Istanbul Technical University- Spring 2018

BLG 506E COMPUTER VISION

Assignment 4

Aydin Ayanzadeh

student number: **504161503**

ayanzadeh17@itu.edu.tr

Q1) Initialization of parameters of Neural Network is a very important steps of implementing a good deep network. Batch Normalization is a new technique which is developed in 2015 by Loffe and Szegedy [1]. This technique help us to initialize the weights of network as zero mean and unit Gaussian distribution. According to [1] batch normalization not only allow us to be less careful about parameter initialization but also let us to use much higher learning rate and eliminates the need for dropout. When it comes to implementation, we add a batch normalization layer right after convolution of fully connected layer and before non linearity. In this technique we normalize each d-dimensional input x using following perfectly differentiable function.

Sergey Ioffe [1] believe that normalizing each input of a layer is capable to change what that layer can present, so to overcome this problem, they introduce pairs of parameters γ ^(k) , β ^(K) . Using them we scale and shift normalized value and we make sure that transformation which we use in the network can represent the identity transform [1].

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}$$

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1. Batch Normalizing Transform, applied to activation x over a mini-batch [1]

Q2: Batch Normalization

1- Batch Norm Forward

For completing the forward path of batch normalization for training, I have simply implemented above algorithm using following codes.
As suggested in the assignment I have updated the running averages for mean and variance using exponential decay and based on the momentum parameter. In order to implement batch normalization for test time, as described in assignment file, and in contrast with [1] I have used running average without additional estimation step. Please note that paper suggests a different test-time approach. My implementation can be seen as follow:

```
#Find the mean of  mini-batch
      sample_mean=np.mean(x,axis=0)
#find variance
      sample_var=np.sum((x-sample_mean)**2,axis=0) / N
# normalizing
Xhat: find distaance of x and sample_mean over(  eps for numeric stability and take root
from them)
      Xhat = (x - sample_mean) / np.sqrt(sample_var + eps)
#scaling and shifting
      out = gamma * Xhat + beta
#stor the variables in cache
      cache = (x, eps, gamma, beta, Xhat, sample_mean, sample_var)

  #running_mean = momentum * running_mean + (1 - momentum) * sample_mean
#please follow the ipyn explanation
      running_mean = momentum * running_mean + (1 - momentum) * sample_mean
 #  running_var = momentum * running_var + (1 - momentum) * sample_var
      running_var = momentum * running_var + (1 - momentum) * sample_var
```

1- Batch Norm Backward

Using computational graph makes back-propagation processes more easier. We take local gradients and multiply with gradient flowing from above in reverse order of forward pass. Source code for backward pass is written in file: cs231n/layers.py,

Step 9: It is an addition gate and the local gradients are 1, so gradients from are equally Distributed.

Step 8: It is a multiply gate so its local gradients are switched input values. We calculate the gradients of inputs by chain rule, multiplying local gradients with gradient of output.

Step 7: It is again multiply gate and swaps the inputs and multiply with gradient of Output.

Step 6: local gradient is −1/(x^2)and we multiply it with gradient flowing from top layer using the chain rule.
Step 5: Local gradient is −1 /2*sqrt(x) and we multiply it with gradient from above.

Step 4: It is like add gate and we distribute gradient from above over all samples in the Mini-batch

Step 3: Local gradient is 2x and we multiply it with gradient from above.
Step 2: We sum up two gradients coming to one gate. Local gradients are -1 and 1, so multiply it with gradient from above and sum up the result for sample mean gradient over rows.
Step 1: It is like add gate and we distribute gradient from above over all samples in mini-batch.
Step 0: Finally, we add two gradients coming from above to get dx.

```
#unpacking the variable of Fw pass
   x,eps, gamma, beta,Xhat,running_mean,running_var=cache
   N,D=x.shape
#calculate the gradients of input by chain rule, multiplying local gradients gradient of output

   dXhat=dout*gamma
#multiply local gradient over  gradient flowing
 # from top layer by chain rule
   dvarX=running_var+eps
 #local gradient is -0.5 * (1/sqrt(var + eps)) and we multiply it with
```

```
 # gradient from above

grad_running_var=np.sum(np.multiply(dXhat,x-running_mean),axis=0)*(-0.5)*(dvarX)**
(-3.0/2.0)

                                    d_running_mean=np.sum(dXhat*(-1.0/np.sqrt(dvarX)          )
,axis=0)+grad_running_var*(np.sum(-2*(x-running_mean),axis=0)/N)
#add gate and we distribute gradient from above over all rows in each column

dx=dXhat*(1.0/np.sqrt(dvarX))+grad_running_var*2*(x-running_mean)/N+d_running_
mean*1.0/N
   dgamma=np.sum(dout*Xhat,axis=0)
   dbeta=np.sum(dout,axis=0)
```

BatchNorm Backward Alternative:

In back-propagation we used concept of computational graphs which makes the task of calculating gradients easier. In addition, we can take the derivatives of equations in fig2 and use them in back-propagation. It might be helpful in small networks like this, however in complex network and sometimes impossible to take derivatives output with respect to inputs.

Comparison Results of batch normalization

After analyzing the result with the help of batch normalization we can reaching an conclusion that:

1. much more robust in the bad initialization
2. much more robust to avoid overfitting（one some scale without bn the model the a bigger gap between val and train）
3. prevent vanishing gradient and gradient explosion problem

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$
$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$
$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

Fig 2: Derivatives of batch normalization.

```
#unpacking the variable of Fw pass
   x,eps, gamma, beta,Xhat,running_mean,running_var=cache
   N,D=x.shape
#calculate the gradients of input by chain rule, multiplying local gradients
gradient of output

   dXhat=dout*gamma
#multiply local gradient over  with gradient flowing
  # from top layer using chain rule
   dvarX=running_var+eps
  #local gradient is -0.5 * (1/sqrt(var + eps)) and we multiply it with
  # gradient from above

grad_running_var=np.sum(np.multiply(dXhat,x-running_mean),axis=0)*(-0.5)*(d
varX)**(-3.0/2.0)

                        d_running_mean=np.sum(dXhat*(-1.0/np.sqrt(dvarX)        )
,axis=0)+grad_running_var*(np.sum(-2*(x-running_mean),axis=0)/N)
```

```
#add gate and we distribute gradient from above over all rows in each column

dx=dXhat*(1.0/np.sqrt(dvarX))+grad_running_var*2*(x-running_mean)/N+d_ru
nning_mean*1.0/N
   dgamma=np.sum(dout*Xhat,axis=0)
   dbeta=np.sum(dout,axis=0)

 x, eps, gamma, beta, Xhat, running_mean, running_var = cache
   N, D = x.shape
   Mean_dist=x - running_mean
#gradient with the respect of gamma
   dgamma=np.sum(dout*Xhat,axis=0)
#gradient withe the respect of Beta
   dbeta=np.sum(dout,axis=0)
   run_sum=running_var + eps
     dx1=(N * dout - np.sum(dout, axis=0)- (Mean_dist) * (run_sum) ** (-1.0) *
np.sum(dout * (Mean_dist), axis=0))
#gradient with the respect of x
   dx2=(1.0 / N) * gamma * (run_sum) ** (-1.0 / 2.0)
   dx = dx2*dx1
```

Q3: Fully Connected Neural Networks

We followed steps in FullyConnectedNets.ipynb

1) Affine forward

In this part just like previous assignment, we need to multiply the input by wight matrix and then we should add the bias. Please note that before multiplying we need to reshape the input. Affine forward code are written here.

```
  row_x = np.reshape(x, (x.shape[0], -1))
  out=np.dot(row_x,w)+b
```

2)Affine_backward

When it comes to backward path, its simple and very similar to backward path. In this case we get the gradient of top layer and inputs and weights of layer as input. The gradient with respect to x can be calculated by multiplying the gradient of top by weights of current layer. Please note that we need to reshape the results in order to be like shape of the x. Similar to gradient with respect to x, gradient with respect to W can be computed by multiplying the x transpose and gradient of top. Please consider that, we should reshape the x to be able to do the dot product. Gradient with respect to bias can be computed by taking a sum over axis zero of gradient of top. The codes of this part are written below:

```
# Gradient with respect to x and W
  dx1=np.dot(dout,w.T)
  dx = np.reshape(dx1,x.shape)
# Gradient with respect to b
  db=np.sum(dout,axis=0)
  dw=np.dot(np.reshape(x, (x.shape[0], -1)).T,dout)
```

3) relu_forward

Forward path for rectified linear units (RelUs) can easily implemented by following line of code:

```
out = np.maximum(0, x)
```

4) relu_backward

ReLU layer just compares inputs with zero and takes the maximum. Inspired by the fact that in backward path, max gate acts like a router I have implemented the relu_backward path as following:

```
    dx=dout
    dx[x<=0]=0
```

5) Two layer network

In this part of assignment, we are asked to implement a two layers neural network using modules that we have implemented in previous steps. My implementation for this part can be seen in fc_net.py file. This implementation is very similar to our previous assignment's implementation, the only difference is using modules. In __init__ function we have simply define some parameters and also initialize weights and biases for both of layers.

```
    self.params['W1'] = weight_scale * np.random.randn(input_dim, hidden_dim)
    self.params['b1'] = np.zeros(hidden_dim)
    self.params['W2'] = weight_scale * np.random.randn(hidden_dim, num_classes)
    self.params['b2'] = np.zeros(num_classes)
#Implement the forward pass for the two-layer net, computing the    #
    # class scores for X and storing them in the scores variable.
    affine_relu_out, affine_relu_cach = affine_relu_forward(X, self.params['W1'],
self.params['b1'])
    affine_out, affine_cache = affine_forward(affine_relu_out, self.params['W2'],
self.params['b2'])
    scores = affine_out
```

In loss function, I use provided functions in *layer_utils.py* file. For computing the scores I use affine_relu_forward function which gives us activations of first layer. Then I use affine_forward function which is using for computing scores. Please note that difference of this two functions is that the former doesn't have ReLU layer. Computing gradients of weights and biases can be done using affine_backeard function which I implemented in previous part and also provided affine_relu_backward. Firs I use affine_backward in order to calculate gradients with respect to w2, b2 and outputs of first layer. Afterwards, we use affine_relu_backward to calculate gradient with respect to weights and biases and also input vector.

6)solver

Using following model I could get **53.80%**validation accuracy on dataset with the given set(batch size,learning rate, learning_decay). Figure2 shows visualization of training loss nad train / validation accuracy for following model.

```
# i just put this part of code for good explanation of approach.
solver = Solver(model, data,
   update_rule='sgd',
   optim_config={
   'learning_rate': 1e-3,
   },
   lr_decay=0.8,
   num_epochs=10, batch_size=100,
   print_every=100)
```

7) Multilayer network

This part of assignment aims to implement a fully connected network with an arbitrary number of hidden layers. It contains 3 part. Initialization, computing score and finally computing loss and gradients. In initialization part we need to initialize and store weight and bias for all layers of network. For this aim I create a list which respectively contains input size, size of all hidden layers and size of final layer.

Then I use this list in following manner to initialize weights and biases for all of layers, In order to compute the scores from first layer we start and iteratively calculate the activations of each layer. Please note that for this aim I use affine_relu_forward function. But we can't use affine_relu_forward function for last layer, so I've implemented last layers procedure separately using affine_forward function.

With the help of softmax function we can get loss and gradient with respect to scores. But please note that we need to add regularization term too.Afterwards, we need to compute gradients with respect to weights and biases. In this case also I have implemented it in two parts. First I compute the gradients with respect to last layers weights and biases using affine_backward function. And in next step I compute gradient of hidden layers using affine_relu_backward function. In the below you can find the result of implementation (without using dropout and batch normalization).

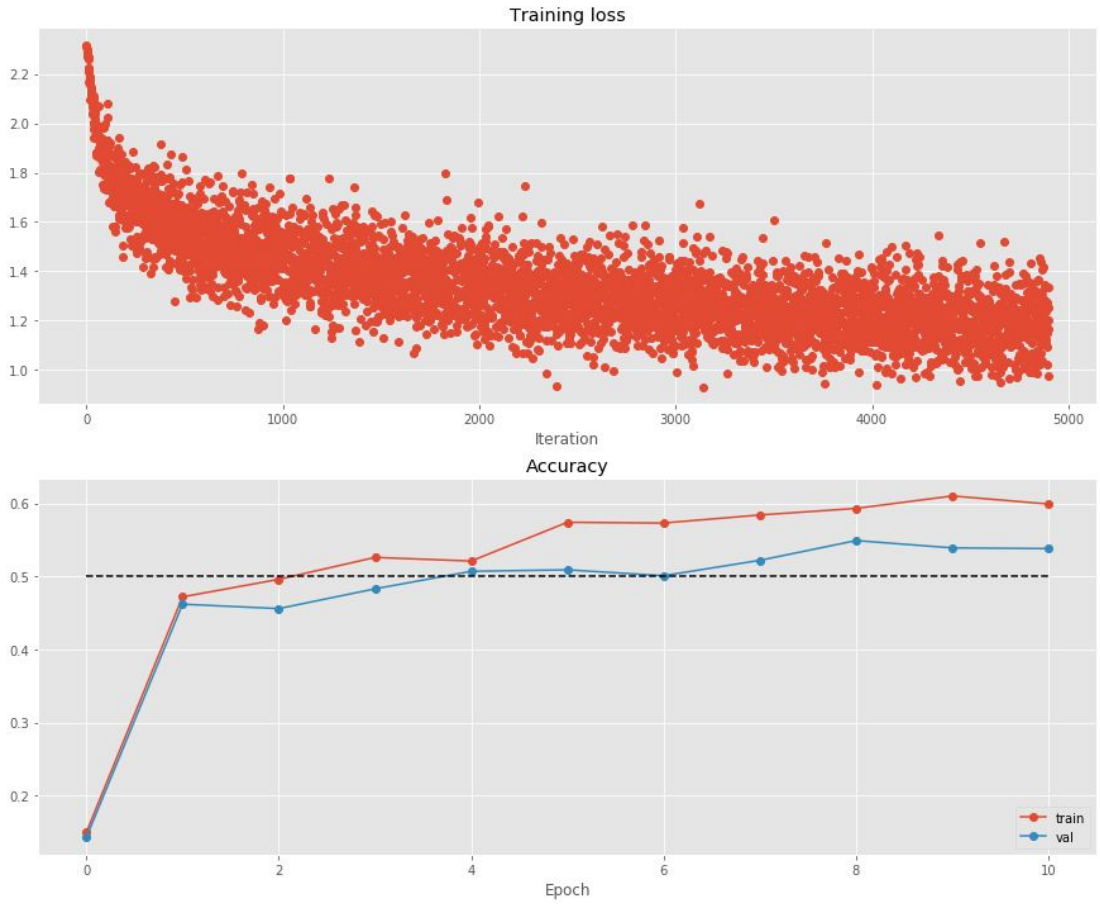

Fig2.Visualization of training loss and train and val accuracy for proposed approach.

in the Fig3 you will see the visualization of three-layer Network to overfit 50 training examples. with the given iteration.

Running check with reg = 0
Initial loss: 2.29461318783
W1 relative error: 1.40e-07
W2 relative error: 2.98e-06
W3 relative error: 1.05e-07
b1 relative error: 8.84e-09
b2 relative error: 1.31e-09
b3 relative error: 9.81e-11

Running check with reg = 3.14
Initial loss: 6.71422109668
W1 relative error: 1.13e-07
W2 relative error: 8.49e-08
W3 relative error: 5.27e-08
b1 relative error: 1.27e-06
b2 relative error: 1.43e-09
b3 relative error: 2.22e-10

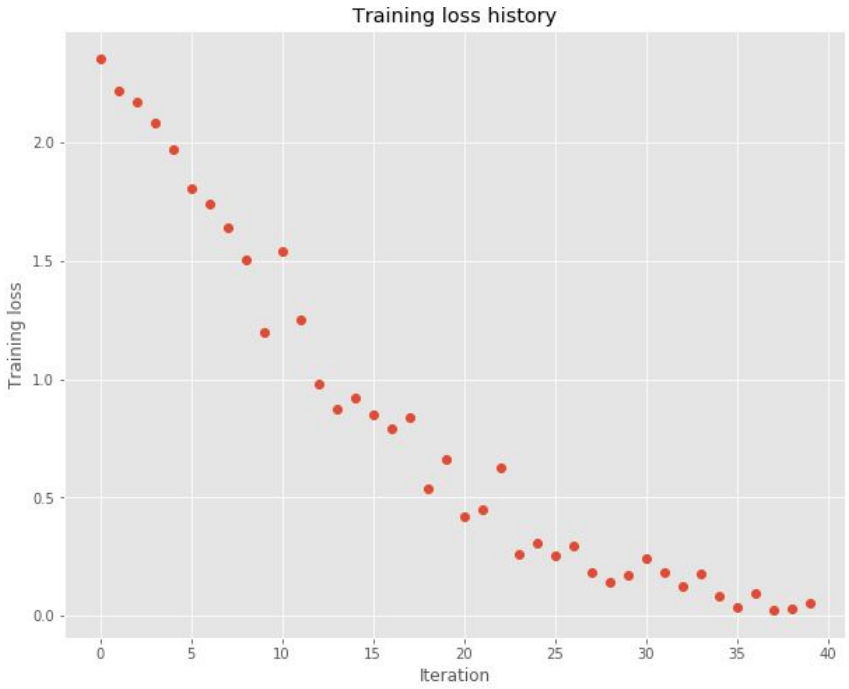

Fig3.three-layer Net visualization to overfit 50 training examples. In the given iteration

Note: for rest part of the code for explained part in above(**Multilayer network**), please check the fc_net.py for more detail. In there, First we initialize neural network parameters for FullyConnectedNet in file *fc_net.py* Based on the using implemented convenience layers of forward pass, we calculate scores (lines 241-259). We check if we should use batch-norm and add batch-norm layer after affine forward.we have the implementation of the forward pass for the fully-connected network. Finally, we will have the convenience layers of backward pass for back-propagation (lines 293-313).

Reference

[1] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” ArXiv 150203167 Cs, Feb. 2015.