softmax

November 16, 2022

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs260/assignments/assignment4/'
FOLDERNAME = 'cs260/assignments/assignment4/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs260/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs260/assignments/assignment4/cs260/datasets
/content/drive/My Drive/cs260/assignments/assignment4
```

1 Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission.

In this exercise you will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**

- **visualize** the final learned weights

[17]:
```
import random
import numpy as np
from cs260.data_utils import load_CIFAR10
import matplotlib.pyplot as plt



%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'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[18]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
 ↪num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs260/datasets/cifar-10-batches-py'



    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
```

```
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev =␣
 ↪get_CIFAR10_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)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

Your code for this section will all be written inside `cs260/classifiers/softmax.py`.

[20]:
```
# First implement the naive softmax loss function with nested loops.
# Open the file cs260/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs260.classifiers.softmax import softmax_loss_naive
import time
# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.354812
sanity check: 2.302585
```

Inline Question 1

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

Your Answer : *Since CIFAR-10 consists of samples which belong to one of ten classes, the probability of the correct class will be $1/10 = 0.1$. The softmax loss is the negative log probability of the correct class, therefore it is -log(0.1).

-

[21]:
```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# Use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs260.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# Do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 0.337641 analytic: 0.337641, relative error: 5.230949e-08
numerical: -0.504864 analytic: -0.504864, relative error: 7.229281e-08
numerical: -0.737189 analytic: -0.737189, relative error: 3.805613e-08
numerical: 0.529677 analytic: 0.529677, relative error: 3.055796e-08
```

```
numerical: 0.117159 analytic: 0.117159, relative error: 5.673477e-07
numerical: -1.701997 analytic: -1.701997, relative error: 3.828592e-08
numerical: -0.518785 analytic: -0.518785, relative error: 1.251077e-07
numerical: 3.189274 analytic: 3.189274, relative error: 1.250522e-08
numerical: -0.410828 analytic: -0.410828, relative error: 3.006575e-08
numerical: -5.221586 analytic: -5.221586, relative error: 9.604616e-09
numerical: 1.258206 analytic: 1.258206, relative error: 3.986848e-08
numerical: -0.161458 analytic: -0.161458, relative error: 2.468905e-07
numerical: -3.762994 analytic: -3.762994, relative error: 3.450950e-09
numerical: 1.525461 analytic: 1.525461, relative error: 4.152747e-08
numerical: 3.047994 analytic: 3.047993, relative error: 2.567302e-08
numerical: 3.533032 analytic: 3.533032, relative error: 4.247571e-09
numerical: -0.017505 analytic: -0.017505, relative error: 5.396662e-07
numerical: -0.583828 analytic: -0.583828, relative error: 4.614186e-09
numerical: 0.470189 analytic: 0.470189, relative error: 2.869188e-08
numerical: 5.577975 analytic: 5.577975, relative error: 9.788597e-09
```

[22]:
```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs260.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# We use the Frobenius norm to compare the two versions of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.354812e+00 computed in 37.043528s
vectorized loss: 2.354812e+00 computed in 0.017367s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[23]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
```

```
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

from cs260.classifiers.linear_classifier import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################

# Provided as a reference. You may or may not want to change these␣
↪hyperparameters
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
grid_search = [(learning_rate, regularization_strength) for learning_rate in␣
↪learning_rates \
               for regularization_strength in regularization_strengths]
for learning_rate, regularization_strength in grid_search:

    softmax = Softmax()

    # train a linear SVM on the training set
    loss_hist = softmax.train(X_train, y_train, learning_rate=learning_rate,
                              reg=regularization_strength, num_iters=1500,␣
↪verbose=False)

    # make predictions on the training and validation sets
    y_train_pred = softmax.predict(X_train)
    y_val_pred = softmax.predict(X_val)

    # compute the accuracy on the training and validation sets
    current_y_train_accuracy = np.mean(y_train_pred == y_train)
    current_y_val_accuracy = np.mean(y_val_pred == y_val)

    # store results
    results[(learning_rate, regularization_strength)] = \
    (current_y_train_accuracy, current_y_val_accuracy)

    # store the best validation accuracy and the LinearSVM object
    if current_y_val_accuracy > best_val:
```

```
        best_val = current_y_val_accuracy
        best_softmax = softmax

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

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' %␣
↪best_val)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.350980 val accuracy: 0.365000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.326245 val accuracy: 0.344000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.350755 val accuracy: 0.348000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.327204 val accuracy: 0.334000
best validation accuracy achieved during cross-validation: 0.365000
```

[24]:
```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.352000
```

Inline Question 2 - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

YourAnswer : TRUE

YourExplanation :

When using SVM ,as long as the new datapoint is labeled correctly, the loss will be 0 and leaving the loss unchanged. However when using softmax classifier loss, no matter if it is labeled correct, there will always we a loss. Even if the loss is very small, the loss still changed overall.

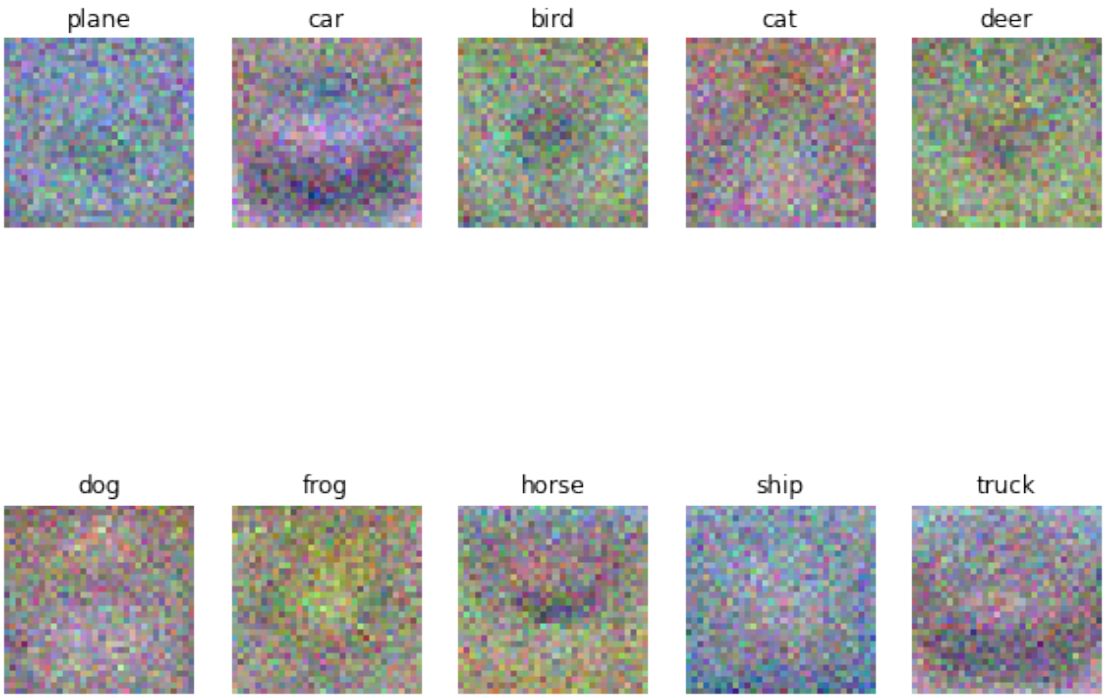
[25]:
```
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

w_min, w_max = np.min(w), np.max(w)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
```

```
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



[ ]:

two_layer_net

November 16, 2022

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs260/assignments/assignment4/'
FOLDERNAME = 'cs260/assignments/assignment4/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs260/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs260/assignments/assignment4/cs260/datasets
/content/drive/My Drive/cs260/assignments/assignment4
```

1 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:

```
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:

```
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.

[10]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs260.classifiers.fc_net import *
from cs260.data_utils import get_CIFAR10_data
from cs260.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs260.solver import Solver

%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'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

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))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[11]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('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,))
```

2 Affine layer: forward

Open the file `cs260/layers.py` and implement the `affine_forward` function.

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

[12]:
```
# Test the affine_forward function

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

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

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

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

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

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

3 Affine layer: backward

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

[13]:
```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x,␣
↪dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w,␣
↪dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b,␣
↪dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

4 ReLU activation: forward

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

[14]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)
```

```
out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

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

5 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:

[15]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

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

5.1 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. Tanh

5.2 Answer:

1) sigmoid :- . The main reason why we use the sigmoid function is that it exist between(0 to1). . therefore ,it is especially useful for models where we have to predict the probability as an output. . since the probability of anything exist only between the range of 0 and 1, sigmoid is the right choice. . The sigmoid function plays an important role in the field of machine learning and is considered as one of the most widely used so-called activation functions . More specifically in the contents of Logistic regression the signal is used to predict the outcome of binary classification problems. . It is the difference table real function define for real input values and containing positive derivatives everywhere with a specific degree of smoothness . The sigmoid function appears in the output layer of the deep learning models and is used for predicting probability based outputs . The sigmoid function is represented as A= 1/1+e–x . The graph of the sigmoid function is 'S' shape . It is non linear in nature . We can stack layer . Combination of this function are also non linear . It has a smooth gradient too

*pros : - smooth gradient preventing jumps in output values - output values bond between 0 and 1 normalising the output of each neurone therefore it is especially useful for models where we have to predict the probability as an output S . Clear prediction . Unlike linear function the output of the activation function is always going to be in range(0 ,1) compare to (-inf,inf) of linear function So we have our activation bound in a range

*Cons : - vanishing gradient: for very hai aur very low values of X there is almost no change to the prediction causing advising gradient problem this can result in the network refusing to learn further or being too slow to reach an accurate prediction. . Outputs not zero centred . Computationally expensive.

2)ReLu (Rectified linear unit) .ReLu activation function is one of the most popular activation functions for deep learning and convolutional neural networks. . However the function itself is deceptively simple. . The rectified linear activation function or Relu for short is a piecewise linear function that will output the input directly if it is positive otherwise it will output 0 . The rectified linear activation function overcomes the vanishing gradient problem allowing models to learn faster and perform better

*Pros : - allows to avoid vanishing gradient problem _ computationally efficient - it is a less computationally expensive than sigmoid

*Cons : - the dying relu problem when input approach zero or negative the gradient of the function becomes zero the network cannot perform backpropagation and cannot learn - one of its limitation is that it should only be used within hidden layers of a neural network model

3)leaky ReLu :

. Leaky ReLu is one attempt to fix the dying ReLu or dead neurones problems. . Instead of the function being zero when X<0,a leaky ReLu will instead have a small negative slope(of 0.01.or so) . That is the function computes f(X)=1(X<0)(ax)+1(X>=0) . Some people report success with this form of activation function but the result are not always consistent.

*Cons : - Leaky relu does not provide consistent predictions for negative input values

*Pros : - allows the negative slope to be learn otherwise like ReLu.

6 "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:

[16]:
```
from cs260.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w,␣
 ↪b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w,␣
 ↪b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w,␣
 ↪b)[0], b, dout)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

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

7 Loss layer: Softmax

Now implement the loss and gradient for softmax in the `softmax_loss` function in `cs260/layers.py`. This should be similar to what you implemented in `cs260/classifiers/softmax.py`.

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

[17]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
```

```
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x,␣
↪verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should␣
↪be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

8 Two-layer network

Open the file `cs260/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.

[18]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
```

```
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 7.98e-08
b1 relative error: 1.56e-08
```

```
b2 relative error: 7.76e-10
```

9 Solver

Open the file `cs260/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 `30%` accuracy on the validation set.

[22]:
```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 30% #
# accuracy on the validation set. Use SGD as your update rule and a learning #
# rate of 1e-5. (Everything else should be default parameters.)                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

model = TwoLayerNet(hidden_dim=500, reg = 0)
solver = Solver(model, data,update_rule='sgd', optim_config={'learning_rate':␣
↪1e-3,},lr_decay=0.95, num_epochs=10, batch_size=200,
                print_every=100)
solver.train()

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

```
(Iteration 1 / 2450) loss: 2.310175
(Epoch 0 / 10) train acc: 0.156000; val_acc: 0.166000
(Iteration 101 / 2450) loss: 1.807931
(Iteration 201 / 2450) loss: 1.622035
(Epoch 1 / 10) train acc: 0.444000; val_acc: 0.464000
(Iteration 301 / 2450) loss: 1.603373
(Iteration 401 / 2450) loss: 1.470298
(Epoch 2 / 10) train acc: 0.482000; val_acc: 0.461000
(Iteration 501 / 2450) loss: 1.421997
(Iteration 601 / 2450) loss: 1.416555
(Iteration 701 / 2450) loss: 1.410559
(Epoch 3 / 10) train acc: 0.555000; val_acc: 0.508000
(Iteration 801 / 2450) loss: 1.368212
(Iteration 901 / 2450) loss: 1.257052
(Epoch 4 / 10) train acc: 0.552000; val_acc: 0.509000
```

```
(Iteration 1001 / 2450) loss: 1.457195
(Iteration 1101 / 2450) loss: 1.492640
(Iteration 1201 / 2450) loss: 1.296773
(Epoch 5 / 10) train acc: 0.562000; val_acc: 0.522000
(Iteration 1301 / 2450) loss: 1.268756
(Iteration 1401 / 2450) loss: 1.131604
(Epoch 6 / 10) train acc: 0.587000; val_acc: 0.496000
(Iteration 1501 / 2450) loss: 1.145399
(Iteration 1601 / 2450) loss: 1.042058
(Iteration 1701 / 2450) loss: 1.095024
(Epoch 7 / 10) train acc: 0.580000; val_acc: 0.496000
(Iteration 1801 / 2450) loss: 1.305515
(Iteration 1901 / 2450) loss: 1.158958
(Epoch 8 / 10) train acc: 0.632000; val_acc: 0.531000
(Iteration 2001 / 2450) loss: 1.266857
(Iteration 2101 / 2450) loss: 1.004632
(Iteration 2201 / 2450) loss: 0.897495
(Epoch 9 / 10) train acc: 0.621000; val_acc: 0.532000
(Iteration 2301 / 2450) loss: 1.090900
(Iteration 2401 / 2450) loss: 1.061330
(Epoch 10 / 10) train acc: 0.652000; val_acc: 0.522000
```

10 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.30 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.

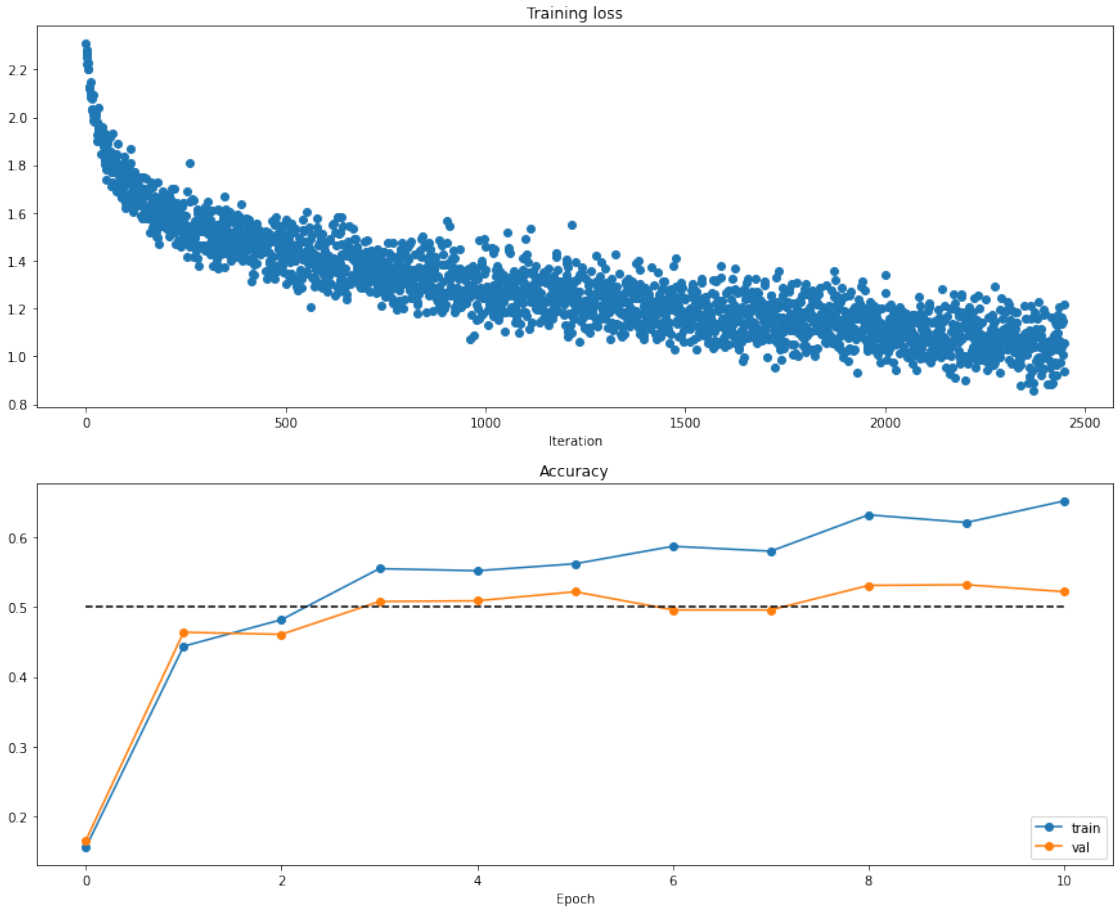
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[23]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
```

```
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```

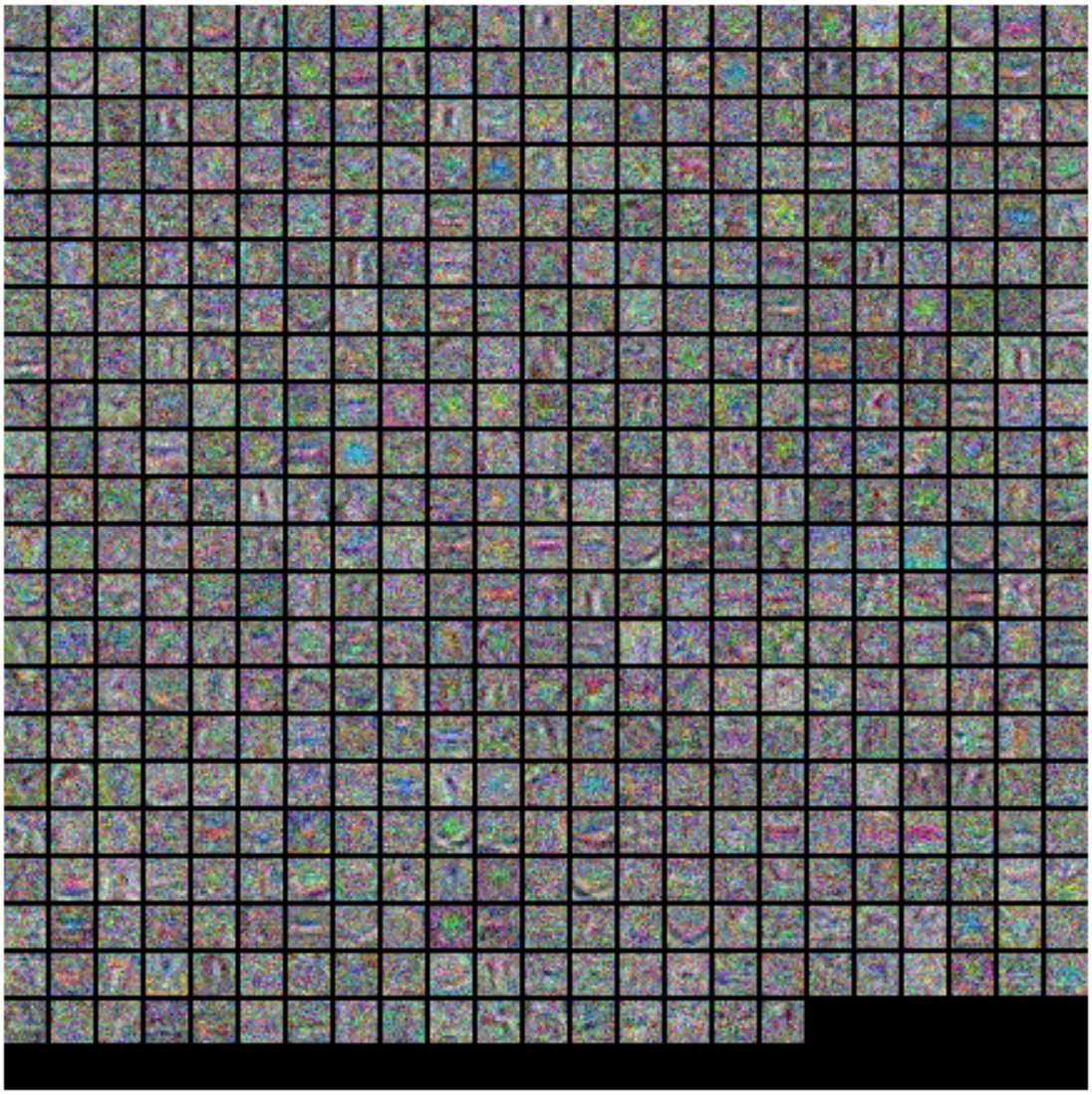


[24]:
```
from cs260.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(3, 32, 32, -1).transpose(3, 1, 2, 0)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



11 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.).

[43]:
```
best_model = None


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained␣
↪#
# model in best_model.                                                         ␣
↪#
#                                                                              ␣
↪#
# 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 thexs previous exercises.                       ␣
↪  #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

results = {}
best_val = -1

learning_rates = np.geomspace(3e-4, 3e-2, 3)
regularization_strengths = np.geomspace(1e-6, 1e-2, 5)

import itertools

for lr, reg in itertools.product(learning_rates, regularization_strengths):
```

```
    # Create Two Layer Net and train it with Solver
    model = TwoLayerNet(hidden_dim=128, reg=reg)
    solver = Solver(model, data, optim_config={'learning_rate': lr},␣
 ↪num_epochs=10, verbose=False)
    solver.train()

    # Compute validation set accuracy and append to the dictionary
    results[(lr, reg)] = solver.best_val_acc

    # Save if validation accuracy is the best
    if results[(lr, reg)] > best_val:
        best_val = results[(lr, reg)]
        best_model = model

# Print out results.
for lr, reg in sorted(results):
    val_accuracy = results[(lr, reg)]
    print('lr %e reg %e val accuracy: %f' % (lr, reg, val_accuracy))

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

```
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:86:
RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/content/drive/My Drive/cs260/assignments/assignment4/cs260/layers.py:994:
RuntimeWarning: overflow encountered in subtract
  shifted_logits = x - np.max(x, axis=1, keepdims=True)
/content/drive/My Drive/cs260/assignments/assignment4/cs260/layers.py:994:
RuntimeWarning: invalid value encountered in subtract
  shifted_logits = x - np.max(x, axis=1, keepdims=True)
```

```
lr 3.000000e-04 reg 1.000000e-06 val accuracy: 0.519000
lr 3.000000e-04 reg 1.000000e-05 val accuracy: 0.529000
lr 3.000000e-04 reg 1.000000e-04 val accuracy: 0.521000
lr 3.000000e-04 reg 1.000000e-03 val accuracy: 0.521000
lr 3.000000e-04 reg 1.000000e-02 val accuracy: 0.527000
lr 3.000000e-03 reg 1.000000e-06 val accuracy: 0.336000
lr 3.000000e-03 reg 1.000000e-05 val accuracy: 0.337000
lr 3.000000e-03 reg 1.000000e-04 val accuracy: 0.186000
lr 3.000000e-03 reg 1.000000e-03 val accuracy: 0.306000
lr 3.000000e-03 reg 1.000000e-02 val accuracy: 0.372000
lr 3.000000e-02 reg 1.000000e-06 val accuracy: 0.109000
lr 3.000000e-02 reg 1.000000e-05 val accuracy: 0.141000
```

```
lr 3.000000e-02 reg 1.000000e-04 val accuracy: 0.132000
lr 3.000000e-02 reg 1.000000e-03 val accuracy: 0.150000
lr 3.000000e-02 reg 1.000000e-02 val accuracy: 0.152000
best validation accuracy achieved during cross-validation: 0.529000
```

12 Test your model!

Run your best model on the validation and test sets. You should achieve above 45% accuracy on the validation set and the test set.

[44]:
```
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.529
```

[45]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.523
```

12.1 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.

YourAnswer :

1 and 3

YourExplanation :

The gap is caused by the hyperparameters overfitting the validation set. To reduce this gap, we can increase the training size or increase regularization strength to prevent overfitting. In addition, we can use cross-validation instead of a single validation set to tune hyperparameters.

[ ]: