Deep Learning Lab Session

Artificial Neural Networks for Handwritten Digits Recognition

Corentin RAFFLIN

Introduction

During this lab session, you will implement, train and test a Neural Network for the Handwritten Digits Recognition problem [1] with different settings of hyperparameters. You will use the MNIST dataset which was constructed from scanned documents available from the National Institute of Standards and Technology (NIST). Images of digits were taken from a variety of scanned documents, normalized in size and centered.

Figure 1: MNIST digits examples

This assignment includes a written part of programms to help you understand how to build and train your neural net and then to test your code and get results.

1. NeuralNetwork.py
2. transfer_functions.py
3. utils.py

Functions defined inside the python files mentionned above can be imported using the python command "from filename import function".

You will use the following libraries:

1. numpy : for creating arrays and using methods to manipulate arrays;
2. matplotlib : for making plots.

Before starting the lab, please launch the cell below. After that, you may not need to do any imports during the lab.

In [ ]:

```
# All imports
from NeuralNetwork import NeuralNetwork
from transfer_functions import *
from utils import *
import numpy as np
import matplotlib
```

Section 1 : Your First Neural Network

Part 1: Before designing and writing your code, you will first work on a neural network by hand. Consider the following neural network with two inputs $x = (x_1, x_2)$, one hidden layer and a single output unit y. The initial weights are set to random values. Neurons 6 and 7 represent biases. Bias values are equal to 1. You will consider a training sample whose feature vector is $x = (0.8, 0.2)$ and whose label is $y = 0.4$.

Assume that neurons have a sigmoid activation function $f(x) = \frac{1}{(1+e^{-x})}$. The loss function L is a Mean Squared Error (MSE): if o denotes the output of the neural network, then the loss for a given sample (o, y) is $L(o, y) = ||o - y||^2$. In the following, you will assume that if you want to backpropagate the error on a whole batch, you will backpropagate the average error on that batch. More formally, let $((x^{(1)}, y^{(1)}), \dots,$ be a batch and $o^{(k)}$ the

$(x^{(1)}, y^{(1)}))$

output associated to $x^{(k)}$. Then the total error $\bar{L}$ will be as follows:

$$\bar{L} = \frac{1}{N}\sum_{k=1}^{N} L(o^{(k)}, y^{(k)})$$.

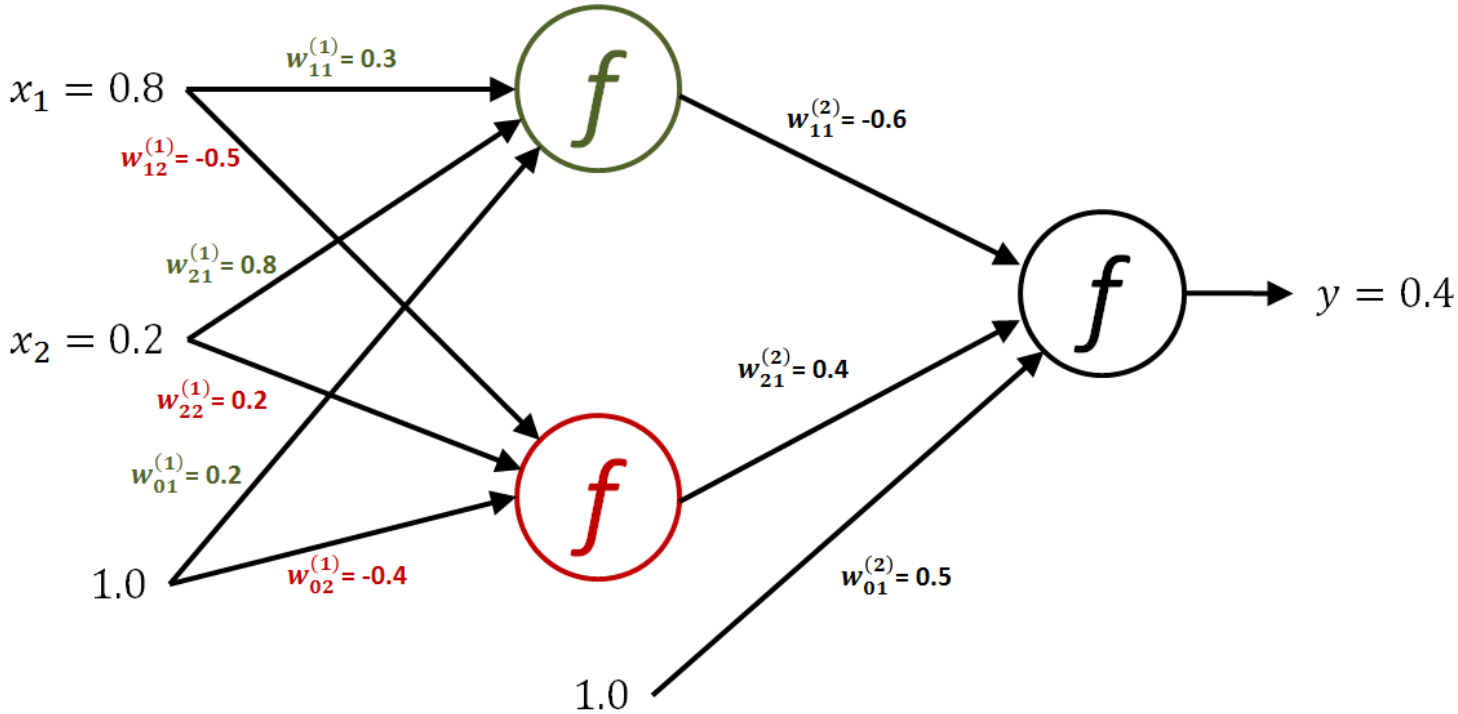


Figure 2: Neural network

Question 1.1.1: Compute the new values of weights $w_{i,j}$ after a forward pass and a backward pass, and the outputs of the neural network before and after the backward path, when the learning rate is λ=5. $w_{i,j}$ is the weight of the connexion between neuron i and neuron j. Please detail your computations in the cell below and print your answers.

In [ ]:

```
lr = 5.0
x0, x1, x2 = 1, 0.8, 0.2 # x0=1 added
w1_01, w1_11, w1_21, w1_02, w1_12, w1_22 = 0.2, 0.3, 0.8, -0.4, -0.5, 0.2
w2_01, w2_11, w2_21 = 0.5, -0.6, 0.4
y = 0.4

u1_1 = x0*w1_01+x1*w1_11+x2*w1_21 # Activation of the green neuron
u1_2 = x0*w1_02+x1*w1_12+x2*w1_22 # Activation of the red neuron

o1_0 = 1 # Output of hidden neuron
o1_1 = sigmoid(u1_1) # Output of the green neuron
o1_2 = sigmoid(u1_2) # Output of the red neuron

u2_1 = o1_0*w2_01+o1_1*w2_11+o1_2*w2_21 # Activation of black neuron
o2_1 = sigmoid(u2_1) # Output of the black neuron


print("=== FORWARD PASS 1 ===")
print("o =", o2_1)


dL_du2 = 2*(o2_1-y)*dsigmoid(u2_1) #Partial error gradient at output layer

# Partial derivatives of the loss wrt weights of the second layer
dL_w2_01 = dL_du2*o1_0
dL_w2_11 = dL_du2*o1_1
```

```
dL_w2_21 = dL_du2*o1_2


# Partial derivatives of the loss wrt weights of the first layer
dL_w1_01 = x0 * dL_du2 * w2_11 * dsigmoid(u1_1)
dL_w1_11 = x1 * dL_du2 * w2_11 * dsigmoid(u1_1)
dL_w1_21 = x2 * dL_du2 * w2_11 * dsigmoid(u1_1)
dL_w1_02 = x0 * dL_du2 * w2_21 * dsigmoid(u1_2)
dL_w1_12 = x1 * dL_du2 * w2_21 * dsigmoid(u1_2)
dL_w1_22 = x2 * dL_du2 * w2_21 * dsigmoid(u1_2)

# Weights updates
w1_01 -= lr*dL_w1_01
w1_11 -= lr*dL_w1_11
w1_21 -= lr*dL_w1_21
w1_02 -= lr*dL_w1_02
w1_12 -= lr*dL_w1_12
w1_22 -= lr*dL_w1_22
w2_01 -= lr*dL_w2_01
w2_11 -= lr*dL_w2_11
w2_21 -= lr*dL_w2_21

print("=== BACKWARD PASS ===")
print("w1_01 =", w1_01)
print("w1_11 =", w1_11)
print("w1_21 =", w1_21)
print("w1_02 =", w1_02)
print("w1_12 =", w1_12)
print("w1_22 =", w1_22)
print("w2_01 =", w2_01)
print("w2_11 =", w2_11)
print("w2_21 =", w2_21)

u1_1 = x0*w1_01+x1*w1_11+x2*w1_21 # Activation of the green neuron
u1_2 = x0*w1_02+x1*w1_12+x2*w1_22 # Activation of the red neuron

o1_0 = 1 # Output of hidden neuron
o1_1 = sigmoid(u1_1) # Output of the green neuron
o1_2 = sigmoid(u1_2) # Output of the red neuron

u2_1 = o1_0*w2_01+o1_1*w2_11+o1_2*w2_21 # Activation of black neuron
o2_1 = sigmoid(u2_1) # Output of the black neuron

print("=== FORWARD PASS 2 ===")
print("o =", o2_1)
```

```
=== FORWARD PASS 1 ===
o = 0.5597295991095778
=== BACKWARD PASS ===
w1_01 = 0.25403317902693395
w1_11 = 0.3432265432215471
w1_21 = 0.8108066358053868
w1_02 = -0.4341841377344243
w1_12 = -0.5273473101875394
w1_22 = 0.19316317245311515
w2_01 = 0.10637455535192764
w2_11 = -0.8541467506279606
w2_21 = 0.27457272177725717
=== FORWARD PASS 2 ===
o = 0.40648823589210104
```

Part 2: Neural Network Implementation

In Part 1, you computed weight updates for one sample. This is what we do for the stochastic gradient descent algorithm. However in the rest of the lab, you will be to implement the batch version of the gradient descent.

Please read all source files carefully and understand the data structures and all functions. You are to complete the

missing code. First you should define the neural network (using the NeuralNetwork class, see in the [NeuralNetwork.py] file) and reinitialise weights. Then you will need to complete the feedforward() and the backpropagate() functions.

Question 1.2.1: Implement the feedforward() function.

In [ ]:

```
class NeuralNetwork(NeuralNetwork):
    def feedforward(self, inputs):
        transfer_f = self.transfer_f
        inputs = [x + [1.] for x in inputs]
        self.input = np.array(inputs) # Shape = [batch_size, number_of_input_values+1]
        u_1 = np.dot(self.input, self.W_input_to_hidden)# Compute activations for the hidde
n layer
              # Shape of u_1 should be [batch_size, number_of_neurons_in_hidden_layer]
        self.u_hidden = u_1
        self.o_hidden = np.ones((u_1.shape[0], u_1.shape[1]+1)) # Shape = [batch_size, numb
er_of_hidden_values+1]
        self.o_hidden[:, :-1] = transfer_f(self.u_hidden) # Compute output of hidden layer
        u_2 = np.dot(self.o_hidden, self.W_hidden_to_output)# Compute activations for the ou
tput layer
        self.u_output = u_2
        self.o_output = transfer_f(self.u_output) # Compute output of output layer
```

Question 1.2.2: Test your implementation: create the Neural Network defined in Part 1 and see if the feedforward() function you implemented gives the same results as the ones you found by hand.

In [ ]:

```
# First define your neural network
input_layer_size=2
hidden_layer_size=2
output_layer_size=1
model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)

# Then initialize the weights according to Figure 2
W_input_to_hidden = np.array([[0.3, -0.5], [0.8, 0.2], [0.2, -0.4]])
W_hidden_to_output = np.array([[-0.6], [0.4], [0.5]])
model.weights_init(W_input_to_hidden, W_hidden_to_output)

# Feed test values
test = [[0.8, 0.2]]
model.feedforward(test)

# Print the output
print("Output =", model.o_output[0,0])
```

```
Output = 0.5597295991095776
```

Question 1.2.3: Implement the backpropagate() function.

In [ ]:

```
class NeuralNetwork(NeuralNetwork):
    def backpropagate(self, targets, learning_rate=5.0):
        transfer_df = self.transfer_df
        l = learning_rate
        targets = np.array(targets) # Target outputs
        self.dL_du_output = 2*(self.o_output-targets)*transfer_df(self.u_output) # Compute
partial derivative of loss with respect to activations of output layer
        self.dL_du_hidden = np.dot(self.dL_du_output, self.W_hidden_to_output.T[:,:-1])*tra
nsfer_df(self.u_hidden) # Compute partial derivative of loss with respect to activations of
hidden layer
        # Compute partial derivative of loss with respect to weights
```

```
        dW_input_to_hidden = np.dot(self.input.T,self.dL_du_hidden)
        dW_hidden_to_output = np.dot(self.o_hidden.T, self.dL_du_output)
        # Make updates
        self.W_hidden_to_output = self.W_hidden_to_output - (l * dW_hidden_to_output)/len(t
argets)
        self.W_input_to_hidden = self.W_input_to_hidden - l*dW_input_to_hidden/len(targets)
```

Question 1.2.4: Test your implementation: create the Neural Network defined in Part 1 and see if the backpropagate() function you implemented gives the same weight updates as the ones you found by hand. Do another forward pass and see if the new output is the same as the one you obtained in Question 1.1.1.

In [ ]:

```
# First define your neural network
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)

# Then initialize the weights according to Figure 2
W_input_to_hidden = np.array([[0.3, -0.5], [0.8, 0.2], [0.2, -0.4]])
W_hidden_to_output = np.array([[-0.6], [0.4], [0.5]])
model.weights_init(W_input_to_hidden, W_hidden_to_output)

# Feed test values
test = [[0.8, 0.2]]
model.feedforward(test)

# Backpropagate
targets = [[0.4]]
model.backpropagate(targets)

# Print weights
print("W_input_to_hidden =", model.W_input_to_hidden)
print("W_hidden_to_output =", model.W_hidden_to_output)

# Feed test values again
model.feedforward(test)

# Print the output
print("Output =", model.o_output)
```

```
W_input_to_hidden = [[ 0.34322654 -0.52734731]
 [ 0.81080664  0.19316317]
 [ 0.25403318 -0.43418414]]
W_hidden_to_output = [[-0.85414675]
 [ 0.27457272]
 [ 0.10637456]]
Output = [[0.40648824]]
```

Checked your implementations and found that everything was fine? Congratulations! You can move to the next section.

Section 2: Handwritten Digits Recognition

The MNIST dataset consists of handwritten digit images. It is split into a training set containing 60,000 samples and a test set containing 10,000 samples. In this Lab Session, the official training set of 60,000 images is divided into an actual training set of 50,000 samples a validation set of 10,000 samples. All digit images have been size-normalized and centered in a fixed size image of 28 x 28 pixels. Images are stored in byte form: you will use the NumPy python library to convert data files into NumPy arrays that you will use to train your Neural Networks.

You will first work with a small subset of MNIST (1000 samples), then on a very small subset of MNIST (10 samples),

and eventually run a model on the whole one.

The MNIST dataset is available in the Data folder. To get the training, testing and validation data, run the load_data() function.

In [ ]:

```
# Just run that cell ;-)
training_data, validation_data, test_data = load_data()
small_training_data = (training_data[0][:1000], training_data[1][:1000])
small_validation_data = (validation_data[0][:200], validation_data[1][:200])
indices = [1, 3, 5, 7, 2, 0, 13, 15, 17, 4]
vsmall_training_data = ([training_data[0][i] for i in indices], [training_data[1][i] for i
in indices])
```

```
Loading MNIST data .....
Done.
```

In [ ]:

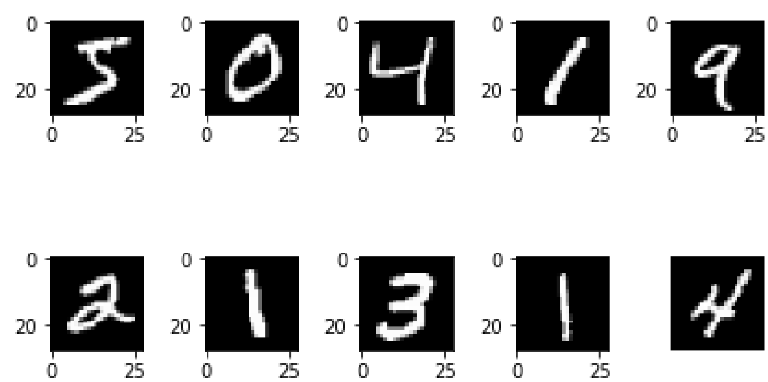
```
print(1*3)
```

In [ ]:

```
print("test")
```

In [ ]:

```
# And you can run that cell if you want to see what the MNIST dataset looks like
ROW = 2
COLUMN = 5
for i in range(ROW * COLUMN):
    # train[i][0] is i-th image data with size 28x28
    image = np.array(training_data[0][i]).reshape(28, 28)
    plt.subplot(ROW, COLUMN, i+1)
    plt.imshow(image, cmap='gray')  # cmap='gray' is for black and white picture.
plt.axis('off')  # do not show axis value
plt.tight_layout()   # automatic padding between subplots
plt.show()
```



Part 1: Build a bigger Neural Network

The input layer of the neural network that you will build contains neurons encoding the values of the input pixels. The training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits. Thus, the input layer contains 784=28×28 units. The second layer of the network is a hidden layer. We set the number of neurons in the hidden layer to 30. The output layer contains 10 neurons.

Question 2.1.1: Create the network described above using the NeuralNetwork class.

In [ ]:

```
input_layer_size = 784
```

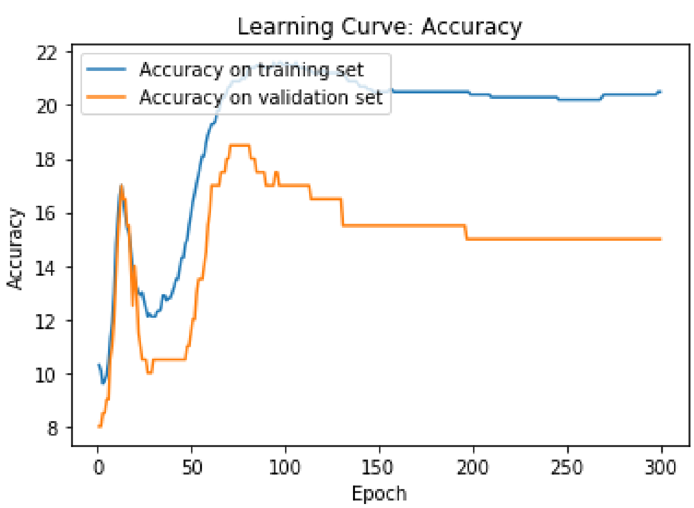
```
hidden_layer_size = 30
output_layer_size = 10

# Define your neural network
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)
```

Question 2.1.2: Train your Neural Network on the small subset of MNIST (300 iterations) and print the new accuracy on test data. You will use small_validation_data for validation. Try different learning rates (0.1, 1.0, 10.0). You should use the train() function of the NeuralNetwork class to train your network, and the weights_init() function to reinitialize weights between tests. Print the accuracy of each model on test data using the predict() function.

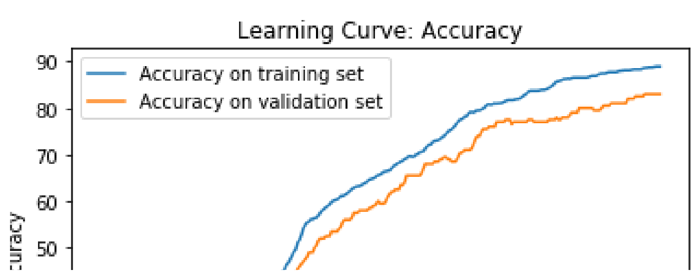
In [ ]:

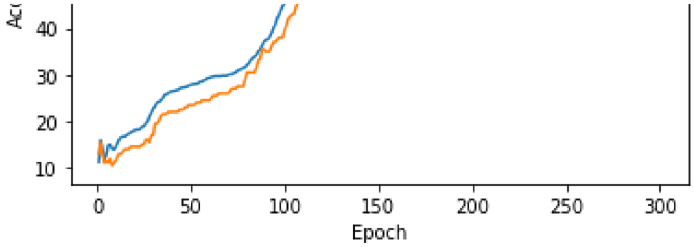
```
# Train NN and print accuracy on test data

# Learning rate 0.1
print("Model - 30 hidden neurons - learning rate = 0.1 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 0.1)
print("Prediction : ", mnist_model.predict(test_data))

# Learning rate 1.
print("\nModel - 30 hidden neurons - learning rate = 1 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 1.0)
print("Prediction : ", mnist_model.predict(test_data))

# Learning rate 10.
print("\nModel - 30 hidden neurons - learning rate = 10 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 10.0)
print("Prediction : ", mnist_model.predict(test_data))
```

```
Model - 30 hidden neurons - learning rate = 0.1 - small_validation_data
Training time: 17.145373582839966
```

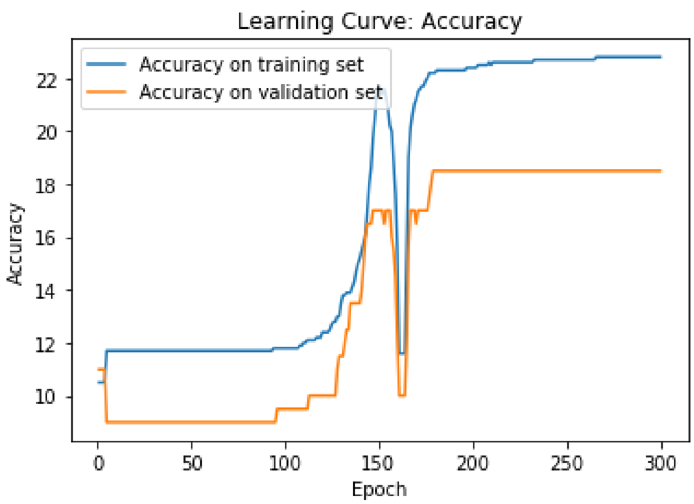



```
Prediction :  1926

Model - 30 hidden neurons - learning rate = 1 - small_validation_data
Training time: 16.54010248184204
```






```
Prediction :  8307

Model - 30 hidden neurons - learning rate = 10 - small_validation_data
Training time: 17.283888816833496
```




```
Prediction :  2114
```

Comment

In these plots, we can see that for our neural network with 30 hidden neurons, a good trade off for the learning rate seems to be a learning rate of 1. Indeed for the two other tested learning rates 10 and 0.1 our neural network gives very poor results on both validation and trainning data with the accuracy being under 30, whereas with the learning rate of 1, the accuracy is over 80 on both data.

If we look at the prediction given over the 10,000 data, we can see that the prediction is way better with the model having a learning rate of 1 : 83% accuracy on the test set versus around 20% for the two other models.

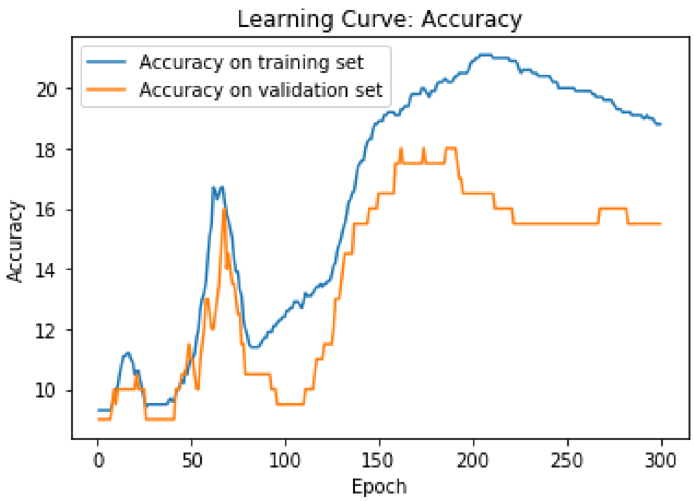
Question 2.1.3: Do the same with 15 and 75 hidden neurons.

In [ ]:

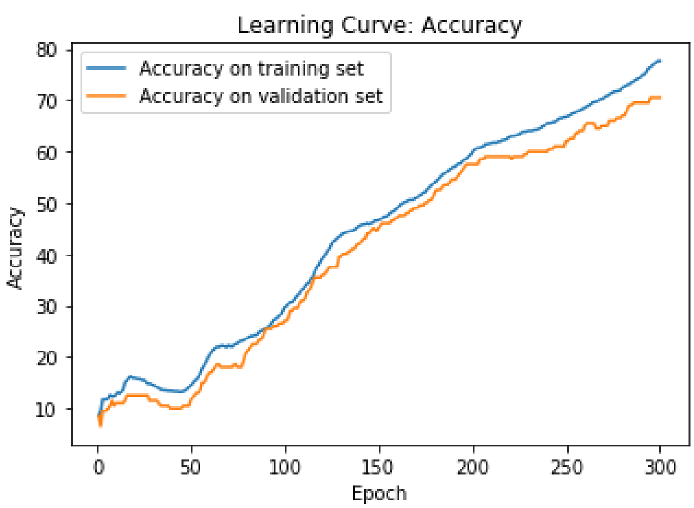
```
# Define your neural network
# 15 hidden neurons
hidden_layer_size = 15
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)

# Learning rate 0.1
print("Model - 15 hidden neurons - learning rate = 0.1 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 0.1)
print("Prediction : ", mnist_model.predict(test_data))

# Learning rate 1.
print("\nModel - 15 hidden neurons - learning rate = 1 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 1.0)
print("Prediction : ", mnist_model.predict(test_data))
```

```python
# Learning rate 10.
print("\nModel - 15 hidden neurons - learning rate = 10 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 10.0)
print("Prediction : ", mnist_model.predict(test_data))
```

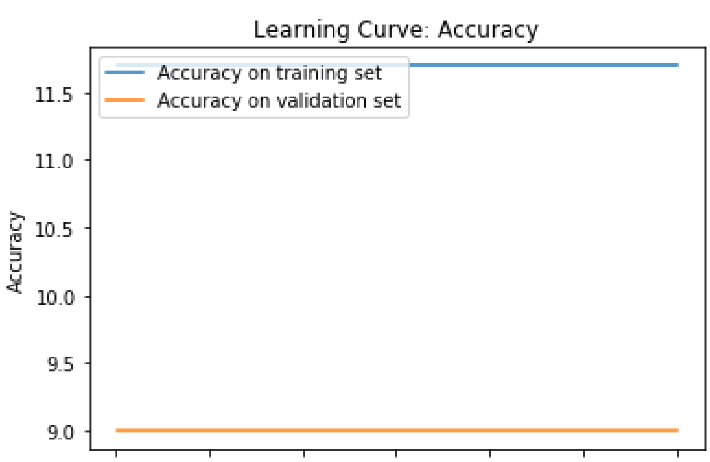
```
Model - 15 hidden neurons - learning rate = 0.1 - small_validation_data
Training time: 18.982152223587036
```




```
Prediction :  1693

Model - 15 hidden neurons - learning rate = 1 - small_validation_data
Training time: 19.049890756607056
```




```
Prediction :  7232

Model - 15 hidden neurons - learning rate = 10 - small_validation_data
Training time: 18.55118179321289
```




```
Prediction :  1028
```

Comment

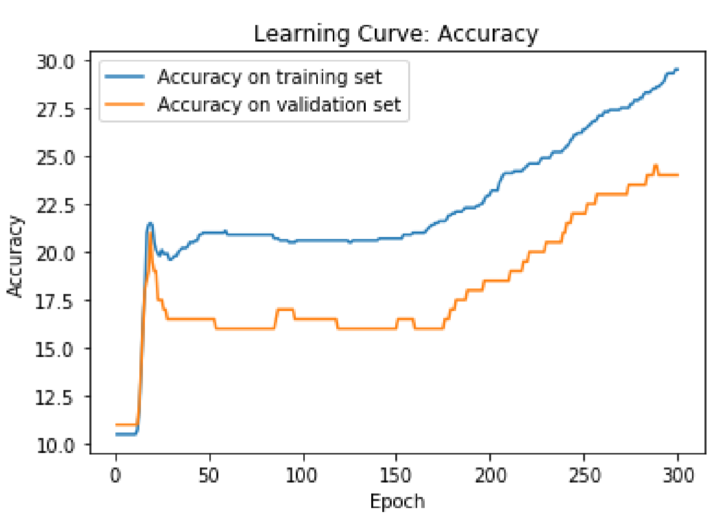
For this neural network with 15 hidden neurons, 1 is still the best value for the learning rate, as before we can compare the accuracy at the end of the 300 iterations on the training and validation sets. When the learning rate is 0.1 or 10, as for the neural network with 30 hidden neurons, the neural network with 15 hidden neurons does not seem to converge toward a relevant state.

Yet, this model (15 hidden neurons - learning rate 1) is less efficient than for the model with 30 hidden neurons and a learning rate of 1 seeing that the maximum accuracy is inferior for this model and also the prediction is less good. Nevertheless there may be less overfitting with this model due to the fact that the neural network is less complex so we are less likely to overfit but perfomance decreases in average.

In [ ]:

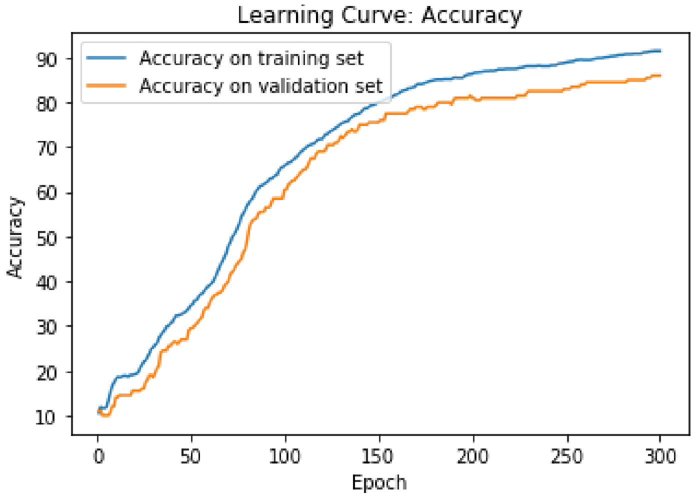
```
# Define your neural network
# 75 hidden neurons
hidden_layer_size=75
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)

# Learning rate 0.1
print("Model - 75 hidden neurons - learning rate = 0.1 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 0.1)
print("Prediction : ", mnist_model.predict(test_data))

# Learning rate 1.
print("\nModel - 75 hidden neurons - learning rate = 1 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 1.0)
print("Prediction : ", mnist_model.predict(test_data))

# Learning rate 10.
print("\nModel - 75 hidden neurons - learning rate = 10 - small_validation_data")
mnist_model.weights_init()
mnist_model.train(small_training_data, small_validation_data, 300, 10.0)
print("Prediction : ", mnist_model.predict(test_data))
```

```
Model - 75 hidden neurons - learning rate = 0.1 - small_validation_data
Training time: 21.670027017593384
```

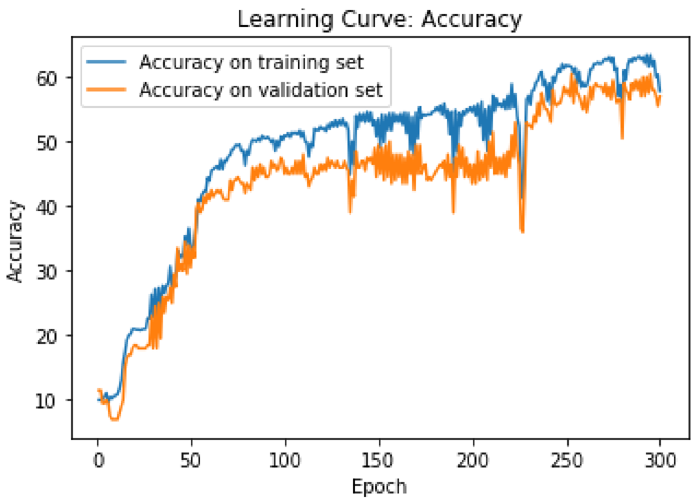



```
Prediction :  2719

Model - 75 hidden neurons - learning rate = 1 - small_validation_data
```

```
Training time: 24.57131314277649
```




```
Prediction :  8485

Model - 75 hidden neurons - learning rate = 10 - small_validation_data
Training time: 23.967211961746216
```




```
Prediction :  5181
```

Comment

For this neural network with 75 hidden neurons, 1 is still the best value for the learning rate as it gives the best result (highest accuracy) for both the validation and training sets. For all the different learning rates, we can notice that there is a better accuracy with 75 hidden neurons than with 15 and 30. This can be explained by the fact that we use a more complex neural network. Nevertheless, because of this complexity we can more or less notice that for for the learning rates 0.1 and 1 there are more overfit than for the neural network with less hidden neurons.

We can notice the influence of the learning rate in these plots: when it is too small (0.1 in our case) the perfomance of the neural network increases very slowly at each iteration. When the learning rate is too high (10 in our case) the accuracy oscillates a lot and the accuracy seems to never be as good as the accuracy of the neural network train with a good learning rate (1 in our case).

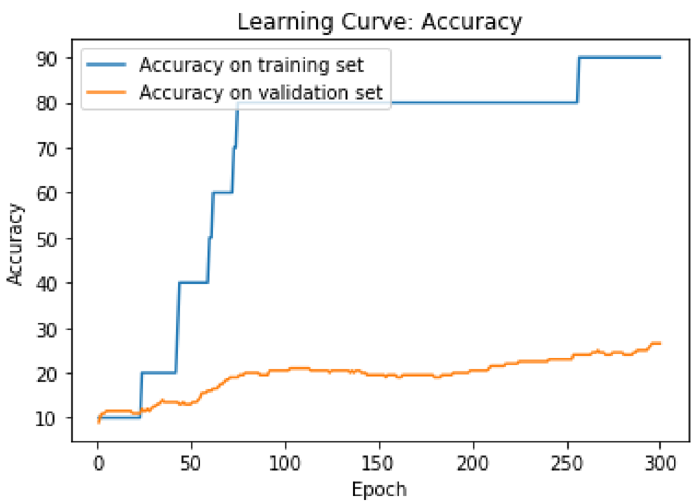
Question 2.1.3: Repeat Questions 2.1.2 and 2.1.3 on the very small datasets. You will use small_validation_data for validation.

In [ ]:

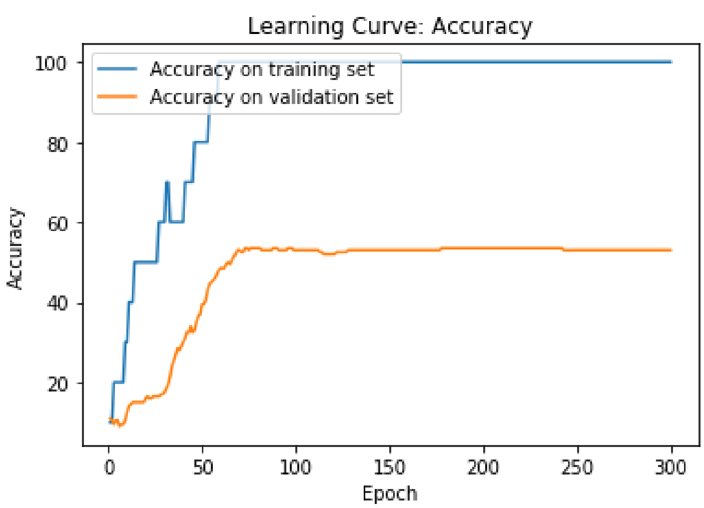
```
# Train NN and print accuracy on test data
# 30 hidden neurons
```

```
hidden_layer_size=30
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)
# Learning rate 0.1
print("Model - 30 hidden neurons - learning rate = 0.1 - small_validation_data - vsmall_tra
ining_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 0.1)
print("Prediction : ", mnist_model.predict(test_data))
# Learning rate 1.
print("\nModel - 30 hidden neurons - learning rate = 1 - small_validation_data - vsmall_tra
ining_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 1.0)
print("Prediction : ", mnist_model.predict(test_data))
# Learning rate 10.
print("\nModel - 30 hidden neurons - learning rate = 10 - small_validation_data - vsmall_tr
aining_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 10.0)
print("Prediction : ", mnist_model.predict(test_data))
```

```
Model - 30 hidden neurons - learning rate = 0.1 - small_validation_data - vsmall_training_da
ta
Training time: 1.8186264038085938
```

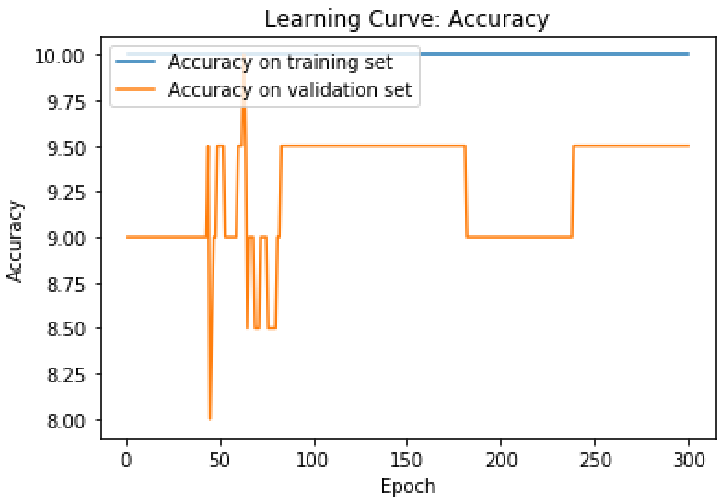



```
Prediction :  2667

Model - 30 hidden neurons - learning rate = 1 - small_validation_data - vsmall_training_data
Training time: 1.4791676998138428
```




```
Prediction :  5240

Model - 30 hidden neurons - learning rate = 10 - small_validation_data - vsmall_training_dat
a
```

```
Training time: 1.4035751819610596
```

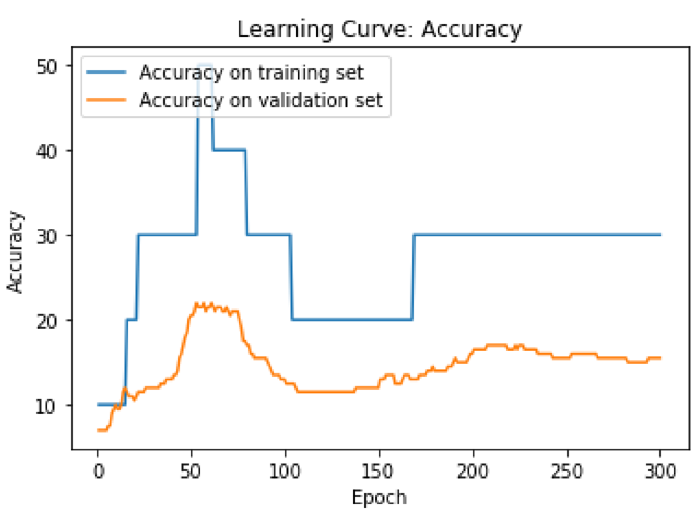



```
Prediction :  1135
```

In [ ]:

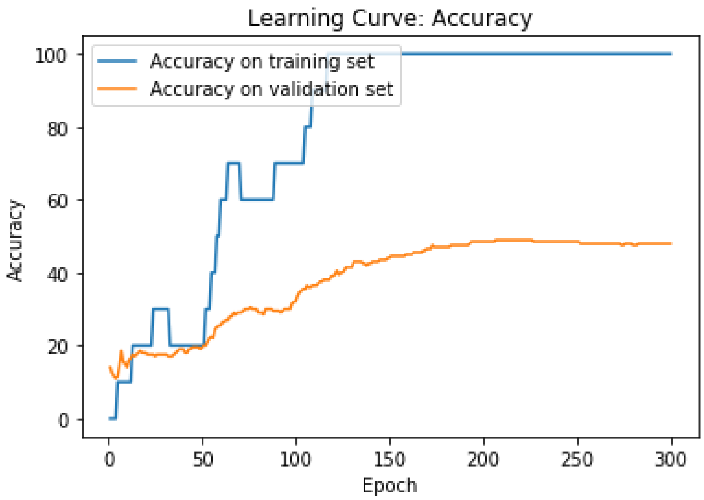
```python
# 15 hidden neurons
hidden_layer_size=15
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)
# Learning rate 0.1
print("Model - 15 hidden neurons - learning rate = 0.1 - small_validation_data - vsmall_tra
ining_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 0.1)
print("Prediction : ", mnist_model.predict(test_data))
# Learning rate 1.
print("Model - 15 hidden neurons - learning rate = 1 - small_validation_data - vsmall_train
ing_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 1.0)
print("Prediction : ", mnist_model.predict(test_data))
# Learning rate 10.
print("Model - 15 hidden neurons - learning rate = 10 - small_validation_data - vsmall_train
ing_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 10.0)
print("Prediction : ", mnist_model.predict(test_data))
```

```
Model - 15 hidden neurons - learning rate = 0.1 - small_validation_data - vsmall_training_da
ta
Training time: 1.575927734375
```

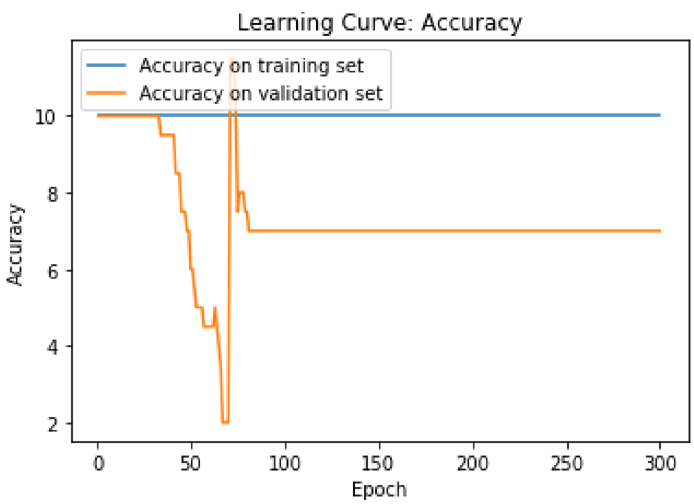



```
Prediction :  1625
```

```
Model - 15 hidden neurons - learning rate = 1 - small_validation_data - vsmall_training_data
Training time: 1.3760077953338623
```




```
Prediction :  4889
Model - 15 hidden neurons - learning rate = 10 - small_validation_data - vsmall_training_dat
a
Training time: 1.507505178451538
```

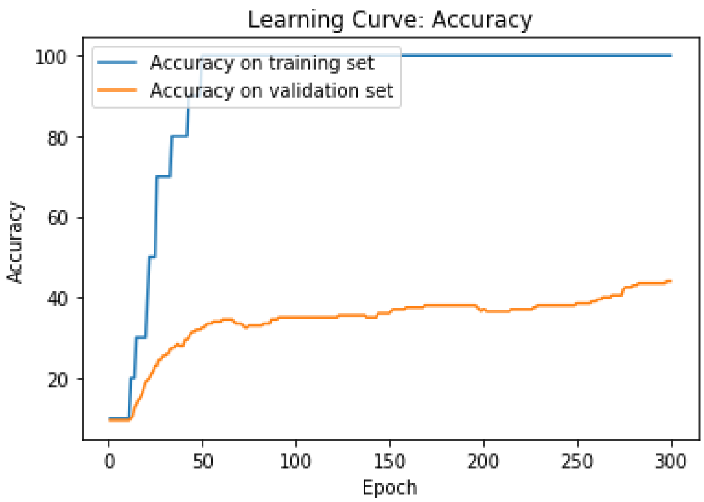



```
Prediction :  933
```

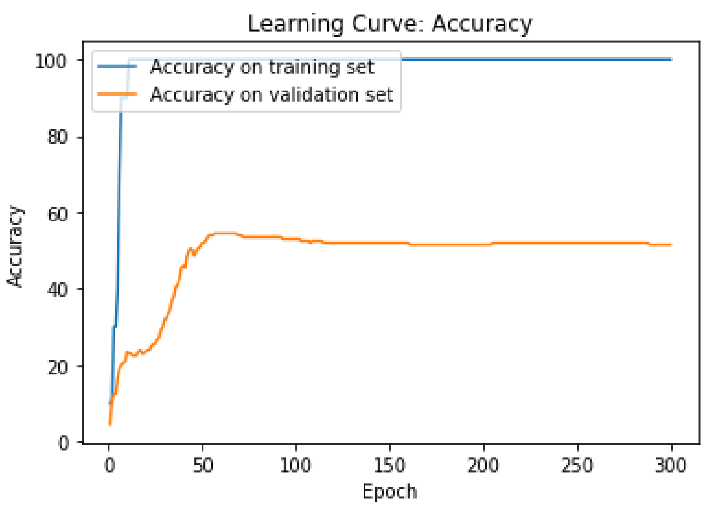
In [ ]:

```
# 75 hidden neurons
hidden_layer_size=75
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)
# Learning rate 0.1
print("Model - 75 hidden neurons - learning rate = 0.1 - small_validation_data - vsmall_tra
ining_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 0.1)
print("Prediction : ", mnist_model.predict(test_data))
# Learning rate 1.
print("\nModel - 75 hidden neurons - learning rate = 1 - small_validation_data - vsmall_tra
ining_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 1.0)
print("Prediction : ", mnist_model.predict(test_data))
# Learning rate 10.
print("\nModel - 75 hidden neurons - learning rate = 10 - small_validation_data - vsmall_tr
aining_data")
mnist_model.weights_init()
mnist_model.train(vsmall_training_data, small_validation_data, 300, 10.0)
print("Prediction : ", mnist_model.predict(test_data))
```

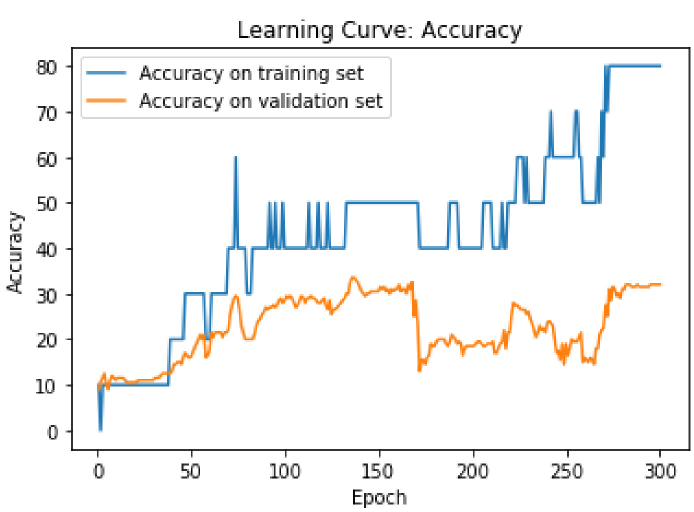
```
Model - 75 hidden neurons - learning rate = 0.1 - small_validation_data - vsmall_training_da
ta
Training time: 1.699739933013916
```




```
Prediction :  3995

Model - 75 hidden neurons - learning rate = 1 - small_validation_data - vsmall_training_data
Training time: 1.775327444076538
```




```
Prediction :  5183

Model - 75 hidden neurons - learning rate = 10 - small_validation_data - vsmall_training_dat
a
Training time: 1.5028529167175293
```




```
Prediction :  3508
```

Comment

We can notice that there is a huge overfitting in all the plots : the performance is definitively good on the small training set (better than with a larger training set) but it is poor on the validation set (the validation set being the same since the beginning of this lab section). This overfitting is due to the fact that these neural networks are trained with not enough data so the model we obtained does not generalize well on test/validation data.

Apart from this, the same trends can be observe regarding the influence of the number of hidden neurons and the learning rate.

Question 2.1.5: Explain the results you obtained at Questions 2.1.2, 2.1.3 and 2.1.4.

Finally we can draw three conclusion from the previous comments:

It is noticeable that the accuracy increases when the number of hidden neurons goes up from 15 to 30, but the accuracy does not progress between 30 and 75 neurons. This is explained because of the underfitting phenomenon when the number of neurons is under dimensioned for the problem. Yet at a point, adding neurons does not improve the performance and will increase the chances to overfit.

Regarding the learning rate and as we noticed before, if it is too small the training will take too much iteration to reach an efficient model. If the learning rate is too high, the training is not efficient as the model changes too quickly. In our case, a good value for the learning rate is 1.

Then when comparing training with a large sample and a small sample, we notice that there is always a strong overfitting with the smaller sample. This result was expected as with small training set, the model may overfit the training data and therefore not generalize well on the validation data.

Question 2.1.6: Among all the numbers of hidden neurons and learning rates you tried in previous questions, which ones would you expect to achieve best performances on the whole dataset? Justify your answer.

The learning rate of 1 gives with no doubt the best result according to all tests. With this learning rate, neural networks with with 30 and 75 hidden neurons give almost a same result but 75 hidden neurons gives the best prediction (8458 against 8307). We will train both models to see if one really is better than the other.

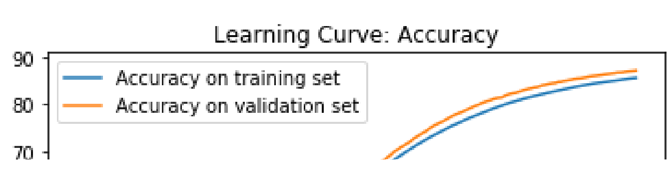
There is a risk of overfitting with 75 hidden neurons but as we will train on more data it should help to solve the problem.

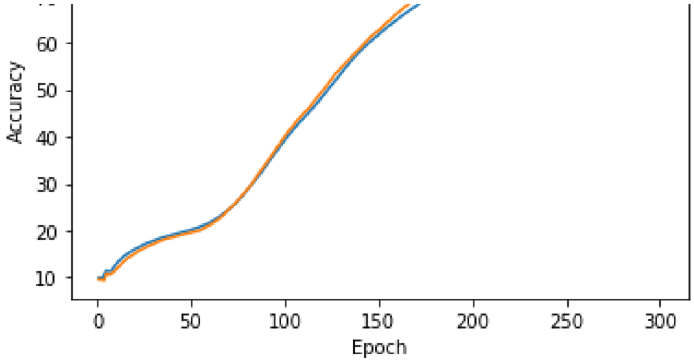
Question 2.1.7: Train a model with the number of hidden neurons and the learning rate you chose in Question 2.1.6 and print its accuracy on the test set. You will use validation_data for validation. Training can be long on the whole dataset (~40 minutes): we suggest that you work on the optional part while waiting for the training to finish.

In [ ]:

```
hidden_layer_size=30
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)
print("Model - 30 hidden neurons - learning rate = 1 - validation_data - training_data")
mnist_model.weights_init()
mnist_model.train(training_data, validation_data, 300, 1)
print("Prediction : ", mnist_model.predict(test_data))
```

```
Model - 30 hidden neurons - learning rate = 1 - validation_data - training_data
Training time: 1360.3531391620636
```



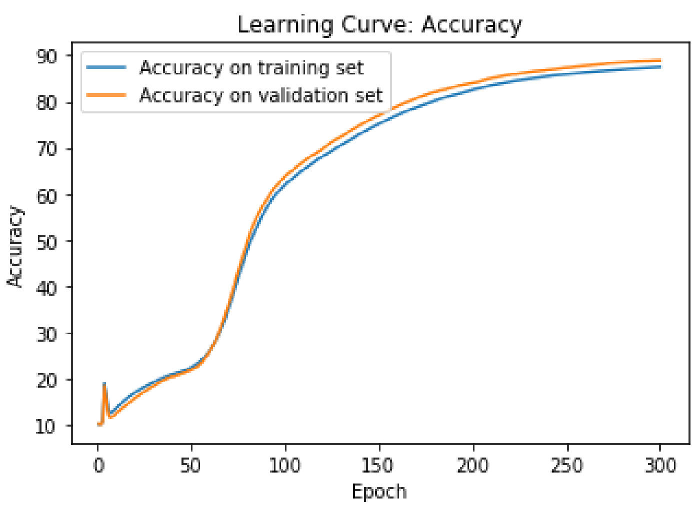



```
Prediction :  8665
```

In [ ]:

```python
hidden_layer_size=75
mnist_model = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)
print("Model - 75 hidden neurons - learning rate = 1 - validation_data - training_data")
mnist_model.weights_init()
mnist_model.train(training_data, validation_data, 300, 1)
print("Prediction : ", mnist_model.predict(test_data))
```

```
Model - 75 hidden neurons - learning rate = 1 - validation_data - training_data
Training time: 1361.6162462234497
```




```
Prediction :  8833
```

Comment

First we can notice that training these neural networks on a larger data set only slightly improves the accuracy. We can also note that the prediction on test data is still slightly better for the neural network with 75 hidden neurons than for the neural network with 30 hidden neurons. At the end of the day, both neural networks could be kept as good ones but we could decide to use the one with 30 hidden neurons if we wanted to keep the simplest neural network (even if here the training time is quite the same).

Part 2 (optional): Another loss function

In classification problems, we usually replace the sigmoids in the output layer by a "softmax" function and the MSE loss by a "cross-entropy" loss. More formally, let $u = (u_1, \ldots, u_n)$ be the vector representing the activation of the output layer of a Neural Network. The output of that neural network is $o = (o_1, \ldots, o_n) = \text{softmax}(u)$, and

$$\text{softmax}(u) = \left(\frac{e^{u_1}}{\sum_{k=1}^{n} e^{u_k}}, \ldots, \frac{e^{u_n}}{\sum_{k=1}^{n} e^{u_k}}\right)$$

If $t = (t_1, \ldots, t_n)$ is a vector of non-negative targets such that $\sum_{k=1}^{n} t_k = 1$ (which is the case in classification problems, where one target is equal to 1 and all others are equal to 0), then the cross-entropy loss is defined as follows:

$$L_{xe}(o, t) = -\sum_{k=1}^{n} t_k \log(o_k)$$

Question 2.2.1: Let L_{xe} be the cross-entropy loss function and $u_i, i \in \{1, \ldots, n\}$, be the activations of the output neurons. Let us assume that the transfer function of the output neurons is the softmax function. Targets are $t_1, \ldots, t_n$. Derive a formula for $\frac{\partial L_{xe}}{\partial u_i}$ (details of your calculations are not required).

Answer: $\frac{\partial L_{xe}}{\partial u_i} = \frac{e^{u_i}}{\sum_{k=1}^{n} e^{u_k}} - t_i$

Question 2.2.2: Implement a new feedforward() function and a new backpropagate() function adapted to the cross-entropy loss instead of the MSE loss.

In [ ]:

```
class NeuralNetwork(NeuralNetwork):
    def feedforward_xe(self, inputs):
        return self.feedforward(inputs)

    def backpropagate_xe(self, targets, learning_rate=5.0):
        transfer_df = self.transfer_df
        l = learning_rate
        targets = np.array(targets) # Target outputs

        #### Change with regard to backpropagate
        self.dL_du_output = dsoftmax(self.u_output, targets)*transfer_df(self.u_output) # Compute partial derivative of loss with respect to activations of output layer
        ### End of change with regard to backpropagate

        self.dL_du_hidden = np.dot(self.dL_du_output, self.W_hidden_to_output.T[:,:-1])*transfer_df(self.u_hidden) #Compute partial derivative of loss with respect to activations of hidden layer
        # Compute partial derivative of loss with respect to weights
        dW_input_to_hidden = np.dot(self.input.T,self.dL_du_hidden) #self.input*self.dL_du_hidden
        dW_hidden_to_output = np.dot(self.o_hidden.T, self.dL_du_output) #self.o_hidden*self.dL_du_output
        # Make updates
        self.W_hidden_to_output = self.W_hidden_to_output - (l * dW_hidden_to_output)/len(targets)
        self.W_input_to_hidden = self.W_input_to_hidden - l*dW_input_to_hidden/len(targets)
```

Question 2.2.3: Create a new Neural Network with the same architecture as in Question 2.1.1 and train it using the softmax cross-entropy loss.

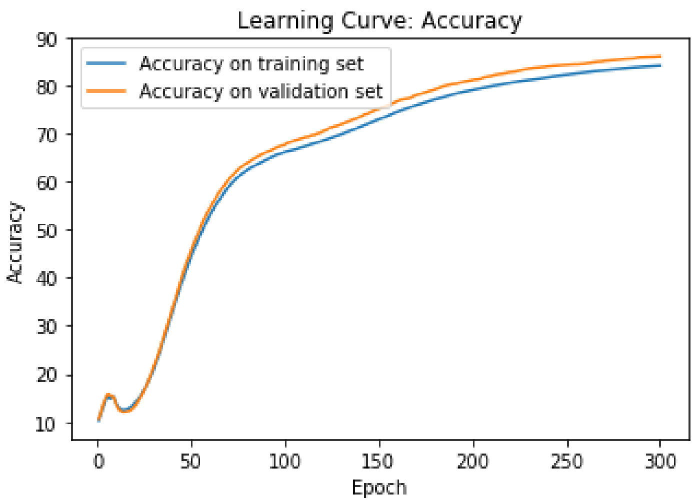
In [ ]:

```
def dsoftmax(x,t):
    return softmax(x) - t
```

In [ ]:

```
input_layer_size=784
hidden_layer_size=30
output_layer_size=10

# Define your neural network
mnist_model_xe = NeuralNetwork(input_layer_size, hidden_layer_size, output_layer_size)

# Train NN and print accuracy on validation data
print("Model xe - 30 hidden neurons - learning rate = 1 - validation_data - training_data")

mnist_model_xe.weights_init()
mnist_model_xe.train_xe(training_data, validation_data, 300, 1.0)
print("Prediction : ", mnist_model_xe.predict(test_data))
```

```
Model xe - 30 hidden neurons - learning rate = 1 - validation_data - training_data
Training time: 1325.011126756668
```

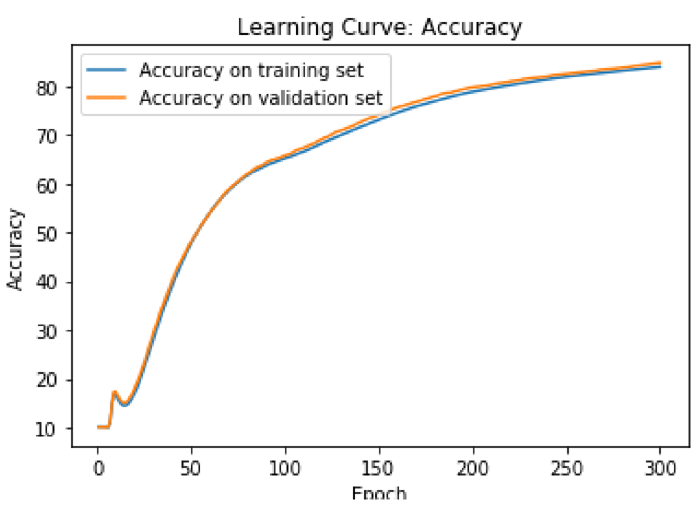



```
Prediction :  8517
```

In [ ]:

```
# Print accuracy on test data
mnist_model_xe.weights_init()
mnist_model_xe.train_xe(training_data, test_data, 300, 1.0)
print("Prediction : ", mnist_model_xe.predict(test_data))
```

```
Training time: 1335.1579492092133
```



```
Prediction :  8473
```

Question 2.2.4: Compare your results with the MSE loss and with the cross-entropy loss.

We can see that this new neural network using cross-entropy loss gives a slightly worse prediction (8517) than the model with MSE loss (8665) when both use 30 hidden neurons and a learning rate of 1. At the end of the day, both models reached a similar accuracy after 300 iterations. Yet we can see that the cross-entropy loss model converges faster and can thus be considered as an improved version in this way. For example if we take 50 iterations for the MSE loss we only have an accuracy of 20 whereas the accuracy is around 45 for the cross-entropy loss model.

THE END!