Ben-Gurion University of the Negev
Faculty of Engineering Sciences
Department of Software and Information systems Engineering

Deep Learning
Assignment 1

The purpose of the assignment

Enabling students to experiment with building a simple neural network "from scratch" and to obtain a deep understanding of the forward/backward propagation process.

Submission instructions

a) The assignment due date: 23/11/2022
b) The assignment is to be carried out using PyTorch.
c) Submission in **pairs** only.
d) Plagiarism of any kind (e.g., GitHub) is forbidden.
e) You are allowed to implement auxiliary functions in addition to those defined. These functions need to be documented. Also, they need to be mentioned and explained in the assignment report.
f) The network needs to be implemented using vectorization, as shown in lecture #1.
g) The entire project will be submitted as a single zip file. It is the students' responsibility to make sure that the file is valid (i.e. can be opened correctly).

1. Implement the following functions, which are used to carry out the forward propagation process:
    a. **initialize_parameters(layer_dims)**

        input: an array of the dimensions of each layer in the network (layer 0 is the size of the flattened input, layer L is the output softmax)

        output: a dictionary containing the initialized W and b parameters of each layer (W1…WL, b1…bL).

        Hint: Use the randn and zeros functions of numpy to initialize W and b, respectively

    b. **linear_forward(A, W, b)**

        Description: Implement the linear part of a layer's forward propagation.

        input:
        A – the activations of the previous layer
        W – the weight matrix of the current layer (of shape [size of current layer, size of previous layer])

B – the bias vector of the current layer (of shape [size of current layer, 1])

Output:
Z – the linear component of the activation function (i.e., the value before applying the non-linear function)
linear_cache – a dictionary containing A, W, b (stored for making the backpropagation easier to compute)

c. softmax(Z)

Input:
Z – the linear component of the activation function

Output:
A – the activations of the layer
activation_cache – returns Z, which will be useful for the backpropagation

note:
Softmax can be thought of as a sigmoid for multi-class problems. The formula for softmax for each node in the output layer is as follows:

$$Softmax(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

d. relu(Z)

Input:
Z – the linear component of the activation function

Output:
A – the activations of the layer
activation_cache – returns Z, which will be useful for the backpropagation

e. linear_activation_forward(A_prev, W, B, activation)

Description:
Implement the forward propagation for the LINEAR->ACTIVATION layer

Input:
A_prev – activations of the previous layer
W – the weights matrix of the current layer
B – the bias vector of the current layer
Activation – the activation function to be used (a string, either “softmax” or “relu”)

Output:
A – the activations of the current layer
cache – a joint dictionary containing both linear_cache and activation_cache

f. L_model_forward(X, parameters, use_batchnorm)

Description:
Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SOFTMAX

computation

Input:
X – the data, numpy array of shape (input size, number of examples)
parameters – the initialized W and b parameters of each layer
use_batchnorm - a boolean flag used to determine whether to apply batchnorm after the activation (note that this option needs to be set to "false" in Section 3 and "true" in Section 4).

Output:
AL – the last post-activation value
caches – a list of all the cache objects generated by the linear_forward function

g. compute_cost(AL, Y)

Description:
Implement the cost function defined by equation. The requested cost function is *categorical cross-entropy loss*. The formula is as follows :

$cost = -\frac{1}{m} * \sum_1^m \sum_1^C y_i \log(\hat{y})$, where y_i is one for the true class ("ground-truth") and $\hat{y}$ is the softmax-adjusted prediction (this link provides a good overview).

Input:
AL – probability vector corresponding to your label predictions, shape (num_of_classes, number of examples)
Y – the labels vector (i.e. the ground truth)

Output:
cost – the cross-entropy cost

h. apply_batchnorm(A)

Description:
performs batchnorm on the received activation values of a given layer.

Input:
A - the activation values of a given layer

output:
NA - the normalized activation values, based on the formula learned in class

2. Implement the following functions, which are used to carry out the backward propagation process:
   a. Linear_backward(dZ, cache)

   description:
   Implements the linear part of the backward propagation process for a single layer

   Input:

dZ – the gradient of the cost with respect to the linear output of the current layer (layer l)
cache – tuple of values (A_prev, W, b) coming from the forward propagation in the current layer

Output:
dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
dW -- Gradient of the cost with respect to W (current layer l), same shape as W
db -- Gradient of the cost with respect to b (current layer l), same shape as b

b. linear_activation_backward(dA, cache, activation)

Description:
Implements the backward propagation for the LINEAR->ACTIVATION layer. The function first computes dZ and then applies the linear_backward function.

Some comments:

- The derivative of ReLU is $f'(x) = \begin{cases} 1 & if\ x > 0 \\ 0 & otherwise \end{cases}$
- The derivative of the softmax function is: $p_i - y_i$, where p_i is the softmax-adjusted probability of the class and y_i is the "ground truth" (i.e. 1 for the real class, 0 for all others)
- You should use the activations cache created earlier for the calculation of the activation derivative and the linear cache should be fed to the linear_backward function

Input:
dA – post activation gradient of the current layer
cache – contains both the linear cache and the activations cache

Output:
dA_prev – Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
dW – Gradient of the cost with respect to W (current layer l), same shape as W
db – Gradient of the cost with respect to b (current layer l), same shape as b

c. relu_backward (dA, activation_cache)

Description:
Implements backward propagation for a ReLU unit

Input:
dA – the post-activation gradient
activation_cache – contains Z (stored during the forward propagation)

Output:
dZ – gradient of the cost with respect to Z

d. softmax_backward (dA, activation_cache)

Description:
Implements backward propagation for a softmax unit

Input:
dA – the post-activation gradient
activation_cache – contains Z (stored during the forward propagation)

Output:
dZ – gradient of the cost with respect to Z

e. L_model_backward(AL, Y, caches)

Description:
Implement the backward propagation process for the entire network.

Some comments:
the backpropagation for the softmax function should be done only once as only the output layers uses it and the RELU should be done iteratively over all the remaining layers of the network.

Input:
AL - the probabilities vector, the output of the forward propagation (L_model_forward)
Y - the true labels vector (the "ground truth" - true classifications)
Caches - list of caches containing for each layer: a) the linear cache; b) the activation cache

Output:
Grads - a dictionary with the gradients

```
grads["dA" + str(l)] = ...
grads["dW" + str(l)] = ...
grads["db" + str(l)] = ...
```

f. Update_parameters(parameters, grads, learning_rate)

Description:
Updates parameters using gradient descent

Input:
parameters – a python dictionary containing the DNN architecture's parameters
grads – a python dictionary containing the gradients (generated by L_model_backward)
learning_rate – the learning rate used to update the parameters (the "alpha")

Output:
parameters – the updated values of the parameters object provided as input

3. In this section you will use the functions you created in the previous sections to train the network and produce predictions. For this purpose, implement the following functions:

a. **L_layer_model(X, Y, layers_dims, learning_rate, num_iterations, batch_size)**
   Description:
   Implements a L-layer neural network. All layers but the last should have the ReLU activation function, and the final layer will apply the softmax activation function. The size of the output layer should be equal to the number of labels in the data. Please select a batch size that enables your code to run well (i.e. no memory overflows while still running relatively fast).

   Hint: the function should use the earlier functions in the following order: initialize -> L_model_forward -> compute_cost -> L_model_backward -> update parameters

   Input:
   X – the input data, a numpy array of shape (height*width , number_of_examples)
   Comment: since the input is in grayscale we only have height and width, otherwise it would have been height*width*3
   Y – the "real" labels of the data, a vector of shape (num_of_classes, number of examples)
   Layer_dims – a list containing the dimensions of each layer, including the input
   batch_size – the number of examples in a single training batch.

   Output:
   parameters – the parameters learnt by the system during the training (the same parameters that were updated in the update_parameters function).
   costs – the values of the cost function (calculated by the compute_cost function). One value is to be saved after each 100 training iterations (e.g. 3000 iterations -> 30 values).

b. **Predict(X, Y, parameters)**
   Description:
   The function receives an input data and the true labels and calculates the accuracy of the trained neural network on the data.

   Input:
   X – the input data, a numpy array of shape (height*width, number_of_examples)
   Y – the "real" labels of the data, a vector of shape (num_of_classes, number of examples)
   Parameters – a python dictionary containing the DNN architecture's parameters

   Output:
   accuracy – the accuracy measure of the neural net on the provided data (i.e. the percentage of the samples for which the correct label receives the hughest confidence score). Use the softmax function to normalize the output values.

4. Use the code you wrote to classify the MNIST dataset and present a summary report
   a. You may use publicly available code to download and preprocess the data. Note that there is a predefined division between the train and test set. Use 20% of the training set as a validation set (samples need to be randomly chosen).
   b. Run your network using the following configuration:
      - 4 layers (aside from the input layer), with the following sizes: 20,7,5,10

- Do not activate the batchnorm option at this point
- The input at each iteration needs to be "flattened" to a matrix of [m,784], where m is the number of samples
- Use a learning rate of 0.009
- Train the network until there is no improvement on the validation set (or the improvement is very small) for 100 training steps (this is the stopping criterion). Please include in the report the number of iterations and epochs needed to train your network. Also, specify the batch size.

c. Please include the following details in your report:
- The final accuracy values for the train, validation and test sets.
- The cost value for each 100 training steps. Please make sure that the index of the training step will also be included in the report. Print the values from the L_layer_model

d. All the information requested above will be included in a .docx file uploaded with the code.

5. Repeat section 4 when the batchnorm function is "on". Analyze and compare this experiment to the previous one (performance, running time, number of training steps etc.). There is no need to update the parameters of the batchnorm in the way described in the lecture (that is, use z_{norm}^{i} and not $\tilde{z}^{i}$).

6. **Bonus 10%:** modify the code so that it supports the L2 norm functionality. In addition to the code, please provide a short explanation about the changes done in the code. Compare the values of the weights of your architecture with and without this change.