```
# 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. 'cs231n/assignments/assignment1/'
FOLDERNAME = '682/assignment2/cs682'
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/$FOLDERNAME/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/682/assignment2/cs682/datasets
--2021-10-26 16:55:10--  https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  42.4MB/s    in 4.3s

2021-10-26 16:55:14 (38.1 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/682/assignment2/cs682
```

Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more 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.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer and Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

```
# !python setup.py build_ext --inplace
%cd /content/drive/My Drive/682/assignment2
```

```
/content/drive/My Drive/682/assignment2
```

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs682.classifiers.fc_net import *
from cs682.data_utils import get_CIFAR10_data
from cs682.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs682.solver import Solver

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

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

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

rel_error(5, 5.2)
```

```
0.01960784313725492
```

```
# 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,))
```

Affine layer: foward

Open the file `cs682/layers.py` and implement the `affine_forward` function.

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

```
# 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)

## gives x shape of actual matrix X.shape = (2, 4,5,6)
## X shape: (N, d_1, ..., d_k) and contains a minibatch of N
## examples, where each example x[i] has shape (d_1, ..., d_k).
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(out)
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
[[1.49834967 1.70660132 1.91485297]
 [3.25553199 3.5141327  3.77273342]]
Testing affine_forward function:
difference:  9.769849468192957e-10
```

Affine layer: backward

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

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

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout)
```

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

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

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

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

ReLU activation: forward

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

```
# 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,       ]])
print(type(out))
print(type(correct_out))
# 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))
```

```
     <class 'numpy.ndarray'>
     <class 'numpy.ndarray'>
     Testing relu_forward function:
     difference:  4.999999798022158e-08
```

ReLU activation: backward

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

```
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
```

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

Answer:

If the backpropagated gradient keeps becoming smaller, there will be close to nothing updated in the inputs when the next forward pass happens. The gradient depends on the input to the layer from the activation function, if this input is less, the gradient will be less. So,

1. The sigmoid makes inputs close to the tail extremely small. For eg, if x=[-1000], sig(x) ~ 0.
2. For inputs < 0 , ReLU makes them zero. Hence, vanishing gradient for ReLU appears in cases like x = [-10, -3, 0, -2]. and these neurons can never be updated.
3. Leaky ReLU is a tweak of ReLU where instead of 0 the function assigns some considerable value to inputs. But then here in order to witness vanishing gradient are input should be all zeros.

"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 `cs682/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the

```
from cs682.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

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

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

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

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

Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs682/layers.py`.

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

```
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: svm_loss(x, y)[0], x, verbose=False)
```

```
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be around the order of
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

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 svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

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

Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs682/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

```
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,  16.092
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  16.188
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  16.284
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

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

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

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

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
```

```
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

Open the file `cs682/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 at least `50%` accuracy on the validation set.

```
model = TwoLayerNet()
best_solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################
best_val = -1
lrs = [1/10**i for i in np.arange(5)]
regs = np.unique([i/(10)**i for i in np.random.randint(2, 7, 5)])
hiddens = np.unique(np.random.randint(20, 200, 10))

for lr in lrs:
  for reg in regs:
    for hidden in hiddens:
        model = TwoLayerNet(hidden_dim = hidden, reg= reg)
        solver = Solver(model, data, optim_config={'learning_rate':lr}, update_rule='sgd',
                            lr_decay=0.95, num_epochs=10,batch_size=200, print_every=-1, verb
        solver.train()

        val_accuracy = solver.best_val_acc

        if best_val < val_accuracy:
            best_val = val_accuracy
            best_solver = solver

        print(f'lr: {lr} reg: {reg} hidden: {hidden}  val accuracy: {val_accuracy}')

print(f'best validation accuracy achieved: {best_val}')
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

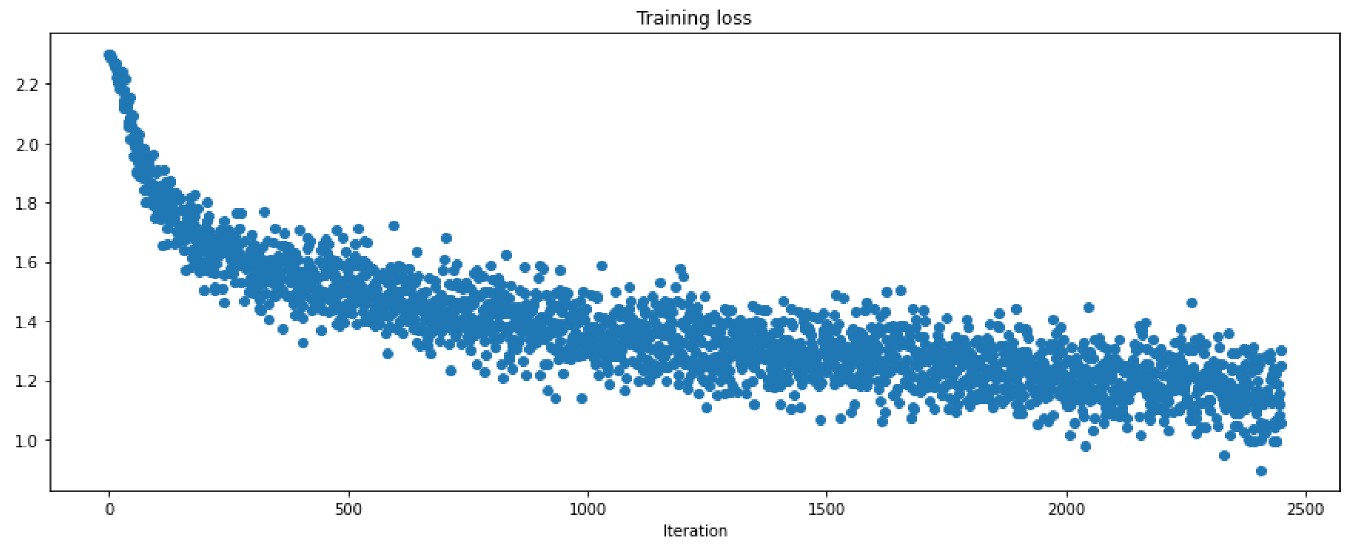
```
lr: 1.0 reg: 6e-06 hidden: 26  val accuracy: 0.087
lr: 1.0 reg: 6e-06 hidden: 42  val accuracy: 0.09
lr: 1.0 reg: 6e-06 hidden: 70  val accuracy: 0.129
```

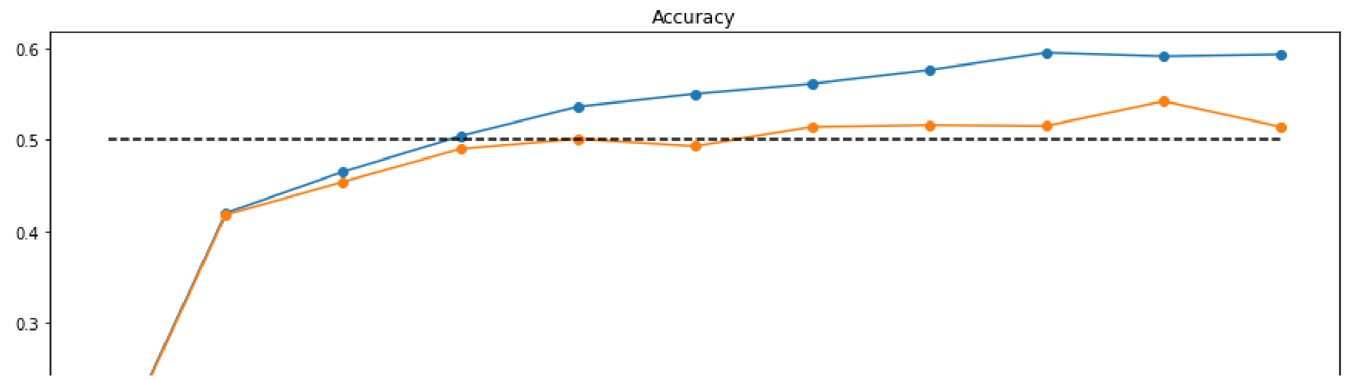
```
lr: 1.0 reg: 6e-06 hidden: 74  val accuracy: 0.113
lr: 1.0 reg: 6e-06 hidden: 80  val accuracy: 0.104
lr: 1.0 reg: 6e-06 hidden: 82  val accuracy: 0.087
lr: 1.0 reg: 6e-06 hidden: 127  val accuracy: 0.118
lr: 1.0 reg: 6e-06 hidden: 136  val accuracy: 0.098
lr: 1.0 reg: 6e-06 hidden: 145  val accuracy: 0.105
lr: 1.0 reg: 6e-06 hidden: 175  val accuracy: 0.119
lr: 1.0 reg: 0.003 hidden: 26  val accuracy: 0.12
lr: 1.0 reg: 0.003 hidden: 42  val accuracy: 0.136
lr: 1.0 reg: 0.003 hidden: 70  val accuracy: 0.087
lr: 1.0 reg: 0.003 hidden: 74  val accuracy: 0.1
lr: 1.0 reg: 0.003 hidden: 80  val accuracy: 0.096
lr: 1.0 reg: 0.003 hidden: 82  val accuracy: 0.147
lr: 1.0 reg: 0.003 hidden: 127  val accuracy: 0.119
lr: 1.0 reg: 0.003 hidden: 136  val accuracy: 0.114
lr: 1.0 reg: 0.003 hidden: 145  val accuracy: 0.087
lr: 1.0 reg: 0.003 hidden: 175  val accuracy: 0.094
lr: 1.0 reg: 0.02 hidden: 26  val accuracy: 0.113
lr: 1.0 reg: 0.02 hidden: 42  val accuracy: 0.176
lr: 1.0 reg: 0.02 hidden: 70  val accuracy: 0.087
lr: 1.0 reg: 0.02 hidden: 74  val accuracy: 0.112
lr: 1.0 reg: 0.02 hidden: 80  val accuracy: 0.113
lr: 1.0 reg: 0.02 hidden: 82  val accuracy: 0.122
lr: 1.0 reg: 0.02 hidden: 127  val accuracy: 0.102
lr: 1.0 reg: 0.02 hidden: 136  val accuracy: 0.111
lr: 1.0 reg: 0.02 hidden: 145  val accuracy: 0.107
lr: 1.0 reg: 0.02 hidden: 175  val accuracy: 0.088
lr: 0.1 reg: 6e-06 hidden: 26  val accuracy: 0.134
lr: 0.1 reg: 6e-06 hidden: 42  val accuracy: 0.087
lr: 0.1 reg: 6e-06 hidden: 70  val accuracy: 0.138
lr: 0.1 reg: 6e-06 hidden: 74  val accuracy: 0.129
lr: 0.1 reg: 6e-06 hidden: 80  val accuracy: 0.094
lr: 0.1 reg: 6e-06 hidden: 82  val accuracy: 0.125
lr: 0.1 reg: 6e-06 hidden: 127  val accuracy: 0.095
lr: 0.1 reg: 6e-06 hidden: 136  val accuracy: 0.138
lr: 0.1 reg: 6e-06 hidden: 145  val accuracy: 0.149
lr: 0.1 reg: 6e-06 hidden: 175  val accuracy: 0.103
lr: 0.1 reg: 0.003 hidden: 26  val accuracy: 0.13
lr: 0.1 reg: 0.003 hidden: 42  val accuracy: 0.11
lr: 0.1 reg: 0.003 hidden: 70  val accuracy: 0.098
lr: 0.1 reg: 0.003 hidden: 74  val accuracy: 0.107
lr: 0.1 reg: 0.003 hidden: 80  val accuracy: 0.146
lr: 0.1 reg: 0.003 hidden: 82  val accuracy: 0.105
lr: 0.1 reg: 0.003 hidden: 127  val accuracy: 0.12
lr: 0.1 reg: 0.003 hidden: 136  val accuracy: 0.171
lr: 0.1 reg: 0.003 hidden: 145  val accuracy: 0.105
lr: 0.1 reg: 0.003 hidden: 175  val accuracy: 0.145
lr: 0.1 reg: 0.02 hidden: 26  val accuracy: 0.087
lr: 0.1 reg: 0.02 hidden: 42  val accuracy: 0.154
lr: 0.1 reg: 0.02 hidden: 70  val accuracy: 0.106
lr: 0.1 reg: 0.02 hidden: 74  val accuracy: 0.137
lr: 0.1 reg: 0.02 hidden: 80  val accuracy: 0.161
lr: 0.1 reg: 0.02 hidden: 82  val accuracy: 0.136
lr: 0.1 reg: 0.02 hidden: 127  val accuracy: 0.098
lr: 0.1 reg: 0.02 hidden: 136  val accuracy: 0.098
lr: 0.1 reg: 0.02 hidden: 145  val accuracy: 0.149
```

```
# Run this cell to visualize training loss and train / val accuracy

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

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(best_solver.train_acc_history, '-o', label='train')
plt.plot(best_solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(best_solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```





Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs682/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization; we will add those features soon.

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. Do the initial losses seem reasonable?

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

```
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('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 6.86e-09
W2 relative error: 3.52e-08
W3 relative error: 1.32e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

As another sanity check, make sure you can overfit 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 initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

```
# 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 = 2e-2
learning_rate = 2e-3
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, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 3.604568
(Epoch 0 / 20) train acc: 0.140000; val_acc: 0.116000
(Epoch 1 / 20) train acc: 0.300000; val_acc: 0.137000
(Epoch 2 / 20) train acc: 0.540000; val_acc: 0.150000
(Epoch 3 / 20) train acc: 0.640000; val_acc: 0.162000
(Epoch 4 / 20) train acc: 0.760000; val_acc: 0.168000
(Epoch 5 / 20) train acc: 0.840000; val_acc: 0.175000
(Iteration 11 / 40) loss: 0.622784
(Epoch 6 / 20) train acc: 0.880000; val_acc: 0.188000
(Epoch 7 / 20) train acc: 0.900000; val_acc: 0.184000
(Epoch 8 / 20) train acc: 0.920000; val_acc: 0.190000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.187000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.179000
(Iteration 21 / 40) loss: 0.404246
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.204000
```

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

```
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.203000
```

```
# 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'],
}

np.random.seed(1)
learning_rate = [1e-2, 1e-3, 2e-2, 2e-3]
weight_scales = [1e-4, 2e-3, 1e-2, 5e-2, 1e-1, 4e-3]
best_tr_acc = -1
best_solver = None
for lr in learning_rate:
  for ws in weight_scales:
      model = FullyConnectedNet([100,100,100,100],
                      weight_scale=ws, dtype=np.float64)
      solver = Solver(model, small_data,
                      print_every=-1, num_epochs=20, batch_size=25,
                      update_rule='sgd',
                      optim_config={
                        'learning_rate': lr,
                      }, verbose = False
              )
      solver.train()
      train_accuracy = solver.train_acc_history[len(solver.train_acc_history) - 1]
      if (train_accuracy > best_tr_acc):
        best_tr_acc = train_accuracy
        best_solver = solver
        print(f"learning_rate={learning_rate}, weight_scale={ws}, train_accuracy={train_accurac

        for key, value in sorted(model.params.items()):
            if key[0] == 'W':
                print(f"{key} : {value.mean()}")

best_solver.train()

plt.plot(best_solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
learning_rate=[0.01, 0.001, 0.02, 0.002], weight_scale=0.0001, train_accuracy=0.16
W1 : 2.3379918178315844e-07
W2 : -6.705410105077629e-07
W3 : 1.5553581603587551e-06
W4 : 1.8505657795072092e-06
W5 : -3.7048341144137637e-06
learning_rate=[0.01, 0.001, 0.02, 0.002], weight_scale=0.002, train_accuracy=0.16
W1 : -2.8462080893222337e-06
W2 : -1.6908855194341173e-05
W3 : -1.7889561799614805e-05
W4 : 2.840172751611848e-05
W5 : 0.000122150005797344
learning_rate=[0.01, 0.001, 0.02, 0.002], weight_scale=0.01, train_accuracy=0.16
W1 : -4.992114784818131e-06
W2 : 1.1821244610160595e-05
W3 : 8.878385014081328e-06
W4 : 7.876492945960907e-05
W5 : 0.0001426579830202422
```

Inline Question 2:

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

```
W2 : -7.134295643847695e+40
```

Answer:

Five layer net takes shorter time to minimum loss due to increase learning power of more layers. Also, five layer network is more sensitive to the initialization scale. Mostly due to the fact that the weights are subjected to updation for a long time in deeper network and if they were initialized small, they tend to create small gradients which would mean no learning of these weights. If the weights are initialized large, then gradients will be enormous leading to weights jumping way beyond the optimal again leading to less accuracy.

```
W4 : 5.622979248483589e-07
```

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.

```
W1 : 4.1706879818067464e-06
```

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. See the Momentum Update section at https://compsci682-fa19.github.io/notes/neural-networks-3/#sgd for more information.

Open the file `cs682/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.

```
from cs682.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']))
```

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.

```
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': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

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 net.

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 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 nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

```
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.                                                     #
################################################################################
learning_rate = [1e-2, 1e-3, 2e-2, 2e-3]
weight_scales = [1e-4, 2e-3, 1e-2, 5e-2, 1e-1, 4e-3]
best_val = -1

for lr in learning_rate:
  for ws in weight_scales:
      model = FullyConnectedNet([100,100,100,100],
                      weight_scale=ws, dtype=np.float64)
      solver = Solver(model, small_data,
                      print_every=-1, num_epochs=20, batch_size=25,
                      update_rule='sgd',
                      optim_config={
                        'learning_rate': lr,
                      }, verbose = False
              )
      solver.train()

      val_accuracy = solver.best_val_acc

      if best_val < val_accuracy:
          best_val = val_accuracy
          best_model = model

      print(f'lr: {lr} reg: {reg} hidden: {hidden}  val accuracy: {val_accuracy}')

print(f'best validation accuracy achieved: {best_val}')


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

```
lr: 0.01 reg: 3.14 hidden: 175  val accuracy: 0.105
lr: 0.01 reg: 3.14 hidden: 175  val accuracy: 0.095
lr: 0.01 reg: 3.14 hidden: 175  val accuracy: 0.372
lr: 0.01 reg: 3.14 hidden: 175  val accuracy: 0.366
lr: 0.01 reg: 3.14 hidden: 175  val accuracy: 0.098
lr: 0.01 reg: 3.14 hidden: 175  val accuracy: 0.113
```

```
lr: 0.001 reg: 3.14 hidden: 175  val accuracy: 0.105
lr: 0.001 reg: 3.14 hidden: 175  val accuracy: 0.113
lr: 0.001 reg: 3.14 hidden: 175  val accuracy: 0.119
lr: 0.001 reg: 3.14 hidden: 175  val accuracy: 0.363
lr: 0.001 reg: 3.14 hidden: 175  val accuracy: 0.256
lr: 0.001 reg: 3.14 hidden: 175  val accuracy: 0.113
lr: 0.02 reg: 3.14 hidden: 175  val accuracy: 0.113
lr: 0.02 reg: 3.14 hidden: 175  val accuracy: 0.113
lr: 0.02 reg: 3.14 hidden: 175  val accuracy: 0.4
lr: 0.02 reg: 3.14 hidden: 175  val accuracy: 0.379
lr: 0.02 reg: 3.14 hidden: 175  val accuracy: 0.105
lr: 0.02 reg: 3.14 hidden: 175  val accuracy: 0.113
lr: 0.002 reg: 3.14 hidden: 175  val accuracy: 0.105
lr: 0.002 reg: 3.14 hidden: 175  val accuracy: 0.113
lr: 0.002 reg: 3.14 hidden: 175  val accuracy: 0.156
lr: 0.002 reg: 3.14 hidden: 175  val accuracy: 0.366
lr: 0.002 reg: 3.14 hidden: 175  val accuracy: 0.276
lr: 0.002 reg: 3.14 hidden: 175  val accuracy: 0.105
best validation accuracy achieved: 0.4
```

Test your model!

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

```
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:  0.4
Test set accuracy:  0.38
```