Network Analysis
Lab assignment 1: Image recognition using deep networks

Group 2
Aline D'Intino Gonçalves (5132185)
Amaru Sinchico Arias (5022096)
Priyanka Kumar (7319533)
Sybren Bouwman (5098386)

Table of contents

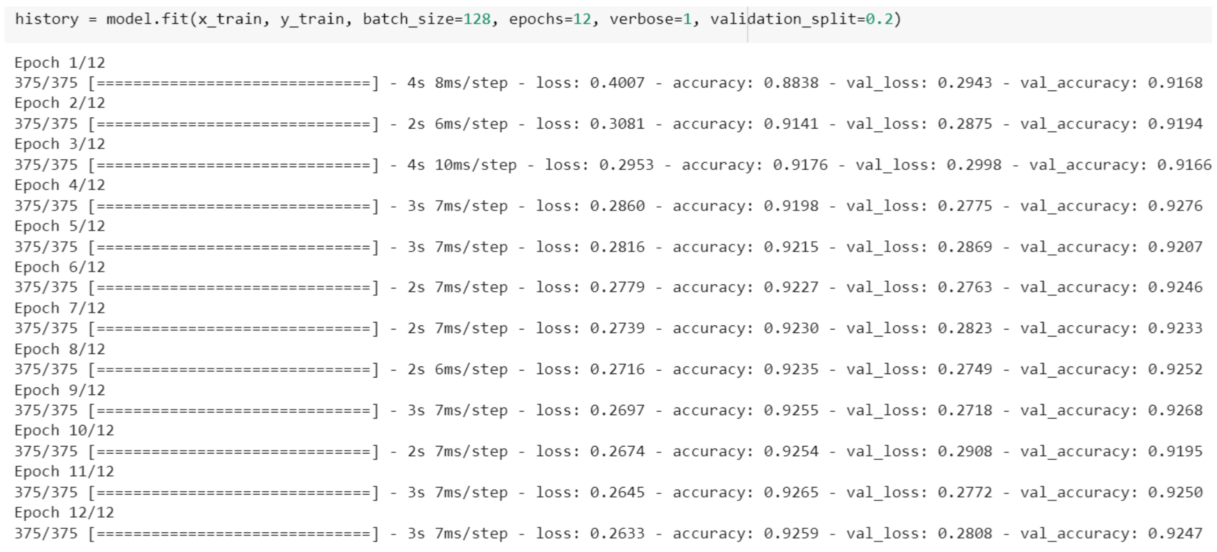
Exercise one: Identifying handwritten numbers

Question 1 (3 points)
Discuss with your group, then describe to your teacher, a list of applications where automatic recognition of hand-written numbers would be useful.

List of applications where automatic recognition of hand-written numbers would be useful:
- Making notes on the computer can quite be a hassle, when you want to store the notes on the computer (especially when formulas are involved). An automatic recognition of hand-written numbers would be useful to transfer the hand-written notes (of formulas) to the computer.
- Digital scanner: text or digit recognition and conversion to digital media (.doc .pdf)
- Automatic scanning of the hand-written student numbers and grades of exam-papers, to enter the grades of students in osiris.
- Read the hand-written phone number of forms or documents, to enter it automatically in the system, for example patient phone numbers that are written on paper forms.
- Automatic scanning of hand-written numbers on documents or forms when applying for a passport or ID-card, like birth-date, document-number, postal code, phone number etc.
- Scan the hand-written product numbers on papers or forms and then automatically select products in the warehouse or factory.
- Scan hand-written numbers of quiz or forms, this can be seen as automatic data-entry
- Maybe also in some countries, scan/recognize hand-written numbers of car-number-plates of documents or forms for administration purposes.
- Hand-written postal codes and house numbers on documents for administration purposes.

Question 2 (5 points)
Show your teacher the text from your console, with how long it took for each epoch to run and the training performance history.

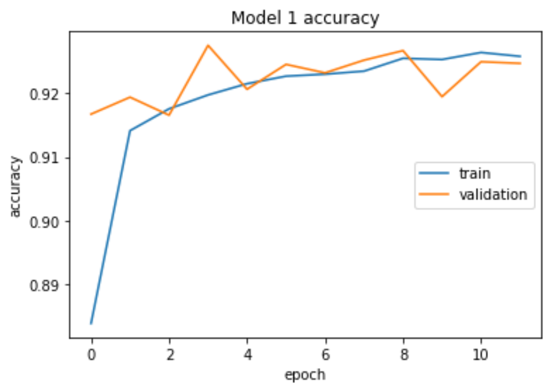
The following text is the output of the model fit, for each epoch information about how long it took to run and the training performance is given:

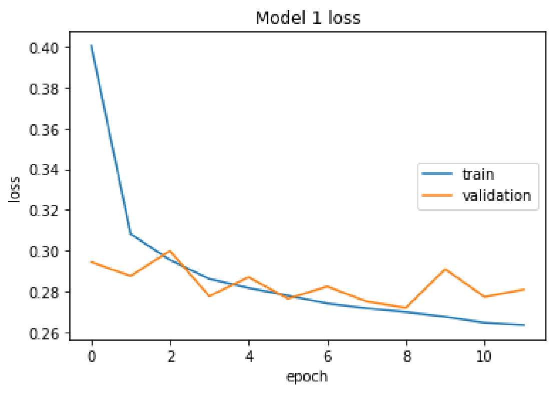


```
history = model.fit(x_train, y_train, batch_size=128, epochs=12, verbose=1, validation_split=0.2)

Epoch 1/12
375/375 [==============================] - 4s 8ms/step - loss: 0.4007 - accuracy: 0.8838 - val_loss: 0.2943 - val_accuracy: 0.9168
Epoch 2/12
375/375 [==============================] - 2s 6ms/step - loss: 0.3081 - accuracy: 0.9141 - val_loss: 0.2875 - val_accuracy: 0.9194
Epoch 3/12
375/375 [==============================] - 4s 10ms/step - loss: 0.2953 - accuracy: 0.9176 - val_loss: 0.2998 - val_accuracy: 0.9166
Epoch 4/12
375/375 [==============================] - 3s 7ms/step - loss: 0.2860 - accuracy: 0.9198 - val_loss: 0.2775 - val_accuracy: 0.9276
Epoch 5/12
375/375 [==============================] - 3s 7ms/step - loss: 0.2816 - accuracy: 0.9215 - val_loss: 0.2869 - val_accuracy: 0.9207
Epoch 6/12
375/375 [==============================] - 2s 7ms/step - loss: 0.2779 - accuracy: 0.9227 - val_loss: 0.2763 - val_accuracy: 0.9246
Epoch 7/12
375/375 [==============================] - 2s 7ms/step - loss: 0.2739 - accuracy: 0.9230 - val_loss: 0.2823 - val_accuracy: 0.9233
Epoch 8/12
375/375 [==============================] - 2s 6ms/step - loss: 0.2716 - accuracy: 0.9235 - val_loss: 0.2749 - val_accuracy: 0.9252
Epoch 9/12
375/375 [==============================] - 3s 7ms/step - loss: 0.2697 - accuracy: 0.9255 - val_loss: 0.2718 - val_accuracy: 0.9268
Epoch 10/12
375/375 [==============================] - 2s 7ms/step - loss: 0.2674 - accuracy: 0.9254 - val_loss: 0.2908 - val_accuracy: 0.9195
Epoch 11/12
375/375 [==============================] - 3s 7ms/step - loss: 0.2645 - accuracy: 0.9265 - val_loss: 0.2772 - val_accuracy: 0.9250
Epoch 12/12
375/375 [==============================] - 3s 7ms/step - loss: 0.2633 - accuracy: 0.9259 - val_loss: 0.2808 - val_accuracy: 0.9247
```

Question 3 (3 points)
Plot the training history and show this to your teacher.

The first plot shows how the accuracy progresses and the second plot shows how the loss (error) progresses differently across the epochs for the training and validation sets.





Question 4 (5 points)
Discuss with your group, then describe to your teacher how the accuracy on the training and validation sets progress differently across epochs, and what this tells us about the generalization of the model.

For model 1 we passed our 784 input units (flattened pixels) into a 256-unit fully connected hidden layer. This in turn has fed into our label (in the output) layer, whose activation is a softmax function to give the probability that each image is each digit. We then initialize model 1 using the keras function keras.Sequential(). After this the fully-connected layers are defined by the keras function keras.layers.Dense(). The softmax function is a generalization of the logistic function to multiple dimensions. This function is often used as the last activation function of a neural network. This is done to normalize the output of the neural network and to predict the probabilities that are associated with a multinoulli distribution.[1]

The generalization of the neural network, means how good our model is at learning from the given training set and applying the learned information on the validation set (on the data that is not used for training). If the neural network performs well on the validation set (that is not used for training), then we can say that the model is generalized well on the given data. We have plot loss (or error) vs. epoch and accuracy vs. epoch in two graphs (see question 3). For a good model we expect the loss to decrease and the accuracy to increase as the number of epochs increases. Also, we expect both loss and accuracy to stabilize after some point. As a result, the learning curve (the lines in the graphs) will get better and better. This will lead to convergence of the training and validation lines (for a good model with no overfitting).

The accuracy of training and the validation set progress, in our model, differently across the epochs:

- The training accuracy starts from 0.8838 (see epoch 0 in the graph in question 3) and increases steadily until 0.9255 (see epoch 8), then decreases a little to 0.9254 (see epoch 9), after this the accuracy increases to 0.9265 (see epoch 10) and then decreases to 0.9259 in the last epoch.
- The validation accuracy starts higher than the training accuracy, this time the accuracy starts from 0.9168 (see epoch 0 in the graph above). The validation accuracy fluctuates, giving some increases and decreases in the accuracy. For example, the validation accuracy increases to 0.9194 (see epoch 1), then decreases to 0.9166 (see epoch 2), after this the accuracy increases to 0.9276 (see epoch 3), then decreases to 0.9207 (see epoch 4), etc. This pattern of fluctuation is visible for each epoch, until the last epoch (with a value of 0.9247) that is slightly close to the accuracy of the training data. There is no stabilization in the validation accuracy of model 1.

The more epochs the model fit has, the more the model has 'seen' the data. In that case we need to be cautious of overfitting the data. Our model shows that the accuracy of the training set is overall lower than the accuracy of the validation set, but the model didn't stabilize on the validation set (as can

[1] Adefami, O. (2022, January 5). A Subtle Introduction to The Softmax Function - Kernel-X. Medium. Retrieved 1 March 2022, from https://medium.com/kernel-x/a-subtle-introduction-to-the-softmax-function-a56591711d4

seen from the fluctuations in the accuracy). The accuracy of the training and validation sets convergence in the last epoch, so it can be said that there is no case of overfitting. We also can see the number of loss (errors) fluctuations, and don’t see stabilization. The model ran quickly because of its simple structure. The downside of this simple structure is that the model has a limited ability to generalize to new data (validation set) that it is not trained on, because of the fluctuations and no stabilization in the line.

Question 5 (2 points)
Show your teacher what values you get for the model’s accuracy and loss.

```
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print('loss:', loss)
print('accuracy:', accuracy)
```

```
loss: 0.28929948806762695
accuracy: 0.9243999719619751
```

Question 6 (5 points)
Discuss with your group, then describe to your teacher, whether this accuracy is sufficient for some uses of automatic hand-written digit classification.

In general accuracy measures how well our model predicts by comparing the model predictions with the true values in terms of percentage. The accuracy of a model is usually determined after the model parameters are tuned and no further learning is taking place. The validation data is then fed into the model and the number of mistakes (zero-one loss) are recorded, after comparing it to the true targets. After this, the percentage of misclassification is calculated.

An accuracy of 0.9243 seems to be quite high. We think that the accuracy has to be at least above 99% (0.99), to be able to use the model for some uses of automatic hand-written digit classification. We assume this because the potential errors could have extreme effects, depending on what the model is used for. Beside, the accuracy of 0.9243 is only on the validation set and might hint to some overfitting of the training set. In that case, the accuracy on the validation set might be much lower. The lower the loss value, the better our model works. In general, the lower the loss, the better a model is (unless the model has overfitted to the training data). Our model gave a high accuracy and low loss, so the model makes small errors on just some of the data.

Based on the application, the model could be sufficient but also not. The sufficient application of the model depends on the case in which the model is applied. An accuracy of 0.9243 is for example not sufficient for automatic hand-written digit classification if it is used for reading hand-written student numbers and grades of documents or patient files, because this could lead to small errors in recognizing the hand-written digits using this model. This has quite a big impact, because the grades determine if the students have passed or failed an exam. In general this model is not sufficient enough for some uses of automatic hand-written digit classification, as there is a change of small erreros. These small errors need human validation, so humans will need to check if the predictions are right.

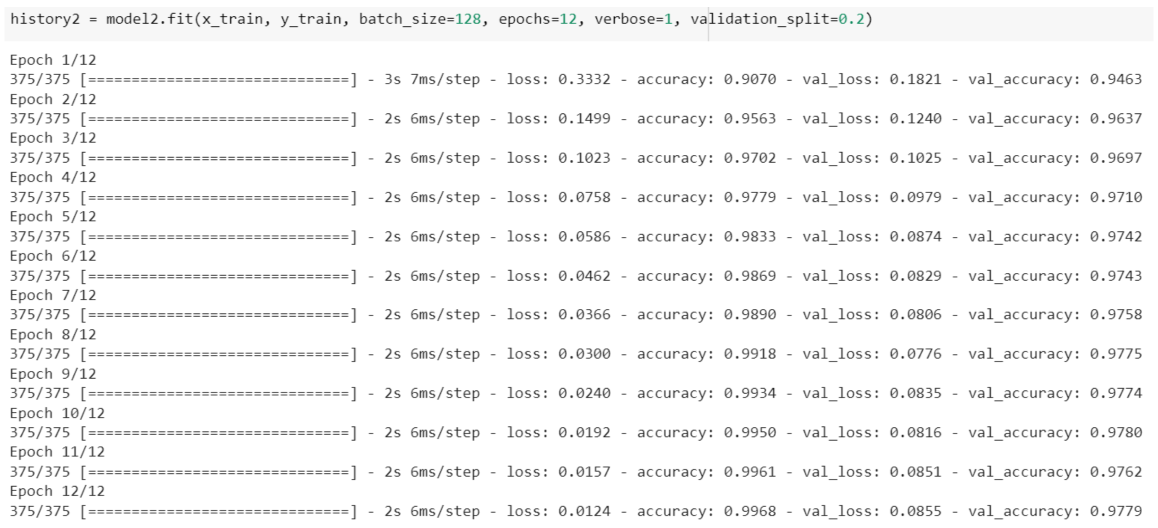
Question 7 (5 points)
In the previous model, we did not specify an activation function for our hidden layer, so it used the default linear activation. Discuss with your group, then describe to your teacher, how linear activation of units limits the possible computations this model can perform.

- A linear function is a straight-line function where the activation is proportional to the input, the so-called weighted sum of the input. When applying this in all the nodes the activation function will then work like a linear regression. The final layer of the model will work as a linear function of the first layer of the model. This will lead to the problem of gradient descent when the differentiation is done. This has a constant output and the output will not be confined between any range. This could lead to problems during the backpropagation. During the backpropagation the rate of the change of error is constant, this will ruin the output and the logic of backpropagation.[2]

- Backpropagation: It's not possible to use backpropagation as the derivative of the function, it is constant and has no relation to the input. So, backpropagation can then not be applied to find how the neural weights should have been changed, if errors are found. This is because of the simple notion that gradients are no longer dependent on the input values, because they are always the same. There is then no way to validate the performance of the model to improve the model.
- In deep networks, activation functions need to be calculated a lot of times (millions) as they are applied after every layer in the network. All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. A linear activation function turns the neural network into just one layer. Basically, a linear activation function turns the neural network into just one layer. So the model becomes a linear model, because all the layers can be considered as a linear combination of individual linear layers[3]. So for example, the second layer of the model is the output of a linear function of the previous input layer. This means that using a non-linear activation function, the model can learn more complex structures in the data.

Question 8 (2 points)
Now make a similar model with a rectified activation in the first hidden layer, by adding the extra argument: activation = 'relu' to the model definition for this layer. Then compile, fit and evaluate the model. Plot the training history and show it to your teacher.

The following text is the output of the model fit, for each epoch information about how long it took to run and the training performance is given:



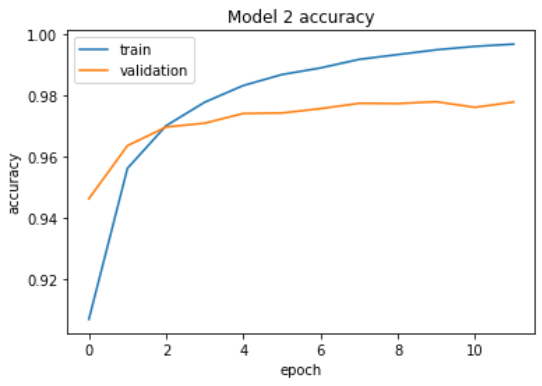
```
history2 = model2.fit(x_train, y_train, batch_size=128, epochs=12, verbose=1, validation_split=0.2)
```

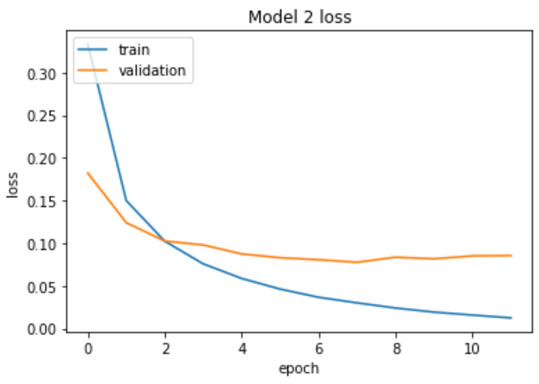
```
Epoch 1/12
375/375 [==============================] - 3s 7ms/step - loss: 0.3332 - accuracy: 0.9070 - val_loss: 0.1821 - val_accuracy: 0.9463
Epoch 2/12
375/375 [==============================] - 2s 6ms/step - loss: 0.1499 - accuracy: 0.9563 - val_loss: 0.1240 - val_accuracy: 0.9637
Epoch 3/12
375/375 [==============================] - 2s 6ms/step - loss: 0.1023 - accuracy: 0.9702 - val_loss: 0.1025 - val_accuracy: 0.9697
Epoch 4/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0758 - accuracy: 0.9779 - val_loss: 0.0979 - val_accuracy: 0.9710
Epoch 5/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0586 - accuracy: 0.9833 - val_loss: 0.0874 - val_accuracy: 0.9742
Epoch 6/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0462 - accuracy: 0.9869 - val_loss: 0.0829 - val_accuracy: 0.9743
Epoch 7/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0366 - accuracy: 0.9890 - val_loss: 0.0806 - val_accuracy: 0.9758
Epoch 8/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0300 - accuracy: 0.9918 - val_loss: 0.0776 - val_accuracy: 0.9775
Epoch 9/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0240 - accuracy: 0.9934 - val_loss: 0.0835 - val_accuracy: 0.9774
Epoch 10/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0192 - accuracy: 0.9950 - val_loss: 0.0816 - val_accuracy: 0.9780
Epoch 11/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0157 - accuracy: 0.9961 - val_loss: 0.0851 - val_accuracy: 0.9762
Epoch 12/12
375/375 [==============================] - 2s 6ms/step - loss: 0.0124 - accuracy: 0.9968 - val_loss: 0.0855 - val_accuracy: 0.9779
```

The first plot shows how the accuracy progresses and the second plot shows how the loss (error) progresses differently across the epochs for the training and validation sets.

[2] Panneerselvam, L. (2021, April 14). Activation Functions | What are Activation Functions? Analytics Vidhya. Retrieved 1 March 2022, from https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-guide/

[3] Versloot, C. (2022, February 14). Why you shouldn't use a linear activation function. GitHub. Retrieved 1 March 2022, from https://github.com/christianversloot/machine-learning-articles/blob/main/why-you-shouldnt-use-a-linear-activation-function.md





Question 9 (5 points)
Discuss with your group, then describe to your teacher, how this training history differs from the previous model, for the training and validation sets. Describe what this tells us about the generalization of the model.

Model 2 is a similar model as model 1, but model 2 is made with a rectified activation (ReLu) in the first hidden layer. The rectified activation will return 0 if it receives a negative input. When the rectified activation receives a positive value, then the function will return the same positive value (that was the input).[4]

The accuracy of the training set when the ReLU activation is implemented (model 2), is higher with a starting accuracy of 0.9070 compared to the score of model 1 with accuracy of 0.8838 (see epoch 0 in graphs). The training accuracy of model 2 is also increasing rapidly till the last epoch with a value of 0.9968. The training and validation accuracy cross each other at epoch 2, which is one epoch sooner than without the ReLU activation.

The accuracy (of the epochs) for the validation set for model 2 are also higher than that of the validation set for model 1. The accuracy of the validation set of model 2 starts at around 0.9463 and that of model 1 starts at around 0.9168. Another thing that stands out is that the validation accuracy does not fluctuate that much as without the ReLU activation (model 1). In model 1 (without ReLU activation), the accuracy of the validation set has more increases and decreases over the different epochs.

The accuracy of the training and validation sets of model 2 are further away from each other. This was not the case with model 1, where the training and validation accuracy were closer to each other. The accuracy of the training and validation sets are getting away from each other and are not converging towards each other. It can be said that the model is overfitting. There is overfitting, because the model fits on the training set but has a limited capability of generalization, so a limited fit on the validation set (accuracy of validation is lower than the training). This was not the case of model 1, in this model the training and validation accuracy were converging towards each other. The limited generalization of model 1 was mainly because of no stabilization and the fluctuations over the epochs. Model 2 also has a limited generalization, because of no convergence of the training and validation accuracy and the limited fit on the validation set. Overall, it can be said that model 2 has a limited generalization, just like model 1.

When looking at the loss (error) of the training and validation sets, it also can be said that the model is overfitting. The validation loss has started to increase from epoch 7 (with a loss of 0.0776), while the training loss is decreasing. This was not the case of the previous model (model 1). The training error for model 1 is decreasing, while the validation error is fluctuating and there is no stabilization.

[4] DeepAI. (2020, June 25). Rectified Linear Units. Retrieved 1 March 2022, from https://deepai.org/machine-learning-glossary-and-terms/rectified-linear-units

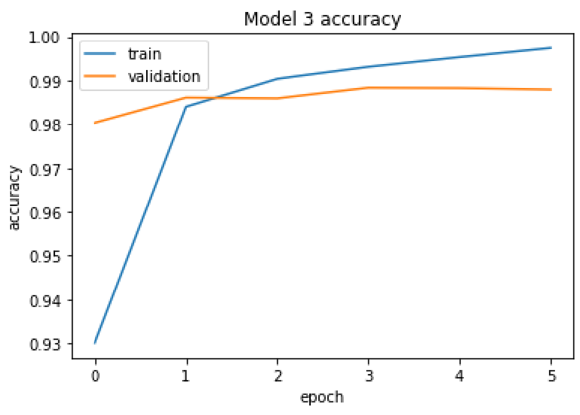
Deep convolutional networks

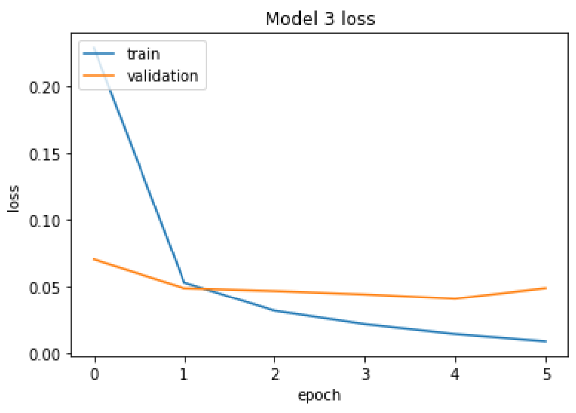
Question 10 (2 points)
Plot the training history and show this to your teacher.

The following text is the output of the model fit, for each epoch information about how long it took to run and the training performance is given:

```
Epoch 1/6
375/375 [==============================] - 116s 306ms/step - loss: 0.2285 - accuracy: 0.9301 - val_loss: 0.0703 - val_accuracy: 0.9803
Epoch 2/6
375/375 [==============================] - 112s 299ms/step - loss: 0.0530 - accuracy: 0.9840 - val_loss: 0.0487 - val_accuracy: 0.9861
Epoch 3/6
375/375 [==============================] - 111s 296ms/step - loss: 0.0320 - accuracy: 0.9904 - val_loss: 0.0467 - val_accuracy: 0.9859
Epoch 4/6
375/375 [==============================] - 112s 299ms/step - loss: 0.0220 - accuracy: 0.9931 - val_loss: 0.0442 - val_accuracy: 0.9883
Epoch 5/6
375/375 [==============================] - 117s 312ms/step - loss: 0.0146 - accuracy: 0.9953 - val_loss: 0.0409 - val_accuracy: 0.9883
Epoch 6/6
375/375 [==============================] - 122s 325ms/step - loss: 0.0092 - accuracy: 0.9974 - val_loss: 0.0489 - val_accuracy: 0.9879
```

The first plot shows how the accuracy progresses and the second plot shows how the loss (error) progresses differently across the epochs for the training and validation sets.





Question 11 (5 points)
Discuss with your group, then describe to your teacher, how the training history differs from the previous model, for the training and validation sets. What does this tell us about the generalization of the model?

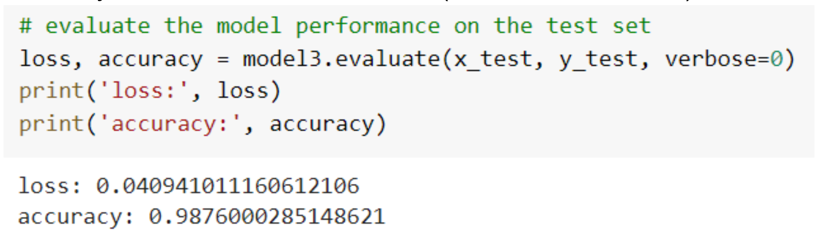
The accuracy scores of the convolutional network model (model 3), are overall higher than those of the previous model (model 2). The accuracy of the training set of model 3 starts from 0.9301, which is higher than that of the previous model where the accuracy score starts from 0.907. The accuracy of the training set is increasing significantly in the first epoch and then increases slightly every epoch after to the last epoch with an accuracy of 0.9974. The accuracy of the training set for model 3 follows the same structure as the accuracy of the training set for model 2. For the validation set, the accuracy of model 3 starts from 0.9803. This value is higher than the starting value of the accuracy of model 2, which starts from 0.9463. The accuracy of the validation set of model 3 is less fluctuating and seems more steady than that of the accuracy of the validation set of model 2.

We see that the training accuracy surpasses the validation accuracy, which is the same as with the previous model. The difference here is that these accuracies cross each other at a higher value. In this model, it seems that overfitting occurs between the first and the second epoch at an accuracy value of around 0.985, whereas with the previous model it was at around 0.965. So it seems overfitting occurs faster in this model than the previous model where overfitting seems to occur between epochs two and three.

For this model, it can also be said that the model is overfitting because of the limited generalization of the model on the validation set and the absence of convergence of the two sets. The loss (error) of the training and validation sets also indicate that overfitting is present in this model. The validation loss started to increase while the training loss is still decreasing. This was also the same for the previous model. The train and validation loss cross each other at the first epoch in model 3 whereas the loss of model 2 for the train and validation set cross each other at epoch 2. So overfitting occurs faster in model 3 than in model 2.

Question 12 (2 points)
Show your teacher what values you get for the model's accuracy and loss.

Accuracy and loss values of model 3 (convolutional model):



```
# evaluate the model performance on the test set
loss, accuracy = model3.evaluate(x_test, y_test, verbose=0)
print('loss:', loss)
print('accuracy:', accuracy)
```

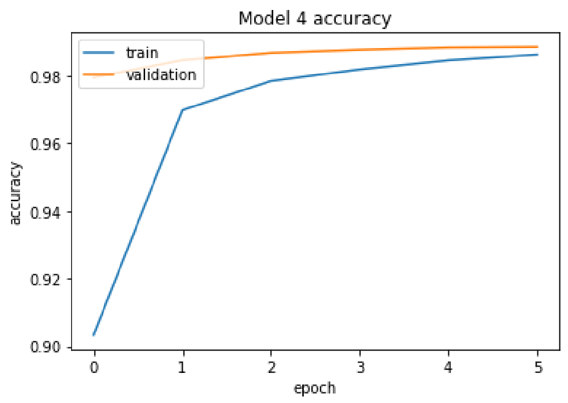
```
loss: 0.040941011160612106
accuracy: 0.9876000285148621
```

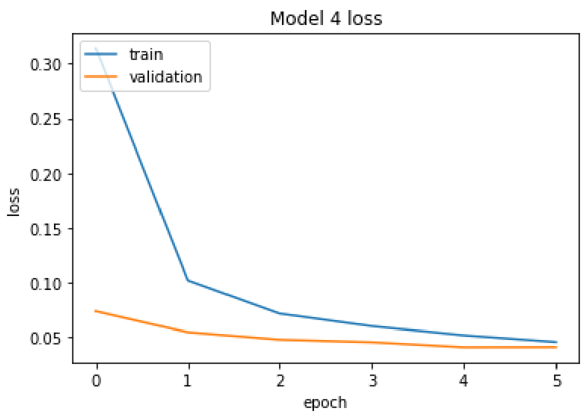
Question 13 (5 points)
Discuss with your group, then describe to your teacher, whether this accuracy is sufficient for some uses of automatic hand-written digit classification.

The accuracy of 0.9876 (model 3) is higher than that of the previous model (model 2), which has an accuracy of 0.9243. Even though this value of accuracy is quite high, the model still seems to make small errors on the data. This convolutional network model (model 3) performs better than the previous model with ReLu (model 2). As described above, both models seem to be overfitting the data. This is not ideal for practical use of automatic hand-written digit classification. An accuracy of 0.9876 is still not enough for fully automating certain processes since it would still make errors. However, it does come close to an accuracy which would be needed for automating number recognition processes. We have to keep in mind that overfitting is occuring in model 3. Whether the accuracy is sufficient or not largely depends on what it is used for. If this is used for important cases where small errors could lead to extreme effects, i.e. patient files or student numbers and grades, this accuracy is not sufficient at all. In general, this model also needs some human validation on the prediction that would be made with this model, to check the small errors that potentially can occur.

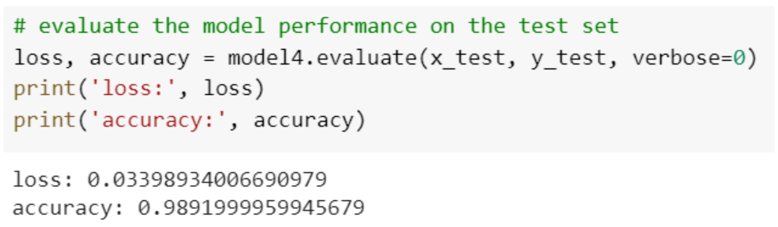
Question 14 (3 points)
Discuss with your group, then describe to your teacher, how the training history differs from the previous (convolutional) model, for both the training and validation sets, and for the time taken to run each model epoch.

The first plot shows how the accuracy progresses and the second plot shows how the loss (error) progresses differently across the epochs for the training and validation sets.





Accuracy and loss values of model 4 (convolutional model with dropout):



```
# evaluate the model performance on the test set
loss, accuracy = model4.evaluate(x_test, y_test, verbose=0)
print('loss:', loss)
print('accuracy:', accuracy)
```

```
loss: 0.03398934006690979
accuracy: 0.9891999959945679
```

\This model (model 4) has almost the same running time per epoch as the previous (convolutional) model (model 3). The dropout makes the model a bit more complex as the run time per epoch is a bit more/higher. With the addition of the dropout we see that no overfitting occurs, as the training accuracy never exceeds the validation accuracy. This also can be seen in the loss (error) of the model. The validation set is decreasing and the training set is also decreasing. Where the loss (error) of the validation is lower than the training set. The validation accuracy is almost the same for both models. For the training accuracy this differs, as the accuracy for the dropout model (model 4) is around 0.98 and the accuracy of the previous (convolutional) model (model 3) is 0.997 at the sixth epoch. In the training set of the dropout model, we started with a low accuracy (on first epoch) and later on followed the same patterns as before with high accuracy at ends, and in the validation set both had similar results. Even though the accuracy is very similar, model 3 jumps to a higher accuracy after the first epoch, whereas model 4 increases more slowly due to dropout in the model.

The accuracy of the validation set is more steady than the training set, and is increasing from 0.9794 (see epoch 0) to 0.9884 (see last epoch). The accuracy score of the validation set is overall higher than the training set. The loss of the model with dropout seems the same for the validation set as the loss of the previous model. However, the loss of the training set differs. The loss of the model with dropout of the training set seems much higher than those of the previous model (see graphs). Also here, the training loss never exceeds the validation loss, which is not the case with the previous model (see graph of model 3 in question 10). The loss at the sixth epoch is almost the same for both the training and the validation set with a value of around 0.04.

Question 15 (3 points)
Discuss with your group, then describe to your teacher, what this tells us about the generalization of the two models.

The generalization of the neural network, means how good our model is at learning from the given training set and applying the learned information on the validation set (on the data that is not used for training). If the neural network performs well on the validation set (that is not used for training), then we can say that the model is generalized well on the given data. The goal of a good model is to generalize well from the training data to any data from the problem domain (in our case the validation set). This allows the model to make predictions in the future on the data that the model has not seen yet. To describe how well the model learns and generalizes to new data, we can use the terms overfitting and underfitting. These are the biggest causes for poor performance of the model.[5]

For the two convolutional network models (model 3 and model 4) six epochs were used. We see that the training and validation accuracy and loss of model 4 converge towards each other, which is not the case of model 3. The training and validation accuracy and loss of model 3 are diverging after some point, specifically after epoch 1 (see graph in question 10). In model 4, the training accuracy and loss never exceed the values of the validation set (see graph in question 14), so there is no indication of overfitting. The validation accuracy and loss of model 3 decreases after epoch 1. After this epoch, these values are then lower than that of the training set. For model 4 it can be said that the model learns well and generalizes to new data (validation set). The converging values of the training and validation accuracy and loss of model 4 are an indication that there is good generalization of model 4. Which is not the case for model 3, which has a limited generalization to new data. It can be said that for model 3 overfitting may have occurred.

[5] Brownlee, J. (2019b, August 12). Overfitting and Underfitting With Machine Learning Algorithms. Machine Learning Mastery. Retrieved 1 March 2022, from https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/

Exercise two: Identifying objects from images

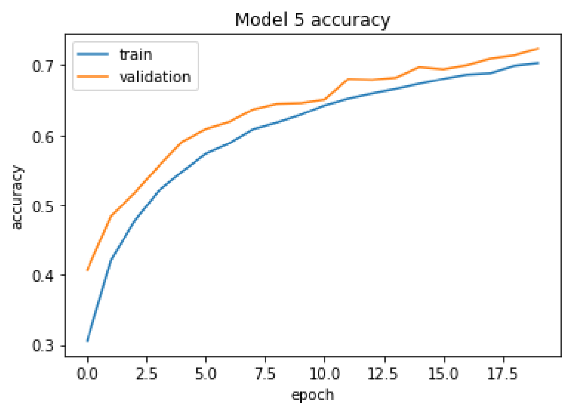
Question 16 (6 points)
Before fitting the model, show your teacher the code you used to define the model described here.

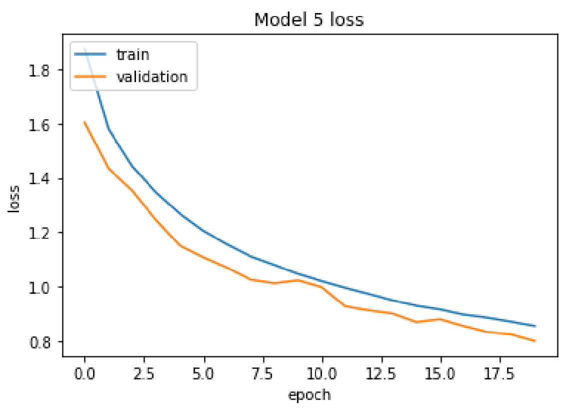
The code that is used to define the model using the convolutional network with dropout:

```
model5 = keras.Sequential()

model5.add(keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation="relu", input_shape = (32, 32, 3), padding = 'same'))
model5.add(keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation="relu"))
model5.add(keras.layers.MaxPool2D(pool_size=(2,2)))
model5.add(keras.layers.Dropout(rate=0.25))

# After max pooling and dropout layers, repeat these layers again
# Add conv, conv, pool, dropout, after the existing conv, conv, pool, dropout
model5.add(keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation="relu", padding = 'same'))
model5.add(keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation="relu"))
model5.add(keras.layers.MaxPool2D(pool_size=(2,2)))
model5.add(keras.layers.Dropout(rate=0.25))

# Flatten the result and link it to a larger fully-connected layer than before, using 512 units instead of 128, with dropout as before.
model5.add(keras.layers.Flatten())
model5.add(keras.layers.Dense(512, activation="relu"))
model5.add(keras.layers.Dropout(rate=0.5))
model5.add(keras.layers.Dense(10,activation="softmax"))

# In compiling the model, we will use a specialised optimizer module
model5.compile(loss='categorical_crossentropy', optimizer=keras.optimizers.RMSprop(learning_rate=0.0001, decay=1e-6), metrics='accuracy')

# In the model fit command, set the batch size to 32 and the number of epochs to 20
history5 = model5.fit(x_train, y_train, batch_size=32, epochs=20, verbose=1, validation_data=(x_test, y_test), shuffle = True)
```

Question 17 (2 points)
Plot the training history and show it to your teacher.

The first plot shows how the accuracy progresses and the second plot shows how the loss (error) progresses differently across the epochs for the training and validation sets.

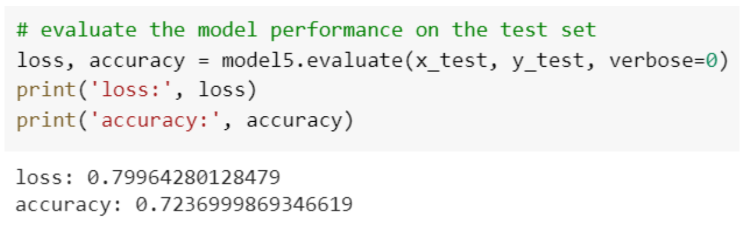




Question 18 (5 points)
Discuss with your group, then describe to your teacher, how the training history differs from the convolutional model for digit recognition and why.

The model with 2 layers of convolution (4 if we consider 2 conv x 2 sub conv) is more complex, and turns the model slower than before. The reason for this is that we now have a 3D array of an image, so basically we come from 784 to 3072 (32x32x3) input size, it turns the model slower. The pictures used in the model are natural images which makes them a lot more complex with more dimensions and patterns within the picture. Classifying natural images makes the model more complex whereas in the convolution model we only used pictures of numbers that were less difficult to classify. The accuracy is a lot lower compared to the digit recognition accuracy (~70% from 98%), the complexity of images can explain this behaviour.

The accuracy of the model (model 5) doesn't jump as high after the first epoch but continues to improve due to the smaller learning rate. The validation accuracy is higher than the training accuracy but both are still rather low, compared to the previous models (model 4). Both the accuracy of the training and validation sets have the form of a curve, and this was not the case with the previous convolutional model (see graph in question 14). Besides that, the accuracy of the validation set is higher than the training set, so we can assume that the model can generalize on the validation set. Different from the previous model, the training and the validation loss (error) are decreasing in a curve, and the loss (error) of the validation is lower than the training set. Due to the shape of the curve, it is expected that with more epochs, it will achieve a higher accuracy (after more epochs). Training the model with 20 epochs give the following values for the accuracy and the loss (error):



```
# evaluate the model performance on the test set
loss, accuracy = model5.evaluate(x_test, y_test, verbose=0)
print('loss:', loss)
print('accuracy:', accuracy)
```

```
loss: 0.79964280128479
accuracy: 0.7236999869346619
```

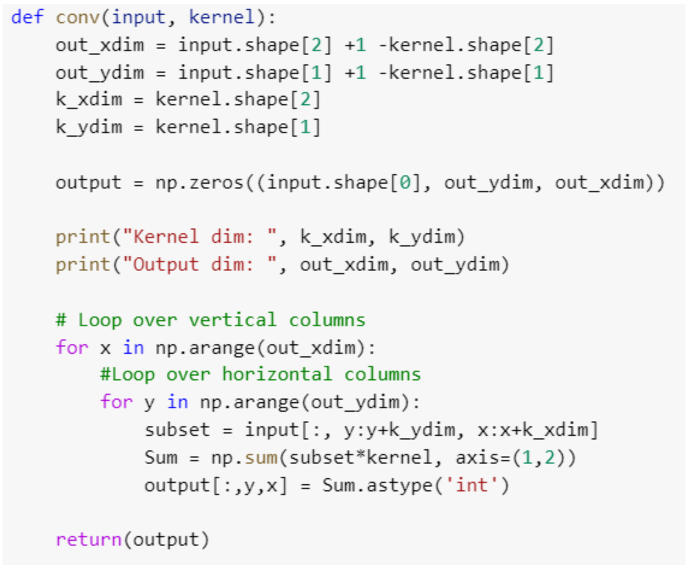
Question 19 (4 points)
Discuss with your group, then describe to your teacher, how the time taken for each training epoch differs from the convolutional model for digit recognition. Give several factors that may contribute to this difference.

Factors that may contribute to the difference in the time that is taken by each training epoch:
- Number of layers in the model (from 1 to 2): with 2 layers the model has more weights and neurons so more process timing. The more layers the model has, the more time it takes to train the neural network model.
- Input size (from 784,1) to (32,32,3): images in 3D have more dimensions, so the size of filters for definition makes the model more complex and requires more computations.
- The model that took the most time to run was connected to a larger fully-connected layer, where 512 units were used instead of 128 (this was used for the other models).
- The learning rate for the model (model 5) that takes the most time is set to 0.0001. This indicates the amount that the weights in the model are updated during the training process. A small learning rate requires more training epochs, because of the smaller changes that the model will make to update the weights. The previous models had a learning rate of 1, where the last model (that took the most of the time) had a learning rate of 0.0001.
- The last model has more parameters, this leads to better estimation of the function. However a model with more parameters becomes larger and such large models are slow.

Exercise three: Low-level functions

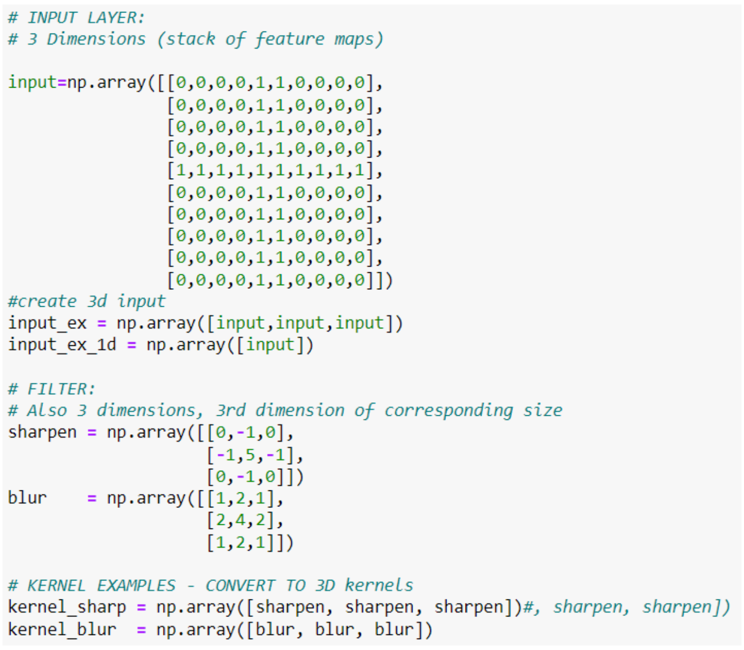
Question 20 (5 points)
Write a simple convolution operation that achieves the convolution operation efficiently for two dimensional and three-dimensional inputs. This should allow you to input a set of convolutional filters ('kernels' in Keras's terminology) and an input layer (or image) as inputs. The input layer should have a third dimension, representing a stack of feature maps, and each filter should have a third dimension of corresponding size. The function should output a number of two-dimensional feature maps corresponding to the number of input filters, though these can be stacked into a third dimension like the input layer. After agreeing on a common function with your group members, show this to your teacher.

The code for the convolution operation function that achieves the convolution operation efficiently for two dimensional and three-dimensional inputs:



```
def conv(input, kernel):
    out_xdim = input.shape[2] +1 -kernel.shape[2]
    out_ydim = input.shape[1] +1 -kernel.shape[1]
    k_xdim = kernel.shape[2]
    k_ydim = kernel.shape[1]

    output = np.zeros((input.shape[0], out_ydim, out_xdim))

    print("Kernel dim: ", k_xdim, k_ydim)
    print("Output dim: ", out_xdim, out_ydim)

    # Loop over vertical columns
    for x in np.arange(out_xdim):
        #Loop over horizontal columns
        for y in np.arange(out_ydim):
            subset = input[:, y:y+k_ydim, x:x+k_xdim]
            Sum = np.sum(subset*kernel, axis=(1,2))
            output[:,y,x] = Sum.astype('int')

    return(output)
```

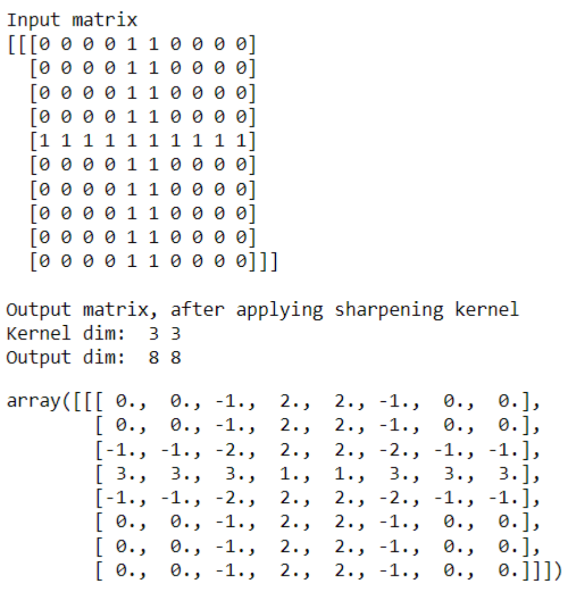
Set of convolutional filters and an input layer (or image) as inputs. The input structure is as follows: the featuremap is represented by the 2nd and 3rd dimension of the matrix, and the stack of feature maps by the 1st dimension. e.g. input[z,y,x], where z is the color channel, y the y-dimension of the image and x the x-dimension of the image.



```
# INPUT LAYER:
# 3 Dimensions (stack of feature maps)

input=np.array([[0,0,0,0,1,1,0,0,0,0],
                [0,0,0,0,1,1,0,0,0,0],
                [0,0,0,0,1,1,0,0,0,0],
                [0,0,0,0,1,1,0,0,0,0],
                [1,1,1,1,1,1,1,1,1,1],
                [0,0,0,0,1,1,0,0,0,0],
                [0,0,0,0,1,1,0,0,0,0],
                [0,0,0,0,1,1,0,0,0,0],
                [0,0,0,0,1,1,0,0,0,0],
                [0,0,0,0,1,1,0,0,0,0]])
#create 3d input
input_ex = np.array([input,input,input])
input_ex_1d = np.array([input])

# FILTER:
# Also 3 dimensions, 3rd dimension of corresponding size
sharpen = np.array([[0,-1,0],
                    [-1,5,-1],
                    [0,-1,0]])
blur    = np.array([[1,2,1],
                    [2,4,2],
                    [1,2,1]])

# KERNEL EXAMPLES - CONVERT TO 3D kernels
kernel_sharp = np.array([sharpen, sharpen, sharpen])#, sharpen, sharpen])
kernel_blur  = np.array([blur, blur, blur])
```

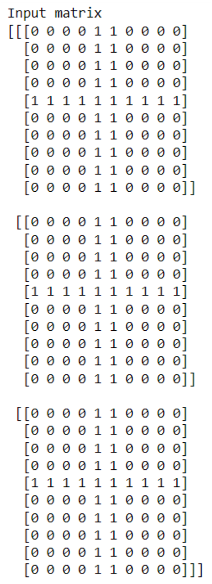
Two examples of what the function *(conv(input, kernel))* gives as output, on of two dimensional and third dimensional:

Two dimensional

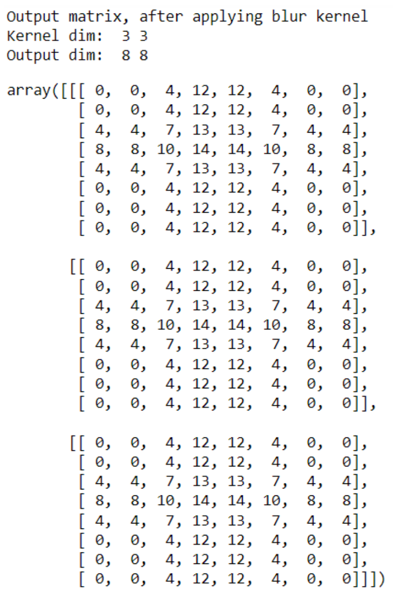


```
Input matrix
[[0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]]]

Output matrix, after applying sharpening kernel
Kernel dim:  3 3
Output dim:  8 8

array([[[ 0.,  0., -1.,  2.,  2., -1.,  0.,  0.],
        [ 0.,  0., -1.,  2.,  2., -1.,  0.,  0.],
        [-1., -1., -2.,  2.,  2., -2., -1., -1.],
        [ 3.,  3.,  3.,  1.,  1.,  3.,  3.,  3.],
        [-1., -1., -2.,  2.,  2., -2., -1., -1.],
        [ 0.,  0., -1.,  2.,  2., -1.,  0.,  0.],
        [ 0.,  0., -1.,  2.,  2., -1.,  0.,  0.],
        [ 0.,  0., -1.,  2.,  2., -1.,  0.,  0.]]])
```

Third dimensional:



```
Input matrix
[[[0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]]

 [[0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]]

 [[0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]]]
```



```
Output matrix, after applying blur kernel
Kernel dim:  3 3
Output dim:  8 8

array([[[ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 4,  4,  7, 13, 13,  7,  4,  4],
        [ 8,  8, 10, 14, 14, 10,  8,  8],
        [ 4,  4,  7, 13, 13,  7,  4,  4],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0]],

       [[ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 4,  4,  7, 13, 13,  7,  4,  4],
        [ 8,  8, 10, 14, 14, 10,  8,  8],
        [ 4,  4,  7, 13, 13,  7,  4,  4],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0]],

       [[ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 4,  4,  7, 13, 13,  7,  4,  4],
        [ 8,  8, 10, 14, 14, 10,  8,  8],
        [ 4,  4,  7, 13, 13,  7,  4,  4],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0],
        [ 0,  0,  4, 12, 12,  4,  0,  0]]])
```

Question 21 (2 points)
Write a simple function that achieves rectified linear (relu) activation over a whole feature map, with a threshold at zero. After agreeing on a common function with your group members, show this to your teacher.

Some code examples of the rectified linear (relu) activation functions:

```python
# rectified linear function
def relu(feature_map):
  return max(0.0, feature_map)
```

```python
def ReLu(x):
    return max(0, x)

def ReLu_feat(featuremap):
    return np.vectorize(ReLu)(featuremap)
```

```python
def relu(feature_map):
    #Preparing the output of the ReLU activation function.
    relu_out = np.zeros(feature_map.shape)
    for map_num in range(feature_map.shape[-1]):
        for r in np.arange(0,feature_map.shape[0]):
            for c in np.arange(0, feature_map.shape[1]):
                relu_out[r, c, map_num] = np.max([feature_map[r, c, map_num], 0])
    return relu_out
```

Question 22 (3 points)
Write a simple function that achieves max pooling. This should allow you to specify the spatial extent of the pooling, with the size of the output feature map changing accordingly. After agreeing on a common function with your group members, show this to your teacher.

The code for the max pooling function:
```python
#input is a stack of feature maps

def maxpool(input, xdim, ydim):
    out_xdim = int(input.shape[2] / xdim)
    out_ydim = int(input.shape[1] / ydim)
    out_zdim = input.shape[0]
    print("Dimensions of Max Pool result:", out_ydim, "x", out_xdim)
    output = np.zeros((out_zdim, out_ydim, out_xdim))

    for z in range(0,input.shape[0]):
        fmap = input[z]
        for x in range(0, input.shape[2], xdim):
            for y in range(0, input.shape[1], ydim):
                subset = fmap[y:y+ydim, x:x+xdim]
                output[z, int(y/ydim),int(x/xdim)] = np.max(subset)
    return output.astype('int')

print(input_ex)
maxpool(input_ex, 2,2)
```

The kernel size of the maxpool matrix is defined by xdim and ydim and the stride is equal to the dimensions of the kernel. The first loop goes over the stack of feature maps and the following two loops loop over the x and y dimensions of the input matrix. For a 2x2 kernel, the for loop range is like 0,2,4,6,8 and for a 3x3 kernel 0,3,6,9,12 etc. These iterators are then divided by the kernel dimensions to get the location integers for the output matrix (0,1,2,3,4). This works since the output dimension is always input_dim/kernel_dim, e.g. a 6x6 input gets a 3x3 output matrix.
Note: the size of the input matrix must be divisible by the size of the maxpool submatrix

Example of output using the function *(maxpool(input, xdim, ydim))*:
```
Input matrix
[[[0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]]

 [[0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]]

 [[0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]
  [0 0 0 0 1 1 0 0 0 0]]]
```

```
array([[[0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0],
        [1, 1, 1, 1, 1],
        [0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0]],

       [[0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0],
        [1, 1, 1, 1, 1],
        [0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0]],

       [[0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0],
        [1, 1, 1, 1, 1],
        [0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0]]])
```

Question 23 (2 points)
Write a simple function that achieves normalization within each feature map, modifying the feature map so that its mean value is zero and its standard deviation is one.

```
def normalize(input):
    mean = np.mean(input)
    std  = np.std(input)
    return ((input - mean) / std)
```

Question 24 (5 points)
Write a function that produces a fully-connected layer. This should allow you to specify the number of output nodes, and link each of these to every node in an input stack of feature maps. You should probably begin by flattening this stack of feature maps into a 1-dimensional matrix. After agreeing on a common function with your group members, show this to your teacher.

```
# INPUT LAYER (random created):
# 3 Dimensions (stack of feature maps)
import numpy as np
input = np.array([[1,0,0,0,1,1,0,0,0,1],
                 [0,0,0,0,1,1,0,0,0,0],
                 [0,0,0,0,1,1,0,0,1,0],
                 [0,0,0,0,1,1,0,0,0,0],
                 [1,1,0,1,1,0,1,0,1,1],
                 [0,1,0,0,1,1,0,0,0,0],
                 [0,1,0,0,1,1,0,0,1,0],
                 [0,0,0,0,1,1,0,0,0,0],
                 [0,1,0,0,1,1,0,0,0,0],
                 [0,0,1,0,1,1,0,0,0,1]])

#create 3d input
input = np.array([input,input,input])
input = np.array([input])
input
```

The code for the fully-connected layer function. In the function the number of output nodes can be specified in the output_size. First the input (stack of feature maps) will be flattened into a 1-dimensional matrix using .flatten(). The weights are initialized with a random value. Then we took the dot product of the flatted input values and the weights.

```
import numpy as np

# fully-connected layer
# output_size = specify the number of output nodes
def layer(input, output_size):

  # flatten the features
  flat_input = input.flatten()
  # specify the weights
  weights = np.random.rand(len(flat_input), output_size) - 0.5

  # returns output for the given input
  output = np.dot(flat_input, weights)
  return output
```

Now some examples will be given, these are the output of the function *layer(input, output_size)*:

```
layer(input, 5)
```

```
array([-2.36433641,  3.70484699, -1.25250036, -4.01988769, -2.24049114])
```

```
layer(input, 10)
```

```
array([[-2.0571493 ,  0.25742646, -0.70730954,  0.92770046,  0.34914269,
        -1.70308802, -0.9604507 , -0.09708905,  2.29351988,  3.24940173]])
```

```
layer(input, 20)
```

```
array([ 1.61040334, -0.31272282,  2.69985953,  0.72132493, -2.45392823,
       -6.837265  , -0.08578421,  3.28821617, -0.01530242, -3.32442827,
        3.83577317,  3.38335052,  0.19412391,  4.29674004, -1.01967296,
       -2.21255447,  1.15002848,  1.40832141,  2.35887554, -3.46440026])
```

Question 25 (2 points)
Write a function that converts the activation of a 1-dimensional matrix (such as the output of a fully-connected layer) into a set of probabilities that each matrix element is the most likely classification. This should include the algorithmic expression of a softmax (normalized exponential) function. After agreeing on a common function with your group members, show this to your teacher.

The code for the function that converts the activation of a 1-dimensional matrix into a set of probabilities:

```
from numpy import exp

# calculate the softmax of 1-dimensional matrix
# algorithmic expression of a softmax (normalised exponential) function
def softmax(matrix_1d):
  e = exp(matrix_1d)
  return e / e.sum()
```

We applied this function on the output of the fully-connected layer. Examples of the output of the function with different output_size for the function *layer(input, output_size)*:

```
# convert output of fully-connected layer into a set of probabilities
data = layer(input, 10)
result = softmax(data)

# print the probabilities
print(result)

# print the sum of the probabilities
print(sum(result))
```

```
[7.32041242e-04 3.14137075e-01 4.95747301e-04 9.89842024e-04
 2.95952235e-02 7.51966570e-03 3.23434462e-04 5.19180805e-01
 2.51093359e-02 1.01916830e-01]
0.9999999999999999
```

```python
# convert output of fully-connected layer into a set of probabilities
data = layer(input, 20)
result = softmax(data)

# print the probabilities
print(result)

# print the sum of the probabilities
print(sum(result))
```

```
[5.72408549e-02 1.17625166e-01 6.10575220e-04 7.74985651e-05
 3.23282739e-04 4.77891545e-03 2.84044487e-04 7.46889906e-01
 1.97956383e-02 3.88314379e-03 9.64852789e-06 1.65664722e-02
 3.64137292e-05 1.62476210e-03 2.36096919e-02 1.34326757e-05
 5.93479955e-03 5.65828686e-04 1.12605571e-04 1.73196240e-05]
1.0
```

```python
# convert output of fully-connected layer into a set of probabilities
data = layer(input, 50)
result = softmax(data)

# print the probabilities
print(result)

# print the sum of the probabilities
print(sum(result))
```

```
[5.38985184e-03 7.00086668e-03 2.88918783e-04 6.21020314e-05
 3.87712049e-03 9.67024717e-06 1.13732192e-02 5.25222584e-04
 1.01022126e-03 8.81171597e-05 4.18345667e-05 4.10418703e-04
 1.24646436e-02 1.08711273e-03 3.12266899e-04 6.61581950e-05
 7.55715333e-04 7.48359231e-05 6.36312523e-05 4.12515869e-04
 3.68591946e-03 1.03053047e-04 1.38627435e-01 7.72496376e-03
 8.89196552e-02 1.79665315e-04 1.36597458e-04 6.08291592e-05
 9.36304011e-04 1.17753604e-03 1.66526834e-03 7.98219739e-02
 1.01316246e-03 1.68282287e-03 2.73608102e-01 7.76202191e-03
 9.62514037e-04 1.95104847e-01 8.45128218e-05 1.45372317e-04
 1.57962125e-02 9.41639663e-03 8.46707605e-03 6.33850196e-03
 3.11332040e-03 6.80676780e-02 6.65808182e-03 8.67542297e-04
 7.80336708e-05 3.24801574e-02]
0.9999999999999999
```