
batch_sizes = [200, 400]
optimizers = ['SGD', 'Adam', 'Adadelta']

dnn_hidden_possibilities = [
    '1000', '2000', #shallow
    '100, 100', '500,500',
    '100, 100, 100', '500,500,500,500',
    '500, 400, 300, 100',
    '100, 100, 100, 100, 100', '500,500,500,500,500', #deep
    '800,400,200,100,50'
]

#360 total combinations

#mlp_settings = list(itertools.product(*[learning_rates, max_steps, #batch_sizes,
    optimizers, dnn_hidden_possibilities]))
#len(mlp_settings) = 360
```



(a) Accuracy (b) Loss

Figure 2: Results for the MLP Pytorch implementation. The best model is coloured in red and the default in gray

In the figures 2 some models are compared to the default performance. The red curve shows a steady increase of the training accuracy but only marginal improvements on the test set. In the experiments, the best model achieved an accuracy of 54.2 % on the test with the following parameters:

```
#best parameters
learning_rate = 0.0002
max_steps = 3000
batch_size = 200
optimizer = 'Adam'
dnn_hidden_units = ['800,400,200,100,50']
```

Metric	Best	Last
Train Acc	0.845	0.81
Train Loss	0.4965	0.4965
Test Acc	0.5418	0.5348
Test Loss	1.3878	1.6623

Table 2: Results for best MLP model determined by the grid search

In the experiments we could observe different effects of the parameters settings. For instance, a larger learning rate (e.g. 0.02) lead to bad results probably because the weight updates become too big and the model can't learn properly. Furthermore, the Adam optimizer yields better accuracies in many cases when compared to SGD or Adadelta. The number of training iterations normally did not make a significant difference but it should not be too big or the tendency to overfit may become higher. The greatest effect on the performance however, had the parameter `dnn_hidden_units` that determines the number and features size of the network's layers. In general, medium deep architectures outperformed both shallower or deeper architectures. The best results were achieved with layer comprising different features sizes, while also decreasing towards the output layer.

3 Custom Module: Batch Normalization

3.1 Automatic differentiation

The implementation of the Batch Normalization with automatic differentiation can be seen in the file `custom_batchnorm.py` in the designated `class CustomBatchNormAutograd(nn.Module)` that consists of the initialization and forward pass.

3.2 Manual implementation of backwards pass

The gradients are derived in the first step with respect to $\hat{x_j}$ to derive the gradients with respect to x_j in the second step.

(i) We note that $\frac{\partial L}{\partial y_i}$ will be given to the module as a function parameter

(ii) We apply the chain rule to derive the gradients as follows:

$$\frac{\partial L}{\partial \gamma_j} = \sum_{s=1}^{B}\sum_{i=1}^{C} \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \gamma_j} = \sum_s \frac{\partial L}{\partial y_j^s}\frac{\partial y_j^s}{\partial \gamma_j} = \sum_s \frac{\partial L}{\partial y_j^s}\hat{x}_j^s = \frac{\partial L}{\partial y_j}\cdot \hat{x}_j. \tag{22}$$

Note that we sum over mini-batches of size B and for all indices C.

$$\frac{\partial L}{\partial \beta_j} = \sum_{s=1}^{B}\sum_{i=1}^{C} \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \beta_j} = \sum_s \frac{\partial L}{\partial y_j^s}\frac{\partial y_j^s}{\partial \gamma_j} = \sum_s \frac{\partial L}{\partial y_j^s} \tag{23}$$

$$\frac{\partial L}{\partial \hat{x}_j} = \sum_{s=1}^{B}\sum_{i=1}^{C} \frac{\partial L}{\partial y_i^s}\frac{\partial y_i^s}{\partial \hat{x}_j} = \sum_s \frac{\partial L}{\partial y^s}\cdot \gamma_j \tag{24}$$

(iii) We continue with the mean and variance by applying the chain rule using $\hat{x}$:

$$\frac{\partial L}{\partial \mu} = \frac{\partial L}{\partial \hat{x}^s}\frac{\partial \hat{x}^s}{\partial \mu} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial \mu} \tag{25}$$

To compute this gradient we need to derive the two missing gradients as follows:

$$\frac{\partial \hat{x}^s}{\partial \mu} = \frac{-1}{\sqrt{\sigma^2+\epsilon}} \tag{26}$$

$$\frac{\partial \sigma^2}{\partial \mu} = \frac{1}{B}\sum_{s=1}^{B} -2(x_s - \mu) \tag{27}$$

We also compute the gradient with respect to σ^2:

$$\frac{\partial L}{\partial \sigma^2} = \frac{\partial L}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial \sigma^2} \tag{28}$$

and with

$$\frac{\partial \hat{x}}{\partial \sigma^2} = -\frac{1}{2}\sum_{s=1}^{B}(x_s - \mu)(\sigma^2+\epsilon)^{-1.5} \tag{29}$$

we can combine 26 and 27 to get

$$\frac{\partial L}{\partial \mu} = \sum_{s=1}^{B}\frac{\partial L}{\partial \hat{x}_s}\cdot\frac{-1}{\sqrt{\sigma^2+\epsilon}} \tag{30}$$

(iv) Lastly we apply the chain rule to compute the gradient with respect to x_j:

$$\frac{\partial L}{\partial x_j} = \frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial x_j} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial x_j} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x_j} \tag{31}$$

The all except three gradients are already computed, so we continue with

$$\frac{\partial \hat{x}_i}{\partial x_j} = \frac{-1}{\sqrt{\sigma^2+\epsilon}} \tag{32}$$

$$\frac{\partial \mu}{\partial x_j} = \frac{1}{B} \tag{33}$$

$$\frac{\partial \sigma^2}{\partial x_j} = \frac{2(x_j - \mu)}{B} \tag{34}$$

Now we put together these gradients with 24, 30 and 28 to obtain:

$$\frac{\partial L}{\partial x_j} = \frac{1}{B}\frac{1}{\sqrt{\sigma^2+\epsilon}}\left[B\frac{\partial L}{\partial \hat{x}_j} - \sum_{i=1}^{B}\frac{\partial L}{\partial \hat{x}_i} - \hat{x}_j\sum_{i=1}^{B}\frac{\partial L}{\partial \hat{x}_i}\hat{x}_i\right] \tag{35}$$

For the implementation of the gradients computed in 30, 28, 24 and 35 we see that we only need to store γ, $\hat{x}$ and the fraction $\frac{1}{\sqrt{\sigma^2+\epsilon}}$ for the backpropagation. Using the gradient-wise notation makes it easy to implement the `class CustomBatchNormManualFunction` and corresponding `class CustomBatchNormManualModule`.

4 PyTorch CNN

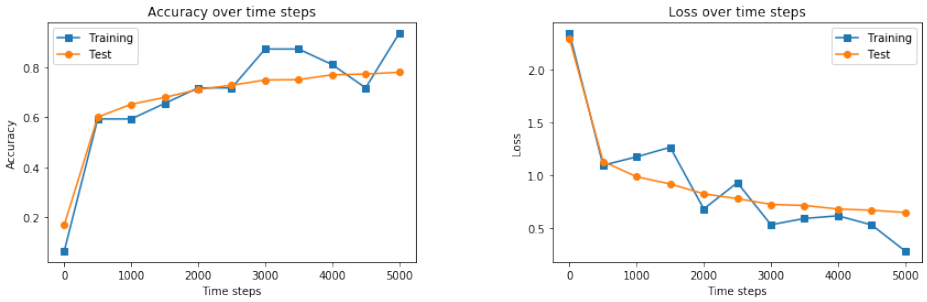
In the last part of the assignment we are going to build a Convolutional Neural Network (CNN) based on the VVG architecture [3]. The concrete architecture is specified in figure 4 (see appendix) and run with default parameter setting as follows:

```
# Default constants
LEARNING_RATE_DEFAULT = 1e-4
BATCH_SIZE_DEFAULT = 32
MAX_STEPS_DEFAULT = 5000
EVAL_FREQ_DEFAULT = 500
OPTIMIZER_DEFAULT = 'ADAM'
```

In comparison to the previous models, the CNN is trained over 5000 iterations with a mini-bach size of 32. The accuracy and loss curves displayed in figure 3 clearly demonstrate that the CNN classifier achieves a higher accuracy than the simpler MLP, which had an accuracy of ca. $40 - 50\%$. Despite the very high training accuracy of up to 93% the CNN does not show overfitting behaviour when compared to the test set.

Metric	Best	Last
Train Acc	0.9375	0.9375
Train Loss	0.2848	0.2848
Test Acc	0.6494	0.6494
Test Loss	0.7817	0.7817

Table 3: Training and validation results of the ConvNet



(a) Accuracy (b) Loss

Figure 3: Results for the default ConvNet Pytorch implementation

[3] https://arxiv.org/abs/1409.1556

A Appendix

Name	Kernel	Stride	Padding	Channels In/Out
conv1	3×3	1	1	3/64
maxpool1	3×3	2	1	64/64
conv2	3×3	1	1	64/128
maxpool2	3×3	2	1	128/128
conv3_a	3×3	1	1	128/256
conv3_b	3×3	1	1	256/256
maxpool3	3×3	2	1	256/256
conv4_a	3×3	1	1	256/512
conv4_b	3×3	1	1	512/512
maxpool4	3×3	2	1	512/512
conv5_a	3×3	1	1	512/512
conv5_b	3×3	1	1	512/512
maxpool5	3×3	2	1	512/512
avgpool	1×1	1	0	512/512
linear	–	-	-	512/10

Table 1. Specification of ConvNet architecture. All *conv* blocks consist of 2D-convolutional layer, followed by Batch Normalization layer and ReLU layer.

Figure 4: Specific architecture description of the ConvNet as given in the assignment