```
[info] ROOT=/home/kim1484/assignment1_colab/assignment1
[info] DATA_DIR=/home/kim1484/assignment1_colab/assignment1/cs231n/datasets
[ready] 환경 준비 완료.
```

Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks using a modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```python
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```python
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

```
[ready] Visualization & autoreload settings complete.
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

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

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

```
dout shape: (10, 5)
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

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

$Your Answer:$ 1, 2
Sigmoid: About the sigmoide if the input is too large or small the gradient will get zero (or close to zero).
ReLU: About ReLU, if the input is negative the result will be zero. Which means the gradient will be also zero.
Leaky ReLU: Always have a small non-zero gradient(α) for negative inputs, which avoids zero gradient flow during backpropagation

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

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

Loss layers: Softmax

Now implement the loss and gradient for softmax in the `softmax_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py`. Other loss functions (e.g. `svm_loss`) can also be implemented in a modular way, however, it is not required for this assignment.

You can make sure that the implementations are correct by running the following:

```
Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.483503037636722e-09
```

Two-layer network

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. Read through it to make sure you understand the API. You can run the cell below to test your implementation.

```
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: 3.31e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

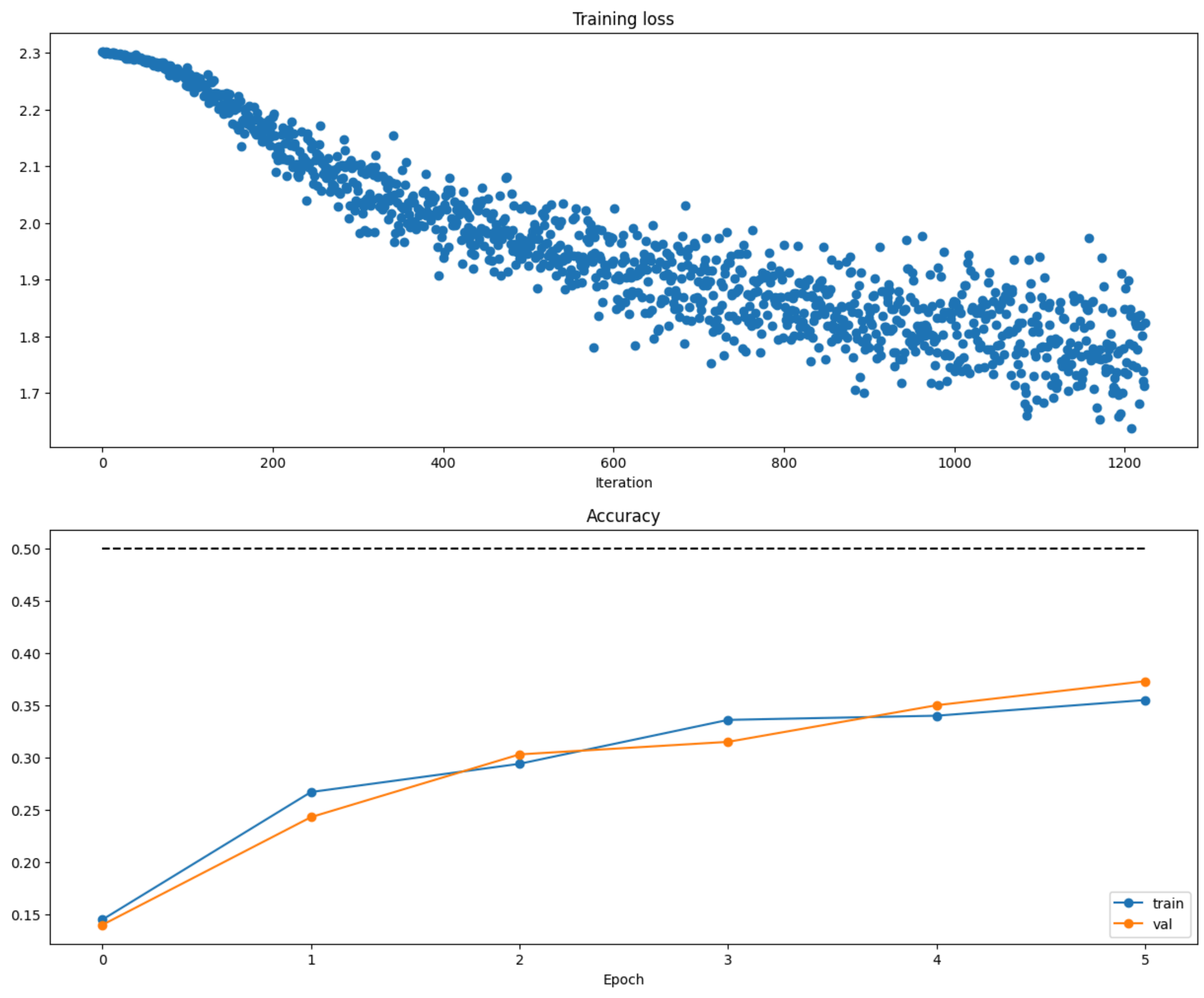
```
(Iteration 1 / 1225) loss: 2.301725
(Epoch 0 / 5) train acc: 0.145000; val_acc: 0.140000
(Iteration 101 / 1225) loss: 2.241923
(Iteration 201 / 1225) loss: 2.187425
(Epoch 1 / 5) train acc: 0.267000; val_acc: 0.243000
(Iteration 301 / 1225) loss: 2.056790
(Iteration 401 / 1225) loss: 1.937978
(Epoch 2 / 5) train acc: 0.294000; val_acc: 0.303000
(Iteration 501 / 1225) loss: 1.924555
(Iteration 601 / 1225) loss: 1.933743
(Iteration 701 / 1225) loss: 1.832777
(Epoch 3 / 5) train acc: 0.336000; val_acc: 0.315000
(Iteration 801 / 1225) loss: 1.960827
(Iteration 901 / 1225) loss: 1.832752
(Epoch 4 / 5) train acc: 0.340000; val_acc: 0.350000
(Iteration 1001 / 1225) loss: 1.739182
(Iteration 1101 / 1225) loss: 1.940517
(Iteration 1201 / 1225) loss: 1.848443
(Epoch 5 / 5) train acc: 0.355000; val_acc: 0.373000
```

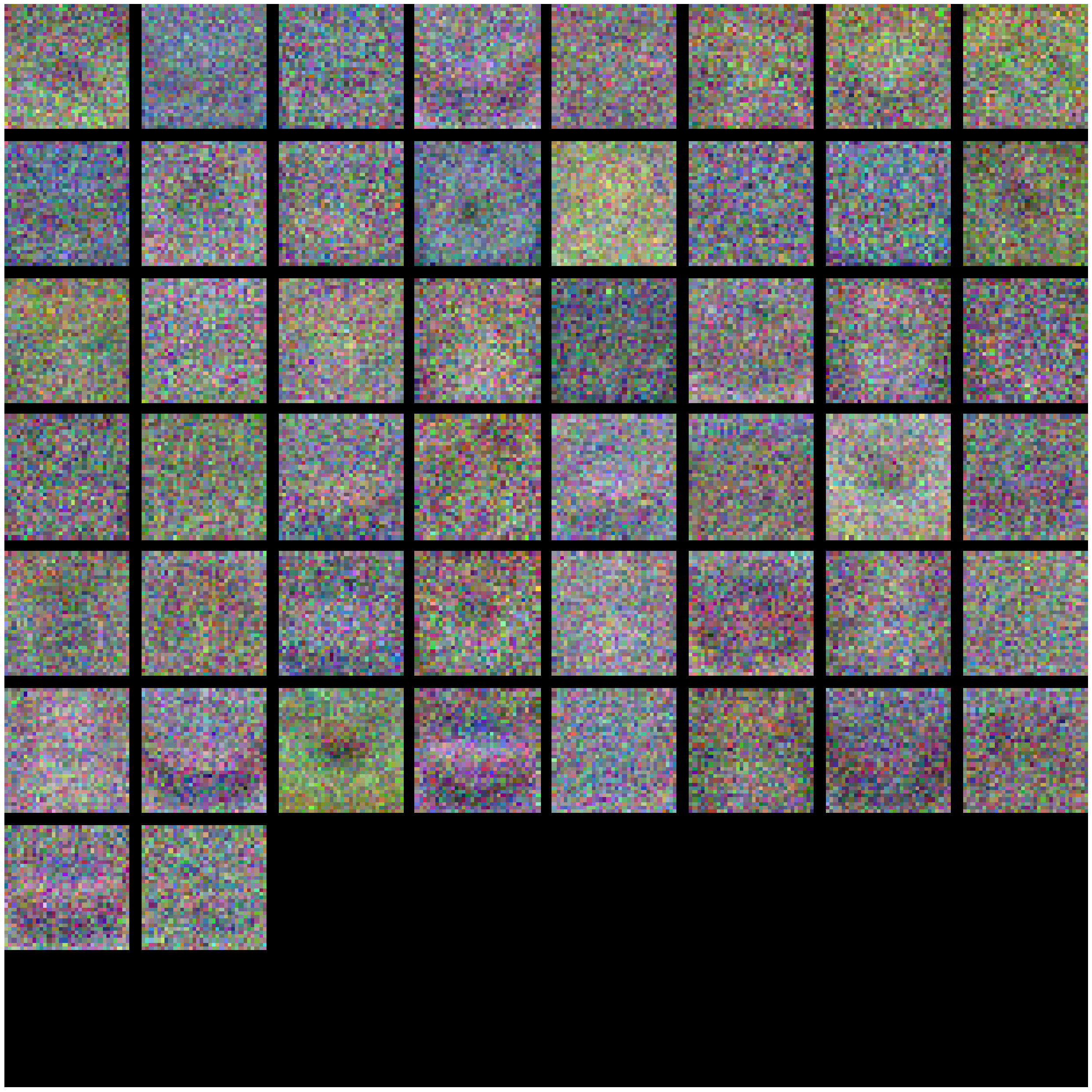
Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

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

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

```
==============================================================
hidden size: 50, learning rate: 0.0001, lr decay: 0.75
==============================================================
(Iteration 1 / 3675) loss: 2.302208
(Epoch 0 / 15) train acc: 0.089000; val_acc: 0.103000
(Iteration 101 / 3675) loss: 2.269872
(Iteration 201 / 3675) loss: 2.180729
(Epoch 1 / 15) train acc: 0.225000; val_acc: 0.243000
(Iteration 301 / 3675) loss: 2.139334
(Iteration 401 / 3675) loss: 2.055889
(Epoch 2 / 15) train acc: 0.292000; val_acc: 0.291000
(Iteration 501 / 3675) loss: 2.027681
(Iteration 601 / 3675) loss: 2.000881
(Iteration 701 / 3675) loss: 1.865790
(Epoch 3 / 15) train acc: 0.299000; val_acc: 0.311000
(Iteration 801 / 3675) loss: 2.005687
(Iteration 901 / 3675) loss: 1.927090
(Epoch 4 / 15) train acc: 0.327000; val_acc: 0.325000
(Iteration 1001 / 3675) loss: 1.814586
(Iteration 1101 / 3675) loss: 1.919126
(Iteration 1201 / 3675) loss: 1.914328
(Epoch 5 / 15) train acc: 0.315000; val_acc: 0.327000
(Iteration 1301 / 3675) loss: 1.823837
(Iteration 1401 / 3675) loss: 1.949548
(Epoch 6 / 15) train acc: 0.292000; val_acc: 0.335000
(Iteration 1501 / 3675) loss: 1.867285
(Iteration 1601 / 3675) loss: 1.902650
(Iteration 1701 / 3675) loss: 1.849380
(Epoch 7 / 15) train acc: 0.330000; val_acc: 0.345000
(Iteration 1801 / 3675) loss: 1.920131
(Iteration 1901 / 3675) loss: 1.899820
(Epoch 8 / 15) train acc: 0.351000; val_acc: 0.352000
(Iteration 2001 / 3675) loss: 1.814006
(Iteration 2101 / 3675) loss: 1.946156
(Iteration 2201 / 3675) loss: 1.786926
(Epoch 9 / 15) train acc: 0.380000; val_acc: 0.353000
(Iteration 2301 / 3675) loss: 1.788916
(Iteration 2401 / 3675) loss: 1.863340
(Epoch 10 / 15) train acc: 0.353000; val_acc: 0.357000
(Iteration 2501 / 3675) loss: 1.810877
(Iteration 2601 / 3675) loss: 1.833393
(Epoch 11 / 15) train acc: 0.349000; val_acc: 0.358000
(Iteration 2701 / 3675) loss: 1.727624
(Iteration 2801 / 3675) loss: 1.838444
(Iteration 2901 / 3675) loss: 1.770730
(Epoch 12 / 15) train acc: 0.348000; val_acc: 0.360000
(Iteration 3001 / 3675) loss: 1.900193
(Iteration 3101 / 3675) loss: 1.717027
(Epoch 13 / 15) train acc: 0.333000; val_acc: 0.357000
(Iteration 3201 / 3675) loss: 1.829480
(Iteration 3301 / 3675) loss: 1.702334
(Iteration 3401 / 3675) loss: 1.772162
(Epoch 14 / 15) train acc: 0.364000; val_acc: 0.360000
(Iteration 3501 / 3675) loss: 1.861890
(Iteration 3601 / 3675) loss: 1.749665
```

```
(Epoch 15 / 15) train acc: 0.363000; val_acc: 0.359000
==============================================================
hidden size: 75, learning rate: 0.0003, lr decay: 0.8
==============================================================
(Iteration 1 / 3675) loss: 2.302020
(Epoch 0 / 15) train acc: 0.127000; val_acc: 0.122000
(Iteration 101 / 3675) loss: 2.094417
(Iteration 201 / 3675) loss: 1.877332
(Epoch 1 / 15) train acc: 0.311000; val_acc: 0.354000
(Iteration 301 / 3675) loss: 1.798343
(Iteration 401 / 3675) loss: 1.729738
(Epoch 2 / 15) train acc: 0.387000; val_acc: 0.398000
(Iteration 501 / 3675) loss: 1.778323
(Iteration 601 / 3675) loss: 1.652451
(Iteration 701 / 3675) loss: 1.700581
(Epoch 3 / 15) train acc: 0.401000; val_acc: 0.432000
(Iteration 801 / 3675) loss: 1.727696
(Iteration 901 / 3675) loss: 1.687392
(Epoch 4 / 15) train acc: 0.468000; val_acc: 0.448000
(Iteration 1001 / 3675) loss: 1.582012
(Iteration 1101 / 3675) loss: 1.622367
(Iteration 1201 / 3675) loss: 1.649013
(Epoch 5 / 15) train acc: 0.424000; val_acc: 0.446000
(Iteration 1301 / 3675) loss: 1.665999
(Iteration 1401 / 3675) loss: 1.473240
(Epoch 6 / 15) train acc: 0.434000; val_acc: 0.463000
(Iteration 1501 / 3675) loss: 1.600735
(Iteration 1601 / 3675) loss: 1.621141
(Iteration 1701 / 3675) loss: 1.568323
(Epoch 7 / 15) train acc: 0.433000; val_acc: 0.465000
(Iteration 1801 / 3675) loss: 1.441128
(Iteration 1901 / 3675) loss: 1.675594
(Epoch 8 / 15) train acc: 0.496000; val_acc: 0.458000
(Iteration 2001 / 3675) loss: 1.550722
(Iteration 2101 / 3675) loss: 1.475482
(Iteration 2201 / 3675) loss: 1.533579
(Epoch 9 / 15) train acc: 0.442000; val_acc: 0.466000
(Iteration 2301 / 3675) loss: 1.547075
(Iteration 2401 / 3675) loss: 1.466843
(Epoch 10 / 15) train acc: 0.466000; val_acc: 0.465000
(Iteration 2501 / 3675) loss: 1.417340
(Iteration 2601 / 3675) loss: 1.573513
(Epoch 11 / 15) train acc: 0.449000; val_acc: 0.462000
(Iteration 2701 / 3675) loss: 1.457753
(Iteration 2801 / 3675) loss: 1.491303
(Iteration 2901 / 3675) loss: 1.535973
(Epoch 12 / 15) train acc: 0.448000; val_acc: 0.467000
(Iteration 3001 / 3675) loss: 1.542745
(Iteration 3101 / 3675) loss: 1.580296
(Epoch 13 / 15) train acc: 0.447000; val_acc: 0.467000
(Iteration 3201 / 3675) loss: 1.640223
(Iteration 3301 / 3675) loss: 1.594003
(Iteration 3401 / 3675) loss: 1.440307
(Epoch 14 / 15) train acc: 0.465000; val_acc: 0.469000
(Iteration 3501 / 3675) loss: 1.404369
(Iteration 3601 / 3675) loss: 1.470520
```

```
(Epoch 15 / 15) train acc: 0.481000; val_acc: 0.468000
==============================================================
hidden size: 100, learning rate: 0.0005, lr decay: 0.85
==============================================================
(Iteration 1 / 3675) loss: 2.305878
(Epoch 0 / 15) train acc: 0.103000; val_acc: 0.125000
(Iteration 101 / 3675) loss: 1.942257
(Iteration 201 / 3675) loss: 1.827047
(Epoch 1 / 15) train acc: 0.374000; val_acc: 0.405000
(Iteration 301 / 3675) loss: 1.703828
(Iteration 401 / 3675) loss: 1.634592
(Epoch 2 / 15) train acc: 0.447000; val_acc: 0.435000
(Iteration 501 / 3675) loss: 1.631321
(Iteration 601 / 3675) loss: 1.662849
(Iteration 701 / 3675) loss: 1.470808
(Epoch 3 / 15) train acc: 0.428000; val_acc: 0.467000
(Iteration 801 / 3675) loss: 1.641560
(Iteration 901 / 3675) loss: 1.480923
(Epoch 4 / 15) train acc: 0.470000; val_acc: 0.470000
(Iteration 1001 / 3675) loss: 1.531020
(Iteration 1101 / 3675) loss: 1.577232
(Iteration 1201 / 3675) loss: 1.436203
(Epoch 5 / 15) train acc: 0.494000; val_acc: 0.474000
(Iteration 1301 / 3675) loss: 1.378047
(Iteration 1401 / 3675) loss: 1.346023
(Epoch 6 / 15) train acc: 0.500000; val_acc: 0.488000
(Iteration 1501 / 3675) loss: 1.441503
(Iteration 1601 / 3675) loss: 1.304212
(Iteration 1701 / 3675) loss: 1.370535
(Epoch 7 / 15) train acc: 0.511000; val_acc: 0.500000
(Iteration 1801 / 3675) loss: 1.401865
(Iteration 1901 / 3675) loss: 1.447340
(Epoch 8 / 15) train acc: 0.514000; val_acc: 0.503000
(Iteration 2001 / 3675) loss: 1.316364
(Iteration 2101 / 3675) loss: 1.420318
(Iteration 2201 / 3675) loss: 1.514687
(Epoch 9 / 15) train acc: 0.535000; val_acc: 0.499000
(Iteration 2301 / 3675) loss: 1.331169
(Iteration 2401 / 3675) loss: 1.238247
(Epoch 10 / 15) train acc: 0.551000; val_acc: 0.500000
(Iteration 2501 / 3675) loss: 1.337534
(Iteration 2601 / 3675) loss: 1.319425
(Epoch 11 / 15) train acc: 0.525000; val_acc: 0.493000
(Iteration 2701 / 3675) loss: 1.365279
(Iteration 2801 / 3675) loss: 1.389755
(Iteration 2901 / 3675) loss: 1.358643
(Epoch 12 / 15) train acc: 0.536000; val_acc: 0.499000
(Iteration 3001 / 3675) loss: 1.464928
(Iteration 3101 / 3675) loss: 1.218806
(Epoch 13 / 15) train acc: 0.552000; val_acc: 0.505000
(Iteration 3201 / 3675) loss: 1.260621
(Iteration 3301 / 3675) loss: 1.240901
(Iteration 3401 / 3675) loss: 1.371293
(Epoch 14 / 15) train acc: 0.547000; val_acc: 0.504000
(Iteration 3501 / 3675) loss: 1.256126
(Iteration 3601 / 3675) loss: 1.435832
```

```
(Epoch 15 / 15) train acc: 0.553000; val_acc: 0.509000
==============================================================
hidden size: 125, learning rate: 0.0007, lr decay: 0.9
==============================================================
(Iteration 1 / 3675) loss: 2.298223
(Epoch 0 / 15) train acc: 0.130000; val_acc: 0.135000
(Iteration 101 / 3675) loss: 1.786951
(Iteration 201 / 3675) loss: 1.693566
(Epoch 1 / 15) train acc: 0.413000; val_acc: 0.407000
(Iteration 301 / 3675) loss: 1.629973
(Iteration 401 / 3675) loss: 1.670703
(Epoch 2 / 15) train acc: 0.461000; val_acc: 0.456000
(Iteration 501 / 3675) loss: 1.619464
(Iteration 601 / 3675) loss: 1.499812
(Iteration 701 / 3675) loss: 1.530245
(Epoch 3 / 15) train acc: 0.467000; val_acc: 0.460000
(Iteration 801 / 3675) loss: 1.490324
(Iteration 901 / 3675) loss: 1.451270
(Epoch 4 / 15) train acc: 0.530000; val_acc: 0.479000
(Iteration 1001 / 3675) loss: 1.372859
(Iteration 1101 / 3675) loss: 1.370228
(Iteration 1201 / 3675) loss: 1.320328
(Epoch 5 / 15) train acc: 0.530000; val_acc: 0.489000
(Iteration 1301 / 3675) loss: 1.174534
(Iteration 1401 / 3675) loss: 1.378404
(Epoch 6 / 15) train acc: 0.518000; val_acc: 0.492000
(Iteration 1501 / 3675) loss: 1.332694
(Iteration 1601 / 3675) loss: 1.219642
(Iteration 1701 / 3675) loss: 1.335411
(Epoch 7 / 15) train acc: 0.557000; val_acc: 0.504000
(Iteration 1801 / 3675) loss: 1.289719
(Iteration 1901 / 3675) loss: 1.406917
(Epoch 8 / 15) train acc: 0.559000; val_acc: 0.502000
(Iteration 2001 / 3675) loss: 1.261266
(Iteration 2101 / 3675) loss: 1.184197
(Iteration 2201 / 3675) loss: 1.267301
(Epoch 9 / 15) train acc: 0.555000; val_acc: 0.515000
(Iteration 2301 / 3675) loss: 1.149572
(Iteration 2401 / 3675) loss: 1.189977
(Epoch 10 / 15) train acc: 0.591000; val_acc: 0.524000
(Iteration 2501 / 3675) loss: 1.317291
(Iteration 2601 / 3675) loss: 1.154614
(Epoch 11 / 15) train acc: 0.576000; val_acc: 0.515000
(Iteration 2701 / 3675) loss: 1.095757
(Iteration 2801 / 3675) loss: 1.239589
(Iteration 2901 / 3675) loss: 1.293036
(Epoch 12 / 15) train acc: 0.574000; val_acc: 0.512000
(Iteration 3001 / 3675) loss: 1.127370
(Iteration 3101 / 3675) loss: 1.251010
(Epoch 13 / 15) train acc: 0.597000; val_acc: 0.517000
(Iteration 3201 / 3675) loss: 1.272345
(Iteration 3301 / 3675) loss: 1.163338
(Iteration 3401 / 3675) loss: 1.140771
(Epoch 14 / 15) train acc: 0.628000; val_acc: 0.518000
(Iteration 3501 / 3675) loss: 1.175282
(Iteration 3601 / 3675) loss: 1.092050
```

```
(Epoch 15 / 15) train acc: 0.601000; val_acc: 0.539000
==============================================================
hidden size: 150, learning rate: 0.001, lr decay: 0.95
==============================================================
(Iteration 1 / 3675) loss: 2.305420
(Epoch 0 / 15) train acc: 0.153000; val_acc: 0.151000
(Iteration 101 / 3675) loss: 1.717234
(Iteration 201 / 3675) loss: 1.659075
(Epoch 1 / 15) train acc: 0.423000; val_acc: 0.430000
(Iteration 301 / 3675) loss: 1.628145
(Iteration 401 / 3675) loss: 1.510267
(Epoch 2 / 15) train acc: 0.496000; val_acc: 0.459000
(Iteration 501 / 3675) loss: 1.517149
(Iteration 601 / 3675) loss: 1.469220
(Iteration 701 / 3675) loss: 1.466176
(Epoch 3 / 15) train acc: 0.521000; val_acc: 0.506000
(Iteration 801 / 3675) loss: 1.466712
(Iteration 901 / 3675) loss: 1.270487
(Epoch 4 / 15) train acc: 0.561000; val_acc: 0.518000
(Iteration 1001 / 3675) loss: 1.418880
(Iteration 1101 / 3675) loss: 1.349000
(Iteration 1201 / 3675) loss: 1.280850
(Epoch 5 / 15) train acc: 0.555000; val_acc: 0.503000
(Iteration 1301 / 3675) loss: 1.202312
(Iteration 1401 / 3675) loss: 1.247353
(Epoch 6 / 15) train acc: 0.551000; val_acc: 0.528000
(Iteration 1501 / 3675) loss: 1.350306
(Iteration 1601 / 3675) loss: 1.271361
(Iteration 1701 / 3675) loss: 1.185882
(Epoch 7 / 15) train acc: 0.587000; val_acc: 0.513000
(Iteration 1801 / 3675) loss: 1.100769
(Iteration 1901 / 3675) loss: 1.123216
(Epoch 8 / 15) train acc: 0.598000; val_acc: 0.529000
(Iteration 2001 / 3675) loss: 1.128339
(Iteration 2101 / 3675) loss: 1.265283
(Iteration 2201 / 3675) loss: 1.356814
(Epoch 9 / 15) train acc: 0.594000; val_acc: 0.514000
(Iteration 2301 / 3675) loss: 1.148923
(Iteration 2401 / 3675) loss: 1.146622
(Epoch 10 / 15) train acc: 0.596000; val_acc: 0.525000
(Iteration 2501 / 3675) loss: 1.160169
(Iteration 2601 / 3675) loss: 1.083604
(Epoch 11 / 15) train acc: 0.605000; val_acc: 0.539000
(Iteration 2701 / 3675) loss: 1.133964
(Iteration 2801 / 3675) loss: 1.115286
(Iteration 2901 / 3675) loss: 1.033042
(Epoch 12 / 15) train acc: 0.599000; val_acc: 0.541000
(Iteration 3001 / 3675) loss: 1.153512
(Iteration 3101 / 3675) loss: 0.956935
(Epoch 13 / 15) train acc: 0.626000; val_acc: 0.543000
(Iteration 3201 / 3675) loss: 1.093087
(Iteration 3301 / 3675) loss: 1.029433
(Iteration 3401 / 3675) loss: 1.144068
(Epoch 14 / 15) train acc: 0.661000; val_acc: 0.533000
(Iteration 3501 / 3675) loss: 0.991575
(Iteration 3601 / 3675) loss: 1.059259
```

```
(Epoch 15 / 15) train acc: 0.620000; val_acc: 0.541000
best validation accuracy achieved during cross-validation: 0.543000
```

Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation set and the test set.

```
Validation set accuracy:  0.551

Test set accuracy:  0.546

best_two_layer_net.npy saved.
```

Inline Question 2:

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.

$\color{blue}{\textit{Your Answer :}}$ 1, 3

$\color{blue}{\textit{Your Explanation :}}$

1. Training on a larger and more diverse dataset allows the model to learn a wider variety of patterns, improving generalization and thus reducing the gap between training and test accuracy.
2. Increasing the number of hidden units expands the model’s capacity, allowing it to fit the training data more accurately. This can increase the gap.
3. A stronger regularization term prevents the model from fitting the training data too closely, thereby reducing overfitting and improving performance on the test set.