Istanbul Technical University- Spring 2017-2018
COMPUTER VISION
Homework 3
Two-Layer Network

Aydin Ayanzadeh
student number: 504161503
ayanzadeh17@itu.edu.tr
a.ayanzadeh@gmail.com

March 28, 2018

1)

The proposed neural network in for this homework is a two layers fully connected neural network. According to the CS231[1], by fully connected we mean neurons between two neighbor layers are fully connected to each other but neurons of a single layer do not have any connections. In Figure 1 example of such networks with 3 inputs, and two hidden layers which respectively have 4 and 2 neurons, can be seen . In proposed network input size is 32*32*3(Input size of CIFAR10 datasets), moreover first hidden layer has 50 neurons and second hidden layer has 10 neurons which gives scores for each class in dataset (size of class species). Another thing to consider is that this network uses a ReLU activation function after a first hidden layer and uses a softmax classifier at the end. As a note is initialization step, which in that weights are initialized to small random values.

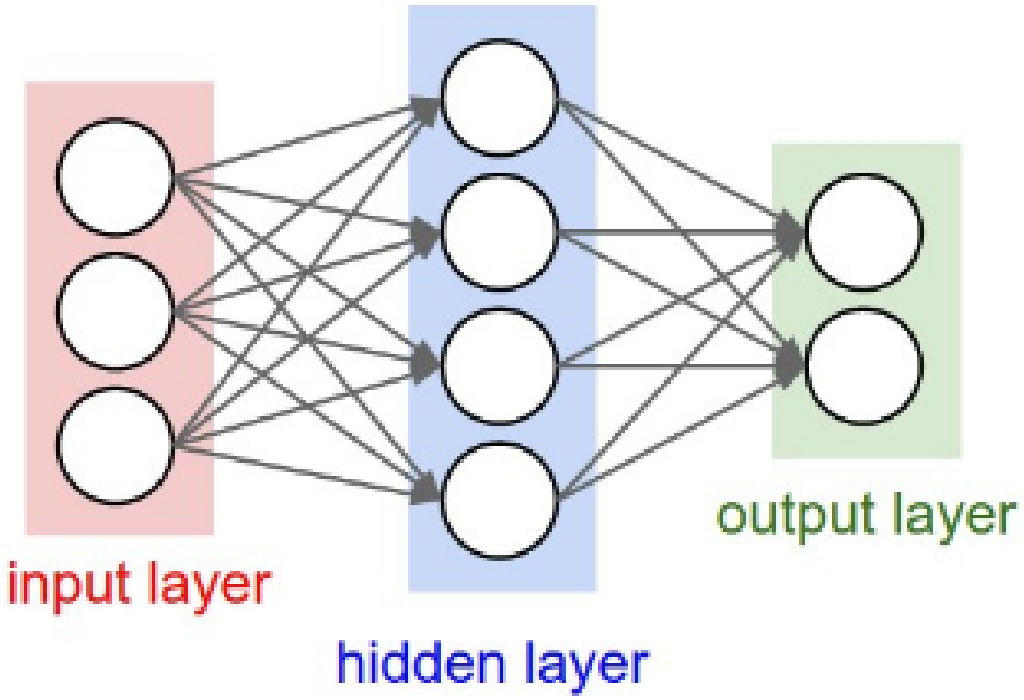

Figure 1: 3-layer Neural Network, with 2 hidden layer

[1]http://cs231n.github.io/neural-networks-1/

2) Forward pass

compute scores: For Implementing the first part of the forward pass which uses the weights and biases to compute the scores for all inputs, Firstly, we define function for calculating ReLU function. Afterwards, we compute the dot product of inputs of X with the weight of first layer. Adding the bias of first layer (b1) will give us output of all of first layer. however, we have to send the results to ReLU function before sending to further layer. At the end of this part, we will have another dot product for h and second layers weights (W2) and adding up the bias of the second layer(b2) gives the scores. In this layer we do not have the Relu.

$$f(x) = \max(0, x) \tag{1}$$

compute loss: In Softmax loss we need to calculate log of where f j is the j_th element of the vector of class score f. In the following line of the code Ive implement this formula for each dataset and I have saved it in s vector.

In further steps, we should subtracts the score of correct class from s for each datapoint. And then adds all of the elements of result vector. Now we have the a loss which is a sum over all training examples. Finally, we have divided the score by number of input datapoints and add the regularization term to the loss.

$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \tag{2}$$

$$L_i = -log\left(\frac{e^{f_{y_i}}}{\Sigma_j e^{f_j}}\right) \tag{3}$$

$$R(W) = \Sigma_k \Sigma_l W_{k,l}^2 \tag{4}$$

$$L = \frac{1}{N}\Sigma_i L_i + \frac{1}{2}\lambda R(W) \tag{5}$$

Where f_{y_i} is score for the correct class, N is the number of input images, k is number of classes, and l is number of layers. We first calculate the natural exponential of scores, and sum of them which are nominator and denominator in equation 2. Afterwards, we calculate probabilities for all classes. Then, to compute the loss for every image, we take log of correct classes probabilities using equation 3, and use 5 to computes total loss.

compute gradients in Backward pass: For computing the gradients of weight1, weight2 and b1 and b2 I follow instructions on Stanford CS231 course. Fist thing first, for computing the gradients we need to calculate gradient of scores which have been calculated using Softmax. Following formula is what we are going to implement:

For this aim we need normalized probabilities that can be calculated using following lines of codes

Now, we can compute gradient of scores using probs. Please note that we need to divide results by number of input datapoints (N).

Then, by doing a dot product between activation's of first hidden layer and gradient of scores and also adding gradient of regularization we get gradients of weights for second layer of network:

Now we have gradients for weights of second layer and we want to back-propagate to layer one. For this, first we do a dot product for gradient of scores and weights of second layer and back-propagate through ReLU function:

Figure 2: diagram of changing the loss with number of iterations

Finally, we can compute gradient of weights and bias for first layer using following lines of codes, similar to what we did for second layer:

$$\frac{\partial L_i}{\partial f_k} = p_k - 1(y_i = k)$$

$$\frac{d}{dw}(\frac{1}{2}\lambda w^2) = \lambda w$$

$$\frac{dr}{dx} = 1(x > 0)$$

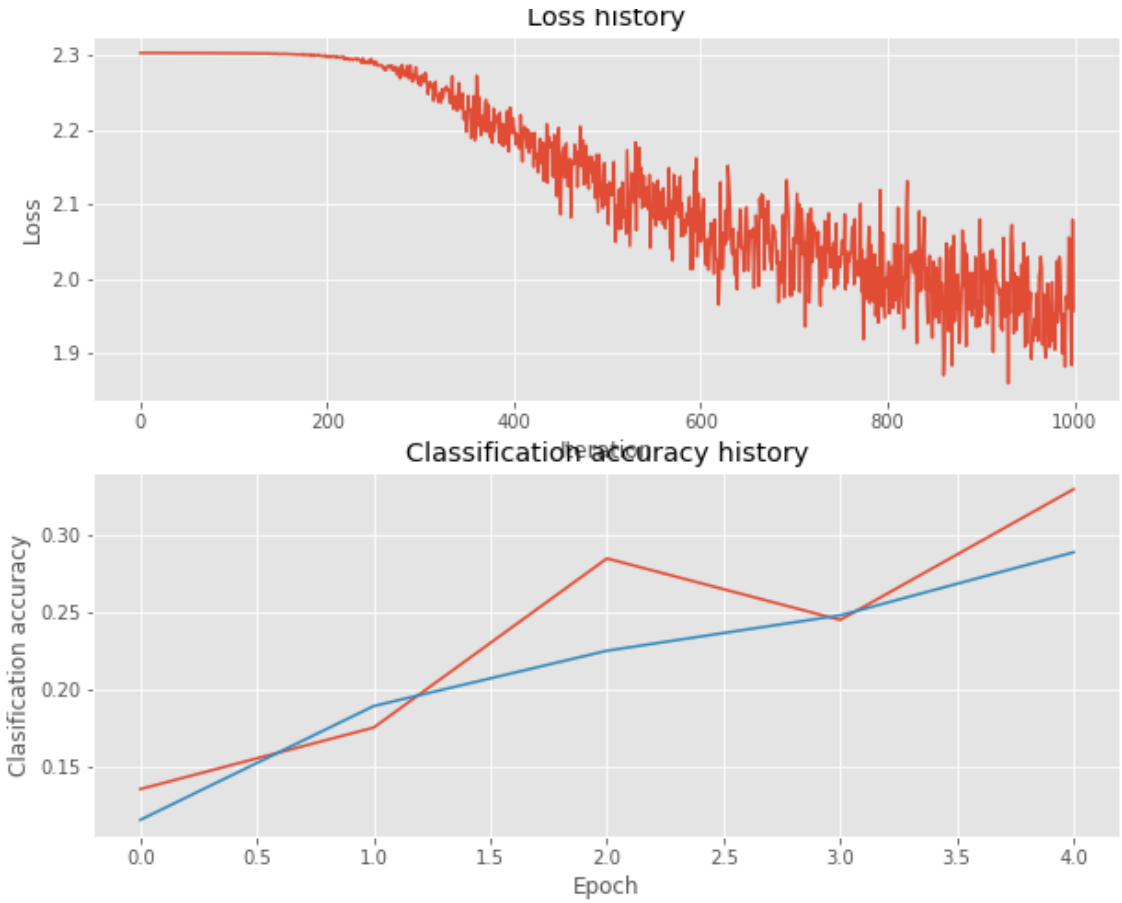

Figure 3: loss history for 1000 iterations, bottom: classification accuracy history for both train and validation sets

Tuning the hyper-parameters At first the performance is not very good and validation accuracy is 0.287, however after hyper-parameters tuning in which we experiment architectures with different hidden layer sizes (50, 128,164, 256, 512), learning rates [1e-3, 1.2e-3, 1.4e-3, 1.6e-3], and regularization strength has assumed [1e-1,3e-1,4e-1], we found the best architecture with hidden layer size 512, learning rate 0.00160, and regularization strength 0.1, and the validation accuracy of 0.54 percent with the explained hyper parameters. Using final model on test images achieves **0.5430(about 2 percent upper than determined accuracy)** test accuracy: . Figure 4 shows the visualization of weights learned in the layer.

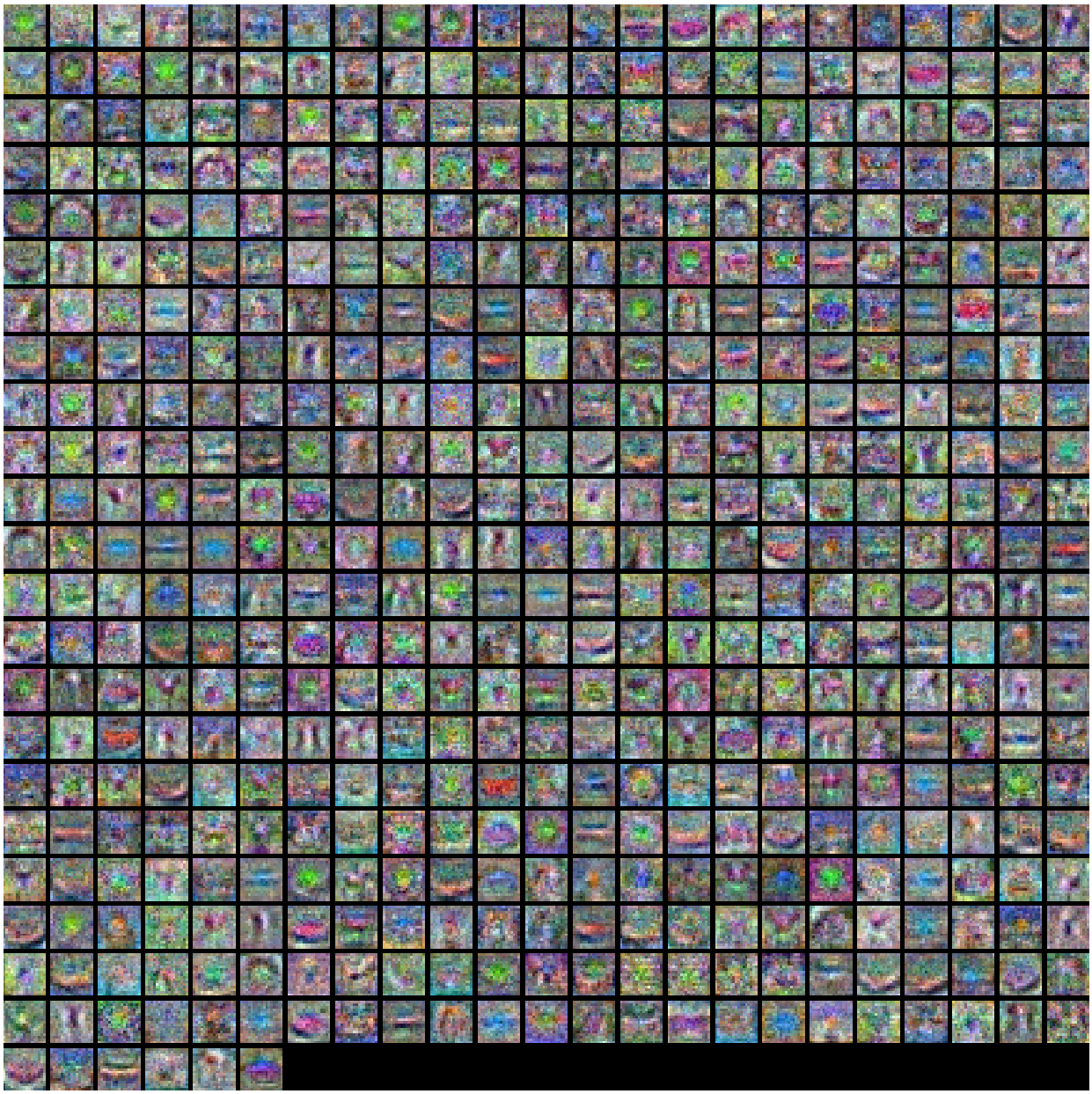

Figure 4: Visualization of first layer weights.