T1. Compute the forward and backward pass of the following computation. Note that this is a simplified residual connection.

$$x_1 = ReLU(x_0 * w_0 + b_0)$$
$$y_1 = x_1 * w_1 + b_1$$
$$z = ReLU(y_1 + x_0)$$

Let $x_0 = 1.0$, $w_0 = 0.3$, $w_1 = -0.2$, $b_0 = 0.1$, $b_1 = -0.3$. Find the gradient of z with respect to w_0, w_1, b_0, and b_1.

$x_1 = ReLU(1(0.3)+0.1) = 0.4$

$y_1 = 0.4(-0.2) + (-0.3) = -0.38$

$z = ReLU(-0.38 + 0.4) = 0.02$

$$\frac{\partial z}{\partial w_0} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} \cdot \frac{\partial x_1}{\partial w_0} = 1 \cdot w_1 \cdot x_0 = w_1 x_0 = (-0.2)(1.0) = -0.2$$

$$\frac{\partial z}{\partial w_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial w_1} = 1 \cdot x_1 = x_1 = 0.4$$

$$\frac{\partial z}{\partial b_0} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} \cdot \frac{\partial x_1}{\partial b_0} = 1 \cdot w_1 \cdot 1 = w_1 = -0.2$$

$$\frac{\partial z}{\partial b_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial b_1} = 1 \cdot 1 = 1$$

T2) (32, 1024), (32, 512), (32, 1) respectively

T3) $(31 \times 1024) + (1025 \times 512) + (513) = 557057$

T4) $\frac{\partial}{\partial h_i} P(y=i)$, $P(y=i) = \frac{e^{h_j}}{\sum_k e^{h_k}}$

case $i=j$; $\frac{\partial}{\partial h_i} \frac{e^{h_j}}{\sum_k e^{h_k}} = e^{h_j} \frac{\partial}{\partial h_i}\left(\frac{1}{\sum_k e^{h_k}}\right) + \frac{1}{\sum_k e^{h_k}} \cdot \frac{\partial}{\partial h_i} e^{h_j}$

$$= e^{h_j}\left(-\frac{e^{h_i}}{\left(\sum_k e^{h_k}\right)^2}\right) + \frac{e^{h_i}}{\sum_k e^{h_k}}$$

$$= \frac{e^{h_j}}{\left(\sum_k e^{h_k}\right)^2}\left(\left(\sum_k e^{h_k}\right) - e^{h_i}\right)$$

case $i \neq j$; $\frac{\partial}{\partial h_i} \frac{e^{h_j}}{\sum_k e^{h_k}} = e^{h_j} \frac{\partial}{\partial h_i} \frac{1}{\sum_k e^{h_k}}$

$$= e^{h_j} \cdot -\frac{e^{h_i}}{\left(\sum_k e^{h_k}\right)^2}$$

$$= -\frac{e^{h_j+h_i}}{\left(\sum_k e^{h_k}\right)^2}$$

$$\frac{\partial}{\partial h_i}(-L) = \frac{\partial}{\partial h_i}\left(\sum_j y_j \log(P(y=j))\right)$$

$$= \sum_{j \neq i}\left(\frac{\partial}{\partial h_i} y_j \log(P(y=j))\right) + \frac{\partial}{\partial h_i} y_i \log(P(y=i))$$

$$= \sum_{j \neq i}\left(y_j \cdot \frac{1}{\cancel{P(y=j)}}\left(-\frac{\cancel{e^{h_j}}+h_i}{\left(\sum_k e^{h_k}\right)^{\cancel{2}}}\right)\right) + y_i \cdot \frac{1}{P(y=i)} \cdot \frac{e^{h_i}}{\left(\sum_k e^{h_k}\right)^2}\left(\left(\sum_k e^{h_k}\right) - e^{h_i}\right)$$

$$= \sum_{j \neq i} y_j \cdot \frac{-e^{h_i}}{\left(\sum_k e^{h_k}\right)} + y_i \cdot \frac{1-e^{h_i}}{\sum_k e^{h_k}}$$

$$= -\frac{e^{h_i}}{\sum_k e^{h_k}} \cdot \underbrace{\sum_j y_j}_{=1} + y_i \quad = -P(y=i) + y_i$$

$$\therefore \frac{\partial}{\partial h_i}(L) = P(y=i) - y_i$$

Simple_Neural_Network_Lab

March 13, 2024

1 Two-Layer Neural Networks

In this part of the homework, we will work on building a simple Neural Network to classify digits using MNIST dataset. There are many powerful neural network frameworks nowadays that are relatively straightforward to use. However, we believe that in order to understand the theory behind neural networks, and to have the right intuitions in order to build, use, and debug more complex architectures, one must first start from the basics.

Your task is to complete the missing codes needed for training and testing of a simple fully-connected neural network.

The missing parts will be marked with TODO#N, where N represents the task number.

Many parts in this homework are modified from Stanford's cs231n assignments.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

from __future__ import print_function

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

from cattern.neural_net import TwoLayerNet

%load_ext autoreload
%autoreload 2
```

[ ]:
```
# Just a function that verifies if your answers are correct

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

We will use the class `TwoLayerNet` in the file `cattern/neural_net.py` to represent instances of our network. The network parameters (weights and biases) are stored in the instance variable `self.params` where the keys are parameter names and values are numpy arrays. Below, we initialize some toy data and a toy model that will guide you with your implementation.

[ ]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

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

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

2 Forward pass: compute scores

Open the file `cattern/neural_net.py` and look at the method `TwoLayerNet.loss`. This function takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Complete TODO#1 in `TwoLayerNet.loss`, by implementing the first part of the forward pass which uses the weights and biases to compute the scores for all inputs. The scores refer to the output of the network just before the softmax layer.

[ ]:
```
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
```

3 Forward pass: compute loss

In the same function, complete TODO#2 by implementing the second part that computes the data and regularizaion loss.

```
[ ]: loss, _ = net.loss(X, y, reg=0.1)
     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)))
     print(loss, correct_loss)
```

```
Difference between your loss and correct loss:
1.7985612998927536e-13
1.3037878913298202 1.30378789133
```

4 Backward pass

Implement the rest of the function by completing TODO#3. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

```
[ ]: from cattern.gradient_check import eval_numerical_gradient

     # Use numeric gradient checking to check your implementation of the backward␣
      ↪pass.
     # If your implementation is correct, the difference between the numeric and
     # analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

     loss, grads = net.loss(X, y, reg=0.05)

     # these should all be less than 1e-8 or so
     for param_name in grads:
         f = lambda W: net.loss(X, y, reg=0.05)[0]
         param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
      ↪verbose=False)
         print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
      ↪grads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447646e-11
W1 max relative error: 4.090896e-09
b1 max relative error: 2.738421e-09
```

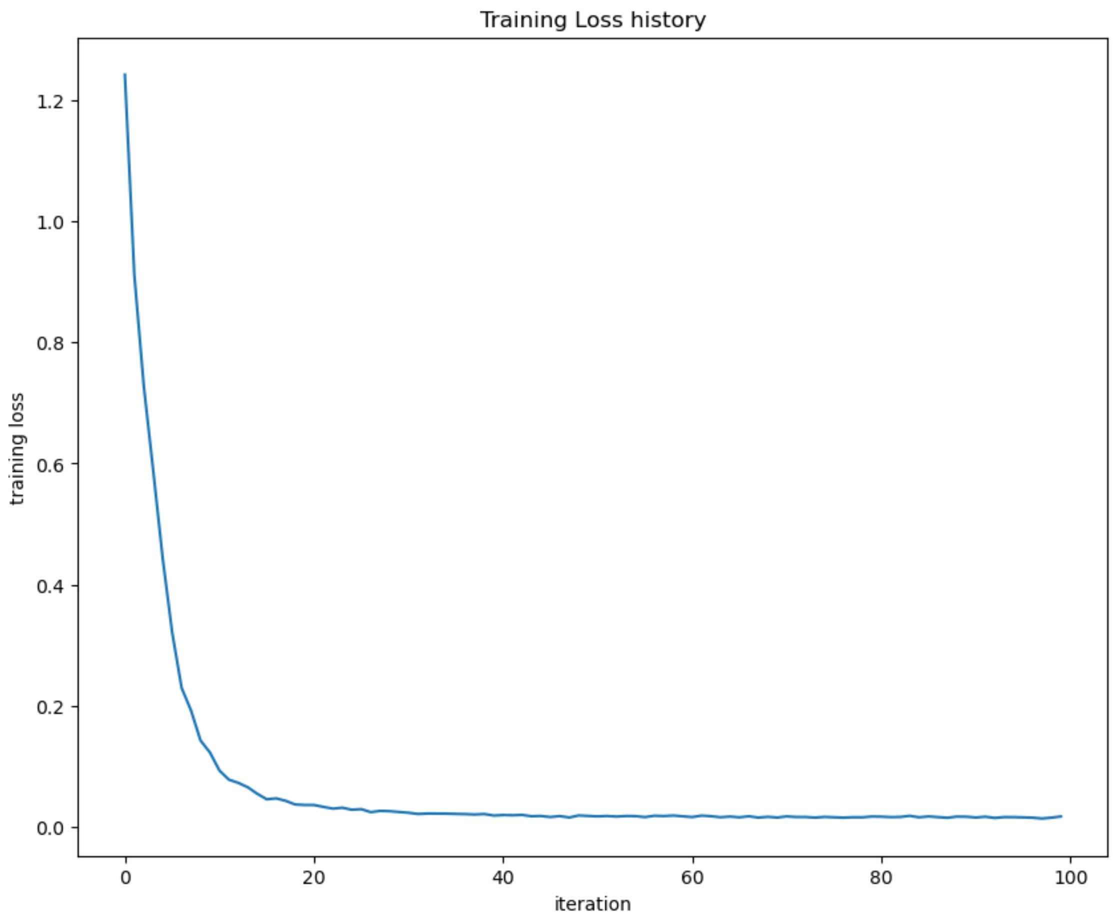
5 Train the network

To train the network we will use stochastic gradient descent (SGD). Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure (TODO#4-6). You will also have to implement `TwoLayerNet.predict` (TODO#7), as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.05.

```
[ ]: 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:  0.017143643532923733
```



6 Load the data

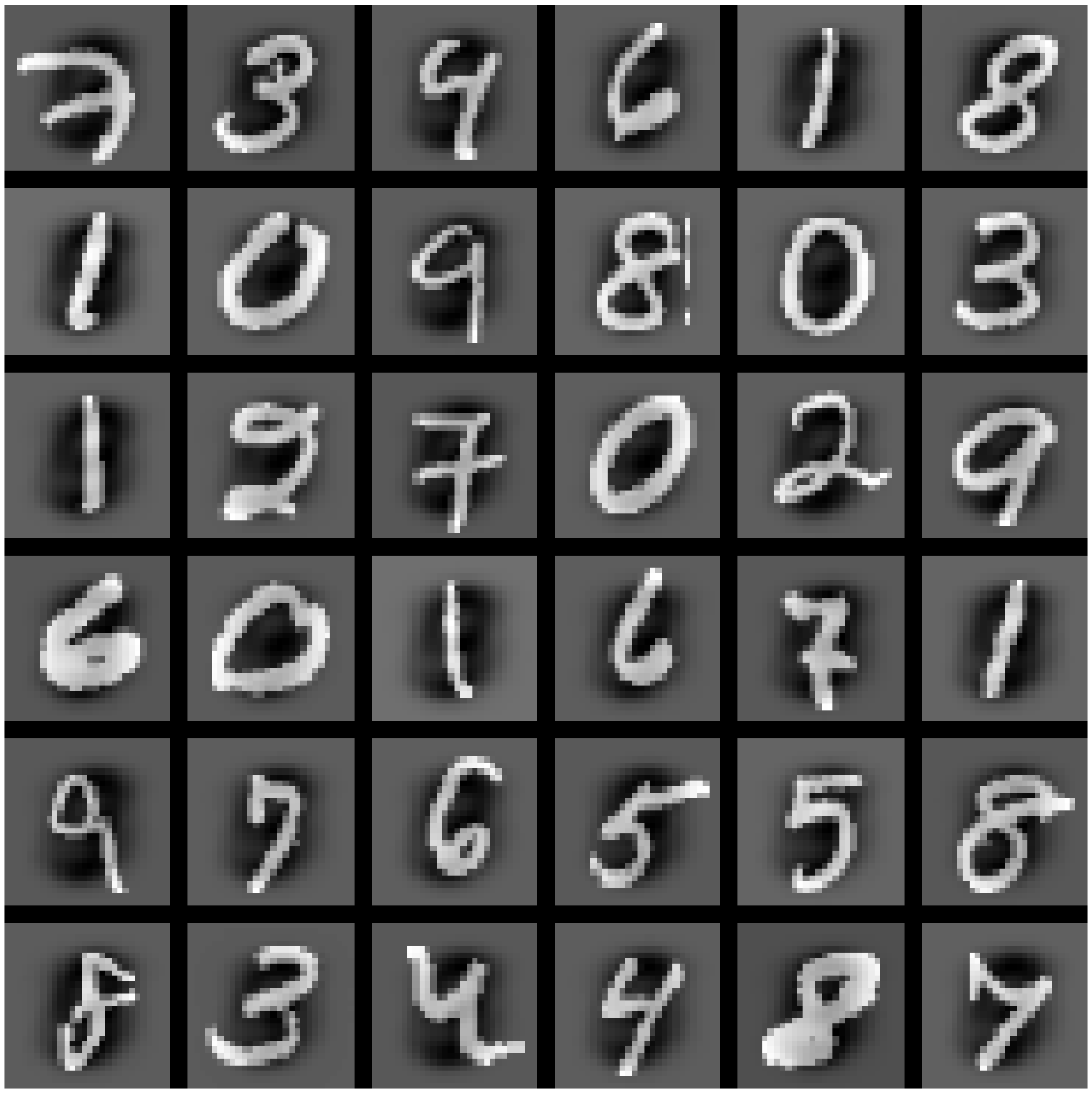
Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up MNIST data so we can use it to train a classifier on a real dataset.

```
[ ]: from mnist_data import load_mnist


     def get_mnist_data(num_training=55000, num_validation=5000, num_test=10000):
         """
         Load the MNIST dataset from disk and perform preprocessing to prepare
         it for the two-layer neural net classifier.
         """
         # Load the raw MNIST data
         X_train, y_train, X_val, y_val, X_test, y_test = load_mnist.
      ↪read_data_sets('mnist_data')

         # Normalize the data: subtract the mean image
```

```
    mean_image = np.mean(X_train, axis=0)
    X_train = X_train - mean_image
    X_val = X_val - mean_image
    X_test = X_test - mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test

X_train, y_train, X_val, y_val, X_test, y_test = get_mnist_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Extracting mnist_data/train-images-idx3-ubyte.gz
Extracting mnist_data/train-labels-idx1-ubyte.gz
Extracting mnist_data/t10k-images-idx3-ubyte.gz
Extracting mnist_data/t10k-labels-idx1-ubyte.gz
Train data shape:  (55000, 784)
Train labels shape:  (55000,)
Validation data shape:  (5000, 784)
Validation labels shape:  (5000,)
Test data shape:  (10000, 784)
Test labels shape:  (10000,)
```

[ ]:
```
from mnist_data.vis_utils import visualize_grid

# Visualize mnist data

def show_mnist_image(data):
  data = data.reshape(-1, 28, 28, 1)
  plt.imshow(visualize_grid(data, padding=3).astype('uint8').squeeze(axis=2))
  plt.gca().axis('off')
  plt.show()

show_mnist_image(X_train[:36])
```



7 Train a network

To train our network we will use SGD with momentum. We will use fixed learning rate to train this model.

```
[ ]: input_size = 28 * 28 * 1
     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=2000, batch_size=200,
                 learning_rate=1e-4, learning_rate_decay=1,
                 reg=0, verbose=True)

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

```
iteration 0 / 2000: loss 2.302592
iteration 100 / 2000: loss 2.302232
iteration 200 / 2000: loss 2.300876
iteration 300 / 2000: loss 2.291126
iteration 400 / 2000: loss 2.239348
iteration 500 / 2000: loss 2.008828
iteration 600 / 2000: loss 1.814898
iteration 700 / 2000: loss 1.425479
iteration 800 / 2000: loss 1.268225
iteration 900 / 2000: loss 0.994899
iteration 1000 / 2000: loss 0.847616
iteration 1100 / 2000: loss 0.758159
iteration 1200 / 2000: loss 0.633341
iteration 1300 / 2000: loss 0.536447
iteration 1400 / 2000: loss 0.478063
iteration 1500 / 2000: loss 0.397137
iteration 1600 / 2000: loss 0.415201
iteration 1700 / 2000: loss 0.385851
iteration 1800 / 2000: loss 0.517132
iteration 1900 / 2000: loss 0.425830
Validation accuracy:  0.8922
```

8 Learning Rate Decay

In the previous run, we used the same learning rate during the whole training process. This fix-sized learning rate disregards the benefit of larger learning rate at the beginning of the training, and it might suffer from overshooting around the minima.

Add learning rate decay to the train function, run the model again with larger starting learning rate and learning rate decay, then compare the losses.

[ ]:
```
net = TwoLayerNet(input_size, hidden_size, num_classes)
stats_LRDecay = net.train(X_train, y_train, X_val, y_val,
                          num_iters=2000, batch_size=200,
                          learning_rate=5e-4, learning_rate_decay=0.95,
                          reg=0, verbose=True)

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

```
iteration 0 / 2000: loss 2.302581
iteration 100 / 2000: loss 2.119326
iteration 200 / 2000: loss 0.804760
iteration 300 / 2000: loss 0.492363
iteration 400 / 2000: loss 0.407158
iteration 500 / 2000: loss 0.286767
iteration 600 / 2000: loss 0.318665
iteration 700 / 2000: loss 0.363674
iteration 800 / 2000: loss 0.273462
iteration 900 / 2000: loss 0.278906
iteration 1000 / 2000: loss 0.263698
iteration 1100 / 2000: loss 0.181063
iteration 1200 / 2000: loss 0.257531
iteration 1300 / 2000: loss 0.353448
iteration 1400 / 2000: loss 0.188915
iteration 1500 / 2000: loss 0.165813
iteration 1600 / 2000: loss 0.248227
iteration 1700 / 2000: loss 0.259703
iteration 1800 / 2000: loss 0.292172
iteration 1900 / 2000: loss 0.196248
Validation accuracy:  0.9432
```

9 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.94 on the validation set. This isn't very good for MNIST data which has reports of up to 0.99 accuracy.

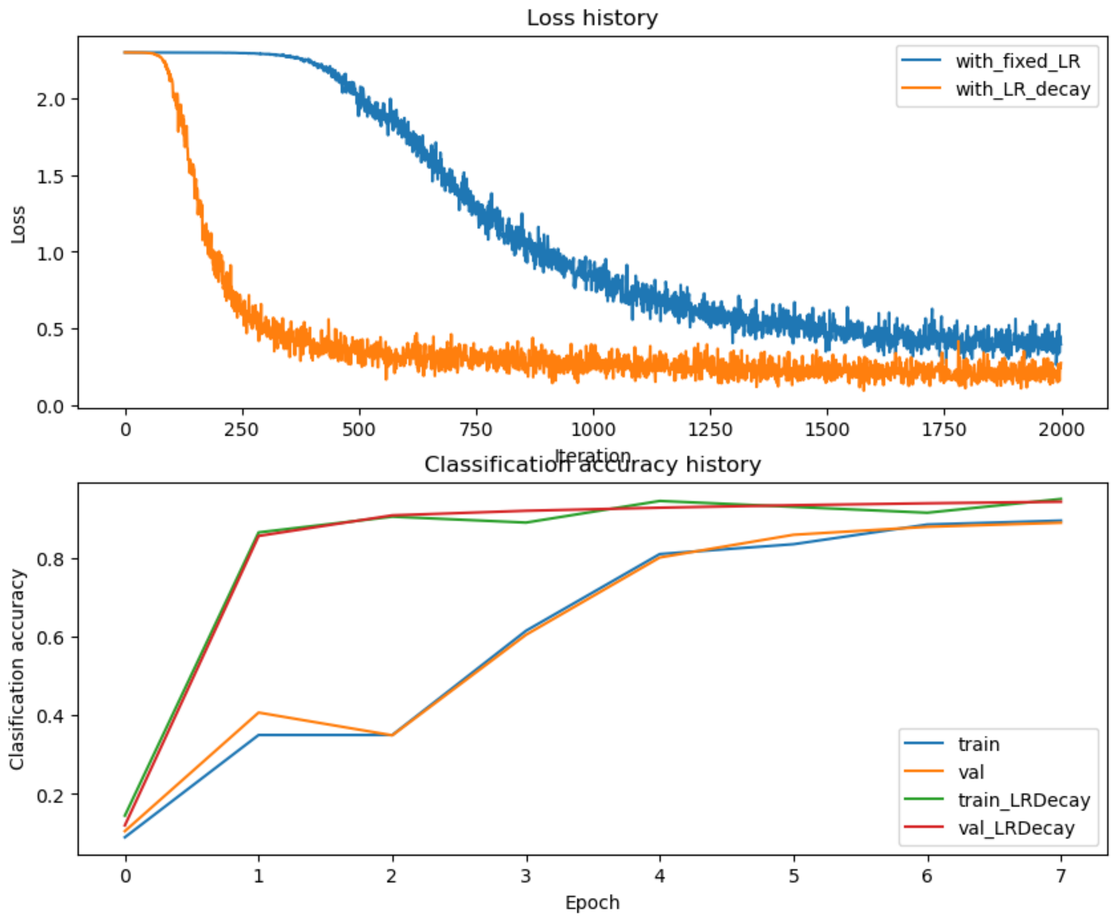
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

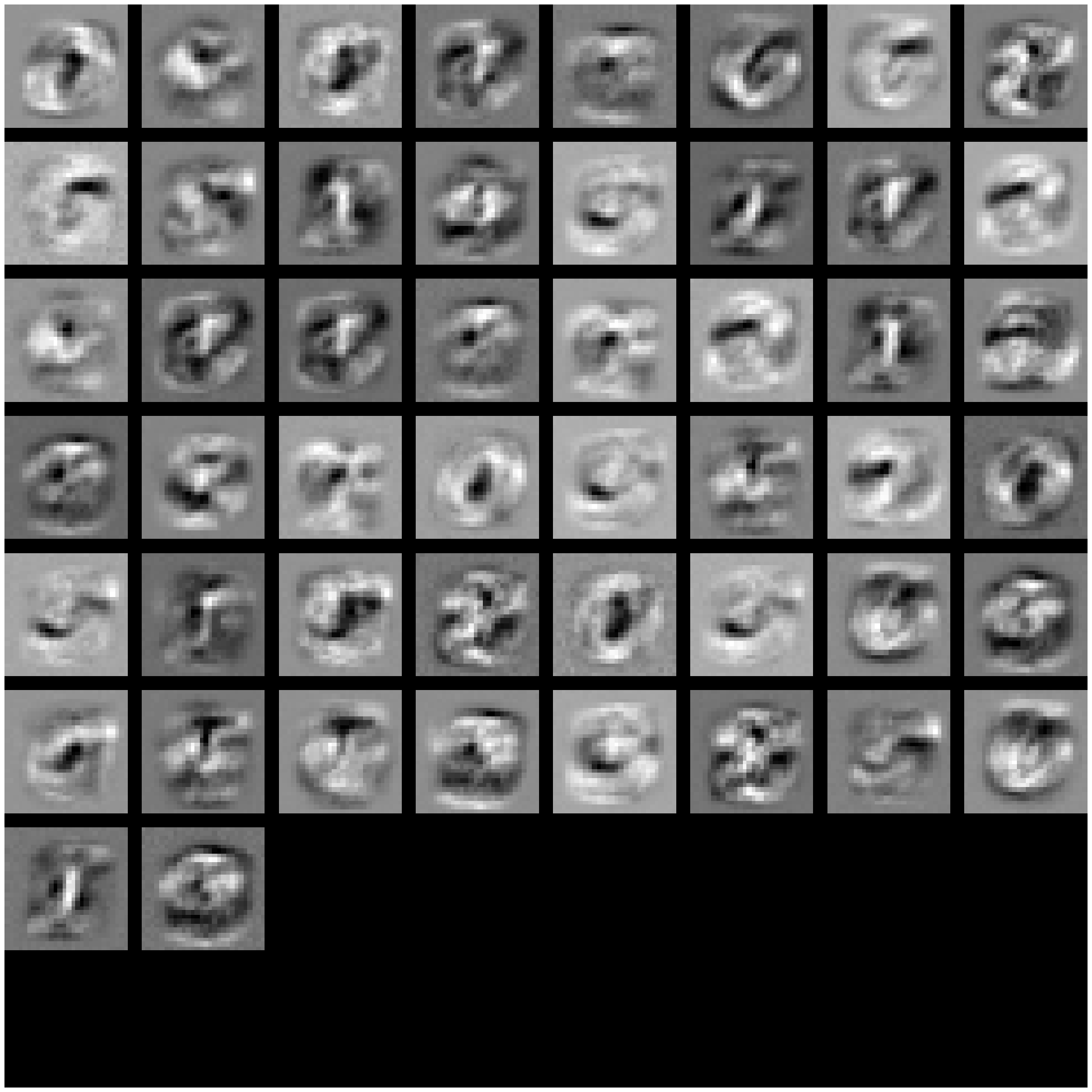
(You can think of the first layer weights as a projection W^TX. This is very similar to how we project our training data using PCA projection in our previous homework. Just like how we visualize the eigenfaces. We can also visualize the weights of the neural network in the same manner.)

Below, we will also show you lossed between two models we trained above. Do you notice the difference between the two?

[ ]:
```python
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'], label='with_fixed_LR')
plt.plot(stats_LRDecay['loss_history'], label='with_LR_decay')
plt.title('Loss history')
plt.legend()
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.plot(stats_LRDecay['train_acc_history'], label='train_LRDecay')
plt.plot(stats_LRDecay['val_acc_history'], label='val_LRDecay')
plt.title('Classification accuracy history')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.show()
```



[ ]:
```
# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(28, 28, 1, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8').squeeze(axis=2))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



10 Tune your hyperparameters

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.

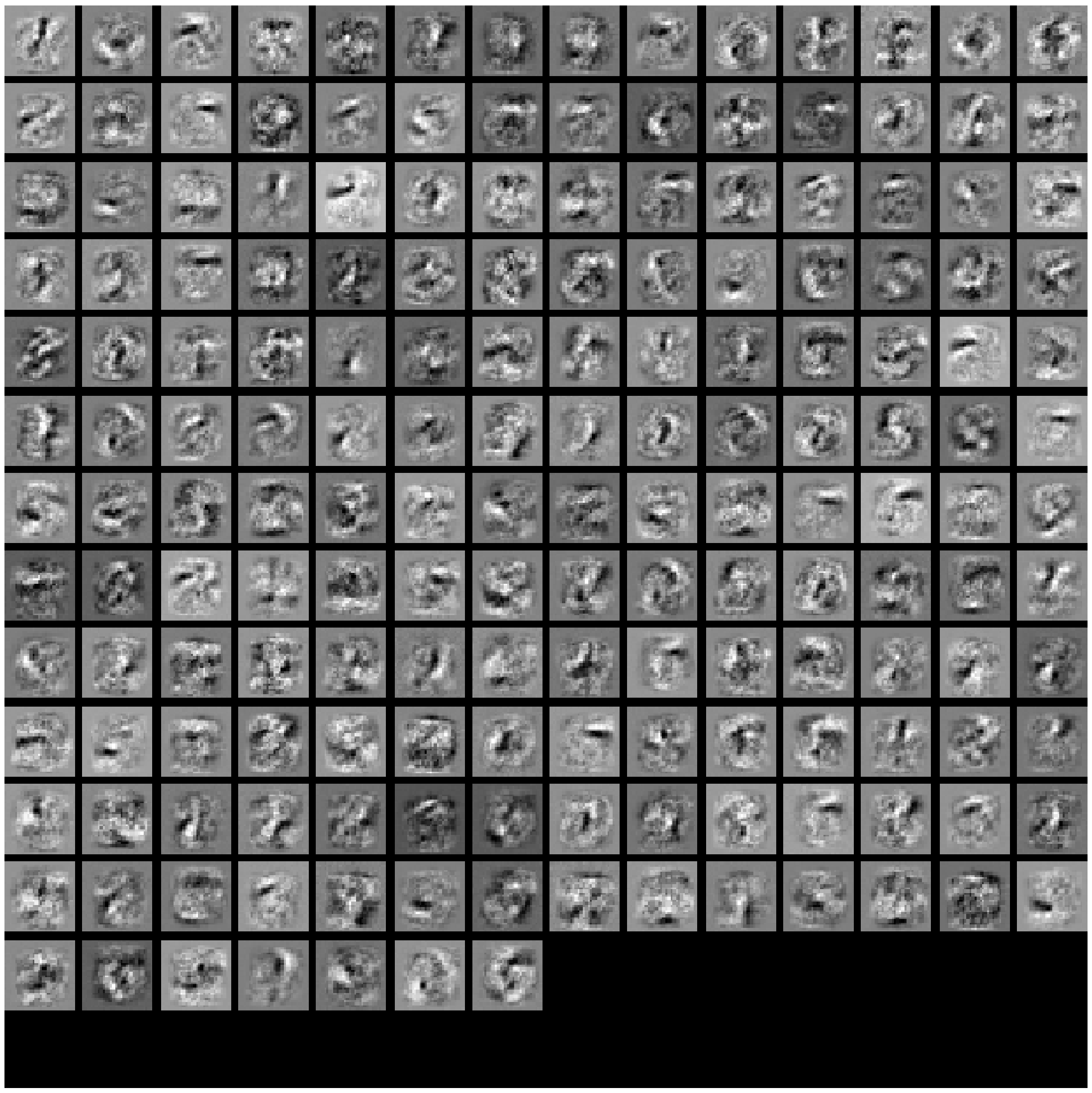
Approximate results. You should be aim to achieve a classification accuracy of greater than 97.4% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on MNIST as you can, 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.).

[ ]:
```
best_net = None # store the best model into this

################################################################################
# TODO#8: Tune hyperparameters using the validation set. Store your best␣
 ↪trained#
# model in best_net.                                                           ␣
 ↪#
#                                                                              ␣
 ↪#
# To help debug your network, it may help to use visualizations similar to the ␣
 ↪#
# ones we used above; these visualizations will have significant qualitative   ␣
 ↪#
# differences from the ones we saw above for the poorly tuned network.         ␣
 ↪#
#                                                                              ␣
 ↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
 ↪#
# write code to sweep through possible combinations of hyperparameters         ␣
 ↪#
# automatically like we did on the previous exercises.                         ␣
 ↪#
################################################################################

# Define hyperparameter ranges
param_ranges = {
    # 'hidden_size': [50, 100, 200, 75, 125, 150, 175],
    'hidden_size': [175],
    # 'learning_rate': [1e-3, 5e-4, 1e-4],
    'learning_rate': [1e-3],
    # 'learning_rate_decay': [0.95, 0.75, 0.5],
    'learning_rate_decay': [0.98],
    # 'reg': [1e-3, 5e-4, 1e-4, 0.95, 0.75, 0.5],
    'reg': [5e-6],
    'num_iters': [10000],
    # 'batch_size': [100, 200, 300],
    'batch_size': [100],
}

best_val_acc = -1

# Random search (but manual anyway lol)
num_iter = 1

for _ in range(num_iter):
    # Sample hyperparameters randomly from the ranges
    hyperparameters = {
        param: np.random.choice(values) for param, values in param_ranges.
 ↪items()
    }

    # Train a model with the sampled hyperparameters
    model = TwoLayerNet(input_size, hyperparameters['hidden_size'], num_classes)
    stats = model.train(X_train, y_train, X_val, y_val,
                        learning_rate=hyperparameters['learning_rate'],
                       ␣
 ↪learning_rate_decay=hyperparameters['learning_rate_decay'],
                        reg=hyperparameters['reg'],
                        num_iters=hyperparameters['num_iters'],
                        batch_size=hyperparameters['batch_size'])

    # Evaluate the model on the validation set
    val_acc = (model.predict(X_val) == y_val).mean()

    # Check if this model has the best validation accuracy
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_net = model

    print(f"Validation accuracy for hyperparameters {hyperparameters}:␣
 ↪{val_acc}")

print("Best validation accuracy:", best_val_acc)
#################################################################################
#                                END OF TODO#8                                 ␣
 ↪#
#################################################################################
```

```
Validation accuracy for hyperparameters {'hidden_size': 175, 'learning_rate':
0.001, 'learning_rate_decay': 0.98, 'reg': 5e-06, 'num_iters': 10000,
'batch_size': 100}: 0.9806
Best validation accuracy: 0.9806
```

```
[ ]: # visualize the weights of the best network
     show_net_weights(best_net)
```



11 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 96.3%.

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

```
Test accuracy:  0.9785
```