Convolutional neural networks

In this notebook, we'll put together our convolutional layers to implement a 3-layer CNN. Then, we'll ask you to implement a CNN that can achieve > 65% validation error on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

If you have not completed the Spatial BatchNorm Notebook, please see the following description from that notebook:

Please copy and paste your prior implemented code from HW #4 to start this assignment. If you did not correctly implement the layers in HW #4, you may collaborate with a classmate to use their layer implementations from HW #4. You may also visit TA or Prof OH to correct your implementation.

You'll want to copy and paste from HW #4:

```
- layers.py for your FC network layers, as well as batchnorm and dropout.
- layer_utils.py for your combined FC network layers.
- optim.py for your optimizers.
```

Be sure to place these in the `nndl/` directory so they're imported correctly. Note, as announced in class, we will not be releasing our solutions.

In [1]:
```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from nndl.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_g
radient
from nndl.layers import *
from nndl.conv_layers import *
from cs231n.fast_layers import *
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-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))))
```

In [2]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].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,)
```

Three layer CNN

In this notebook, you will implement a three layer CNN. The `ThreeLayerConvNet` class is in `nndl/cnn.py` . You'll need to modify that code for this section, including the initialization, as well as the calculation of the loss and gradients. You should be able to use the building blocks you have either earlier coded or that we have provided. Be sure to use the fast layers.

The architecture of this CNN will be:

conv - relu - 2x2 max pool - affine - relu - affine - softmax

We won't use batchnorm yet. You've also done enough of these to know how to debug; use the cells below.

Note: As we are implementing several layers CNN networks. The gradient error can be expected for the `eval_numerical_gradient()` function. If your `W1 max relative error` and `W2 max relative error` are around or below 0.01, they should be acceptable. Other errors should be less than 1e-5.

In [ ]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose
=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num
, grads[param_name])))
```

Overfit small dataset

To check your CNN implementation, let's overfit a small dataset.

In [ ]:
```python
num_train = 100
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'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

In [ ]:
```python
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

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

Train the network

Now we train the 3 layer CNN on CIFAR-10 and assess its accuracy.

In [ ]:
```python
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

Get > 65% validation accuracy on CIFAR-10.

In the last part of the assignment, we'll now ask you to train a CNN to get better than 65% validation accuracy on CIFAR-10.

Things you should try:

- Filter size: Above we used 7x7; but VGGNet and onwards showed stacks of 3x3 filters are good.
- Number of filters: Above we used 32 filters. Do more or fewer do better?
- Batch normalization: Try adding spatial batch normalization after convolution layers and vanilla batch normalization aafter affine layers. Do your networks train faster?
- Network architecture: Can a deeper CNN do better? Consider these architectures:
  - [conv-relu-pool]xN - conv - relu - [affine]xM - [softmax or SVM]
  - [conv-relu-pool]XN - [affine]XM - [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN - [affine]xM - [softmax or SVM]

Tips for training

For each network architecture that you try, you should tune the learning rate and regularization strength. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.

In [10]:
```python
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #
model = ThreeLayerConvNet(  filter_size=3,
                            num_filters=128,
                            weight_scale=0.001,
                            hidden_dim=1024,
                            reg=0.002)

solver = Solver(model, data,
                num_epochs=20, batch_size=128,
                update_rule='adam',
                optim_config={
                  'learning_rate': 5e-4,
                },
                verbose=True, print_every=100)
solver.train()

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 7640) loss: 2.336331
(Epoch 0 / 20) train acc: 0.098000; val_acc: 0.088000
(Iteration 101 / 7640) loss: 1.670949
(Iteration 201 / 7640) loss: 1.511465
(Iteration 301 / 7640) loss: 1.482682
(Epoch 1 / 20) train acc: 0.538000; val_acc: 0.529000
(Iteration 401 / 7640) loss: 1.593825
(Iteration 501 / 7640) loss: 1.343916
(Iteration 601 / 7640) loss: 1.368077
(Iteration 701 / 7640) loss: 1.415682
(Epoch 2 / 20) train acc: 0.645000; val_acc: 0.616000
(Iteration 801 / 7640) loss: 1.417564
(Iteration 901 / 7640) loss: 1.178932
(Iteration 1001 / 7640) loss: 1.110138
(Iteration 1101 / 7640) loss: 1.101641
(Epoch 3 / 20) train acc: 0.716000; val_acc: 0.622000
(Iteration 1201 / 7640) loss: 1.276990
(Iteration 1301 / 7640) loss: 1.147939
(Iteration 1401 / 7640) loss: 1.121312
(Iteration 1501 / 7640) loss: 1.105080
(Epoch 4 / 20) train acc: 0.680000; val_acc: 0.641000
(Iteration 1601 / 7640) loss: 1.257500
(Iteration 1701 / 7640) loss: 1.118692
(Iteration 1801 / 7640) loss: 1.162882
(Iteration 1901 / 7640) loss: 1.152434
(Epoch 5 / 20) train acc: 0.728000; val_acc: 0.647000
(Iteration 2001 / 7640) loss: 0.933882
(Iteration 2101 / 7640) loss: 1.007060
(Iteration 2201 / 7640) loss: 1.018173
(Epoch 6 / 20) train acc: 0.740000; val_acc: 0.626000
(Iteration 2301 / 7640) loss: 1.053089
(Iteration 2401 / 7640) loss: 0.852686
(Iteration 2501 / 7640) loss: 0.836276
(Iteration 2601 / 7640) loss: 0.973918
(Epoch 7 / 20) train acc: 0.772000; val_acc: 0.655000
(Iteration 2701 / 7640) loss: 0.833156
(Iteration 2801 / 7640) loss: 0.960028
(Iteration 2901 / 7640) loss: 0.869738
(Iteration 3001 / 7640) loss: 0.894365
(Epoch 8 / 20) train acc: 0.755000; val_acc: 0.638000
(Iteration 3101 / 7640) loss: 1.066237
(Iteration 3201 / 7640) loss: 0.925506
(Iteration 3301 / 7640) loss: 0.901143
(Iteration 3401 / 7640) loss: 0.895855
(Epoch 9 / 20) train acc: 0.780000; val_acc: 0.620000
(Iteration 3501 / 7640) loss: 1.014221
(Iteration 3601 / 7640) loss: 0.876468
(Iteration 3701 / 7640) loss: 0.873967
(Iteration 3801 / 7640) loss: 0.898300
(Epoch 10 / 20) train acc: 0.780000; val_acc: 0.630000
(Iteration 3901 / 7640) loss: 0.906659
(Iteration 4001 / 7640) loss: 0.734977
(Iteration 4101 / 7640) loss: 0.798377
(Iteration 4201 / 7640) loss: 0.797348
(Epoch 11 / 20) train acc: 0.824000; val_acc: 0.670000
(Iteration 4301 / 7640) loss: 0.720607
(Iteration 4401 / 7640) loss: 0.774881
(Iteration 4501 / 7640) loss: 0.938662
(Epoch 12 / 20) train acc: 0.826000; val_acc: 0.661000
(Iteration 4601 / 7640) loss: 0.838905
(Iteration 4701 / 7640) loss: 0.690243
```

```
(Iteration 4801 / 7640) loss: 0.606958
(Iteration 4901 / 7640) loss: 0.607002
(Epoch 13 / 20) train acc: 0.814000; val_acc: 0.643000
(Iteration 5001 / 7640) loss: 0.661225
(Iteration 5101 / 7640) loss: 0.810502
(Iteration 5201 / 7640) loss: 0.741615
(Iteration 5301 / 7640) loss: 0.676790
(Epoch 14 / 20) train acc: 0.822000; val_acc: 0.661000
(Iteration 5401 / 7640) loss: 0.763202
(Iteration 5501 / 7640) loss: 0.799008
(Iteration 5601 / 7640) loss: 0.720595
(Iteration 5701 / 7640) loss: 0.797355
(Epoch 15 / 20) train acc: 0.833000; val_acc: 0.655000
(Iteration 5801 / 7640) loss: 0.644746
(Iteration 5901 / 7640) loss: 0.649774
(Iteration 6001 / 7640) loss: 0.674265
(Iteration 6101 / 7640) loss: 0.644854
(Epoch 16 / 20) train acc: 0.867000; val_acc: 0.679000
(Iteration 6201 / 7640) loss: 0.759045
(Iteration 6301 / 7640) loss: 0.766314
(Iteration 6401 / 7640) loss: 0.625347
(Epoch 17 / 20) train acc: 0.866000; val_acc: 0.649000
(Iteration 6501 / 7640) loss: 0.650522
(Iteration 6601 / 7640) loss: 0.564891
(Iteration 6701 / 7640) loss: 0.890038
(Iteration 6801 / 7640) loss: 0.613686
(Epoch 18 / 20) train acc: 0.860000; val_acc: 0.650000
(Iteration 6901 / 7640) loss: 0.586294
(Iteration 7001 / 7640) loss: 0.735426
(Iteration 7101 / 7640) loss: 0.626093
(Iteration 7201 / 7640) loss: 0.606037
(Epoch 19 / 20) train acc: 0.856000; val_acc: 0.650000
(Iteration 7301 / 7640) loss: 0.767815
(Iteration 7401 / 7640) loss: 0.694980
(Iteration 7501 / 7640) loss: 0.537208
(Iteration 7601 / 7640) loss: 0.575530
(Epoch 20 / 20) train acc: 0.890000; val_acc: 0.667000
```