

# knn

January 7, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment1/'
FOLDERNAME = 'icv83551/assignments/assignment1/'
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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
[2]: !sh icv83551/datasets/get_datasets.sh
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload Upload	Total	Spent	Left	Speed
100	162M	100	162M	0	0	11.2M	0
0:00:14	0:00:14	0:00:14	0:00:14	0:00:14	0:00:14	0:00:14	0:00:14
8360k	0:00:06	0:00:07	15.6M	11.2M	0	0:00:14	0:00:14
0:00:14	0:00:14	0:00:14	0:00:14	0:00:14	0:00:14	0:00:14	0:00:14
8754k							

```
curl: (6) Could not resolve host: cifar-10-python.tar.gz
x cifar-10-batches-py/
x cifar-10-batches-py/data_batch_4
x cifar-10-batches-py/readme.html
x cifar-10-batches-py/test_batch
x cifar-10-batches-py/data_batch_3
x cifar-10-batches-py/batches.meta
x cifar-10-batches-py/data_batch_2
x cifar-10-batches-py/data_batch_5
x cifar-10-batches-py/data_batch_1
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
```

```
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a
href="https://cs231n.stanford.edu/imagenet_val_25.npz">here</a>.</p>
</body></html>
```

## 1 k-Nearest Neighbor (kNN) exercise

\*Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transferring the labels of the k most similar training examples
- The value of k is cross-validated

In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

```
[ ]: # Run some setup code for this notebook.

import random
import numpy as np
from icv83551.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the
↪notebook
# rather than in a new window.
%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'

# %load_ext autoreload
# %autoreload 2

[2]: # Load the raw CIFAR-10 data.
cifar10_dir = 'icv83551/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause
↪memory issue)
try:
    del X_train, y_train
    del X_test, y_test
```

```

    print('Clear previously loaded data.')
except:
    pass

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

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)

```

```

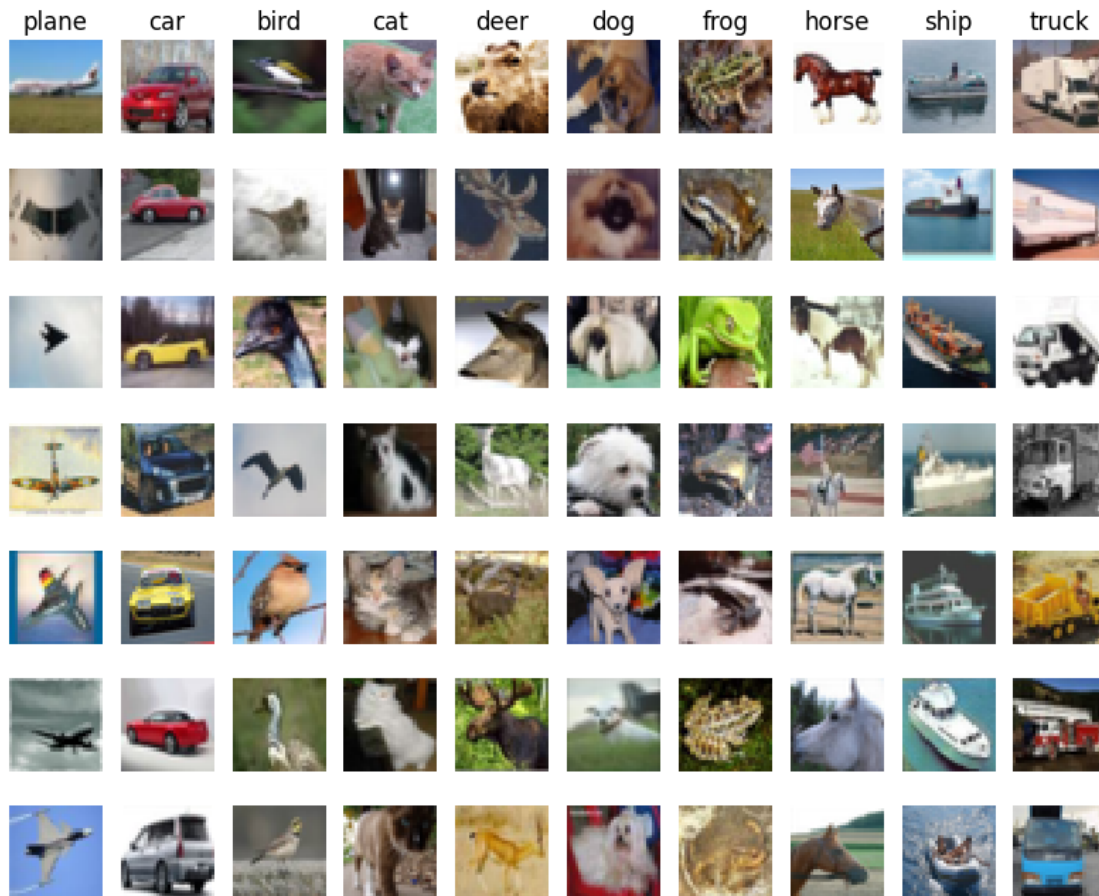
Training data shape: (50000, 32, 32, 3)
Training labels shape: (50000,)
Test data shape: (10000, 32, 32, 3)
Test labels shape: (10000,)

```

```

[3]: # Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', '
↳ 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
    if i == 0:
        plt.title(cls)
plt.show()

```



```
[4]: # Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

(5000, 3072) (500, 3072)

```
[5]: from icv83551.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# 'noop' stands for "no operation": in this context, training a kNN classifier
# does not perform any real learning step, it simply stores the labeled data
# so it can be used later to find nearest neighbors during prediction.

# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

Lets begin with computing the distance matrix between all training and test examples. For example, if there are  $N_{tr}$  training examples and  $N_{te}$  test examples, this stage should result in a  $N_{te} \times N_{tr}$  matrix where each element  $(i,j)$  is the distance between the i-th test and j-th train example.

**Note:** For the three distance computations that we require you to implement in this notebook, you may not use the `np.linalg.norm()` function that numpy provides.

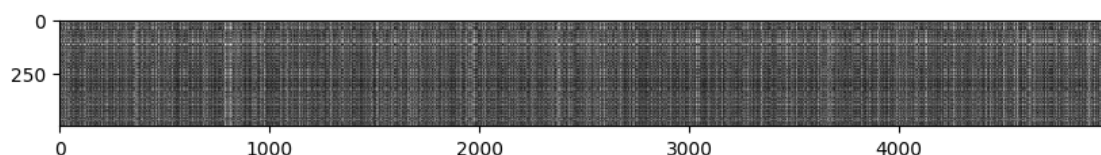
First, open `icv83551/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

```
[6]: # Open icv83551/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

(500, 5000)

```
[7]: # We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



### Inline Question 1

Notice the structured patterns in the distance matrix, where some rows or columns are visibly brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

*Your Answer :* rows represents the distance of a specific test sample to the entire train data and a column represents the distance between a single training sample and the entire test data. with better features or a distance measure, we would expect to see a better, and less consistent distribution of distances. i.e., some training sample should be closer to some test samples and other closer to other test samples, based on the class of each training and test sample. darker row represents a test sample that is generally closer to the train samples distributions (and bright represents samples which are generally not as close to the training data distribution), and vice versa. darker columns represents training samples that are closer to the test samples distribution. We can assume our dataset has some common structure, such that samples that follows this structure gets small distance scores throughout the matrix. For example, such pattern can be a background with a moderate intensity (closer to all background) and an object at the center. a large-distance sample may have an extreme background, such as black or white, and maybe an object at the sides. it is safe to assume our classifier gives very high weight to the background, as it represents most pixels.

```
[8]: # Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

Got 137 / 500 correct => accuracy: 0.274000

You should expect to see approximately 27% accuracy. Now let's try out a larger k, say k = 5:

```
[9]: y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

Got 139 / 500 correct => accuracy: 0.278000

You should expect to see a slightly better performance than with k = 1.

### Inline Question 2

We can also use other distance metrics such as L1 distance. For pixel values  $p_{ij}^{(k)}$  at location  $(i, j)$  of some image  $I_k$ ,

the mean  $\mu$  across all pixels over all images is

$$\mu = \frac{1}{nhw} \sum_{k=1}^n \sum_{i=1}^h \sum_{j=1}^w p_{ij}^{(k)}$$

And the pixel-wise mean  $\mu_{ij}$  across all images is

$$\mu_{ij} = \frac{1}{n} \sum_{k=1}^n p_{ij}^{(k)}.$$

The general standard deviation  $\sigma$  and pixel-wise standard deviation  $\sigma_{ij}$  is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. To clarify, both training and test examples are preprocessed in the same way.

1. Subtracting the mean  $\mu$  ( $\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$ .)
2. Subtracting the per pixel mean  $\mu_{ij}$  ( $\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$ .)
3. Subtracting the mean  $\mu$  and dividing by the standard deviation  $\sigma$ .
4. Subtracting the pixel-wise mean  $\mu_{ij}$  and dividing by the pixel-wise standard deviation  $\sigma_{ij}$ .
5. Rotating the coordinate axes of the data, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

*Your Answer :*

1, 2 and 3

*Your Explanation :*

1: as L1 distance is the absolute distance between pixels, subtracting a constant value from all pixels will not change the L1 scores, which means the performances will not be affected as well.

2. this options is like subtracting a constant image (the mean image) from all the images. it will again not change the per-pixel L1 score between images, thus not changing the overall average L1 score between the images, therefore not effecting the overall performances.
3.  $(a/std - b/std) = (a-b)/std$ . Scaling the entire dataset by a constant value  $(1/std)$  will scale the L1 scores accordingly, but will not change the order, thus will not affect the overall performances.

```
[10]: # Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words,
↪ reshape
```

```

# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')

```

One loop difference was: 0.000000  
 Good! The distance matrices are the same

```

[11]: # Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test, dists)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')

```

No loop difference was: 0.000000  
 Good! The distance matrices are the same

```

[12]: # Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took
    to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

```



```
# You should see significantly faster performance with the fully vectorized_  
↪ implementation!
```

```
# NOTE: depending on what machine you're using,  
# you might not see a speedup when you go from two loops to one loop,  
# and might even see a slow-down.
```

Two loop version took 11.430212 seconds  
One loop version took 11.143710 seconds  
No loop version took 0.096696 seconds

### 1.0.1 Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value  $k = 5$  arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

```
[16]: from tqdm import tqdm  
  
num_folds = 5  
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]  
  
X_train_folds = np.array_split(X_train, num_folds)  
y_train_folds = np.array_split(y_train, num_folds)  
#####  
# TODO: #  
# Split up the training data into folds. After splitting, X_train_folds and #  
# y_train_folds should each be lists of length num_folds, where #  
# y_train_folds[i] is the label vector for the points in X_train_folds[i]. #  
# Hint: Look up the numpy array_split function. #  
#####  
  
# A dictionary holding the accuracies for different values of k that we find  
# when running cross-validation. After running cross-validation,  
# k_to_accuracies[k] should be a list of length num_folds giving the different  
# accuracy values that we found when using that value of k.  
k_to_accuracies = {}  
  
#####  
# TODO: #  
# Perform k-fold cross validation to find the best value of k. For each #  
# possible value of k, run the k-nearest-neighbor algorithm num_folds times, #  
# where in each case you use all but one of the folds as training data and the #  
# last fold as a validation set. Store the accuracies for all fold and all #  
# values of k in the k_to_accuracies dictionary. #  
#####  
classifier = KNearestNeighbor()
```

```

for fold_idx in tqdm(range(num_folds), desc="Cross-validation folds"):
    X_val = X_train_folds[fold_idx]
    y_val = y_train_folds[fold_idx]
    X_train_cv = np.vstack([X_train_folds[i] for i in range(num_folds) if i !=
↪fold_idx])
    y_train_cv = np.hstack([y_train_folds[i] for i in range(num_folds) if i !=
↪fold_idx])

    for k in tqdm(k_choices, desc="K size", leave=False):
        classifier.train(X_train_cv, y_train_cv)
        dists = classifier.compute_distances_no_loops(X_val)
        y_val_pred = classifier.predict_labels(dists, k=k)
        num_correct = np.sum(y_val_pred == y_val)
        accuracy = float(num_correct) / len(y_val)

        if k not in k_to_accuracies:
            k_to_accuracies[k] = []
        k_to_accuracies[k].append(accuracy)

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))

```

```

Cross-validation folds:  0%|          | 0/5 [00:00<?, ?it/s]0.00s - Debugger
warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to
disable this validation.
Cross-validation folds: 100%|         | 5/5 [00:16<00:00,  3.36s/it]

```

```

k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000

```

```

k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000

```

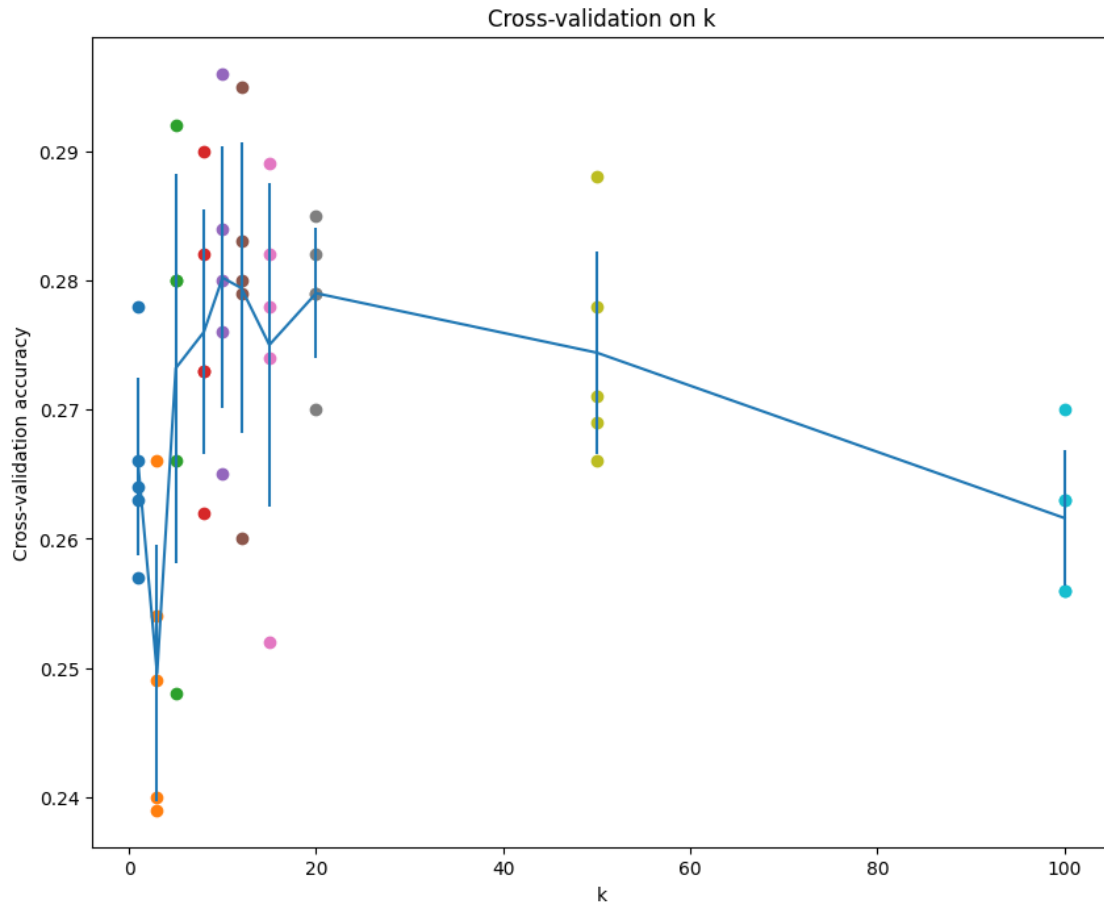
```

[17]: # plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.
    ↪items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.
    ↪items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)

```

```
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



```
[18]: # Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

Got 141 / 500 correct => accuracy: 0.282000

### Inline Question 3

Which of the following statements about  $k$ -Nearest Neighbor ( $k$ -NN) are true in a classification setting, and for all  $k$ ? Select all that apply. 1. The decision boundary of the  $k$ -NN classifier is linear. 2. The training error of a 1-NN will always be lower than or equal to that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the  $k$ -NN classifier grows with the size of the training set. 5. None of the above.

*Your Answer :*

2 and 4

*Your Explanation :*

2: The training error of 1-NN is by default 0 (everything is predicted as itself.)

4: As the inference process of KNN is simply comparing to all the training samples, its complexity is  $O(n)$ , where  $n$  is the size of the training set, Thus, larger training set means longer inference time (putting aside things like multi processing etc, which are of course not really scaleable).

[ ]:

# softmax

January 7, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment1/'
FOLDERNAME = 'icv83551/assignments/assignment1/'
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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 1 Softmax Classifier 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** using 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

```
[ ]: # Run some setup code for this notebook.
import random
import numpy as np
```

```

from icv83551.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%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'

# %load_ext autoreload
# %autoreload 2

```

## 1.1 CIFAR-10 Data Loading and Preprocessing

```

[2]: # Load the raw CIFAR-10 data.
cifar10_dir = 'icv83551/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause
↳memory issue)
try:
    del X_train, y_train
    del X_test, y_test
    print('Clear previously loaded data.')
except:
    pass

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

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)

```

```

Training data shape: (50000, 32, 32, 3)
Training labels shape: (50000,)
Test data shape: (10000, 32, 32, 3)
Test labels shape: (10000,)

```

```

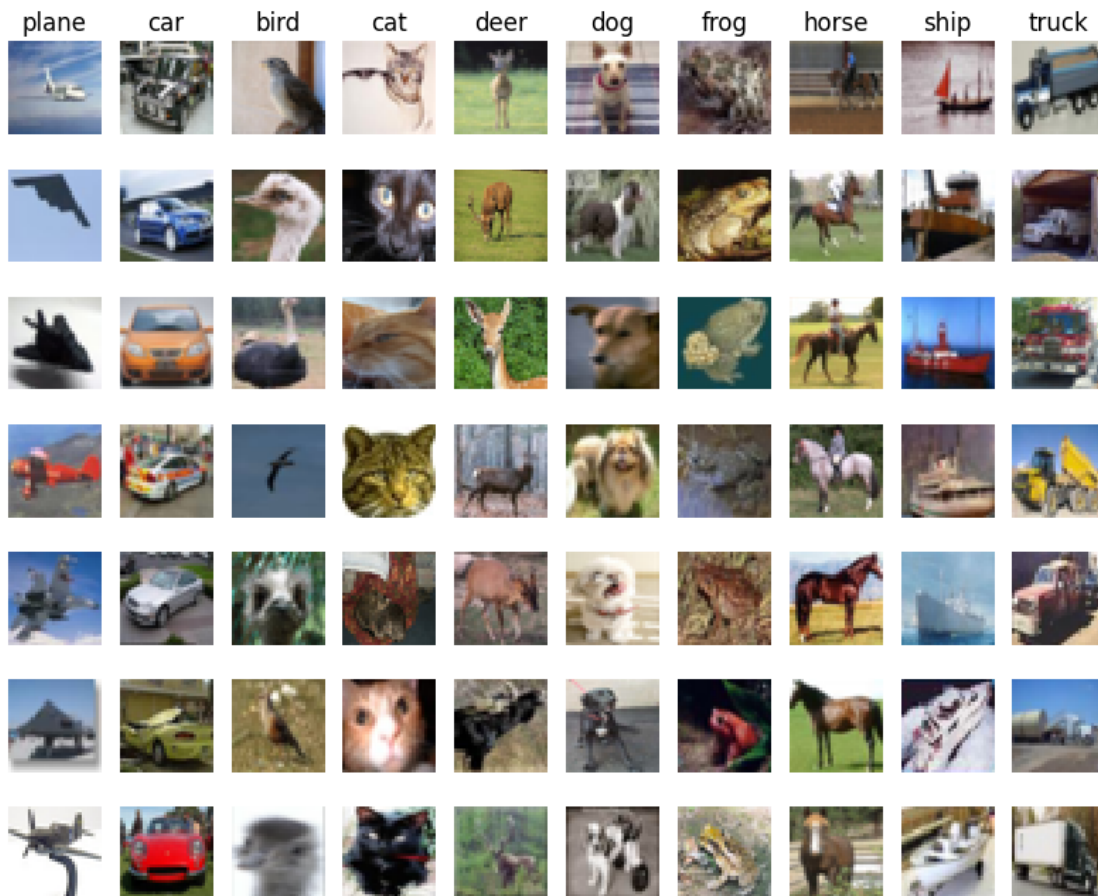
[3]: # Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', '
↳ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):

```

```

idxs = np.flatnonzero(y_train == y)
idxs = np.random.choice(idxs, samples_per_class, replace=False)
for i, idx in enumerate(idxs):
    plt_idx = i * num_classes + y + 1
    plt.subplot(samples_per_class, num_classes, plt_idx)
    plt.imshow(X_train[idx].astype('uint8'))
    plt.axis('off')
    if i == 0:
        plt.title(cls)
plt.show()

```



```

[4]: # Split the data into train, val, and test sets. In addition we will
      # create a small development set as a subset of the training data;
      # we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

```



```

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

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)

```

```

Train data shape: (49000, 32, 32, 3)
Train labels shape: (49000,)
Validation data shape: (1000, 32, 32, 3)
Validation labels shape: (1000,)
Test data shape: (1000, 32, 32, 3)
Test labels shape: (1000,)

```

```

[5]: # 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))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)

```

```
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

```
Training data shape: (49000, 3072)
Validation data shape: (1000, 3072)
Test data shape: (1000, 3072)
dev data shape: (500, 3072)
```

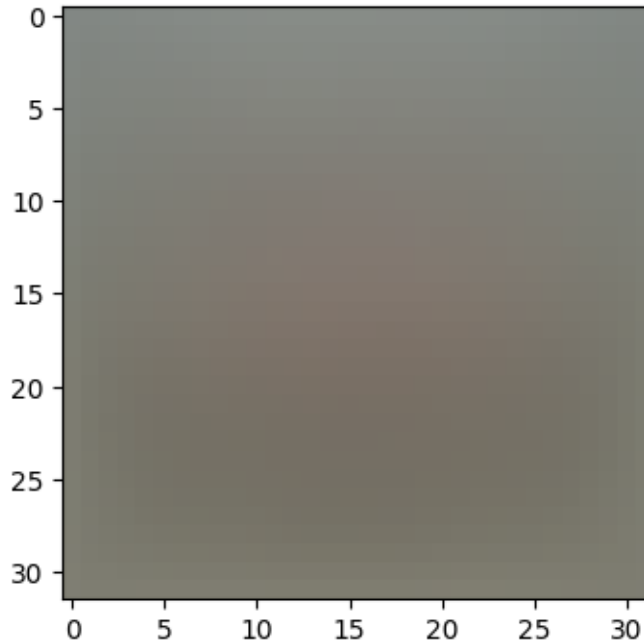
```
[6]: # Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean
    ↪ image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

# third: append the bias dimension of ones (i.e. bias trick) so that our
    ↪ classifier
# only has to worry about optimizing a single weight matrix W.
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))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)

## 1.2 Softmax Classifier

Your code for this section will all be written inside `icv83551/classifiers/softmax.py`.

As you can see, we have prefilled the function `softmax_loss_naive` which uses for loops to evaluate the softmax loss function.

```
[7]: # Evaluate the naive implementation of the loss we provided for you:
from icv83551.classifiers.softmax import softmax_loss_naive
import time

# generate a random Softmax classifier weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))

# 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: 2.336002

loss: 2.336002

sanity check: 2.302585

```
[8]: y_dev.max()
```

```
[8]: np.int64(9)
```

### Inline Question 1

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

*Your Answer :* As we start with random  $W$  matrix, we expect all scores to distribute roughly uniformly. i.e., after softmax, each class get  $\sim 0.1$  probability.

The loss is defined as  $-\text{np.log}(p[y\_i])$

We have 10 classes, thus the average loss should be about  $-\log(0.1)$

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the softmax loss function and implement it inline inside the function `softmax_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

```
[9]: # Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should
# match
# almost exactly along all dimensions.
from icv83551.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)

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
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)
```

```
numerical: -1.001693 analytic: -1.001693, relative error: 1.985434e-08
numerical: -0.229607 analytic: -0.229607, relative error: 4.930766e-08
numerical: 3.090595 analytic: 3.090595, relative error: 1.335951e-08
numerical: 1.419315 analytic: 1.419315, relative error: 5.984676e-09
numerical: 1.829732 analytic: 1.829732, relative error: 3.771782e-08
numerical: 1.079658 analytic: 1.079658, relative error: 5.521798e-08
numerical: 3.766103 analytic: 3.766103, relative error: 1.159254e-08
numerical: -2.283111 analytic: -2.283111, relative error: 1.052323e-08
```

```

numerical: -0.568008 analytic: -0.568008, relative error: 3.424084e-09
numerical: -0.615975 analytic: -0.615975, relative error: 3.427315e-08
numerical: 2.632810 analytic: 2.632810, relative error: 1.793633e-08
numerical: 0.412096 analytic: 0.412096, relative error: 6.291976e-09
numerical: 2.273995 analytic: 2.273995, relative error: 1.569720e-08
numerical: -2.189725 analytic: -2.189725, relative error: 2.016841e-08
numerical: -4.931376 analytic: -4.931376, relative error: 1.132592e-09
numerical: -0.102201 analytic: -0.102202, relative error: 2.584271e-07
numerical: -2.895055 analytic: -2.895055, relative error: 2.419489e-08
numerical: -0.832302 analytic: -0.832302, relative error: 4.699191e-09
numerical: 2.598509 analytic: 2.598509, relative error: 2.719187e-08
numerical: 2.006951 analytic: 2.006951, relative error: 5.022950e-08

```

## Inline Question 2

Although gradcheck is reliable softmax loss, it is possible that for SVM loss, once in a while, a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a svm loss gradient check could fail? How would change the margin affect of the frequency of this happening?

Note that SVM loss for a sample  $(x_i, y_i)$  is defined as:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

Where  $j$  iterates over all classes except the correct class  $y_i$  and  $s_j$  denotes the classifier score for  $j^{th}$  class.  $\Delta$  is a scalar margin. *Hint: the SVM loss function is not strictly speaking differentiable.*

*Your Answer :* The svm loss is not differentiable at the exact point where  $0 = s_j - s_{y_i} + \Delta$ .

For  $0 < s_j - s_{y_i} + \Delta$  the gradient is 1, and for  $0 > s_j - s_{y_i} + \Delta$  the gradient is 0.

That is not a reason for a real concern, as those points are extremely rare and the chance the optimizer will land exactly on one of these points is very slim. Moreover, even if it does happens, the gradients will just fall into either 0 or 1 and the optimization will converge just fine.

A simple example of such mismatch is for  $s_j - s_{y_i} + \Delta = 0.001$ , and  $\epsilon = 0.01$ . As  $s_j - s_{y_i} + \Delta > 0$ , the analytical gradient is 1. Yet, the numerical gradient is  $\frac{\max(0, 0.001+0.01) - \max(0, 0.001-0.01)}{2 \times 0.01} = \frac{0.011-0}{0.02} = 0.55$ .

As the main factors are just the total sum  $s_j - s_{y_i} + \Delta$  and the numerical  $\epsilon$  value, changing the margin  $\Delta$  will not affect the frequency, but rather the scores values for which this happens.

```

[10]: # Next implement the function softmax_loss_vectorized; for now only compute the
      ↪ loss;
      # we will implement the gradient in a moment.
      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 icv83551.classifiers.softmax import softmax_loss_vectorized
      tic = time.time()

```

```

loss_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))

# The losses should match but your vectorized implementation should be much
↪faster.
print('difference: %f' % (loss_naive - loss_vectorized))

```

Naive loss: 2.336002e+00 computed in 0.056188s  
 Vectorized loss: 2.336002e+00 computed in 0.005197s  
 difference: -0.000000

```

[11]: # Complete the implementation of softmax_loss_vectorized, and compute the
↪gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)

```

Naive loss and gradient: computed in 0.079240s  
 Vectorized loss and gradient: computed in 0.001786s  
 difference: 0.000000

### 1.2.1 Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `icv83551/classifiers/linear_classifier.py`.

```

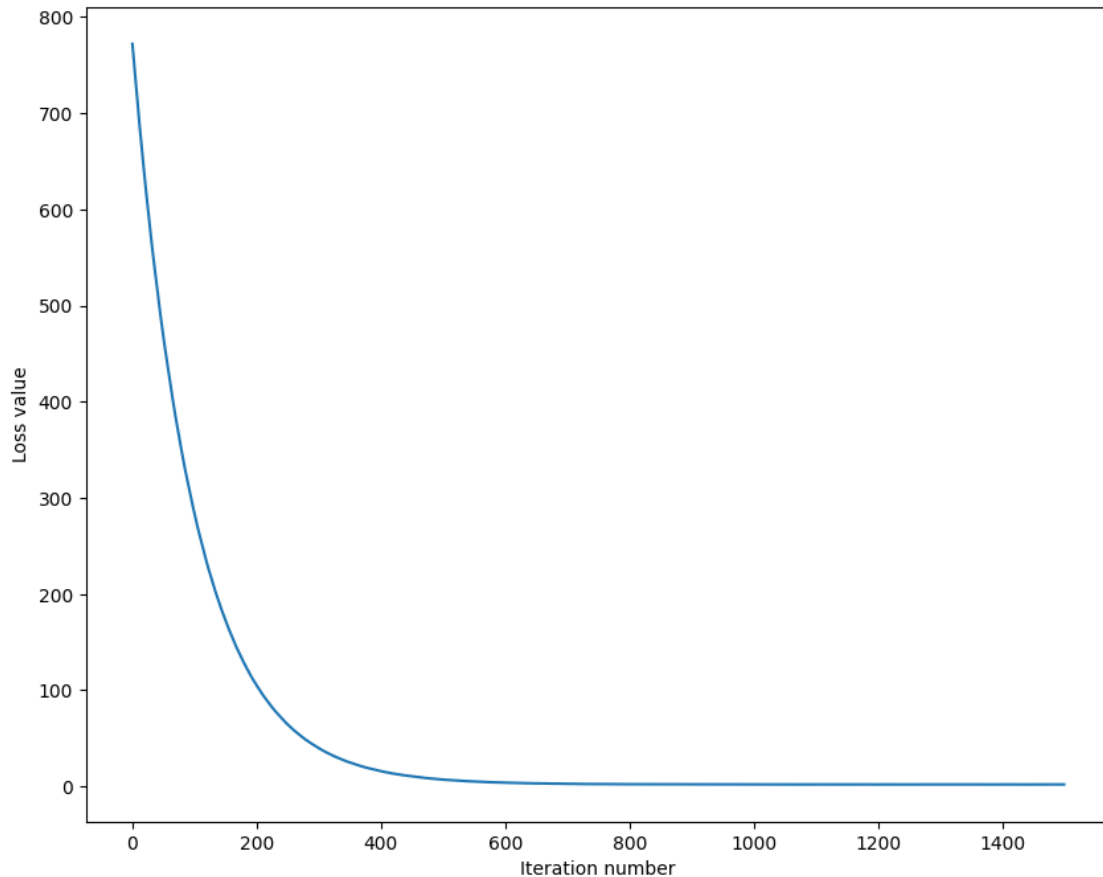
[16]: # In the file linear_classifier.py, implement SGD in the function
# LinearClassifier.train() and then run it with the code below.
from icv83551.classifiers import Softmax
softmax = Softmax()

```

```
tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                           num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 771.995025
iteration 100 / 1500: loss 283.421613
iteration 200 / 1500: loss 104.959146
iteration 300 / 1500: loss 39.811603
iteration 400 / 1500: loss 15.880718
iteration 500 / 1500: loss 7.161137
iteration 600 / 1500: loss 3.914981
iteration 700 / 1500: loss 2.775388
iteration 800 / 1500: loss 2.296916
iteration 900 / 1500: loss 2.168671
iteration 1000 / 1500: loss 2.148775
iteration 1100 / 1500: loss 2.149669
iteration 1200 / 1500: loss 2.072666
iteration 1300 / 1500: loss 2.130773
iteration 1400 / 1500: loss 2.124903
That took 1.642461s
```

```
[17]: # A useful debugging strategy is to plot the loss as a function of
      # iteration number:
      plt.plot(loss_hist)
      plt.xlabel('Iteration number')
      plt.ylabel('Loss value')
      plt.show()
```



```
[19]: # Write the LinearClassifier.predict function and evaluate the performance on
# both the training and validation set
# You should get validation accuracy of about 0.34 (> 0.33).
y_train_pred = softmax.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.324041
validation accuracy: 0.335000
```

```
[20]: # Save the trained model for autograder.
softmax.save("softmax.npy")
```

```
softmax.npy saved.
```

```
[21]: # 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 about 0.365 (> 0.36) on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter search.
# This may be caused by extreme values, and is not a bug.

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1    # The highest validation accuracy that we have seen so far.
best_softmax = None # The Softmax object that achieved the highest validation_
    ↪rate.

#####
# TODO:                                     #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a Softmax on the.       #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the Softmax object that achieves this.   #
# accuracy in best_softmax.                                                    #
#                                                                               #
# Hint: You should use a small value for num_iters as you develop your        #
# validation code so that the classifiers don't take much time to train; once  #
# you are confident that your validation code works, you should rerun the     #
# code with a larger value for num_iters.                                     #
#####

# Provided as a reference. You may or may not want to change these_
    ↪hyperparameters
learning_rates = [1e-7, 1e-6]
regularization_strengths = [2.5e4, 1e4]

for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate=lr, reg=reg,
                       num_iters=1500, verbose=False)
        y_train_pred = softmax.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = softmax.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)
        results[(lr, reg)] = (train_accuracy, val_accuracy)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax

```

```

        print('New best validation accuracy: %f' % best_val)

# 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)

```

```

New best validation accuracy: 0.338000
New best validation accuracy: 0.371000
lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.350837 val accuracy: 0.371000
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.323531 val accuracy: 0.338000
lr 1.000000e-06 reg 1.000000e+04 train accuracy: 0.350122 val accuracy: 0.368000
lr 1.000000e-06 reg 2.500000e+04 train accuracy: 0.330082 val accuracy: 0.336000
best validation accuracy achieved during cross-validation: 0.371000

```

```

[22]: # Visualize the cross-validation results
import math
import pdb

# pdb.set_trace()

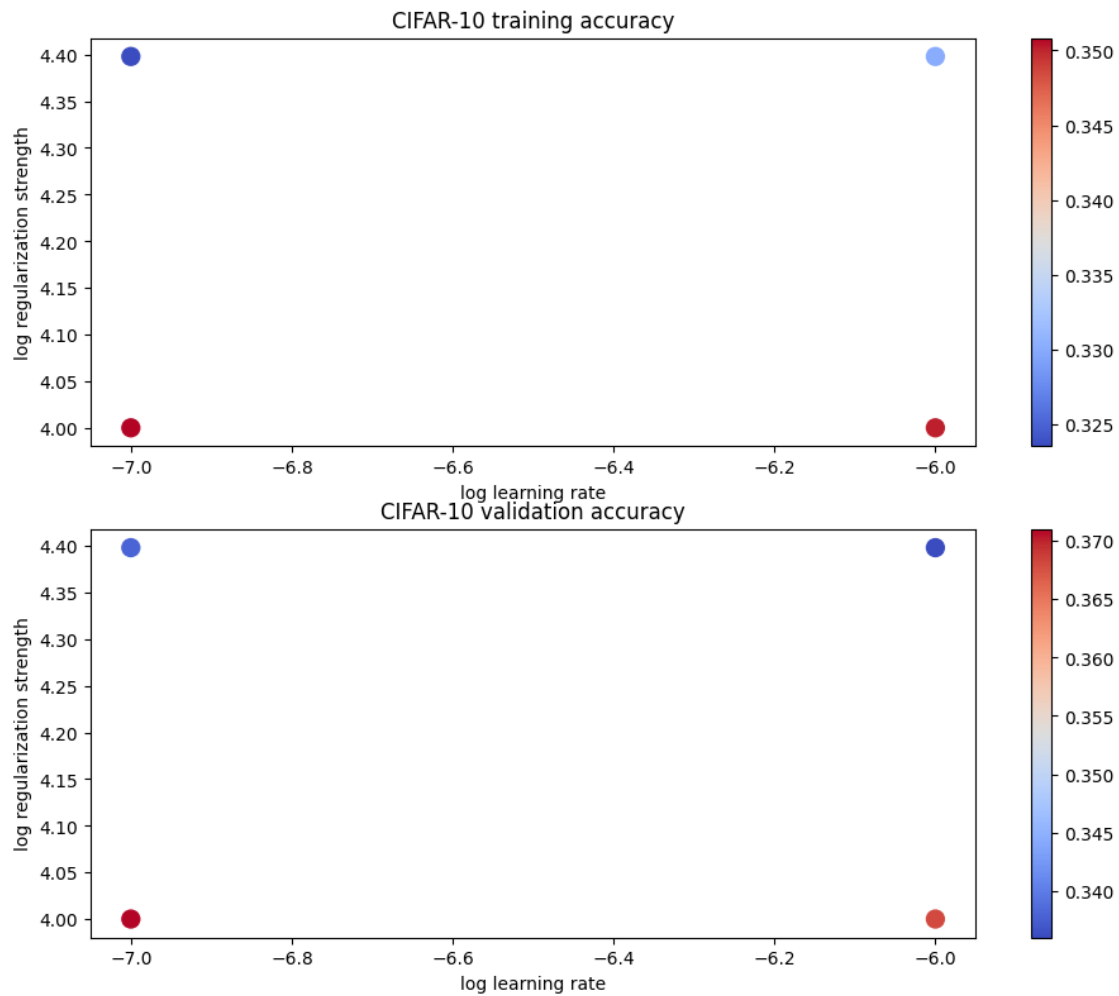
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')

```

```
plt.show()
```



```
[ ]: # 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 classifier on raw pixels final test set accuracy: %f' %_
      ↪test_accuracy)
```

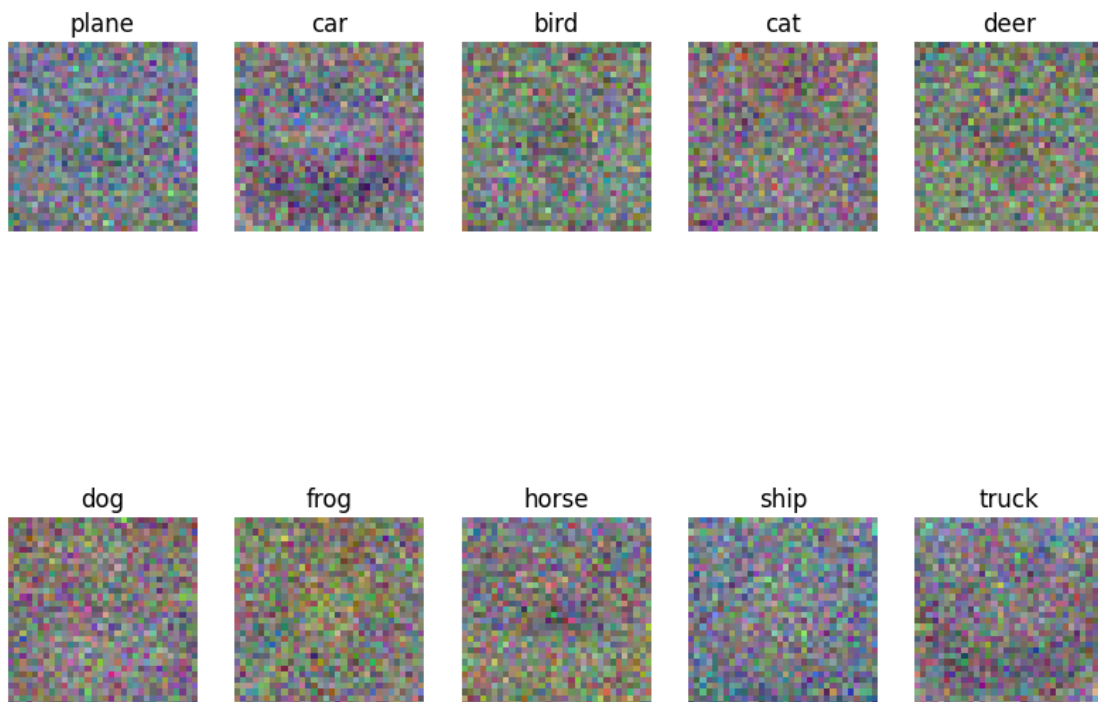
Softmax classifier on raw pixels final test set accuracy: 0.358000

```
[24]: # Save best softmax model
best_softmax.save("best_softmax.npy")
```

best\_softmax.npy saved.

```
[25]: # Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these
# or may not be nice to look at.
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])
```



### Inline question 3

Describe what your visualized Softmax classifier weights look like, and offer a brief explanation for why they look the way they do.

*Your Answer :*

The most important thing to understand is that the weights are optimized to match as best as

possible the general pattern of each class (a linear classifier is a dot-product between the image and the weights, thus maximizing for cross-correlation). If I must extract somethings from the weights images, they probably look mostly for color matching. for example, blue-ish for ship and plane. green-ish for frog etc. It is also possible to vaguely see darker pixels around the shape itself for some weights. This is probably also due to common color structure, where the object has more shade, thus is a bit darker than the background.

**Inline Question 4 - 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 change the softmax loss, but leave the SVM loss unchanged.

*Your Answer :*

Yes.

*Your Explanation :*

The SVM loss just wants all samples to be within their class by over a defined margin. For any sample where the score on the right class is greater than all other classes by over the defined margin, the loss is set to 0 (as well as the gradients). Thus, adding a sample with a big enough confidence will have no affect on the SVM Loss. On the other hand, the softmax loss always tries to make the margin as big as possible for each sample, thus any new sample be be taken into considaratoin and will get a loss.

[ ]:

# two\_layer\_net

January 7, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment1/'
FOLDERNAME = 'icv83551/assignments/assignment1/'
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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 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.

```
[ ]: # As usual, a bit of setup
import time
import numpy as np
import matplotlib.pyplot as plt
from icv83551.classifiers.fc_net import *
from icv83551.data_utils import get_CIFAR10_data
from icv83551.gradient_check import eval_numerical_gradient, \
    eval_numerical_gradient_array
from icv83551.solver import Solver
# %load_ext autoreload
# %autoreload 2
%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'

#%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))))
```

```
[2]: # 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 `icv83551/layers.py` and implement the `affine_forward` function.

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

```

[3]: # 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.

```

[4]: from icv83551.layers import affine_forward, affine_backward
# 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: 9.579607670311995e-11
dw error: 1.2604803862099753e-10
db error: 7.72167374950876e-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:

```

[5]: # 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:

```
[6]: 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. Leaky ReLU

*Your Answer :*

Sigmoid - values far from zero (either positive or negative) gets to a near platou values (1 or -1 respectively), which leads to very small gradients  
ReLU - all negative values get a gradient of 0

## 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 `icv83551/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:

```
[7]: from icv83551.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:  1.70644420803027e-10
dw error:  8.162088973990222e-11
db error:  7.82672402145899e-12

```

## 7 Loss layers: Softmax

Now implement the loss and gradient for softmax in the `softmax_loss` function in `icv83551/layers.py`. These should be similar to what you implemented in `icv83551/classifiers/softmax.py`. Other loss functions (e.g. `svm_loss`) can also be implemented in a modular way, however, it is not required for this assignment.

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

```

[9]: 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: 2.3025458445007376
dx error: 8.234144091578429e-09
```

## 8 Two-layer network

Open the file `icv83551/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.

```
[10]: 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: 1.22e-08
W2 relative error: 3.43e-10
b1 relative error: 9.83e-09
b2 relative error: 2.53e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.53e-07
W2 relative error: 1.37e-07
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10

```

## 9 Solver

Open the file `icv83551/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about 36% accuracy on the validation set.

```

[26]: from icv83551.solver import Solver

input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes, reg=1e-5)

```

```

# solver = Solver(model, data)
solver = Solver(model, data,
                 optim_config={'learning_rate': 2e-5},
                 # lr_decay=0.95,
                 num_epochs=10)

#####
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                           #
#####
solver.train()
print('Final training accuracy: ', solver.train_acc_history[-1])
print('Final validation accuracy: ', solver.val_acc_history[-1])
#####
#                               END OF YOUR CODE                               #
#####

```

```

(Iteration 1 / 4900) loss: 2.300102
(Epoch 0 / 10) train acc: 0.120000; val_acc: 0.115000
(Iteration 11 / 4900) loss: 2.300685
(Iteration 21 / 4900) loss: 2.303492
(Iteration 31 / 4900) loss: 2.298462
(Iteration 41 / 4900) loss: 2.300970
(Iteration 51 / 4900) loss: 2.297714
(Iteration 61 / 4900) loss: 2.297367
(Iteration 71 / 4900) loss: 2.297402
(Iteration 81 / 4900) loss: 2.296018
(Iteration 91 / 4900) loss: 2.298539
(Iteration 101 / 4900) loss: 2.299290
(Iteration 111 / 4900) loss: 2.294557
(Iteration 121 / 4900) loss: 2.296831
(Iteration 131 / 4900) loss: 2.293868
(Iteration 141 / 4900) loss: 2.292235
(Iteration 151 / 4900) loss: 2.293439
(Iteration 161 / 4900) loss: 2.296132
(Iteration 171 / 4900) loss: 2.292537
(Iteration 181 / 4900) loss: 2.288976
(Iteration 191 / 4900) loss: 2.286964
(Iteration 201 / 4900) loss: 2.292520
(Iteration 211 / 4900) loss: 2.283055
(Iteration 221 / 4900) loss: 2.290513
(Iteration 231 / 4900) loss: 2.295965
(Iteration 241 / 4900) loss: 2.286696
(Iteration 251 / 4900) loss: 2.289248
(Iteration 261 / 4900) loss: 2.292884
(Iteration 271 / 4900) loss: 2.278595
(Iteration 281 / 4900) loss: 2.276570

```

(Iteration 291 / 4900) loss: 2.286481  
(Iteration 301 / 4900) loss: 2.281792  
(Iteration 311 / 4900) loss: 2.284691  
(Iteration 321 / 4900) loss: 2.280628  
(Iteration 331 / 4900) loss: 2.273395  
(Iteration 341 / 4900) loss: 2.278036  
(Iteration 351 / 4900) loss: 2.272667  
(Iteration 361 / 4900) loss: 2.273524  
(Iteration 371 / 4900) loss: 2.262468  
(Iteration 381 / 4900) loss: 2.268565  
(Iteration 391 / 4900) loss: 2.256781  
(Iteration 401 / 4900) loss: 2.274695  
(Iteration 411 / 4900) loss: 2.255908  
(Iteration 421 / 4900) loss: 2.260410  
(Iteration 431 / 4900) loss: 2.264190  
(Iteration 441 / 4900) loss: 2.267506  
(Iteration 451 / 4900) loss: 2.258559  
(Iteration 461 / 4900) loss: 2.276062  
(Iteration 471 / 4900) loss: 2.256845  
(Iteration 481 / 4900) loss: 2.254204  
(Epoch 1 / 10) train acc: 0.217000; val\_acc: 0.195000  
(Iteration 491 / 4900) loss: 2.267900  
(Iteration 501 / 4900) loss: 2.266459  
(Iteration 511 / 4900) loss: 2.249664  
(Iteration 521 / 4900) loss: 2.265962  
(Iteration 531 / 4900) loss: 2.269433  
(Iteration 541 / 4900) loss: 2.257772  
(Iteration 551 / 4900) loss: 2.228083  
(Iteration 561 / 4900) loss: 2.254977  
(Iteration 571 / 4900) loss: 2.251344  
(Iteration 581 / 4900) loss: 2.243755  
(Iteration 591 / 4900) loss: 2.236968  
(Iteration 601 / 4900) loss: 2.221671  
(Iteration 611 / 4900) loss: 2.232036  
(Iteration 621 / 4900) loss: 2.233951  
(Iteration 631 / 4900) loss: 2.258788  
(Iteration 641 / 4900) loss: 2.207828  
(Iteration 651 / 4900) loss: 2.229827  
(Iteration 661 / 4900) loss: 2.226769  
(Iteration 671 / 4900) loss: 2.198036  
(Iteration 681 / 4900) loss: 2.198428  
(Iteration 691 / 4900) loss: 2.208439  
(Iteration 701 / 4900) loss: 2.236734  
(Iteration 711 / 4900) loss: 2.200859  
(Iteration 721 / 4900) loss: 2.211841  
(Iteration 731 / 4900) loss: 2.205248  
(Iteration 741 / 4900) loss: 2.203617  
(Iteration 751 / 4900) loss: 2.157336

(Iteration 761 / 4900) loss: 2.159330  
(Iteration 771 / 4900) loss: 2.202751  
(Iteration 781 / 4900) loss: 2.230718  
(Iteration 791 / 4900) loss: 2.192078  
(Iteration 801 / 4900) loss: 2.188349  
(Iteration 811 / 4900) loss: 2.168929  
(Iteration 821 / 4900) loss: 2.207729  
(Iteration 831 / 4900) loss: 2.166900  
(Iteration 841 / 4900) loss: 2.237598  
(Iteration 851 / 4900) loss: 2.217551  
(Iteration 861 / 4900) loss: 2.129286  
(Iteration 871 / 4900) loss: 2.151703  
(Iteration 881 / 4900) loss: 2.169010  
(Iteration 891 / 4900) loss: 2.168120  
(Iteration 901 / 4900) loss: 2.166187  
(Iteration 911 / 4900) loss: 2.203817  
(Iteration 921 / 4900) loss: 2.182217  
(Iteration 931 / 4900) loss: 2.146114  
(Iteration 941 / 4900) loss: 2.159399  
(Iteration 951 / 4900) loss: 2.152807  
(Iteration 961 / 4900) loss: 2.148066  
(Iteration 971 / 4900) loss: 2.161022  
(Epoch 2 / 10) train acc: 0.244000; val\_acc: 0.244000  
(Iteration 981 / 4900) loss: 2.214264  
(Iteration 991 / 4900) loss: 2.128491  
(Iteration 1001 / 4900) loss: 2.155844  
(Iteration 1011 / 4900) loss: 2.086207  
(Iteration 1021 / 4900) loss: 2.111863  
(Iteration 1031 / 4900) loss: 2.117687  
(Iteration 1041 / 4900) loss: 2.174678  
(Iteration 1051 / 4900) loss: 2.118128  
(Iteration 1061 / 4900) loss: 2.157886  
(Iteration 1071 / 4900) loss: 2.053711  
(Iteration 1081 / 4900) loss: 2.108864  
(Iteration 1091 / 4900) loss: 2.081513  
(Iteration 1101 / 4900) loss: 2.165619  
(Iteration 1111 / 4900) loss: 2.139243  
(Iteration 1121 / 4900) loss: 2.089207  
(Iteration 1131 / 4900) loss: 2.139976  
(Iteration 1141 / 4900) loss: 2.143759  
(Iteration 1151 / 4900) loss: 2.168750  
(Iteration 1161 / 4900) loss: 2.106117  
(Iteration 1171 / 4900) loss: 2.107280  
(Iteration 1181 / 4900) loss: 2.150736  
(Iteration 1191 / 4900) loss: 2.061467  
(Iteration 1201 / 4900) loss: 2.118890  
(Iteration 1211 / 4900) loss: 2.178759  
(Iteration 1221 / 4900) loss: 2.011134



(Iteration 1231 / 4900) loss: 2.071667  
(Iteration 1241 / 4900) loss: 2.109130  
(Iteration 1251 / 4900) loss: 2.108171  
(Iteration 1261 / 4900) loss: 2.137263  
(Iteration 1271 / 4900) loss: 2.094564  
(Iteration 1281 / 4900) loss: 2.122439  
(Iteration 1291 / 4900) loss: 2.114631  
(Iteration 1301 / 4900) loss: 2.149765  
(Iteration 1311 / 4900) loss: 2.098737  
(Iteration 1321 / 4900) loss: 2.019791  
(Iteration 1331 / 4900) loss: 2.130629  
(Iteration 1341 / 4900) loss: 2.090901  
(Iteration 1351 / 4900) loss: 2.102474  
(Iteration 1361 / 4900) loss: 2.042550  
(Iteration 1371 / 4900) loss: 2.081534  
(Iteration 1381 / 4900) loss: 2.065812  
(Iteration 1391 / 4900) loss: 2.155227  
(Iteration 1401 / 4900) loss: 2.102313  
(Iteration 1411 / 4900) loss: 2.056813  
(Iteration 1421 / 4900) loss: 2.160819  
(Iteration 1431 / 4900) loss: 2.048437  
(Iteration 1441 / 4900) loss: 2.122674  
(Iteration 1451 / 4900) loss: 2.011335  
(Iteration 1461 / 4900) loss: 2.043053  
(Epoch 3 / 10) train acc: 0.231000; val\_acc: 0.264000  
(Iteration 1471 / 4900) loss: 1.987498  
(Iteration 1481 / 4900) loss: 2.060930  
(Iteration 1491 / 4900) loss: 2.065134  
(Iteration 1501 / 4900) loss: 2.117835  
(Iteration 1511 / 4900) loss: 2.039889  
(Iteration 1521 / 4900) loss: 2.053419  
(Iteration 1531 / 4900) loss: 2.103360  
(Iteration 1541 / 4900) loss: 2.085921  
(Iteration 1551 / 4900) loss: 2.043875  
(Iteration 1561 / 4900) loss: 1.983912  
(Iteration 1571 / 4900) loss: 1.991437  
(Iteration 1581 / 4900) loss: 2.078837  
(Iteration 1591 / 4900) loss: 1.980594  
(Iteration 1601 / 4900) loss: 2.003006  
(Iteration 1611 / 4900) loss: 1.948427  
(Iteration 1621 / 4900) loss: 2.027215  
(Iteration 1631 / 4900) loss: 2.027591  
(Iteration 1641 / 4900) loss: 2.133646  
(Iteration 1651 / 4900) loss: 2.001554  
(Iteration 1661 / 4900) loss: 2.002813  
(Iteration 1671 / 4900) loss: 2.010251  
(Iteration 1681 / 4900) loss: 1.949003  
(Iteration 1691 / 4900) loss: 2.015734

(Iteration 1701 / 4900) loss: 2.025597  
(Iteration 1711 / 4900) loss: 1.994970  
(Iteration 1721 / 4900) loss: 1.980615  
(Iteration 1731 / 4900) loss: 2.001614  
(Iteration 1741 / 4900) loss: 1.930526  
(Iteration 1751 / 4900) loss: 1.984483  
(Iteration 1761 / 4900) loss: 1.967865  
(Iteration 1771 / 4900) loss: 1.988420  
(Iteration 1781 / 4900) loss: 1.990803  
(Iteration 1791 / 4900) loss: 2.075331  
(Iteration 1801 / 4900) loss: 2.070436  
(Iteration 1811 / 4900) loss: 2.027911  
(Iteration 1821 / 4900) loss: 1.917065  
(Iteration 1831 / 4900) loss: 2.042647  
(Iteration 1841 / 4900) loss: 1.854948  
(Iteration 1851 / 4900) loss: 2.004822  
(Iteration 1861 / 4900) loss: 2.029764  
(Iteration 1871 / 4900) loss: 1.990127  
(Iteration 1881 / 4900) loss: 1.919989  
(Iteration 1891 / 4900) loss: 1.955371  
(Iteration 1901 / 4900) loss: 1.957978  
(Iteration 1911 / 4900) loss: 1.954215  
(Iteration 1921 / 4900) loss: 1.875621  
(Iteration 1931 / 4900) loss: 2.015688  
(Iteration 1941 / 4900) loss: 2.008238  
(Iteration 1951 / 4900) loss: 2.008747  
(Epoch 4 / 10) train acc: 0.287000; val\_acc: 0.277000  
(Iteration 1961 / 4900) loss: 2.028697  
(Iteration 1971 / 4900) loss: 2.002085  
(Iteration 1981 / 4900) loss: 1.898267  
(Iteration 1991 / 4900) loss: 1.999805  
(Iteration 2001 / 4900) loss: 1.937351  
(Iteration 2011 / 4900) loss: 1.971001  
(Iteration 2021 / 4900) loss: 1.917787  
(Iteration 2031 / 4900) loss: 2.065764  
(Iteration 2041 / 4900) loss: 1.993105  
(Iteration 2051 / 4900) loss: 1.905744  
(Iteration 2061 / 4900) loss: 2.030663  
(Iteration 2071 / 4900) loss: 2.013007  
(Iteration 2081 / 4900) loss: 1.983675  
(Iteration 2091 / 4900) loss: 2.024939  
(Iteration 2101 / 4900) loss: 2.053551  
(Iteration 2111 / 4900) loss: 2.013748  
(Iteration 2121 / 4900) loss: 2.033672  
(Iteration 2131 / 4900) loss: 1.944241  
(Iteration 2141 / 4900) loss: 1.976431  
(Iteration 2151 / 4900) loss: 1.984029  
(Iteration 2161 / 4900) loss: 2.072301

(Iteration 2171 / 4900) loss: 1.916909  
(Iteration 2181 / 4900) loss: 1.989951  
(Iteration 2191 / 4900) loss: 2.091576  
(Iteration 2201 / 4900) loss: 1.998683  
(Iteration 2211 / 4900) loss: 1.990794  
(Iteration 2221 / 4900) loss: 1.961312  
(Iteration 2231 / 4900) loss: 1.877324  
(Iteration 2241 / 4900) loss: 1.920013  
(Iteration 2251 / 4900) loss: 2.026169  
(Iteration 2261 / 4900) loss: 1.966741  
(Iteration 2271 / 4900) loss: 2.019930  
(Iteration 2281 / 4900) loss: 1.894054  
(Iteration 2291 / 4900) loss: 1.988339  
(Iteration 2301 / 4900) loss: 1.998341  
(Iteration 2311 / 4900) loss: 1.996153  
(Iteration 2321 / 4900) loss: 1.949806  
(Iteration 2331 / 4900) loss: 1.921081  
(Iteration 2341 / 4900) loss: 2.022889  
(Iteration 2351 / 4900) loss: 1.916384  
(Iteration 2361 / 4900) loss: 1.978843  
(Iteration 2371 / 4900) loss: 1.911265  
(Iteration 2381 / 4900) loss: 2.004065  
(Iteration 2391 / 4900) loss: 1.996565  
(Iteration 2401 / 4900) loss: 1.900906  
(Iteration 2411 / 4900) loss: 1.957828  
(Iteration 2421 / 4900) loss: 1.971705  
(Iteration 2431 / 4900) loss: 1.974156  
(Iteration 2441 / 4900) loss: 2.037362  
(Epoch 5 / 10) train acc: 0.253000; val\_acc: 0.294000  
(Iteration 2451 / 4900) loss: 1.960389  
(Iteration 2461 / 4900) loss: 1.980093  
(Iteration 2471 / 4900) loss: 1.960589  
(Iteration 2481 / 4900) loss: 1.983134  
(Iteration 2491 / 4900) loss: 1.910375  
(Iteration 2501 / 4900) loss: 1.957560  
(Iteration 2511 / 4900) loss: 1.920204  
(Iteration 2521 / 4900) loss: 1.944320  
(Iteration 2531 / 4900) loss: 1.948980  
(Iteration 2541 / 4900) loss: 1.834301  
(Iteration 2551 / 4900) loss: 1.821880  
(Iteration 2561 / 4900) loss: 2.047744  
(Iteration 2571 / 4900) loss: 1.706409  
(Iteration 2581 / 4900) loss: 1.948809  
(Iteration 2591 / 4900) loss: 2.073658  
(Iteration 2601 / 4900) loss: 1.907236  
(Iteration 2611 / 4900) loss: 2.010194  
(Iteration 2621 / 4900) loss: 1.956620  
(Iteration 2631 / 4900) loss: 1.949347

(Iteration 2641 / 4900) loss: 2.009167  
(Iteration 2651 / 4900) loss: 2.011557  
(Iteration 2661 / 4900) loss: 1.904864  
(Iteration 2671 / 4900) loss: 1.856737  
(Iteration 2681 / 4900) loss: 2.001098  
(Iteration 2691 / 4900) loss: 1.855135  
(Iteration 2701 / 4900) loss: 1.978196  
(Iteration 2711 / 4900) loss: 1.946455  
(Iteration 2721 / 4900) loss: 1.913564  
(Iteration 2731 / 4900) loss: 1.957544  
(Iteration 2741 / 4900) loss: 1.967620  
(Iteration 2751 / 4900) loss: 1.953531  
(Iteration 2761 / 4900) loss: 1.875078  
(Iteration 2771 / 4900) loss: 1.917581  
(Iteration 2781 / 4900) loss: 1.957824  
(Iteration 2791 / 4900) loss: 1.907017  
(Iteration 2801 / 4900) loss: 1.904392  
(Iteration 2811 / 4900) loss: 1.923286  
(Iteration 2821 / 4900) loss: 1.852359  
(Iteration 2831 / 4900) loss: 1.945012  
(Iteration 2841 / 4900) loss: 1.961290  
(Iteration 2851 / 4900) loss: 1.947928  
(Iteration 2861 / 4900) loss: 1.941723  
(Iteration 2871 / 4900) loss: 1.971545  
(Iteration 2881 / 4900) loss: 1.928609  
(Iteration 2891 / 4900) loss: 1.940932  
(Iteration 2901 / 4900) loss: 1.917871  
(Iteration 2911 / 4900) loss: 1.939914  
(Iteration 2921 / 4900) loss: 1.898690  
(Iteration 2931 / 4900) loss: 1.880973  
(Epoch 6 / 10) train acc: 0.330000; val\_acc: 0.313000  
(Iteration 2941 / 4900) loss: 1.814877  
(Iteration 2951 / 4900) loss: 2.053091  
(Iteration 2961 / 4900) loss: 1.908383  
(Iteration 2971 / 4900) loss: 1.823472  
(Iteration 2981 / 4900) loss: 1.884099  
(Iteration 2991 / 4900) loss: 1.923982  
(Iteration 3001 / 4900) loss: 1.973707  
(Iteration 3011 / 4900) loss: 1.934228  
(Iteration 3021 / 4900) loss: 1.874854  
(Iteration 3031 / 4900) loss: 1.840925  
(Iteration 3041 / 4900) loss: 1.939927  
(Iteration 3051 / 4900) loss: 1.944019  
(Iteration 3061 / 4900) loss: 1.867050  
(Iteration 3071 / 4900) loss: 2.078613  
(Iteration 3081 / 4900) loss: 1.992530  
(Iteration 3091 / 4900) loss: 1.923639  
(Iteration 3101 / 4900) loss: 1.873523

(Iteration 3111 / 4900) loss: 1.970488  
(Iteration 3121 / 4900) loss: 2.014125  
(Iteration 3131 / 4900) loss: 1.833933  
(Iteration 3141 / 4900) loss: 1.901996  
(Iteration 3151 / 4900) loss: 1.864725  
(Iteration 3161 / 4900) loss: 1.891709  
(Iteration 3171 / 4900) loss: 1.745603  
(Iteration 3181 / 4900) loss: 1.980608  
(Iteration 3191 / 4900) loss: 1.795496  
(Iteration 3201 / 4900) loss: 1.965947  
(Iteration 3211 / 4900) loss: 1.810707  
(Iteration 3221 / 4900) loss: 1.910265  
(Iteration 3231 / 4900) loss: 1.880716  
(Iteration 3241 / 4900) loss: 1.906585  
(Iteration 3251 / 4900) loss: 1.986157  
(Iteration 3261 / 4900) loss: 1.923821  
(Iteration 3271 / 4900) loss: 1.921593  
(Iteration 3281 / 4900) loss: 1.999810  
(Iteration 3291 / 4900) loss: 1.929879  
(Iteration 3301 / 4900) loss: 1.823325  
(Iteration 3311 / 4900) loss: 1.887567  
(Iteration 3321 / 4900) loss: 1.685032  
(Iteration 3331 / 4900) loss: 1.993009  
(Iteration 3341 / 4900) loss: 1.970227  
(Iteration 3351 / 4900) loss: 1.873249  
(Iteration 3361 / 4900) loss: 1.933632  
(Iteration 3371 / 4900) loss: 1.792251  
(Iteration 3381 / 4900) loss: 1.957796  
(Iteration 3391 / 4900) loss: 1.897359  
(Iteration 3401 / 4900) loss: 1.848666  
(Iteration 3411 / 4900) loss: 1.793155  
(Iteration 3421 / 4900) loss: 1.841487  
(Epoch 7 / 10) train acc: 0.316000; val\_acc: 0.324000  
(Iteration 3431 / 4900) loss: 1.794619  
(Iteration 3441 / 4900) loss: 1.884691  
(Iteration 3451 / 4900) loss: 1.877202  
(Iteration 3461 / 4900) loss: 1.839536  
(Iteration 3471 / 4900) loss: 1.847504  
(Iteration 3481 / 4900) loss: 1.948634  
(Iteration 3491 / 4900) loss: 1.856373  
(Iteration 3501 / 4900) loss: 1.828667  
(Iteration 3511 / 4900) loss: 1.974542  
(Iteration 3521 / 4900) loss: 1.849244  
(Iteration 3531 / 4900) loss: 1.907955  
(Iteration 3541 / 4900) loss: 1.901726  
(Iteration 3551 / 4900) loss: 1.928077  
(Iteration 3561 / 4900) loss: 1.854559  
(Iteration 3571 / 4900) loss: 1.894384

(Iteration 3581 / 4900) loss: 1.934481  
(Iteration 3591 / 4900) loss: 1.899610  
(Iteration 3601 / 4900) loss: 1.850856  
(Iteration 3611 / 4900) loss: 1.869714  
(Iteration 3621 / 4900) loss: 1.772823  
(Iteration 3631 / 4900) loss: 1.977673  
(Iteration 3641 / 4900) loss: 1.845498  
(Iteration 3651 / 4900) loss: 1.844265  
(Iteration 3661 / 4900) loss: 1.874559  
(Iteration 3671 / 4900) loss: 1.775784  
(Iteration 3681 / 4900) loss: 1.850005  
(Iteration 3691 / 4900) loss: 1.842061  
(Iteration 3701 / 4900) loss: 1.974004  
(Iteration 3711 / 4900) loss: 1.853923  
(Iteration 3721 / 4900) loss: 1.901476  
(Iteration 3731 / 4900) loss: 1.892669  
(Iteration 3741 / 4900) loss: 1.838163  
(Iteration 3751 / 4900) loss: 1.844069  
(Iteration 3761 / 4900) loss: 1.846319  
(Iteration 3771 / 4900) loss: 1.802174  
(Iteration 3781 / 4900) loss: 1.902295  
(Iteration 3791 / 4900) loss: 1.865558  
(Iteration 3801 / 4900) loss: 1.816603  
(Iteration 3811 / 4900) loss: 1.908116  
(Iteration 3821 / 4900) loss: 1.940495  
(Iteration 3831 / 4900) loss: 1.862111  
(Iteration 3841 / 4900) loss: 1.822498  
(Iteration 3851 / 4900) loss: 1.901890  
(Iteration 3861 / 4900) loss: 1.818659  
(Iteration 3871 / 4900) loss: 1.860726  
(Iteration 3881 / 4900) loss: 1.766247  
(Iteration 3891 / 4900) loss: 1.875973  
(Iteration 3901 / 4900) loss: 1.726673  
(Iteration 3911 / 4900) loss: 1.743322  
(Epoch 8 / 10) train acc: 0.361000; val\_acc: 0.337000  
(Iteration 3921 / 4900) loss: 1.837811  
(Iteration 3931 / 4900) loss: 1.674891  
(Iteration 3941 / 4900) loss: 1.752966  
(Iteration 3951 / 4900) loss: 1.871717  
(Iteration 3961 / 4900) loss: 1.907674  
(Iteration 3971 / 4900) loss: 1.918060  
(Iteration 3981 / 4900) loss: 1.860025  
(Iteration 3991 / 4900) loss: 1.906872  
(Iteration 4001 / 4900) loss: 1.919901  
(Iteration 4011 / 4900) loss: 1.738649  
(Iteration 4021 / 4900) loss: 1.786530  
(Iteration 4031 / 4900) loss: 1.829307  
(Iteration 4041 / 4900) loss: 1.807917

(Iteration 4051 / 4900) loss: 1.926477  
(Iteration 4061 / 4900) loss: 1.860842  
(Iteration 4071 / 4900) loss: 1.850725  
(Iteration 4081 / 4900) loss: 1.751811  
(Iteration 4091 / 4900) loss: 1.806531  
(Iteration 4101 / 4900) loss: 1.786225  
(Iteration 4111 / 4900) loss: 1.902095  
(Iteration 4121 / 4900) loss: 1.813515  
(Iteration 4131 / 4900) loss: 1.846067  
(Iteration 4141 / 4900) loss: 1.877602  
(Iteration 4151 / 4900) loss: 1.812567  
(Iteration 4161 / 4900) loss: 1.753938  
(Iteration 4171 / 4900) loss: 1.811979  
(Iteration 4181 / 4900) loss: 2.050289  
(Iteration 4191 / 4900) loss: 1.897956  
(Iteration 4201 / 4900) loss: 1.839796  
(Iteration 4211 / 4900) loss: 1.892059  
(Iteration 4221 / 4900) loss: 1.830836  
(Iteration 4231 / 4900) loss: 1.786535  
(Iteration 4241 / 4900) loss: 1.847452  
(Iteration 4251 / 4900) loss: 1.812838  
(Iteration 4261 / 4900) loss: 1.833551  
(Iteration 4271 / 4900) loss: 1.760989  
(Iteration 4281 / 4900) loss: 1.888192  
(Iteration 4291 / 4900) loss: 1.762609  
(Iteration 4301 / 4900) loss: 1.782405  
(Iteration 4311 / 4900) loss: 1.819507  
(Iteration 4321 / 4900) loss: 1.877829  
(Iteration 4331 / 4900) loss: 1.872674  
(Iteration 4341 / 4900) loss: 1.762979  
(Iteration 4351 / 4900) loss: 1.747081  
(Iteration 4361 / 4900) loss: 1.681507  
(Iteration 4371 / 4900) loss: 1.703866  
(Iteration 4381 / 4900) loss: 1.737593  
(Iteration 4391 / 4900) loss: 1.906464  
(Iteration 4401 / 4900) loss: 1.803606  
(Epoch 9 / 10) train acc: 0.334000; val\_acc: 0.357000  
(Iteration 4411 / 4900) loss: 1.804347  
(Iteration 4421 / 4900) loss: 1.834587  
(Iteration 4431 / 4900) loss: 1.823297  
(Iteration 4441 / 4900) loss: 1.735493  
(Iteration 4451 / 4900) loss: 1.796263  
(Iteration 4461 / 4900) loss: 1.790046  
(Iteration 4471 / 4900) loss: 1.751358  
(Iteration 4481 / 4900) loss: 1.791927  
(Iteration 4491 / 4900) loss: 1.908877  
(Iteration 4501 / 4900) loss: 1.850133  
(Iteration 4511 / 4900) loss: 1.819179

```
(Iteration 4521 / 4900) loss: 1.893194
(Iteration 4531 / 4900) loss: 1.761785
(Iteration 4541 / 4900) loss: 1.760283
(Iteration 4551 / 4900) loss: 1.849578
(Iteration 4561 / 4900) loss: 1.843952
(Iteration 4571 / 4900) loss: 1.801694
(Iteration 4581 / 4900) loss: 1.798573
(Iteration 4591 / 4900) loss: 1.671874
(Iteration 4601 / 4900) loss: 1.914171
(Iteration 4611 / 4900) loss: 1.856668
(Iteration 4621 / 4900) loss: 1.672959
(Iteration 4631 / 4900) loss: 1.842552
(Iteration 4641 / 4900) loss: 1.774273
(Iteration 4651 / 4900) loss: 1.791479
(Iteration 4661 / 4900) loss: 1.839304
(Iteration 4671 / 4900) loss: 1.913976
(Iteration 4681 / 4900) loss: 1.901859
(Iteration 4691 / 4900) loss: 1.925343
(Iteration 4701 / 4900) loss: 1.784213
(Iteration 4711 / 4900) loss: 1.788709
(Iteration 4721 / 4900) loss: 1.848446
(Iteration 4731 / 4900) loss: 1.903870
(Iteration 4741 / 4900) loss: 1.827938
(Iteration 4751 / 4900) loss: 1.789689
(Iteration 4761 / 4900) loss: 1.735562
(Iteration 4771 / 4900) loss: 1.759569
(Iteration 4781 / 4900) loss: 1.857208
(Iteration 4791 / 4900) loss: 1.935385
(Iteration 4801 / 4900) loss: 1.752231
(Iteration 4811 / 4900) loss: 1.823224
(Iteration 4821 / 4900) loss: 1.766268
(Iteration 4831 / 4900) loss: 1.712064
(Iteration 4841 / 4900) loss: 1.836687
(Iteration 4851 / 4900) loss: 1.844082
(Iteration 4861 / 4900) loss: 1.892904
(Iteration 4871 / 4900) loss: 1.906281
(Iteration 4881 / 4900) loss: 1.737576
(Iteration 4891 / 4900) loss: 1.904193
(Epoch 10 / 10) train acc: 0.343000; val_acc: 0.363000
Final training accuracy: 0.343
Final validation accuracy: 0.363
```

## 10 Debug the training

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

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies



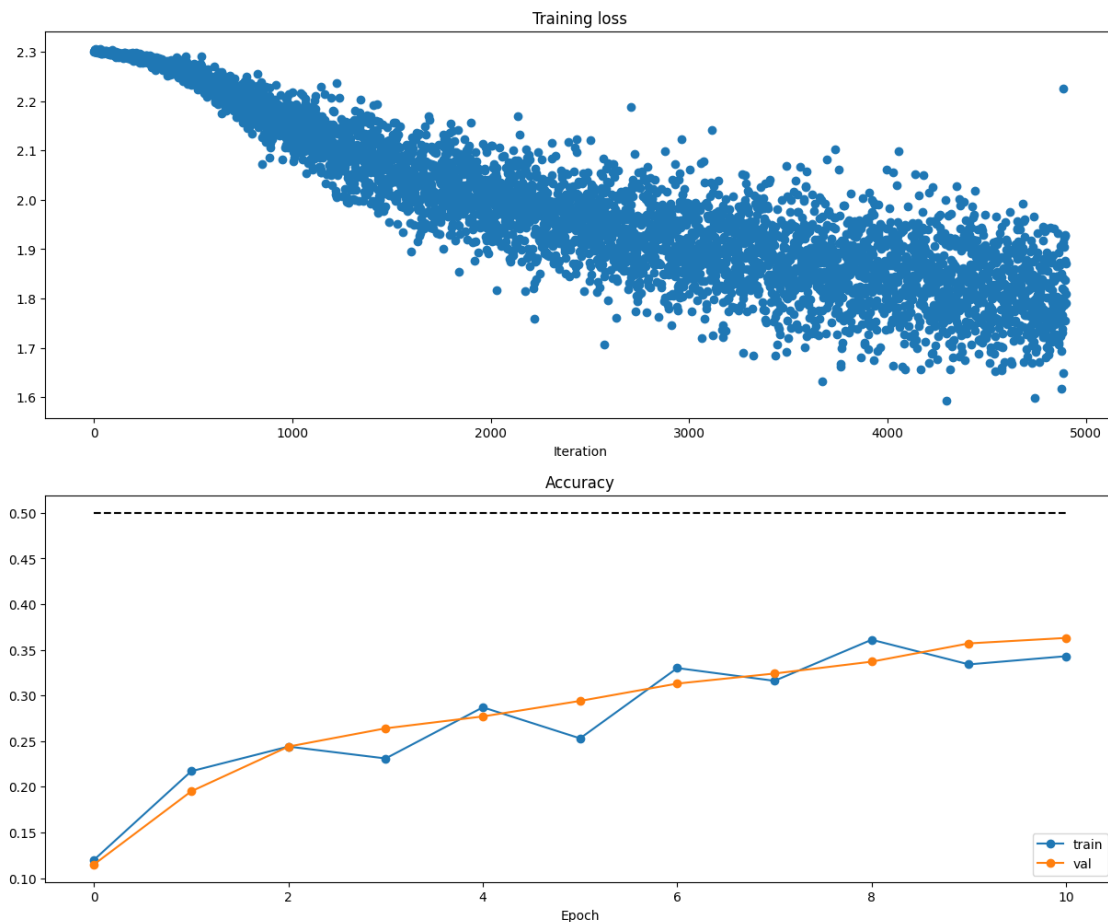
on the training and validation sets during optimization.

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

[27]: *# 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()
```

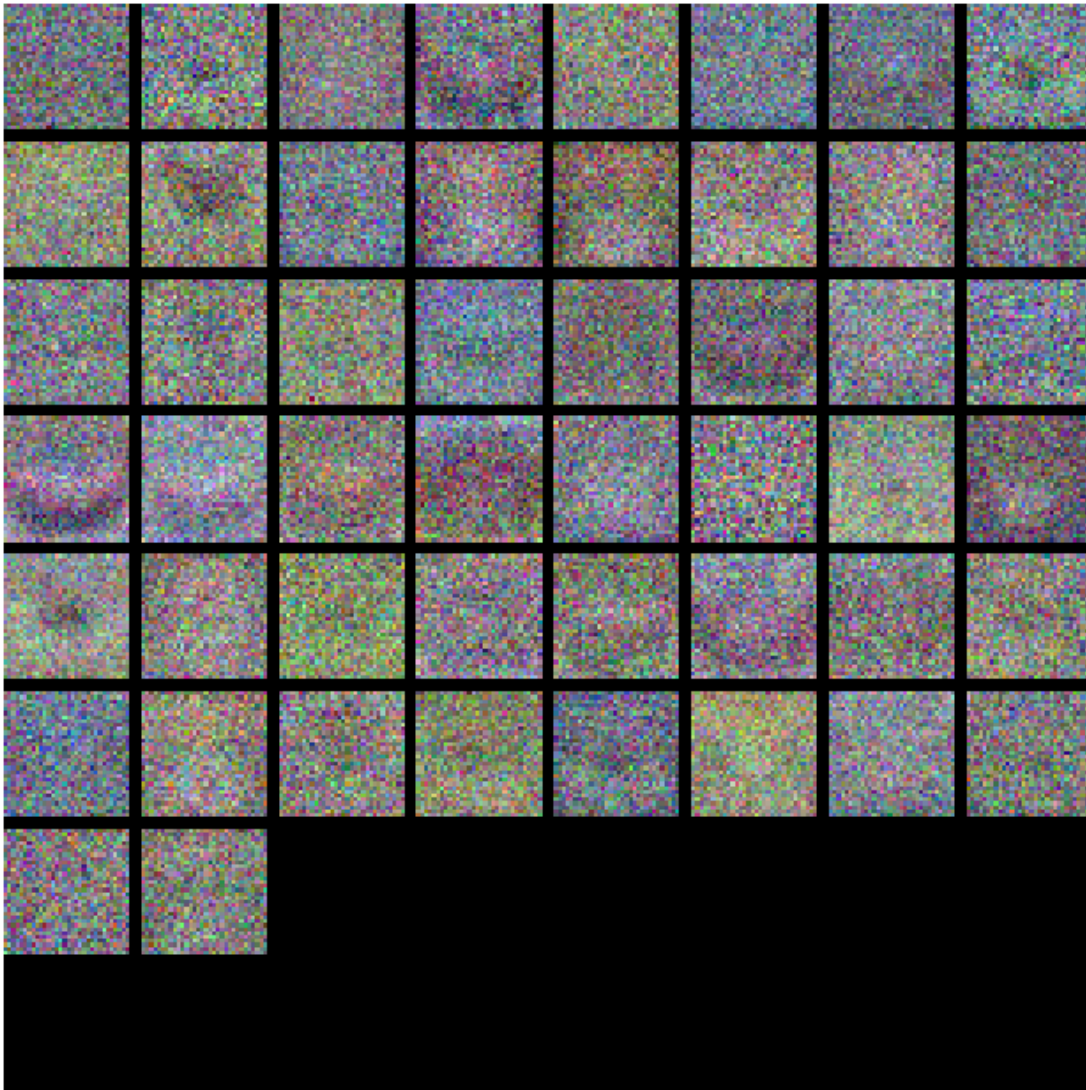


```
[28]: from icv83551.vis_utils import visualize_grid

# 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, number 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:** Your 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 to implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

```
[29]: best_model = None
      best_val_acc = 0

#####
# 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 the previous exercises.
↪ #
#####

# Define hyperparameter search space
learning_rates = [1e-3, 5e-4, 1e-4]
hidden_sizes = [100, 200, 300]
reg_strengths = [1e-3, 1e-4, 1e-5]

input_size = 32 * 32 * 3
num_classes = 10

results = {}

# Grid search over hyperparameters
for lr in learning_rates:
    for hidden_size in hidden_sizes:
        for reg in reg_strengths:
            print(f'Training with lr={lr}, hidden_size={hidden_size},
↪reg={reg}')

            model = TwoLayerNet(input_size, hidden_size, num_classes, reg=reg)

            solver = Solver(model, data,
                            optim_config={'learning_rate': lr},
                            lr_decay=0.95,
                            num_epochs=10,
                            batch_size=200,
                            print_every=1000,
                            verbose=False)

            solver.train()

            train_acc = solver.train_acc_history[-1]
            val_acc = solver.val_acc_history[-1]

            results[(lr, hidden_size, reg)] = (train_acc, val_acc)

            print(f'  Train acc: {train_acc:.4f}, Val acc: {val_acc:.4f}')

# Update best model if this one is better
if val_acc > best_val_acc:
    best_val_acc = val_acc
    best_model = model
    best_solver = solver

```

```

        best_params = (lr, hidden_size, reg)

print('\n' + '='*60)
print(f'Best validation accuracy: {best_val_acc:.4f}')
print(f'Best hyperparameters: lr={best_params[0]},  

      ↪hidden_size={best_params[1]}, reg={best_params[2]}')

#####
#                               END OF YOUR CODE                               #
#####

```

```

Training with lr=0.001, hidden_size=100, reg=0.001
  Train acc: 0.5970, Val acc: 0.5060
Training with lr=0.001, hidden_size=100, reg=0.0001
  Train acc: 0.6030, Val acc: 0.5010
Training with lr=0.001, hidden_size=100, reg=1e-05
  Train acc: 0.6080, Val acc: 0.5190
Training with lr=0.001, hidden_size=200, reg=0.001
  Train acc: 0.6310, Val acc: 0.5260
Training with lr=0.001, hidden_size=200, reg=0.0001
  Train acc: 0.6280, Val acc: 0.5260
Training with lr=0.001, hidden_size=200, reg=1e-05
  Train acc: 0.6050, Val acc: 0.5180
Training with lr=0.001, hidden_size=300, reg=0.001
  Train acc: 0.6190, Val acc: 0.5070
Training with lr=0.001, hidden_size=300, reg=0.0001
  Train acc: 0.6390, Val acc: 0.5160
Training with lr=0.001, hidden_size=300, reg=1e-05
  Train acc: 0.6290, Val acc: 0.5460
Training with lr=0.0005, hidden_size=100, reg=0.001
  Train acc: 0.5760, Val acc: 0.5120
Training with lr=0.0005, hidden_size=100, reg=0.0001
  Train acc: 0.5750, Val acc: 0.5040
Training with lr=0.0005, hidden_size=100, reg=1e-05
  Train acc: 0.5540, Val acc: 0.5110
Training with lr=0.0005, hidden_size=200, reg=0.001
  Train acc: 0.5680, Val acc: 0.5100
Training with lr=0.0005, hidden_size=200, reg=0.0001
  Train acc: 0.5700, Val acc: 0.5230
Training with lr=0.0005, hidden_size=200, reg=1e-05
  Train acc: 0.5440, Val acc: 0.5330
Training with lr=0.0005, hidden_size=300, reg=0.001
  Train acc: 0.5920, Val acc: 0.5050
Training with lr=0.0005, hidden_size=300, reg=0.0001
  Train acc: 0.6020, Val acc: 0.5230
Training with lr=0.0005, hidden_size=300, reg=1e-05
  Train acc: 0.5840, Val acc: 0.5240

```

```

Training with lr=0.0001, hidden_size=100, reg=0.001
  Train acc: 0.4180, Val acc: 0.4370
Training with lr=0.0001, hidden_size=100, reg=0.0001
  Train acc: 0.4360, Val acc: 0.4400
Training with lr=0.0001, hidden_size=100, reg=1e-05
  Train acc: 0.4460, Val acc: 0.4300
Training with lr=0.0001, hidden_size=200, reg=0.001
  Train acc: 0.4540, Val acc: 0.4520
Training with lr=0.0001, hidden_size=200, reg=0.0001
  Train acc: 0.4290, Val acc: 0.4410
Training with lr=0.0001, hidden_size=200, reg=1e-05
  Train acc: 0.4580, Val acc: 0.4450
Training with lr=0.0001, hidden_size=300, reg=0.001
  Train acc: 0.4710, Val acc: 0.4520
Training with lr=0.0001, hidden_size=300, reg=0.0001
  Train acc: 0.4600, Val acc: 0.4400
Training with lr=0.0001, hidden_size=300, reg=1e-05
  Train acc: 0.4160, Val acc: 0.4480

```

```

=====
Best validation accuracy: 0.5460
Best hyperparameters: lr=0.001, hidden_size=300, reg=1e-05

```

## 12 Test your model!

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

```

[30]: 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:  0.546

```

```

[31]: 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:  0.524

```

```

[32]: # Save best model
      best_model.save("best_two_layer_net.npy")

```

```

best_two_layer_net.npy saved.

```

```

[33]: from icv83551.vis_utils import visualize_grid

      # Visualize the weights of the network

      def show_net_weights(net):

```

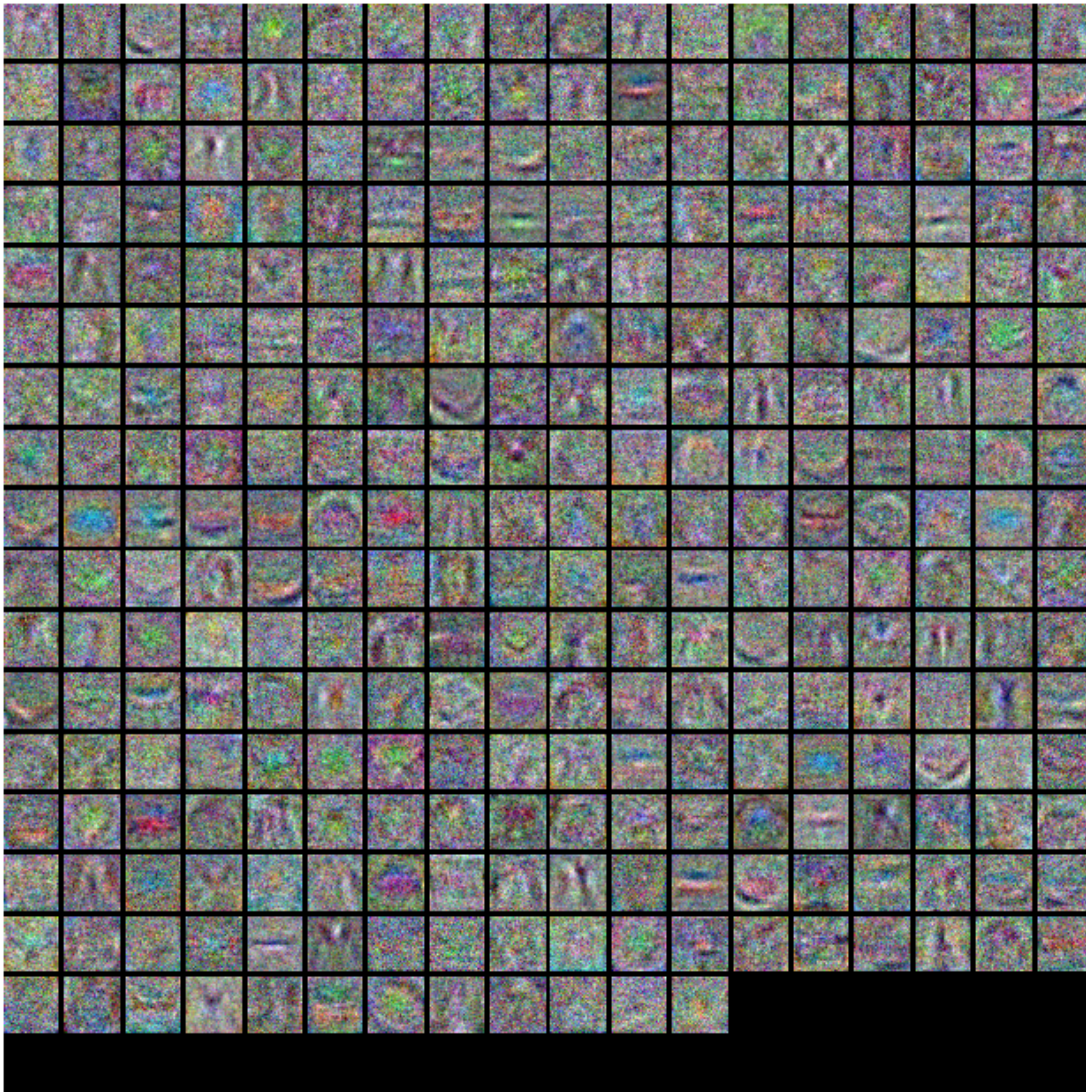


```

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(best_model)

```



### 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.

*Your Answer :*

1 and 3

*Your Explanation :*

overfitting means the network capacity is too large for the training set, and the model is able to memorize the training set instead of learning valuable patterns. to mitigate overfitting, we want to make memorization harder by either enlarging the training set or decreasing the model's capacity. Thus, the right answers are:

1 - larger dataset makes it harder to memorize

3 - increasing regularization penalizes complex solutions (larger weights values), thus constrains the effective capacity.

[ ]:

[ ]:



# features

January 7, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment1/'
FOLDERNAME = 'icv83551/assignments/assignment1/'
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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 1 Image features exercise

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

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

```
[ ]: import random
import numpy as np
from icv83551.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'

# %load_ext autoreload
# %autoreload 2
```

## 1.1 Load data

Similar to previous exercises, we will load CIFAR-10 data from disk.

```
[2]: from icv83551.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'icv83551/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may
    # cause memory issue)
    try:
        del X_train, y_train
        del X_test, y_test
        print('Clear previously loaded data.')
    except:
        pass

    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]

    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_CIFAR10_data()
```

## 1.2 Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your own interest.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The `extract_features` function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

```
[3]: from icv83551.features import extract_features, hog_feature, color_histogram_hsv

# num_color_bins = 10 # Number of bins in the color histogram
num_color_bins = 25 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,
    nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
```

[illegible]

### 1.3 Train Softmax classifier on features

Using the Softmax code developed earlier in the assignment, train Softmax classifiers on top of the features extracted above; this should achieve better results than training them directly on top of raw pixels.

```
[4]: # Use the validation set to tune the learning rate and regularization strength

from icv83551.classifiers.linear_classifier import Softmax

learning_rates = [1e-11, 1e-10, 1e-9, 1e-8, 1e-7, 1e-6]
regularization_strengths = [5e3, 5e4, 5e5, 5e6, 5e7]
best_softmax = None
best_val = -1
results = {}

for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train_feats, y_train, learning_rate=lr, reg=reg,
                      num_iters=5000, verbose=False)
        y_train_pred = softmax.predict(X_train_feats)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = softmax.predict(X_val_feats)
        val_accuracy = np.mean(y_val == y_val_pred)
        results[(lr, reg)] = (train_accuracy, val_accuracy)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax
            print('New best validation accuracy: %f' % best_val)

#####
# 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 Softmax; save#
# the best trained classifier in best_softmax. If you carefully tune the model, #
# you should be able to get accuracy of above 0.42 on the validation set.      #
#####

# Print out results.
for lr, reg in sorted(results, key=lambda x: results[x][1], reverse=True):
    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: %f' % best_val)
```

```
New best validation accuracy: 0.088000
New best validation accuracy: 0.091000
New best validation accuracy: 0.106000
New best validation accuracy: 0.132000
New best validation accuracy: 0.147000
New best validation accuracy: 0.427000
New best validation accuracy: 0.431000
New best validation accuracy: 0.434000
```

```
/Users/manela/PycharmProjects/computer_vision_course/assignment1/icv83551/classifiers/softmax.py:68: RuntimeWarning: divide by zero encountered in log
```

```
logp = np.log(p)
```

```
/Users/manela/PycharmProjects/computer_vision_course/assignment1/icv83551/classifiers/softmax.py:69: RuntimeWarning: overflow encountered in scalar multiply
```

```
loss = -logp[np.arange(X.shape[0]), y].mean() + reg * np.sum(W * W)
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/numpy/_core/fromnumeric.py:86: RuntimeWarning: overflow encountered in reduce
```

```
return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
```

```
/Users/manela/PycharmProjects/computer_vision_course/assignment1/icv83551/classifiers/softmax.py:69: RuntimeWarning: overflow encountered in multiply
```

```
loss = -logp[np.arange(X.shape[0]), y].mean() + reg * np.sum(W * W)
```

```
/Users/manela/PycharmProjects/computer_vision_course/assignment1/icv83551/classifiers/softmax.py:72: RuntimeWarning: overflow encountered in multiply
```

```
dW = X.T.dot(dW) / X.shape[0] + 2 * reg * W
```

```
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.419245 val accuracy: 0.434000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.423878 val accuracy: 0.431000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.413245 val accuracy: 0.431000
lr 1.000000e-10 reg 5.000000e+07 train accuracy: 0.422490 val accuracy: 0.427000
lr 1.000000e-06 reg 5.000000e+03 train accuracy: 0.423816 val accuracy: 0.418000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.420000 val accuracy: 0.415000
lr 1.000000e-09 reg 5.000000e+07 train accuracy: 0.415510 val accuracy: 0.415000
lr 1.000000e-06 reg 5.000000e+04 train accuracy: 0.411612 val accuracy: 0.411000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.418245 val accuracy: 0.407000
lr 1.000000e-08 reg 5.000000e+07 train accuracy: 0.332592 val accuracy: 0.326000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.307531 val accuracy: 0.316000
lr 1.000000e-06 reg 5.000000e+05 train accuracy: 0.317980 val accuracy: 0.307000
lr 1.000000e-07 reg 5.000000e+03 train accuracy: 0.255878 val accuracy: 0.260000
lr 1.000000e-10 reg 5.000000e+06 train accuracy: 0.123204 val accuracy: 0.147000
lr 1.000000e-10 reg 5.000000e+03 train accuracy: 0.119980 val accuracy: 0.132000
lr 1.000000e-10 reg 5.000000e+05 train accuracy: 0.115265 val accuracy: 0.124000
lr 1.000000e-08 reg 5.000000e+03 train accuracy: 0.118020 val accuracy: 0.114000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.114184 val accuracy: 0.113000
lr 1.000000e-11 reg 5.000000e+05 train accuracy: 0.109163 val accuracy: 0.106000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.093776 val accuracy: 0.102000
lr 1.000000e-11 reg 5.000000e+04 train accuracy: 0.094633 val accuracy: 0.091000
lr 1.000000e-10 reg 5.000000e+04 train accuracy: 0.093653 val accuracy: 0.090000
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.087000 val accuracy: 0.090000
```

```

lr 1.000000e-11 reg 5.000000e+03 train accuracy: 0.095429 val accuracy: 0.088000
lr 1.000000e-11 reg 5.000000e+07 train accuracy: 0.095653 val accuracy: 0.087000
lr 1.000000e-07 reg 5.000000e+07 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-06 reg 5.000000e+06 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-06 reg 5.000000e+07 train accuracy: 0.100265 val accuracy: 0.087000
lr 1.000000e-11 reg 5.000000e+06 train accuracy: 0.095224 val accuracy: 0.080000
lr 1.000000e-09 reg 5.000000e+03 train accuracy: 0.074449 val accuracy: 0.072000
best validation accuracy achieved: 0.434000

```

```

[5]: # Evaluate your trained Softmax on the test set: you should be able to get at least 0.42
y_test_pred = best_softmax.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)

```

0.421

```

[6]: # Save best softmax model
best_softmax.save("best_softmax_features.npy")

```

best\_softmax\_features.npy saved.

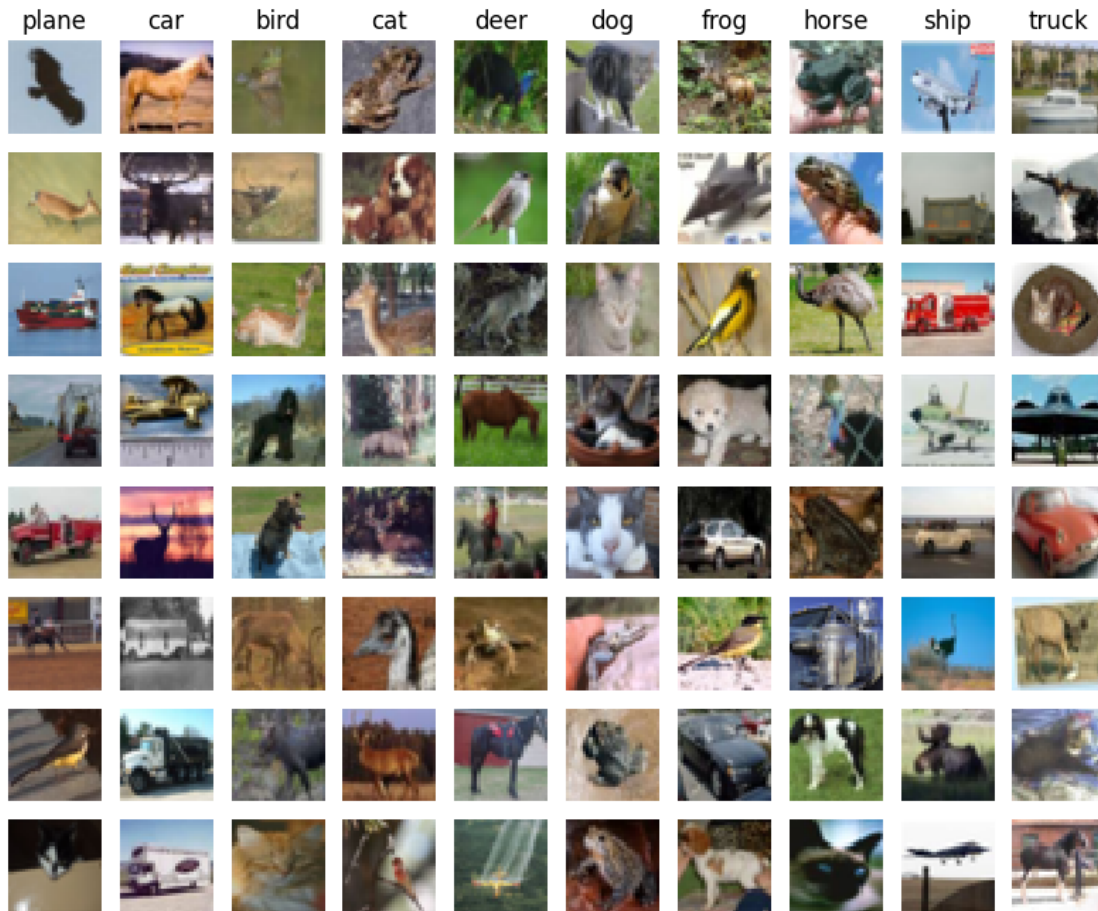
```

[7]: # An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls + 1)

        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()

```



### 1.3.1 Inline question 1:

Describe the misclassification results that you see. Do they make sense?

*Your Answer :*

The model uses linear rules on the color and hog features. that means that even a single feature which strongly agrees (match) with the wrong class might change the prediction. Looking at the misclassification images, I can see some samples that makes alot of sense, such as:

a bird with open wings classified as a plane

trucks that are classified as cars and vice versa

images with blue background that are classified as ships ets..

## 1.4 Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.



For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

```
[8]: # Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 170)
```

```
(49000, 169)
```

```
[9]: from icv83551.classifiers.fc_net import TwoLayerNet
from icv83551.solver import Solver

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

data = {
    'X_train': X_train_feats,
    'y_train': y_train,
    'X_val': X_val_feats,
    'y_val': y_val,
    'X_test': X_test_feats,
    'y_test': y_test,
}

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

#####
# TODO: Train a two-layer neural network on image features. You may want to #
# cross-validate various parameters as in previous sections. Store your best #
# model in the best_net variable. #####
# Define hyperparameter search space
learning_rates = [5e-1, 2e-1, 1e-1, 3e-2]
hidden_sizes = [300, 500]
reg_strengths = [1e-3, 1e-4, 1e-5]

results = {}
best_val_acc = -1
```

```

# Grid search over hyperparameters
for lr in learning_rates:
    for hidden_size in hidden_sizes:
        for reg in reg_strengths:
            print(f'Training with lr={lr}, hidden_size={hidden_size},  

↪ reg={reg}')

            model = TwoLayerNet(input_dim, hidden_size, num_classes, reg=reg)

            solver = Solver(model, data,
                            optim_config={'learning_rate': lr},
                            lr_decay=0.95,
                            num_epochs=200,
                            batch_size=200,
                            print_every=1000,
                            verbose=False)

            solver.train()

            train_acc = solver.train_acc_history[-1]
            val_acc = solver.val_acc_history[-1]

            results[(lr, hidden_size, reg)] = (train_acc, val_acc)

            print(f'  Train acc: {train_acc:.4f}, Val acc: {val_acc:.4f}')

            # Update best model if this one is better
            if val_acc > best_val_acc:
                best_val_acc = val_acc
                best_model = model
                best_solver = solver
                best_params = (lr, hidden_size, reg)

print('\n' + '='*60)
print(f'Best validation accuracy: {best_val_acc:.4f}')
print(f'Best hyperparameters: lr={best_params[0]},  

↪ hidden_size={best_params[1]}, reg={best_params[2]}')

```

```

Training with lr=0.5, hidden_size=300, reg=0.001
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.8850, Val acc: 0.6020
Training with lr=0.5, hidden_size=300, reg=0.0001
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.9100, Val acc: 0.5900
Training with lr=0.5, hidden_size=300, reg=1e-05
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.8980, Val acc: 0.5750

```

Training with lr=0.5, hidden\_size=500, reg=0.001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.9170, Val acc: 0.6160  
 Training with lr=0.5, hidden\_size=500, reg=0.0001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.9610, Val acc: 0.5900  
 Training with lr=0.5, hidden\_size=500, reg=1e-05  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.9230, Val acc: 0.5830  
 Training with lr=0.2, hidden\_size=300, reg=0.001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.7170, Val acc: 0.5980  
 Training with lr=0.2, hidden\_size=300, reg=0.0001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.7730, Val acc: 0.5900  
 Training with lr=0.2, hidden\_size=300, reg=1e-05  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.8240, Val acc: 0.6220  
 Training with lr=0.2, hidden\_size=500, reg=0.001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.8510, Val acc: 0.6010  
 Training with lr=0.2, hidden\_size=500, reg=0.0001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.8470, Val acc: 0.6090  
 Training with lr=0.2, hidden\_size=500, reg=1e-05  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.8180, Val acc: 0.5970  
 Training with lr=0.1, hidden\_size=300, reg=0.001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.6730, Val acc: 0.6040  
 Training with lr=0.1, hidden\_size=300, reg=0.0001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.6890, Val acc: 0.5940  
 Training with lr=0.1, hidden\_size=300, reg=1e-05  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.6990, Val acc: 0.6160  
 Training with lr=0.1, hidden\_size=500, reg=0.001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.7040, Val acc: 0.6050  
 Training with lr=0.1, hidden\_size=500, reg=0.0001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.7090, Val acc: 0.6090  
 Training with lr=0.1, hidden\_size=500, reg=1e-05  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.7240, Val acc: 0.6080  
 Training with lr=0.03, hidden\_size=300, reg=0.001  
 Validation accuracy has not improved for 30 epochs. Stopping early.  
 Train acc: 0.5610, Val acc: 0.5550

```

Training with lr=0.03, hidden_size=300, reg=0.0001
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.6010, Val acc: 0.5530
Training with lr=0.03, hidden_size=300, reg=1e-05
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.5790, Val acc: 0.5420
Training with lr=0.03, hidden_size=500, reg=0.001
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.5400, Val acc: 0.5340
Training with lr=0.03, hidden_size=500, reg=0.0001
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.5620, Val acc: 0.5470
Training with lr=0.03, hidden_size=500, reg=1e-05
Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.5750, Val acc: 0.5520

```

```

=====
Best validation accuracy: 0.6220
Best hyperparameters: lr=0.2, hidden_size=300, reg=1e-05

```

```

[10]: best_net = TwoLayerNet(input_dim, best_params[1], num_classes,
    ↪reg=best_params[2])
solver = Solver(best_net, data,
    optim_config={'learning_rate': best_params[0]},
    lr_decay=0.95,
    num_epochs=200,
    batch_size=200,
    print_every=1000,
    verbose=False)
solver.train()

train_acc = solver.train_acc_history[-1]
val_acc = solver.val_acc_history[-1]

print(f'  Train acc: {train_acc:.4f}, Val acc: {val_acc:.4f}')

```

```

Validation accuracy has not improved for 30 epochs. Stopping early.
  Train acc: 0.7760, Val acc: 0.6070

```

```

[11]: # 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')

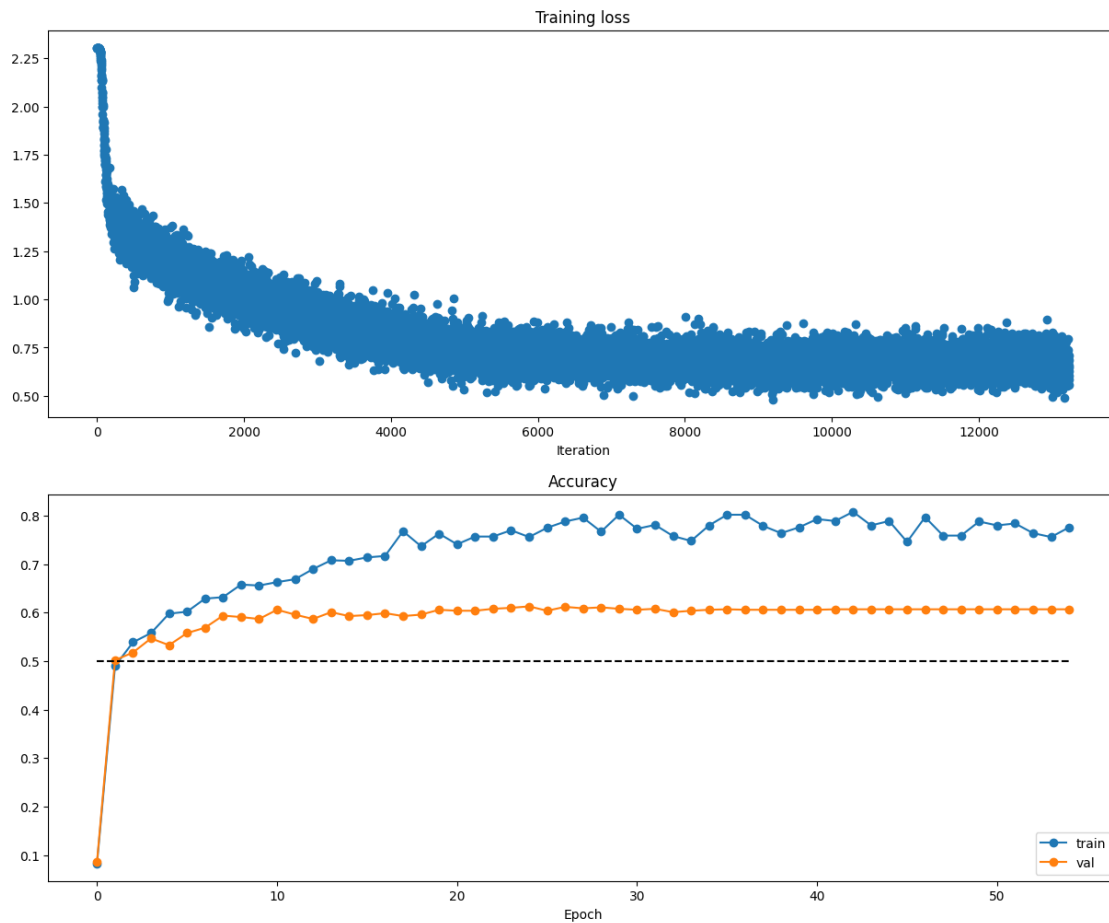
plt.subplot(2, 1, 2)

```

```

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()

```



[13]: *# Run your best neural net classifier on the test set. You should be able  
# to get more than 58% accuracy. It is also possible to get >60% accuracy  
# with careful tuning.*

```

y_test_pred = np.argmax(best_net.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)

```

0.615

```
[14]: # Save best model  
best_net.save("best_two_layer_net_features.npy")
```

best\_two\_layer\_net\_features.npy saved.

```
[ ]:
```

# FullyConnectedNets

January 7, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment1/'
FOLDERNAME = 'icv83551/assignments/assignment1/'
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/$FOLDERNAME/icv83551/datasets/
# %cd /content/drive/My\ Drive/$FOLDERNAME
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 1 Multi-Layer Fully Connected Network

In this exercise, you will implement a fully connected network with an arbitrary number of hidden layers.

```
[ ]: # from google.colab import drive
# drive.mount('/content/drive')
```

Read through the `FullyConnectedNet` class in the file `icv83551/classifiers/fc_net.py`.

Implement the network initialization, forward pass, and backward pass. Throughout this assignment, you will be implementing layers in `icv83551/layers.py`. You can re-use your implementations for `affine_forward`, `affine_backward`, `relu_forward`, `relu_backward`, and `softmax_loss` from before. For right now, don't worry about implementing dropout or batch/layer normalization yet, as you will add those features later.

```
[ ]: # Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from icv83551.classifiers.fc_net import *
from icv83551.data_utils import get_CIFAR10_data
from icv83551.gradient_check import eval_numerical_gradient, \
    eval_numerical_gradient_array
from icv83551.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"

# %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))))
```

```
[2]: # Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

## 1.1 Initial Loss and Gradient Check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. This is a good way to see if the initial losses seem reasonable.

For gradient checking, you should expect to see errors around  $1e-7$  or less.

```
[3]: np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
    print("Running check with reg = ", reg)
```



```

model = FullyConnectedNet(
    [H1, H2],
    input_dim=D,
    num_classes=C,
    reg=reg,
    weight_scale=5e-2,
    dtype=np.float64
)

loss, grads = model.loss(X, y)
print("Initial loss: ", loss)

# Most of the errors should be on the order of e-7 or smaller.
# NOTE: It is fine however to see an error for W2 on the order of e-5
# for the check when reg = 0.0
for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name],
    verbose=False, h=1e-5)
    print(f"{name} relative error: {rel_error(grad_num, grads[name])}")

```

```

Running check with reg = 0
Initial loss: 2.300479089768492
W1 relative error: 1.0252674374835853e-07
W2 relative error: 2.2120479343503063e-05
W3 relative error: 4.5623278699709044e-07
b1 relative error: 4.660094372886962e-09
b2 relative error: 2.085654276112763e-09
b3 relative error: 1.689724888469736e-10
Running check with reg = 3.14
Initial loss: 7.052114776533016
W1 relative error: 6.862884935858817e-09
W2 relative error: 9.627270211033322e-08
W3 relative error: 2.6171457581890515e-08
b1 relative error: 1.4752427323253937e-08
b2 relative error: 1.7223750761525226e-09
b3 relative error: 2.378772438198909e-10

```

As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy within 20 epochs.

```

[4]: # TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {

```

```

    "X_train": data["X_train"][:num_train],
    "y_train": data["y_train"][:num_train],
    "X_val": data["X_val"],
    "y_val": data["y_val"],
}

weight_scale = 1e-2    # Experiment with this!
learning_rate = 1e-2   # Experiment with this!

model = FullyConnectedNet(
    [100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule="sgd",
    optim_config={"learning_rate": learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()

```

```

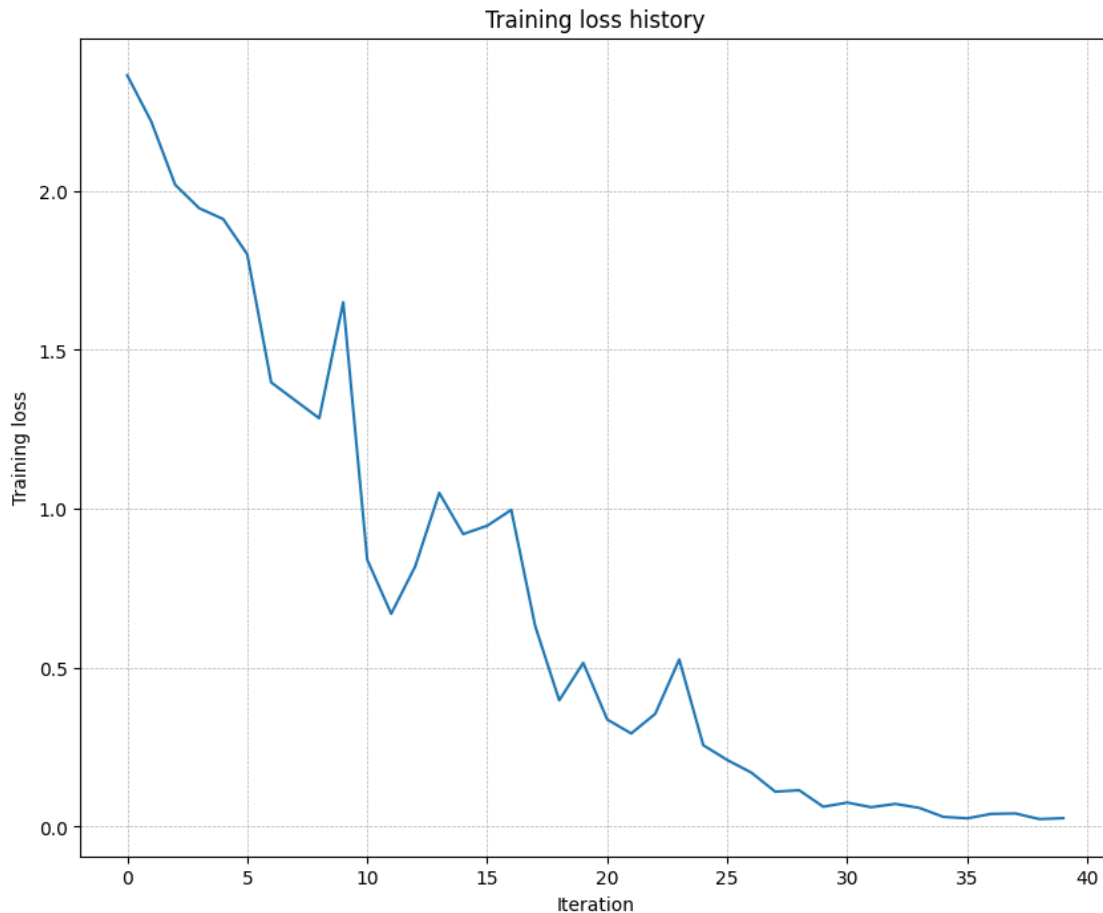
(Iteration 1 / 40) loss: 2.363364
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.108000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.127000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.172000
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.184000
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.181000
(Epoch 5 / 20) train acc: 0.740000; val_acc: 0.190000
(Iteration 11 / 40) loss: 0.839976
(Epoch 6 / 20) train acc: 0.740000; val_acc: 0.187000
(Epoch 7 / 20) train acc: 0.740000; val_acc: 0.183000
(Epoch 8 / 20) train acc: 0.820000; val_acc: 0.177000
(Epoch 9 / 20) train acc: 0.860000; val_acc: 0.200000
(Epoch 10 / 20) train acc: 0.920000; val_acc: 0.191000
(Iteration 21 / 40) loss: 0.337174

```

```

(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.189000
(Epoch 12 / 20) train acc: 0.940000; val_acc: 0.180000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.199000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.199000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.195000
(Iteration 31 / 40) loss: 0.075911
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.182000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.201000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.207000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.185000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.192000

```



Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

```

[5]: # TODO: Use a five-layer Net to overfit 50 training examples by
      # tweaking just the learning rate and initialization scale.

```

```

num_train = 50
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

# learning_rate = 1e-2 # Experiment with this!
# weight_scale = 1e-5 # Experiment with this!
weight_scale = 2 / 100 # Experiment with this!

overfitted = False
for weight_scale in [1e-3, 3e-3, 1e-2, 3e-2, 1e-1, 3e-1, 1]:
    if overfitted:
        break
    for learning_rate in [1e-4, 3e-4, 1e-3, 3e-3, 1e-2, 3e-2, 1e-1]:
        model = FullyConnectedNet(
            [100, 100, 100, 100],
            weight_scale=weight_scale,
            dtype=np.float64
        )
        solver = Solver(
            model,
            small_data,
            print_every=10,
            num_epochs=20,
            batch_size=25,
            update_rule='sgd',
            optim_config={'learning_rate': learning_rate},
            verbose=False,
        )
        solver.train()
        last_train_acc = sum(solver.train_acc_history[-3:]) / 3
        if last_train_acc == 1.0:
            print('Achieved 100% training accuracy!')
            print(f'Learning rate: {learning_rate}, Weight scale: {weight_scale},  

            ↪Final training accuracy: {last_train_acc}')
            overfitted = True
            break
            print(f'Learning rate: {learning_rate}, Weight scale: {weight_scale}, Final  

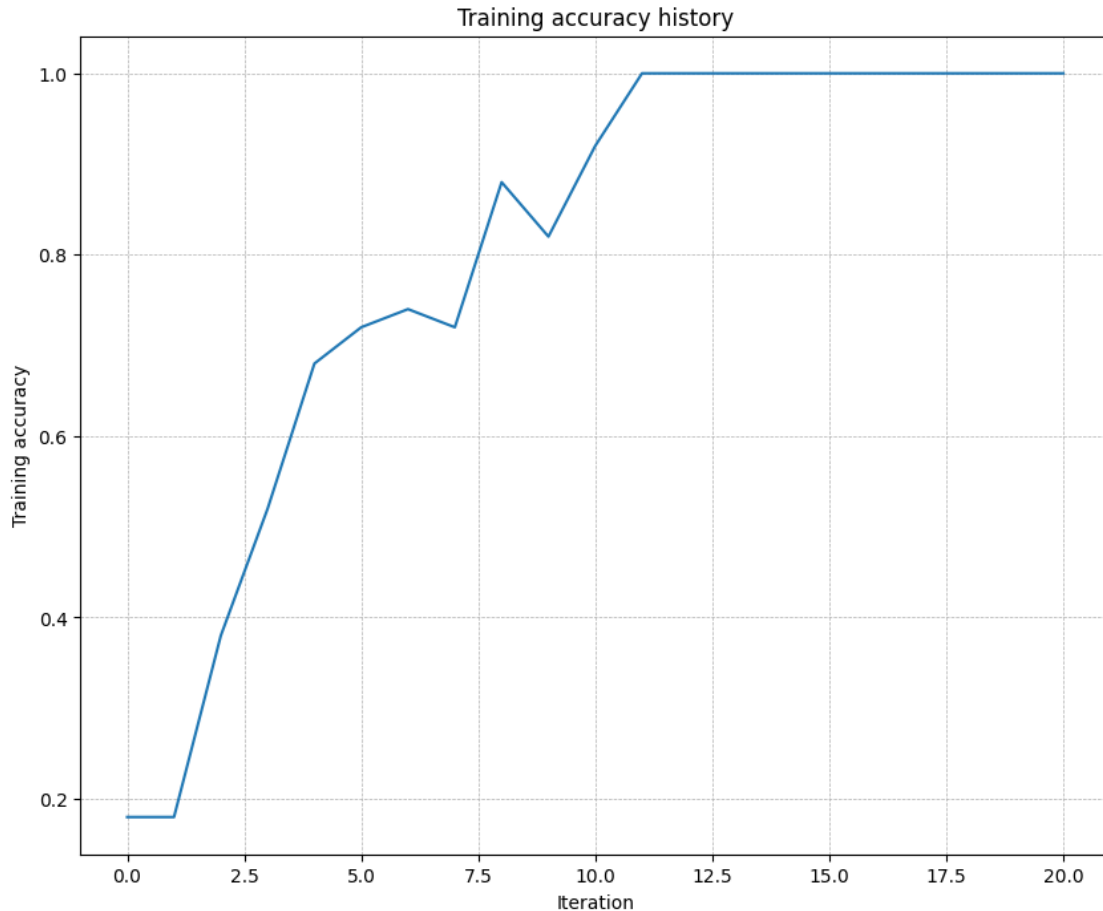
            ↪training accuracy: {last_train_acc}')

plt.plot(solver.train_acc_history)
plt.title('Training accuracy history')
plt.xlabel('Iteration')

```

```
plt.ylabel('Training accuracy')
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
Learning rate: 0.0001, Weight scale: 0.001, Final training accuracy: 0.16
Learning rate: 0.0003, Weight scale: 0.001, Final training accuracy: 0.16
Learning rate: 0.001, Weight scale: 0.001, Final training accuracy: 0.16
Learning rate: 0.003, Weight scale: 0.001, Final training accuracy: 0.16
Learning rate: 0.01, Weight scale: 0.001, Final training accuracy: 0.16
Learning rate: 0.03, Weight scale: 0.001, Final training accuracy: 0.16
Learning rate: 0.1, Weight scale: 0.001, Final training accuracy: 0.16
Learning rate: 0.0001, Weight scale: 0.003, Final training accuracy: 0.16
Learning rate: 0.0003, Weight scale: 0.003, Final training accuracy: 0.16
Learning rate: 0.001, Weight scale: 0.003, Final training accuracy: 0.16
Learning rate: 0.003, Weight scale: 0.003, Final training accuracy: 0.16
Learning rate: 0.01, Weight scale: 0.003, Final training accuracy: 0.16
Learning rate: 0.03, Weight scale: 0.003, Final training accuracy: 0.16
Learning rate: 0.1, Weight scale: 0.003, Final training accuracy: 0.16
Learning rate: 0.0001, Weight scale: 0.01, Final training accuracy:
0.04666666666666667
Learning rate: 0.0003, Weight scale: 0.01, Final training accuracy: 0.08
Learning rate: 0.001, Weight scale: 0.01, Final training accuracy: 0.16
Learning rate: 0.003, Weight scale: 0.01, Final training accuracy: 0.08
Learning rate: 0.01, Weight scale: 0.01, Final training accuracy: 0.16
Learning rate: 0.03, Weight scale: 0.01, Final training accuracy: 0.16
Learning rate: 0.1, Weight scale: 0.01, Final training accuracy: 0.16
Learning rate: 0.0001, Weight scale: 0.03, Final training accuracy: 0.16
Learning rate: 0.0003, Weight scale: 0.03, Final training accuracy: 0.16
Learning rate: 0.001, Weight scale: 0.03, Final training accuracy:
0.23333333333333333
Learning rate: 0.003, Weight scale: 0.03, Final training accuracy: 0.44
Learning rate: 0.01, Weight scale: 0.03, Final training accuracy:
0.8266666666666667
Learning rate: 0.03, Weight scale: 0.03, Final training accuracy: 0.5
Learning rate: 0.1, Weight scale: 0.03, Final training accuracy:
0.38000000000000006
Learning rate: 0.0001, Weight scale: 0.1, Final training accuracy:
0.8866666666666667
Achieved 100% training accuracy!
Learning rate: 0.0003, Weight scale: 0.1, Final training accuracy: 1.0
```



## 1.2 Inline Question 1:

Did you notice anything about the comparative difficulty of training the three-layer network vs. training the five-layer network? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

## 1.3 Answer:

It seems the five-layer network was significantly more sensitive to the initialization scale. The reason is that as you pass the input through more layers, the activations tend more to go either to 0 and die (due to the ReLU activation) or to get too high and diverge. Either way, it is harder to converge effectively and with the more layers you must choose your initialization scale more precisely.

## 2 Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

## 2.1 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent.

Open the file `icv83551/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than  $e-8$ .

```
[6]: from icv83551.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {"learning_rate": 1e-3, "velocity": v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
    [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
    [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
    [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
    [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
    [ 0.5406,      0.55475789,  0.56891579,  0.58307368,  0.59723158],
    [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
    [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
    [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print("next_w error: ", rel_error(next_w, expected_next_w))
print("velocity error: ", rel_error(expected_velocity, config["velocity"]))
```

```
next_w error:  8.882347033505819e-09
velocity error: 4.269287743278663e-09
```

Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

```
[7]: num_train = 4000
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

solvers = {}
```

```

for update_rule in ['sgd', 'sgd_momentum']:
    print('Running with ', update_rule)
    model = FullyConnectedNet(
        [100, 100, 100, 100, 100],
        weight_scale=5e-2
    )

    solver = Solver(
        model,
        small_data,
        num_epochs=5,
        batch_size=100,
        update_rule=update_rule,
        optim_config={'learning_rate': 5e-3},
        verbose=True,
    )
    solvers[update_rule] = solver
    solver.train()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"loss_{update_rule}")
    axes[1].plot(solver.train_acc_history, label=f"train_acc_{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"val_acc_{update_rule}")

for ax in axes:
    ax.legend(loc="best", ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()

```

```

Running with sgd
(Iteration 1 / 200) loss: 2.520819
(Epoch 0 / 5) train acc: 0.120000; val_acc: 0.124000
(Iteration 11 / 200) loss: 2.252506
(Iteration 21 / 200) loss: 2.246141
(Iteration 31 / 200) loss: 2.036052
(Epoch 1 / 5) train acc: 0.263000; val_acc: 0.243000

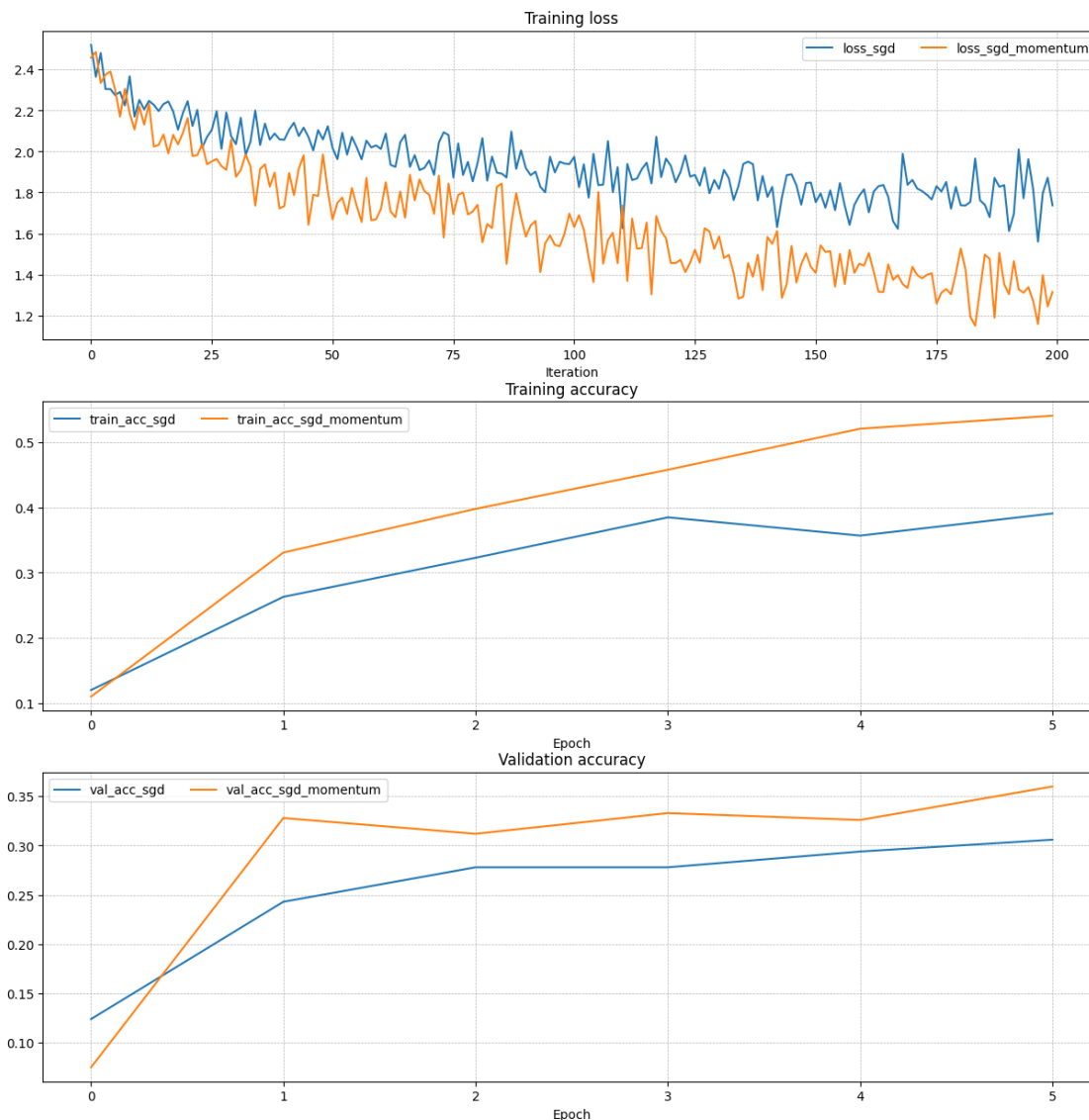
```



```

(Iteration 41 / 200) loss: 2.057502
(Iteration 51 / 200) loss: 2.017136
(Iteration 61 / 200) loss: 2.012591
(Iteration 71 / 200) loss: 1.956560
(Epoch 2 / 5) train acc: 0.323000; val_acc: 0.278000
(Iteration 81 / 200) loss: 1.944430
(Iteration 91 / 200) loss: 1.918957
(Iteration 101 / 200) loss: 1.974382
(Iteration 111 / 200) loss: 1.625976
(Epoch 3 / 5) train acc: 0.385000; val_acc: 0.278000
(Iteration 121 / 200) loss: 1.930750
(Iteration 131 / 200) loss: 1.817689
(Iteration 141 / 200) loss: 1.778752
(Iteration 151 / 200) loss: 1.752776
(Epoch 4 / 5) train acc: 0.357000; val_acc: 0.294000
(Iteration 161 / 200) loss: 1.816014
(Iteration 171 / 200) loss: 1.861751
(Iteration 181 / 200) loss: 1.737964
(Iteration 191 / 200) loss: 1.611947
(Epoch 5 / 5) train acc: 0.391000; val_acc: 0.306000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 2.458359
(Epoch 0 / 5) train acc: 0.110000; val_acc: 0.075000
(Iteration 11 / 200) loss: 2.219963
(Iteration 21 / 200) loss: 2.163423
(Iteration 31 / 200) loss: 1.876720
(Epoch 1 / 5) train acc: 0.331000; val_acc: 0.328000
(Iteration 41 / 200) loss: 1.733695
(Iteration 51 / 200) loss: 1.668764
(Iteration 61 / 200) loss: 1.719714
(Iteration 71 / 200) loss: 1.789270
(Epoch 2 / 5) train acc: 0.398000; val_acc: 0.312000
(Iteration 81 / 200) loss: 1.740063
(Iteration 91 / 200) loss: 1.584311
(Iteration 101 / 200) loss: 1.632208
(Iteration 111 / 200) loss: 1.736153
(Epoch 3 / 5) train acc: 0.458000; val_acc: 0.333000
(Iteration 121 / 200) loss: 1.455618
(Iteration 131 / 200) loss: 1.586691
(Iteration 141 / 200) loss: 1.582863
(Iteration 151 / 200) loss: 1.407608
(Epoch 4 / 5) train acc: 0.521000; val_acc: 0.326000
(Iteration 161 / 200) loss: 1.442950
(Iteration 171 / 200) loss: 1.437285
(Iteration 181 / 200) loss: 1.526296
(Iteration 191 / 200) loss: 1.303706
(Epoch 5 / 5) train acc: 0.541000; val_acc: 0.360000

```



## 2.2 RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `icv83551/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

**NOTE:** Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.” COURSE: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, “Adam: A Method for Stochastic Optimization”, ICLR 2015.

```
[8]: # Test RMSProp implementation
from icv83551.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
    [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
    [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
    [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
    [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
    [ 0.5976,      0.6126277,   0.6277108,   0.64284931,   0.65804321],
    [ 0.67329252,  0.68859723,   0.70395734,   0.71937285,   0.73484377],
    [ 0.75037008,  0.7659518,    0.78158892,   0.79728144,   0.81302936],
    [ 0.82883269,  0.84469141,   0.86060554,   0.87657507,   0.8926    ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

next\_w error: 9.524687511038133e-08

cache error: 2.6477955807156126e-09

```
[11]: # Test Adam implementation
from icv83551.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
    [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
    [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
    [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
    [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
```

```

expected_v = np.asarray([
    [ 0.69966,      0.68908382,  0.67851319,  0.66794809,  0.65738853,],
    [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
    [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
    [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
    [ 0.48,          0.49947368,  0.51894737,  0.53842105,  0.55789474],
    [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
    [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
    [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85         ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))

```

```

next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09

```

Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

```

[12]: learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
    print('Running with ', update_rule)
    model = FullyConnectedNet(
        [100, 100, 100, 100, 100],
        weight_scale=5e-2
    )
    solver = Solver(
        model,
        small_data,
        num_epochs=5,
        batch_size=100,
        update_rule=update_rule,
        optim_config={'learning_rate': learning_rates[update_rule]},
        verbose=True
    )
    solvers[update_rule] = solver
    solver.train()
    print()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')

```

```

axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"{update_rule}")
    axes[1].plot(solver.train_acc_history, label=f"{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"{update_rule}")

for ax in axes:
    ax.legend(loc='best', ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()

```

Running with adam

```

(Iteration 1 / 200) loss: 2.730335
(Epoch 0 / 5) train acc: 0.155000; val_acc: 0.151000
(Iteration 11 / 200) loss: 2.047510
(Iteration 21 / 200) loss: 2.038886
(Iteration 31 / 200) loss: 2.024488
(Epoch 1 / 5) train acc: 0.354000; val_acc: 0.299000
(Iteration 41 / 200) loss: 1.682610
(Iteration 51 / 200) loss: 1.524325
(Iteration 61 / 200) loss: 1.868994
(Iteration 71 / 200) loss: 1.527333
(Epoch 2 / 5) train acc: 0.407000; val_acc: 0.351000
(Iteration 81 / 200) loss: 1.609008
(Iteration 91 / 200) loss: 1.504935
(Iteration 101 / 200) loss: 1.609296
(Iteration 111 / 200) loss: 1.513047
(Epoch 3 / 5) train acc: 0.513000; val_acc: 0.366000
(Iteration 121 / 200) loss: 1.407577
(Iteration 131 / 200) loss: 1.626030
(Iteration 141 / 200) loss: 1.224390
(Iteration 151 / 200) loss: 1.487880
(Epoch 4 / 5) train acc: 0.549000; val_acc: 0.375000
(Iteration 161 / 200) loss: 1.319126
(Iteration 171 / 200) loss: 1.207846
(Iteration 181 / 200) loss: 1.221166
(Iteration 191 / 200) loss: 1.151690
(Epoch 5 / 5) train acc: 0.573000; val_acc: 0.377000

```

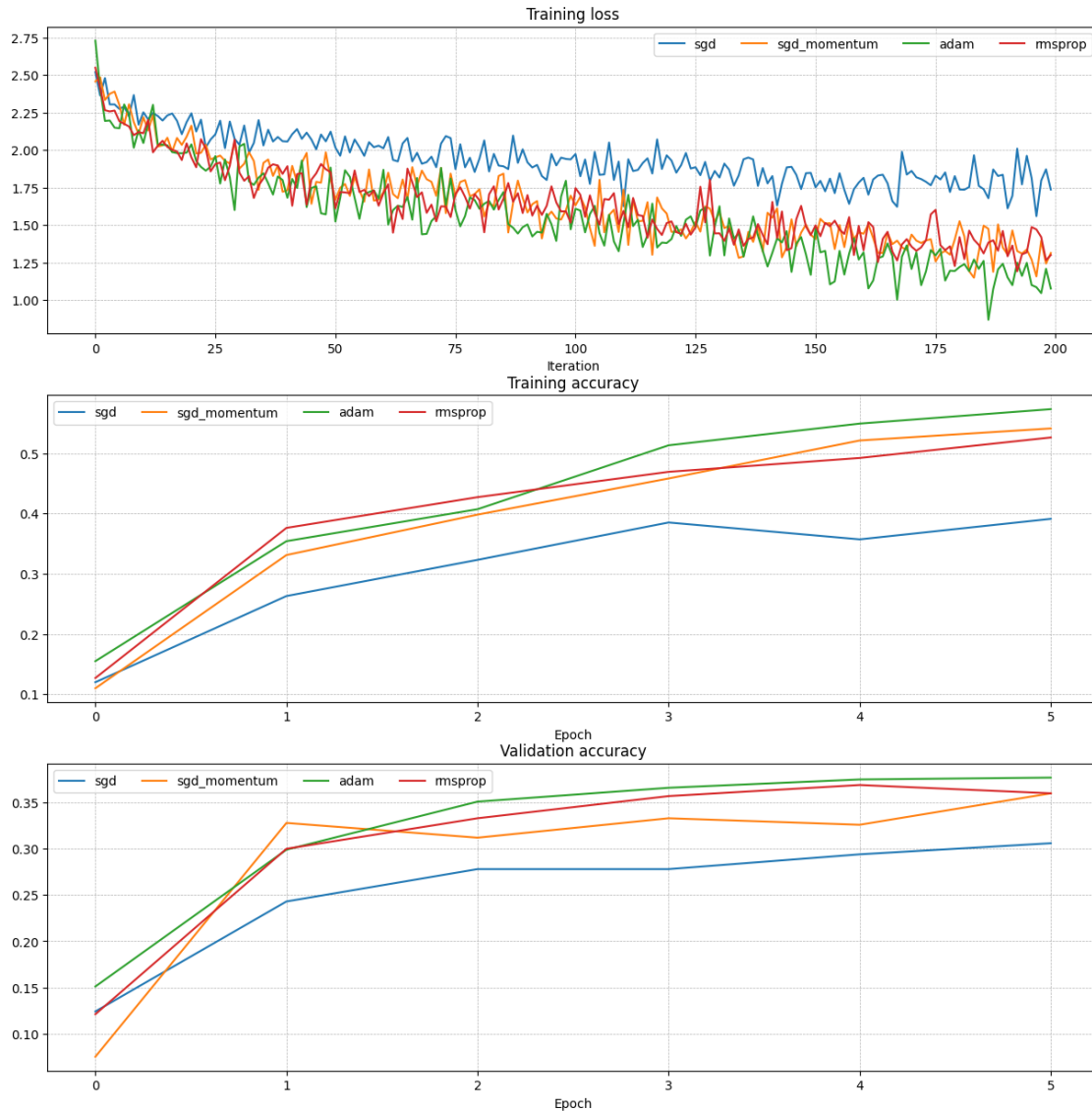
Running with rmsprop

```

(Iteration 1 / 200) loss: 2.549949
(Epoch 0 / 5) train acc: 0.127000; val_acc: 0.121000
(Iteration 11 / 200) loss: 2.112888

```

(Iteration 21 / 200) loss: 1.952220  
(Iteration 31 / 200) loss: 1.849937  
(Epoch 1 / 5) train acc: 0.376000; val\_acc: 0.300000  
(Iteration 41 / 200) loss: 1.893265  
(Iteration 51 / 200) loss: 1.612033  
(Iteration 61 / 200) loss: 1.713402  
(Iteration 71 / 200) loss: 1.636767  
(Epoch 2 / 5) train acc: 0.427000; val\_acc: 0.333000  
(Iteration 81 / 200) loss: 1.666341  
(Iteration 91 / 200) loss: 1.700850  
(Iteration 101 / 200) loss: 1.747257  
(Iteration 111 / 200) loss: 1.595460  
(Epoch 3 / 5) train acc: 0.469000; val\_acc: 0.357000  
(Iteration 121 / 200) loss: 1.525150  
(Iteration 131 / 200) loss: 1.445891  
(Iteration 141 / 200) loss: 1.372794  
(Iteration 151 / 200) loss: 1.498672  
(Epoch 4 / 5) train acc: 0.492000; val\_acc: 0.369000  
(Iteration 161 / 200) loss: 1.332871  
(Iteration 171 / 200) loss: 1.356975  
(Iteration 181 / 200) loss: 1.420717  
(Iteration 191 / 200) loss: 1.292355  
(Epoch 5 / 5) train acc: 0.526000; val\_acc: 0.360000



## 2.3 Inline Question 2:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

## 2.4 Answer:

The cache is not normalized, nor averaged. the cache is defined as the sum of squares of all previous gradients. As the gradients are squared, the cache grows monotonically at each step throughout the training. The denominator ( $\sqrt{\text{cache}} + \epsilon$ ) gets very large, and the thus update gets very small.

Adam will not have this problem, as the effective  $v$  divider is an EMA estimator, rather than a simple sum, thus not growing vastly over time.

## 3 Train a Good Model!

Train the best fully connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully connected network.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the next assignment, we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional networks rather than fully connected networks.

**Note:** In the next assignment, you will learn techniques like BatchNormalization and Dropout which can help you train powerful models.

```
[44]: best_model = None

#####
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might #
# find batch/layer normalization and dropout useful. Store your best model in #
# the best_model variable. #
#####
# ****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)****

# Define hyperparameter search space
# learning_rates = [1e-1, 1e-2, 1e-3]
learning_rates = [1e-3]
hidden_sizes = [200]
reg_strengths = [0.01]
num_hiddens = [3]
weight_scale = [1e-2, 1e-3, 1e-4]

input_size = 32 * 32 * 3
num_classes = 10

best_val_acc = -1
results = {}

def kaiming(size): return np.sqrt(2.0 / size)

# Grid search over hyperparameters
```



```

for scale in weight_scale:
    for lr in learning_rates:
        for hidden_size in hidden_sizes:
            for reg in reg_strengths:
                for n_hiddens in num_hiddens:
                    print(f'Training with lr={lr}, hidden_size={hidden_size},
↪n_hiddens={n_hiddens}, reg={reg}')
                    model = FullyConnectedNet(
                        input_dim=input_size,
                        hidden_dims=[hidden_size] * n_hiddens,
                        # weight_scale=[kaiming(hidden_size if i > 0 else
↪input_size) for i in range(n_hiddens+1)],
                        weight_scale=scale,
                        dtype=np.float32,
                    )

                    solver = Solver(model, data,
                                    optim_config={'learning_rate': lr},
                                    lr_decay=0.95,
                                    num_epochs=200,
                                    batch_size=200,
                                    print_every=1000,
                                    update_rule='adam',
                                    verbose=False)

                    solver.train()

                    train_acc = solver.train_acc_history[-1]
                    val_acc = solver.val_acc_history[-1]

                    results[(lr, hidden_size, reg)] = (train_acc, val_acc)

                    print(f'  Train acc: {train_acc:.4f}, Val acc: {val_acc:.
↪4f}')

                    # Update best model if this one is better
                    if val_acc > best_val_acc:
                        best_val_acc = val_acc
                        best_model = model
                        best_solver = solver
                        best_params = (lr, hidden_size, reg, n_hiddens)

print('\n' + '='*60)
print(f'Best validation accuracy: {best_val_acc:.4f}')
print(f'Best hyperparameters: lr={best_params[0]},
↪hidden_size={best_params[1]}, reg={best_params[2]}')

```

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

Training with lr=0.001, hidden\_size=200, n\_hiddens=3, reg=0.01

Training: 7%| | 3304/49000 [00:19<04:03, 187.81it/s]/Users/manela/Pyc  
harmProjects/computer\_vision\_course/assignment1/icv83551/layers.py:729:

RuntimeWarning: divide by zero encountered in log

```
logp = np.log(p)
```

Validation accuracy has not improved for 10 epochs. Stopping early after 5390 iterations.

Train acc: 0.8620, Val acc: 0.5540

Training with lr=0.001, hidden\_size=200, n\_hiddens=3, reg=0.01

Validation accuracy has not improved for 10 epochs. Stopping early after 5390 iterations.

Train acc: 0.8120, Val acc: 0.5180

Training with lr=0.001, hidden\_size=200, n\_hiddens=3, reg=0.01

Validation accuracy has not improved for 10 epochs. Stopping early after 5635 iterations.

Train acc: 0.8100, Val acc: 0.5430

=====

Best validation accuracy: 0.5540

Best hyperparameters: lr=0.001, hidden\_size=200, reg=0.01

## 4 Test Your Model!

Run your best model on the validation and test sets. You should achieve at least 50% accuracy on the validation set and the test set.

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

Validation set accuracy: 0.556

Test set accuracy: 0.548

```
[46]: best_model.save('best_fully_connected_net.npy')
```

```
best_fully_connected_net.npy saved.
```

```
[ ]:
```