# Assignment 03
# TDT4265 - Datasyn og Dyplæring

Elias Mohammed Elfarri
Christopher Michael Vibe

Februar, 2021

# Contents

**1 Task 1 - Theory** **1**
1.a . . . . . 1
1.b . . . . . 3
1.c . . . . . 3
1.d . . . . . 3
1.e . . . . . 4
1.f . . . . . 4
1.g . . . . . 5

**2 Task 2 - Convolutional Neural Networks** **6**
2.a . . . . . 6
2.b . . . . . 6

**3 Task 3 - Deep Convolutional Network for Image Classification** **7**
3.a (Code implementation found in task3.py) . . . . . 7
3.b . . . . . 9
3.c (Code implementation found in task3cd.py) . . . . . 10
3.d (Code implementation found in task3cd.py) . . . . . 14
3.e (Code implementation found in task3e.py) . . . . . 15
3.f . . . . . 15

**4 Task 4 - Transfer Learning with ResNet** **16**
4.a (Code implementation found in task4a.py) . . . . . 16
4.b . . . . . 17
4.c . . . . . 18

# 1 Task 1 - Theory

## 1.a

For an image I with a pixel range of 0 to 7:

$$\begin{bmatrix} 1 & 0 & 2 & 3 & 1 \\ 3 & 2 & 0 & 7 & 0 \\ 0 & 6 & 1 & 1 & 4 \end{bmatrix}$$

and a Sobel kernel K:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

To retain the 3x5 dimension of the image after convolution, it is common and simple to use a method which is called zero padding and represents the image in the form:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 3 & 1 & 0 \\ 0 & 3 & 2 & 0 & 7 & 0 & 0 \\ 0 & 0 & 6 & 1 & 1 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The convolving of this image with the sobel kernel will give the following results:

$$0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 + 0 \cdot (-2) + 1 \cdot 0 + 0 \cdot 2 + 0 \cdot (-1) + 3 \cdot 0 + 2 \cdot 1 = 2$$

$$0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 + 1 \cdot (-2) + 0 \cdot 0 + 2 \cdot 2 + 3 \cdot (-1) + 2 \cdot 0 + 0 \cdot 1 = -1$$

$$0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 + 0 \cdot (-2) + 2 \cdot 0 + 3 \cdot 2 + 2 \cdot (-1) + 0 \cdot 0 + 7 \cdot 1 = 11$$

$$0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 + 2 \cdot (-2) + 3 \cdot 0 + 1 \cdot 2 + 0 \cdot (-1) + 7 \cdot 0 + 0 \cdot 1 = -2$$

$$0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 + 3 \cdot (-2) + 1 \cdot 0 + 0 \cdot 2 + 7 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 = -13$$

$$0 \cdot (-1) + 1 \cdot 0 + 0 \cdot 1 + 0 \cdot (-2) + 3 \cdot 0 + 2 \cdot 2 + 0 \cdot (-1) + 0 \cdot 0 + 6 \cdot 1 = 10$$

$$1 \cdot (-1) + 0 \cdot 0 + 2 \cdot 1 + 3 \cdot (-2) + 2 \cdot 0 + 0 \cdot 2 + 0 \cdot (-1) + 6 \cdot 0 + 1 \cdot 1 = -4$$

$$0 \cdot (-1) + 2 \cdot 0 + 3 \cdot 1 + 2 \cdot (-2) + 0 \cdot 0 + 7 \cdot 2 + 6 \cdot (-1) + 1 \cdot 0 + 1 \cdot 1 = 8$$

$$2 \cdot (-1) + 3 \cdot 0 + 1 \cdot 1 + 0 \cdot (-2) + 7 \cdot 0 + 0 \cdot 2 + 1 \cdot (-1) + 1 \cdot 0 + 4 \cdot 1 = 2$$

$$3 \cdot (-1) + 1 \cdot 0 + 0 \cdot 1 + 7 \cdot (-2) + 0 \cdot 0 + 0 \cdot 2 + 1 \cdot (-1) + 4 \cdot 0 + 0 \cdot 1 = -18$$

$$0 \cdot (-1) + 3 \cdot 0 + 2 \cdot 1 + 0 \cdot (-2) + 0 \cdot 0 + 6 \cdot 2 + 0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 = 14$$

$$3 \cdot (-1) + 2 \cdot 0 + 0 \cdot 1 + 0 \cdot (-2) + 6 \cdot 0 + 1 \cdot 2 + 0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 = -1$$

$$2 \cdot (-1) + 0 \cdot 0 + 7 \cdot 1 + 6 \cdot (-2) + 1 \cdot 0 + 1 \cdot 2 + 0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 = -1$$

$$0 \cdot (-1) + 7 \cdot 0 + 0 \cdot 1 + 1 \cdot (-2) + 1 \cdot 0 + 4 \cdot 2 + 0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 = 6$$

$$7 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 + 1 \cdot (-2) + 4 \cdot 0 + 0 \cdot 2 + 0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 = -9$$

Resulting in the following output 3x5 convolved image:

$$\begin{bmatrix} 2 & -1 & 11 & -2 & -13 \\ 10 & -4 & 8 & 2 & -18 \\ 14 & -1 & -5 & 6 & -9 \end{bmatrix}$$

Note that the mapping here is not from 0 to 7 and some values are even being negative. This is because our kernel is a Sobel x operator which is approximately the same as the gradient in the x direction.High values, both negative and positive are supposed to be interpreted as an edge being detected. If we use first a blur kernel on an image, then Sobel x as well as the Sobel y operator (on two different blurred images), we would be able to find the magnitude between the x and y operator and map this onto the 0 to 7 pixel range. This will result into showing the edge detection of the given image. Futhermore the activation function squeezes this convolved image into the correct pixel range.

### 1.b

**Max Pooling (biggest contribution)** - This returns the max value in a receptive field, such that small translational spatial shifts will still result in the same max value in said receptive field. Hence why max pooling provides translational invariance for small translations in the input.

**Convolutional layer (some contribution)** - Whilst a convolutional layer does not directly reduce sensitivity to translational variations. It provides something called translational equivariance. This means that if the input shifts by a given translation, so does the convolved output. More technically small translations in the input, will result in the same features being detected but also shifted. This way the convolutional layer sets up the max pooling for translational invariance.

**Activation function (no contribution)** - The activation function just is supposed to add non-linearities between each convolutional layer. Otherwise without it, stacking multiple convolutional layers will just be the same as one convolution as they are linear operations. The activation functions also provides a degree to which how much the input responded to convoluting with a kernel filter. In other words activation functions do not contribute to translational invariance.

### 1.c

The formula for general padding assuming that height and width are the same is:

$$P = \frac{(S-1)W + K - S}{2}$$

Where S is for stride, W is for width, K is for kernel. By assumption of this function, kernel size is of odd numbers, and width and stride are integers. When we plug in the numbers we get the following:

$$P = \frac{(1-1)W + 5 - 1}{2} = \frac{5-1}{2} = \frac{4}{2} = \underline{\underline{2}}$$

Hence we need 2 padding layers on each side to retain the dimensions of the input image after the convolution layer.

### 1.d

Since width and height are of equal size we denote a single parameter for input size $I$ as well as an output size $O$. K, S and P are kernel size, stride and padding layer respectively. The following formula gives the output size after one has chosen the appropriate parameters:

$$O = \frac{I - K + 2P}{S} + 1$$

We rewrite this to find the kernel size:

$$K = I + 2P + S - OS = 512 + 2 \cdot 0 + 1 - 504 \cdot 1 = \underline{\underline{9}}$$

Hence the kernel size is therefore 9x9.

### 1.e

Here we can again use the same formula, but with different parameters plugged in:

$$O = \frac{I - K + 2P}{S} + 1 = \frac{504 - 2 + 2 \cdot 0}{2} + 1 = 251 + 1 = \underline{\underline{252}}$$

Giving an output of 252x252 for each of the 12 feature maps that have gone through the subsampling after the first convolutional layer, or 12x252x252.

### 1.f

The size of each feature map after convolution with a kernel of 3x3, stride 1, no padding and an input of 252x252 gives the following:

$$O = \frac{I - K + 2P}{S} + 1 = \frac{252 - 3 + 2 \cdot 0}{1} + 1 = 249 + 1 = \underline{\underline{250}}$$

such that the size of each feature map after the second convolutional layer is 250x250 or 12x250x250, where there are 12 feature maps.

**1.g**

| **Layers** | **Output size (ChxWxH)** | **Weights** | **Biases** | **Parameters** |
|---|---|---|---|---|
| Input | 3x32x32 | - | - | - |
| Conv2D(F=32,K=5,S=1,P=2) | 32x32x32 | 3x32x5x5 | 32 | 2432 |
| ReLU | 32x32x32 | - | - | - |
| MaxPool(K=2,S=2) | 32x16x16 | - | - | - |
| Conv2D(F=64,K=5,S=1,P=2) | 64x16x16 | 32x64x5x5 | 64 | 51264 |
| ReLU | 64x16x16 | - | - | - |
| MaxPool(K=2,S=2) | 64x8x8 | - | - | - |
| Conv2D(F=128,K=5,S=1,P=2) | 128x8x8 | 64x128x5x5 | 128 | 204928 |
| ReLU | 128x8x8 | - | - | - |
| MaxPool(K=2,S=2) | 128x4x4 | - | - | - |
| Flatten | 2048 | - | - | - |
| Fully-Connected(2048 -> 64) | 64 | 2048x64 | 64 | 131136 |
| ReLU | 64 | - | - | - |
| Fully-Connected(64 -> 10) | 10 | 64x10 | 10 | 650 |
| **Total sum** | - | 390112 | 298 | 390410 |

The total trainable parameters in this CNN is 390410, where the process is shown in the table above.

# 2 Task 2 - Convolutional Neural Networks

## 2.a

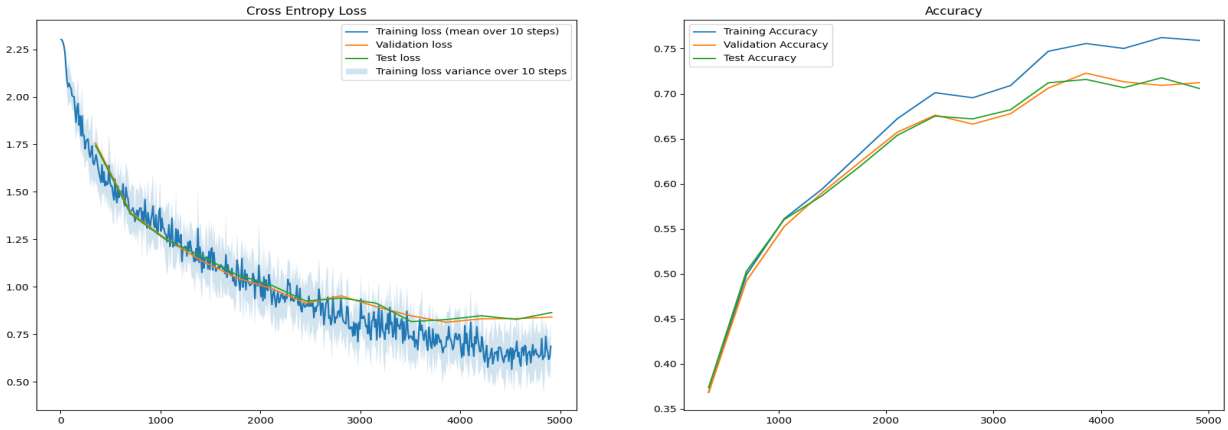


Figure 1: loss and accuracy over a CNN model, using network architecture/-parameters from the network described in 1.g

## 2.b

| - | **Loss** | **Accuracy** |
|---|---|---|
| Training | 0.7095 | 75.91% |
| Validation | 0.8413 | 71.22% |
| Test | 0.8652 | 70.58% |

# 3 Task 3 - Deep Convolutional Network for Image Classification

## 3.a (Code implementation found in task3.py)

| Model 1 | | | |
|---|---|---|---|
| **Layer** | **Layer Type** | **Filters** | **Activation** |
| 1 | Conv2D | 64 | - |
| 1 | Batchnorm2D | - | ReLU |
| 1 | Conv2D | 64 | - |
| 1 | Batchnorm2D | - | ReLU |
| 1 | Maxpool2D | - | - |
| 2 | Conv2D | 64 | - |
| 2 | Batchnorm2D | - | ReLU |
| 2 | Conv2D | 64 | - |
| 2 | Batchnorm2D | - | ReLU |
| 2 | Maxpool2D | - | - |
| 3 | Conv2D | 64 | - |
| 3 | Batchnorm2D | - | ReLU |
| 3 | Conv2D | 64 | - |
| 3 | Batchnorm2D | - | ReLU |
| 3 | Maxpool2D | - | - |
| - | Flatten | - | - |
| 4 | Fully-connected | 64 | ReLU |
| 5 | Fully-connected | 10 | Softmax |

Model 1 had the best performance from the two models with a test accuracy of over 80% in 7 epochs. It was used with an **Adam-optimizer** with regularization (AdamW in pytorch). The parameters included a batch size of 32, learning rate of 3e-4, weight decay of 0.05 and finally data augmentation of the type horizontal flips (50% probability) and random rotations between -3 and 3 degrees. The first conv2D has a kernel size of $7x7$, padding of $3x3$ and stride of 1. The rest of the conv2Ds have kernel size of $5x5$, padding of $2x2$ and stride of 1. All maxpool2Ds have kernel size of $2x2$ and stride of $2x2$. Finally the weight initialization was not altered and stayed default to what pytorch uses for ReLU.

| Model 2 | | | |
|---|---|---|---|
| **Layer** | **Layer Type** | **Filters** | **Activation** |
| 1 | Conv2D | 32 | ReLU |
| 1 | Maxpool2D | - | - |
| 2 | Conv2D | 64 | ReLU |
| 2 | Maxpool2D | - | - |
| 3 | Conv2D | 128 | ReLU |
| 3 | Maxpool2D | - | - |
| 4 | Conv2D | 64 | ReLU |
| 4 | Maxpool2D | - | - |
| - | Flatten | - | - |
| 5 | Fully-connected | 64 | ReLU |
| 6 | Fully-connected | 10 | Softmax |

This second model achieved a final test accuracy of 76.09% with convergence in 8 epochs. Model 2 has a lower amount of trainable parameters than model 1 but performs still very well. The same optimizer is used for this model as the architecture above, with a learning rate of 0.001, betas=(0.9, 0.999), weight_decay=0.01. As opposed to model 1 batch size of 64 was used but it was run with data augmentation; colour jitter (hue=.05, saturation=.05) and random horizontal flip (50%). The first conv2D has a kernel of $5x5$ and padding of $2x2$ with stride 1, and the rest of conv2Ds have kernel of $3x3$, padding of $2x2$ and stride of 1. Maxpool2D has a kernel of $2x2$ and stride of $2x2$. This model was highly repetitive so as to reduce the amount of variables during optimization. The intuition behind the filter number increasing and then decreasing was to prepare the pipeline for convergence towards the fully connected end-point. The mappings at the end are only 2x2 images! It could still be improved, but achieved over the benchmark of 75%.

Note that both these models were designed and trained on $32x32$ RGB images.

### 3.b

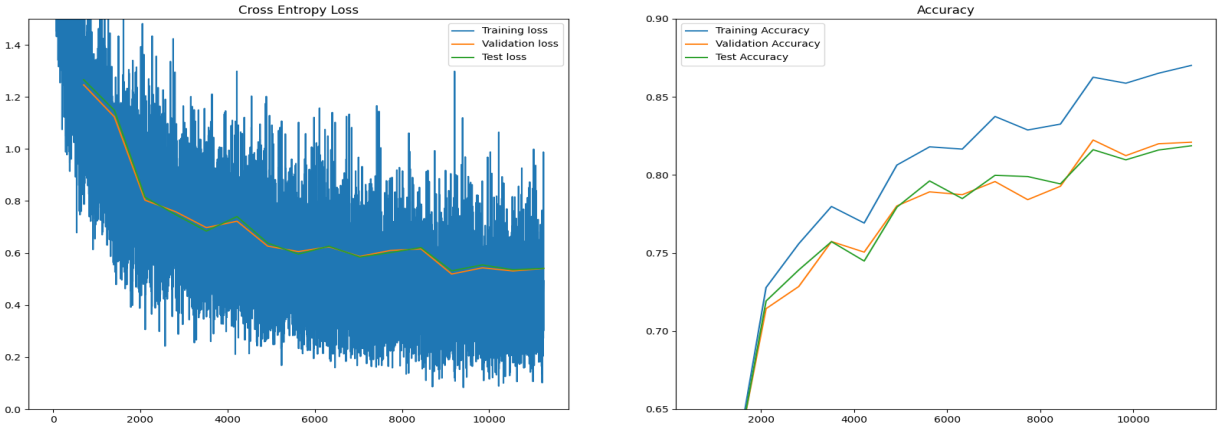


Figure 2: Plots of the best model reaching a final test accuracy of 82.76% converging within 7 epochs.

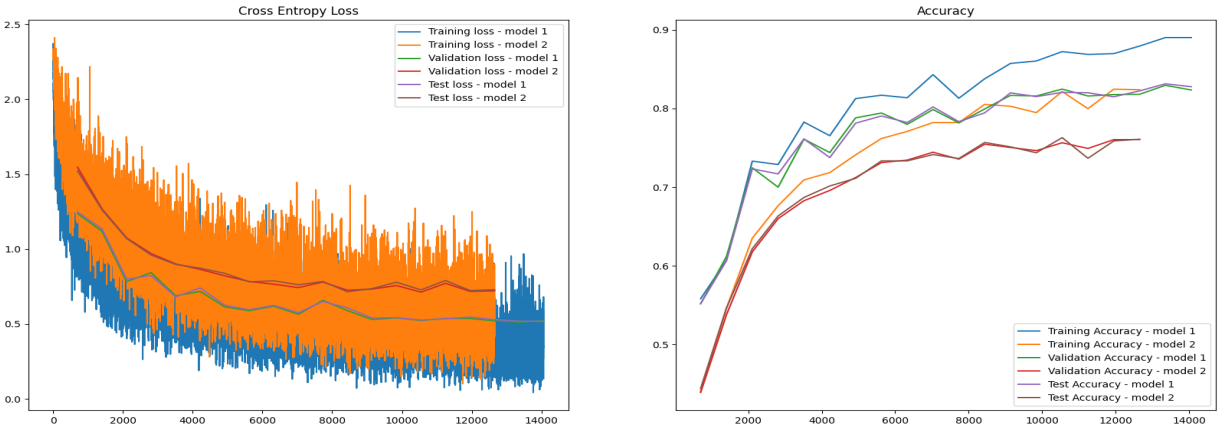


Figure 3: Plots of the best model (model 1) and the next best model (model 2), with final test accuracy of 82.76% and 76.09% respectively and convergence within 7 and 8 epochs.

| Model 1 | | |
|---|---|---|
| - | **Loss** | **Accuracy** |
| Training | 0.1348 | 89.00% |
| Validation | 0.5240 | 82.34% |
| Test | 0.5177 | **82.76%** |
| Model 2 | | |
| Training | 0.6392 | 82.37% |
| Validation | 0.7404 | 76.04% |
| Test | 0.7283 | **76.09%** |

### 3.c (Code implementation found in task3cd.py)

These observations and discussions are limited to model 1. Note that discussion on each bullet point can be found on the next page.
**Improved performance:**

- **Smaller batch size** - From 64 to 32.
- **Adjusting filters** - From [32,64,128] to [64,64,64].
- **Data augmentation** - small rotations, horizontal flips and Gaussian noise.
- **Batchnorm2D** - after each Conv2D.
- **Adam Optimizer** - with regularization and learning rate of 3e-4.
- **Neural Network Architecture** - not necessarily one type that is better than others.
- **Choice of learning rate** - generally based on which optimizer is used.
- **Weight initilization** - In combination with its ideal activation function.

**Worsened performance:**

- **TanH activation** - with Xavier weight initialization instead of ReLU with (kaiming-unifrom).
- **Excessive Data augmentation** - specifically big rotation interval [-90,90].
- **Adam Optimizer** - with high learning rate (5-e2).
- **Drop out** - with too high probability.
- **BatchNorm1D** - in linear/fully-connected layers.

**Why did _ improve performance?**

**Smaller batch size** - This improved performance as the probability for "harder" batches to occur is smaller in a smaller batches. A lower batch size generally is more noisy, results in smaller gradient steps and is known to help with better generalization too.

**Adjusting filters** - This improvement is very data-set and architectur dependent, but in this case increasing filters on the first convolutional layer and decreasing the filters on the last convolutional layer gave a little better performance. This could be explained as micro features that are extracted at the very beginning are a lot more important for performance than the macro features that the last convolutional layer extracts. One could argue that this has something to do with the image size, and if the resolution/detail level was higher, that decreasing macro feature maps would be detrimental to performance.

**Data Augmentation** - This improvement comes from giving the model more data to train on, and thus become better at generalizing. We made use of techniques that are trivial such as flipping and rotating the images. More complicated augmentations were also experimented with such as Gaussian noise. The latter makes it harder for the model to "memorize" data and prevents it from over-fitting/promotes generalization.

**Batchnorm2D** - If done after convolutional layers, it normalizes the feature maps to be within the same range before it goes through an activation function (ReLU) where it is squeezed into the correct picture range. Generally batch normalization increases learning because it limits the span of change in the input data, and helps the deeper layers understand which areas are important to focus on (avoidance of saturation). In statistical terms, batch normalization decreases the covariance shift.

**Adam Optimizer with regularization** - This improvement does not necessarily mean that the Adam Optimizer is better in itself. It is possible to find SGD with momentum that performs as well or better than adam optimizers. However, the real cause for the improvement in this case was the weight decay/L2 regularization that was combined with the adam optimizer. The theme of regularization and its significance has already been covered in assignment 1, recalling to our previous answer, the regularization term introduces penalty to the cost function such that the optimization problem is restricted. This way it motivates the cost function to find a more generalized solution in a slightly modified solution space. When it comes to the adam optimizer, we decided to use this optimizer, because it is adaptive, and required less fine-tuning of hyper-parameters as we search for good model

designs.

**Neural Network Architecture** - Whilst each element in a network architecture can improve overall performance on its own, it is also important to focus on the architecture holistically (more than the sum of its parts). In other words in what sequence the conv2DS, batch normalization happens, how many iterations of said convolution layers, what the kernel, stride and padding looks like for a convolutional layer as well as what type of pooling is used. Whilst it is hard to cover why each of these things are important, it is not hard to understand that the convolutional part of the network creates the fundamental data that is used in the deeper layers and therefore influences their perception of what data is being classified. In model2 we attempted to use some intuition of a pipeline that expands and then narrows before connecting to the fully connected classification portion of the CNN.

**Choice of learning rate** - Higher or lower learning rate can be the difference between better or worse performance. This is of course in relation to what optimizer is used and if said optimizer has a momentum term or not. The reason for why learning rate influences performance is because of how big steps the gradient is allowed to take, if too big it may overshoot and miss on what could've been a great local solution in the solution space. This can be specifically observed if one uses Adam or SGD with momentum and have a learning rate that is too big (5e-2). Thus learning rate should be calibrated such that the given optimizer is performing to its best ability.

**Weight initilization** - Pytorch initializes this by default, there might be other initializations that work better with the activation functions that one has choosen for the network. Specifically, we observed in assignment 2 that TanH performs very well with the Xavier/Randomly normal distributed weight initialization, as it reduced the risk of saturation. Performance improving relationships between activation functions and weight initialization combinations could be found in other combinations, and they would give better results based factors like saturation aversion. In our case ReLU + Kaiming initialization was used, where the Kaiming initialization is essentially the Xavier initialization that is made suitable for the ReLU activation function. The reasons for why the weight initialization works is similar to the one given in assignment 2, in short randomly centering small weight values around zero is beneficial for training.

### Why did _ worsen performance?

**TanH activation** - The TanH activation function much like the Sigmoid suffers from exploding gradient/Saturation issues as well as the vanishing gradient issues. ReLU, on the other hand, stays linear for regions where

sigmoid and TanH are saturated. This way ReLU is more resistant to the vanishing/exploding gradient problem. There are also other activation functions that work well such as Leaky-ReLU, SILU, etc..

**Excessive Data augmentation** - It is not that having bigger rotations, and different kinds of flips/color adjustments are not good for accuracy. The data-augmentation needs to represent the goal or ideal data-set that represents that goal; some rotation is normal when detecting things from an image, but do we want to detect objects in a world where everything is up-side down? This will greatly reduce performance on the test set, as the training set and development set share augmentation. For the task of trying to get the best test accuracy within 10 epochs, having more data augmentation will surely slow how many patterns the model is able to learn within that time frame. It is therefore not surprising that the resulting test accuracy would decrease and hints to under-fitting. An architecture with a greater complexity could also be a solution (able to classify objects, even if they are upside down).

**Adam Optimizer with high learning rate** - Much like SGD with momentum and an adaptive learning rate per variable. If the adam optimizer has a too high learning rate, it would suffer from gradient steps overshooting and perform worse than not using said optimizer at all. This applies to most optimizers, but it was practical to employ this one, as it is adaptive, and less demanding of cumbersome tuning.

**Drop out** - We suspect that dropout would be a great regularization method to use for other network architectures. But specifically for the one we used, it was only damaging. This is probably because there is already regularization going on with the optimizer. The other reason could have to do with not having enough filters in the last convolution layer (only 64). An idea to try out would therefore be not to have L2 regularization, but double the filters on the last convolutional layer to 128 and have a dropout of 50% (each mapping would then be less critical if dropped).

**BatchNorm1D** - Interestingly applying BatchNorm1D only hurt performance on both models. This is hard to explain explicitly, but may have an intuitive explanation. The idea is that the feature extraction part of the CNN passes information forwards using mapped images in 2D. The classification section uses a fully connected neural network to interpret these features. One could say that these two sections of the CNN represent information in each of their language. By applying BatchNorm1D we disrupt the CNN image representation and lose vital information before the fully-connected portion has a chance of interpreting it. This is why model2 was designed to squeeze images to a 2x2 shape at the end of the pipeline, to smooth-en the

transition to the fully-connected portion.

### 3.d (Code implementation found in task3cd.py)

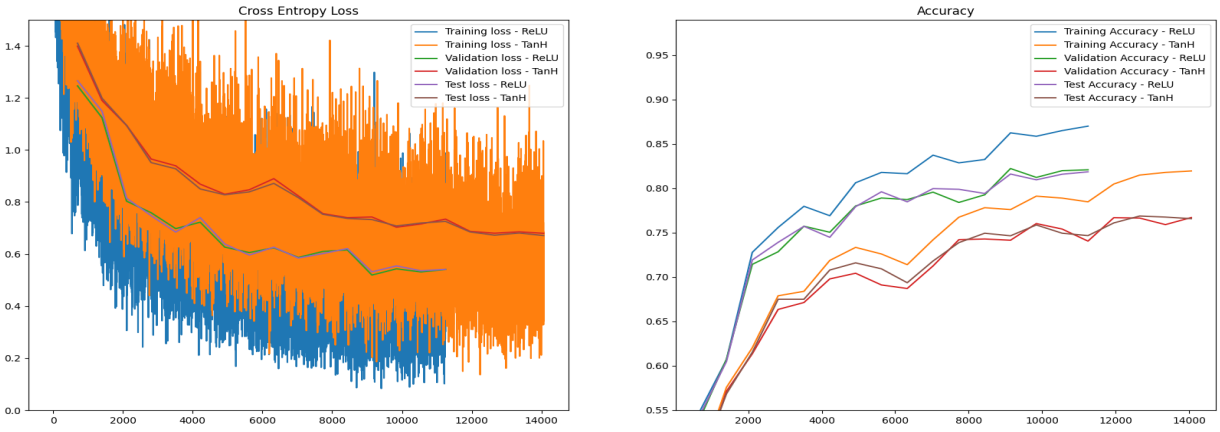


Figure 4: This is model 1 with ReLU as presented in task 3a, and with TanH instead as activation. ReLU gives a test accuracy of 81.87% (82.76% when training for longer) whilst TanH takes longer to converge and gives a test accuracy of 76.59%. So using ReLU gives a significant improvement in convergence and accuracy/loss.

### 3.e (Code implementation found in task3e.py)

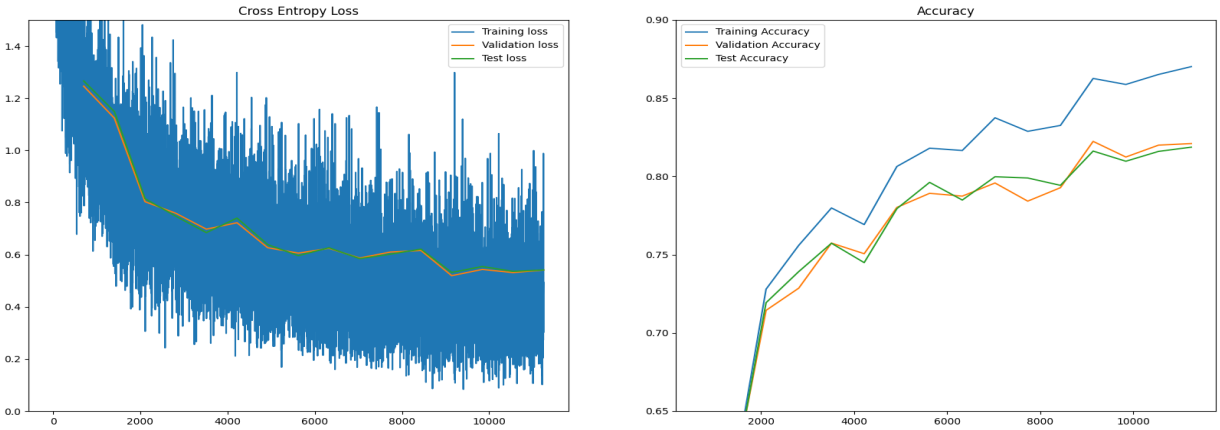


Figure 5: Plots of model 1 - similar to the one displayed in 3b and discussed in subtasks a-d.

Since our model was already over 80% accuracy, we tried to experiment and tweak to get an even better test accuracy than what we got. The methods that were tried was changing the activation function to LeakyReLU and SILU. Data augmentation in the form of Gaussian noise was also tried, as well as optimizer scheduling. These methods gave good performance, but none were substantially better. In conclusion if we insisted on further tweaking, there would be a possibility for small improvements, but are generally satisfied with the final result.

### 3.f

| Model 1 | | |
|---|---|---|
| - | **Loss** | **Accuracy** |
| Training | 0.1348 | 89.00% |
| Validation | 0.5240 | 82.34% |
| Test | 0.5177 | 82.76% |

This is a table of the final training/validation/test loss and accuracy for model 1. One can see that there is a gap forming between training loss and validation/test loss. This could be a sign of over-fitting, especially if we omit early stopping and continue training for many more epochs. We could perhaps reduce the gap with more aggressive L2-regularization or dropout. The fact that the validation and test accuracies are similar indicates that we have not distorted our data-set too much with augmentation; the training and validation sets still represent our original goals for the CNN.

# 4 Task 4 - Transfer Learning with ResNet

## 4.a (Code implementation found in task4a.py)

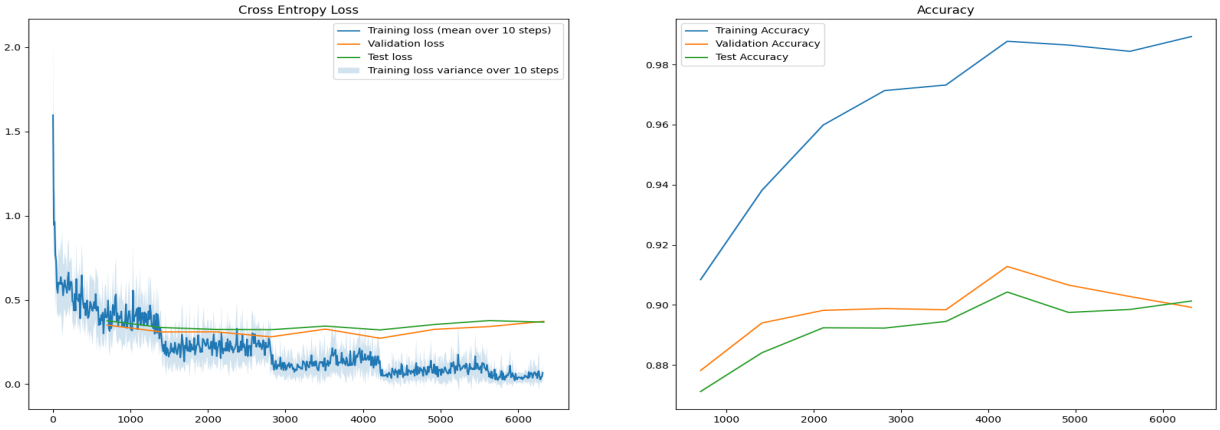


Figure 6: Test accuracy is 90.13% in the right figure.

| Final Data | | |
|---|---|---|
| - | **Loss** | **Accuracy** |
| Training | 0.0174 | 98.93% |
| Validation | 0.3730 | 89.92% |
| Test | 03683 | **90.13%** |

The Adam optimizer was used for the transfer learning model, with batch size 32, learning rate 5e-4, and no data augmentation (assuming transforms.resize() is not considered data augmentation).

**4.b**

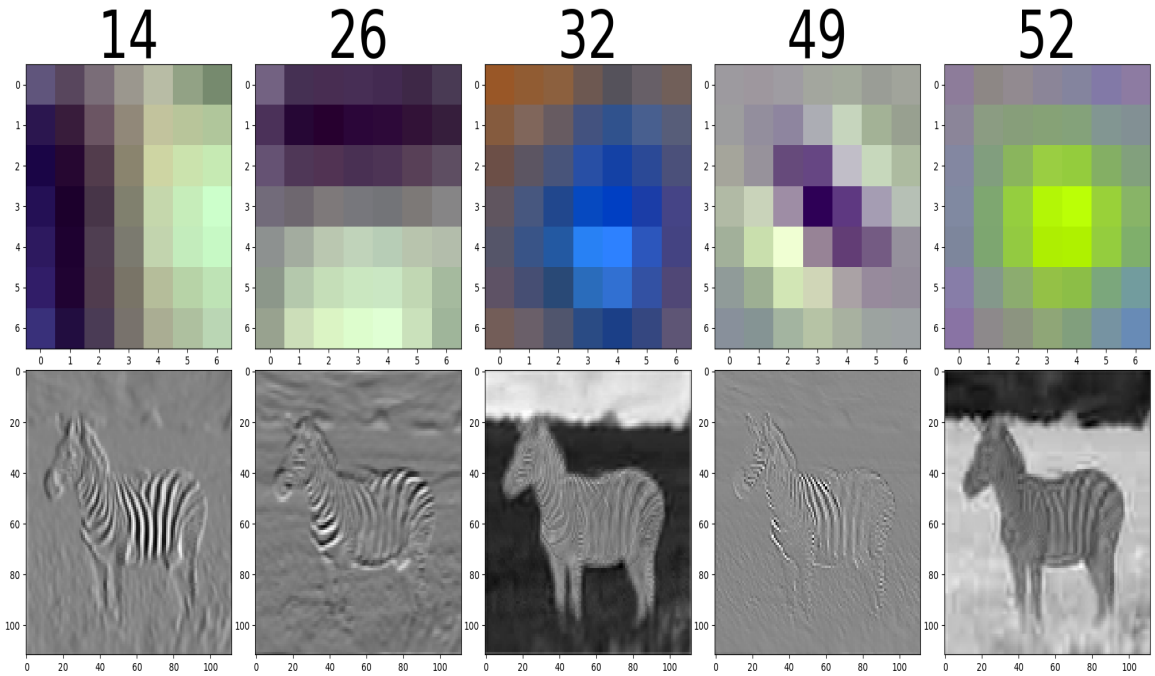


Figure 7: The visualisation of the filters and activation. With filters on the top row and the activation on the bottom row. Each column represents the filter position in the convolutional layer, and is specified on the top. This is taken from the first convolutional layer of ResNet18.

**Observation** - It seems as if the feature maps extracted from the first convolution layer tries to emphasize the micro features of the zebra. In one image the filter focuses on vertical edges, the next one focuses on horizontal edges, then abstract object shape and so on. In other words each filter provides a new lens for "seeing" the object.

### 4.c

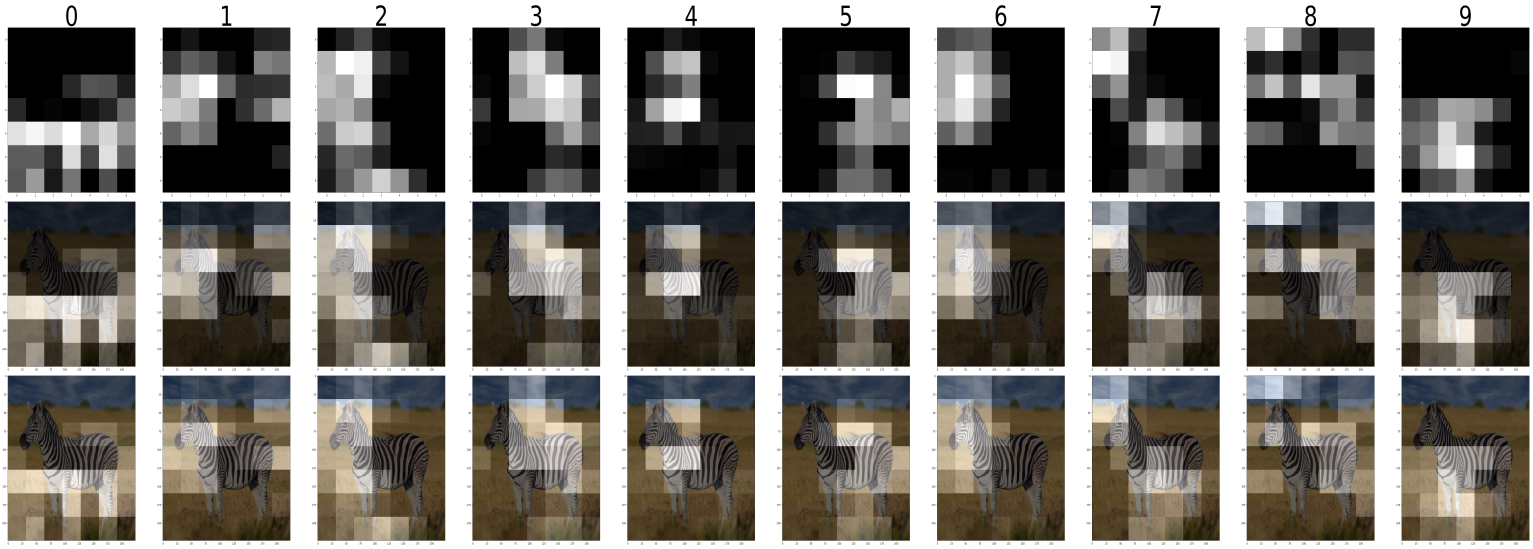


Figure 8: Visualisation of the 10 first layers of the last convolution layer of the ResNet18 model. Note that this is after it has been forwarded through the entire convolutional part of the network,including a normalization layer at the end. The last two rows has the particular activations from the last convolutional layer on top of the source image. The opacity of the source image is 50% in the last row. This better highlights the macro features that the last convolutional layer is extracting.

**Observation** - in contrast to 4b the last filter mappings seem to focus on high-level abstractions. Such as a pair of legs, a head or a body seen in feature map 0, 2 and 3 respectively. The highlighted regions on the pixel maps indicate these macro features.