BLG 506E COMPUTER VISION ASSIGNMENT 4

We start with FullyconnectedNets

1- We work on CIFAR10-dataset at this assignment. It has 50000 Train Data and 10000 Test Data. Firstly, we load CIFAR10 dataset. We split the data into 49000 train, 1000 validation and 1000 test sets.

2- affine_forward function

First of all, an affine layer or fully connected layer is a layer of an artificial neural network in which all contained nodes connect all nodes of the subsequent layer.

For every connection to an affine layer, the input to a node is a linear combination of the outputs of the previous layer with an added bias. The output of a node is then calculated by passing this input through an activation function. Mathematically, this is expressed as :

$y = f(W * x + b)$

affine function at this assignment has defined as affine_forward(x, w, b).

- x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
- w: A numpy array of weights, of shape (D, M)
- b: A numpy array of biases, of shape (M,)

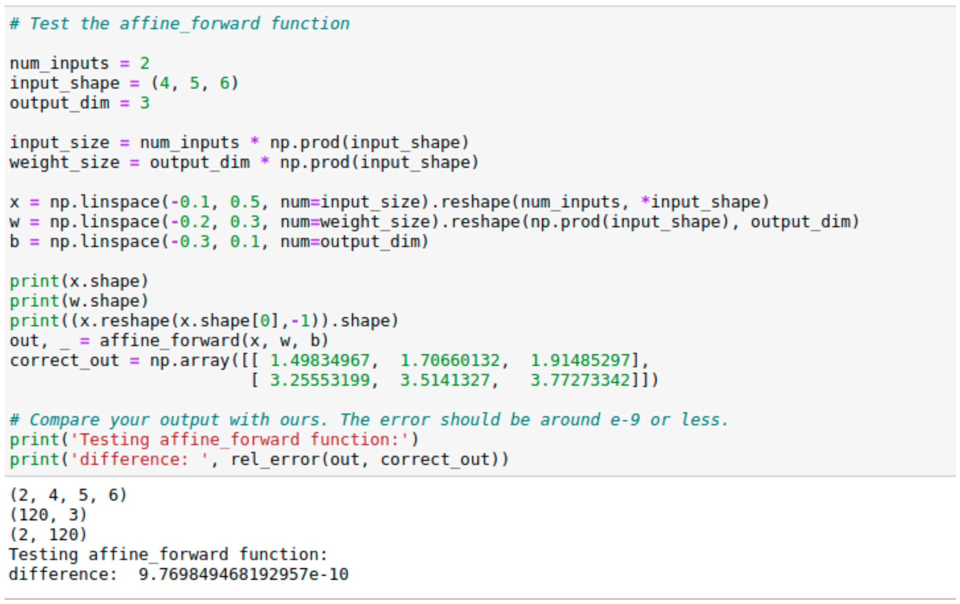
returns function output and cache. Shape of output is (N,M)

```
def affine_forward(x, w, b):
        out = None
###################
        N_tr = x.shape[0]
        x0 = x.reshape(N_tr,-1)
        out = x0.dot(w) + b
        cache = (x, w, b)
#####################
        return out, cache
```

x.shape[0] gives us training number.

We will multiply X by W, but here is a matrix product. The size of x (2, 4, 5, 6). Here is the number of 2 entries. The size of w is (120, 3). As a result, we cannot multiply them. If we declare one of its dimensions and make the other "-1" in Python while giving it a matrix, it matches the shape of the new matrix to the first one. So if we give the 1st dimension 2, the other dimension is 120 from 4 * 5 * 6.

Then we dot product x0 with w and add bias



```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

print(x.shape)
print(w.shape)
print((x.reshape(x.shape[0],-1)).shape)
out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
(2, 4, 5, 6)
(120, 3)
(2, 120)
Testing affine_forward function:
difference:  9.769849468192957e-10
```

3- affine_backward function

out = x0 * w + b, we derive using chain rule. So,

dx0 = dout * w

dw = dout * x0

db = dout *1 so the expression just sums of dout.

finally because of we have reshaped x to x0;

dx = dx0.reshape(x.shape)

```
def affine_backward(dout, cache):
          x, w, b = cache
           dx, dw, db = None, None, None
          out = None
##########################
          N_tr = x.shape[0]
```

```
        x0 = x.reshape(N_tr,-1)
        db = dout.sum(axis=0)
        dw = x0.T.dot(dout)
        dx0 = dout.dot(w.T)
        dx = dx0.reshape(x.shape)

############################
        return dx, dw, db
```

4- relu_forward and relu_backward

```
def relu_forward(x):
        out = None
##########################
        out = np.maximum(0, x)
#########################
        cache = x
        return out, cache
```

ReLU activation function = max(0, x).

if x>0, it is x and if x<0, it is 0.

```
def relu_backward(dout, cache):
        dx, x = None, cache
##########################
        dx = np.zeros(x.shape)
        dx[x>0] = 1
        dx = dx * dout
```

dReLU(x)/dx => if x>0, it is 1; otherwise it is 0

Inline Question 1: We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

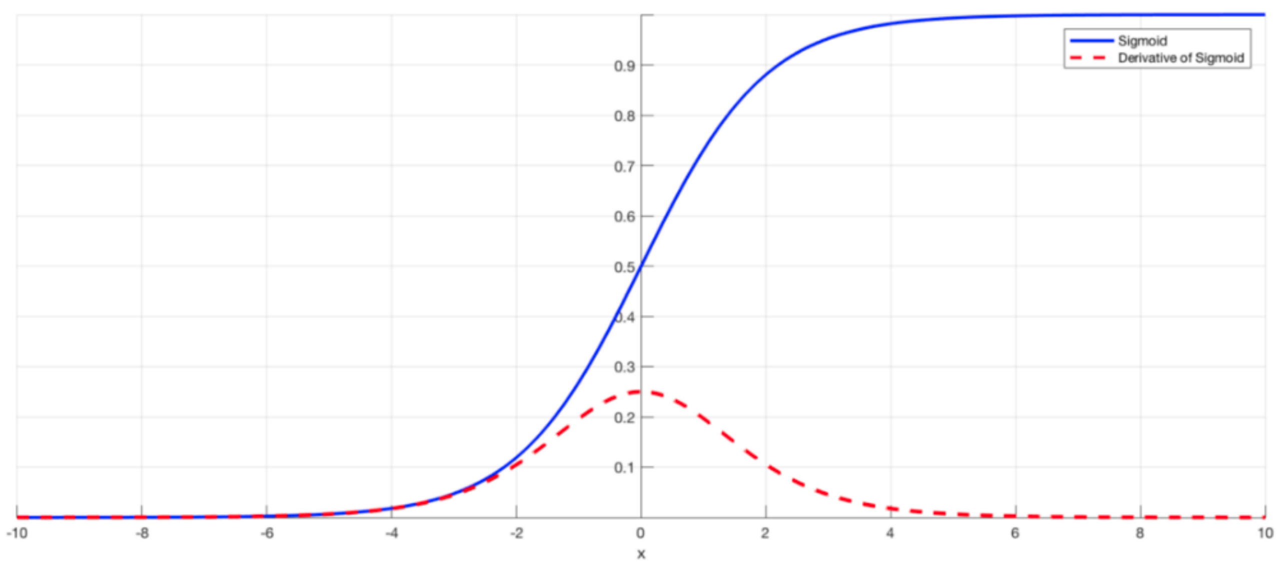
1.Sigmoid

2.ReLU

3.Leaky ReLU

Answer*:*

1.Sigmoid: When neuron's activation saturates at 1and 0, the gradient becomes almost zero. This creates difficulties in learning.



2. ReLU: If inputs tend to make x<= 0 then the most of the neurons will always have 0 gradient updates hence closed or dead.

3.Leaky ReLU: It solves the dead ReLU problem. 0.01 is coefficient of leakage.

So the answer is Sigmoid and ReLU.

5- affine_relu_forward and affine_relu_backward

We run affine_relu_forward and affine_relu_backward function.

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12

As you can see, the derivative error is very low.
```

6- Complete TwoLayerNet class(__init__ and loss method)

def __init__(self, input_dim=3*32*32, hidden_dim=100, num_classes=10, weight_scale=1e-3, reg=0.0):

    self.params = {}

    self.reg = reg

###

    self.params["W1"] = np.random.normal(scale = weight_scale, size = (input_dim, hidden_dim))

    self.params["b1"] = np.zeros(hidden_dim,)

    self.params["W2"] = np.random.normal(scale = weight_scale, size = (hidden_dim, num_classes))

    self.params["b2"] = np.zeros(num_classes,)

Here, we initialize the weights and biaeses. But weights should be intialized from a Gaussian centered at 0.0.

For normal(Gaussian) distribution, we use numpy.random.normal(loc=0.0, scale, size) function. Because of default value of loc paramater is 0.0, I dont change it and scale parameter is weight_scale. Then we store in the dictionary params.

def loss(self, X, y=None):

    scores = None

##

    W1, b1 = self.params['W1'], self.params['b1']

    W2, b2 = self.params['W2'], self.params['b2']

```
        relu1_out, relu1_cache = affine_relu_forward(X, W1, b1)
        X2 = relu1_out
        scores, relu2_cache =  affine_relu_forward(X2, W2, b2)
```

###

Firstly, we calculate forward pass. We take weights and biases parameters from params dictionary. Then we need to apply fully connected layer and use ReLU. For this, we use affine_relu_forward function two times. We send the result of the first layer+ReLU to second layer. After apply second affine layer and use ReLU activation function, we assignment the result to scores variable.

###

```
        loss, dscores = softmax_loss(scores, y) #softmax_loss(x,y) returns loss, dx
        loss += 0.5 * self.reg * ( np.sum(W1 * W1) + np.sum(W2 * W2) )


        dX2, dW2, db2 = affine_relu_backward(dscores, relu2_cache)
        dW2 += self.reg * W2
        dX1, dW1, db1 = affine_relu_backward(dX2, relu1_cache)
        dW1 += self.reg * dW1


        grads['W2'] = dW2
        grads['b2'] = db2
        grads['W1'] = dW1
        grads['b1'] = db1
```

###

```
        return loss, grads
```

Secondly we calculate backward pass. Here firstly we compute data loss using Softmax. softmax_loss(x, y) returns loss and dx. So,

Loss, dscores = softmax_loss(scores, y)

Then we add L2 regularization. Then we use affine_relu_backward function two times and add regularization. Then we store weights and biases at dictionary grads.

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 1.00e+00
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

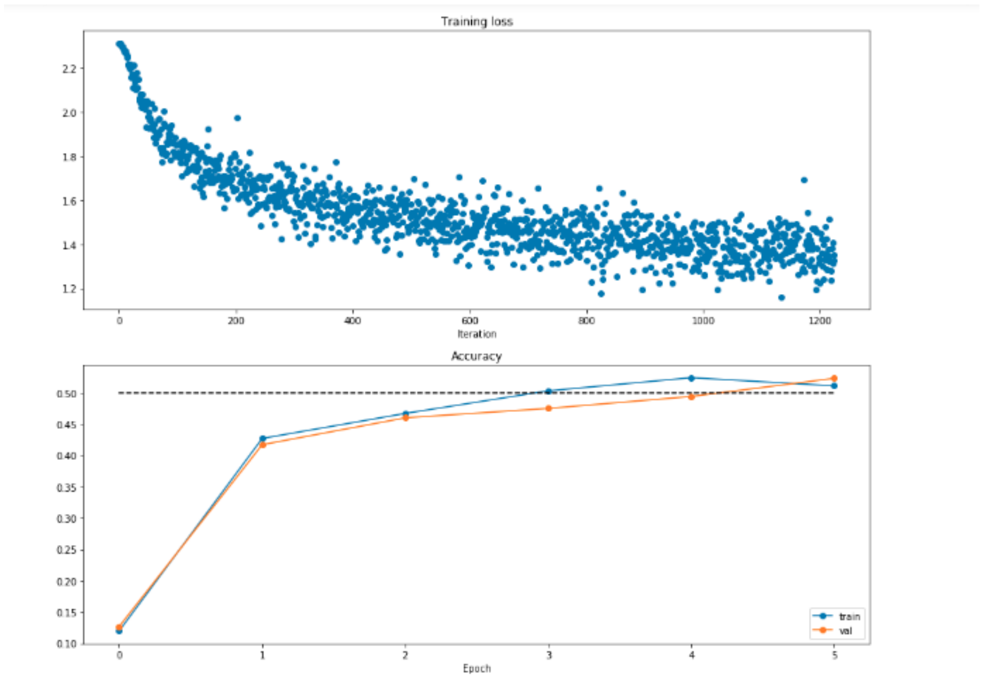
7-

```
model = TwoLayerNet()

solver = None

###############

best_val = -1

#We tune hyperparameters(hidden layer)

hidden_sizes = [50, 100, 150, 200]

max_count = 5

for hidden_size in hidden_sizes:

    for count in range(max_count):

        #We tune hyperparameters(regularization, learning rate)

        reg = 10**np.random.uniform(-6,4)

        lr = 10**np.random.uniform(-3,-4)

        #We create Two Layer Neural Network and we initialize Two Neural Network

        model = TwoLayerNet(hidden_dim = hidden_size, reg= reg)


        solvernet = Solver(model, data, update_rule='sgd',

                optim_config={'learning_rate': 1e-3},

                lr_decay=0.95, num_epochs=5,

                batch_size=200, print_every=1000000,verbose = False)


        #We train

        solvernet.train()
        # Predict on the validation set
        val_accuracy = solvernet.best_val_acc

        # Store best values
        if (val_accuracy > best_val):
            best_val = val_accuracy
            solver = solvernet

        # Print results
        print('lr %e reg %e hid %d  val accuracy: %f' % (lr, reg, hidden_size, val_accuracy))
print('best validation accuracy achieved: %f' % best_val)
print('solver: ',solver)
###############
```

We tune hypeparameters(regularization, learning rate and hidden size). We create Two Layer Network with hidden dim and regularization parameters. Then we use Solver to train a TwoLayerNet.

And we achieves %50 accuracy.

```
lr 5.018968e-04 reg 3.115590e-06 hid 200  val accuracy: 0.500000
lr 2.964274e-04 reg 2.204433e-05 hid 200  val accuracy: 0.496000
lr 7.694918e-04 reg 1.874784e+00 hid 200  val accuracy: 0.520000
lr 2.375050e-04 reg 7.417132e-03 hid 200  val accuracy: 0.505000
best validation accuracy achieved: 0.523000
solver:  <cs231n.solver.Solver object at 0x7fa9578cc630>
```



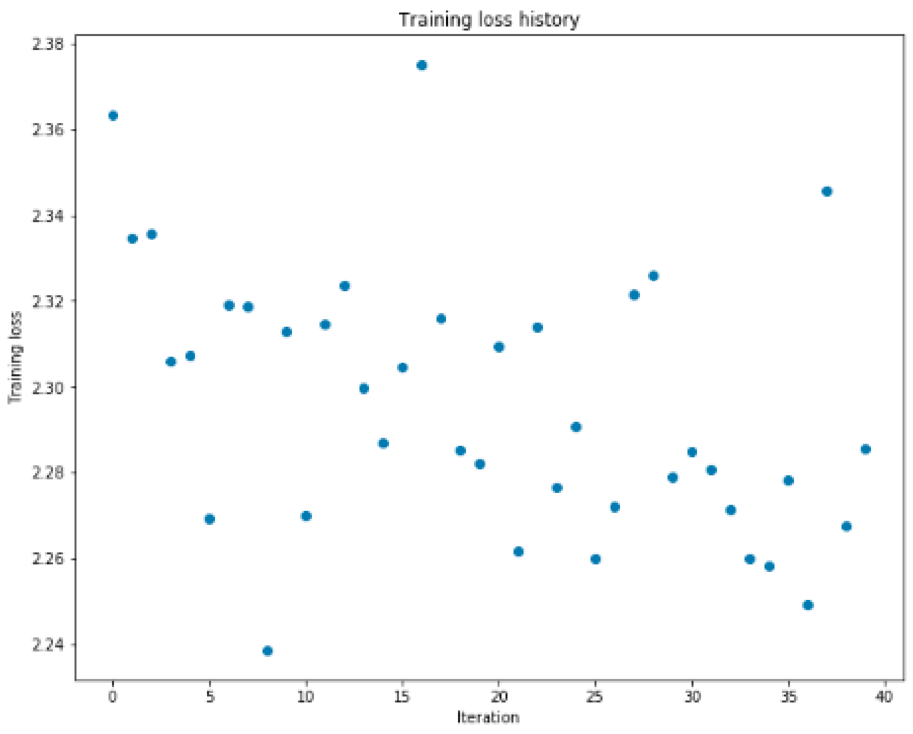
8-

Firstly when we run, weight_scale is equal to 1e-2 and learning_rate is equal to 1e-4.

```python
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

weight_scale = 1e-2   # Experiment with this!
learning_rate = 1e-4  # Experiment with this!
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

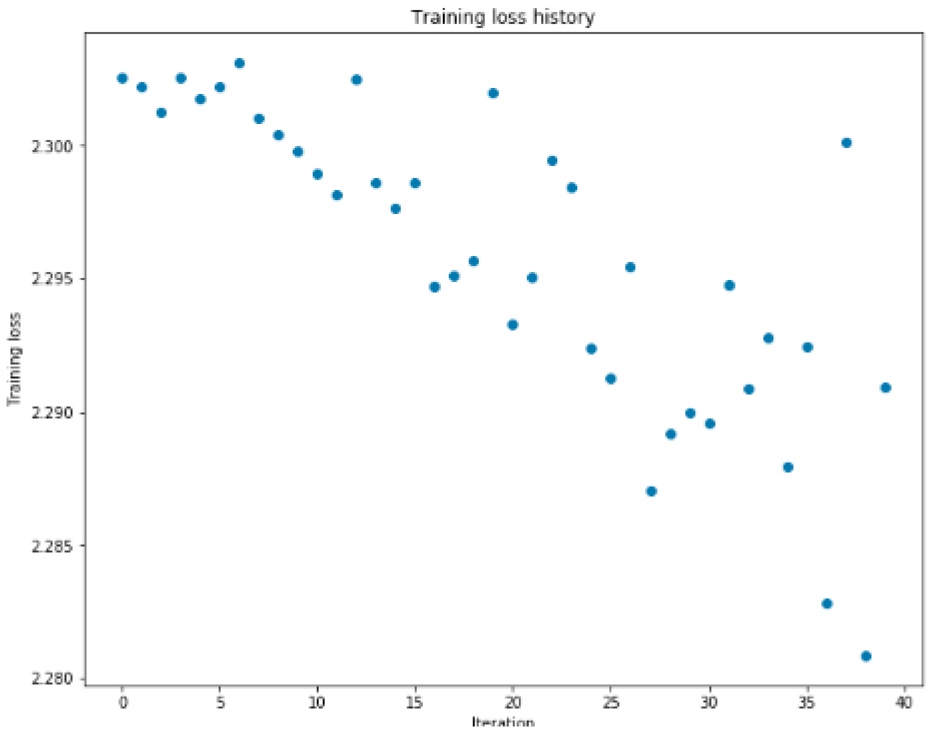
```
(Iteration 1 / 40) loss: 2.363364
(Epoch 0 / 20) train acc: 0.020000; val_acc: 0.105000
(Epoch 1 / 20) train acc: 0.020000; val_acc: 0.106000
(Epoch 2 / 20) train acc: 0.020000; val_acc: 0.110000
(Epoch 3 / 20) train acc: 0.020000; val_acc: 0.110000
(Epoch 4 / 20) train acc: 0.040000; val_acc: 0.109000
(Epoch 5 / 20) train acc: 0.040000; val_acc: 0.111000
(Iteration 11 / 40) loss: 2.270022
(Epoch 6 / 20) train acc: 0.040000; val_acc: 0.111000
(Epoch 7 / 20) train acc: 0.060000; val_acc: 0.112000
(Epoch 8 / 20) train acc: 0.060000; val_acc: 0.111000
(Epoch 9 / 20) train acc: 0.040000; val_acc: 0.110000
(Epoch 10 / 20) train acc: 0.040000; val_acc: 0.109000
(Iteration 21 / 40) loss: 2.309562
(Epoch 11 / 20) train acc: 0.060000; val_acc: 0.110000
(Epoch 12 / 20) train acc: 0.060000; val_acc: 0.110000
```

```
(Epoch 19 / 20) train acc: 0.100000; val_acc: 0.118000
(Epoch 20 / 20) train acc: 0.100000; val_acc: 0.120000
```



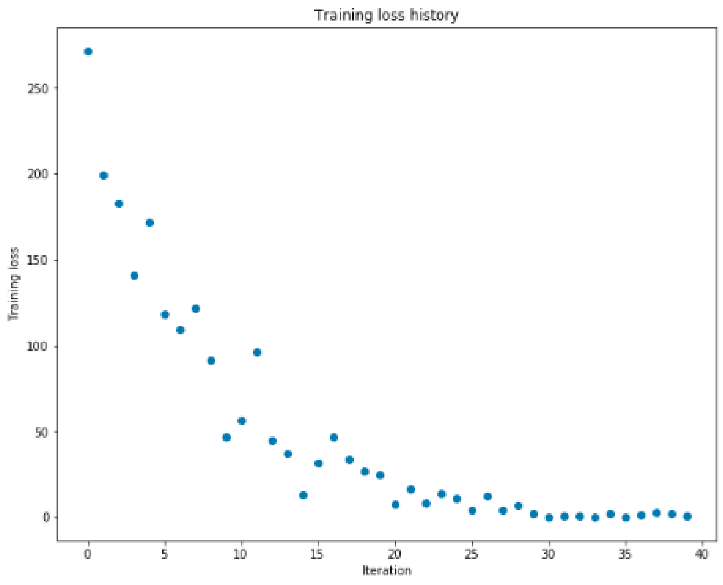
Then we change learning_rate to 2e-2 but the train accuracy was worse. It didn’t reach %100 accuracy.

```
(Epoch 16 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 17 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 18 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 19 / 20) train acc: 0.160000; val_acc: 0.079000
(Epoch 20 / 20) train acc: 0.160000; val_acc: 0.079000
```



Then we change weight_scale to 1e-1 and the train accuracy was better.. It reached %100 accuracy.

```
(Iteration 31 / 40) loss: 0.004566
(Epoch 16 / 20) train acc: 0.920000; val_acc: 0.113000
(Epoch 17 / 20) train acc: 0.940000; val_acc: 0.116000
(Epoch 18 / 20) train acc: 0.940000; val_acc: 0.114000
(Epoch 19 / 20) train acc: 0.920000; val_acc: 0.115000
(Epoch 20 / 20) train acc: 0.980000; val_acc: 0.112000
```



`Inline Question 1:` Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

Answer*:*

Comparing the difficulty of training these two networks, the five layer net was more sensitive to the initialization scale.Although weigt_scale and learning rate are the same, the less layered one seems to have more deviation at the beginning.

9-

def sgd_momentum(w, dw, config=None):

```
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('momentum', 0.9)
    v = config.get('velocity', np.zeros_like(w))
######################
     lr = config.setdefault('learning_rate', 1e-2)
    moment = config.setdefault('momentum', 0.9)

    v = moment * v - lr * dw #integrate velocity
    next_w = w + v #integrate positon
#######################
     config['velocity'] = v
    return next_w, config
```

Firstly we integrate velocity and then integrate positon for momentum update.

def rmsprop(w, dw, config=None):

```
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('decay_rate', 0.99)
    config.setdefault('epsilon', 1e-8)
```

```
        config.setdefault('cache', np.zeros_like(w))
########3
        lr = config.setdefault('learning_rate', 1e-2)
        dr = config.setdefault('decay_rate', 0.99)
        eps = config.setdefault('epsilon', 1e-8)
        cache = config.setdefault('cache', np.zeros_like(w))

        cache = dr * cache + (1 - dr) * dw**2
        next_w = w - lr * dw / (np.sqrt(cache) + eps)

        config['cache'] = cache
#######
         return next_w, config
```

The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead. Here, decay_rate is a hyperparameter and typical values are [0.9, 0.99, 0.999]

RMSProp still modulates the learning rate of each weight based on the magnitudes of its gradients, which has a beneficial equalizing effect, but unlike Adagrad the updates do not get monotonically smaller.

```
def adam(w, dw, config=None):
        if config is None: config = {}
        config.setdefault('learning_rate', 1e-3)
        config.setdefault('beta1', 0.9)
        config.setdefault('beta2', 0.999)
        config.setdefault('epsilon', 1e-8)
        config.setdefault('m', np.zeros_like(w))
        config.setdefault('v', np.zeros_like(w))
        config.setdefault('t', 0)
        #########
        lr = config.setdefault('learning_rate', 1e-3)
        beta1 = config.setdefault('beta1', 0.9)
        beta2 = config.setdefault('beta2', 0.999)
        eps = config.setdefault('epsilon', 1e-8)
         m = config.setdefault('m', np.zeros_like(w))
         v = config.setdefault('v', np.zeros_like(w))
         t = config.setdefault('t', 0)

         t += 1 # modify t
        m = beta1 * m + (1 - beta1) * dw
        mt = m / (1 - beta1 ** t)
        v = beta2 * v + (1 - beta2) * dw**2
        vt = v / (1 - beta2 ** 2)
        next_w = w - lr * mt / (np.sqrt(vt) + eps)

        config['m'] = m
        config['v'] = v
        config['t'] = t
        #########
    return next_w, config
```

I got reference from cs231n neural network document. Here, t is our iteration counter.

Inline Question 3:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

Answer:

The update becomes very small because at each iteration we are adding up the squared gradients. As the optimization progresses the cache can slow down the process more than necessary. Adam optimizer accumulates gradients into it's cache much more slowly thanks to the beta hyperparameter and therefore suffers less under the condition John is seen, but ultimately can slow down as well as much later.

10 -

```
best_model = None

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

best_val = -1

num_classes = 10


#We tune hyperparameters(hidden layer)

hidden_sizes = [100] * 4

max_count = 15


for count in range(max_count):

    #We tune hyperparameters(regularization, learning rate)

    reg = 10**np.random.uniform(-3, 3)

    lr = 10**np.random.uniform(-5, -2)

    ws = 10**np.random.uniform(-2, -1)


    #We create Two Layer Neural Network and we initialize Fully Connected Network

    model = FullyConnectedNet(hidden_dims = hidden_sizes, reg= reg, weight_scale=ws)


    solvernet = Solver(model, data, update_rule='adam',

                optim_config={'learning_rate': lr},

                lr_decay=0.95, num_epochs=5,

                batch_size=200, print_every=1000000,verbose = False)


    #We train

    solvernet.train()


    # Predict on the validation set
    val_accuracy = solvernet.best_val_acc


    # Store best values
    if (val_accuracy > best_val):
        best_val = val_accuracy
        best_model = model


    # Print results
    print('lr %e reg %e ws %e   val accuracy: %f' % (lr, reg, ws, val_accuracy))


print('best validation accuracy: %f' % best_val)
print('best model: ',solvernet)


# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

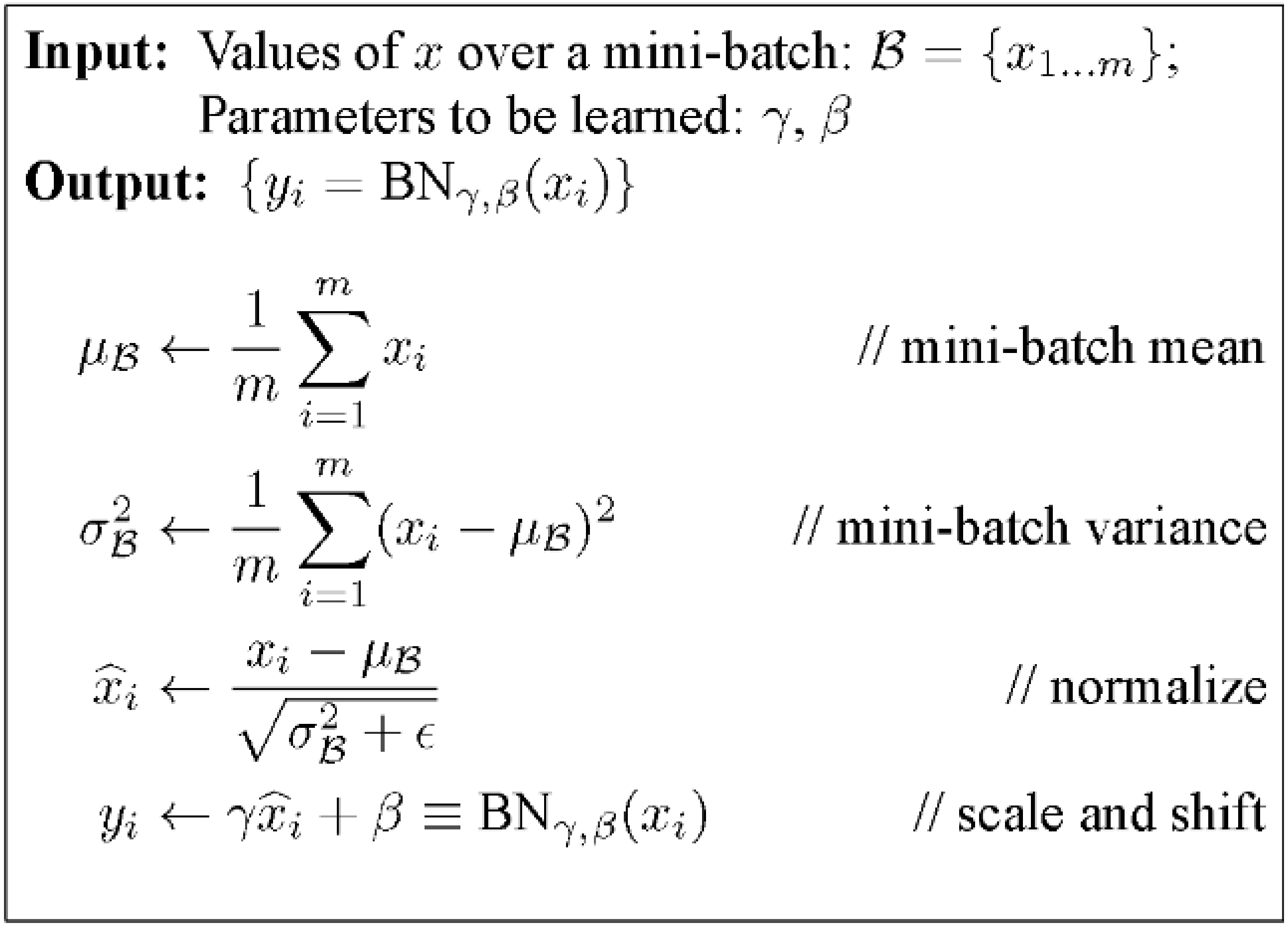
```
Results:
lr 4.977647e-04 reg 6.617402e-02 ws 2.693245e-02   val accuracy: 0.495000
lr 6.821125e-03 reg 1.175963e-03 ws 2.719198e-02   val accuracy: 0.525000
lr 2.316130e-04 reg 3.279283e-03 ws 1.225345e-02   val accuracy: 0.406000
lr 1.234230e-04 reg 7.870898e-03 ws 4.304900e-02   val accuracy: 0.383000
lr 6.390513e-04 reg 3.224054e-02 ws 2.544405e-02   val accuracy: 0.507000
lr 1.167326e-05 reg 3.299181e-03 ws 2.730571e-02   val accuracy: 0.225000
lr 2.348728e-04 reg 2.490779e-02 ws 2.387440e-02   val accuracy: 0.441000
lr 3.087040e-03 reg 1.911191e-03 ws 1.393565e-02   val accuracy: 0.520000
lr 2.784190e-04 reg 2.711998e-03 ws 1.626681e-02   val accuracy: 0.437000
lr 3.359292e-04 reg 2.771585e-03 ws 6.088676e-02   val accuracy: 0.363000
lr 4.447199e-04 reg 2.992993e-02 ws 2.800259e-02   val accuracy: 0.489000
lr 8.264621e-05 reg 4.494209e-03 ws 2.728778e-02   val accuracy: 0.398000
lr 8.998421e-03 reg 7.584525e-02 ws 1.320867e-02   val accuracy: 0.438000
lr 1.334652e-05 reg 7.992302e-02 ws 1.715106e-02   val accuracy: 0.127000
lr 1.608599e-04 reg 6.685497e-03 ws 1.948511e-02   val accuracy: 0.438000
best validation accuracy: 0.525000
best model:  <cs231n.solver.Solver object at 0x7f295e32ad30>
```

I made a few attempts before seeing these results. I was getting very low results when I selected regularization between 10 ^ 3 and 10 ^ -3. Likewise, I changed the learning rate. Finally, I was able to see an accuracy of around 50%.

11-

def batchnorm_forward(x, gamma, beta, bn_param):

```
        mode = bn_param['mode']
        eps = bn_param.get('eps', 1e-5)
        momentum = bn_param.get('momentum', 0.9)

        N, D = x.shape
        running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
        running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

        out, cache = None, None
        if mode == 'train':
#########################
        #step1: calculate mean
        mu = np.mean(x, axis=0)

        #step2: subtract mean vector of every trainings example
        xmu = x - mu

        #step3: following the lower branch - calculation denominator
        sqxmu = xmu ** 2

        #step4: calculate variance
        var = np.var(x, axis=0)

        #step5: add eps for numerical stability, then sqrt
        sqvar = np.sqrt(var + eps)

        #step6: invert sqrtwar
        invvar = 1./sqvar

        #step7: execute normalization
        xhat = xmu * invvar
        #step8: Nor the two transformation steps
        out = gamma * xhat + beta

        #store intermediate
        cache = (xhat,gamma,xmu,invvar,sqvar,var,eps)



        running_mean = momentum * running_mean + (1 - momentum) * mu
        running_var = momentum * running_var + (1 - momentum) * var
 elif mode == 'test':
        #normalize
      x_normalize = (x - running_mean) / np.sqrt(running_var + eps)
      out = gamma * x_normalize + beta
else:
      raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
bn_param['running_mean'] = running_mean
bn_param['running_var'] = running_var
return out, cache
```



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

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

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

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

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

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

We use the above formula when calculating batch normalization. Here;

x: data of shape

mu: mean

xmu = traning sample – mean

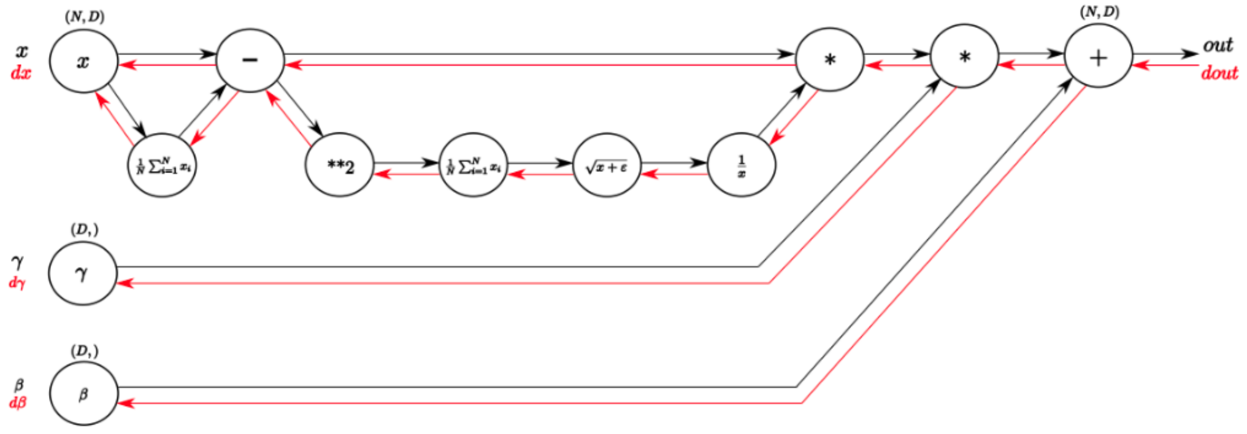
var: variance

invvar = inverse variance

gamma: scale parameter of shape

and at each timestep we need update the running averages for mean and variance using

an exponential decay based on the momentum parameter. So we calculate running_mean and running_var.



Backpropagation follows these steps in reverse order, as we are literally backpassing through the computational graph. We use the above graph when calculating batch normalization forward and backward function

```
def batchnorm_backward(dout, cache):
        dx, dgamma, dbeta = None, None, None
################
        N, D = dout.shape #get dout shape
        #unfold the variables stored in cache
        xhat, gamma, xmu, ivar, sqrtvar, var, eps = cache


        #step9
```

```
dbeta = np.sum(dout, axis=0)
dgammax = dout

#step8
dgamma = np.sum(dgammax*xhat, axis=0)
dxhat = dgammax * gamma

#step7
divar = np.sum(dxhat*xmu, axis=0)
dxmu1 = dxhat * ivar

#step6
dsqrtvar = -1. / (sqrtvar**2) * divar

#step5
dvar = 0.5 * 1. / np.sqrt(var+eps) * dsqrtvar

#step4
dsq = 1. / N * np.ones((N, D)) * dvar

#step3
dxmu2 = 2 * xmu * dsq

#step2
dx1 = dxmu1 + dxmu2
dmu = -1 * np.sum(dx1, axis=0)

#step1
dx2 = 1. / N * np.ones((N, D)) * dmu
```

```
    #step0
    dx = dx1 + dx2

#################
return dx, dgamma, dbeta
```

def batchnorm_backward_alt(dout, cache):

```
    dx, dgamma, dbeta = None, None, None
#################
    N, D = dout.shape #get shape

    xhat, gamma, xmu, invvar, sqrtvar, var, eps = cache

     dxhat = dout * gamma
     dx = 1.0/N * invvar * (N*dxhat - np.sum(dxhat, axis=0) - xhat*np.sum(dxhat*xhat,
axis=0))
    dbeta = np.sum(dout, axis=0)
    dgamma = np.sum(xhat*dout, axis=0)
#################
    return dx, dgamma, dbeta
```

We take a simpler step backprob by looking at the explanations in the code and graph above.

12-

When using batch normalization, store scale and shift parameters for the first layer in gamma1 and beta1; for the second layer use gamma2 and beta2, etc. Scale parameters should be initialized to ones and shift parameters should be initialized to zeros. So if we use batch normalization, we initialize gamma and beta params.

Not only for batch normalization, but to make it a bit more general if we come across it in the future, I am changing it if there is any normalization.

Additions at def __init__()

```
if self.normalization != None:#True(batchnorm/other norms)
    for i in range(self.num_layers-1):#gamma=scale params, beta = shift params
```

```
                    self.params['gamma'+str(i+1)] = np.ones(alllayer[i+1])#Scale parameters
should be initialized to ones
               self.params['beta' +str(i+1)] = np.zeros(alllayer[i+1])#Shift parameters should be
initialized to zeros.
```

Then at forward pass at loss function, In general we apply affine layer and than we use ReLU activation function. But now, we apply affine layer and than we use batch normalization and finally we use ReLU activation function. affine -> batch norm -> relu

Additions at def loss()

#additions forward pass

```
for i in range(self.num_layers-1):
        Wi = self.params['W'+str(i+1)]#W1, W2
        bi = self.params['b'+str(i+1)]#b1, b2

        if self.normalization == 'batchnorm': # affine -> batch norm -> relu
                gamma, beta = self.params['gamma'+str(i+1)], self.params['beta'+str(i+1)]
                fc_out, fc_cache = affine_forward(xi, Wi, bi)
                bn_out, bn_cache = batchnorm_forward(fc_out, gamma, beta, self.bn_params[i])
                xi, relu_cache = relu_forward(bn_out)
                caches[i+1] = (fc_cache, bn_cache, relu_cache)
        else:
                fc_out, fc_cache = affine_forward(xi, Wi, bi)
                xi, relu_cache = relu_forward(fc_out)
                caches[i+1] = (fc_cache, relu_cache)
# The last layer
      scores, last_cache = affine_forward(xi, self.params['W'+str(self.num_layers)],
self.params['b'+ str(self.num_layers)])
      caches[self.num_layers] = last_cache
```

Then at backward pass at loss function, our processing order must be drelu -> dbatchnorm -> daffine.

#additions backward pass

```
if self.normalization == 'batchnorm':
            # drelu -> dbatchnorm -> daffine
            fc_cache, bn_cache, relu_cache = caches[i]
            dbn_out = relu_backward(dout, relu_cache)
            dfc_out, grads['gamma'+str(i)], grads['beta'+str(i)] = 
batchnorm_backward(dbn_out, bn_cache)
            dout, grads['W'+str(i)], grads['b'+str(i)] = affine_backward(dfc_out, fc_cache)
```

Then we run forward and backward using batch normalization.

13-

Batch normalization: forward

```
Before batch normalization:
  means:  [ -2.3814598  -13.18038246   1.91780462]
  stds:   [27.18502186 34.21455511 37.68611762]

After batch normalization (gamma=1, beta=0)
  means:  [ 6.88338275e-17  7.82707232e-17 -8.04911693e-18]
  stds:   [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means:  [11. 12. 13.]
  stds:   [0.99999999 1.99999999 2.99999999]

After batch normalization (test-time):
  means:  [-0.03927354 -0.04349152 -0.10452688]
  stds:   [1.01531428 1.01238373 0.97819988]
```

Batch normalization: backward

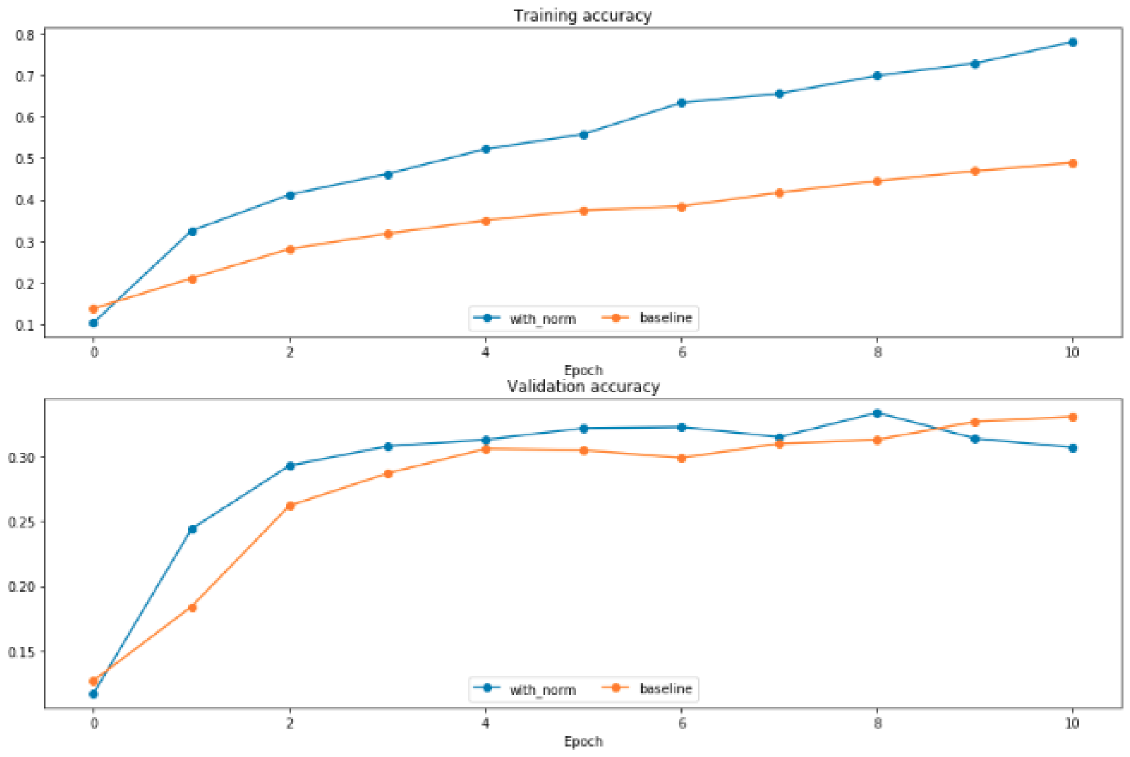
```
dx error:  1.6674604875341426e-09
dgamma error:  7.417225040694815e-13
dbeta error:  2.379446949959628e-12
```

Batch normalization: alternative backward

```
dx difference:  9.890497291190823e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 2.18x
```

Fully Connected Nets with Batch Normalization

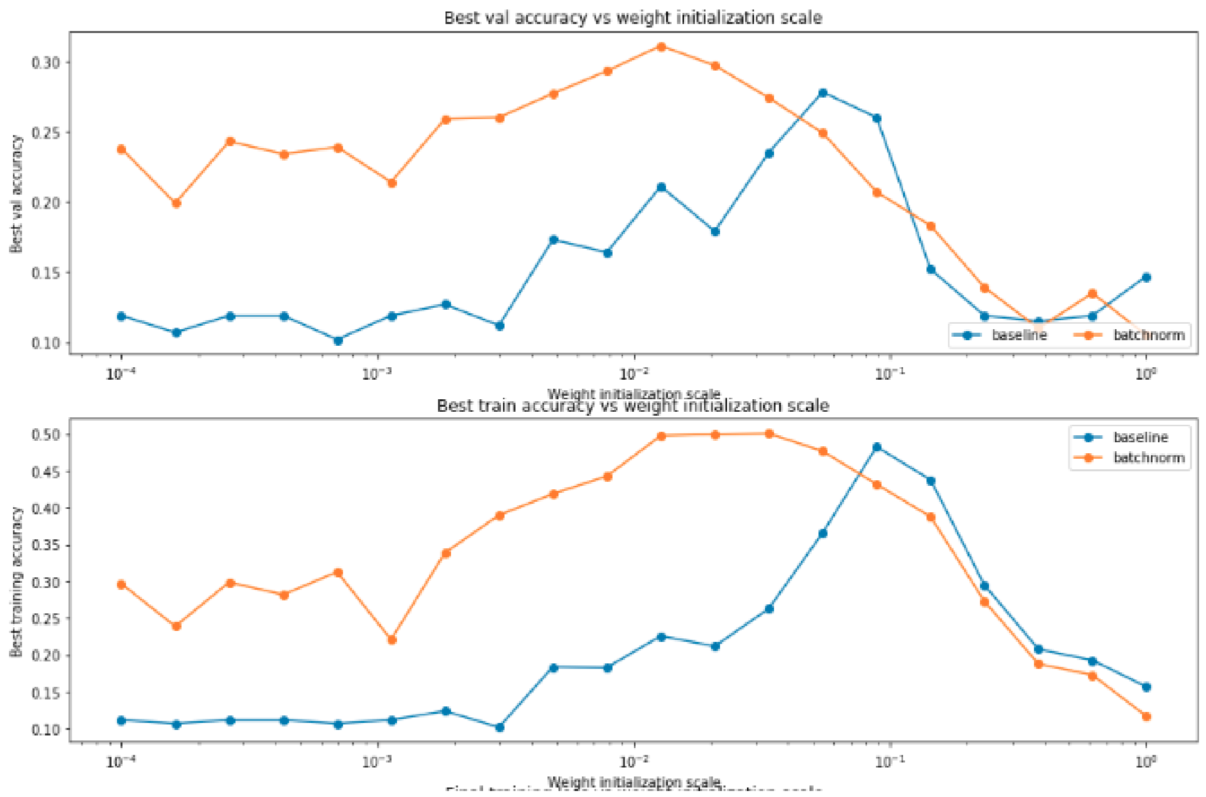
```
Running check with reg =  0
Initial loss:  2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 3.11e-06
W3 relative error: 4.05e-10
b1 relative error: 4.44e-08
b2 relative error: 2.22e-08
b3 relative error: 1.01e-10
beta1 relative error: 7.33e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 6.96e-09
gamma2 relative error: 2.41e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.28e-06
W3 relative error: 1.00e+00
b1 relative error: 5.55e-09
b2 relative error: 2.22e-08
b3 relative error: 2.10e-10
beta1 relative error: 6.65e-09
beta2 relative error: 3.39e-09
gamma1 relative error: 6.27e-09
gamma2 relative error: 5.28e-09
```

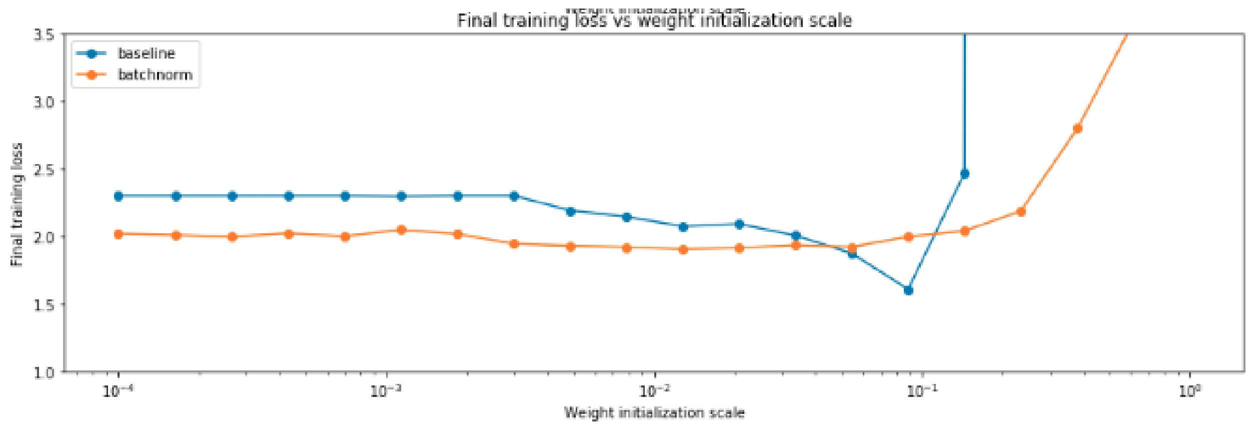


As can be seen, when using batch normalization, training accuracy reaches 0.8, it seems that the use of batch normalization in validation accuracy does not make much difference.

14-

We run batchnorm-weight initialization interaction test.





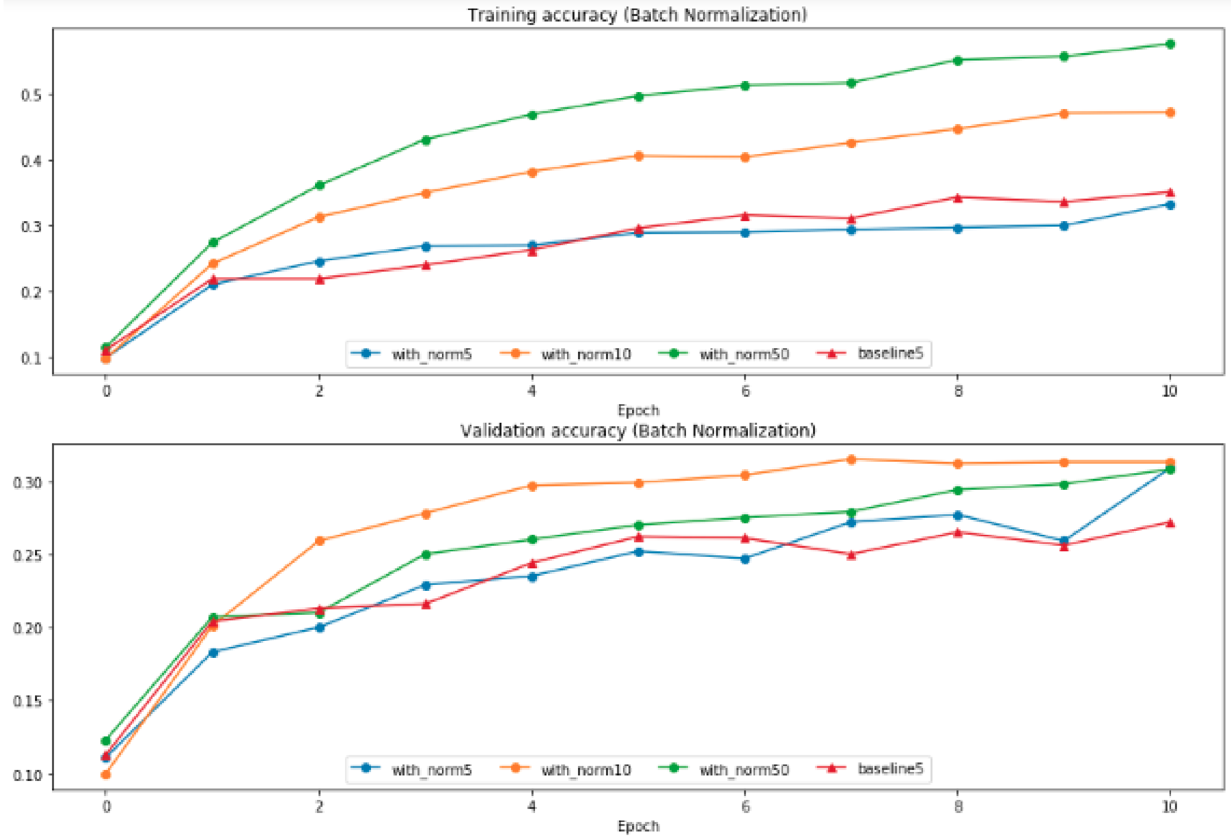
Inline Question 1:

Describe the results of this experiment. How does the scale of weight initialization affect models with/without batch normalization differently, and why?

Answer:

The first and second graph show "validation accuracy" and "train accuracy", respectively, with "weight initialization". Here we can see that the graph that emerges in both are similar. In the second graph, we can see the gradients that have vanishing in small initial weights. Here we have solved this problem using batchnorm. But this process has not changed the best train accuracy we can reach. In both the first graph and the second graph, we see that accuracy decreases when weight initiallize approaches 0. This situation appears more clearly in the first graph.
The third graph shows exploding gradients and is very evident for weight scale values greater than 1e-1 in the baseline model. However, the batchnorm model does not suffer from this problem.

15-



Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

Answer:

According to the results, we can see that the batch size affects directly the accuracy of batch normalization If we increase the size of batch, our batch normalization can have better results. I think the reason is exactly like the input layer. The samples will be closer to the sample for inner activations. Another situation I can extract from these graphics is that if the batch is too small for you, it may damage our model. In the first chart, norm5 is below the situation where we do not use batch normalization.

16-

Thank you for surprize.

17-

Dropout is an extremely effective, simple and recently introduced regularization technique. While training, dropout is implemented by only keeping a neuron active with some probability p or setting it to zero otherwise.

For the forward pass, we know that each neuron has a probability of being turned off by probability p. It is possible to model the application of Dropout, during training phase, by transforming the input as:

```
def dropout_forward(x, dropout_param):
     p, mode = dropout_param['p'], dropout_param['mode']
    if 'seed' in dropout_param:
        np.random.seed(dropout_param['seed'])

    mask = None
    out = None
    if mode == 'train':
##############
        mask = (np.random.random(x.shape) < p) / p
        out = x * mask
##############
    elif mode == 'test':
        out = x
    cache = (dropout_param, mask)
```

```
            out = out.astype(x.dtype, copy=False)


           return out, cache
```

Here,

x: Input data, of any shape

p: Dropout parameter. We keep each neuron output with probability p.

mode: 'test' or 'train'. If the mode is train, then perform dropout;

If prop drop > p, it is 1 and prop drop < p, it is 0. So we can use mask here.

```
mask = (np.random.random(x.shape) < p) / p
out = x * mask
```

```
def dropout_backward(dout, cache):
           dropout_param, mask = cache
           mode = dropout_param['mode']

           dx = None
           if mode == 'train':
###########
                      dx = dout * mask
###########
           elif mode == 'test':
                      dx = dout
           return dx
```

we calculate “out” using “out = x * mask” formula.

So at dropout backward function, we can calculate “dx” using “dx = dout * mask “ formula.

Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by p in the dropout layer? Why does that happen?

Answer:

in standard dropout during training you multiply each hidden neurons by a p random variable, so that "on average" each neuron x has value E [dropout_mask * x] = p E[x], so the network adapts to having each activations halved . Thus, at testing time you need to bring each activation to the same "range", by multiplying by p.
But, we can "fix" average value for each neuron in the network by multiplying it by 1/p during training time. This way, "typical range" of each neuron stays the same regardless of whether we use dropout or not, and we don't need any changes during the test time.

18-

Additions in FullyConnectedNet/loss functions

```
        #dropout
 if self.use_dropout:
        xi, d_cache = dropout_forward(xi, self.dropout_param)
        caches['dropout'+str(i+1)] = d_cache
```

Than, when we calculate loss, dropout backward except the last layer (before relu backward)

```
if self.use_dropout:
        out = dropout_backward(dout, caches['dropout'+str(i)])
```

19-

Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

Answer:

When we look at train accuracy in the first graphic, we see that the model is overfitting. with dropout the accuracies are smaller than without dropout. So we avoided overfitting with the dropout.
In the validation graphic, we can see that with dropout we get better results. look at here, we see that with dropout we are regularizing our model and we are reducing overfitting.

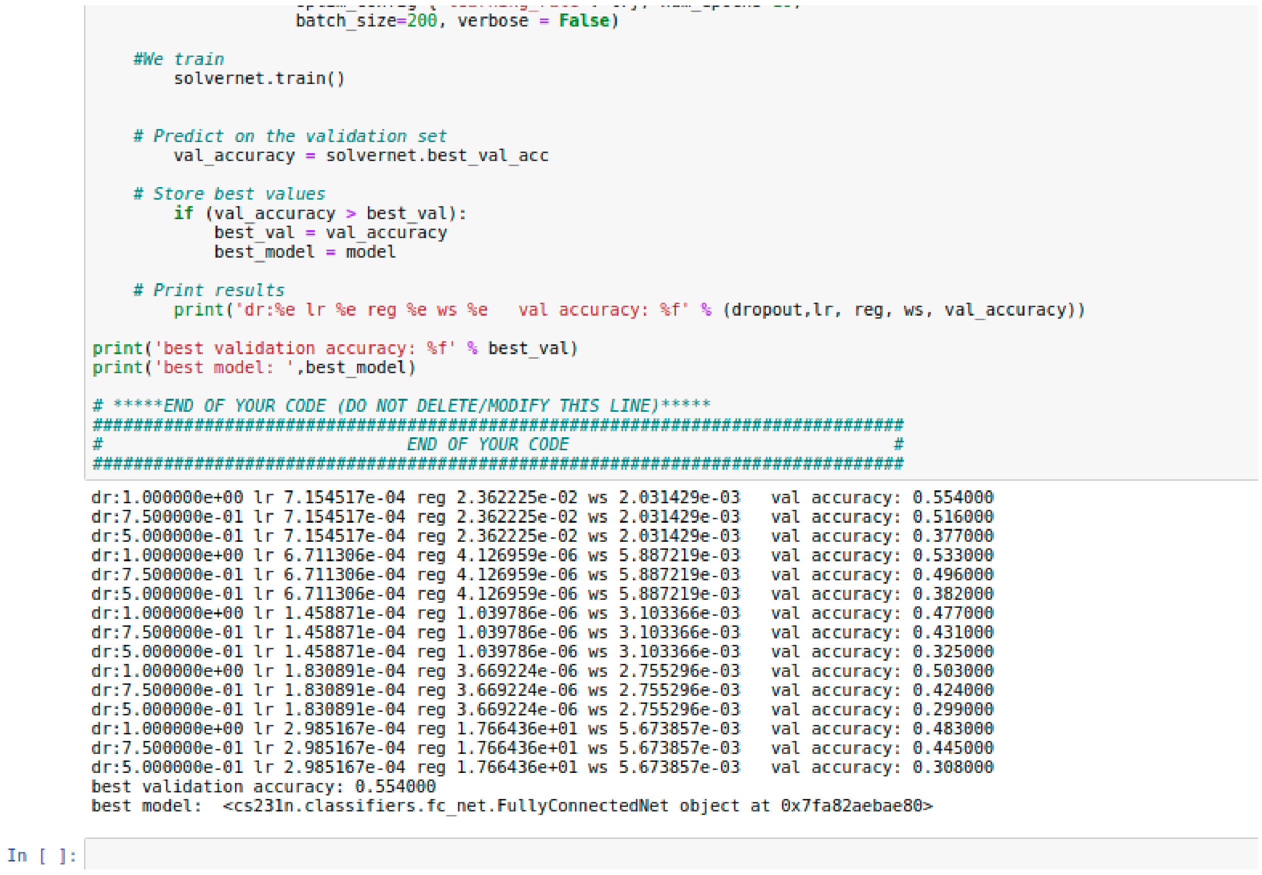
Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). If we are concerned about overfitting, how should we modify p (if at all) when we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

Answer:

Actually i think this makes no sense. Because if we decide to decrease the size of the hidden layers, we are not required to modify p because the number of neurons, which will be dropped out, will be proportional according to the size of the hidden layers.

20-



```
                    batch_size=200, verbose = False)

    #We train
        solvernet.train()


    # Predict on the validation set
        val_accuracy = solvernet.best_val_acc

    # Store best values
        if (val_accuracy > best_val):
            best_val = val_accuracy
            best_model = model

    # Print results
        print('dr:%e lr %e reg %e ws %e   val accuracy: %f' % (dropout,lr, reg, ws, val_accuracy))

print('best validation accuracy: %f' % best_val)
print('best model: ',best_model)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
dr:1.000000e+00 lr 7.154517e-04 reg 2.362225e-02 ws 2.031429e-03   val accuracy: 0.554000
dr:7.500000e-01 lr 7.154517e-04 reg 2.362225e-02 ws 2.031429e-03   val accuracy: 0.516000
dr:5.000000e-01 lr 7.154517e-04 reg 2.362225e-02 ws 2.031429e-03   val accuracy: 0.377000
dr:1.000000e+00 lr 6.711306e-04 reg 4.126959e-06 ws 5.887219e-03   val accuracy: 0.533000
dr:7.500000e-01 lr 6.711306e-04 reg 4.126959e-06 ws 5.887219e-03   val accuracy: 0.496000
dr:5.000000e-01 lr 6.711306e-04 reg 4.126959e-06 ws 5.887219e-03   val accuracy: 0.382000
dr:1.000000e+00 lr 1.458871e-04 reg 1.039786e-06 ws 3.103366e-03   val accuracy: 0.477000
dr:7.500000e-01 lr 1.458871e-04 reg 1.039786e-06 ws 3.103366e-03   val accuracy: 0.431000
dr:5.000000e-01 lr 1.458871e-04 reg 1.039786e-06 ws 3.103366e-03   val accuracy: 0.325000
dr:1.000000e+00 lr 1.830891e-04 reg 3.669224e-06 ws 2.755296e-03   val accuracy: 0.503000
dr:7.500000e-01 lr 1.830891e-04 reg 3.669224e-06 ws 2.755296e-03   val accuracy: 0.424000
dr:5.000000e-01 lr 1.830891e-04 reg 3.669224e-06 ws 2.755296e-03   val accuracy: 0.299000
dr:1.000000e+00 lr 2.985167e-04 reg 1.766436e+01 ws 5.673857e-03   val accuracy: 0.483000
dr:7.500000e-01 lr 2.985167e-04 reg 1.766436e+01 ws 5.673857e-03   val accuracy: 0.445000
dr:5.000000e-01 lr 2.985167e-04 reg 1.766436e+01 ws 5.673857e-03   val accuracy: 0.308000
best validation accuracy: 0.554000
best model:  <cs231n.classifiers.fc_net.FullyConnectedNet object at 0x7fa82aebae80>
```

```
In [ ]:
```

```
model = FullyConnectedNet(hidden_dims = hidden_sizes,
                          reg= reg,
                          weight_scale=ws,
                          dropout=dropout,
                          normalization='batchnorm')
```

I used the methods I learned so far here.

Dropout = [1, 0.75, 0.5]

normalization= Batch Normalization

Update Rule = Adams

As a result, I reached a validation accuracy of 55%.