Machine Learning for Human Vision and Language

Lab Assignment 1

Group members:

Riccardo Bassani 6866840
David-Paul Niland 6688721

Exercise one: Identifying Handwritten Numbers

Can you think of another application where automatic recognition of hand-written numbers would be useful? (Question 1, 3 points)

Recognition of hand-written numbers would be also useful in reading hand-filled forms containing digits data as IBAN codes, phone numbers, birth dates…
Another interesting application could be teaching children how to write numbers: a program able to recognise digits could substitute the teacher in the process of checking whether the number has been correctly written.

Multilayer perceptron: Training and evaluation.

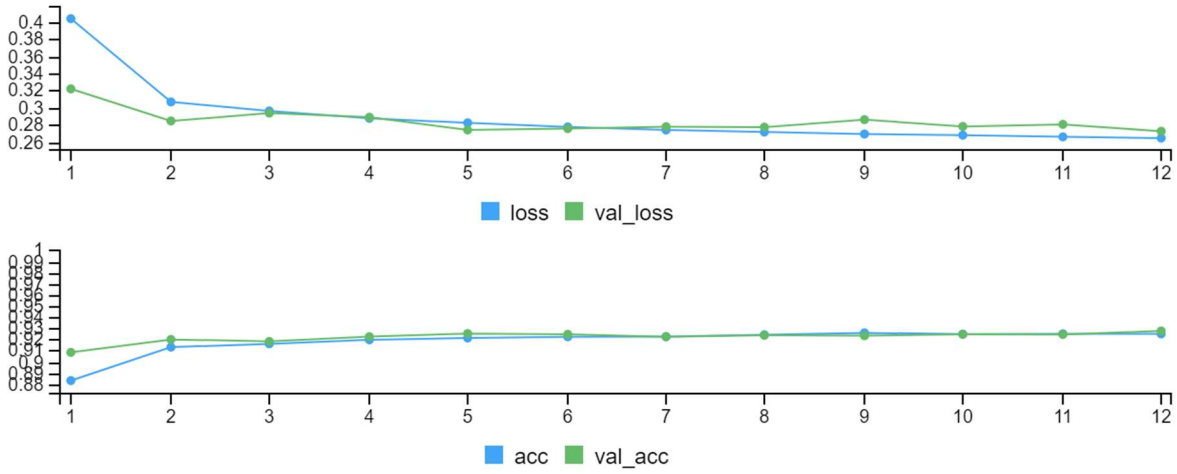
In the output text in your console, how long did each epoch take to run? (Question 2, 2 points)

Each epoch took approximately 3 seconds to run.
On average each sample took 60 us, with a standard deviation of 4 us.
The times taken by the different epochs appear then homogeneous.

Further Details

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/12
48000/48000 [==============================] - 3s 62us/sample - loss: 0.4036 - acc: 0.8832 - val_loss: 0.3220 - val_acc: 0.9085
Epoch 2/12
48000/48000 [==============================] - 3s 64us/sample - loss: 0.3068 - acc: 0.9132 - val_loss: 0.2847 - val_acc: 0.9201
Epoch 3/12
48000/48000 [==============================] - 3s 61us/sample - loss: 0.2963 - acc: 0.9162 - val_loss: 0.2938 - val_acc: 0.9183
Epoch 4/12
48000/48000 [==============================] - 3s 58us/sample - loss: 0.2877 - acc: 0.9198 - val_loss: 0.2890 - val_acc: 0.9226
Epoch 5/12
48000/48000 [==============================] - 3s 60us/sample - loss: 0.2825 - acc: 0.9214 - val_loss: 0.2742 - val_acc: 0.9254
Epoch 6/12
48000/48000 [==============================] - 3s 66us/sample - loss: 0.2777 - acc: 0.9225 - val_loss: 0.2758 - val_acc: 0.9247
Epoch 7/12
48000/48000 [==============================] - 3s 60us/sample - loss: 0.2742 - acc: 0.9227 - val_loss: 0.2780 - val_acc: 0.9225
Epoch 8/12
48000/48000 [==============================] - 3s 56us/sample - loss: 0.2718 - acc: 0.9243 - val_loss: 0.2775 - val_acc: 0.9238
Epoch 9/12
48000/48000 [==============================] - 3s 62us/sample - loss: 0.2695 - acc: 0.9260 - val_loss: 0.2862 - val_acc: 0.9234
Epoch 10/12
48000/48000 [==============================] - 3s 54us/sample - loss: 0.2681 - acc: 0.9249 - val_loss: 0.2783 - val_acc: 0.9247
Epoch 11/12
48000/48000 [==============================] - 3s 54us/sample - loss: 0.2662 - acc: 0.9252 - val_loss: 0.2807 - val_acc: 0.9246
Epoch 12/12
48000/48000 [==============================] - 3s 58us/sample - loss: 0.2648 - acc: 0.9251 - val_loss: 0.2727 - val_acc: 0.9277
```

Plot the training history and add it to your answers. (Question 3, 3 points)



Describe how the accuracy on the training and validation sets progress differently across epochs, and what this tells us about the generalisation of the model. (Question 4, 5 points).

Both the train accuracy and the validation accuracy reach a good value already after the first epoch (0.8832 and 0.9085 respectively).
No signs of overfitting are shown. Indeed, the beginning validation accuracy is even higher than the train accuracy (this does not affect the quality of the model since both the accuracies increase during the training).
This means that the generalization of the model is very good, since it can make predictions on previously unseen data without losing accuracy.
The train accuracy increases meaningfully in the second epoch (reaching 0.9132) and slightly in the following epochs, stabilizing at around 0.9250 in the last epochs. This happens without compromising the validation accuracy, which also increases slightly and stabilizes at 0.9250.
The generalization of the model remains therefore excellent through the training.

What values do you get for the model's accuracy and loss? (Question 5, 2 points)

The model performance on the test set:

- loss 0.2788603
- acc 0.9255

Discuss whether this accuracy is sufficient for some uses of automatic hand-written digit classification. (Question 6, 5 points)

An accuracy of 0.9255 is encouraging, but still it is not acceptable for the majority of the tasks.
In tasks such as reading IBAN codes, house numbers or postal codes, every single digit is relevant and a 7.5% chance of error in the classification of a digit would lead to an error almost always (the probability of classifying correctly a n-digit sequence would be 0.9255^n. e.g. for 5 digits 0.68, for 15 digits 0.31).

There are indeed particular cases in which even a model with a 0.9255 accuracy could be acceptable and useful.
An example is the task consisting in calculating the average of a large set of numbers (for example when a large number of students are asked to vote in a satisfaction questionnaire).
An error in the identification of a single number would not affect drastically the final result and since these errors do not occur too often the task can be completed with good results.

Therefore, generally speaking, such an accuracy would be useful only for tasks in which an error in the recognition of a single digit is tolerable, as long as it does not occur too often.

How does linear activation of units limit the possible computations this model can perform? (Question 7, 5 points)

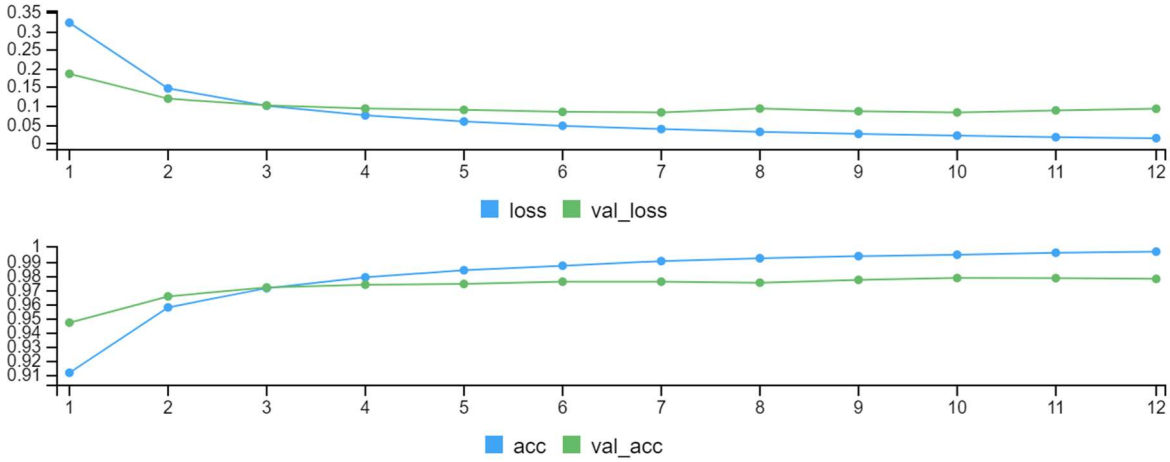
Linear activation functions present two major drawbacks.

Firstly, several hidden layers with a linear activation function can be reduced to a single layer, since a linear combination of linear functions is still a linear function. Therefore, this type of activation function does not allow to take advantage of multiple layers.
Nevertheless, this particular model has only one layer, hence this problem is not relevant.

What matters the most is the impossibility of using backpropagation having a linear activation function. The derivative of a linear function, indeed, is not related to the input (it is constant), then the gradient descent technique cannot be used.
This means the error cannot be minimised with low computational resources, which is possible through backpropagation.

Multilayer perceptron: changing model parameters.

Now make a similar model with a rectified activation in the first hidden layer. Plot the training history and add it to your answers. (Question 8, 2 points)



How does the training history differ from the previous model, for the training and validation sets? What does this tell us about the generalisation of the model? (Question 9, 5 points)

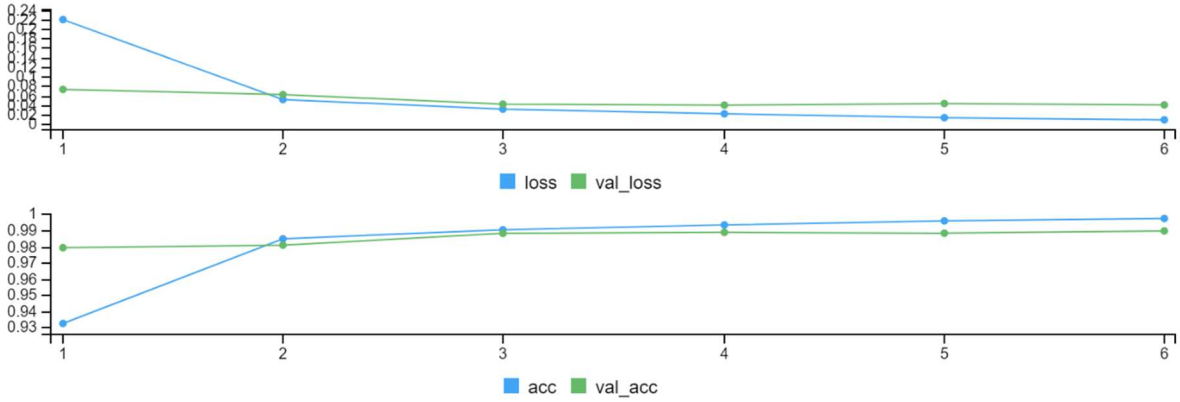
The accuracies increase in a similar way as before, but the model is now performing better, reaching an accuracy of 0.9966 for the training set.
Nevertheless, it is now slightly overfitted on the training set, which can be said noticing that the final accuracy value for the validation set is lower (0.9773).
Therefore, despite of the better performance in general, the generalisation of the model is now worse.

How does the new model's accuracy on test set classification differ from the previous model? Why do you think this is? (Question 10, 5 points)

The model's accuracy on test set classification is now significantly better than the old model's one.
This happens because a nonlinear activation function can better train the network, by allowing complex relationships in the data to be learned and by using the backpropagation.
ReLU has other advantages over a simple linear function, for example it allows for a true zero value to be output.

Deep convolutional networks

Define a convolutional learning model following the instructions. Plot the training history and add it to your answers. (Question 11, 2 points)



How does the training history differ from the previous model, for the training and validation sets? What does this tell us about the generalisation of the model? (Question 12, 5 points)

The accuracies increase in a similar way to the previous models and the final training set accuracy of the model is almost identical to the one of the single layer model (0.9973 vs 0.9966).
The main progress is that the difference between the validation set and the training set final accuracy is now lower (0.0077 vs 0.0193); this means that this model is better at generalising to unseen data.

What values do you get for the model's accuracy and loss? (Question 13, 2 points)

The model performance on the test set:

- loss: 0.0299
- acc: 0.9906

Discuss whether this accuracy is sufficient for some uses of automatic hand-written digit classification. (Question 14, 5 points)

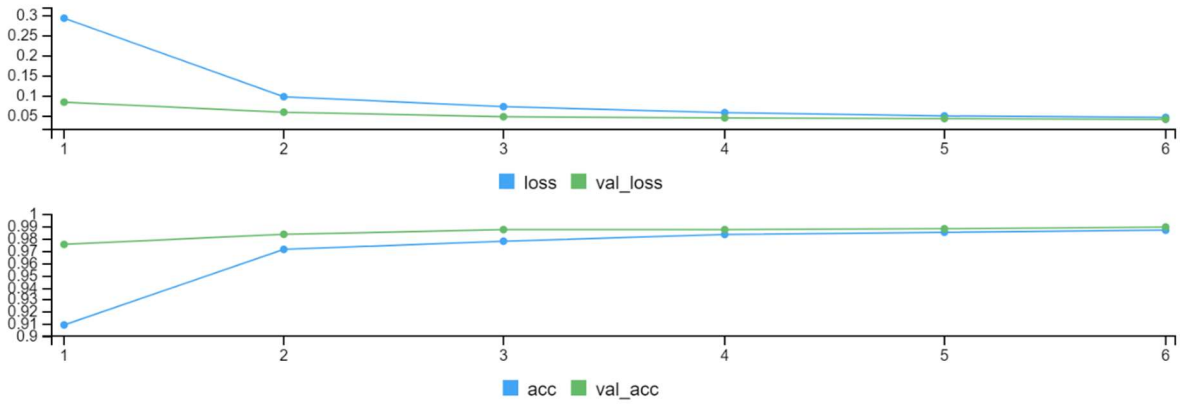
An accuracy of 0.9906 is still not sufficient for tasks which require that every single digit is correctly classified, but this model can perform very well in tasks where the single number is not important. An example of this could be a large satisfaction questionnaire where few errors won't drastically change the overall result.

Describe the principles of overfitting and how dropout can reduce this. (Question 15, 5 points)

Training a neural network in an improper way (small dataset, too many features, too many epochs) can lead to overfitting, i.e. the network sets its weights in order to maximise its accuracy when working on the training set. When it occurs, the excellent performance achieved on the training set will not be reached when the model is used to work on new, unseen data, since the model is built to work on a specific dataset.

An efficient method to prevent overfitting is called dropout and it consists in randomly ignoring some neurons in order to simulate training many neural networks with different architectures in parallel. Dropping out neurons prevents units from co-adapting too much, which means it avoids that units may change in a way that they fix up the mistakes of the other units.
If some neurons are randomly ignored, a hidden unit cannot rely on other specific units to correct its mistakes and it must perform well in a wide variety of different contexts provided by the other hidden units, therefore the network generalises better.

Add dropout layers after the max pooling stage and after the fully-connected layer. How does the training history differ from the previous (convolutional) model, for both the training and validation sets, and for the time taken to run each model epoch? (Question 16, 5 points)



In this model the overall performance is marginally worse than in the previous convolutional model, but the validation accuracy is never lower than the training accuracy, thus no signs of overfitting are present.
Due to the dropout, each epoch takes less time than in the previous model, but more epochs would be necessary for the model to converge.

What does this tell us about the generalisation of the two models? (Question 17, 5 points)

The results show how the model in which the dropout is adopted generalises better to unseen data. Indeed, there is not a significant difference between the training set accuracy and the validation set accuracy after the training.

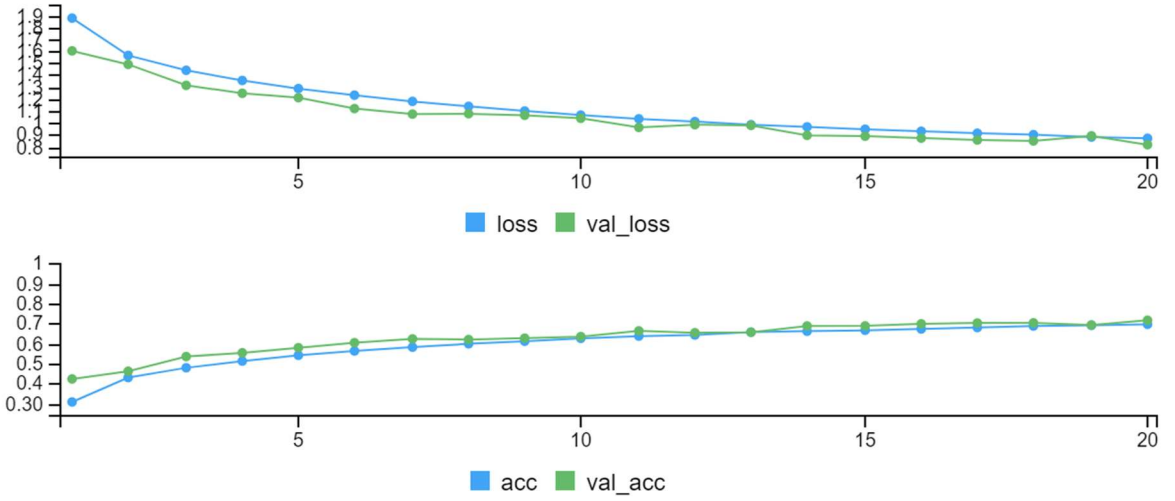
Exercise 2: Identifying objects from images

Define the model using the convolutional network with dropout as a template and following the instructions. What code did you use? (Question 18, 6 points)

```
model <- keras_model_sequential() %>%

        layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu', input_shape = c(32, 32, 3), padding = 'same') %>%
        layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu') %>%
        layer_max_pooling_2d(pool_size = c(2,2)) %>%
        layer_dropout(rate = 0.25) %>%
        layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu', padding = 'same') %>%
        layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu') %>%
        layer_max_pooling_2d(pool_size = c(2,2)) %>%
        layer_dropout(rate = 0.25) %>%
        layer_flatten() %>%
        layer_dense(units = 512, activation = 'relu') %>%#
        layer_dropout(rate = 0.5) %>%
        layer_dense(units = 10, activation = 'softmax')
```

Execute this model fit command. After your fitting is finished, plot the training history and put it in your answers. (Question 19, 2 points)



How does the training history differ from the convolutional model for digit recognition? Why do you think this is? (Question 20, 5 points)

The main difference is that in this case, the accuracy of the model is very low (0.31) after the first epoch and it rises significantly (+ 0.39) throughout the epochs, while in the convolutional model for digit recognition the accuracy was already quite high after the second epoch and it increased less (+ 0.07) through the training.
The number of items in the training set is approximately the same in the two cases, therefore this difference is probably due to the major complexity of the image identification task.
The network needs multiple epochs in order to adjust its weights and reach an acceptable value (0.70), and it is anyway significantly lower than the value obtained in the digit recognition task (0.98-0.99).

How does the time taken for each training epoch differ from the convolutional model for digit recognition? Give several factors that may contribute to this difference. (Question 21, 5 points)

The time taken for each training epoch in this model is 2-3 times greater than in the convolutional model for digit recognition.
This is due to the higher number of hidden layers in the diagram (5 vs 3 without considering dropout, pooling and flatten layers, 11 vs 7 including them), but also to the highest number of units in the dense layer (512 vs 128). The latter, indeed, increases the number of parameters of the model, then the computational load of the training.
Another factor may be the larger input, that could make heavier the computational load in the early layers.

Read the research paper "Performance-optimised hierarchical models predict neural responses in higher visual cortex". Write a short (~500 word) summary of the experimental approach and results. (Question 22, 10 points)

The experiment aims to show how performance optimisation can lead to build a hierarchical model that not only can reach human ability in objects recognition, but also is highly predictive of neural response in the IT cortex, a higher ventral cortical area of the human and monkey brain responsible for object recognition and capable of variance tolerance.

In order to do so, several models were initially drawn from a large parameter space of convolutional neural networks; to measure categorization performance, support vector machine linear classifiers were trained on model output layer units and their accuracy was computed.

To assess the models' neural predictivity for each target IT neural site, a synthetic neuron composed of a linear weighting of model outputs that would best match that site on fixed sample images was identified. After this was done, response predictions against actual neural site's output was tested on novel images.

Performance was significantly correlated with neural predictions and none of the 57 individual model parameters correlated nearly as strongly with IT predictions as performance, indicating that the

performance/IT prediction correlation cannot be explained by simpler mechanistic considerations such as receptive field size of the top layer.
Several published models where then tested at three levels of object view variation, showing that although some of them can match IT and human performance at low levels of variation, performance drops quickly at higher variation levels.

In order to overcome this limit, simple three layers hierarchical CNNs were substituted by combinations of deeper CNN networks and hierarchical modular optimization (HMO) was used in order to find especially high-performing architectures.

As a pre-training step, the HMO selection procedure was applied on a screening task, leading to a four-layer CNN with 1,250 top-level outputs, which resulted to be robust to large amounts of variation. Moreover, an object-level and a category level generalization test were performed, showing that the model's prediction generalizes robustly both with unseen objects and categories.

Not only did the last HMO's layer result in being highly predictive of IT neural response, but also its penultimate layer was found to be highly predictive of V4 neural responses. This supports the hypothesis that V4 corresponds to an intermediate layer in a hierarchical model with an effective model of IT as last layer.

After conducting the experiment, a principled method for achieving greatly improved predictive models of neural responses in higher ventral cortex was proposed, considering two biological constrains that shaped visual cortex. Since some tests revealed that ideal observers with high categorization performance appear to be significantly less predictive than the HMO model, indeed, not only high performance, but also the hierarchical model architecture was considered necessary in order to achieve IT neural predictivity.

Respecting these constrains and not tuning parameters using neural data, a model was obtained with high predictivity abilities in IT and V4. The results also support the hypothesis that considers IT as the product of an evolutionary/developmental process that selected for high performance on recognition on tasks like those used in the experiment.

Exercise three: Play time

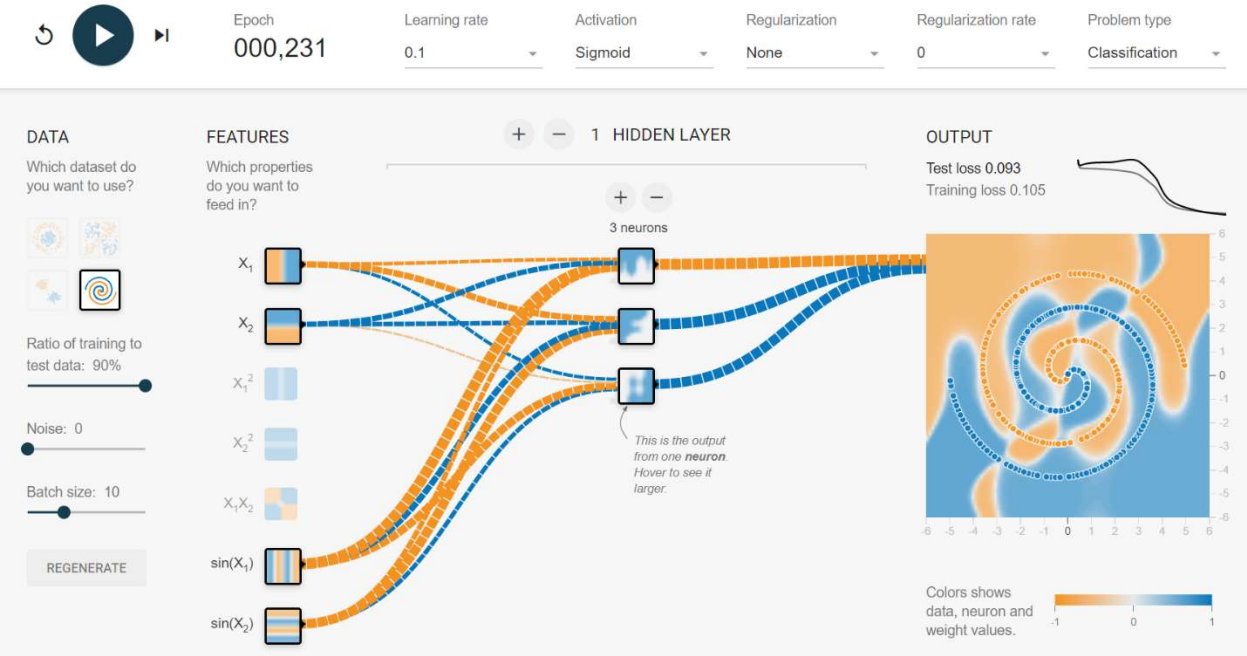
Play around with these settings and see how they affect your ability to learn classification of different data sets. Write down what you found and how you interpret the effects of these settings. (Question 23, 10 points)

Firstly, it is evident that the classification process is easier with certain datasets and harder with others. Even using a simple network with 7 inputs and 2 neurons in a single hidden layer, with the circle and exclusive or datasets the classification process is able to reach a loss of less than 0.01 respectively in about 0.030, 0.020 epochs. With the gaussian datasets it can do it immediately, while with the spiral dataset it hardly goes below 0.5.

Moreover, the features selected strongly influences the loss.
Working with the circle datasets, removing complex features like sin(x1), sin(x2) or x1*x2 does not affects particularly the loss, but if x1, x2, x1^2 and x2^2 are not present, the loss struggles to go below 0.3.
On the other hand, when training on the exclusive or dataset, using only the x1*x2 feature works fine, since it is very good to extract information from that specific datasets (the similarity between the feature and the dataset is also graphically evident).
This shows that feature selection matters a lot when working on different datasets.

We'll focus now on the spiral dataset to analyze how other settings influence the loss.
- Adding neurons to the last hidden layer without adding more layers does not necessarily increase the loss.
- Also adding multiple hidden layers using a linear activation function does not improve the performance, since a combination of linear function is still a linear function.
- Using ReLU or tanh activation functions slightly improve the performances.
- A strong improvement is achieved increasing the number of neurons for each hidden layer while using a nonlinear activation function.
For instance, adding 5 hidden layers with 5 neurons each makes the loss reach 0.05.
This happens because the network can now create more complex relations between the different layers, while using backpropagation to correct his weights.
- Backpropagation appears to depend on the batch size: when increasing the batch size to 30, the loss easily reaches 0.01 and when decreased to 1, the model generated is completely wrong, with a loss above 0.5.
A middle value like 5 or 10 makes the model perform even better.
- A low ratio of training data to test data causes the model to overfit since a small amount of training data does not allow for proper generalization. An extremely low ratio (e.g. 10%) causes the model to underfit, since there is not enough data for it to be trained properly.
- Selecting the right features can lead to an efficient model with a surprisingly low number of layers and neurons. This is the case with the spiral dataset and the model we built to answer question 24.

What is the minimum you need in the network to classify the spiral shape with a test set loss of below 0.1? (Question 24, 4 points)



Focusing on the important features improves the performance, allowing us to reach a test loss inferior to 0.1 quickly and with a small network with a single hidden layer with three neurons.

Exercise four: Low-level functions

In the following answers the code is given with a main method which allows to test the function on some given data. When relevant, function parameters can be changed by modifying the relative global variables.

Write a simple function that achieves the convolution operation efficiently for two dimensional and three-dimensional inputs. This should allow you 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 dimensional like the input layer. Give your code as the answer. (Question 25, 5 points)

```
import numpy


# the values below can be modified to test the function on different values
# paying attention to the @requires
IMG_SIZE = 5
KERNEL_SIZE = 3
STACK_DEPTH = 2
NUMBER_OF_FILTERS = 5



# set the limits for the values in the matrices
MIN_VALUE = -5
MAX_VALUE = 5


# @requires that the input features maps and the kernels are squares and that the
# size of the kernels is odd
# @requires that the kernels size is not greater than the images size
# compute a single feature map for each filter
def convolution(input_feature_maps, filters):

    output_features_maps = numpy.zeros((NUMBER_OF_FILTERS, IMG_SIZE, IMG_SIZE))
    for index, filter in enumerate(filters):
        # the actual convolution
        conv = apply_conv(input_feature_maps, filter)
        output_features_maps[index] = conv
    # returns a list of lists of feature maps
    return output_features_maps
```

```
# applies a filter to the stack of feature maps
def apply_conv(img, filter):

    img_shape = numpy.shape(img)
    size = img_shape[1]
    stack_depth = img_shape[0]
    shift = (filter.shape[1] - 1) // 2

    # the feature map to return
    conv = numpy.zeros([size, size])

    # padding
    padding_dimension = (stack_depth, size + 2 * shift, size + 2 * shift)
    padded_matrix = numpy.zeros(padding_dimension)
    padded_matrix[:, shift:size + shift, shift:size + shift] = img

    # matrix convolution
    for i in range(1, padded_matrix.shape[1] - 1):
        for j in range(1, padded_matrix.shape[2] - 1):
            img_fraction = padded_matrix[:, (i - shift): (i + shift) + 1,
                           (j - shift): (j + shift) + 1]
            tot = 0
            for depth in range(stack_depth):
                tot += numpy.sum(numpy.multiply(img_fraction[depth, :, :],
                                                filter[depth, :, :]))
            conv[i - 1][j - 1] = tot

    return conv


def main():

    feature_maps = numpy.array(numpy.random.randint(low=MIN_VALUE,
        high=MAX_VALUE, size=((STACK_DEPTH, IMG_SIZE, IMG_SIZE))))
    filters = list()
    for i in range(NUMBER_OF_FILTERS):
        fltr = numpy.array(numpy.random.randint(low= MIN_VALUE,
            high= MAX_VALUE, size=((STACK_DEPTH, KERNEL_SIZE, KERNEL_SIZE))))
        filters.append(fltr)

    convolved = convolution(feature_maps, filters)

    print("\nOriginal stack of images:\n\n", feature_maps)
    print("\n\nKernels:")
    for index, f in enumerate(filters):
        print("\nKernel ", index, "\n", f)
    print("\n\noutput features maps:")
    for feature_map in convolved:
            print("\n", feature_map)


if __name__ == "__main__":
    main()
```

Write a simple function that achieves rectified linear (relu) activation, with a threshold at zero. Give your code as the answer. (Question 26, 2 points)

```
import numpy

def relu(value):

    return numpy.maximum(0, value)


def main():

    print("\n", "output of the relu function when applied to a matrix:\n")
    matrix = numpy.array((numpy.random.randint(low = -10, high = 10, size=(3, 3))))
    print("Original matrix:\n", matrix)
    print("\nAfter relu:\n", relu(matrix))

if name == "main":
    main()
```

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. Give your code as the answer. (Question 27, 3 points)

```
import numpy

# the size of the image, the spatial extent and the stride can be modified here
IMG_SIZE = 7
SPATIAL_EXTENT = 3
STRIDE = 2


def max_pooling (input_feature_map, size, stride):

    w = (input_feature_map.shape[0] - size) // stride + 1
    pooled_map = numpy.zeros((w, w))
    pi = -1
    for i in range(0, input_feature_map.shape[0] - size + 1, stride):
        pi += 1
        pj = -1
        for j in range(0, input_feature_map.shape[0] - size + 1, stride):
            pj += 1
            img_fraction = input_feature_map[i : i+size, j : j+size]
            max_value = numpy.max(img_fraction)
            pooled_map[pi][pj] = max_value
    return pooled_map



def main():
    img1 = numpy.random.rand(IMG_SIZE, IMG_SIZE)
    img1 = numpy.around(img1 * 10, 0)
    print("\ninput feature map:")
    print("\n", img1)
    pooled = max_pooling(img1, SPATIAL_EXTENT, STRIDE)
    print("\npooled feature map:")
    print("\n", pooled)

if __name__ == "__main__":
    main()
```

Write a simple function that achieves normalisation within each feature map, modifying the feature map so that its mean value is zero and its standard deviation is one. Give your code as the answer. (Question 28, 4 points)

```
import numpy

ROUND_DIGITS = 15


def normalize(feature_map):

    return (feature_map - feature_map.mean()) / feature_map.std()


def main():
    i = numpy.array([[1,7,13],[-3,2,3],[5,21,3]])
    normalized = normalize(i)
    print("\n\nThe normalized array\n")
    print(normalized)
    print("\n\nMean of the elements, rounding to the 15th digit\n")
    print(round((numpy.mean(normalized)), ROUND_DIGITS))
    print("\n\nVariance of the elements, rounding to the 15th digit\n")
    print(round(numpy.var(normalized), ROUND_DIGITS))


if __name__ == "__main__":
    main()
```

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 a stack of feature maps. The stack of feature maps will typically be flattened into a 1-dimensional matrix first. (Question 29, 5 points)

```
import numpy


def full_connected_layer(input_layer, output_size):

    flattened_input = input_layer.flatten()
    input_size = len(flattened_input)
    output_layer = [0] * output_size
    weights = numpy.random.rand(output_size, input_size)
    for i in range(output_size):
        value = numpy.sum(numpy.multiply(flattened_input, weights[i]))
        output_layer[i] = value

    return output_layer, weights


def main():

    input_layer = numpy.array([[1,2,3,4],[5,6,7,8]])
    output_layer, weights = full_connected_layer(input_layer, 5)
    print("\nValues of the output layer's neurons:")
    print("\n", output_layer)
    print("\nValues of the weights:")
    for w in weights:
        print("\n", w)


if __name__ == "__main__":
    main()
```

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 (normalised exponential) function. (Question 30, 2 points)

```python
import numpy


def softmax(x):

    e_x = numpy.exp(x)
    return e_x / sum(e_x)


def main():
    print(softmax([2.0, 1.0, 0.1]))


if __name__ == "__main__":
    main()
```

Explain the principle of backpropagation of error in plain English. This can be answered with minimal mathematical content, and should be IN YOUR OWN WORDS. What is backpropagation trying to achieve, and how does it do so? (Question 31, 8 points)

Deep neural networks are learning networks which extract and transform features in order to perform operation like regression or classification.

They are typically composed by multiples layers, and neurons of each layers are activated depending on neurons of the previous layers. The weights between these layers (or the filters weights in DCNN) are initially randomly set and the network would perform poorly if they could not be modified.
The network needs then to learn how to link the layers by progressively modifying the links between them; these links are called weights.

In order to adjust the weights, the network takes advantage of a loss function, a function of the weights that returns a value corresponding how bad the network is performing.
Namely, in classification, only one neuron of the output layer is expected to assume a value of one, while the other values should be zeros. A common loss function, the quadratic loss function, compute the difference between each output neuron activation value and the expected value (0 or 1) and returns the sum of the squares of all these differences.
The activation value of the output neurons depends on the weights between the last hidden layer and the last layer and on the activation value of the last hidden layer (which itself depends on the weight of that link it to the previous layer and so on…). Therefore, it is affected by all the weights in the network.
Being the loss value a function of the weights, it is possible to look at which weights are most responsible for increasing it, by walking back through the network and using the chain rule to compute the derivatives of the loss function with regard to the weights of the networks.

Backpropagation does this and exploits a mathematic technique called gradient descent in order to modify the weights to minimize the error.
This technique can be easily understood looking at the two-dimension scenario. Being the starting loss value the pick of a hill, the goal is to find the direction over the x and y axes where the loss value decreases most (i.e. the direction opposite to the gradient in that point).
After moving in that direction, the gradient is re-calculated and another step is done in the computed direction, until the loss goes below the desired value, or it cannot decrease anymore.
One problem with this technique occurs when there is the risk of encountering a local minimum for the loss function instead of a global minimum. To prevent this there are dedicate techniques, but an easy solution is to use a convex loss function.

By generalizing this concept to multi-variables functions it is possible to find the best values for the weights of the network in order to minimize the lost. In these way links are created between the layers that a human cannot really understand (and then impossible to program a priori), and complex patterns emerge that allow the network to perform the task it is required to do.

Bonus questions

Describe the process of backpropagation in mathematical terms. Here, explain (in English) what each equation you give does, and relate this to the answers given in Question 31. (Question 32, 5 points).

As said in the answer to question 31, in a deep neural network the activation value of the neurons in each layer depends on the previous layer. Namely, this relation can be expressed as:

eq 1. $z(2) = XW(1)$ for the first hidden layer and

eq 2. $z(i+1) = a(i)W(i)$ for the next layers

where:

- X is the input matrix, where each row contains a sample
- W(i) is a matrix containing the weights between the (i-1)-th hidden layer and the i-th hidden layer
- z(i) is a matrix where each row contains the activation value of the i-th layer neurons for a specific sample before applying the activation function
- a(i) is a matrix where each row contains the activation value of the i-th layer neurons for a specific sample after applying the activation function

From these expressions is evident how the i-th layers is a function of all the previous layers and weights and, if the activation function f(x) is not linear, it is not a linear function.

By applying f(x) to z(N), where N stays for the index of the output layers, we obtain the activation values of the output layer's neuron, Y, where each row r contains the values for a specific sample.
Each element y(r,i) represent the activation value of a specific neuron for a specific sample and for each combination of neuron and sample it exists an expected value k(r,i), which is the goal to reach at the end of the training.
The loss function introduced in the answer to question 31 would then be:

eq 3. $J = \frac{1}{2} \sum ((y(r,i)-k(r,i))^2)$

Where the coefficient ½ is added to make the calculus easier.
But from eq 1 it is evident that J = J(X, W), where X represents the input and W the weights of the network. The input cannot be modified, then the goal of backpropagation is to modify the weight in order to minimize the loss.
This can be done by calculating the gradient of J and by modifying the current weights subtracting the gradient. Since the gradient represents the grow direction, subtracting it from the weights will decrease the value of the loss function. In doing this subtraction, the gradient is multiplied by a coefficient η, the

learning rate, which determines how much the weights are modified each time the gradient is re-calculated.

eq 4. $W(t+1) = W(t) - \boldsymbol{\eta}\nabla W\, J(y, y; W)$

This technique goes under the name of gradient descent.
The gradient is calculated taking advantage of the chain rules for derivatives, which allow us to find the derivative of loss function with respect to weight W.
Being the loss equal to

eq 5. $loss = J(R(Z(AW)))$

where:

- A contains the activation values of the last hidden layer
- W contains the weights between the last hidden layer and the output
- Z is the input in the last layer, hence the product between the activation values of the last hidden layer and the weights
- R is the activation function

we obtain the following:

eq 6. $J'(W)=J'(R)\cdot R'(Z)\cdot Z'(W)$

where:

- $Z'(W) = A$, hence for each weight the derivative of Z with regard to W is the activation value of the neuron the weight is linking
- $R'(Z) = 0$ if $Z<0$, 1 if $Z>0$
- $J'(R) = y\text{-}k$ (here is where the coefficient turns out to make things simpler)

By extending the chain rule to previous layers it is possible to get the derivative of the loss function with respect to each weight, thus calculating the gradient and adjusting the weights to improve the performance.