BLG 506E COMPUTER VISION ASSIGNMENT 3

Two Layer Neural Networks

We will use two_layer_net_ipynb notebook and neural_net.py files for this assignment.

We initialize toy data and a toy model that we will use to develop your implementation.

def init_toy_model():

```
np.random.seed(0)
return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)
```

When we call to “TwoLayerNet(object)”, firstly enter into “__init__() “ function. So Initialize the model here. Weights are initialized to small random values and biases are initialized to zero.

def __init__(self, input_size, hidden_size, output_size, std=1e-4):

```
self.params = {}
self.params['W1'] = std * np.random.randn(input_size, hidden_size)
self.params['b1'] = np.zeros(hidden_size)
self.params['W2'] = std * np.random.randn(hidden_size, output_size)
self.params['b2'] = np.zeros(output_size)
```

TwoLayerNet has an input dimension of **N** and **D** features, a hidden layer dimension of **H**, and performs classification over **C** classes.

we initialized __init__(self, input_size, hidden_size, output_size, std=1e-4) function.

Here, Inputs:

- input_size: The dimension D of the input data.
- hidden_size: The number of neurons H in the hidden layer.
- output_size: The number of classes C.

and shape of parameters that we initialized:

- W1: First layer weights; has shape (D, H)
- b1: First layer biases; has shape (H,)
- W2: Second layer weights; has shape (H, C)
- b2: Second layer biases; has shape (C,)

Then, we open TwoLayerNet.loss function in cs231n/classifiers/neural_net.py and compute scores using weights and biases, so we perform **forward pass.**

```
def loss(self, X, y=None, reg=0.0):
    # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Compute the forward pass
    scores = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    fc1 = X.dot(W1) + b1     # fully connected-1
    af1 = np.maximum(0, fc1)  # use ReLU as activation function
    scores = af1.dot(W2) + b2 # fully connected-2
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # If the targets are not given then jump out, we're done
    if y is None:
       return scores

    # Compute the loss
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    #we calculate scores above
      #calculate softmax loss
    #N = X.shape[0]
    #P = e^s/Sum(e^s), s = f(xi,W), keep dimenson.
    P = np.exp(scores)/np.sum(np.exp(scores), axis=1, keepdims=True)

    L = -np.sum(np.log(P[np.arange(N), y]))#L =sum(-log(P))
```

 # Average loss

loss= L/N #X.shape[0]

 # add Regularization to loss

loss += reg * (np.sum(W2 * W2) + np.sum(W1 * W1))

L2 regularization, R(W1) + R(W2) = Sum(W1*W1) + Sum(W2*W2)

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

This loss function takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Firstly, we unpack variables from “params dictionary”. Then we perform forward pass.

Scores = f = W2.max(0,W1x + b1) + b2

Here, we use ReLu activation function.

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

fc1 = X.dot(W1) + b1 # fully connected-1

af1 = np.maximum(0, fc1) # use ReLU as activation function

scores = af1.dot(W2) + b2 # fully connected-2

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

```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720745909845e-08
```

After the forward pass we need to compute the loss. Here, we use Softmax Loss.

Softmax loss is:

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

P = e^s/Sum(e^s), s = f(xi,W)

L =sum(-log(P))

Then we calculate average loss and add regularization both W1 and W2.

R(W1) + R(W2) = Sum(W1*W1) + Sum(W2*W2)

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
     #N = X.shape[0]
      P = np.exp(scores)/np.sum(np.exp(scores), axis=1, keepdims=True)
      #P = e^s/Sum(e^s), s = f(xi,W), keep dimenson.
      L = -np.sum(np.log(P[np.arange(N), y]))#L =sum(-log(P))
   # Average loss
      loss= L/N #X.shape[0]

   # L2 regularization, R(W1) + R(W2) = Sum(W1*W1) + Sum(W2*W2)
      loss += reg * (np.sum(W2 * W2) + np.sum( W1 * W1 ))

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

We use **keepdims=True** in P = np.exp(scores)/np.sum(np.exp(scores), axis=1, keepdims=True) formule. Because, operands could not be broadcast together with shapes.

```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-93-770f3759e7c2> in <module>
----> 1 loss, _ = net.loss(X, y, reg=0.05)
      2 correct_loss = 1.30378789133
      3
      4 # should be very small, we get < 1e-12
      5 print('Difference between your loss and correct loss:')

~/Desktop/computervision/assignment1/cs231n/classifiers/neural_net.py in loss(self, X, y, reg)
    103         #N = X.shape[0]
    104
--> 105         P = np.exp(scores)/np.sum(np.exp(scores), axis=1, )#P = e^s/Sum(e^s), s = f(xi,W), keep dimenson.
    106         L = -np.sum(np.log(P[np.arange(N), y]))#L =sum(-log(P))
    107

ValueError: operands could not be broadcast together with shapes (5,3) (5,)
```

```
]: loss, _ = net.loss(X, y, reg=0.05)
   correct_loss = 1.30378789133

   # should be very small, we get < 1e-12
   print('Difference between your loss and correct loss:')
   print(np.sum(np.abs(loss - correct_loss)))

   Difference between your loss and correct loss:
   1.7985612998927536e-13
```

Then we perfoms **backward pass**. We implement the rest of the function. This will compute the gradient of the loss with respect to the variables W1, b1, W2, b2.

 # Backward pass: compute gradients

```
    grads = {}
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
      #gradient of softmax = Xi*(1 - pi)  if i = j, so Xi - Xi*pi and subtract Xi from i!=j case
      P[np.arange(N) ,y] -= 1
      P /= N #Avarage

  #Because of we calculate Backward pass, find firstly gradient of W2 and then gradient of W1
   #To calculate gradient of Weight , we use this formule = dW = X.T.dot(p) from softmax.py

    #W2 gradient
    grads['W2'] = af1.T.dot(P)

    #b2 gradient
    grads['b2'] = P.sum(axis=0)

    # W1 gradient
    dW1 = P.dot(W2.T)

#because of we use ReLU above as activation function, we also need to get gradient of this.

    dfc1 = dW1 * (fc1>0)
    grads['W1'] = X.T.dot(dfc1)
```

```
        # b1 gradient
        grads['b1'] = dfc1.sum(axis=0)


    # regularization gradient
        grads['W1'] += reg * 2 * W1
        grads['W2'] += reg * 2 * W2


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

Here firstly, gradient of softmax is Xi*(1 - pi) if i = j, so Xi - Xi*pi and subtract Xi from i!=j case. Then we calculate average P.

```
P[np.arange(N) ,y] -= 1
P /= N
```

Then, because of we calculate Backward pass, find firstly gradient of W2 and then gradient of W1. To calculate gradient of Weight , we use this formule = dW = X.T.dot(p) from softmax.py

W2 gradient:

```
grads['W2'] = af1.T.dot(P)
```

b2 gradient:

```
grads['b2'] = P.sum(axis=0)
```

W1 gradient

```
dW1 = P.dot(W2.T)
```

And then, because of we used ReLU above as activation function, we also need to get gradient of this.

```
dfc1 = dW1 * (fc1>0)
grads['W1'] = X.T.dot(dfc1)
```

and finally,

b1 gradient:

```
grads['b1'] = dfc1.sum(axis=0)
```

Then we calculate regularization of W1 and W2 gradient

```
  grads['W1'] += reg * 2 * W1
  grads['W2'] += reg * 2 * W2
```

Train the network

To train the network we will use stochastic gradient descent (SGD).

```
def train(self, X, y, X_val, y_val,
          learning_rate=1e-3, learning_rate_decay=0.95,
          reg=5e-6, num_iters=100,
          batch_size=200, verbose=False):

    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

    for it in range(num_iters):
      X_batch = None
      y_batch = None

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

      batch_indices = np.random.choice(num_train, batch_size)#divide num train into batch size.
      X_batch = X[batch_indices]
      y_batch = y[batch_indices]

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

      # Compute loss and gradients using the current minibatch
```

```
    loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
    loss_history.append(loss)

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

    #update the parameters
    self.params['W1'] -= learning_rate * grads['W1']
    self.params['b1'] -= learning_rate * grads['b1']
    self.params['W2'] -= learning_rate * grads['W2']
    self.params['b2'] -= learning_rate * grads['b2']



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

    if verbose and it % 100 == 0:
        print('iteration %d / %d: loss %f' % (it, num_iters, loss))

    # Every epoch, check train and val accuracy and decay learning rate.
    if it % iterations_per_epoch == 0:
        # Check accuracy
        train_acc = (self.predict(X_batch) == y_batch).mean()
        val_acc = (self.predict(X_val) == y_val).mean()
        train_acc_history.append(train_acc)
        val_acc_history.append(val_acc)

        # Decay learning rate
        learning_rate *= learning_rate_decay

return {
  'loss_history': loss_history,
  'train_acc_history': train_acc_history,
```

```
  'val_acc_history': val_acc_history,
}
```

İn this function, firstly, we create a random minibatch of training data and labels. We divide num train into batch size.

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


  batch_indices = np.random.choice(num_train, batch_size)
  X_batch = X[batch_indices]
  y_batch = y[batch_indices]


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

Then, we use the gradients in the grads dictionary to update the parameters of the network (stored in the dictionary self.params) using stochastic gradient descent.

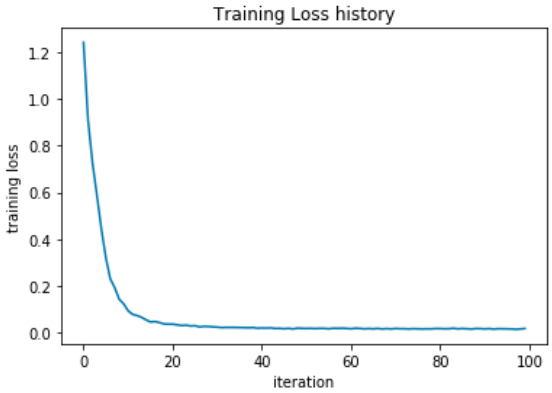
```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#update the parameters
 self.params['W1'] -= learning_rate * grads['W1']
 self.params['b1'] -= learning_rate * grads['b1']
 self.params['W2'] -= learning_rate * grads['W2']
  self.params['b2'] -= learning_rate * grads['b2']


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

Then we implement net.predict function. We use argmax function in scores to predict.

```
def predict(self, X):
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


    scores = self.loss(X) #get scores
    y_pred = np.argmax(scores, axis=1) #predict



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

```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.017149607938732048
```



Load the data

Now we load up our CIFAR-10 data so we can use it to train a classifier on a real dataset. We split the data into 49000 train, 1000 validation and 1000 test sets.

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Train a network

To train our network, we use Stochastic Gradient Descent (SGD).In addition, we adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we reduce the learning rate by multiplying it by a decay rate.

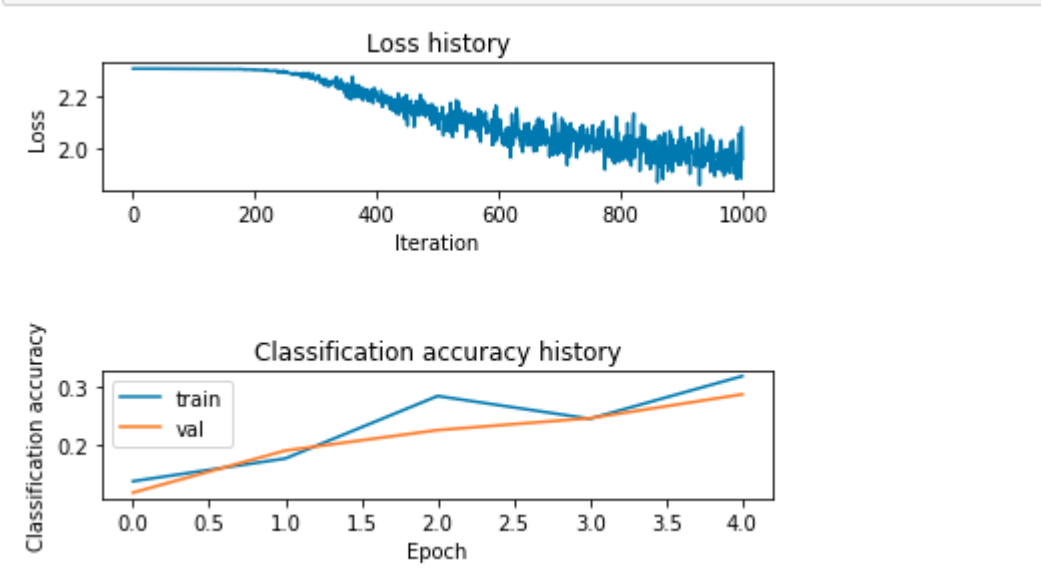
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

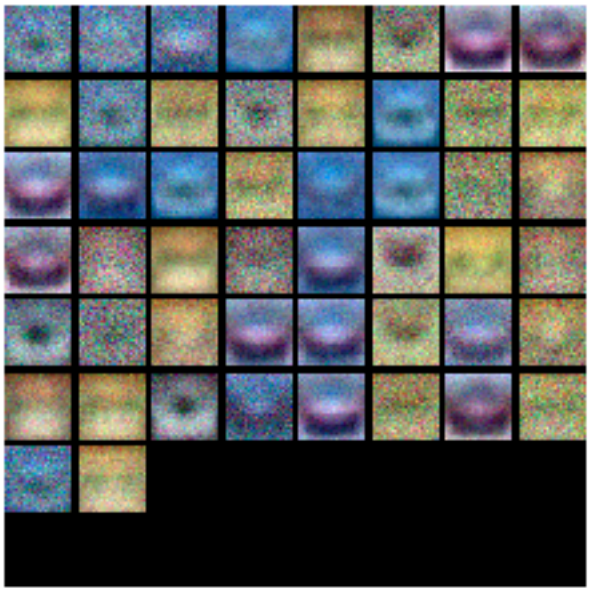
Debug the training

Now, we find that validation accuracy is 0.287 with default parameters. But this isn't very good. So we need optimization.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.



Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.



Tune your hyperparameters

Inline Question:
Explain your hyperparameter tuning process below

Answer: We tune 3 parameters here: regularization, learning rate and hidden layer. Here we want to find the best model in this network by changing these 3 parameters.

I will get 50, 100, 150, 200 values as hidden size.
I will use values between 10 ^ -4 and 10 ^ 0 as the regularization strength.
I will use values between 10 ^ -3 and 10 ^ -4 as the Learning Rate.
These values are my initial values for now. If validation accuracy does not come out as expected, I will try to optimize my network by changing these values.

Here, for each hidden size value, I will train a model by choosing randomly in the range in which I set the learning rate and regularization strength values in each iteration.

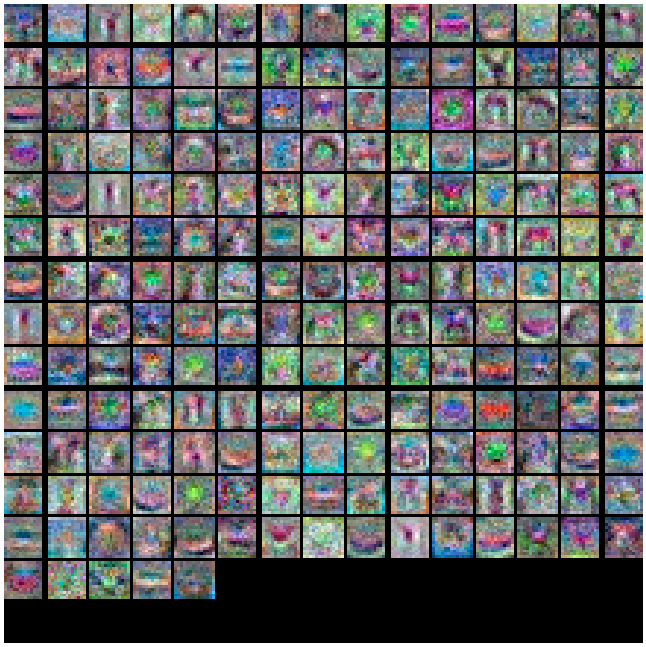
If I cannot find a validation accuracy of around 52%, I will retrain by changing these values.

```
best_net = None # store the best model into this
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
best_val = -1
input_size = 32*32*3
num_classes = 10
#We tune hyperparameters(hidden layer)
hidden_sizes = [50, 100, 150, 200]

max_count = 10
for hidden_size in hidden_sizes:
    for count in range(max_count):
        #We tune hyperparameters(regularization, learning rate)
        reg = 10**np.random.uniform(-4,0)
        lr = 10**np.random.uniform(-3,-4)
        #We create Two Layer Neural Network and we initialize Two Neural Network
        neural_net = TwoLayerNet(input_size,hidden_size, num_classes)

        #We train
        neural_net.train(X_train, y_train, X_val, y_val, num_iters=2000,
batch_size=200,learning_rate=lr,reg=reg, verbose=False)

        # We predict on the training set
        train_accuracy = (neural_net.predict(X_train) == y_train).mean()

        # Predict on the validation set
        val_accuracy = (neural_net.predict(X_val) == y_val).mean()
```

```
# Store best values
    if (val_accuracy > best_val):
        best_val = val_accuracy
        best_net = neural_net

    # Print results
    print('lr %e reg %e hid %d  train accuracy: %f val accuracy: %f' % (lr, reg, hidden_size,
train_accuracy, val_accuracy))

print('best validation accuracy achieved: %f' % best_val)
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
lr 3.848167e-04 reg 4.824614e-01 hid 50  train accuracy: 0.474265 val accuracy: 0.470000
lr 8.018790e-04 reg 1.242590e-04 hid 50  train accuracy: 0.522776 val accuracy: 0.482000
lr 1.077168e-04 reg 8.620255e-04 hid 50  train accuracy: 0.363204 val accuracy: 0.369000
lr 2.331425e-04 reg 1.748696e-01 hid 50  train accuracy: 0.433327 val accuracy: 0.445000
lr 2.843052e-04 reg 1.448861e-01 hid 50  train accuracy: 0.454020 val accuracy: 0.459000
lr 1.081102e-04 reg 5.361667e-01 hid 50  train accuracy: 0.367163 val accuracy: 0.379000
lr 1.496629e-04 reg 1.704625e-02 hid 50  train accuracy: 0.394592 val accuracy: 0.396000
lr 8.730414e-04 reg 3.829889e-01 hid 50  train accuracy: 0.514490 val accuracy: 0.496000
lr 3.057384e-04 reg 1.897822e-04 hid 50  train accuracy: 0.465469 val accuracy: 0.449000
lr 3.404794e-04 reg 5.802378e-03 hid 50  train accuracy: 0.475163 val accuracy: 0.454000
lr 1.634174e-04 reg 1.946045e-03 hid 100  train accuracy: 0.415796 val accuracy: 0.421000
lr 2.037616e-04 reg 3.535235e-04 hid 100  train accuracy: 0.440653 val accuracy: 0.455000
lr 1.542845e-04 reg 6.193092e-02 hid 100  train accuracy: 0.406959 val accuracy: 0.422000
lr 7.588008e-04 reg 7.619509e-04 hid 100  train accuracy: 0.542959 val accuracy: 0.499000
lr 2.775591e-04 reg 8.884747e-04 hid 100  train accuracy: 0.467796 val accuracy: 0.469000
lr 1.830748e-04 reg 3.427475e-02 hid 100  train accuracy: 0.428327 val accuracy: 0.428000
lr 1.285512e-04 reg 9.005808e-01 hid 100  train accuracy: 0.382816 val accuracy: 0.375000
lr 5.238973e-04 reg 3.502791e-02 hid 100  train accuracy: 0.518490 val accuracy: 0.486000
lr 1.345757e-04 reg 6.190169e-04 hid 100  train accuracy: 0.394469 val accuracy: 0.398000
lr 7.481534e-04 reg 4.054743e-01 hid 100  train accuracy: 0.523082 val accuracy: 0.486000
lr 2.745482e-04 reg 8.089669e-02 hid 150  train accuracy: 0.468939 val accuracy: 0.456000
lr 1.521307e-04 reg 1.028766e-01 hid 150  train accuracy: 0.409163 val accuracy: 0.407000
lr 1.639119e-04 reg 9.104051e-04 hid 150  train accuracy: 0.418918 val accuracy: 0.430000
lr 1.343684e-04 reg 7.572326e-01 hid 150  train accuracy: 0.389959 val accuracy: 0.397000
lr 3.908987e-04 reg 3.513112e-04 hid 150  train accuracy: 0.497347 val accuracy: 0.485000
lr 3.461086e-04 reg 1.030083e-04 hid 150  train accuracy: 0.488184 val accuracy: 0.471000
lr 4.564214e-04 reg 3.622317e-04 hid 150  train accuracy: 0.516184 val accuracy: 0.495000
lr 2.101464e-04 reg 1.961367e-03 hid 150  train accuracy: 0.443939 val accuracy: 0.448000
lr 3.912638e-04 reg 6.957723e-03 hid 150  train accuracy: 0.502857 val accuracy: 0.482000
lr 7.176323e-04 reg 1.304942e-01 hid 150  train accuracy: 0.545857 val accuracy: 0.503000
lr 1.282287e-04 reg 3.123586e-03 hid 200  train accuracy: 0.394245 val accuracy: 0.390000
lr 1.393362e-04 reg 4.569525e-01 hid 200  train accuracy: 0.398653 val accuracy: 0.395000
lr 3.488434e-04 reg 3.151363e-04 hid 200  train accuracy: 0.492429 val accuracy: 0.467000
lr 8.095502e-04 reg 6.688271e-04 hid 200  train accuracy: 0.570816 val accuracy: 0.522000
lr 6.796238e-04 reg 3.499068e-04 hid 200  train accuracy: 0.558204 val accuracy: 0.524000
lr 1.515488e-04 reg 3.138842e-01 hid 200  train accuracy: 0.408733 val accuracy: 0.418000
lr 1.141394e-04 reg 7.475557e-04 hid 200  train accuracy: 0.384531 val accuracy: 0.380000
lr 2.481795e-04 reg 9.194853e-02 hid 200  train accuracy: 0.461122 val accuracy: 0.466000
lr 6.695480e-04 reg 9.581665e-04 hid 200  train accuracy: 0.550061 val accuracy: 0.511000
lr 1.419574e-04 reg 5.578850e-02 hid 200  train accuracy: 0.403510 val accuracy: 0.398000
best validation accuracy achieved: 0.524000
```

We found the en validation accuracy as 0.524000. This looks good for now. Then we visualize the weights of the best network.



Finally, we evaluate our final trained network on the test set;

```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.517
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1.Train on a larger dataset.

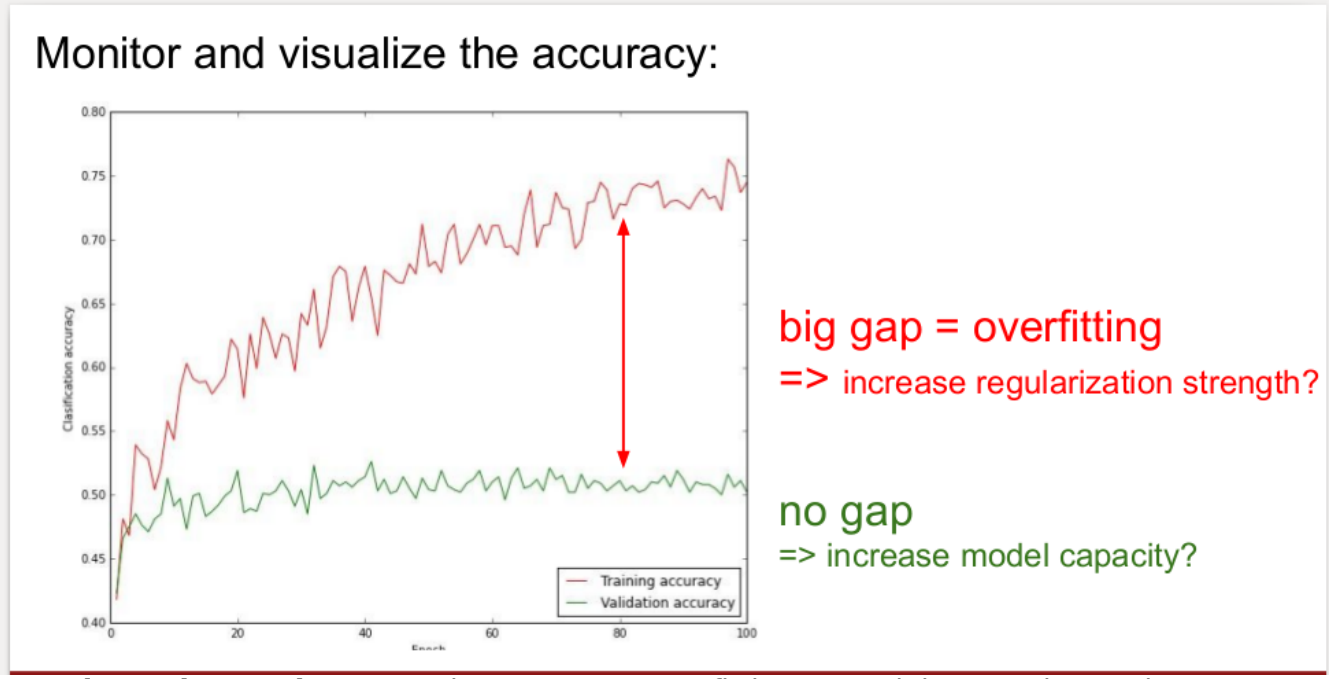
2.Add more hidden units.

3.Increase the regularization strength.

4.None of the above.

Answer:1 and 3

Explanation:



1. **Train o a larger datasets**:Big gap means overfitting. So training on a larger dataset prevents overfitting.[TRUE]

2. **Add more hidden units**:Increasing the number of hidden layers will cause accuracy in the test set to decrease. It will cause your network to overfit to the training set, that is, it will learn the training data, but it won't be able to generalize to new unseen data.[FALSE]

3: **Increase the regularization strength:** As we can see above, increasing the regularization strength prevents overfitting because it reduces the complexity of the model[TRUE]