Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012 (https://arxiv.org/abs/1207.0580)

In [1]:
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from data_utils import get_CIFAR10_data
from implementations.layers import dropout_forward


%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-i
%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 [2]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in 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,)
```

Dropout in Tensorflow

Let's first try the dropout function in Tensorflow.

In [3]:
```
np.random.seed(15009)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:

  # please read the documentation of tf.nn.dropout carefully
  out = tf.Session().run(tf.nn.dropout(x, keep_prob=p))

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print()
```

```
WARNING:tensorflow:From <ipython-input-3-bdbd85df5008>:7: calling dropout
(from tensorflow.python.ops.nn_ops) with keep_prob is deprecated and will
be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1
- keep_prob`.
Running tests with p =  0.25
Mean of input:  9.998768973493084
Mean of train-time output:  9.9957137660448
Fraction of train-time output set to zero:  0.750112

Running tests with p =  0.4
Mean of input:  9.998768973493084
Mean of train-time output:  10.015868072002652
Fraction of train-time output set to zero:  0.599244

Running tests with p =  0.7
Mean of input:  9.998768973493084
Mean of train-time output:  9.98812569165609
Fraction of train-time output set to zero:  0.30056
```

Dropout forward pass

In the file `implementations/layers.py` , implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

In [4]:
```
np.random.seed(15009)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
  out = dropout_forward(x, {'mode': 'train', 'p': p})

  # Hint: The tensorflow dropout does not have a mode parameter to specify
  out_test = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean(
  print()
```

```
Running tests with p =  0.25
Mean of input:  9.998768973493084
Mean of train-time output:  10.076325491035233
Mean of test-time output:  9.998768973493084
Fraction of train-time output set to zero:  0.748048
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  9.998768973493084
Mean of train-time output:  9.986985273167813
Mean of test-time output:  9.998768973493084
Fraction of train-time output set to zero:  0.600492
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  9.998768973493084
Mean of train-time output:  10.025574363127927
Mean of test-time output:  9.998768973493084
Fraction of train-time output set to zero:  0.29814
Fraction of test-time output set to zero:  0.0
```

Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

Answer:

This happens to keep the "range" of the output neuron within normal levels of no dropout. If the values are not divided by p this could cause the neuron to saturate.

Fully-connected nets with Dropout

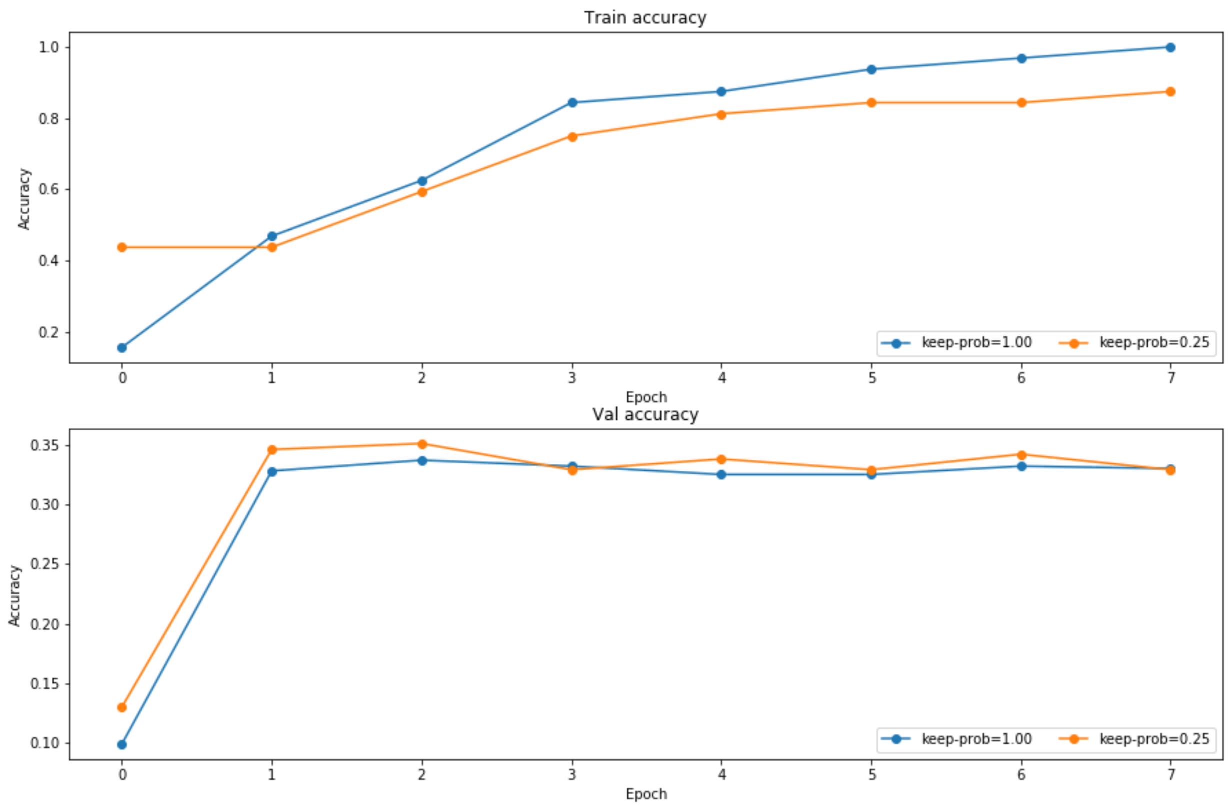
The assignment provides an implementation of fully connected neural network. You need to add dropout layters to the implementation. *Now you can use* `tf.nn.dropout` .

Regularization experiment: As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

In [7]:
```
from implementations.fc_net import FullyConnectedNet

# Train two identical nets, one with dropout and one without
np.random.seed(15009)
num_train = 500

X_train = data['X_train'][:num_train]
X_train = np.reshape(X_train, [X_train.shape[0], -1]) / 255
y_train = data['y_train'][:num_train]
X_val = data['X_val']
X_val = np.reshape(X_val, [X_val.shape[0], -1]) / 255
y_val = data['y_val']

traces={}
dropout_choices = [1, 0.25]
for keep_prob in dropout_choices:
  model = FullyConnectedNet(input_size=X_train[0].size, hidden_size=[500],



  train_trace= model.train(X=X_train, y=y_train, X_val=X_val, y_val=y_val,
                           learning_rate=2e-3, keep_prob=keep_prob,
                           reg=np.float32(5e-6), num_iters=1000,
                           batch_size=32, verbose=True)

  traces[keep_prob] = train_trace
```

```
iteration 0 / 1000: objective 74.464775
iteration 100 / 1000: objective 43.078403
iteration 200 / 1000: objective 32.892120
iteration 300 / 1000: objective 17.114698
iteration 400 / 1000: objective 14.227290
iteration 500 / 1000: objective 21.940580
iteration 600 / 1000: objective 3.830667
iteration 700 / 1000: objective 3.937410
iteration 800 / 1000: objective 2.006174
iteration 900 / 1000: objective 2.123654
iteration 0 / 1000: objective 76.679871
iteration 100 / 1000: objective 44.354172
iteration 200 / 1000: objective 42.335323
iteration 300 / 1000: objective 27.721495
iteration 400 / 1000: objective 28.461147
iteration 500 / 1000: objective 44.789948
iteration 600 / 1000: objective 15.377269
iteration 700 / 1000: objective 11.519280
iteration 800 / 1000: objective 6.838014
iteration 900 / 1000: objective 7.559201
```

In [8]:
```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for keep_prob in dropout_choices:
  trace = traces[keep_prob]
  train_accs.append(trace['train_acc_history'][-1])
  val_accs.append(trace['val_acc_history'][-1])

plt.subplot(3, 1, 1)
for keep_prob in dropout_choices:
  plt.plot(traces[keep_prob]['train_acc_history'], '-o', label='keep-prob=%
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for keep_prob in dropout_choices:
  plt.plot(traces[keep_prob]['val_acc_history'], '-o', label='keep-prob=%.2
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

Answer:

The train accuracy is higher without dropout, but the validation accuracy using dropout is higher. This suggests using dropout as a regularizer helps prevent overfitting.

Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). How should we modify p, if at all, if we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

Answer:

If we decrease the size of hidden layers we want to increase the keep probability so we are still using a sufficient amount of neurons. This is also because decreasing the size of hidden layers can lead towards underfitting so we want to increase the keep probability to counter act this.

Batch Normalization

In this task, we implement batch normalization, which normalizes hidden layers and makes the training procedure more stable.

Reference: Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015. (https://arxiv.org/abs/1502.03167)

In [2]:
```
# As usual, a bit of setup
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from data_utils import get_CIFAR10_data
from implementations.layers import batchnorm_forward

%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-i
%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)))

def print_mean_std(x,axis=0):
    print('  means: ', x.mean(axis=axis))
    print('  stds:  ', x.std(axis=axis))
    print()
```

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

In [3]:
```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k, v in 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,)
```

Batch normalization: forward

In the file `implementations/layers.py` , implement the batch normalization forward pass in the function `batchnorm_forward` . Once you have done so, run the following to test your implementation.

Referencing the paper linked to above would be helpful!

In [4]:
```
# A very simple example

xtrain = np.array([[10], [20], [30]])

xtest = np.array([[25]])


# initialize parameters for batch normalization
bn_param = {}
bn_param['mode'] = 'train'
bn_param['eps'] = 1e-4
bn_param['momentum'] = 0.95

# initialize the running mean as zero and the running variance as one
bn_param['running_mean'] = np.zeros([1, xtrain.shape[1]])
bn_param['running_var'] = np.ones([1, xtrain.shape[1]])

# gamma and beta do not make changes to the standardization result from the
gamma = np.ones([1])
beta = np.zeros([1])

print('Before batch normalization, xtrain has ')
print_mean_std(xtrain,axis=0)

xnorm = batchnorm_forward(xtrain, gamma, beta, bn_param)

print('After batch normalization, xtrain has ')
print_mean_std(xnorm,axis=0) # The mean and std should be 0 and 1 respectiv


print('After batch normalization, the running mean and the running variance
print(bn_param['running_mean']) # should be 1.0
print(bn_param['running_var']) # should be 4.283


for iter in range(1000):
    xnorm = batchnorm_forward(xtrain, gamma, beta, bn_param)

print('After many iterations, the running mean and the running variance are
print(bn_param['running_mean']) # should be 20, the mean of xtrain
print(bn_param['running_var']) # should be 66.667, the variance of xtrain


# enter test mode,
bn_param['mode'] = 'test'
xtest_norm = batchnorm_forward(xtest, gamma, beta, bn_param)

print('Before batch normalization, xtest becomes ') # should be [[0.6123719
print(xtest_norm)
```

```
Before batch normalization, xtrain has
  means:  [20.]
  stds:   [8.16496581]

After batch normalization, xtrain has
  means:  [0.]
  stds:   [0.99999925]

After batch normalization, the running mean and the running variance are
updated to
[[1.]]
[[4.28333333]]
After many iterations, the running mean and the running variance are upda
ted to
[[20.]]
[[66.66666667]]
Before batch normalization, xtest becomes
[[0.61237198]]
```

In [34]:
```
# Compare with tf.layers.batch_normalization

# Simulate the forward pass for a two-layer network
np.random.seed(15009)
N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn(N, D1)

W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)


# initialize parameters for batch normalization
bn_param = {}
bn_param['mode'] = 'train'
bn_param['eps'] = 1e-4
bn_param['momentum'] = 0.95
bn_param['running_mean'] = np.zeros([1, a.shape[1]])
bn_param['running_var'] = np.ones([1, a.shape[1]])

# random gamma and beta
gamma = np.random.rand(D3) + 1.0
beta = np.random.rand(D3)


# Setting up a tensorflow bn layer using the same set of parameters.

tf.reset_default_graph()
tfa = tf.placeholder(tf.float32, shape=[None, a.shape[1]])

# used to control the mode
is_training = tf.placeholder_with_default(False, (), 'is_training')

# the axis setting is a little strange to me. But you can understand it as
# the running mean
tfa_norm = tf.layers.batch_normalization(tfa, axis=1, momentum=0.95, epsilo
                                          beta_initializer=tf.constant_initi
                                          gamma_initializer=tf.constant_init
                                          moving_mean_initializer=tf.zeros_i
                                          moving_variance_initializer=tf.one
                                          training=is_training)

# this operation is for undating running mean and running variance.
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

session = tf.Session()

# initialize parameters
session.run(tf.global_variables_initializer())


outputs = []
nbatch = 3
batch_size=10

for ibatch in range(nbatch):
    # fetch the batch
    a_batch = a[ibatch * batch_size : (ibatch + 1) * batch_size]

    # batch normlaization with your implementation
    a_nprun = batchnorm_forward(a_batch, gamma, beta, bn_param)

    # batch normalization with the tensorflow layer. Also update the runnin
    a_tfrun, _ = session.run([tfa_norm, update_ops], feed_dict={tfa: a_batc

    print("Training batch %d: difference from the two implementations is %f


# enterining test mode
bn_param['mode'] = 'test'

for ibatch in range(nbatch):
    a_batch = a[ibatch * batch_size : (ibatch + 1) * batch_size]

    a_nprun = batchnorm_forward(a_batch, gamma, beta, bn_param)

    # run batch normalization in test mode. No need to update the running m
    a_tfrun = session.run(tfa_norm, feed_dict={tfa: a_batch.astype(np.float


    print("Test batch %d: difference from the two implementations is %f" %
```

```
Training batch 0: difference from the two implementations is 0.000001
Training batch 1: difference from the two implementations is 0.000015
Training batch 2: difference from the two implementations is 0.000001
Test batch 0: difference from the two implementations is 0.000000
Test batch 1: difference from the two implementations is 0.000000
Test batch 2: difference from the two implementations is 0.000002
```

Fully Connected Nets with Batch Normalization

Now that you have a working implementation for batch normalization. Then you need to go back to your `FullyConnectedNet` in the file `implementations/fc_net.py` . Modify the implementation to add batch normalization.

When the `use_bn` flag is set, the network should apply batch normalization before each ReLU nonlinearity. The outputs from the last layer of the network should not be normalized.

Batchnorm for deep networks

Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

In [5]:
```
from implementations.fc_net import FullyConnectedNet

np.random.seed(15009)
# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

num_train = 1000

X_train = data['X_train'][:num_train]
X_train = np.reshape(X_train, [X_train.shape[0], -1])
y_train = data['y_train'][:num_train]

X_val = data['X_val']
X_val = np.reshape(X_val, [X_val.shape[0], -1])
y_val = data['y_val']


bn_model = FullyConnectedNet(input_size=X_train.shape[1],
                             hidden_size=hidden_dims,
                             output_size=10,
                             centering_data=True,
                             use_dropout=False,
                             use_bn=True)

# use an aggresive learning rate
bn_trace = bn_model.train(X_train, y_train, X_val, y_val,
                          learning_rate=5e-4,
                          reg=np.float32(0.01),
                          keep_prob=0.5,
                          num_iters=800,
                          batch_size=100,
                          verbose=True) # train the model with batch normal
```

```
WARNING:tensorflow:From /Users/thomasklimek/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/op_def_library.py:263: colocate_with
(from tensorflow.python.framework.ops) is deprecated and will be removed
in a future version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From /Users/thomasklimek/Downloads/comp150a2/implement
ations/fc_net.py:213: batch_normalization (from tensorflow.python.layers.
normalization) is deprecated and will be removed in a future version.
Instructions for updating:
Use keras.layers.batch_normalization instead.
WARNING:tensorflow:From /Users/thomasklimek/Downloads/comp150a2/implement
ations/fc_net.py:138: softmax_cross_entropy_with_logits (from tensorflow.
python.ops.nn_ops) is deprecated and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.

WARNING:tensorflow:From /Users/thomasklimek/anaconda3/lib/python3.7/site-
packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflo
w.python.ops.math_ops) is deprecated and will be removed in a future vers
```

```
ion.
Instructions for updating:
Use tf.cast instead.
iteration 0 / 800: objective 232.335617
iteration 100 / 800: objective 62.526760
iteration 200 / 800: objective 5.490889
iteration 300 / 800: objective 3.853646
iteration 400 / 800: objective 3.367236
iteration 500 / 800: objective 3.126927
iteration 600 / 800: objective 2.983306
iteration 700 / 800: objective 2.886577
```

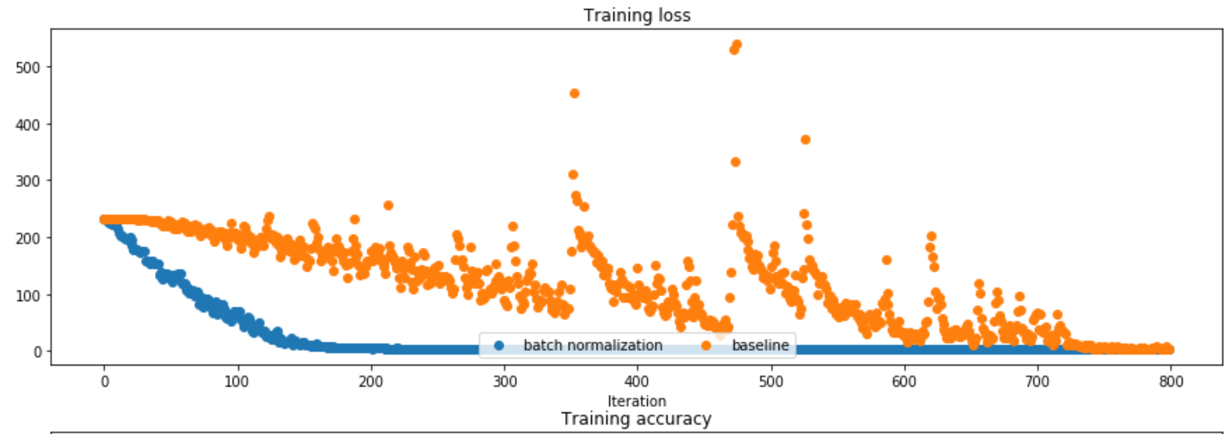
Train a fully connected network without batch normalization

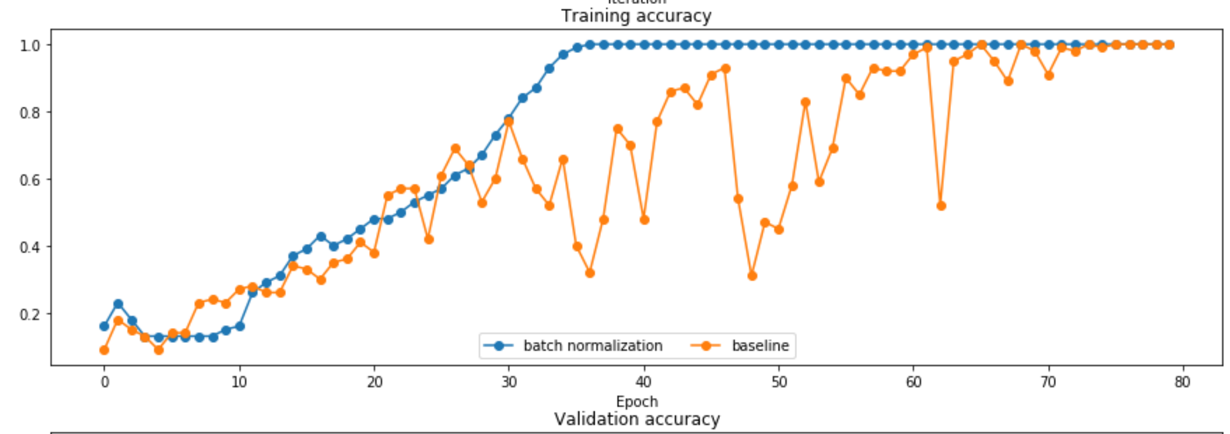
In [6]:
```
model = FullyConnectedNet(input_size=X_train.shape[1],
                          hidden_size=hidden_dims,
                          output_size=10,
                          centering_data=True,
                          use_dropout=False,
                          use_bn=False)

# use an aggresive learning rate
baseline_trace = model.train(X_train, y_train, X_val, y_val,
                            learning_rate=5e-4,
                            reg=np.float32(0.01),
                            num_iters=800,
                            batch_size=100,
                            verbose=True) # train the model without batch r
```

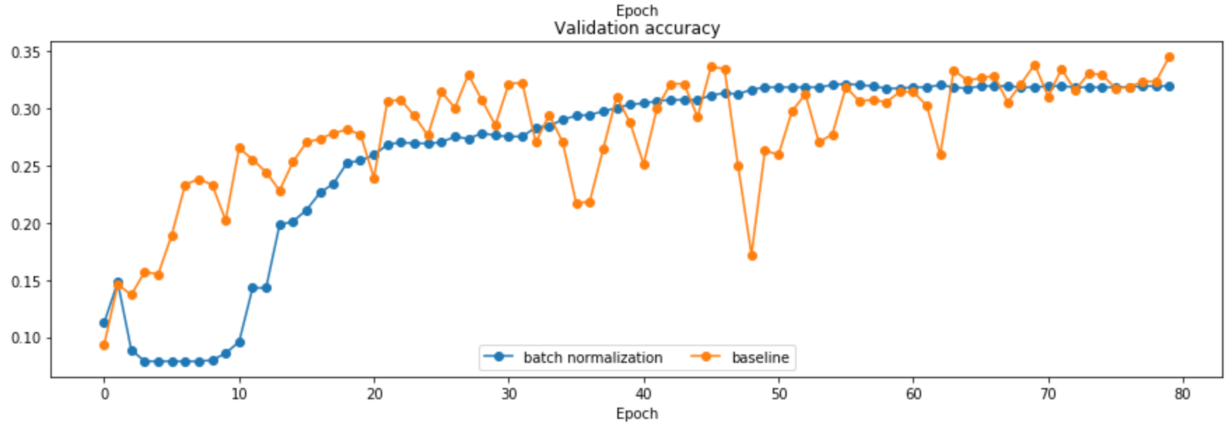
```
iteration 0 / 800: objective 232.582138
iteration 100 / 800: objective 206.320709
iteration 200 / 800: objective 176.051270
iteration 300 / 800: objective 89.852188
iteration 400 / 800: objective 145.754822
iteration 500 / 800: objective 149.130859
iteration 600 / 800: objective 32.878620
iteration 700 / 800: objective 67.951195
```

Run the following to visualize the results from two networks trained above. You should find that using batch normalization helps the network to converge much faster.

In [7]:
```
def plot_training_history(title, label, bl_plot, bn_plots, bl_marker='.', b
    """utility function for plotting training history"""
    plt.title(title)
    plt.xlabel(label)
    num_bn = len(bn_plots)

    for i in range(num_bn):
        label='batch normalization'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)

    label='baseline'
    if labels is not None:
        label += str(labels[0])

    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)


plt.subplot(3, 1, 1)
plot_training_history('Training loss','Iteration', baseline_trace['objectiv
                      [bn_trace['objective_history']], bl_marker='o', bn_ma
plt.subplot(3, 1, 2)
plot_training_history('Training accuracy','Epoch', baseline_trace['train_ac
                      [bn_trace['train_acc_history']], bl_marker='-o', bn_m
plt.subplot(3, 1, 3)
plot_training_history('Validation accuracy','Epoch', baseline_trace['val_ac
                      [bn_trace['val_acc_history']], bl_marker='-o', bn_mar

plt.gcf().set_size_inches(15, 15)
plt.show()
```







Batch normalization and initialization

We will now run a small experiment to study the interaction of batch normalization and weight initialization.

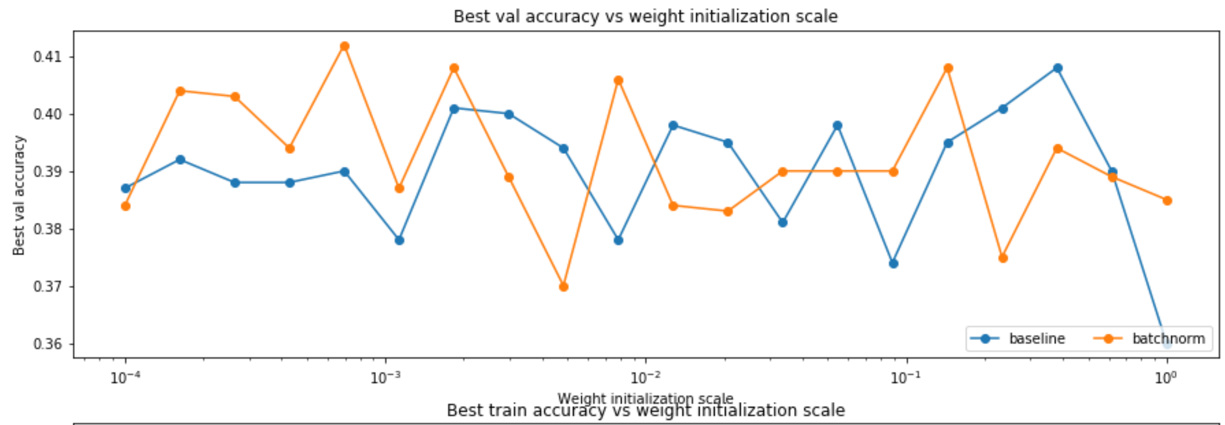
The first cell will train 8-layer networks both with and without batch normalization using different scales for weight initialization. The second cell will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

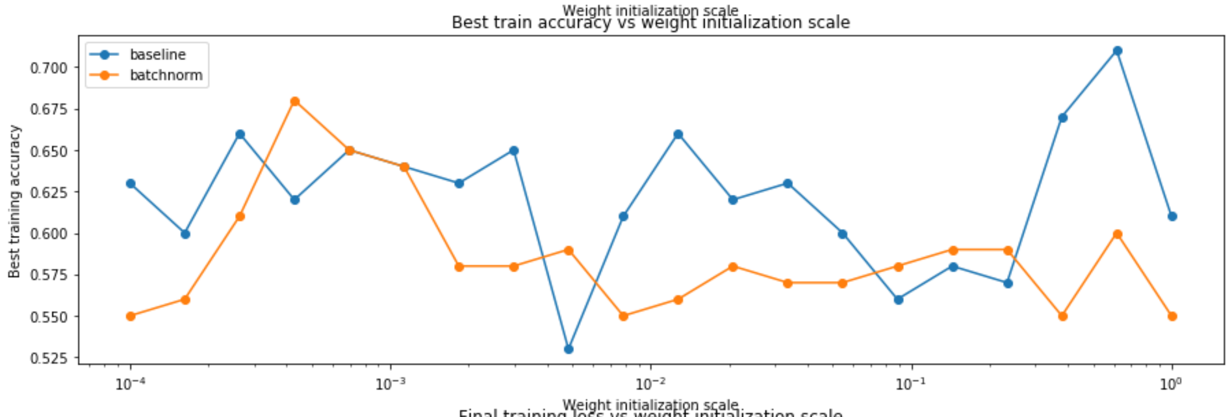
LIPING: I tried multiple configurations, but I did not find significant improvement from batch normalization. See if you can get clear improvement with your configurations.

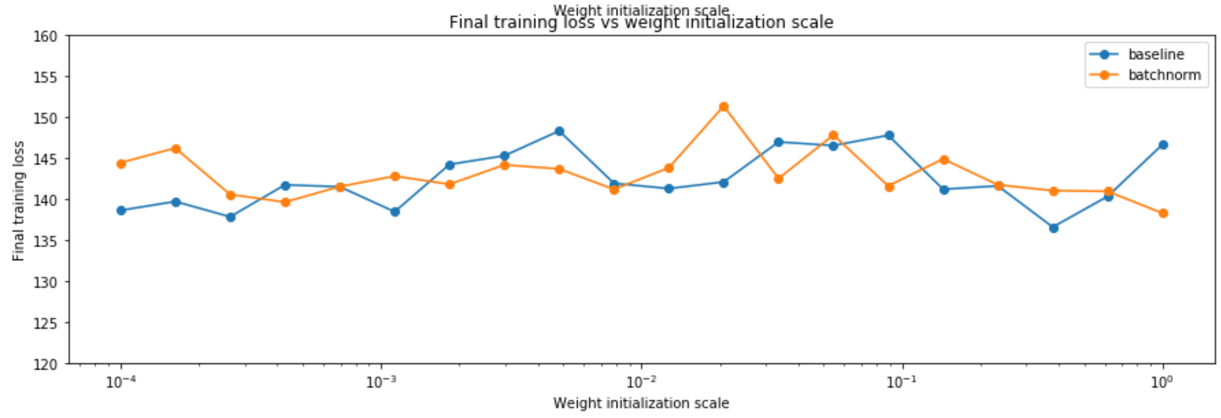
In [ ]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 10000

X_train = data['X_train'][:num_train]
X_train = np.reshape(X_train, [X_train.shape[0], -1])
y_train = data['y_train'][:num_train]

X_val = data['X_val']
X_val = np.reshape(X_val, [X_val.shape[0], -1])
y_val = data['y_val']

bn_net_ws = {}
baseline_ws = {}

weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
  print('Running weight scale=%f at round %d / %d' % (weight_scale, i + 1,


  bn_model = FullyConnectedNet(input_size=X_train.shape[1],
                               hidden_size=hidden_dims,
                               output_size=10,
                               centering_data=True,
                               use_dropout=False,
                               use_bn=True)

  # use an aggresive learning rate
  bn_net_ws[weight_scale] = bn_model.train(X_train, y_train, X_val, y_val,
                            learning_rate=1e-2,
                            reg=np.float32(1e-5),
                            keep_prob=0.5,
                            num_iters=1000,
                            batch_size=100,
                            verbose=True) # train the model with batch normal


  model = FullyConnectedNet(input_size=X_train.shape[1],
                            hidden_size=hidden_dims,
                            output_size=10,
                            centering_data=True,
                            use_dropout=False,
                            use_bn=True)

  # use an aggresive learning rate
  baseline_ws[weight_scale] = model.train(X_train, y_train, X_val, y_val,
                              learning_rate=1e-2,
                              reg=np.float32(1e-5),
                              num_iters=1000,
                              batch_size=100,
                              verbose=True)
```

```
Running weight scale=0.000100 at round 1 / 20
iteration 0 / 1000: objective 233.189835
iteration 100 / 1000: objective 213.591492
iteration 200 / 1000: objective 200.433151
iteration 300 / 1000: objective 187.779266
iteration 400 / 1000: objective 175.908569
iteration 500 / 1000: objective 169.276962
iteration 600 / 1000: objective 166.379684
iteration 700 / 1000: objective 158.826981
iteration 800 / 1000: objective 147.057220
iteration 900 / 1000: objective 145.747345
iteration 0 / 1000: objective 230.705185
iteration 100 / 1000: objective 217.186539
iteration 200 / 1000: objective 201.979187
iteration 300 / 1000: objective 196.978333
iteration 400 / 1000: objective 196.514465
iteration 500 / 1000: objective 188.552139
iteration 600 / 1000: objective 176.873596
iteration 700 / 1000: objective 168.814529
iteration 800 / 1000: objective 161.952850
iteration 900 / 1000: objective 161.383911
Running weight scale=0.000162 at round 2 / 20
iteration 0 / 1000: objective 231.216522
iteration 100 / 1000: objective 209.378464
iteration 200 / 1000: objective 199.480591
iteration 300 / 1000: objective 194.065536
iteration 400 / 1000: objective 179.163879
iteration 500 / 1000: objective 173.117477
iteration 600 / 1000: objective 168.215775
iteration 700 / 1000: objective 163.710190
iteration 800 / 1000: objective 161.352875
iteration 900 / 1000: objective 161.656372
iteration 0 / 1000: objective 234.763535
iteration 100 / 1000: objective 208.443909
iteration 200 / 1000: objective 201.943466
iteration 300 / 1000: objective 189.438141
iteration 400 / 1000: objective 182.224762
iteration 500 / 1000: objective 170.534744
iteration 600 / 1000: objective 158.194061
iteration 700 / 1000: objective 149.503693
iteration 800 / 1000: objective 141.042740
iteration 900 / 1000: objective 148.155731
Running weight scale=0.000264 at round 3 / 20
iteration 0 / 1000: objective 228.916122
iteration 100 / 1000: objective 210.585968
iteration 200 / 1000: objective 204.088882
iteration 300 / 1000: objective 195.176224
iteration 400 / 1000: objective 185.583130
iteration 500 / 1000: objective 177.081573
iteration 600 / 1000: objective 166.818222
iteration 700 / 1000: objective 167.466675
iteration 800 / 1000: objective 164.106979
iteration 900 / 1000: objective 157.886108
iteration 0 / 1000: objective 228.990707
iteration 100 / 1000: objective 210.425278
iteration 200 / 1000: objective 198.664307
iteration 300 / 1000: objective 202.191116
iteration 400 / 1000: objective 179.734634
iteration 500 / 1000: objective 176.176727
iteration 600 / 1000: objective 172.164612
iteration 700 / 1000: objective 166.659698
iteration 800 / 1000: objective 161.414749
iteration 900 / 1000: objective 157.705612
Running weight scale=0.000428 at round 4 / 20
iteration 0 / 1000: objective 230.535263
iteration 100 / 1000: objective 208.512253
iteration 200 / 1000: objective 199.176834
iteration 300 / 1000: objective 189.041626
iteration 400 / 1000: objective 182.453537
iteration 500 / 1000: objective 172.279465
iteration 600 / 1000: objective 172.149673
iteration 700 / 1000: objective 164.576614
iteration 800 / 1000: objective 159.812973
iteration 900 / 1000: objective 148.704346
iteration 0 / 1000: objective 230.348511
iteration 100 / 1000: objective 210.216080
iteration 200 / 1000: objective 202.613388
iteration 300 / 1000: objective 198.318893
iteration 400 / 1000: objective 191.446503
iteration 500 / 1000: objective 172.307831
iteration 600 / 1000: objective 168.532639
iteration 700 / 1000: objective 158.258255
iteration 800 / 1000: objective 163.867310
```

In [45]:
```
# Plot results of weight scale experiment
best_train_accs, bn_best_train_accs = [], []
best_val_accs, bn_best_val_accs = [], []
final_train_loss, bn_final_train_loss = [], []

for ws in weight_scales:
  best_train_accs.append(max(baseline_ws[ws]['train_acc_history']))
  bn_best_train_accs.append(max(bn_net_ws[ws]['train_acc_history']))

  best_val_accs.append(max(baseline_ws[ws]['val_acc_history']))
  bn_best_val_accs.append(max(bn_net_ws[ws]['val_acc_history']))

  final_train_loss.append(np.mean(baseline_ws[ws]['objective_history'][-100
  bn_final_train_loss.append(np.mean(bn_net_ws[ws]['objective_history'][-10

plt.subplot(3, 1, 1)
plt.title('Best val accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best val accuracy')
plt.semilogx(weight_scales, best_val_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
plt.title('Best train accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best training accuracy')
plt.semilogx(weight_scales, best_train_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_train_accs, '-o', label='batchnorm')
plt.legend()

plt.subplot(3, 1, 3)
plt.title('Final training loss vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Final training loss')
plt.semilogx(weight_scales, final_train_loss, '-o', label='baseline')
plt.semilogx(weight_scales, bn_final_train_loss, '-o', label='batchnorm')
plt.legend()
plt.gca().set_ylim(120, 160)

plt.gcf().set_size_inches(15, 15)
plt.show()
```







Inline Question 1:

Describe the results of this experiment. How does the scale of weight initialization affect models with/without batch normalization differently, and why?

## Answer:

```
batch normalization reduces train's dependency on weight initialization.
```

Batch normalization and batch size

We will now run a small experiment to study the interaction of batch normalization and batch size.

The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second cell will plot training accuracy and validation set accuracy over time.

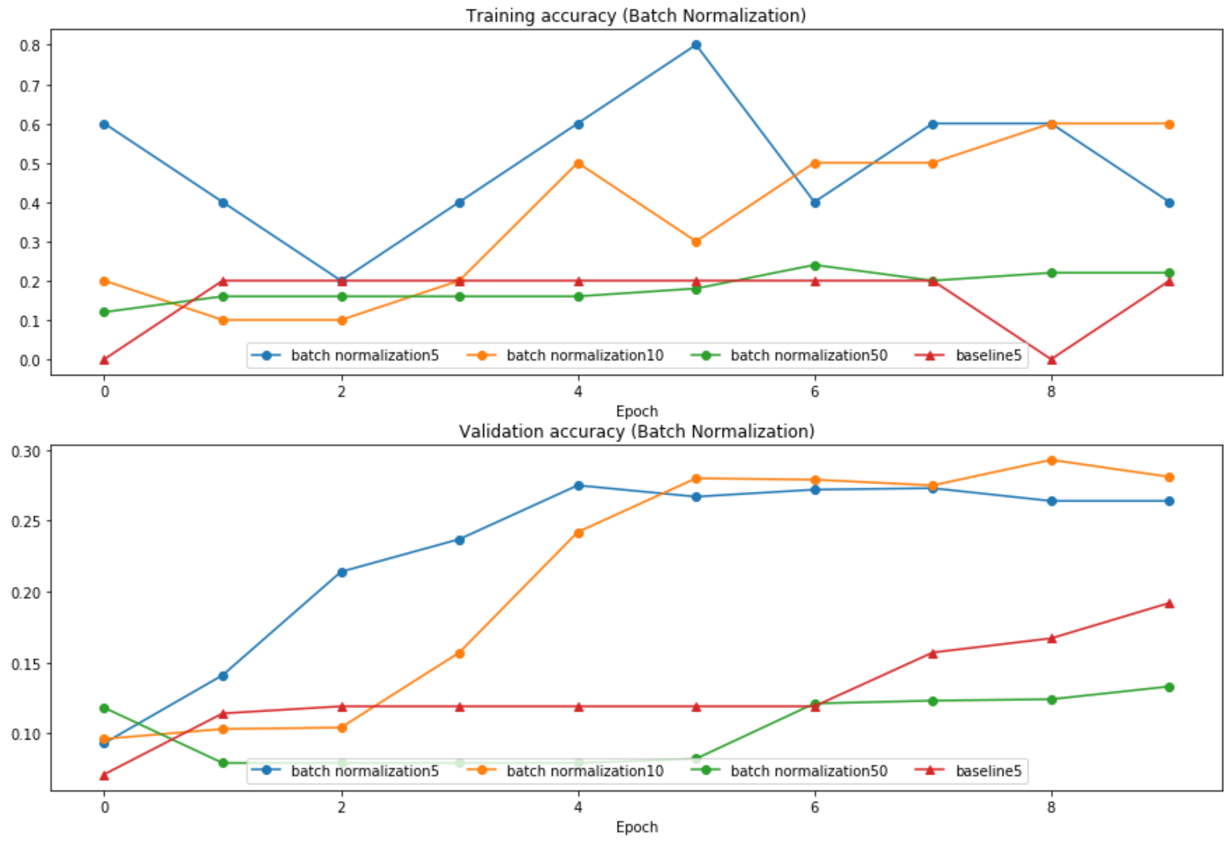
Here is a link about batch sizes in batch normalization: https://www.graphcore.ai/posts/revisiting-small-batch-training-for-deep-neural-networks (https://www.graphcore.ai/posts/revisiting-small-batch-training-for-deep-neural-networks)

In [46]:
```
def run_batchsize_experiments():
    np.random.seed(15009)
    # Try training a very deep net with batchnorm
    hidden_dims = [50, 50, 50, 50, 50]

    num_train = 1000

    X_train = data['X_train'][:num_train]
    X_train = np.reshape(X_train, [X_train.shape[0], -1])
    y_train = data['y_train'][:num_train]

    X_val = data['X_val']
    X_val = np.reshape(X_val, [X_val.shape[0], -1])
    y_val = data['y_val']

    num_epochs = 10
    batch_sizes = [5,10,50]


    batch_size = batch_sizes[0]
    print('No normalization: batch size = ', 5)
    baseline = FullyConnectedNet(input_size=X_train.shape[1],
                                 hidden_size=hidden_dims,
                                 output_size=10,
                                 centering_data=True,
                                 use_dropout=False,
                                 use_bn=False)

    # use an aggresive learning rate
    baseline_trace = baseline.train(X_train, y_train, X_val, y_val,
                                    learning_rate=10**-3,
                                    reg=np.float32(1e-5),
                                    num_iters=num_train * num_epochs // bat
                                    batch_size=batch_size,
                                    verbose=True) # train the model with ba





    bn_traces = []
    for i in range(len(batch_sizes)):

        batch_size = batch_sizes[i]
        print('Normalization: batch size = ',batch_size)


        bn_model = FullyConnectedNet(input_size=X_train.shape[1],
                                 hidden_size=hidden_dims,
                                 output_size=10,
                                 centering_data=True,
                                 use_dropout=False,
                                 use_bn=True)

        # use an aggresive learning rate
        bn_net_trace = bn_model.train(X_train, y_train, X_val, y_val,
                                learning_rate=10**-3,
                                reg=np.float32(1e-5),
                                num_iters=num_train * num_epochs // batch_size ,
                                batch_size=batch_size,
                                verbose=True) # train the model with batch normal


          bn_traces.append(bn_net_trace)

      return bn_traces, baseline_trace, batch_sizes

batch_sizes = [5,10,50]
bn_traces, baseline_trace, batch_sizes = run_batchsize_experiments()
```

```
No normalization: batch size =  5
iteration 0 / 2000: objective 11.514305
iteration 100 / 2000: objective 11.530845
iteration 200 / 2000: objective 11.489232
iteration 300 / 2000: objective 11.541492
iteration 400 / 2000: objective 11.467241
iteration 500 / 2000: objective 11.549950
iteration 600 / 2000: objective 11.446713
iteration 700 / 2000: objective 11.555372
iteration 800 / 2000: objective 11.426663
iteration 900 / 2000: objective 11.555120
iteration 1000 / 2000: objective 11.404947
iteration 1100 / 2000: objective 11.544064
iteration 1200 / 2000: objective 11.375858
iteration 1300 / 2000: objective 11.504647
iteration 1400 / 2000: objective 11.316720
iteration 1500 / 2000: objective 11.314795
iteration 1600 / 2000: objective 11.028091
iteration 1700 / 2000: objective 9.682540
iteration 1800 / 2000: objective 10.528840
iteration 1900 / 2000: objective 9.623591
Normalization: batch size =  5
iteration 0 / 2000: objective 11.261142
iteration 100 / 2000: objective 9.951454
iteration 200 / 2000: objective 9.884268
iteration 300 / 2000: objective 10.035322
iteration 400 / 2000: objective 9.217606
iteration 500 / 2000: objective 9.643195
iteration 600 / 2000: objective 9.521820
iteration 700 / 2000: objective 8.670181
iteration 800 / 2000: objective 9.517444
iteration 900 / 2000: objective 9.966735
iteration 1000 / 2000: objective 10.685647
iteration 1100 / 2000: objective 8.539443
iteration 1200 / 2000: objective 10.305301
iteration 1300 / 2000: objective 9.265452
iteration 1400 / 2000: objective 10.180280
iteration 1500 / 2000: objective 8.352707
iteration 1600 / 2000: objective 10.035075
iteration 1700 / 2000: objective 7.444477
iteration 1800 / 2000: objective 8.845722
iteration 1900 / 2000: objective 7.163485
Normalization: batch size =  10
```

```
iteration 0 / 1000: objective 22.895212
iteration 100 / 1000: objective 19.451822
iteration 200 / 1000: objective 15.644375
iteration 300 / 1000: objective 15.610549
iteration 400 / 1000: objective 12.603591
iteration 500 / 1000: objective 11.346120
iteration 600 / 1000: objective 10.628396
iteration 700 / 1000: objective 9.023713
iteration 800 / 1000: objective 11.033089
iteration 900 / 1000: objective 7.790090
Normalization: batch size =  50
iteration 0 / 200: objective 116.548073
iteration 100 / 200: objective 68.617027
```

In [17]:
```
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch',
                      baseline_trace['train_acc_history'],
                      [trace['train_acc_history'] for trace in bn_traces],
                      bl_marker='-^', bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Batch Normalization)','Epoch',
                      baseline_trace['val_acc_history'],
                      [trace['val_acc_history'] for trace in bn_traces],
                      bl_marker='-^', bn_marker='-o', labels=batch_sizes)

plt.gcf().set_size_inches(15, 10)
plt.show()
```



## Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

## Answer:

Convolutional Networks

We have talked about convolutional neural networks. We will implement convolutional operations and max-pooling operation in this task to get a deeper understanding of the network.

In [7]:
```
# As usual, a bit of setup
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from implementations.layers import *
from data_utils import get_CIFAR10_data

%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-i
%load_ext autoreload
%autoreload 2

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

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

In [8]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in 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,)
```

Convolution Operation

We will implement a convolutional operation with numpy and compare it against an existing convolutional operation.

In [2]:
```
# shape is NCHW
x_shape = (2, 3, 4, 4)

# shape is FCHW
w_shape = (3, 3, 4, 4)

x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)
print(x.shape)
print(w.shape)
print(b.shape)
# permute dimensions to NHWC
x = np.transpose(x, [0, 2, 3, 1])
# permute dimensions to HWCF
w = np.transpose(w, [2, 3, 1, 0])

conv_param = {'stride': 2, 'pad': 1}

correct_out = np.array([[[[-0.08759809, -0.10987781],
                           [-0.18387192, -0.2109216 ]],
                          [[ 0.21027089,  0.21661097],
                           [ 0.22847626,  0.23004637]],
                          [[ 0.50813986,  0.54309974],
                           [ 0.64082444,  0.67101435]]],
                         [[[-0.98053589, -1.03143541],
                           [-1.19128892, -1.24695841]],
                          [[ 0.69108355,  0.66880383],
                           [ 0.59480972,  0.56776003]],
                          [[ 2.36270298,  2.36904306],
                           [ 2.38090835,  2.38247847]]]])
correct_out = np.transpose(correct_out, [0, 2, 3, 1])


tf_out = tf.nn.conv2d(
    tf.constant(x, dtype=tf.float32),
    tf.constant(w, dtype=tf.float32),
    strides=[1, 2, 2, 1],
    padding='SAME',
    use_cudnn_on_gpu=False,
    data_format='NHWC' # NHWC is the default setting of tensorflow
)

tf_conv = tf.nn.bias_add(
    tf_out,
    tf.constant(b, dtype=tf.float32),
    data_format='NHWC')


print('Difference between correct output and tf calculation:', \
                                       rel_error(tf.Session().run(tf_conv),

# Compare your output to ours; difference should be around e-8
out = conv_forward_naive(x, w, b, conv_param)
print('Difference between my implementation and correct output:', rel_error
print('Difference between my implementation and tf calculation:', rel_error
```

```
(2, 3, 4, 4)
(3, 3, 4, 4)
(3,)
Difference between correct output and tf calculation: 4.427582433439594e-
08
Difference between my implementation and correct output: 2.21214764967188
4e-08
Difference between my implementation and tf calculation: 4.52751811619895
93e-08
```

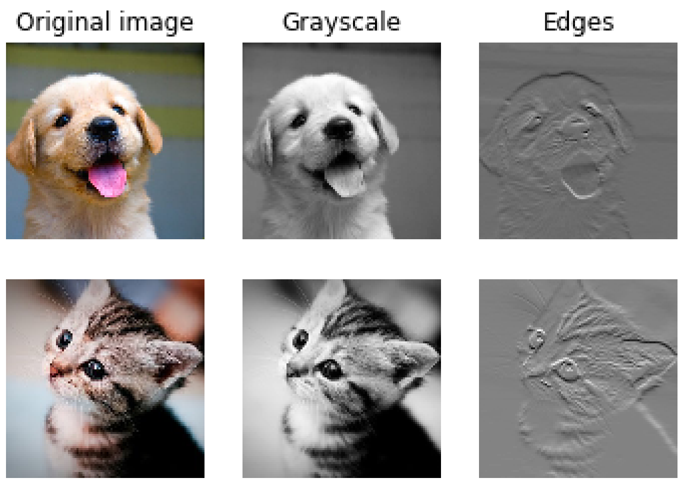
Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

In [5]:
```
from scipy.misc import imread, imresize

kitten, puppy = imread('kitten.jpg'), imread('puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
x = np.zeros((2, img_size, img_size, 3))
x[0, :, :, :] = imresize(puppy, (img_size, img_size))
x[1, :, :, :] = imresize(kitten_cropped, (img_size, img_size))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

w = np.transpose(w, [2, 3, 1, 0])

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_noax(out[0, :, :, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, :, :, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
```

```
imshow_noax(out[1, :, :, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, :, :, 1])
plt.show()
```

```
/Users/thomasklimek/anaconda3/lib/python3.7/site-packages/ipykernel_launc
her.py:3: DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing impor
ts until
/Users/thomasklimek/anaconda3/lib/python3.7/site-packages/ipykernel_launc
her.py:10: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
/Users/thomasklimek/anaconda3/lib/python3.7/site-packages/ipykernel_launc
her.py:11: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```



Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `implementations/layers.py` . Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

```
In [3]: # shape is NCHW
        x_shape = (2, 3, 4, 4)
        x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
        x = np.transpose(x, [0, 2, 3, 1])

        pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

        out = max_pool_forward_naive(x, pool_param)

        correct_out = np.array([[[[-0.26315789, -0.24842105],
                                  [-0.20421053, -0.18947368]],
                                 [[-0.14526316, -0.13052632],
                                  [-0.08631579, -0.07157895]],
                                 [[-0.02736842, -0.01263158],
                                  [ 0.03157895,  0.04631579]]],
                                [[[ 0.09052632,  0.10526316],
                                  [ 0.14947368,  0.16421053]],
                                 [[ 0.20842105,  0.22315789],
                                  [ 0.26736842,  0.28210526]],
                                 [[ 0.32631579,  0.34105263],
                                  [ 0.38526316,  0.4       ]]]])
        correct_out = np.transpose(correct_out, [0, 2, 3, 1])

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

```
Testing max_pool_forward_naive function:
difference:  1.0
```

Multilayer Convolutional Network

You need to build a convolutional network with tensorflow operations: tf.nn.conv2d, tf.nn.relu, and tf.pool. You may want to do so by modifying the fully connected network provided in this assignment.

Overfit small data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

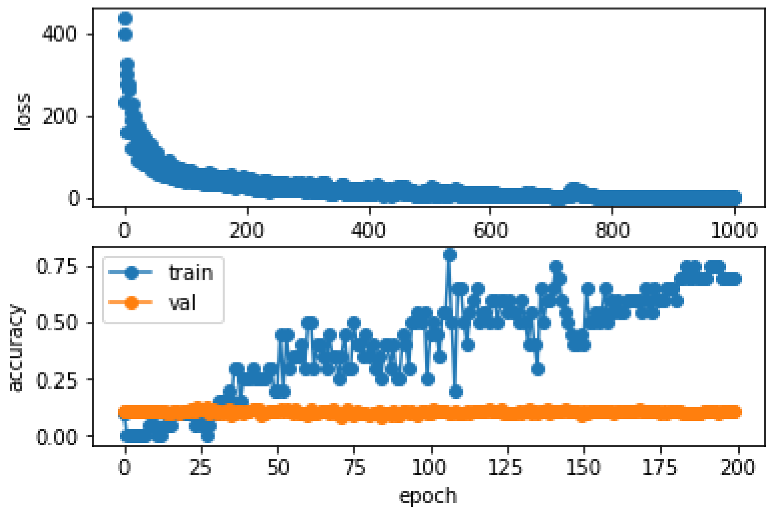
In [9]:
```
np.random.seed(15009)
tf.random.set_random_seed(15009)

from implementations.conv_net import ConvNet

num_train = 100
X_train = data['X_train'][:num_train].transpose([0, 2, 3, 1])
y_train = data['y_train'][:num_train]
X_val = data['X_val'].transpose([0, 2, 3, 1])
y_val = data['y_val']



model = ConvNet(input_size=[32, 32, 3],
                output_size=10,
                filter_size=[[3, 3, 5], [3, 3, 5], [3, 3, 5], [3, 3, 2]],
                pooling_schedule=[1, 3],
                fc_hidden_size=[50],
                use_bn = True,
                use_dropout = False)


trace = model.train(X_train, y_train, X_val, y_val,
            learning_rate=1e-3,
            reg=np.float32(5e-6),
            num_iters=1000,
            batch_size=20,
            verbose=True)
```

```
WARNING: The TensorFlow contrib module will not be included in TensorFlow
2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-con
trib-sunset.md (https://github.com/tensorflow/community/blob/master/rfcs/
20180907-contrib-sunset.md)
  * https://github.com/tensorflow/addons (https://github.com/tensorflow/a
ddons)
If you depend on functionality not listed there, please file an issue.

WARNING:tensorflow:From /Users/thomasklimek/anaconda3/lib/python3.7/site-
packages/tensorflow/python/framework/op_def_library.py:263: colocate_with
(from tensorflow.python.framework.ops) is deprecated and will be removed
 in a future version.
Instructions for updating:
Colocations handled automatically by placer.
8.0 8.0
before convolution (?, 32, 32, 3)
before add bias (?, 32, 32, 5)
after add bias (?, 32, 32, 5)
after convolution (?, 32, 32, 5)
WARNING:tensorflow:From /Users/thomasklimek/Downloads/comp150a2/implement
ations/conv_net.py:275: batch_normalization (from tensorflow.python.layer
s.normalization) is deprecated and will be removed in a future version.
Instructions for updating:
Use keras.layers.batch_normalization instead.
```

```
before convolution (?, 32, 32, 5)
before add bias (?, 32, 32, 5)
after add bias (?, 32, 32, 5)
after convolution (?, 32, 32, 5)
WARNING:tensorflow:From /Users/thomasklimek/Downloads/comp150a2/implement
ations/conv_net.py:269: max_pooling2d (from tensorflow.python.layers.pool
ing) is deprecated and will be removed in a future version.
Instructions for updating:
Use keras.layers.max_pooling2d instead.
before convolution (?, 16, 16, 5)
before add bias (?, 16, 16, 5)
after add bias (?, 16, 16, 5)
after convolution (?, 16, 16, 5)
before convolution (?, 16, 16, 5)
before add bias (?, 16, 16, 2)
after add bias (?, 16, 16, 2)
after convolution (?, 16, 16, 2)
128
WARNING:tensorflow:From /Users/thomasklimek/Downloads/comp150a2/implement
ations/conv_net.py:189: softmax_cross_entropy_with_logits (from tensorflo
w.python.ops.nn_ops) is deprecated and will be removed in a future versio
n.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.

WARNING:tensorflow:From /Users/thomasklimek/anaconda3/lib/python3.7/site-
packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflo
w.python.ops.math_ops) is deprecated and will be removed in a future vers
ion.
Instructions for updating:
Use tf.cast instead.
iteration 0 / 1000: objective 437.744720
iteration 100 / 1000: objective 31.433693
iteration 200 / 1000: objective 21.286558
iteration 300 / 1000: objective 15.192031
iteration 400 / 1000: objective 12.231076
iteration 500 / 1000: objective 8.750280
iteration 600 / 1000: objective 10.414330
iteration 700 / 1000: objective 3.817036
iteration 800 / 1000: objective 3.073395
iteration 900 / 1000: objective 2.773842
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

In [23]:
```
plt.subplot(2, 1, 1)
plt.plot(trace['objective_history'], 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(trace['train_acc_history'], '-o')
plt.plot(trace['val_acc_history'], '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



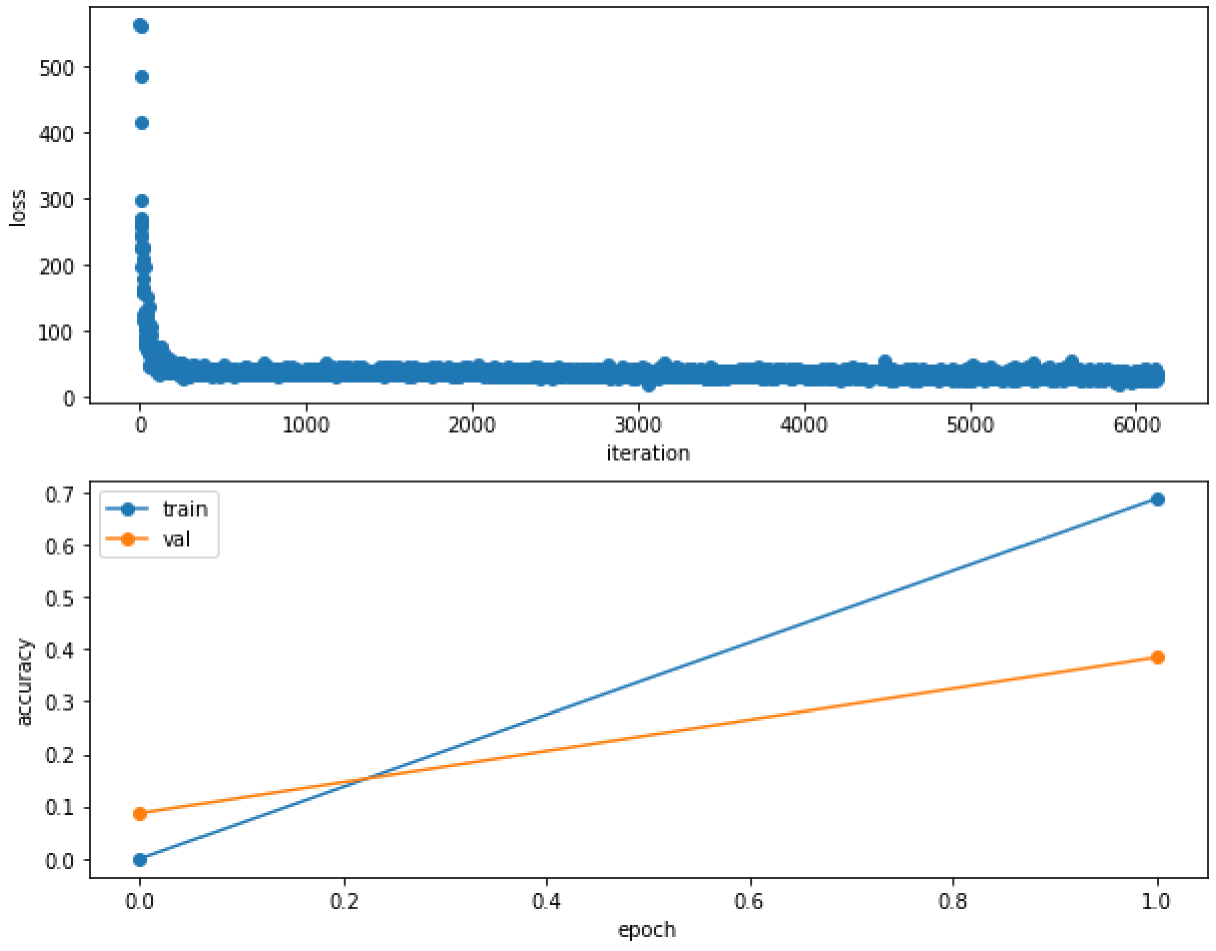
Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

In [19]:
```
X_train = data['X_train'].transpose([0, 2, 3, 1])
y_train = data['y_train']
X_val = data['X_val'].transpose([0, 2, 3, 1])
y_val = data['y_val']

num_train = X_train.shape[0]
batch_size = 16

model = ConvNet(input_size=[32, 32, 3],
                output_size=10,
                filter_size=[[9, 9, 8], [7, 7, 16], [5, 5, 32]],
                pooling_schedule=[0, 1, 2],
                fc_hidden_size=[32],
               use_bn=True)


trace = model.train(X_train, y_train, X_val, y_val,
            learning_rate=1e-3,
            reg=np.float32(5e-6),
            num_iters=(num_train * 2 // batch_size + 1),
            batch_size=batch_size,
            verbose=True)
```

```
iteration 0 / 6126: objective 564.417847
iteration 100 / 6126: objective 58.476097
iteration 200 / 6126: objective 49.103870
iteration 300 / 6126: objective 38.214092
iteration 400 / 6126: objective 37.490974
iteration 500 / 6126: objective 36.985348
iteration 600 / 6126: objective 38.545746
iteration 700 / 6126: objective 42.438438
iteration 800 / 6126: objective 41.128532
iteration 900 / 6126: objective 35.569687
iteration 1000 / 6126: objective 42.883736
iteration 1100 / 6126: objective 38.087051
iteration 1200 / 6126: objective 38.985126
iteration 1300 / 6126: objective 37.515514
iteration 1400 / 6126: objective 37.102352
iteration 1500 / 6126: objective 34.994362
iteration 1600 / 6126: objective 42.087067
iteration 1700 / 6126: objective 36.731857
iteration 1800 / 6126: objective 35.863190
iteration 1900 / 6126: objective 33.324478
iteration 2000 / 6126: objective 38.110695
iteration 2100 / 6126: objective 42.799942
iteration 2200 / 6126: objective 35.978863
iteration 2300 / 6126: objective 37.596661
iteration 2400 / 6126: objective 35.440834
iteration 2500 / 6126: objective 31.667885
iteration 2600 / 6126: objective 38.450424
iteration 2700 / 6126: objective 35.932213
iteration 2800 / 6126: objective 32.386425
iteration 2900 / 6126: objective 34.231255
iteration 3000 / 6126: objective 37.434288
iteration 3100 / 6126: objective 35.995594
iteration 3200 / 6126: objective 33.596657
iteration 3300 / 6126: objective 30.809631
iteration 3400 / 6126: objective 37.208248
iteration 3500 / 6126: objective 33.014923
iteration 3600 / 6126: objective 32.022263
iteration 3700 / 6126: objective 37.906059
iteration 3800 / 6126: objective 37.241680
iteration 3900 / 6126: objective 31.426018
iteration 4000 / 6126: objective 30.822554
iteration 4100 / 6126: objective 39.956539
iteration 4200 / 6126: objective 39.257317
iteration 4300 / 6126: objective 31.160807
iteration 4400 / 6126: objective 34.096138
iteration 4500 / 6126: objective 31.431416
iteration 4600 / 6126: objective 39.975765
iteration 4700 / 6126: objective 29.645998
iteration 4800 / 6126: objective 32.865955
iteration 4900 / 6126: objective 32.707985
iteration 5000 / 6126: objective 32.246025
iteration 5100 / 6126: objective 35.088245
iteration 5200 / 6126: objective 29.250404
iteration 5300 / 6126: objective 29.136822
iteration 5400 / 6126: objective 28.385168
iteration 5500 / 6126: objective 31.430676
iteration 5600 / 6126: objective 27.866518
iteration 5700 / 6126: objective 35.185852
iteration 5800 / 6126: objective 30.769199
iteration 5900 / 6126: objective 30.053125
iteration 6000 / 6126: objective 30.943306
iteration 6100 / 6126: objective 35.488895
```

In [20]:
```
plt.subplot(2, 1, 1)
plt.plot(trace['objective_history'], 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(trace['train_acc_history'], '-o')
plt.plot(trace['val_acc_history'], '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

In [23]:
```
from vis_utils import visualize_grid

grid = visualize_grid(model.get_params()['filter'][0].transpose(3, 0, 1, 2)
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```

```
---------------------------------------------------------------------------
--
AttributeError                            Traceback (most recent call las
t)
<ipython-input-23-1eddac391d97> in <module>
      1 from vis_utils import visualize_grid
      2 
----> 3 grid = visualize_grid(model.get_params()['filter'][0].transpose(3
, 0, 1, 2))
      4 plt.imshow(grid.astype('uint8'))
      5 plt.axis('off')

AttributeError: 'RefVariable' object has no attribute 'transpose'
```

In [ ]: