Task 1

Task 1

a) Convolved image should be 3×5

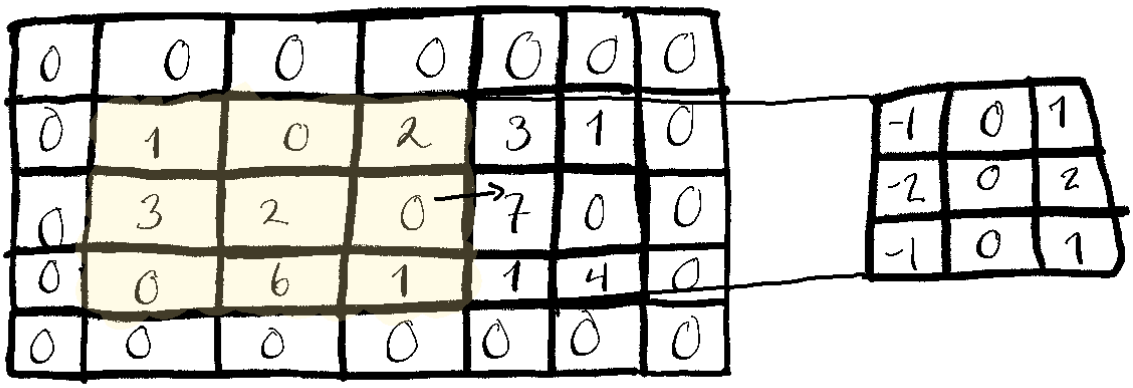
$$\Rightarrow H_2 = \frac{H_1 - F_H + 2P_H}{S_H} + 1 = \frac{3 - 3 + 2 \cdot P_H}{S_H} + 1 = 3$$

$$\Rightarrow \frac{2P_H}{S_H} = 2 \Rightarrow P_H = S_H$$

$$W_2 = \frac{(W_1 - F_W + 2P_W)}{S_W} + 1 = \frac{5 - 3 + 2P_W}{S_W} + 1 = 5$$

$$\Rightarrow \frac{2 + 2P_W}{S_W} = 4 \Rightarrow 1 + P_W = 2S_W$$

Let $S_H = S_W = 1$. Then $P_W = P_H = 1$ solves the task. Thus, we use a stride of 1, and zero-pad by 1 in both width & height-dimension.



a) We slide the filter over the padded image with stride 1 and obtain:

$-2 \cdot 1 =$ -2	$2 - 2 \cdot 2 + 3 =$ 1	$-3 \cdot 2 + 2 - 7 =$ -11	$2 \cdot 2 - 2 =$ 2	$2 \cdot 3 + 7 =$ 13
$-2 \cdot 2 - 6 =$ -10	$1 \cdot 1 - 2 + 2 \cdot 3 - 1 \cdot 1 =$ 4	$-3 \cdot 1 + 2 \cdot 2 - 2 \cdot 7 + 1 \cdot 6 - 1 \cdot 1 =$ -8	$1 \cdot 2 - 1 \cdot 1 + 1 \cdot 1 - 4 \cdot 1 =$ -2	$1 \cdot 3 + 2 \cdot 7 + 1 \cdot 1 =$ 18
$-2 \cdot 1 - 2 \cdot 6 =$ -14	$1 \cdot 3 - 2 \cdot 1 =$ 1	$1 \cdot 2 - 1 \cdot 1 + 2 \cdot 6 - 2 \cdot 1 =$ 5	$2 \cdot 1 - 2 \cdot 4 =$ -6	$1 \cdot 1 + 2 \cdot 1 =$ 9

b) Translational invariance is due to the max-pooling layer. In the maxpooling-operation we create a summary of a pixel-neighbourhood. Thus, if we shift the input slightly, the max value of the convolution output will remain the same.

c) $S_W = S_H = 1$, $F_W = F_H = 5$, $C_2 = 6$

$$H_2 = \frac{H_1 - F_H + 2P_H}{S_H} + 1 = H_1 - 5 + 2P_H + 1$$

want $H_1 = H_2 \Rightarrow 2P_H - 4 = 0 \Rightarrow P_H = \underline{2}$

$$W_2 = \frac{W_1 - F_W + 2P_W}{S_W} + 1 = W_1 - 5 + 2P_W + 1$$

$W_1 = W_2 \Rightarrow \underline{P_W = 2}$

$\Rightarrow \underline{\underline{P_H = P_W = 2}}$

d) $P = 0,\ S = 1,\ F_W = F_H = F$

$$H_2 = \frac{H_1 - F_H + 2P_H}{S_H} + 1 = H_1 - F + 1 = 504$$

$$\Rightarrow F = H_1 - 503$$

$$W_2 = \frac{W_1 - F_W + 2P_W}{S_W} + 1 = W_1 - F + 1 = 504$$

$$\Rightarrow F = W_1 - 503$$

$$H_1 = W_1 - 512$$

$$\Rightarrow F = 512 - 503 = \underline{9}$$

$\Rightarrow$ The kernels are of dimension $\underline{9 \times 9}$

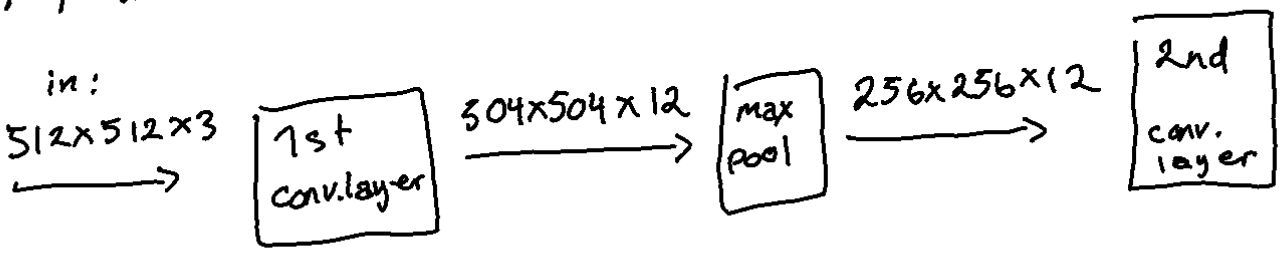
e) $F = 2,\ S = 2$

$$H_2 = \frac{(H_1 - F)}{S} + 1 = \frac{512 - 2}{2} + 1 = \underline{256}$$

$$W_2 = H_2 = 256$$

$\Rightarrow$ The pooled feature maps will be $\underline{256 \times 256}$

f) We have



$P = 0, S = 1$

$$H_2 = \left(\frac{H_1 - F_H + 2P_H}{S_H}\right) + 1 = H_1 - F_H = 256 - 3 + 1 = \underline{254}$$

$W_2 = H_2 = \underline{254}$

$\Rightarrow$ The size of the featuremaps in the 2nd layer is $\underline{254 \times 254}$

g) All filters are 5x5 with P=2, S=1

Max pool 2D - layer: S=2, 2x2

- Each filter has $F_H \times F_W \times C_1 = 5 \times 5 \times 3 = 75$ weights

Layer 1: $F_H \times F_W \times C_1 \times C_2 + b = 75 \cdot 32 + 32$
$= \underline{2432}$

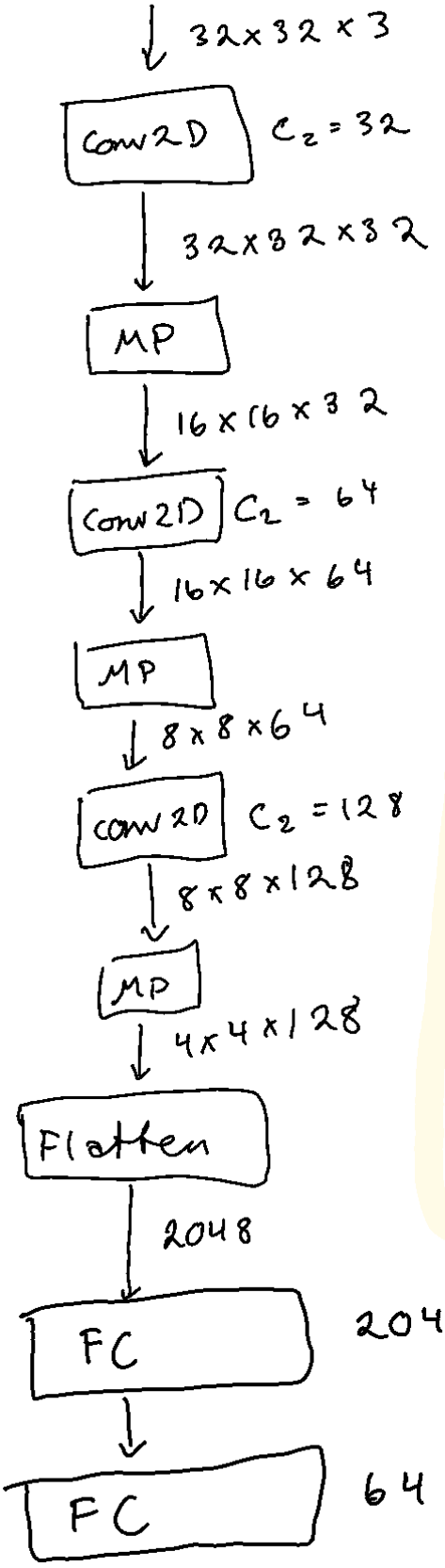
Layer 2: $5 \times 5 \times 32 \times 64 + 64 = \underline{51264}$

Layer 3: $5 \times 5 \times 64 \times 128 + 128 = \underline{204928}$

$\Rightarrow \sum params_{conv} = \underline{258624}$

fully-connected layers →

To find the dimensions of the output of the flatten-layer, we iterate through the layers



Result: in the fully connected layers we have

$i = 2048$

$j = 64$

$k = 10$

which gives

$$W_{FC} = i \cdot j + j \cdot k = 2048 \cdot 64 + 64 \cdot 10 = 131712$$

$$b_{FC} = j + k = 64 + 10 = 74$$

$$\Rightarrow \Sigma params_{FC} = 131786$$

The total number of parameters is then

$$\Sigma params_{conv} + \Sigma params_{FC} = 258624 + 131786 = 390410$$

Task 2

2a)

```
CUDA available:  True
Files already downloaded and verified
Files already downloaded and verified
ExampleModel(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
    (3): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
    (6): Conv2d(64, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
```

2b)

The final accuracies are:

```
Final training accuracy:  0.8796674964438123
Final validation accuracy:  0.7286
Final test accuracy:  0.716
```

Task 3

We create two models. The two have different properties compared to the one in task 2, which had this specification:

Layer	Layer Type	Number of Hidden Units / Number of Filters	Activation Function
1	Conv2D	32	ReLU
1	MaxPool2D	–	–
2	Conv2D	64	ReLU
2	MaxPool2D	–	–
3	Conv2D	128	ReLU
3	MaxPool2D	–	–
	Flatten	–	–
4	Fully-Connected	64	ReLU
5	Fully-Connected	10	Softmax

3. a)

When making both models, we experimented with the hyperparameters to get good working baseline networks. We found that

- the batch size of 64 seemed to work better than 32 or 128, so we stayed on 64 for both models
- increasing the learning rate to 0.1 and decreasing it to 0.01 made the performance worse, so we stayed at 0.05 for both models
- in model 1, we saw that both increasing and decreasing the stride for the kernels in the conv-layers and the pool layers made the performance worse, so we stayed at stride = 1 for the conv layers in and stride = 2 for the pool layers.
- in both models, we saw that changing the numbers of layers and neurons in the affine layers had little effect on the result, so we kept the affine layers from task 2 in both models.

Model 1:

- Same filtersize, padding & stride as in task 2 (5x5, 2, 1)
- Same number of filters in all layers, ([64, 64, 64])

Parameters that stay the same as task 2:

- SGD Optimizer
- Maxpooling of size 2x2, stride 2
- Learning rate = 0.05
- batch size = 64
- Same affine layers

The starting architecture for model 1:

Layer	Layer type	Number of hidden units/filters	Activation function
1	Conv2D	64	ReLU
1	MaxPool2D	-	-
2	Conv2D	64	ReLU
2	MaxPool2D	-	-
3	Conv2D	64	ReLU
3	MaxPool2D	-	-
Flatten			
Fully connected		64	ReLU

Layer	Layer type	Number of hidden units/filters	Activation function
Fully connected		10	Softax

Model 2:

- Smaller filtersize (4x4), padding = 1
- Strided convolutions, i.e. stride = 2, no maxpool

Parameters that stay the same as task 2:

- Same amount of filters, [32, 64, 128], as task 2
- SGD Optimizer
- Learning rate = 0.05
- batch size = 64
- Same affine layers

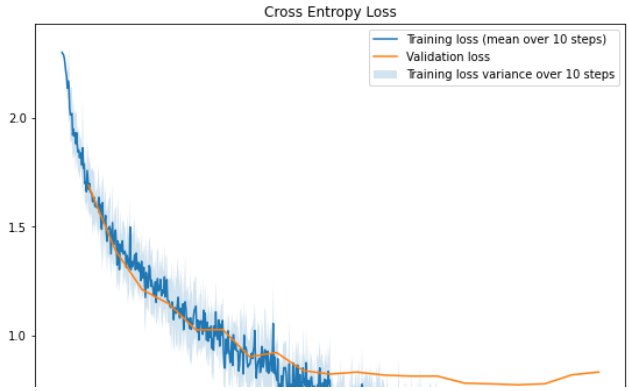
The starting architecture for model 2:

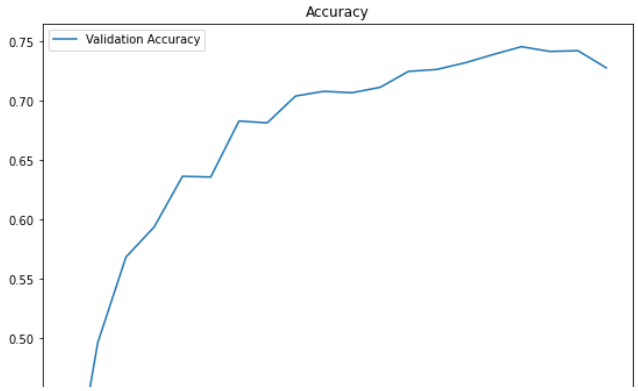
Layer	Layer type	Number of hidden units/filters	Activation function
1	Conv2D	32	ReLU
2	Conv2D	64	ReLU
3	Conv2D	128	ReLU
Flatten			
Fully connected		64	ReLU

3b)

Model 1, baseline

```
CUDA available:  True
Files already downloaded and verified
Files already downloaded and verified
ExampleModel(
  (activation): ReLU()
  (feature_extractor): Sequential(
    (0): Conv2d(3, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
    (3): Conv2d(64, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
    (6): Conv2d(64, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
  )
  (classifier): Sequential(
    (0): Linear(in_features=1024, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=10, bias=True)
  )
)
Early stop criteria met
Early stopping.
```

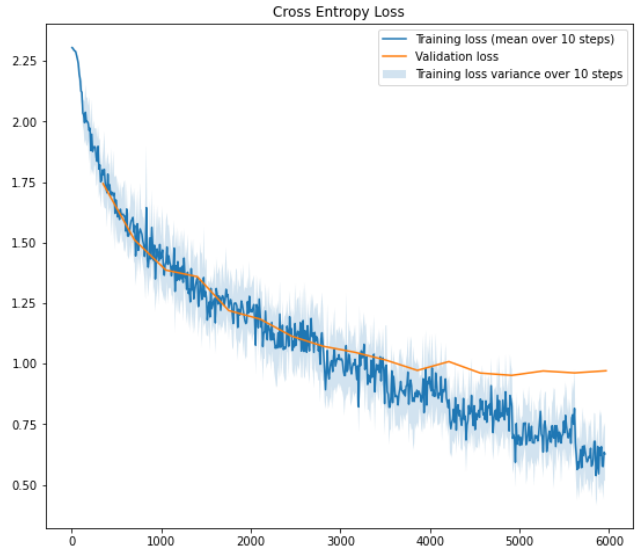


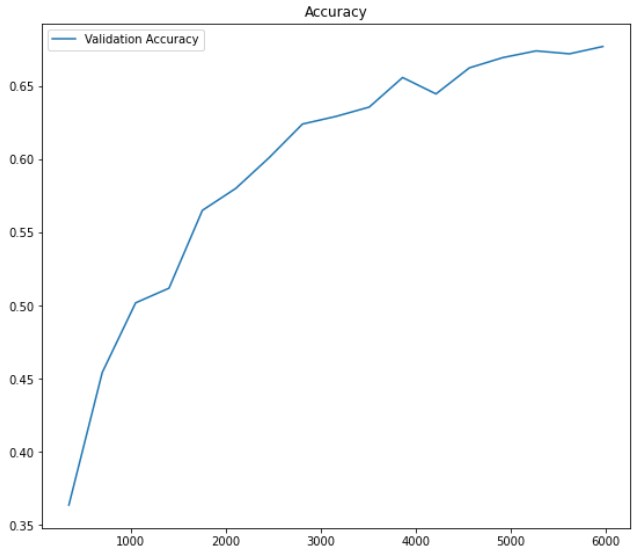



```
Model 1 baseline statistics:
Final training loss:  0.35760901351148
Final training accuracy:  0.8754000711237553
Final validation accuracy:  0.728
Final test accuracy:  0.7323
```

Model 2, baseline

```
CUDA available:  True
Files already downloaded and verified
Files already downloaded and verified
ExampleModel(
  (activation): ReLU()
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (5): ReLU()
  )
  (classifier): Sequential(
    (0): Linear(in_features=2048, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=10, bias=True)
  )
)
Early stop criteria met
Early stopping.
```






```
Model 2 baseline statistics:
Final training loss:  0.5249685832056856
Final training accuracy:  0.8177898293029872
Final validation accuracy:  0.6772
Final test accuracy:  0.6695
```

3c)

Tricks will will try to implement on both models:

- Data augmentation
- Dropout
- Adam Optimizer
- Batch normalization
- Kaiming Normal weight initialization
- The "leaky ReLU", an alternative activation function

Note: we add the changes individually to the baseline models, not incrementally (e. g. when applying dropout, we disable data augmentation from the previous step) to find out the impact of each individual trick. Note that this may lead to neglection of good combinations of tricks, but we try to look for these when improving upon the model later.

- Adding data augmentation

- When adding data augmentation, we double the training set with augmented pictures. We use random cropping and flipping of the images.

Result:

```
Model 1 statistics with data augmentation:
Final training loss:  0.6943914008725588
Final training accuracy:  0.7579903093883357
Final validation accuracy:  0.6941
Final test accuracy:  0.769
```

```
Model 2 statistics with data augmentation:
```

```
Final training loss:  0.7802174814140813
Final training accuracy:  0.7285739687055477
Final validation accuracy:  0.6488
Final test accuracy:  0.7182
```

Data augmentation improves both models fairly significantly. It is worth noting that the training accuracy resembles the test accuracy in both cases, which indicates good generalization - which is what we hope for when adding augmented data. We can also note that the training loss has increased compared to the baseline, which is natural considering there is more tricky data to train on.

Adding dropout

- We try adding dropout after every convolutional layer. We use a dropout-probability of 0.25.

Result:

```
Model 1 statistics with dropout:
Final training loss:  0.6310074235687554
Final training accuracy:  0.7751155761024182
Final validation accuracy:  0.7258
Final test accuracy:  0.7174
```

```
Model 2 statistics using dropout:
Final training loss:  0.8703448267105124
Final training accuracy:  0.690478307254623
Final validation accuracy:  0.6292
Final test accuracy:  0.6256
```

We see that dropout in the convolutional layers are making the performance worse. When we research this, it becomes clear that this is expected, as "adding dropout to convolutional layers seems to amount to multiplying Bernoulli noise into the feature maps of the network", according to this article https://towardsdatascience.com/dropout-on-convolutional-layers-is-weird-5c6ab14f19b2 . Seemingly, adding dropout to convolutional layers has a fundamentally different effect than adding it to fully-connected layers.

Using the Adam Optimizer

- (extension of stochastic gradient descent, using socalled "adaptive moment estimation") with L2 regularization with **lambda** = 1e-5 and a learning rate of 1e-3.

Result:

```
Model 1 statistics using Adam optimizer:
Final training loss:  0.5385317261069801
Final training accuracy:  0.8107441322901849
Final validation accuracy:  0.708
Final test accuracy:  0.7075
```

```
Model 2 statistics using Adam optimizer:
Final training loss:  0.4346224338155383
Final training accuracy:  0.8560855263157895
Final validation accuracy:  0.686
Final test accuracy:  0.6918
```

We see that the adam optimizer improves test accuracy for model 2, while it makes the performance worse for model 1. Tried to research this, but it is hard to say why. Interesting result, though! The main difference between the models is the maxpool-layer and the way the filters are distributed, so we assume it has something to do with this.

Adding Batch normalization

- We add 2D batch normalization after every conv-layer and 1D batch normalization in the affine layers.

Result:

```
Model 1 statistics with batch normalization:
Final training loss:  0.4412491604749373
Final training accuracy:  0.8513958036984353
Final validation accuracy:  0.756
Final test accuracy:  0.7491
```

```
Model 2 statistics with batch normalization:
Final training loss:  0.3515844049189203
Final training accuracy:  0.8864464793741109
Final validation accuracy:  0.6982
Final test accuracy:  0.6977
```

Both networks do train faster. For example, for model 1, the early stop criteria kicked in after about half of the training steps, compared to the baseline model.

Using Kaiming Normal weight initialization

Looking at the current pytorch module in github, we see that the current initialization method for conv2D layer weights is the uniform Kaiming method.

```
def reset_parameters(self) -> None:
    init.kaiming_uniform_(self.weight, a=math.sqrt(5))
    if self.bias is not None:
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in)
        init.uniform_(self.bias, -bound, bound)
```

(retrieved from https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/conv.py)

This method implements the following:

> Fills the input *Tensor* with values according to the method described in *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* - He, K. et al. (2015), using a uniform distribution. The resulting tensor will have values sampled from $\mathcal{U}(-\text{bound}, \text{bound})$ where
>
> $$\text{bound} = \text{gain} \times \sqrt{\frac{3}{\text{fan\_mode}}}$$
>
> Also known as He initialization.

(retrieved from the pytorch documentation, https://pytorch.org/docs/stable/nn.init.html)

We wish to try to see if the method of Kaiming Normal initializing can do good for our models. This is a widely used method combined with the ReLU activation, and it works like this:

> Fills the input *Tensor* with values according to the method described in *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* - He, K. et al. (2015), using a normal distribution. The resulting tensor will have values sampled from $\mathcal{N}(0, \text{std}^2)$ where
>
> $$\text{std} = \frac{\text{gain}}{\sqrt{\text{fan\_mode}}}$$
>
> Also known as He initialization.

(retrieved from the pytorch documentation, https://pytorch.org/docs/stable/nn.init.html)

Result:

```
Model 1 statistics with kaiming normal initialization:
Final training loss:  0.7329840872908384
Final training accuracy:  0.7462660028449503
Final validation accuracy:  0.6694
Final test accuracy:  0.6667
```

```
Model 2 statistics with kaiming normal initialization:
Final training loss:  0.5780047617884483
Final training accuracy:  0.8044541251778093
Final validation accuracy:  0.6404
Final test accuracy:  0.6447
```

We see that the performance becomes worse. Why this happens is hard to say, as there is still an ongoing discussion on wich initialization is best (see e.g. https://datascience.stackexchange.com/questions/13061/when-to-use-he-or-glorot-normal-initialization-over-uniform-init-and-what-are). We do however notice that the normal initialization is used mostly for very deep networks, and ours is not very deep. We conclude with sticking to the uniform kaiming.

Using the "leaky ReLU", an alternative activation function

This activation function is almost identical to the original ReLU, except that it has a tiny negative slope which lets negative values pass with a very small gain.

From the documentation:

Applies the element-wise function:

$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative\_slope} * \min(0, x)$$

or

$$\text{LeakyRELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases}$$

We use 0.1 as value for the negative slope.

Result:

```
Model 1 statistics with leaky relu:
Final training loss:  0.2952378633224099
Final training accuracy:  0.8983152560455192
Final validation accuracy:  0.7444
Final test accuracy:  0.7439
```

```
Model 2 statistics with leaky relu:
Final training loss:  0.5564122333126421
Final training accuracy:  0.8061655405405406
Final validation accuracy:  0.6708
Final test accuracy:  0.6642
```

We see that the leaky ReLU improves performance for model 1, while it worsenes performance for model 2.

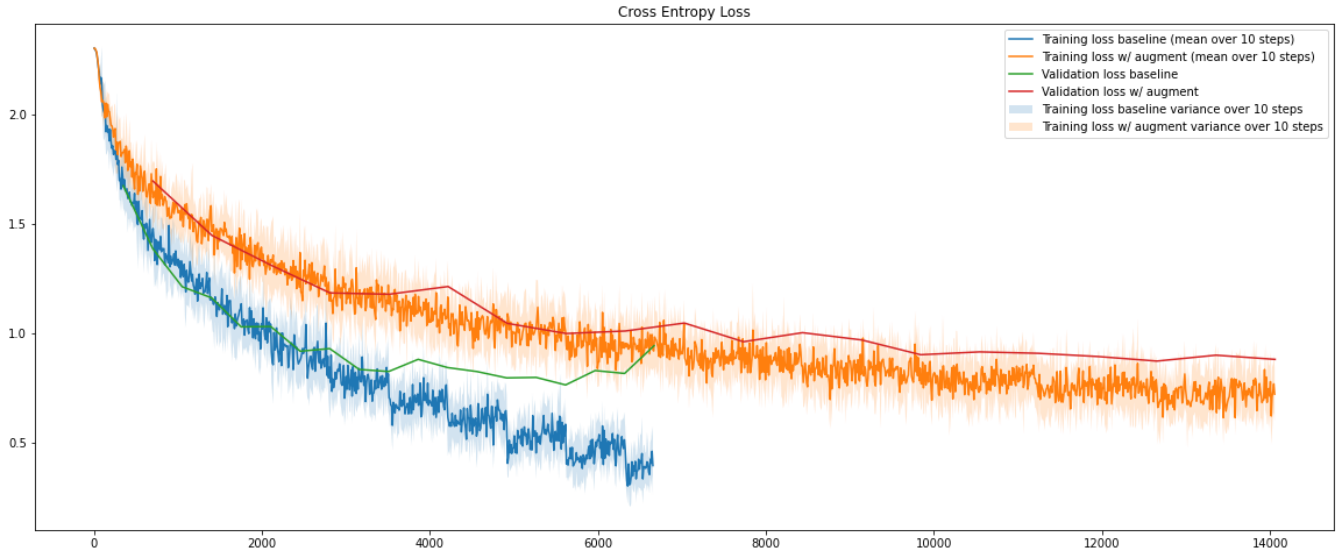
Resulting overview of the tricks, the test accuracy and the difference from the baseline model (the tricks that improved the test accuracy are highlighted):

Trick	Modell 1	+/-	Modell 2	+/-
Baseline	0.7323		0.6695	
Augment	0.769	+0.0367	0.7182	+0.0487
Dropout	0.7174	-0.0149	0.6256	-0.0439
ADAM	0.7075	-0.0248	0.6918	+0.0223
Batch-norm	0.7491	+0.0168	0.6977	+0.0282
normal init	0.6667	-0.0656	0.6447	-0.0248
leaky ReLU	0.7439	+0.0116	0.6642	-0.0053

Note: as we had many features to test, we didnt get to experiment with all of the parameters of the different methods. It may be that e.g. another value for the negative slope of the leaky relu would have been more effective.

3d)

The best model is clearly model 1. We see that the trick that had the largest effect is the data augmentation. Plotting with and without:



As stated when we tried the data augmentation (see 3c), we see that the training- and validation loss increases when

3e)

To get model 1 to 80% test accuracy, we add the tricks that yielded better results, i. e, data augmentation, batch normalization and the leaky relu.

Result:

```
Model 1 statistics with all improvements:
Final training loss:  0.7208784416763067
Final training accuracy:  0.7489775960170697
Final validation accuracy:  0.6982
Final test accuracy:  0.7752
```

Not quite there yet. Using leaky-ReLU with slope 0.2:

```
Model 1 statistics with all improvements:
Final training loss:  0.6788692498741244
Final training accuracy:  0.7647359530583214
Final validation accuracy:  0.7074
Final test accuracy:  0.7785
```

We add one layer of 64 filters:

```
Model 1 statistics with all improvements:
Final training loss:  0.6312451979852842
Final training accuracy:  0.7808054765291608
Final validation accuracy:  0.7212
Final test accuracy:  0.7885
```

Then we change the size of the filters to 128:

```
Model 1 statistics with all improvements:
```

```
Final training loss:  0.5308500185057754
Final training accuracy:  0.8185899715504978
Final validation accuracy:  0.7442
Final test accuracy:  0.8046
```

- There we are.

  Loss and accuracy-plot of the best model:

```
CUDA available:  True
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
ExampleModel(
  (activation): LeakyReLU(negative_slope=0.2)
  (feature_extractor): Sequential(
    (0): Conv2d(3, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_st
    (2): LeakyReLU(negative_slope=0.2)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
    (4): Conv2d(128, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_st
    (6): LeakyReLU(negative_slope=0.2)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fal
    (8): Conv2d(128, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (9): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_st
```

The resulting best model that made it past 80% test accuracy:

Layer	Layer type	Number of hidden units/filters	Activation function
1	Conv2D	128	LeakyReLU
1	MaxPool2D	-	-
2	Conv2D	128	LeakyReLU
2	MaxPool2D	-	-
3	Conv2D	128	LeakyReLU
3	MaxPool2D	-	-
Flatten			
Fully connected		64	ReLU
Fully connected		10	Softax

using, in addition:

- data augmentation
- batch normalization
- leaky relu with slope = 0.2

3f)

For the best model, we see that the final training accuracy is very similar to the final test accuracy. The training loss and validation loss also seems to converge in the same pace, at the same values, which indicates that we do not have significant over- or underfitting.

Task 4

Task 4a)

Hyperparameters used:

epochs = 10, batch_size = 32, learning_rate = 5e-4, early_stop_count = 4, dataloaders4 = load_cifar10(batch_size)

Optimizer: Adam Optimizer

No data augmentation used

```
1.7.1+cu101
CUDA available: True
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to data/cifa
```

170500096/? [00:20<00:00, 100546640.32it/s]

```
Extracting data/cifar10/cifar-10-python.tar.gz to data/cifar10
Files already downloaded and verified
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /roc
```

100% 44.7M/44.7M [00:10<00:00, 4.31MB/s]

```
Model(
  (model): ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bi
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mc
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
      )
```

Achieving a final test accuracy of: 0.8968

The three different accuracy all accomplishes accuracy over 88% given as a minimum in the task.

```
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
```

Task 4b)

```
      (0): BasicBlock(
```

```
Image shape: (224, 224)
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_sta
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
```

Chose to make the activation of the filter gray, matching the figure 2 in the assignments text.

From the filter activations it is possible to still see contours of the zebra. This is expected as the first convolutional layer recognize lines and shapes in the image. This is seen as the shape and also the stripes of the zebra is quite recognizable in all of the activations. The different filters focus on highlighted lines, background, foreground, edges and so on, giving different outcomes on the activations of the corresponding filter.

```
      (downsample): Sequential(
```

Task 4c)

```
    )
```

```
Last conv layer: Sequential(
  (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=Fal
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (2): ReLU(inplace=True)
  (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False
  (4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running
    )
  )
  (5): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runnin
    )
  )
  (6): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runnin
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (relu): ReLU(inplace=True)
```

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
```

The filter activation now gives out a result which do not capture any details in the images and there is nothing in the image saying it is a zebra. This is quite expected since the model abstacts the features from the image into more general concepts as we progress deeper into our model. The last layer in restnet18 is therefore not expected to output a visualization that says something about what the original image contained.

```
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runnin


    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runnin
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runnin
    )
  )
)
Activation shape: torch.Size([1, 512, 7, 7])
```