Homework 4

Week 5-6: Neural Networks

Patthadon Phengpinij

Collaborators. ChatGPT

1 The Basics

In this section, we will review some of the basic materials taught in class. These are simple tasks and integral to the understanding of deep neural networks, but many students seem to misunderstand.

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

$$\begin{aligned} x_1 &= ReLU(x_0 * w_0 + b_0) \\ y_1 &= x_1 * w_1 + b_1 \\ z &= ReLU(y_1 + x_0) \end{aligned}$$

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.

Solution. First, let's compute the forward pass:
1. Compute x_1:

$$\begin{aligned} x_1 &= ReLU(x_0 * w_0 + b_0) \\ &= ReLU(1.0 * 0.3 + 0.1) \\ &= ReLU(0.4) \\ x_1 &= 0.4 \end{aligned}$$

2. Compute y_1:

$$\begin{aligned} y_1 &= x_1 * w_1 + b_1 \\ &= 0.4 * (-0.2) + (-0.3) \\ &= -0.08 - 0.3 \\ y_1 &= -0.38 \end{aligned}$$

3. Compute z:

$$\begin{aligned} z &= ReLU(y_1 + x_0) \\ &= ReLU(-0.38 + 1.0) \\ &= ReLU(0.62) \\ z &= 0.62 \end{aligned}$$

Thus, the output of the forward pass is $\boxed{z = 0.62}$.
Next, let's compute the backward pass:
1. Compute the gradient of z with respect to y_1 and x_0:

$$\begin{aligned} \frac{\partial z}{\partial y_1} &= 1 \quad (\text{since } y_1 + x_0 = 0.62 > 0) \\ \frac{\partial z}{\partial x_0} &= 1 \quad (\text{since } y_1 + x_0 = 0.62 > 0) \end{aligned}$$

2. Compute the gradient of y_1 with respect to x_1, w_1, and b_1:

$$\frac{\partial y_1}{\partial x_1} = w_1 = -0.2$$
$$\frac{\partial y_1}{\partial w_1} = x_1 = 0.4$$
$$\frac{\partial y_1}{\partial b_1} = 1$$

3. Compute the gradient of x_1 with respect to x_0, w_0, and b_0:

$$\frac{\partial x_1}{\partial x_0} = w_0 = 0.3 \quad (\text{since } x_0 * w_0 + b_0 = 0.4 > 0)$$
$$\frac{\partial x_1}{\partial w_0} = x_0 = 1.0 \quad (\text{since } x_0 * w_0 + b_0 = 0.4 > 0)$$
$$\frac{\partial x_1}{\partial b_0} = 1 \quad (\text{since } x_0 * w_0 + b_0 = 0.4 > 0)$$

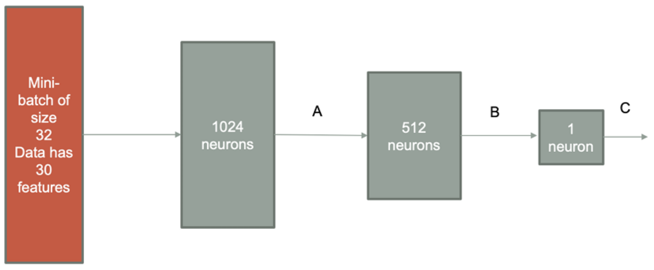
4. Now we can compute the gradients of z with respect to w_0, w_1, b_0, and b_1 using the chain rule:

$$\frac{\partial z}{\partial w_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial w_1} = 1 \cdot 0.4 = 0.4$$
$$\frac{\partial z}{\partial b_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial b_1} = 1 \cdot 1 = 1$$
$$\frac{\partial z}{\partial w_0} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} \cdot \frac{\partial x_1}{\partial w_0} = 1 \cdot (-0.2) \cdot 1.0 = -0.2$$
$$\frac{\partial z}{\partial b_0} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} \cdot \frac{\partial x_1}{\partial b_0} = 1 \cdot (-0.2) \cdot 1 = -0.2$$

Thus, the gradients are:

$$\boxed{\frac{\partial z}{\partial w_0} = -0.2, \quad \frac{\partial z}{\partial w_1} = 0.4, \quad \frac{\partial z}{\partial b_0} = -0.2, \quad \frac{\partial z}{\partial b_1} = 1}$$

T2. Given the following network architecture specifications, determine the size of the output A, B, and C.



Solution. Let's analyze the network architecture step by step:
1. The input layer has a size of **30 features**.
2. The first hidden layer has **1024 neurons**, so the output **A will have a size of 1024**.
3. The second hidden layer has **512 neurons**, so the output **B will have a size of 512**.
4. The output layer has **1 neuron**, so the output **C will have a size of 1**.

T3. What is the total number of learnable parameters in this network? (Don't forget the bias term)

Solution. Since we have three layers of learnable parameters, we will compute the number of parameters for each layer and then sum them up.

1. For the first hidden layer (input to first hidden layer):
    1. Weights: 30 (input features) $\times$ 1024 (neurons) $= 30720$
    2. Biases: 1024 (neurons) $= 1024$
    3. Total for first hidden layer: $30720 + 1024 = 31744$
2. For the second hidden layer (first hidden layer to second hidden layer):
    1. Weights: 1024 (neurons in first hidden layer) $\times$ 512 (neurons in second hidden layer) $= 524288$
    2. Biases: 512 (neurons) $= 512$
    3. Total for second hidden layer: $524288 + 512 = 524800$
3. For the output layer (second hidden layer to output layer):
    1. Weights: 512 (neurons in second hidden layer) $\times$ 1 (neuron in output layer) $= 512$
    2. Biases: 1 (neuron) $= 1$
    3. Total for output layer: $512 + 1 = 513$

Now, we can sum up the total number of learnable parameters:

$$\text{Total parameters} = 31744 + 524800 + 513 = \mathbf{557057} \quad \textbf{learnable parameters}$$

2 Deep Learning from (almost) scratch

In this section we will code simple a neural network model from scratch (numpy). However, before we go into coding let's start with some loose ends, namely the gradient of the softmax layer.

Recall in class we define the softmax layer as:

$$P(y = j) = \frac{exp(h_j)}{\sum_k exp(h_k)}$$

where h_j is the output of the previous layer for class index j.

The cross entropy loss is defined as:

$$L = -\sum_j y_j \log P(y = j)$$

where y_j is 1 if y is class j, and 0 otherwise.

T4. Prove that the derivative of the loss with respect to h_i is $P(y = i) - y_i$. In other words, find $\frac{\partial L}{\partial h_i}$ for $i \in \{0, \ldots, N-1\}$ where N is the number of classes.

Hint: first find $\frac{\partial P(y=j)}{\partial h_j}$ for the case where $j = i$, and the case where $j \neq i$. Then, use the results with chain rule to find the derivative of the loss.

Proof. Consider the cross entropy loss:

$$L = -\sum_j y_j \log P(y=j)$$

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

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

$$\frac{\partial L}{\partial h_i} = -\sum_j y_j \frac{1}{P(y=j)}\frac{\partial P(y=j)}{\partial h_i}$$

Next, we will find $\frac{\partial P(y=j)}{\partial h_i}$ for the two cases:

- When $j=i$:

$$\frac{\partial P(y=j)}{\partial h_i} = \frac{\partial}{\partial h_i}\left[\frac{exp(h_i)}{\sum_k exp(h_k)}\right]$$

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

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

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

$$\frac{\partial P(y=j)}{\partial h_i} = \frac{\left(\sum_{k\neq i} exp(h_k)\right)exp(h_i)}{\left(\sum_k exp(h_k)\right)^2}$$

$$\frac{1}{P(y=j)}\frac{\partial P(y=j)}{\partial h_i} = \frac{\sum_k exp(h_k)}{exp(h_i)} \times \frac{\left(\sum_{k\neq i} exp(h_k)\right)exp(h_i)}{\left(\sum_k exp(h_k)\right)^2}$$

$$\frac{1}{P(y=j)}\frac{\partial P(y=j)}{\partial h_i} = \frac{\sum_{k\neq i} exp(h_k)}{\sum_k exp(h_k)}$$

- When $j \neq i$:

$$\frac{\partial P(y=j)}{\partial h_i} = \frac{\partial}{\partial h_i}\left[\frac{exp(h_j)}{\sum_k exp(h_k)}\right]$$

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

$$= \frac{0 - exp(h_j)\cdot exp(h_i)}{\left(\sum_k exp(h_k)\right)^2}$$

$$\frac{\partial P(y=j)}{\partial h_i} = -\frac{exp(h_j)\cdot exp(h_i)}{\left(\sum_k exp(h_k)\right)^2}$$

$$\frac{1}{P(y=j)}\frac{\partial P(y=j)}{\partial h_i} = \frac{\sum_k exp(h_k)}{exp(h_j)} \times \left[-\frac{exp(h_j)\cdot exp(h_i)}{\left(\sum_k exp(h_k)\right)^2}\right]$$

$$\frac{1}{P(y=j)}\frac{\partial P(y=j)}{\partial h_i} = -\frac{exp(h_i)}{\sum_k exp(h_k)}$$

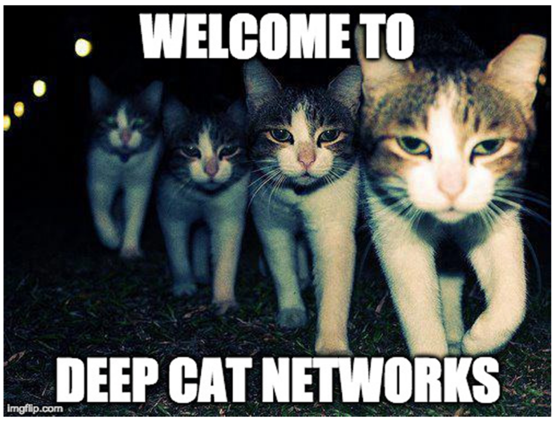
Now, we can substitute the results back into the expression for $\frac{\partial L}{\partial h_i}$:

$$\begin{aligned}
\frac{\partial L}{\partial h_i} &= -\sum_j y_j \frac{1}{P(y=j)} \frac{\partial P(y=j)}{\partial h_i} \\
&= -y_i \frac{1}{P(y=i)} \frac{\partial P(y=i)}{\partial h_i} - \sum_{j \neq i} y_j \frac{1}{P(y=j)} \frac{\partial P(y=j)}{\partial h_i} \\
&= -y_i \frac{\sum_{k \neq i} exp(h_k)}{\sum_k exp(h_k)} + \sum_{j \neq i} y_j \frac{exp(h_i)}{\sum_k exp(h_k)} \\
&= -y_i \frac{\sum_{k \neq i} exp(h_k)}{\sum_k exp(h_k)} + \frac{exp(h_i)}{\sum_k exp(h_k)} \sum_{j \neq i} y_j \\
&= -y_i \frac{\sum_{k \neq i} exp(h_k)}{\sum_k exp(h_k)} + \frac{exp(h_i)}{\sum_k exp(h_k)} (1 - y_i) \\
&= -y_i \frac{\sum_{k \neq i} exp(h_k)}{\sum_k exp(h_k)} + \frac{exp(h_i)}{\sum_k exp(h_k)} - y_i \frac{exp(h_i)}{\sum_k exp(h_k)} \\
&= \frac{exp(h_i)}{\sum_k exp(h_k)} - y_i \frac{\sum_{k \neq i} exp(h_k) + exp(h_i)}{\sum_k exp(h_k)} \\
&= P(y=i) - y_i \frac{\sum_k exp(h_k)}{\sum_k exp(h_k)} \\
\frac{\partial L}{\partial h_i} &= P(y=i) - y_i
\end{aligned}$$

Next, we will code a simple neural network using numpy. Use the starter code `hw4.zip` on github. There are 8 tasks you need to complete in the starter code.

Hints: In order to do this part of the assignment, you will need to find gradients of vectors over matrices. We have done gradients of scalars (Traces) over matrices before, which is a matrix (two-dimensional). However, gradients of vectors over matrices will be a tensor (three-dimensional), and the properties we learned will not work. I highly recommend you find the gradients in parts. In other words, compute the gradient for each element in the the matrix/vector separately. Then, combine the result back into matrices. For more information, you can read this simple guide http://cs231n.stanford.edu/vecDerivs.pdf.

Happy coding.



All the tasks that are in the starter code, also the result and writeup solution, will be given in the **Appendix 1:** Simple Neural Network.

APPENDIX 1

Simple Neural Network

simple-neural-network

February 28, 2026

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 **Tx.**, where `x` represents the task number.

Many parts in this homework are modified from `Stanford's CS231n` assignments.

[1]:
```
from __future__ import print_function

import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Set default configuration of plots.
plt.rcParams["figure.figsize"] = (10.0, 8.0)
plt.rcParams["image.interpolation"] = "nearest"
plt.rcParams["image.cmap"] = "gray"

from cattern.neural_net import TwoLayerNet

%load_ext autoreload
%autoreload 2
```

[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.

[3]:
```python
# 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()
```

[4]:
```python
print(X.shape)
print(y.shape)
```

```
(5, 4)
(5,)
```

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 **T5.** 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.

[5]:
```python
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: ", end="")
print(f"{np.sum(np.abs(scores - correct_scores)):.8f}")
```

```
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: 0.00000004
```

3 Forward Pass: Compute Loss

In the same function, complete **T6.** by implementing the second part that computes the data and regularization loss.

[6]:
```
loss, _ = net.loss(X, y, reg=0.1)
correct_loss = 1.30378789133

print(f"Your loss:")
print(loss)

print()

print("Correct loss:")
print(correct_loss)
print()

# Should be very small, we get < 1e-12
print("Difference between your loss and correct loss: ", end="")
print(f"{np.sum(np.abs(loss - correct_loss)):.13f}")
```

```
Your loss:
1.3037878913298202

Correct loss:
1.30378789133

Difference between your loss and correct loss: 0.0000000000002
```

4 Backward Pass

Implement the rest of the function by completing **T7.** 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:

[7]:
```
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: 4.447625e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

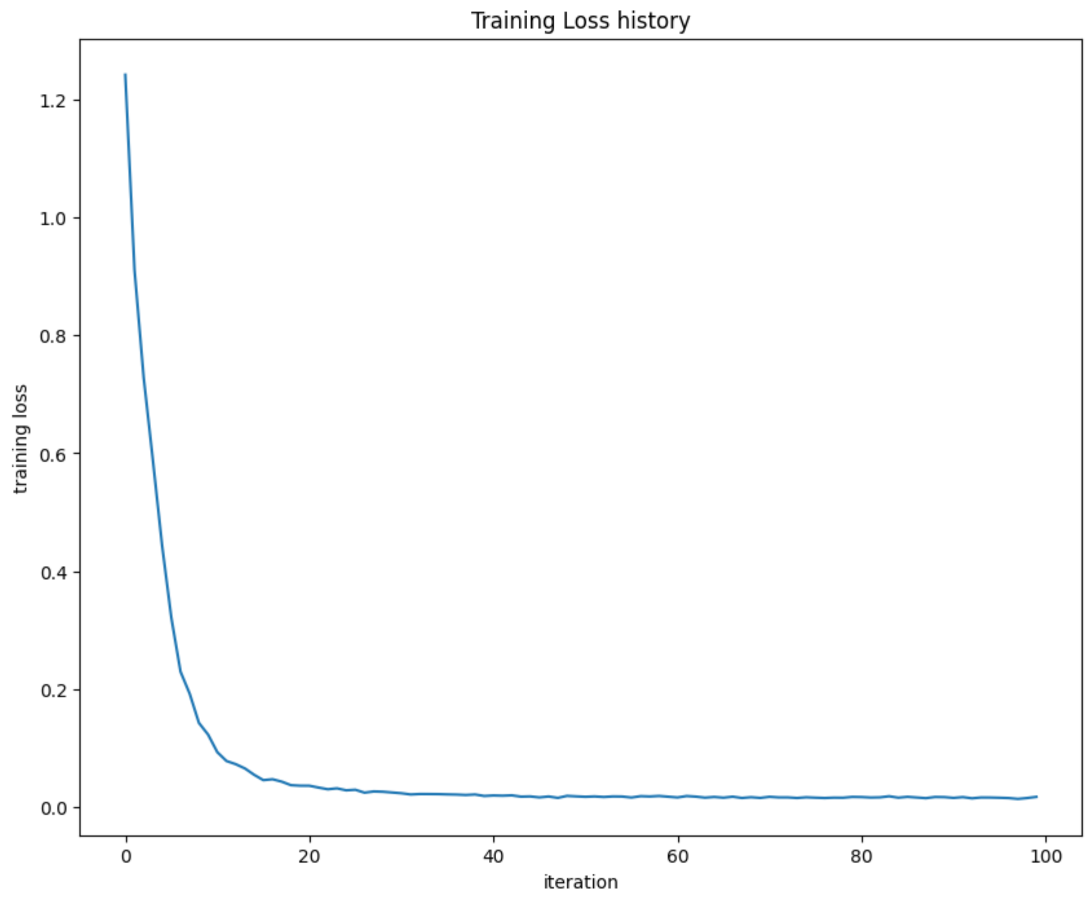
5 Train The Network

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 **(T8. T9. T10.)**. You will also have to implement `TwoLayerNet.predict` **(T11.)**, 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.

[8]:
```
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:  0.017143643532923815
```



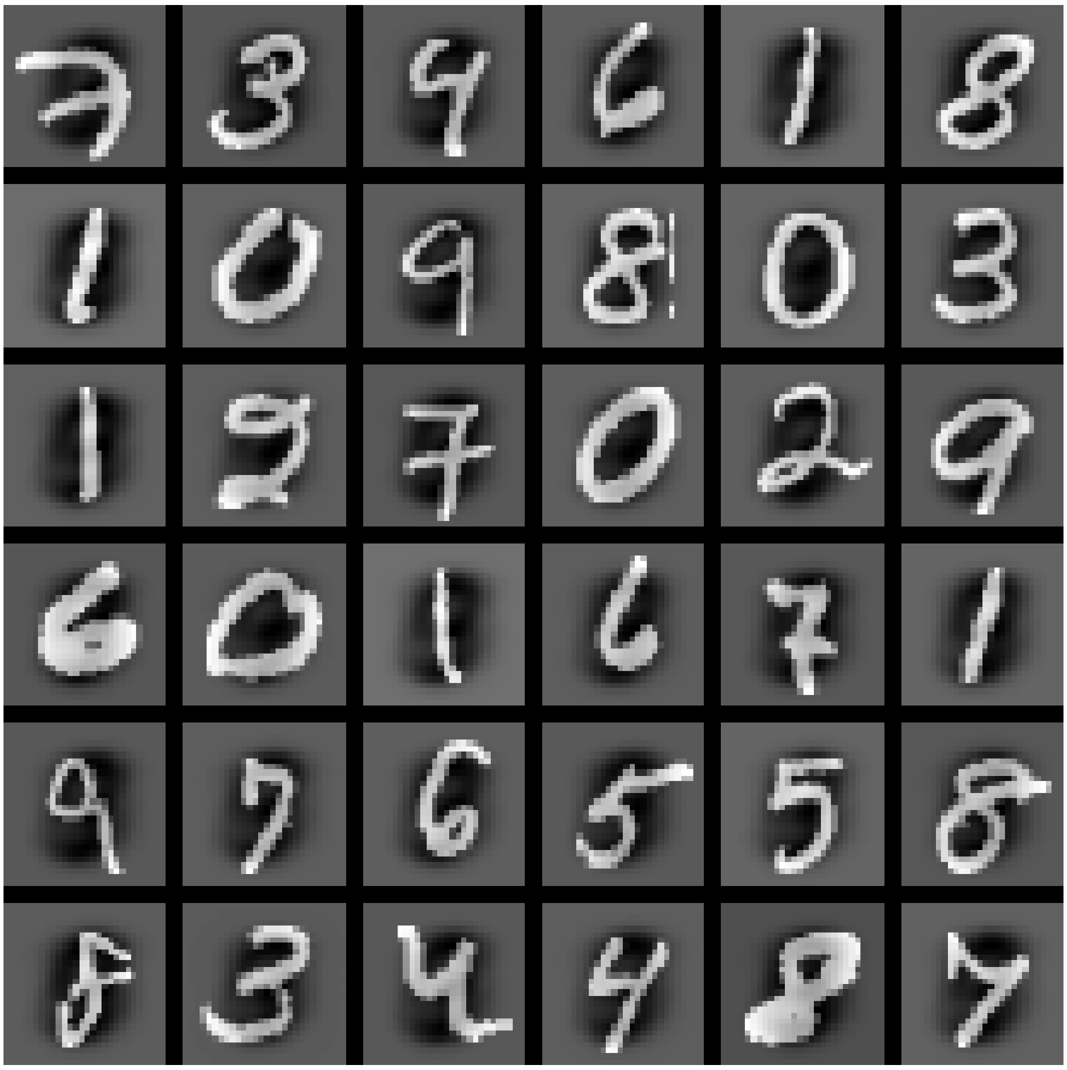
6 Load the data

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.

[9]:
```
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,)
```

[10]:
```
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.

[11]:
```
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.

[12]:
```
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.

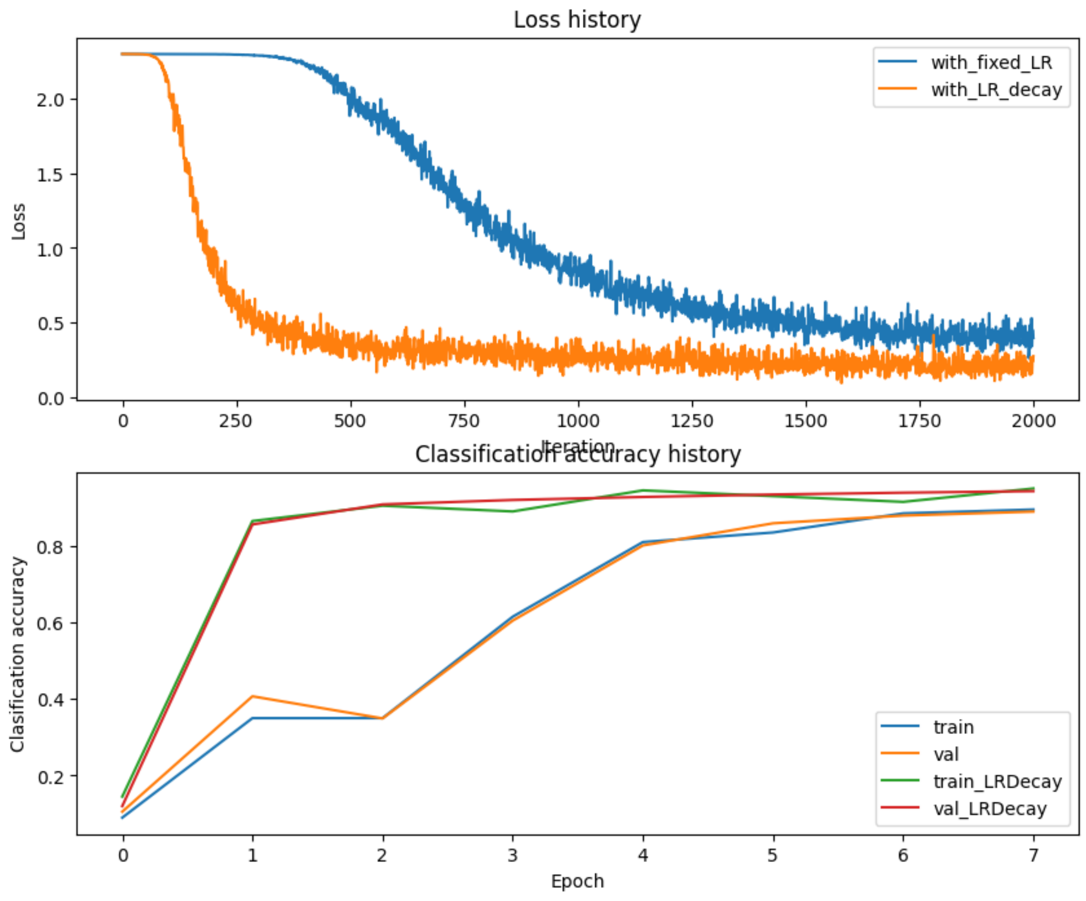
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

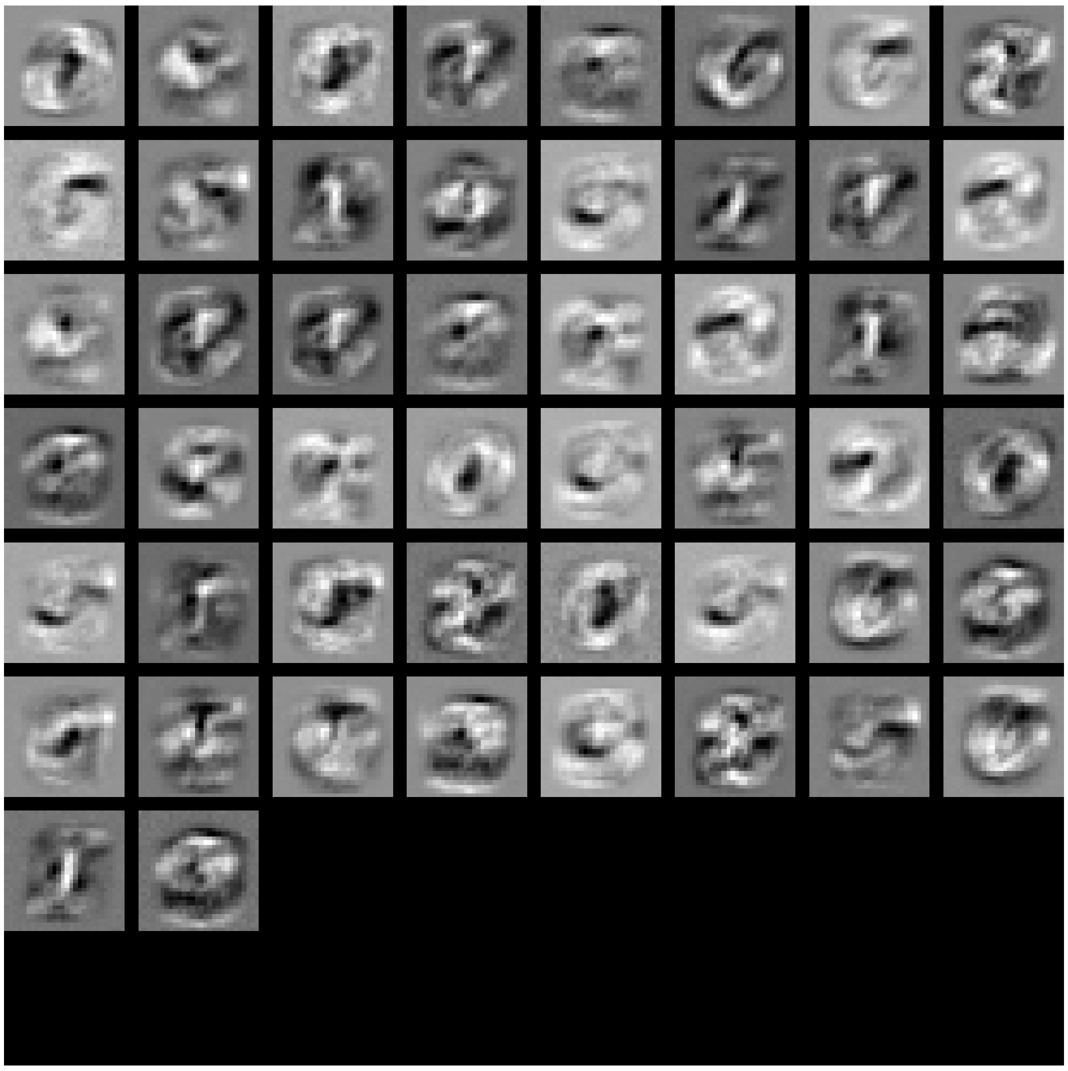
(You can think of the first layer weights as a projection $W^T X$. This is very similar to how we project our training data using PCA projection in our previous homework. Just like how we visualize the eigenfaces. We can also visualize the weights of the neural network in the same manner.)

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

[13]:
```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()
```



[14]:
```python
# 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.

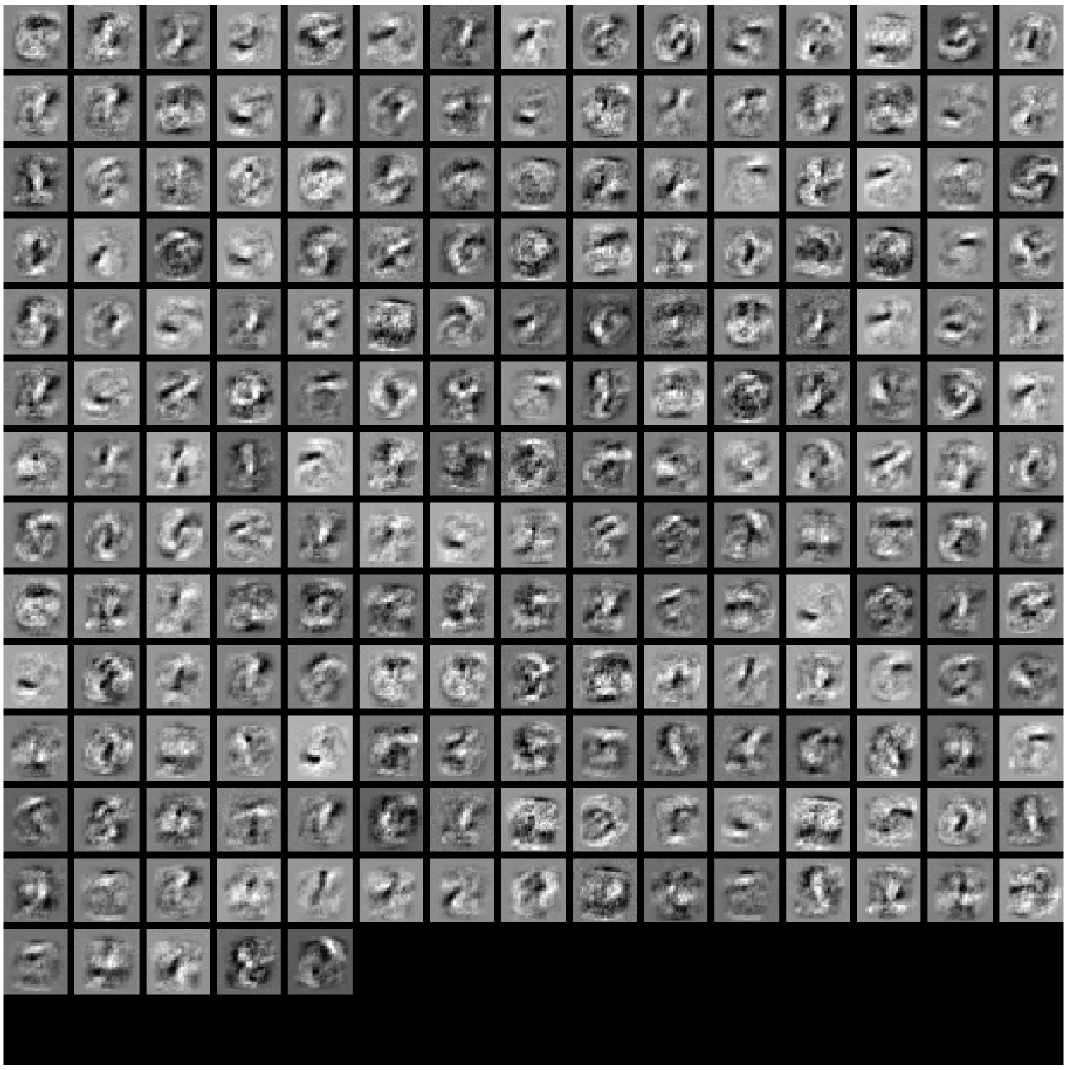
Approximate results. You should be aim to achieve a classification accuracy of greater than 97.4% on the validation set.

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.).

[ ]:
```
################################################################################
# T12: 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.                         #
################################################################################

# Store the best model into this variable.
best_net = TwoLayerNet(input_size, 200, num_classes)
stats_best_net = best_net.train(
    X_train, y_train, X_val, y_val,
    num_iters=4000,
    batch_size=300,
    learning_rate=1e-3,
    learning_rate_decay=0.95,
    reg=0,
    verbose=True
)

# Predict on the validation set
val_acc_best_net = (best_net.predict(X_val) == y_val).mean()
print("Validation accuracy: ", val_acc_best_net)

#################################################################################
#                                   END OF T12                                 ␣
 ↪#
#################################################################################
```

```
iteration 0 / 4000: loss 2.302595
iteration 100 / 4000: loss 0.698376
iteration 200 / 4000: loss 0.412657
iteration 300 / 4000: loss 0.296848
iteration 400 / 4000: loss 0.244699
iteration 500 / 4000: loss 0.287703
iteration 600 / 4000: loss 0.189672
iteration 700 / 4000: loss 0.194406
iteration 800 / 4000: loss 0.165330
iteration 900 / 4000: loss 0.173304
iteration 1000 / 4000: loss 0.114969
iteration 1100 / 4000: loss 0.172537
iteration 1200 / 4000: loss 0.197263
iteration 1300 / 4000: loss 0.137189
iteration 1400 / 4000: loss 0.206309
iteration 1500 / 4000: loss 0.201528
iteration 1600 / 4000: loss 0.104132
iteration 1700 / 4000: loss 0.096440
iteration 1800 / 4000: loss 0.077341
iteration 1900 / 4000: loss 0.134956
iteration 2000 / 4000: loss 0.155198
iteration 2100 / 4000: loss 0.105741
iteration 2200 / 4000: loss 0.109886
iteration 2300 / 4000: loss 0.090519
iteration 2400 / 4000: loss 0.075001
iteration 2500 / 4000: loss 0.123460
iteration 2600 / 4000: loss 0.083451
iteration 2700 / 4000: loss 0.078146
iteration 2800 / 4000: loss 0.074030
iteration 2900 / 4000: loss 0.086285
iteration 3000 / 4000: loss 0.066089
iteration 3100 / 4000: loss 0.072899
iteration 3200 / 4000: loss 0.039844
iteration 3300 / 4000: loss 0.046123
iteration 3400 / 4000: loss 0.047941
iteration 3500 / 4000: loss 0.050394
iteration 3600 / 4000: loss 0.046262
iteration 3700 / 4000: loss 0.039824
iteration 3800 / 4000: loss 0.055677
iteration 3900 / 4000: loss 0.031222
Validation accuracy:  0.9746
```

[16]:
```
# 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%.

[17]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.974
```

APPENDIX 2

Homework 4 Template Code in `CATTERN` Directory

cattern

February 28, 2026

1 neural_net.py

[1]:
```
from __future__ import print_function

import numpy as np
```

[ ]:
```
class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  N, a hidden layer dimension of H, and performs classification over C classes.
  We train the network with a softmax loss function and L2 regularization on the
  weight matrices. The network uses a ReLU nonlinearity after the first fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softmax

  The outputs of the second fully-connected layer are the scores for each class.
  """

  def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values and
    biases are initialized to zero. Weights and biases are stored in the
    variable self.params, which is a dictionary with the following keys:

    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,)

    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.
    """
```

```
        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)


    def loss(self, X, y=None, reg=0.0):
        """
        Compute the loss and gradients for a two layer fully connected neural␣
↪network.

        Inputs:
        - X: Input data of shape (N, D). Each X[i] is a training sample.
        - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
          an integer in the range 0 <= y[i] < C. This parameter is optional; if it
          is not passed then we only return scores, and if it is passed then we
          instead return the loss and gradients.
        - reg: Regularization strength.

        Returns:
        If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
        the score for class c on input X[i].

        If y is not None, instead return a tuple of:
        - loss: Loss (data loss and regularization loss) for this batch of training
          samples.
        - grads: Dictionary mapping parameter names to gradients of those parameters
          with respect to the loss function; has the same keys as self.params.
        """
        # Unpack variables from the params dictionary
        W1, b1 = self.params["W1"], self.params["b1"]
        W2, b2 = self.params["W2"], self.params["b2"]
        N, _ = X.shape

        #############################################################################
        # T5: Perform the forward pass, computing the class scores for the input.   #
        # Store the result in the scores variable, which should be an array of      #
        # shape (N, C). Note that this does not include the softmax                 #
        # HINT: This is just a series of matrix multiplication.                     #
        #############################################################################

        # Compute the forward pass
        X1 = X @ W1 + b1
        X1 = np.maximum(0, X1)
        scores = X1 @ W2 + b2
```

```
#############################################################################
#                                END OF T5                                  #
#############################################################################

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

#############################################################################
# T6: Finish the forward pass, and compute the loss. This should include    #
# both the data loss and L2 regularization for W1 and W2. Store the result #
# in the variable loss, which should be a scalar. Use the Softmax          #
# classifier loss.                                                          #
#############################################################################

# Compute the loss
scores_shifted  = scores - np.max(scores, axis=1, keepdims=True)
scores_softmax  = np.exp(scores_shifted)
scores_softmax /= np.sum(scores_softmax, axis=1, keepdims=True)

loss  = -np.sum(np.log(scores_softmax[np.arange(N), y]))
loss /= N
loss += 1/2 * reg * ((W1 * W1).sum() + (W2 * W2).sum())

#############################################################################
#                                END OF T6                                  #
#############################################################################


#############################################################################
# T7: Compute the backward pass, computing derivatives of the weights      #
# and biases. Store the results in the grads dictionary. For example,      #
# grads["W1"] should store the gradient on W1, and be a matrix of same     #
# size don't forget about the regularization term                           #
#############################################################################

# Backward Pass: Compute Gradients
grads = {}

dscores = scores_softmax
dscores[np.arange(N), y] -= 1
dscores /= N

grads["W2"] = X1.T @ dscores + reg * W2
grads["b2"] = np.sum(dscores, axis=0)

dX1 = dscores @ W2.T
```

```
  dX1[X1 <= 0] = 0

  grads["W1"] = X.T @ dX1 + reg * W1
  grads["b1"] = np.sum(dX1, axis=0)

  ############################################################################
  #                              END OF T7                                   #
  ############################################################################

  return loss, grads


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
):
  """
  Train this neural network using stochastic gradient descent.

  Inputs:
  - X: A numpy array of shape (N, D) giving training data.
  - y: A numpy array of shape (N,) giving training labels; y[i] = c means that
    X[i] has label c, where 0 <= c < C.
  - X_val: A numpy array of shape (N_val, D) giving validation data.
  - y_val: A numpy array of shape (N_val,) giving validation labels.
  - learning_rate: Scalar giving learning rate for optimization.
  - learning_rate_decay: Scalar giving factor used to decay the learning rate
    after each epoch.
  - reg: Scalar giving regularization strength.
  - num_iters: Number of steps to take when optimizing.
  - batch_size: Number of training examples to use per step.
  - verbose: boolean; if true print progress during optimization.
  """

  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):
  #########################################################################
  # T8: Create a random minibatch of training data and labels, storing    #
  # them in X_batch and y_batch respectively.                             #
  # You might find np.random.choice() helpful.                            #
  #########################################################################

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

  #########################################################################
  #                             END OF YOUR T8                            #
  #########################################################################

  # Compute loss and gradients using the current minibatch
  loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
  loss_history.append(loss)

  #########################################################################
  # T9: Use the gradients in the grads dictionary to update the           #
  # parameters of the network (stored in the dictionary self.params)      #
  # using stochastic gradient descent. You'll need to use the gradients   #
  # stored in the grads dictionary defined above.                         #
  #########################################################################

  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 T9                            #
  #########################################################################

  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)
```

```
      ######################################################################
      # T10: Decay learning rate (exponentially) after each epoch          #
      ######################################################################

      # Decay learning rate
      learning_rate *= learning_rate_decay

      ######################################################################
      #                           END OF YOUR T10                          #
      ######################################################################


    return {
      "loss_history": loss_history,
      "train_acc_history": train_acc_history,
      "val_acc_history": val_acc_history,
    }


  def predict(self, X):
    """
    Use the trained weights of this two-layer network to predict labels for
    data points. For each data point we predict scores for each of the C
    classes, and assign each data point to the class with the highest score.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.

    Returns:
    - y_pred: A numpy array of shape (N,) giving predicted labels for each of
      the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
      to have class c, where 0 <= c < C.
    """

    ##########################################################################
    # T11: Implement this function; it should be VERY simple!                #
    ##########################################################################

    scores = np.array(self.loss(X))
    y_pred = np.argmax(scores, axis=1)

    ##########################################################################
    #                             END OF YOUR T11                            #
    ##########################################################################
    return y_pred
```

---

2 data_utils.py

[3]:
```
import os
import pandas as pd

from glob import glob
from functools import reduce
from sklearn.model_selection import train_test_split

article_types = ["article", "encyclopedia", "news", "novel"]
```

[ ]:
```
def generate_words(files):
    """
    Transform list of files to list of words,
    removing new line character
    and replace name entity "<NE>...</NE>" and abbreviation "<AB>...</AB>"␣
  ↪symbol
    """

    repls = {"<NE>" : "","</NE>" : "","<AB>": "","</AB>": ""}

    words_all = []
    for _, file in enumerate(files):
        lines = open(file, "r")
        for line in lines:
            line = reduce(lambda a, kv: a.replace(*kv), repls.items(), line)
            words = [word for word in line.split("|") if word is not "\n"]
            words_all.extend(words)

    return words_all
```

[5]:
```
def create_char_dataframe(words):
    """
    Give list of input tokenized words,
    create dataframe of characters where first character of
    the word is tagged as 1, otherwise 0
    Example
    =======
    [" ", "  "] to dataframe of
    [{"char": " ", "type": ..., "target": 1}, ...,
     {"char": " ", "type": ..., "target": 0}]
    """
    char_dict = []
    for word in words:
        for i, char in enumerate(word):
            if i == 0:
                char_dict.append({"char": char, "target": True})
            else:
                char_dict.append({"char": char, "target": False})

    return pd.DataFrame(char_dict)
```

```
[6]: def generate_best_dataset(best_path, output_path="cleaned_data",␣
     ↪create_val=False):
         """
         Generate CSV file for training and testing data
         Input
         =====
         best_path: str, path to BEST folder which contains unzipped subfolder
             "article", "encyclopedia", "news", "novel"
         cleaned_data: str, path to output folder, the cleaned data will be saved
             in the given folder name where training set will be stored in `train`␣
     ↪folder
             and testing set will be stored on `test` folder
         create_val: boolean, True or False, if True, divide training set into␣
     ↪training set and
             validation set in `val` folder
         """

         if not os.path.isdir(output_path):
             os.mkdir(output_path)

         if not os.path.isdir(os.path.join(output_path, "train")):
             os.makedirs(os.path.join(output_path, "train"))

         if not os.path.isdir(os.path.join(output_path, "test")):
             os.makedirs(os.path.join(output_path, "test"))

         if not os.path.isdir(os.path.join(output_path, "val")) and create_val:
             os.makedirs(os.path.join(output_path, "val"))

         for article_type in article_types:
             files = glob(os.path.join(best_path, article_type, "*.txt"))
             files_train, files_test = train_test_split(files, random_state=0,␣
     ↪test_size=0.1)

             if create_val:
```

```
            files_train, files_val = train_test_split(files_train,␣
↪random_state=0, test_size=0.1)
            val_words = generate_words(files_val)
            val_df = create_char_dataframe(val_words)
            val_df.to_csv(os.path.join(output_path, "val", "df_best_{}_val.csv".
↪format(article_type)), index=False)

        train_words = generate_words(files_train)
        test_words = generate_words(files_test)
        train_df = create_char_dataframe(train_words)
        test_df = create_char_dataframe(test_words)

        train_df.to_csv(os.path.join(output_path, "train", "df_best_{}_train.
↪csv".format(article_type)), index=False)
        test_df.to_csv(os.path.join(output_path, "test", "df_best_{}_test.csv".
↪format(article_type)), index=False)

        print("Save {} to CSV file".format(article_type))
```

---

3 gradient_check.py

[7]:
```
from __future__ import print_function

import numpy as np
from random import randrange
```

[8]:
```
def eval_numerical_gradient(f, x, verbose=True, h=0.00001):
  """
  A naive implementation of numerical gradient of f at x.
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  # Evaluate function value at original point.
  fx = f(x)
  grad = np.zeros_like(x)

  # Iterate over all indexes in x.
  it = np.nditer(x, flags=["multi_index"], op_flags=[["readwrite"]])

  while not it.finished:
    # Evaluate function at x + h.
    ix = it.multi_index
    oldval = x[ix]
```

```
      # Increment by h.
      x[ix] = oldval + h

      # Evalute f(x + h).
      fxph = f(x)
      x[ix] = oldval - h

      # Evalute f(x - h).
      fxmh = f(x)

      # Restore the original value.
      x[ix] = oldval

      # Compute the partial derivative with centered formula.
      grad[ix] = (fxph - fxmh) / (2 * h) # The slope

      if verbose:
        print(ix, grad[ix])

      # Step to next dimension
      it.iternext()

    return grad
```

[9]:
```
def eval_numerical_gradient_array(f, x, df, h=1e-5):
    """
    Evaluate a numeric gradient for a function that accepts a numpy
    array and returns a numpy array.
    """

    grad = np.zeros_like(x)
    it = np.nditer(x, flags=["multi_index"], op_flags=[["readwrite"]])

    while not it.finished:
      ix = it.multi_index

      oldval = x[ix]
      x[ix] = oldval + h

      pos = f(x).copy()
      x[ix] = oldval - h

      neg = f(x).copy()
      x[ix] = oldval

      grad[ix] = np.sum((pos - neg) * df) / (2 * h)
      it.iternext()

    return grad
```

[10]:
```
def eval_numerical_gradient_blobs(f, inputs, output, h=1e-5):
  """
  Compute numeric gradients for a function that operates on input
  and output blobs.

  We assume that f accepts several input blobs as arguments, followed by a blob
  into which outputs will be written. For example, f might be called like this:

  f(x, w, out)

  where x and w are input Blobs, and the result of f will be written to out.

  Inputs:
  - f: function
  - inputs: tuple of input blobs
  - output: output blob
  - h: step size
  """
  numeric_diffs = []
  for input_blob in inputs:
    diff = np.zeros_like(input_blob.diffs)

    it = np.nditer(input_blob.vals, flags=["multi_index"],␣
↪op_flags=[["readwrite"]])

    while not it.finished:
      idx = it.multi_index
      orig = input_blob.vals[idx]

      input_blob.vals[idx] = orig + h

      f(*(inputs + (output,)))

      pos = np.copy(output.vals)
      input_blob.vals[idx] = orig - h

      f(*(inputs + (output,)))

      neg = np.copy(output.vals)
      input_blob.vals[idx] = orig

      diff[idx] = np.sum((pos - neg) * output.diffs) / (2.0 * h)
      it.iternext()

    numeric_diffs.append(diff)

  return numeric_diffs
```

[11]:
```
def eval_numerical_gradient_net(net, inputs, output, h=1e-5):
  return eval_numerical_gradient_blobs(lambda *args: net.forward(), inputs,␣
↪output, h=h)
```

[12]:
```
def grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-5):
  """
  Sample a few random elements and only return numerical
  gradients in these dimensions.
  """

  for _ in range(num_checks):
    ix = tuple([randrange(m) for m in x.shape])

    oldval = x[ix]

    # Increment by h.
    x[ix] = oldval + h

    # Evaluate f(x + h).
    fxph = f(x)

    # Decrement by h.
    x[ix] = oldval - h

    # Evaluate f(x - h).
    fxmh = f(x)

    # Restore the original value.
    x[ix] = oldval

    grad_numerical = (fxph - fxmh) / (2 * h)
    grad_analytic = analytic_grad[ix]
    rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +␣
↪abs(grad_analytic))

    print("numerical: %f analytic: %f, relative error: %e" % (grad_numerical,␣
↪grad_analytic, rel_error))
```

APPENDIX 3

Homework 4 Template Code in `MNIST_DATA` Directory

mnist_data

February 28, 2026

1 load_mnist.py

[1]:
```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy
import gzip
```

[2]:
```
def _read32(bytestream):
  dt = numpy.dtype(numpy.uint32).newbyteorder(">")
  return numpy.frombuffer(bytestream.read(4), dtype=dt)[0]
```

[3]:
```
def extract_images(f):
  """Extract the images into a 4D uint8 numpy array [index, y, x, depth].
  Args:
    f: A file object that can be passed into a gzip reader.
  Returns:
    data: A 4D uint8 numpy array [index, y, x, depth].
  Raises:
    ValueError: If the bytestream does not start with 2051.
  """

  print("Extracting", f.name)

  with gzip.GzipFile(fileobj=f) as bytestream:
    magic = _read32(bytestream)

    if magic != 2051:
      raise ValueError("Invalid magic number %d in MNIST image file: %s" %␣
↪(magic, f.name))

    num_images = _read32(bytestream)
    rows = _read32(bytestream)
    cols = _read32(bytestream)

    buf = bytestream.read(rows * cols * num_images)
```

1

```
    data = numpy.frombuffer(buf, dtype=numpy.uint8)
    data = data.reshape(num_images, rows, cols, 1)

    return data
```

[4]:
```
def dense_to_one_hot(labels_dense, num_classes):
  """Convert class labels from scalars to one-hot vectors."""

  num_labels = labels_dense.shape[0]
  index_offset = numpy.arange(num_labels) * num_classes

  labels_one_hot = numpy.zeros((num_labels, num_classes))
  labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1

  return labels_one_hot
```

[5]:
```
def extract_labels(f, one_hot=False, num_classes=10):
  """Extract the labels into a 1D uint8 numpy array [index].
  Args:
    f: A file object that can be passed into a gzip reader.
    one_hot: Does one hot encoding for the result.
    num_classes: Number of classes for the one hot encoding.
  Returns:
    labels: a 1D uint8 numpy array.
  Raises:
    ValueError: If the bystream doesn't start with 2049.
  """

  print("Extracting", f.name)

  with gzip.GzipFile(fileobj=f) as bytestream:
    magic = _read32(bytestream)

    if magic != 2049:
      raise ValueError("Invalid magic number %d in MNIST label file: %s" %␣
↪(magic, f.name))

    num_items = _read32(bytestream)

    buf = bytestream.read(num_items)

    labels = numpy.frombuffer(buf, dtype=numpy.uint8)

    if one_hot:
      return dense_to_one_hot(labels, num_classes)
    return labels
```

[6]:
```
def read_data_sets(
  train_dir="mnist_data",
  one_hot=False,
  reshape=True,
  validation_size=5000,
  seed=None
):
  """Reads and parses examples from MNIST data files."""

  TRAIN_IMAGES = "train-images-idx3-ubyte.gz"
  TRAIN_LABELS = "train-labels-idx1-ubyte.gz"
  TEST_IMAGES = "t10k-images-idx3-ubyte.gz"
  TEST_LABELS = "t10k-labels-idx1-ubyte.gz"

  local_file = train_dir + "/" + TRAIN_IMAGES

  with open(local_file, "rb") as f:
    train_images = extract_images(f)

  local_file = train_dir + "/" + TRAIN_LABELS
  with open(local_file, "rb") as f:
    train_labels = extract_labels(f, one_hot=one_hot)

  local_file = train_dir + "/" + TEST_IMAGES
  with open(local_file, "rb") as f:
    test_images = extract_images(f)

  local_file = train_dir + "/" + TEST_LABELS
  with open(local_file, "rb") as f:
    test_labels = extract_labels(f, one_hot=one_hot)

  if not 0 <= validation_size <= len(train_images):
    raise ValueError(
        "Validation size should be between 0 and {}. Received: {}."
        .format(len(train_images), validation_size))

  validation_images = train_images[:validation_size]
  validation_labels = train_labels[:validation_size]
  train_images = train_images[validation_size:]
  train_labels = train_labels[validation_size:]

  return train_images, train_labels, validation_images, validation_labels,␣
 ↪test_images, test_labels
```

---

2 vis_utils.py

[7]:
```
import numpy as np
from math import sqrt, ceil
```

[8]:
```
def visualize_grid(Xs, ubound=255.0, padding=1):
  """
  Reshape a 4D tensor of image data to a grid for easy visualization.

  Inputs:
  - Xs: Data of shape (N, H, W, C)
  - ubound: Output grid will have values scaled to the range [0, ubound]
  - padding: The number of blank pixels between elements of the grid
  """

  (N, H, W, C) = Xs.shape

  grid_size = int(ceil(sqrt(N)))
  grid_height = H * grid_size + padding * (grid_size - 1)
  grid_width = W * grid_size + padding * (grid_size - 1)
  grid = np.zeros((grid_height, grid_width, C))

  next_idx = 0
  y0, y1 = 0, H
  for y in range(grid_size):
    x0, x1 = 0, W
    for x in range(grid_size):
      if next_idx < N:
        img = Xs[next_idx]
        low, high = np.min(img), np.max(img)

        grid[y0:y1, x0:x1] = ubound * (img - low) / (high - low)

        next_idx += 1

      x0 += W + padding
      x1 += W + padding

    y0 += H + padding
    y1 += H + padding

  return grid
```

[ ]:
```
def vis_grid(Xs):
  """ visualize a grid of images """

  (N, H, W, C) = Xs.shape
```

```
    A = int(ceil(sqrt(N)))
    G = np.ones(((A * H) + A, (A * W) + A, C), Xs.dtype)
    G *= np.min(Xs)
    n = 0

    for y in range(A):
      for x in range(A):
        if n < N:
          G[
            (y * H) + y: ((y + 1) * H) + y,
            (x * W) + x: ((x + 1) * W) + x,
            :
          ] = Xs[n, :, :, :]
          n += 1

    maxg = G.max()
    ming = G.min()

    G = (G - ming) / (maxg - ming)

    return G
```

[ ]:
```
def vis_nn(rows):
    """ visualize array of arrays of images """

    N = len(rows)
    D = len(rows[0])
    H, W, C = rows[0][0].shape
    Xs = rows[0][0]

    G = np.ones(((N * H) + N, (D * W) + D, C), Xs.dtype)
    for y in range(N):
      for x in range(D):
        G[
          (y * H) + y: ((y + 1) * H) + y,
          (x * W) + x: ((x + 1) * W) + x,
          :
        ] = rows[y][x]

    maxg = G.max()
    ming = G.min()

    G = (G - ming) / (maxg - ming)

    return G
```
