Deep Learning

April 25, 2023

1 Deep Learning methods and Neural Networks

Universal approximation theorem: A continuous function can be approximated to an arbitrary accuracy if one has at least one hidden layer with finite number of neurons in neural network. The non-linear/activation function can be sigmoid (fermi) [Cybenko in 1989], or just general nonpolynomial bounded activation function [Leshno in 1993 and Pinkus in 1999].

The multilayer architecture of NN gives neural networks the potential of being universal approximators.

Given a function $y = F(x)$ with $x \in [0,1]^d$ and $f(z)$ is a non-linear bounded activation function and $\epsilon > 0$ is chosen accuracy, there is a one layer NN with $w \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^n$ and $x \in \mathbb{R}^m$ and $z_j = \sum w_{ij}x_i + b_j$ so that $|\sum_i w_{ij}^{(2)} f(z_i) + b_j - F(x_j)| < \epsilon$.

Conceptually, it is helpful to divide neural networks into four categories: 1. general purpose neural networks for supervised learning,

2. neural networks designed specifically for image processing, the most prominent example of this class being Convolutional Neural Networks (CNNs),
3. neural networks for sequential data such as Recurrent Neural Networks (RNNs), and
4. neural networks for unsupervised learning such as Deep Boltzmann Machines.

In natural science, DNNs and CNNs have already found numerous applications. In statistical physics, they have been applied to detect phase transitions in 2D Ising and Potts models, lattice gauge theories, and different phases of polymers, or solving the Navier-Stokes equation in weather forecasting. Deep learning has also found interesting applications in quantum physics. Various quantum phase transitions can be detected and studied using DNNs and CNNs, topological phases, and even non-equilibrium many-body localization. Representing quantum states as DNNs quantum state tomography are among some of the impressive achievements to reveal the potential of DNNs to facilitate the study of quantum systems.

Figure: Sketch of the neural network, the input layer is on the left ($x_i = a_i^{(0)}$) and the output layer ($a_i^{(L)}$). The latter is compared through the cost function with the target t_i. The layers between 0 and L are called hidden layers, which increase flexibility of the network. $f(z)$ is nonlinear activation function.

An artificial neural network (ANN), is a computational model that consists of layers of connected neurons (sometimes called nodes or units).

The equations are sketched in the figure, and in matrix form read:

$$\mathbf{z}^l = (\mathbf{a}^{l-1})\mathbf{w}^l + \mathbf{b}^l \quad (1)$$

$$a_i^l = f(z_i^l) \quad (2)$$

Here l in a_i^l, z_i^l stands for the layer l. Using input parameters a^{l-1} in layer l we get output z^l, which are than passed through a non-linear activation function f to obtain a^l. This in turn allowes one to calculate the next layer. Note that we used many yet to be determined weights w and b, which are determined so that they best fit the known data, i.e., on input x_i give as good approximation to target t_i as possible.

We start with input x_i, which defines the first layer a^0, and we end with output layer a^L, which delivers the output, and is needed to evaluate the cost function:

$$a_i^0 \equiv x_i \quad (3)$$

$$C(\{w,b\}) = \frac{1}{2}\sum_i (a_i^L - t_i)^2 \quad (4)$$

The target t is the known data we train on, which was called y in the linear regression. To compare with linear regression $\tilde{y}$ is the output layer a_i^L.

NN is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between layers. A wide variety of different ANNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable w_{ij} and b_i.

Withouth the nonlinear activation function NN would be equivalent to the linear regression (convince yourself). The added nonlinearity through activation function f is thus crucial for the success of NN. Many choices of activation functions are in use. We mention just a few: * sigmoid (fermi) $f(z) = 1/(e^{-z}+1)$ * rectified linear unit (Relu) $f(z) = max(0,z)$ * $\tanh(z)$, which is related to fermi by $\tanh(z/2) = f(-z) - f(z)$ * Exponential linear unit (Elu): $f(z) = \text{if}(z<0)(\alpha(e^{tz}-1)\text{else}(z)$ with $z \ll 1$ * Leaky Relu : $f(z) = \text{if}(z<0)(\alpha z)\text{else}(z)$ with $z \ll 1$

1.0.1 Simple example OR and XOR gate

As we will show, the OR gate can be easily fit with linear regression, however, XOR gate can not be, and requires at least one hidden layer.

Figure: OR and XOR gate with line that can or can not describe it.

The OR gate

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	1

(5)

and XOR gate

$$
\begin{array}{c|c|c}
x_1 & x_2 & t \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0
\end{array} \tag{6}
$$

Let's try linear regression. The design matrix should contain a constant and linear term, i.e., $X^T = [1, x_1, x_2]$, which is

$$
X = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{7}
$$

and linear regression gives $\tilde{y} = X\beta = X(X^T X)^{-1} X^T y$.

It is easy to check that for $y_{OR}^T = [0, 1, 1, 1]$ we get $X(X^T X)^{-1} X^T = [1/4, 3/4, 3/4, 5/4]$ while for $y_{XOR}^T = [0, 1, 1, 0]$ we get $X(X^T X)^{-1} X^T = [1/2, 1/2, 1/2, 1/2]$. If we assume that $\tilde{y}_i < 1/2$ means 0 and $\tilde{y}_i > 1/2$ is 1, we reproduce OR get, but clearly fail at XOR.

As we will show below, one hidden layer can easily give XOR gate. A small technicality first: In the linear regression we wanted to have a constant allowed in the fit, hence our X^T started with unity (to allow β_0 as constat). In ML we always add constant explicitely as an additional degree of freedom (see equations above), hence X^T doe not need to have unity, and it will just be $X^T = [x_1, x_2]$. More precisely

$$
X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{8}
$$

For the activation function $f(z)$ we will choose **Relu**: $f(z) = max(z, 0)$.

We will choose two neurons in the hidden layer, hence w_h is $2x2$ matrix and b_h is two component vector, in terms of which $\mathbf{z}^h = \mathbf{X}\mathbf{w}^h + \mathbf{b}^h$, $\mathbf{a}^h = f(\mathbf{z}^h)$, and the output $\mathbf{y} \equiv \mathbf{a}^o = \mathbf{a}^{(h)}\mathbf{w}^o + \mathbf{b}^o$

The minimization would give the following weights

$$
\mathbf{w}^h = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{9}
$$

$$
\mathbf{b}^h = \begin{bmatrix} 0 & -1 \end{bmatrix} \tag{10}
$$

$$
\mathbf{w}^o = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \tag{11}
$$

$$
\mathbf{b}^o = 0 \tag{12}
$$

Which means that

$$\mathbf{z}^h = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \tag{13}$$

$$\mathbf{a}^h = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \tag{14}$$

and finally

$$\mathbf{a}^h \mathbf{w}^o = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{15}$$

which is identical to target t for XOR gate, and concludes our example.

To solve NN problem we usually distinguish between the following steps: 1) The feed forward stage: which randomly initialized weights and biases to produces an output $\mathbf{a}^L$ to be used in the cost function, and compares with the target $\mathbf{t}$. 2) Back propagation stage follows in which one calculates the gradients of weights $\mathbf{w}$ and biases $\mathbf{b}$. Using minimization algorithm we find the local minimum that corresponds to the given randomly chosen weights and biases. 3) We repeat the two steps (1) and (2) until the error of the cost function is acceptable.

1.1 Back propagation and automatic differentiation

It is convenient to differentiate from the end of the NN towards the start, hence we call this back propagation. We start with differentiation the cost function

$$C(\{w, b\}) = \frac{1}{2} \sum_i (a_i^L - t_i)^2,$$

which gives

$$\frac{\partial C}{\partial w_{jk}^L} = \sum_i (a_i^L - t_i) \frac{\partial a_i^L}{w_{jk}^L} \frac{\partial C}{\partial b_j^L} = \sum_i (a_i^L - t_i) \frac{\partial a_i^L}{b_j^L} \tag{16}$$

because $a_i^L = f(z_i^L)$, we have

$$\frac{\partial C}{\partial w_{kj}^L} = \sum_i (a_i^L - t_i) f'(z_i^L) \frac{\partial z_i^L}{w_{kj}^L} \frac{\partial C}{\partial b_j^L} = \sum_i (a_i^L - t_i) f'(z_i^L) \frac{\partial z_i^L}{b_j^L} \tag{17}$$

finally $z_i^L = \sum_j a_j^{L-1} w_{ji} + b_i$, hence

$$\frac{\partial z_i^L}{w_{kj}^L} = a_k^{L-1} \delta_{ij} \tag{18}$$

$$\frac{\partial z_i^L}{b_j^L} = \delta_{ij} \tag{19}$$

which finally gives

$$\frac{\partial C}{\partial w_{kj}^L} = (a_j^L - t_j) f'(z_j^L) a_k^{L-1} \tag{20}$$

$$\frac{\partial C}{\partial b_j^L} = (a_j^L - t_j) f'(z_j^L) \tag{21}$$

Next we define the quatity

$$\delta_j^L \equiv (a_j^L - t_j) f'(z_j^L)$$

in terms of which we can express

$$\frac{\partial C}{\partial w_{kj}^L} = \delta_j^L a_k^{L-1} \tag{22}$$

$$\frac{\partial C}{\partial b_j^L} = \delta_j^L \tag{23}$$

Note that δ_j^L can also be viewed as

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial z_j^L}$$

We then proceed to previous layer, and obtain

$$\frac{\partial C}{\partial w_{kj}^{L-1}} = \sum_{i,n} \frac{\partial C}{\partial a_n^L} \frac{\partial a_n^L}{\partial z_n^L} \frac{\partial z_n^L}{\partial a_i^{L-1}} \frac{\partial a_i^{L-1}}{\partial z_i^{L-1}} \frac{\partial z_i^{L-1}}{\partial w_{kj}^{L-1}} \tag{24}$$

We then note that

$$\frac{\partial C}{\partial a_n^L} \frac{\partial a_n^L}{\partial z_n^L} = \delta_n^L$$

and because $z_n^L = \sum_i a_i^{L-1} w_{in}^L + b_n^L$ we have

$$\frac{\partial z_n^L}{\partial a_i^{L-1}} = w_{in}^L$$

furthermore

$$\frac{\partial a_i^{L-1}}{\partial z_i^{L-1}} = f'(z_i^{L-1})$$

and further $z_i^{L-1} = \sum_k a_k^{L-2} w_{ki}^{L-1} + b_i^{L-1}$ so that

$$\frac{\partial z_i^{L-1}}{\partial w_{kj}^{L-1}} = a_K^{L-2} \delta_{ij}$$

so that collecting all of that leads to

$$\frac{\partial C}{\partial w_{kj}^{L-1}} = \sum_{i,n} \delta_n^L w_{in}^L f'(z_i^{L-1}) \delta_{ij} a_k^{L-2} = \sum_n \delta_n^L w_{jn}^L f'(z_j^{L-1}) a_k^{L-2}$$

Now we will require that

$$\frac{\partial C}{\partial w^l_{kj}} = \delta^l_j a^{l-1}_k \tag{25}$$

which gives us the following expression

$$\delta^{L-1}_j = \sum_n \delta^L_n w^L_{jn} f'(z^{L-1}_j) \tag{26}$$

We can verify that this equation connects every layer with the previous layer, i.e., this equation is valid for every l, not just $L-1$. Similarly we can show that the derivative with respect to b has the same form, namely,

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{27}$$

In conclusion, we just showed that the automatic differentiation in back propagation leads to the following set of equations

$$\frac{\partial C}{\partial w^l_{kj}} = \delta^l_j a^{l-1}_k \tag{28}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{29}$$

in which δ^l_j can be all obtained by the following recursion relation

$$\delta^l_j = \sum_n \delta^{l+1}_n w^{l+1}_{jn} f'(z^l_j) \tag{30}$$

and the starting condition

$$\delta^L_j = (a^L_j - t_j) f'(z^L_j) \tag{31}$$

1.1.1 Final algorithm

1) Initialize all variables to be minimized $\{w, b\}$ and perform the forward pass to compute all a^l parameters.
2) With current values of $\{w, b\}$ and a^l we compute all gradients $\frac{\partial C}{\partial w^l_{kj}}$ and $\frac{\partial C}{\partial b^l_j}$ and using one of the available minimization routines we take a step towards more optimal variables $\{w, b\}$. Usually one uses some type of gradient descent method, as discussed previously

   $$w^{(j+1)} = w^{(j)} - \gamma_j \frac{\partial C}{\partial w^{(j)}}$$

   here j stands for the iteration.
3) We repeat (1) and (2) until we find local minima
4) We change hiperparameter or change initial condistions to try finding different local minima.

1.1.2 Example code from MNIST dataset on handwritten numbers

We will develop NN code to recognize the handwritten digits. The data is stored in MNIST dataset, which is included in sklearn.

[1]:
```
from numpy import *
import matplotlib.pyplot as plt
from sklearn import datasets

# ensure the same random numbers appear every time
random.seed(0)

# display images in notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)


# download MNIST dataset
digits = datasets.load_digits()

# define inputs and labels
inputs = digits.images # x_i
labels = digits.target # t_i

print('inputs = (n_inputs, pixel_width, pixel_height) =',inputs.shape)
print('labels = (n_inputs) =',labels.shape)
```

```
inputs = (n_inputs, pixel_width, pixel_height) = (1797, 8, 8)
labels = (n_inputs) = (1797,)
```

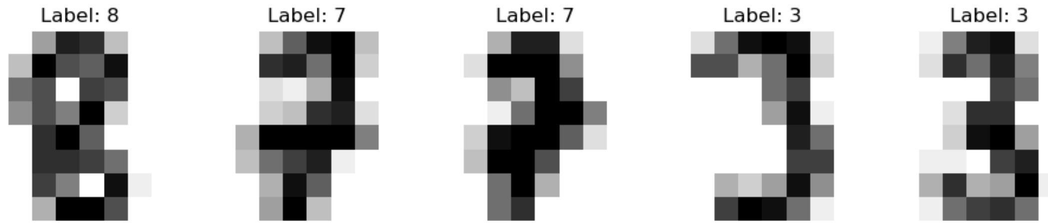
Here we reshape images so that we have **Design matrix** composed of 64 pixels. We also print a few examples of numbers.

[2]:
```
# flatten the image
# the value -1 means dimension is inferred from the remaining dimensions: 8x8 =
↪64
n_inputs,nx,ny = inputs.shape
inputs = inputs.reshape(n_inputs, nx*ny)
print('X = (n_inputs, n_features) =', inputs.shape)

# choose some random images to display
random_indices = random.choice(range(n_inputs), size=5)

for i,image in enumerate(digits.images[random_indices]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title("Label: %d" % digits.target[random_indices[i]])
plt.show()
```

```
X = (n_inputs, n_features) = (1797, 64)
```



First we split the data in 80% training and 20% testing data. Wehich data is training and testing should be choosen at random.

[3]:
```
from sklearn.model_selection import train_test_split

# one-liner from scikit-learn library
train_size = 0.8
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels,␣
 ↪train_size=train_size,test_size=1-train_size)

# equivalently in numpy
def train_test_split_numpy(inputs, labels, train_size):
    n_inputs = len(inputs)
    inputs_shuffled = inputs.copy()
    labels_shuffled = labels.copy()
    random.shuffle(inputs_shuffled)
    random.shuffle(labels_shuffled)

    train_end = int(n_inputs*train_size)
    X_train, X_test = inputs_shuffled[:train_end], inputs_shuffled[train_end:]
    Y_train, Y_test = labels_shuffled[:train_end], labels_shuffled[train_end:]

    return X_train, X_test, Y_train, Y_test
#X_train, X_test, Y_train, Y_test = train_test_split_numpy(inputs, labels,␣
 ↪train_size, test_size)

print("Number of training images: " + str(len(X_train)))
print("Number of test images: " + str(len(X_test)))
```

```
Number of training images: 1437
Number of test images: 360
```

The input and output data have dimensions

$$X \in [n \times 64] \tag{32}$$
$$t \in [n]. \tag{33}$$

It is easier to change the output vector to so-called hot representation, in which $y = 0$ translates into $y = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ and $y = 2$ into $y = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$, etc.

In this way we can use equations for binary choice of 10 chategories. The output vector `Y_onehot` is going to be of dimension $n \times 10$, rather than n.

The function `to_categorical_numpy` implements the hot representation.

```
[4]: # to categorical turns our integer vector into a onehot representation
     def to_categorical_numpy(integer_vector): # integer_vector[n_inputs] contains␣
      ↪number between 0...9
         n_inputs = len(integer_vector)           # inputs
         n_categories = max(integer_vector) + 1  # 10 chategories
         onehot_vector = zeros((n_inputs, n_categories),dtype=int)
         onehot_vector[range(n_inputs), integer_vector] = 1
         return onehot_vector
```

```
[5]: integer_vector=[3,5,4,8,0]
     to_categorical_numpy(integer_vector)
```

```
[5]: array([[0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1],
            [1, 0, 0, 0, 0, 0, 0, 0, 0]])
```

```
[6]: Y_train_onehot, Y_test_onehot = to_categorical_numpy(Y_train),␣
      ↪to_categorical_numpy(Y_test)
```

Figure: NN for recognizing digits.

As said before, the input and the adjusted hot output data have dimensions

$$X \in [n \times 64] \quad (34)$$
$$Y \in [n \times 10]. \quad (35)$$

We will use 50 neurons in the hidden layer, and we have 10 categories, hence our weights will have dimenions:

$$w^{(1)} \in [64 \times 50] \quad (36)$$
$$b^{(1)} \in [50] \quad (37)$$
$$w^{(2)} \in [50 \times 10] \quad (38)$$
$$b^{(2)} \in 10 \quad (39)$$
$$a^{(2)} \in [n \times 10] \quad (40)$$

The equations for our NN models are:

$$z^{(1)} = Xw^{(1)} + b^{(1)} \in [n \times 50] \quad (41)$$
$$a^{(1)} = f^{(1)}(z^{(1)}) \in [n \times 50] \quad (42)$$
$$z^{(2)} = a^{(1)}w^{(2)} + b^{(2)} \in [n \times 10] \quad (43)$$
$$a^{(2)} = f^{(2)}(z^{(2)}) \in [n \times 10] \quad (44)$$

where

$$f^{(1)}(z) = 1/(\exp(-z)+1)$$

and

$$f^{(2)}(z_c) = \frac{\exp z_c}{\sum_{c'=0}^{9} \exp z_{c'}}$$

Note that the output layer uses the `softmax` activation function, because we have the multiple-choice output. The cost function in this case has to maximize the cross entropy, i.e., the probability that the model gets all the answers correct, which is given by

$$P(D|\{w,b\}) = \prod_{i=1}^{n} \prod_{c=0}^{9} P(y_{ic}=1)^{y_{ic}} (1-P(y_{ic}=1)^{1-y_{ic}} \tag{45}$$

here y_{ic} can only take values of 0 or 1, and c runs from 0 to 9, and i runs over all input data n. Here D is the collection of all input data. This is facilitated by the hot vector representation implemented above, in which $y \in [0, 1, ...9]$ is changed to hot representation with y_{ic}.

To maximize $P(D|\{w,b\})$ we minimize $C(\{w,b\}) = -\log(P(D|\{w,b\}))$. The cost function therefore is

$$C(\{w,b\}) = -\sum_{i,c} y_{ic} \log(P_{ic}) + (1-y_{ic}) \log(1-P_{ic}) \tag{46}$$

Note that $a^{(2)}_{ic} \equiv P_{ic}$ is the result of our NN.

Later we we also regularize the cost function with L_2 metric in the following way:

$$C(\{w,b\}) = -\sum_{i,c} y_{ic} \log(P_{ic}) + (1-y_{ic}) \log(1-P_{ic}) + \frac{\lambda}{2}\sum_{hc}(w^{(2)}_{hc})^2 + \frac{\lambda}{2}\sum_{ph}(w^{(1)}_{ph})^2 \tag{47}$$

First we create a random configuration of weights.

[7]:
```
# building our neural network
n_inputs, n_features = X_train.shape
n_hidden_neurons = 50
n_categories = 10

# we make the weights normally distributed using numpy.random.randn
def GiveStartingRandomWeights():
    random.seed(0)
    # weights and bias in the hidden layer
    W_1 = random.randn(n_features, n_hidden_neurons)
    b_1 = zeros(n_hidden_neurons) + 0.01

    # weights and bias in the output layer
    W_2 = random.randn(n_hidden_neurons, n_categories)
    b_2 = zeros(n_categories) + 0.01
    return (W_1, b_1, W_2, b_2)
```

Next we evaluate NN by forward algorithm, and check the accuracy of its predictions.

```
[8]: def mfermi(x):
         return 1/(1 + exp(-x))

     def feed_forward(X, all_weights):
         "identical to feed_forward, except we also return a_1, i.e, hidden layer a"
         W_1, b_1, W_2, b_2 = all_weights
         # weighted sum of inputs to the hidden layer
         z_1 = matmul(X, W_1) + b_1
         # activation in the hidden layer
         a_1 = mfermi(z_1)
         # weighted sum of inputs to the output layer
         z_2 = matmul(a_1, W_2) + b_2
         # softmax output
         # axis 0 holds each input and axis 1 the probabilities of each category
         exp_term = exp(z_2)
         probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)
         # for backpropagation need activations in hidden and output layers
         return a_1, probabilities


     # we obtain a prediction by taking the class with the highest likelihood
     def predict(X, all_weights):
         a_1, probabilities = feed_forward(X, all_weights)
         return (probabilities,argmax(probabilities, axis=1))
```

Checking prediction of NN for one data point. The weights are not yet optimized.

```
[9]: all_weights = GiveStartingRandomWeights()
     (probabilities,predictions) = predict(X_train, all_weights)

     print("probabilities = (n_inputs, n_categories) = " + str(probabilities.shape))
     print("probability that image 0 is in category 0,1,2,...,9 = \n" +␣
      ↪str(probabilities[0]))
     print("probabilities sum up to: " + str(probabilities[0].sum()))
     print()

     print("predictions = (n_inputs) = " + str(predictions.shape))
     print("prediction for image 0: " + str(predictions[0]))
     print("correct label for image 0: " + str(Y_train[0]))
```

```
probabilities = (n_inputs, n_categories) = (1437, 10)
probability that image 0 is in category 0,1,2,…,9 =
[2.23785373e-07 1.47533958e-01 7.28910767e-04 3.32202888e-05
 4.42269923e-05 1.06343900e-04 7.66939998e-03 8.14604377e-01
 4.64970935e-07 2.92788746e-02]
probabilities sum up to: 1.0

predictions = (n_inputs) = (1437,)
```

```
prediction for image 0: 7
correct label for image 0: 6
```

We include `accuracy_score` from sklearn to meassure how large percentage of data is correctly predicted.

[10]:
```
from sklearn.metrics import accuracy_score

(probabilities,predictions) = predict(X_train, all_weights)
print("Old accuracy on training data:", accuracy_score(predictions, Y_train))
```

```
Old accuracy on training data: 0.04314544189283229
```

Next we implement gradients, which are used for back-propagation in function `backpropagation`.

The gradients are somewhat different than derived above because the cost function is obtained from the cross entropy function. Lets firts use cost function C withouth regularization λ.

The gradients are:

$$\frac{\partial C}{\partial w_{jc}^{(2)}} = -\sum_i \left(\frac{y_{ic}}{P_{ic}} - \frac{1-y_{ic}}{1-P_{ic}} \right) \frac{\partial P_{ic}}{\partial w_{jc}^{(2)}} = -\sum_i \frac{y_{ic}-P_{ic}}{P_{ic}(1-P_{ic})} \frac{\partial P_{ic}}{\partial w_{jc}^{(2)}} \tag{48}$$

Next

$$\frac{\partial P_{ic}}{\partial w_{jc}^{(2)}} = \frac{\partial P_{ic}}{\partial z_{ic}} \frac{\partial z_{ic}}{\partial w_{jc}}$$

Since $P_{ic} = f^{(2)}(z_{ic}^{(2)})$ and $z_{ic}^{(2)} = \sum_{j \in hidden} a_{ij}^{(1)} w_{jc}^{(2)} + b_c^{(2)}$ we have

$$\frac{\partial P_{ic}}{\partial w_{jc}^{(2)}} = P_{ic}(1-P_{ic}) a_{ij}^{(1)}$$

which finally gives

$$\frac{\partial C}{\partial w_{jc}^{(2)}} = \sum_i (P_{ic} - y_{ic}) a_{ij}^{(1)} = a^{(1)^T} (a^{(2)} - Y) \tag{49}$$

where we took into account that $a_{ic}^{(2)} = P_{ic}$ and $Y_{ic} = y_{ic}$. Similarly we can see that

$$\frac{\partial C}{\partial b_c^{(2)}} = \sum_i (P_{ic} - y_{ic}) \tag{50}$$

Next we evaluate the derivative in the hidden layer, i.e.,

$$\frac{\partial C}{\partial w_{ph}^{(1)}} = \sum_i \frac{\partial C}{\partial P_{ic}} \frac{\partial P_{ic}}{\partial z_{ic}^{(2)}} \frac{\partial z_{ic}^{(2)}}{\partial a_{ih}^{(1)}} \frac{\partial a_{ih}^{(1)}}{\partial z_{ih}^{(1)}} \frac{\partial z_{ih}^{(1)}}{\partial w_{ph}^{(1)}} \tag{51}$$

which comes from the fact that $P_{ic} = f^{(2)}(z_{ic}^{(2)})$, $z_{ic}^{(2)} = \sum_h a_{ih}^{(1)} w_{hc}^{(2)} + b_c^{(2)}$ and $a_{ih}^{(1)} = f^{(1)}(z_{ih}^{(1)})$ and $z_{ih}^{(1)} = \sum_p X_{ip} w_{ph}^{(1)} + b_h$. We see that $\frac{\partial C}{\partial P_{ic}} = (P_{ic} - y_{ic})/(P_{ic}(1-P_{ic}))$, further $\frac{\partial P_{ic}}{\partial z_{ic}^{(2)}} = P_{ic}(1-P_{ic})$, $\frac{\partial z_{ic}^{(2)}}{\partial a_{ih}^{(1)}} = w_{hc}^{(2)}$, $\frac{\partial a_{ih}^{(1)}}{\partial z_{ih}^{(1)}} = a_{ih}^{(1)}(1-a_{ih}^{(1)})$, $\frac{\partial z_{ih}^{(1)}}{\partial w_{ph}^{(1)}} = X_{ip}$.

Taking all this into account, we get

$$\frac{\partial C}{\partial w^{(1)}_{ph}} = \sum_i X_{ip} a^{(1)}_{ih} (1 - a^{(1)}_{ih}) \sum_c (P_{ic} - y_{ic}) w^{(2)}_{hc} \tag{52}$$

which can also be written as

$$\frac{\partial C}{\partial w^{(1)}_{ph}} = (X^T (a^{(1)} \circ (1 - a^{(1)}) \circ (a^{(2)} - Y)(w^{(2)})^T))_{ph} \tag{53}$$

where we introduced elementwise product $\circ$ defined by $c_{ih} = a_{ih} b_{ih}$ as $c = a \circ b$. Similarly

$$\frac{\partial C}{\partial b^{(1)}_h} = \sum_{i,h} a^{(1)}_{ih} (1 - a^{(1)}_{ih}) \sum_c (P_{ic} - y_{ic}) w^{(2)}_{hc} \tag{54}$$

Finally, when λ is nonzero, we will just add to derivatives

$$\frac{\partial C}{\partial w^{(1)}_{ph}} += \lambda w^{(1)}_{ph} \tag{55}$$

$$\frac{\partial C}{\partial w^{(2)}_{jc}} += \lambda w^{(2)}_{jc} \tag{56}$$

[11]:
```
def backpropagation(X, Y, all_weights):
    a_1, probabilities = feed_forward(X, all_weights)
    W_1, b_1, W_2, b_2 = all_weights
    # error in the output layer
    error_output = probabilities - Y
    # error in the hidden layer
    error_hidden = matmul(error_output, W_2.T) * a_1 * (1 - a_1)

    # gradients for the output layer
    dW2 = matmul(a_1.T, error_output)
    dB2 = sum(error_output, axis=0)

    # gradient for the hidden layer
    dW1 = matmul(X.T, error_hidden)
    dB1 = sum(error_hidden, axis=0)

    return dW2, dB2, dW1, dB1
```

[13]:
```
dW2, dB2, dW1, dB1 = backpropagation(X_train, Y_train_onehot, all_weights)
print('shapes of gradients=', dW2.shape, dB2.shape, dW1.shape, dB1.shape)
```

```
shapes of gradients= (50, 10) (10,) (64, 50) (50,)
```

First we use simple gradient descendent method with fixed `learning rate` γ=`eta`. We evaluate gradient `num_iterations`-times and move towards local minimum.

[14]:
```
def SimpleGradientMethod(X_train, Y_train, all_weights, eta, lmbd,␣
↪num_iterations):
    (W_1,b_1,W_2,b_2) = all_weights
    for i in range(num_iterations):
        # calculate gradients
        dW_2, dB_2, dW_1, dB_1 = backpropagation(X_train, Y_train,␣
↪[W_1,b_1,W_2,b_2])
        # regularization term gradients
        dW_2 += lmbd * W_2
        dW_1 += lmbd * W_1
        # update weights and biases
        W_1 -= eta * dW_1
        b_1 -= eta * dB_1
        W_2 -= eta * dW_2
        b_2 -= eta * dB_2
    return (W_1,b_1,W_2,b_2)
```

We also add regularization to cost function in the form of $\lambda||w||_2^2$. The precision after 100 steps is barely improved.

[15]:
```
eta = 0.01
lmbd = 0.01
num_iterations=100

all_weights = GiveStartingRandomWeights()
all_weights = SimpleGradientMethod(X_train, Y_train_onehot, all_weights, eta,␣
↪lmbd, num_iterations)

error=accuracy_score(predict(X_train,all_weights)[1],Y_train)
print('Accuracy on training data: ', error)
```

```
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
```

```
Accuracy on training data:  0.10438413361169102
```

Next we implement **stochastic gradient descent (SGD)**, which takes a random subset of data (of size `batch_size`), and we compute gradient only for the subset of points. We then move in the steepest descent direction for only this subset. The randomness introduced this way decreases the chance that our opmization scheme gets stuck in a local minima.

If the size of the minibatches are small relative to the number of datapoints ($M < n$), the computation of the gradient is much cheaper since we sum over the datapoints in the $k-th$ minibatch and not all n datapoints.

[16]:
```
def StochasticGradientMethod(X_train, Y_train, all_weights, eta, lmbd,␣
↪batch_size, epochs):
    (W_1,b_1,W_2,b_2) = all_weights
```

```
    data_indices = arange(len(X_train))
    iterations = len(X_train) // batch_size
    print('Number of iterations=', iterations)
    for i in range(epochs):
        for j in range(iterations):
            chosen_datapoints = random.choice(data_indices, size=batch_size,␣
↪replace=False)
            # minibatch training data
            X_batch = X_train[chosen_datapoints]
            Y_batch = Y_train[chosen_datapoints]
            dW_2, dB_2, dW_1, dB_1 = backpropagation(X_batch, Y_batch,␣
↪[W_1,b_1,W_2,b_2])
            # regularization term gradients
            dW_2 += lmbd * W_2
            dW_1 += lmbd * W_1
            # update weights and biases
            W_1 -= eta * dW_1
            b_1 -= eta * dB_1
            W_2 -= eta * dW_2
            b_2 -= eta * dB_2
    return (W_1,b_1,W_2,b_2)
```

Finally we use SG method for learning method γ=`eta`=0.01 and $\lambda = 0.1$ wih `batch_size=100`. The number of iteration over the minibathces (`epochs`) is also choosen at 100. This gives excellent prediction over 98%. A human can typically read with accuracy 98%.

[17]:
```
eta = 0.01
lmbd = 0.1
epochs = 100
batch_size = 100

all_weights = GiveStartingRandomWeights()

all_weights = StochasticGradientMethod(X_train, Y_train_onehot, all_weights,␣
 ↪eta, lmbd, batch_size, epochs)

error=accuracy_score(predict(X_train,all_weights)[1],Y_train)
error2=accuracy_score(predict(X_test,all_weights)[1],Y_test)
print('Accuracy on training data: ', error, error2)
```

```
Number of iterations= 14
Accuracy on training data:  0.9937369519832986 0.9805555555555555
```

1.1.3 Adjust hyperparameters

We now perform a grid search to find the optimal hyperparameters for the network.
Note that we are only using 1 layer with 50 neurons, and human performance is estimated to be

around 98% (2% error rate).

[18]:
```
eta_vals = logspace(-5, 1, 7)
lmbd_vals = logspace(-5, 1, 7)
# store the models for later use
DNN_numpy = zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

# grid search
for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):

        all_weights = GiveStartingRandomWeights()
        all_weights = StochasticGradientMethod(X_train, Y_train_onehot,␣
 ↪all_weights, eta, lmbd, batch_size, epochs)

        error=accuracy_score(predict(X_train,all_weights)[1],Y_train)
        error2=accuracy_score(predict(X_test,all_weights)[1],Y_test)
        DNN_numpy[i][j] = error

        #test_predict = dnn.predict(X_test)

        print('Learning rate=', eta, 'Lambda=', lmbd, 'Accuracy=', error,␣
 ↪error2)
```

```
Number of iterations= 14
Learning rate= 1e-05 Lambda= 1e-05 Accuracy= 0.13569937369519833
0.18055555555555555
Number of iterations= 14
Learning rate= 1e-05 Lambda= 0.0001 Accuracy= 0.13569937369519833
0.18055555555555555
Number of iterations= 14
Learning rate= 1e-05 Lambda= 0.001 Accuracy= 0.13569937369519833
0.18055555555555555
Number of iterations= 14
Learning rate= 1e-05 Lambda= 0.01 Accuracy= 0.13569937369519833
0.18055555555555555
Number of iterations= 14
Learning rate= 1e-05 Lambda= 0.1 Accuracy= 0.13569937369519833
0.18055555555555555
Number of iterations= 14
Learning rate= 1e-05 Lambda= 1.0 Accuracy= 0.13569937369519833
0.18055555555555555
Number of iterations= 14
Learning rate= 1e-05 Lambda= 10.0 Accuracy= 0.13848295059151008
0.18055555555555555
Number of iterations= 14
Learning rate= 0.0001 Lambda= 1e-05 Accuracy= 0.6089074460681977
0.5833333333333334
```

```
Number of iterations= 14
Learning rate= 0.0001 Lambda= 0.0001 Accuracy= 0.6089074460681977
0.5833333333333334
Number of iterations= 14
Learning rate= 0.0001 Lambda= 0.001 Accuracy= 0.6089074460681977
0.5833333333333334
Number of iterations= 14
Learning rate= 0.0001 Lambda= 0.01 Accuracy= 0.6089074460681977
0.5805555555555556
Number of iterations= 14
Learning rate= 0.0001 Lambda= 0.1 Accuracy= 0.6116910229645094
0.5805555555555556
Number of iterations= 14
Learning rate= 0.0001 Lambda= 1.0 Accuracy= 0.6450939457202505
0.6083333333333333
Number of iterations= 14
Learning rate= 0.0001 Lambda= 10.0 Accuracy= 0.8545581071677105
0.8138888888888889
Number of iterations= 14
Learning rate= 0.001 Lambda= 1e-05 Accuracy= 0.9617258176757133
0.8916666666666667
Number of iterations= 14
Learning rate= 0.001 Lambda= 0.0001 Accuracy= 0.9617258176757133
0.8916666666666667
Number of iterations= 14
Learning rate= 0.001 Lambda= 0.001 Accuracy= 0.9617258176757133
0.8916666666666667
Number of iterations= 14
Learning rate= 0.001 Lambda= 0.01 Accuracy= 0.9617258176757133
0.8944444444444445
Number of iterations= 14
Learning rate= 0.001 Lambda= 0.1 Accuracy= 0.9624217118997912 0.9055555555555556
Number of iterations= 14
Learning rate= 0.001 Lambda= 1.0 Accuracy= 0.9826026443980515 0.95
Number of iterations= 14
Learning rate= 0.001 Lambda= 10.0 Accuracy= 0.942936673625609 0.9305555555555556
Number of iterations= 14
Learning rate= 0.01 Lambda= 1e-05 Accuracy= 0.9965205288796103
0.9361111111111111
Number of iterations= 14
Learning rate= 0.01 Lambda= 0.0001 Accuracy= 0.9972164231036882
0.9527777777777777
Number of iterations= 14
Learning rate= 0.01 Lambda= 0.001 Accuracy= 0.9979123173277662
0.9555555555555556
Number of iterations= 14
Learning rate= 0.01 Lambda= 0.01 Accuracy= 0.9979123173277662 0.9472222222222222
Number of iterations= 14
```

```
Learning rate= 0.01 Lambda= 0.1 Accuracy= 0.9937369519832986 0.9805555555555555
Number of iterations= 14
Learning rate= 0.01 Lambda= 1.0 Accuracy= 0.8176757132915797 0.7805555555555556
Number of iterations= 14
Learning rate= 0.01 Lambda= 10.0 Accuracy= 0.20668058455114824
0.18333333333333332
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))

Learning rate= 0.1 Lambda= 1e-05 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))

Learning rate= 0.1 Lambda= 0.0001 Accuracy= 0.10368823938761308
0.08888888888888889
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))

Learning rate= 0.1 Lambda= 0.001 Accuracy= 0.0953375086986778 0.125
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))

Learning rate= 0.1 Lambda= 0.01 Accuracy= 0.10438413361169102
0.07777777777777778
```

```
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))

Learning rate= 0.1 Lambda= 0.1 Accuracy= 0.09394572025052192 0.11666666666666667
Number of iterations= 14
Learning rate= 0.1 Lambda= 1.0 Accuracy= 0.10160055671537926 0.09166666666666666
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))

Learning rate= 0.1 Lambda= 10.0 Accuracy= 0.10160055671537926
0.09166666666666666
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 1.0 Lambda= 1e-05 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 1.0 Lambda= 0.0001 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
```

```
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 1.0 Lambda= 0.001 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 1.0 Lambda= 0.01 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 1.0 Lambda= 0.1 Accuracy= 0.10438413361169102 0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))

Learning rate= 1.0 Lambda= 1.0 Accuracy= 0.10438413361169102 0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
```

```
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 1.0 Lambda= 10.0 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 10.0 Lambda= 1e-05 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 10.0 Lambda= 0.0001 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 10.0 Lambda= 0.001 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
```

```
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 10.0 Lambda= 0.01 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 10.0 Lambda= 0.1 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 10.0 Lambda= 1.0 Accuracy= 0.10438413361169102
0.07777777777777778
Number of iterations= 14

/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:2
: RuntimeWarning: overflow encountered in exp
  return 1/(1 + exp(-x))
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
5: RuntimeWarning: overflow encountered in exp
  exp_term = exp(z_2)
/var/folders/j8/d9m3r0zx7j37l3ktfl_n1xw00000gn/T/ipykernel_20664/1438300027.py:1
6: RuntimeWarning: invalid value encountered in divide
  probabilities = exp_term/sum(exp_term, axis=1, keepdims=True)

Learning rate= 10.0 Lambda= 10.0 Accuracy= 0.10438413361169102
```

```
0.07777777777777778
```

[ ]: