Deep Learning HW 1 Report

Shay Kricheli

April 2021

Function Documentation

1. **Forward propagation**:

   (a) `def initialize_parameters(layer_dims)`
   Initialisation of all layers' weights and biases
   Input variables:

   - `layers_dim`: list of dimensions for each layer in the network (layer 0 is the size of the flattened input, layer `L` is the output softmax)

   Output variables:

   - `parameters`: A dictionary containing the initialized `W` and `b` parameters of each layer (`W_1`,..., `W_L`, `b_1`,...,`b_L`).

   (b) `def linear_forward(A, W, b)`
   Implements the linear part of a layer's forward propagation
   Input variables:

   - `A`: Output of the previous layer, shape size `[previous_layer, 1]`
   - `W`: Weights, shape size `[current_layer, previous_layer]`
   - `W`: Bias, shape size `[current_layer, 1]`

   Output variables:

   - `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) `def softmax(Z)`
   Apply the softmax function
   Input variables:

   - `Z`: The linear component of the activation function

   Output variables:

   - `A`: The activations of the layer
   - `activation_cache`: Returns Z, which will be useful for the backpropagation

   (d) `def relu(Z)`
   Apply relu function
   Input variables:

   - `Z`: The linear component of the activation function

   Output variables:

   - `A`: The activations of the layer
   - `activation_cache`: Returns Z, which will be useful for the backpropagation

(e) `def linear_activation_forward(A_prev, W, B, activation)`
The forward propagation for the LINEAR->ACTIVATION layer
Input variables:

- `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 variables:

- `A`: The activations of the current layer
- `cache`: A joint dictionary containing both `linear_cache` and `activation_cache`

(f) `def L_model_forward(X, parameters, use_batchnorm)`
Forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SOFTMAX computation
Input variables:

- `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

Output variables:

- `AL`: The last post-activation value
- `cache`: A list of all the cache objects generated by the `linear_forward` function

(g) `def compute_cost(AL, Y)`
Calculate the cost function defined by the categorical cross-entropy cost
Input variables:

- `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 variables:

- `cost`: The cross-entropy cost

(h) `def apply_batchnorm(A)`
Performs batchnorm on the received activation values of a given layer
Input variables:

- `A`: The activation values of a given layer

Output variables:

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

2. **Backward propagation**:

(a) `def linear_backward(dZ, cache)`
The linear part of the backward propagation process for a single layer
Input variables:

- `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 variables:

- `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) `def linear_activation_backward(dA, cache, activation)`
Implements the backward propagation for the LINEAR->ACTIVATION layer. The function first computes `dZ` and then applies the `linear_backward` function
Input variables:

- `dA`: The post activation gradient of the current layer
- `cache`: Contains both the linear cache and the activations cache
- `activation`: A flag indicating the activation function to use

Output variables:

- `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`
  Let's assume we derive Layer (2) and it is the last layer: In general our forward computation is as follows:

```
Z2 = W2 @ A1 + b2
A2 = softmax(Z2)
dA = (dL/dA2) = (A2 - y) # (assuming L = CE) => the input to this function
cache = [W2, A1, b2, Z2]
```

  We need to return:

```
dA_prev = (dA1) = dL/dA1
dW = dL/dW2
db = dL/db
```

  Now let's use our functions. We know that calling the activation backwards with `dL/dA2` returns `dZ2`. To calculate `dW2` and `db2` we need `dZ2`, so we call `linear_backwards` which does the following calculation:

```
dW2 = dZ2 @ A1.T
db2 = dZ2
dA_prev = dL/dA1 = (dL/dA2) * (dA2/dZ2) * (dZ2/dA1) = (dL/dZ2) * (dZ2/dA1)
```

  Each `backward_func` is supposed to get `dL/dA_cur` as an input and chain by multiplication:

```
(dL/dA_cur) @ (dA_cur/dZ_cur)
```

  Assuming `y_hat = softmax(Z_last)`, the last layer has:

```
dA_last = dy_hat = (dL/dy_hat) = - (1/m) * sum([1/y_hat_i for y_hat_i in y_hat])
```

(c) `def relu_backward(dA, activation_cache)`
Implements backward propagation for a ReLU unit
Input variables:

- `dA`: The post activation gradient
- `activation_cache`: Contains Z (stored during the forward propagation)

Output variables:

- `dZ`: Gradient of the cost with respect to Z

(d) `def softmax_backward(dA, activation_cache)`
Implements backward propagation for a softmax unit
Input variables:

- `dA`: The post activation gradient
- `activation_cache`: Contains Z (stored during the forward propagation)

Output variables:

- `dZ`: Gradient of the cost with respect to Z

(e) `def L_model_backward(AL, Y, caches)`
Implement the backward propagation process for the entire network. The backpropagation for the softmax function is 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 variables:

- `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 variables:

- `Grads`: A dictionary with the gradients

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

(f) `def Update_parameters(parameters, grads, learning_rate)`
Updates parameters using gradient descent (IN-PLACE)
Input variables:

- `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 variables:

- `parameters`: The updated values of the parameters object provided as input

3. **Train and predict**:

(a) `def L_layer_model(X, Y, layers_dims, learning_rate, num_iterations, batch_size)`
Implements a L-layer neural network. All layers but the last have the ReLU activation function, and the final layer will apply the softmax activation function. The size of the output layer is equal to the number of labels in the data
Input variables:

- `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]`
- `layers_dims`: A list containing the dimensions of each layer, including the input
- `learning_rate`: The learning rate used to train the parameters (the "alpha")
- `num_iterations`: The learning number of iterations to train
- `batch_size`: The number of examples in a single training batch

Output variables:

- `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) `def Predict(X, Y, parameters)`
The function receives an input data and the true labels and calculates the accuracy of the trained neural network on the data
Input variables:

- `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 `[number_of_examples, 1]`
- `parameters`: A python dictionary containing the DNN architecture's parameters

Output variables:

- `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 highest confidence score). Uses the softmax function to normalize the output values

(c) `def split_train(x_train, y_train, p)`
Auxiliary function to split the train into train and validation datasets
Input variables:

- `x_train`: Train data, size `[height*width , number_of_examples]`
- `y_train`: The "real" labels of the data, a vector of shape `[number_of_examples, 1]`
- `p`: Percentage of the validation of all the training data

Output variables:

- `train_tuple`: Train data and labels
- `valid_tuple`: Validation data and labels

Simulation Results

To run the simulation, we used the configuration detailed for the network: 4 layers (aside from the input layer which is of size $28^2 = 784$) with sizes $20, 7, 5, 10$ and learning rate of $\alpha = 0.009$. We implemented a stopping criterion as following (variations of the stopping criterion were allowed in the course forum to achieve better accuracy results): Every 100 batch steps, we check for the highest and lowest accuracy values on the validation set. If their difference is equal or less than a predefined small value $\varepsilon = 10^{-2}$, then the training is stopped. We tested out several values for the the batch size and number of iterations. Choosing a batch size of 64 examples and 10 epochs per training iteration, we got the results displayed in the next page. The results show a much faster and steadier convergence rate, with lower variations in final loss value - all when applying batch normalization. The results demonstrate how it is clearly preferred to apply this technique.

Without Batch Normalization

- Last training step train accuracy: 87.18125%
- Last validation step train accuracy: 86.91666%
- Last training step test accuracy: 86.92%
- Total training iterations: 36, total epochs: 352
- Total training time: 21:22 minutes

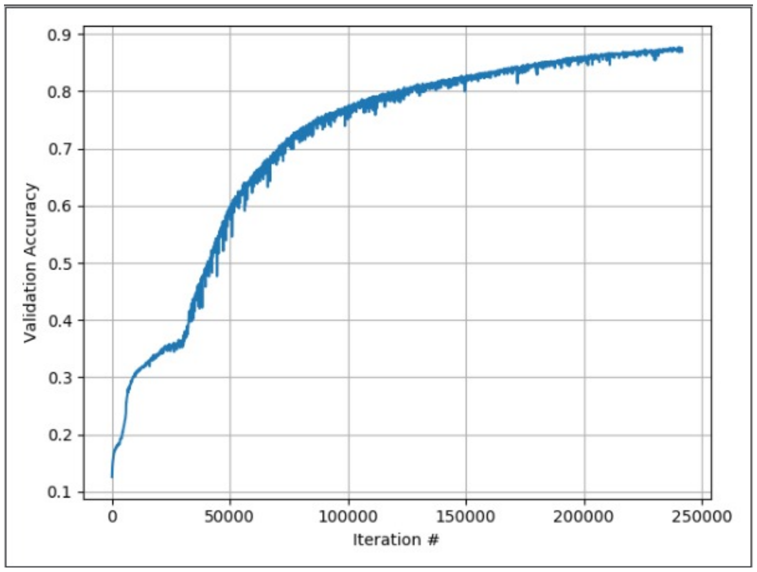

Figure 1: Validation accuracy without batch normalization

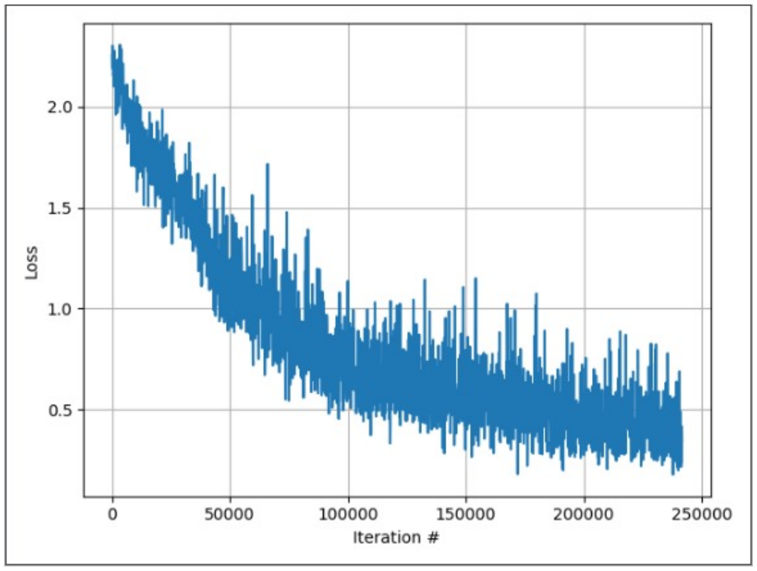

Figure 2: Validation loss without batch normalization

With Batch Normalization

- Last training step train accuracy: 93.04167%
- Last validation step train accuracy: 91.62500%
- Last training step test accuracy: 92.27000%
- Total training iterations: 6, total epochs: 58
- Total training time: 02:21 minutes

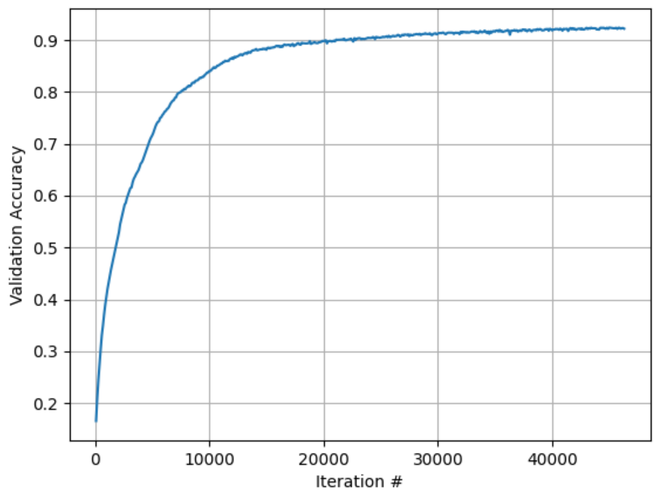

Figure 3: Validation accuracy with batch normalization

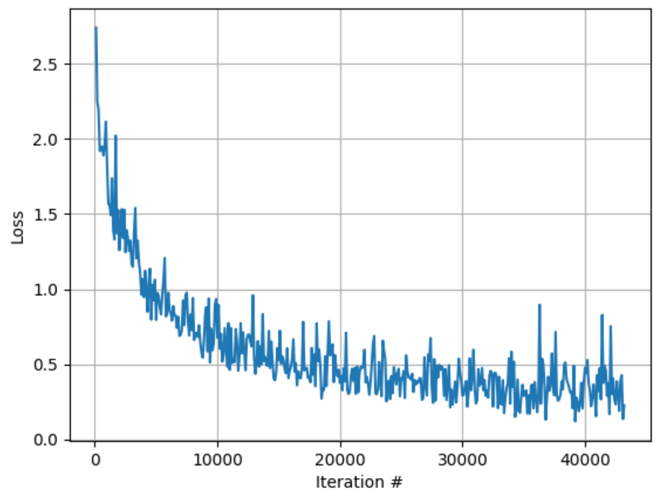

Figure 4: Validation loss with batch normalization