In [ ]:
```python
# 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 = 'cs231n/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/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment1
```

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:

```python
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:

```python
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 [ ]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradie
from cs231n.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-ipytho
%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))))
```

In [ ]:
```
# 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: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

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

In [ ]:
```
# 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
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
```

Affine layer: backward

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

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

_, 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:

In [ ]:
```
# 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
```

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:

In [ ]:
```
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:

1. Sigmoid: gradients vanishing happens when the input is far from 0;
2. ReLU: gradients vanishing happens when the input is less than 0.

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

In [ ]:
```
from cs231n.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]
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0]
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0]

# 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

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

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

In [ ]:
```
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
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=Fal
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be
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:  9.483503037636722e-09
```

Two-layer network

Open the file `cs231n/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.

In [ ]:

```
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, 15.3320
```

```
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.4999
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.6678
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: 3.31e-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

Open the file `cs231n/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.

In [ ]:

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

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                            #
##############################################################################
```

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

data = get_CIFAR10_data()
solver = Solver(model, data,
        update_rule='sgd',
        optim_config={
          'learning_rate': 1e-4,
        },
        lr_decay=0.95,
        num_epochs=5, batch_size=200,
        print_every=100)
solver.train()

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

```
(Iteration 1 / 1225) loss: 2.301725
(Epoch 0 / 5) train acc: 0.145000; val_acc: 0.140000
(Iteration 101 / 1225) loss: 2.241923
(Iteration 201 / 1225) loss: 2.187425
(Epoch 1 / 5) train acc: 0.267000; val_acc: 0.243000
(Iteration 301 / 1225) loss: 2.056790
(Iteration 401 / 1225) loss: 1.937978
(Epoch 2 / 5) train acc: 0.294000; val_acc: 0.303000
(Iteration 501 / 1225) loss: 1.924555
(Iteration 601 / 1225) loss: 1.933743
(Iteration 701 / 1225) loss: 1.832777
(Epoch 3 / 5) train acc: 0.336000; val_acc: 0.315000
(Iteration 801 / 1225) loss: 1.960827
(Iteration 901 / 1225) loss: 1.832752
(Epoch 4 / 5) train acc: 0.340000; val_acc: 0.350000
(Iteration 1001 / 1225) loss: 1.739182
(Iteration 1101 / 1225) loss: 1.940517
(Iteration 1201 / 1225) loss: 1.848443
(Epoch 5 / 5) train acc: 0.355000; val_acc: 0.373000
```

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.

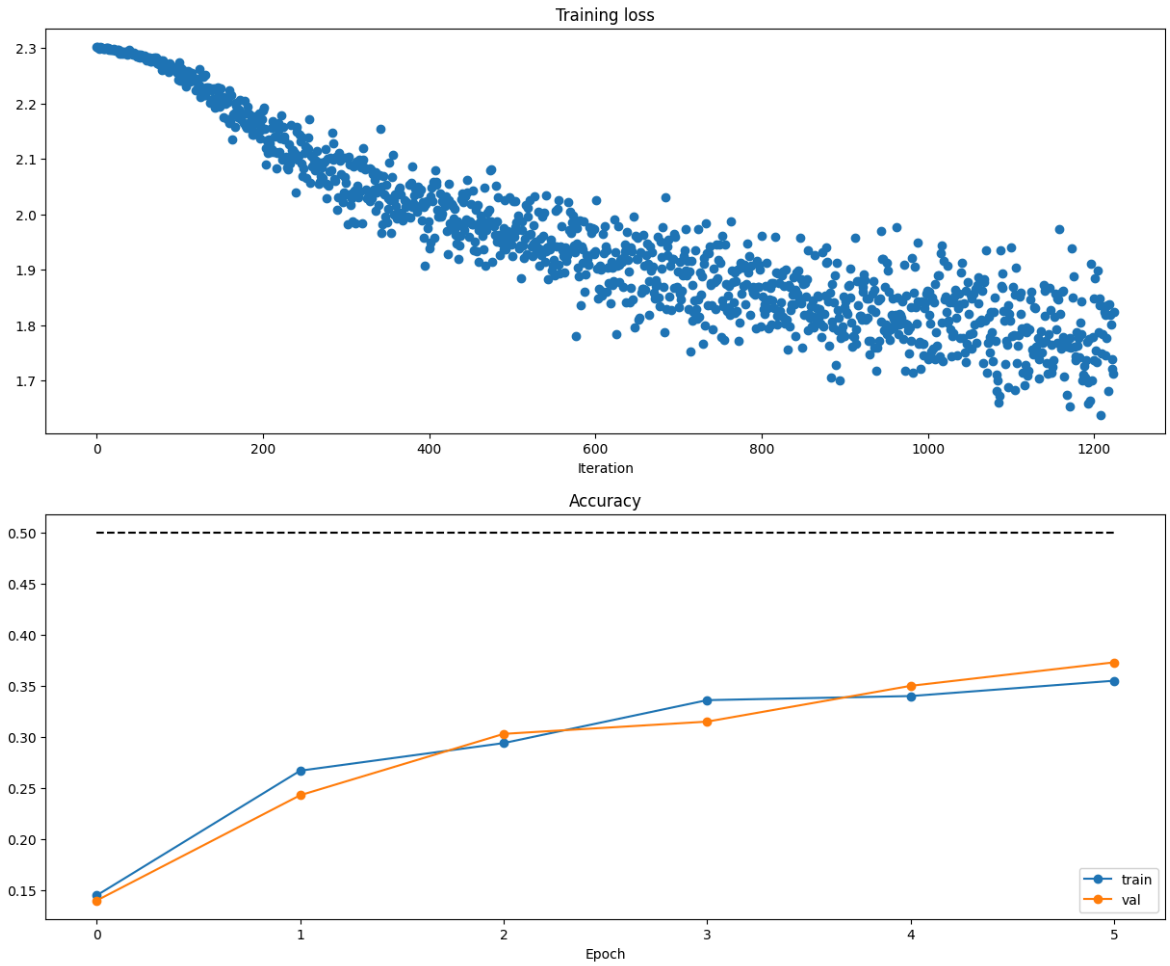
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [ ]:

```python
# 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()
```

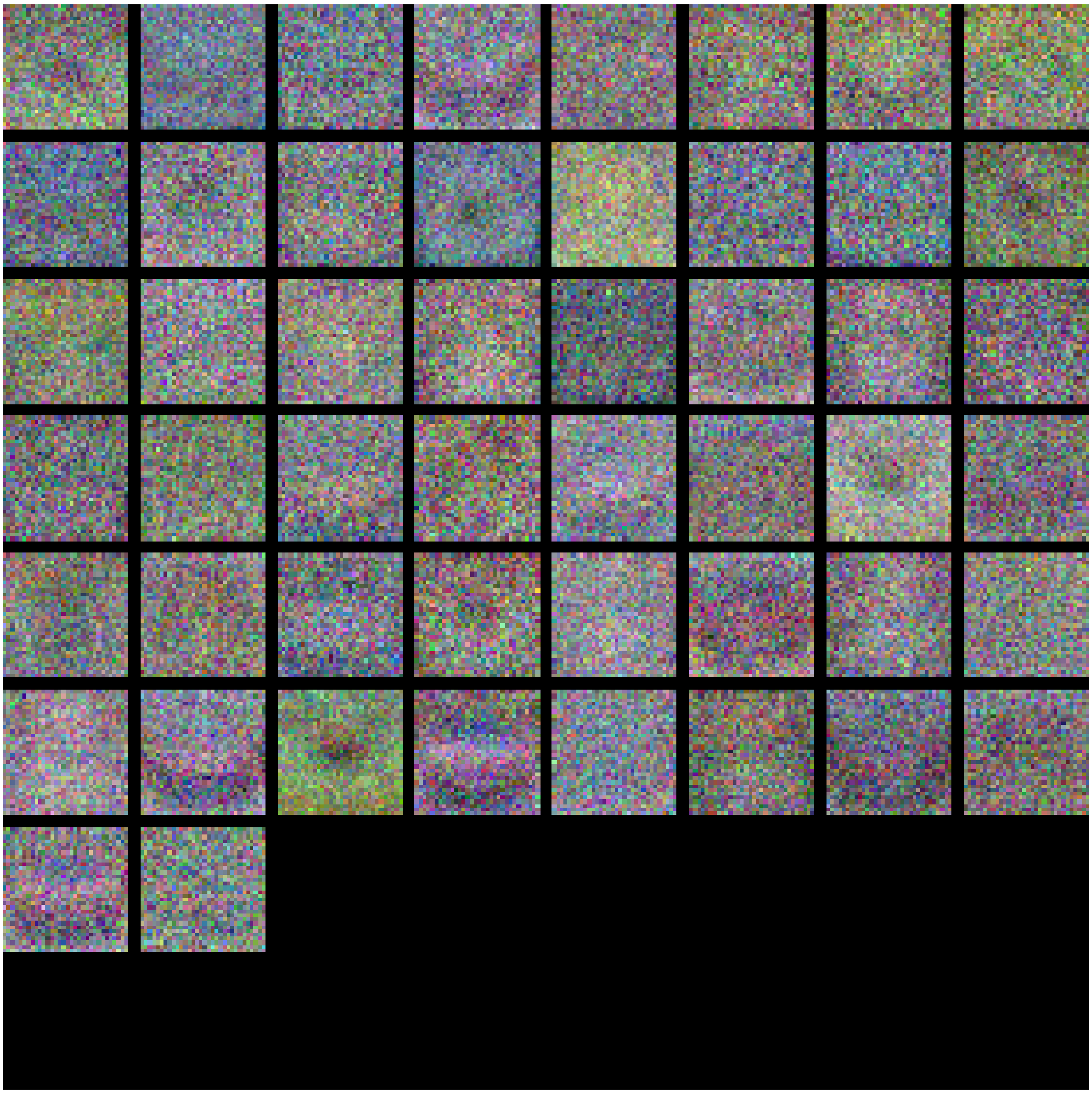


In [ ]:
```
from cs231n.vis_utils import visualize_grid

# 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)
```



Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [ ]:
```
best_model = None
best_val_acc = 0.0
results = {}

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained
# model in best_model.
#
# To help debug your network, it may help to use visualizations similar to the
# ones we used above; these visualizations will have significant qualitative
# differences from the ones we saw above for the poorly tuned network.
#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to
# write code to sweep through possible combinations of hyperparameters
# automatically like we did on thexs previous exercises.
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for reg_strength in [0.5, 0.7, 1.0]:
  for hidden_size in [400]:
    for lr in [5e-4, 1e-3]:
      model = TwoLayerNet(hidden_dim=hidden_size, reg=reg_strength)
      solver = Solver(model, data,
                      update_rule='sgd',
                      optim_config={
                        'learning_rate': lr,
                      },
                      lr_decay=0.95,
                      num_epochs=10, batch_size=200,
                      print_every=1000000)
      solver.train()
      train_acc = solver.train_acc_history[-1]
      val_acc = solver.val_acc_history[-1]
      results[(hidden_size, reg, lr)] = (train_acc, val_acc)
      print(f"Params: hidden_size={hidden_size}, reg={reg_strength}, lr={lr} | T

      if val_acc > best_val_acc:
          best_val_acc = val_acc
          best_model = model

print("Best validation accuracy achieved during tuning:", best_val_acc)
# The result: Params: hidden_size=400, reg=0.5, lr=0.001 | Train acc: 0.5960, Va

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

```
(Iteration 1 / 2450) loss: 2.617604
(Epoch 0 / 10) train acc: 0.101000; val_acc: 0.136000
(Epoch 1 / 10) train acc: 0.429000; val_acc: 0.413000
(Epoch 2 / 10) train acc: 0.458000; val_acc: 0.456000
(Epoch 3 / 10) train acc: 0.482000; val_acc: 0.464000
(Epoch 4 / 10) train acc: 0.508000; val_acc: 0.480000
(Epoch 5 / 10) train acc: 0.514000; val_acc: 0.489000
(Epoch 6 / 10) train acc: 0.523000; val_acc: 0.501000
(Epoch 7 / 10) train acc: 0.553000; val_acc: 0.509000
(Epoch 8 / 10) train acc: 0.535000; val_acc: 0.519000
(Epoch 9 / 10) train acc: 0.514000; val_acc: 0.528000
(Epoch 10 / 10) train acc: 0.566000; val_acc: 0.512000
Params: hidden_size=400, reg=0.5, lr=0.0005 | Train acc: 0.5660, Val acc: 0.5120
(Iteration 1 / 2450) loss: 2.617139
(Epoch 0 / 10) train acc: 0.183000; val_acc: 0.171000
(Epoch 1 / 10) train acc: 0.422000; val_acc: 0.432000
(Epoch 2 / 10) train acc: 0.463000; val_acc: 0.477000
(Epoch 3 / 10) train acc: 0.516000; val_acc: 0.502000
(Epoch 4 / 10) train acc: 0.556000; val_acc: 0.499000
(Epoch 5 / 10) train acc: 0.543000; val_acc: 0.505000
(Epoch 6 / 10) train acc: 0.583000; val_acc: 0.522000
(Epoch 7 / 10) train acc: 0.564000; val_acc: 0.522000
(Epoch 8 / 10) train acc: 0.568000; val_acc: 0.514000
(Epoch 9 / 10) train acc: 0.623000; val_acc: 0.519000
(Epoch 10 / 10) train acc: 0.596000; val_acc: 0.535000
Params: hidden_size=400, reg=0.5, lr=0.001 | Train acc: 0.5960, Val acc: 0.5350
(Iteration 1 / 2450) loss: 2.737459
(Epoch 0 / 10) train acc: 0.139000; val_acc: 0.122000
(Epoch 1 / 10) train acc: 0.412000; val_acc: 0.403000
(Epoch 2 / 10) train acc: 0.458000; val_acc: 0.448000
(Epoch 3 / 10) train acc: 0.480000; val_acc: 0.461000
(Epoch 4 / 10) train acc: 0.503000; val_acc: 0.475000
(Epoch 5 / 10) train acc: 0.479000; val_acc: 0.484000
(Epoch 6 / 10) train acc: 0.543000; val_acc: 0.499000
(Epoch 7 / 10) train acc: 0.532000; val_acc: 0.500000
(Epoch 8 / 10) train acc: 0.510000; val_acc: 0.505000
(Epoch 9 / 10) train acc: 0.556000; val_acc: 0.513000
(Epoch 10 / 10) train acc: 0.552000; val_acc: 0.528000
Params: hidden_size=400, reg=0.7, lr=0.0005 | Train acc: 0.5520, Val acc: 0.5280
(Iteration 1 / 2450) loss: 2.740210
(Epoch 0 / 10) train acc: 0.150000; val_acc: 0.183000
(Epoch 1 / 10) train acc: 0.414000; val_acc: 0.448000
(Epoch 2 / 10) train acc: 0.494000; val_acc: 0.461000
(Epoch 3 / 10) train acc: 0.495000; val_acc: 0.501000
(Epoch 4 / 10) train acc: 0.526000; val_acc: 0.497000
(Epoch 5 / 10) train acc: 0.560000; val_acc: 0.509000
(Epoch 6 / 10) train acc: 0.564000; val_acc: 0.499000
(Epoch 7 / 10) train acc: 0.550000; val_acc: 0.497000
(Epoch 8 / 10) train acc: 0.564000; val_acc: 0.511000
(Epoch 9 / 10) train acc: 0.609000; val_acc: 0.526000
(Epoch 10 / 10) train acc: 0.607000; val_acc: 0.523000
Params: hidden_size=400, reg=0.7, lr=0.001 | Train acc: 0.6070, Val acc: 0.5230
(Iteration 1 / 2450) loss: 2.921649
(Epoch 0 / 10) train acc: 0.156000; val_acc: 0.146000
(Epoch 1 / 10) train acc: 0.416000; val_acc: 0.402000
(Epoch 2 / 10) train acc: 0.433000; val_acc: 0.442000
(Epoch 3 / 10) train acc: 0.497000; val_acc: 0.472000
(Epoch 4 / 10) train acc: 0.512000; val_acc: 0.476000
(Epoch 5 / 10) train acc: 0.508000; val_acc: 0.477000
(Epoch 6 / 10) train acc: 0.532000; val_acc: 0.486000
```

```
(Epoch 7 / 10) train acc: 0.483000; val_acc: 0.498000
(Epoch 8 / 10) train acc: 0.513000; val_acc: 0.501000
(Epoch 9 / 10) train acc: 0.542000; val_acc: 0.505000
(Epoch 10 / 10) train acc: 0.566000; val_acc: 0.501000
Params: hidden_size=400, reg=1.0, lr=0.0005 | Train acc: 0.5660, Val acc: 0.5010
(Iteration 1 / 2450) loss: 2.916187
(Epoch 0 / 10) train acc: 0.188000; val_acc: 0.197000
(Epoch 1 / 10) train acc: 0.426000; val_acc: 0.443000
(Epoch 2 / 10) train acc: 0.473000; val_acc: 0.466000
(Epoch 3 / 10) train acc: 0.498000; val_acc: 0.472000
(Epoch 4 / 10) train acc: 0.504000; val_acc: 0.495000
(Epoch 5 / 10) train acc: 0.548000; val_acc: 0.501000
(Epoch 6 / 10) train acc: 0.528000; val_acc: 0.485000
(Epoch 7 / 10) train acc: 0.530000; val_acc: 0.515000
(Epoch 8 / 10) train acc: 0.535000; val_acc: 0.525000
(Epoch 9 / 10) train acc: 0.580000; val_acc: 0.503000
(Epoch 10 / 10) train acc: 0.558000; val_acc: 0.513000
Params: hidden_size=400, reg=1.0, lr=0.001 | Train acc: 0.5580, Val acc: 0.5130
Best validation accuracy achieved during tuning: 0.535
```

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.

In [ ]:
```
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:  0.535
```

In [ ]:
```
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:  0.53
```

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.

$\color{blue}{\textit{Your Answer :}}$ 1,3

$\color{blue}{\textit{Your Explanation :}}$ For 1, a larger dataset helps the model generalize better and reduces overfitting, which narrows the gap between training and testing accuracy.

For 2, it depends.

For 3, increasing the regularization strength can reduces overfitting, which decreases the gap.

```
In [ ]:
```