CNN

February 23, 2022

1 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.

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.

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

import numpy as np
import matplotlib.pyplot as plt
from nndl.cnn import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient_array,␣
  ↪eval_numerical_gradient
from nndl.layers import *
from nndl.conv_layers import *
from utils.fast_layers import *
from utils.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))))
```

[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,)
```

1.1 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.

[16]:
```
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])))
```

```
W1 max relative error: 0.000472281533099284
W2 max relative error: 0.025845029740362478
W3 max relative error: 3.706450129438602e-05
b1 max relative error: 1.2151661982257614e-05
b2 max relative error: 5.895729661031291e-06
b3 max relative error: 1.6963323024909378e-09
```

1.1.1 Overfit small dataset

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

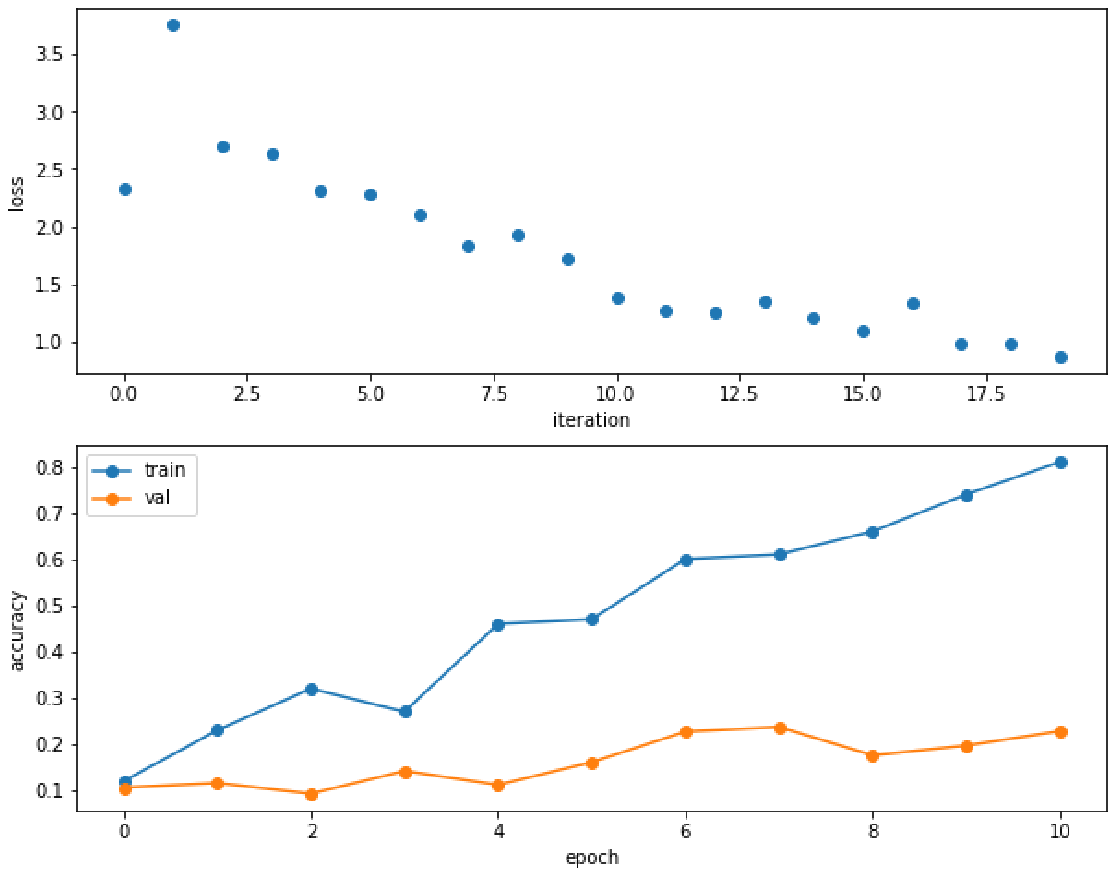
[12]:
```
num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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()
```

```
(Iteration 1 / 20) loss: 2.332895
(Epoch 0 / 10) train acc: 0.120000; val_acc: 0.106000
(Iteration 2 / 20) loss: 3.759300
(Epoch 1 / 10) train acc: 0.230000; val_acc: 0.116000
(Iteration 3 / 20) loss: 2.703979
(Iteration 4 / 20) loss: 2.635007
(Epoch 2 / 10) train acc: 0.320000; val_acc: 0.093000
(Iteration 5 / 20) loss: 2.311823
(Iteration 6 / 20) loss: 2.276250
```

```
(Epoch 3 / 10) train acc: 0.270000; val_acc: 0.142000
(Iteration 7 / 20) loss: 2.112561
(Iteration 8 / 20) loss: 1.836517
(Epoch 4 / 10) train acc: 0.460000; val_acc: 0.112000
(Iteration 9 / 20) loss: 1.931373
(Iteration 10 / 20) loss: 1.719567
(Epoch 5 / 10) train acc: 0.470000; val_acc: 0.161000
(Iteration 11 / 20) loss: 1.387747
(Iteration 12 / 20) loss: 1.277199
(Epoch 6 / 10) train acc: 0.600000; val_acc: 0.227000
(Iteration 13 / 20) loss: 1.250253
(Iteration 14 / 20) loss: 1.350809
(Epoch 7 / 10) train acc: 0.610000; val_acc: 0.237000
(Iteration 15 / 20) loss: 1.211695
(Iteration 16 / 20) loss: 1.096665
(Epoch 8 / 10) train acc: 0.660000; val_acc: 0.176000
(Iteration 17 / 20) loss: 1.335301
(Iteration 18 / 20) loss: 0.983051
(Epoch 9 / 10) train acc: 0.740000; val_acc: 0.197000
(Iteration 19 / 20) loss: 0.983759
(Iteration 20 / 20) loss: 0.876027
(Epoch 10 / 10) train acc: 0.810000; val_acc: 0.228000
```

[13]:

```
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()
```



1.2 Train the network

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

[14]:
```
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()
```

```
(Iteration 1 / 980) loss: 2.304573
(Epoch 0 / 1) train acc: 0.116000; val_acc: 0.113000
(Iteration 21 / 980) loss: 2.096999
(Iteration 41 / 980) loss: 2.068828
(Iteration 61 / 980) loss: 1.767029
```

```
(Iteration 81 / 980) loss: 1.937409
(Iteration 101 / 980) loss: 2.074962
(Iteration 121 / 980) loss: 1.704373
(Iteration 141 / 980) loss: 1.996682
(Iteration 161 / 980) loss: 1.938335
(Iteration 181 / 980) loss: 1.621312
(Iteration 201 / 980) loss: 2.003392
(Iteration 221 / 980) loss: 1.821429
(Iteration 241 / 980) loss: 1.806626
(Iteration 261 / 980) loss: 1.566788
(Iteration 281 / 980) loss: 1.746817
(Iteration 301 / 980) loss: 2.057273
(Iteration 321 / 980) loss: 1.892889
(Iteration 341 / 980) loss: 1.599492
(Iteration 361 / 980) loss: 1.366775
(Iteration 381 / 980) loss: 1.849109
(Iteration 401 / 980) loss: 1.934329
(Iteration 421 / 980) loss: 1.779593
(Iteration 441 / 980) loss: 1.836162
(Iteration 461 / 980) loss: 1.742750
(Iteration 481 / 980) loss: 1.570006
(Iteration 501 / 980) loss: 1.659527
(Iteration 521 / 980) loss: 1.628272
(Iteration 541 / 980) loss: 1.452648
(Iteration 561 / 980) loss: 1.663336
(Iteration 581 / 980) loss: 1.589796
(Iteration 601 / 980) loss: 1.584437
(Iteration 621 / 980) loss: 1.541024
(Iteration 641 / 980) loss: 1.706425
(Iteration 661 / 980) loss: 1.913553
(Iteration 681 / 980) loss: 1.561717
(Iteration 701 / 980) loss: 1.483156
(Iteration 721 / 980) loss: 1.676201
(Iteration 741 / 980) loss: 1.447575
(Iteration 761 / 980) loss: 1.466376
(Iteration 781 / 980) loss: 1.531245
(Iteration 801 / 980) loss: 1.558107
(Iteration 821 / 980) loss: 1.839975
(Iteration 841 / 980) loss: 1.524765
(Iteration 861 / 980) loss: 1.321584
(Iteration 881 / 980) loss: 1.804096
(Iteration 901 / 980) loss: 1.523731
(Iteration 921 / 980) loss: 1.580294
(Iteration 941 / 980) loss: 1.519452
(Iteration 961 / 980) loss: 1.544355
(Epoch 1 / 1) train acc: 0.426000; val_acc: 0.461000
```

2 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.

2.0.1 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]

2.0.2 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.

[15]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a CNN to achieve greater than 65% validation accuracy
#   on CIFAR-10.
# ================================================================ #

model = ThreeLayerConvNet(weight_scale = 0.001,
                          hidden_dim = 500,
                          reg = 0.001,
                          num_filters = 64,
                          filter_size = 3)
solver = Solver(model, data,
                num_epochs=10, batch_size=500,
                update_rule='adam',
                optim_config={
                'learning_rate': 1e-3,
                },
                lr_decay=0.9,
```

```python
                verbose=True, print_every=15)

solver.train()

# print out the validation and test accuracy
y_val_max = np.argmax(model.loss(data['X_val']), axis=1)
y_test_max = np.argmax(model.loss(data['X_test']), axis=1)
print('Validation set accuracy: {}'.format(np.mean(y_val_max == data['y_val'])))
print('Test set accuracy: {}'.format(np.mean(y_test_max == data['y_test'])))

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

```
(Iteration 1 / 980) loss: 2.306710
(Epoch 0 / 10) train acc: 0.108000; val_acc: 0.111000
(Iteration 16 / 980) loss: 1.964806
(Iteration 31 / 980) loss: 1.692346
(Iteration 46 / 980) loss: 1.648157
(Iteration 61 / 980) loss: 1.413976
(Iteration 76 / 980) loss: 1.376599
(Iteration 91 / 980) loss: 1.307572
(Epoch 1 / 10) train acc: 0.578000; val_acc: 0.550000
(Iteration 106 / 980) loss: 1.347474
(Iteration 121 / 980) loss: 1.283153
(Iteration 136 / 980) loss: 1.274767
(Iteration 151 / 980) loss: 1.241681
(Iteration 166 / 980) loss: 1.157503
(Iteration 181 / 980) loss: 1.326320
(Iteration 196 / 980) loss: 1.151916
(Epoch 2 / 10) train acc: 0.642000; val_acc: 0.612000
(Iteration 211 / 980) loss: 1.200976
(Iteration 226 / 980) loss: 1.117324
(Iteration 241 / 980) loss: 1.214689
(Iteration 256 / 980) loss: 1.098974
(Iteration 271 / 980) loss: 1.120339
(Iteration 286 / 980) loss: 1.025518
(Epoch 3 / 10) train acc: 0.644000; val_acc: 0.607000
(Iteration 301 / 980) loss: 0.987239
(Iteration 316 / 980) loss: 0.947281
(Iteration 331 / 980) loss: 1.037511
(Iteration 346 / 980) loss: 0.941855
(Iteration 361 / 980) loss: 0.951923
(Iteration 376 / 980) loss: 0.890922
(Iteration 391 / 980) loss: 0.969366
(Epoch 4 / 10) train acc: 0.715000; val_acc: 0.642000
(Iteration 406 / 980) loss: 0.886650
(Iteration 421 / 980) loss: 0.887079
```

```
(Iteration 436 / 980) loss: 0.933474
(Iteration 451 / 980) loss: 0.841554
(Iteration 466 / 980) loss: 0.815552
(Iteration 481 / 980) loss: 0.820305
(Epoch 5 / 10) train acc: 0.744000; val_acc: 0.652000
(Iteration 496 / 980) loss: 0.757123
(Iteration 511 / 980) loss: 0.822625
(Iteration 526 / 980) loss: 0.838002
(Iteration 541 / 980) loss: 0.755983
(Iteration 556 / 980) loss: 0.791836
(Iteration 571 / 980) loss: 0.805373
(Iteration 586 / 980) loss: 0.760779
(Epoch 6 / 10) train acc: 0.810000; val_acc: 0.656000
(Iteration 601 / 980) loss: 0.785580
(Iteration 616 / 980) loss: 0.640540
(Iteration 631 / 980) loss: 0.677474
(Iteration 646 / 980) loss: 0.673489
(Iteration 661 / 980) loss: 0.735379
(Iteration 676 / 980) loss: 0.689077
(Epoch 7 / 10) train acc: 0.807000; val_acc: 0.654000
(Iteration 691 / 980) loss: 0.682792
(Iteration 706 / 980) loss: 0.658916
(Iteration 721 / 980) loss: 0.578022
(Iteration 736 / 980) loss: 0.625348
(Iteration 751 / 980) loss: 0.598868
(Iteration 766 / 980) loss: 0.562284
(Iteration 781 / 980) loss: 0.641280
(Epoch 8 / 10) train acc: 0.839000; val_acc: 0.668000
(Iteration 796 / 980) loss: 0.557105
(Iteration 811 / 980) loss: 0.552962
(Iteration 826 / 980) loss: 0.605939
(Iteration 841 / 980) loss: 0.616494
(Iteration 856 / 980) loss: 0.496712
(Iteration 871 / 980) loss: 0.489895
(Epoch 9 / 10) train acc: 0.876000; val_acc: 0.682000
(Iteration 886 / 980) loss: 0.522655
(Iteration 901 / 980) loss: 0.528631
(Iteration 916 / 980) loss: 0.495307
(Iteration 931 / 980) loss: 0.496854
(Iteration 946 / 980) loss: 0.517439
(Iteration 961 / 980) loss: 0.522481
(Iteration 976 / 980) loss: 0.438508
(Epoch 10 / 10) train acc: 0.886000; val_acc: 0.670000
Validation set accuracy: 0.682
Test set accuracy: 0.671
```

2.0.3 Final Validation Accuracy: 68.2%

2.0.4 Final Test Accuracy: 67.1%

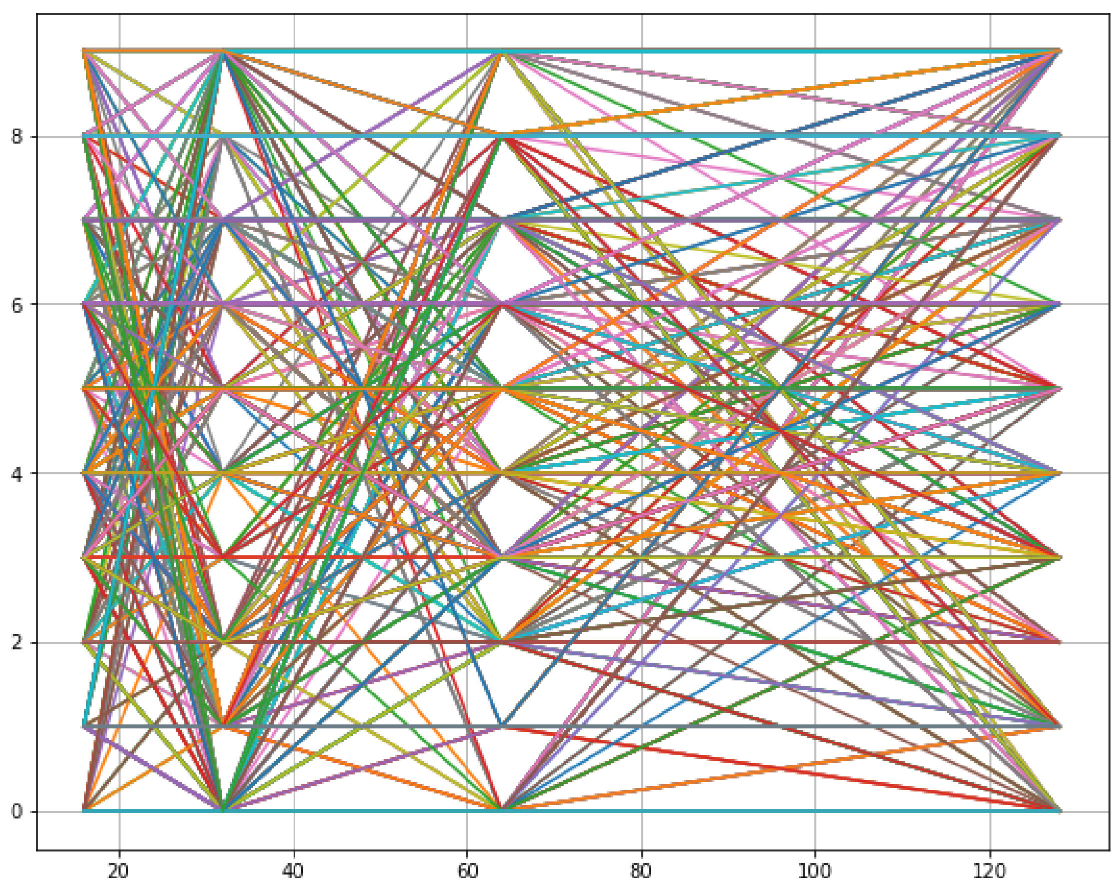
3 Param Optimization Ignore for Grading

[17]:
```
## Optimize for paramaters

y_val_max = []
y_test_max = []

param_values = [16,32,64,128]

for i,param in enumerate(param_values):
    model = ThreeLayerConvNet(weight_scale = 0.001,
                            hidden_dim = 500,
                            reg = 0.001,
                            num_filters = param,
                            filter_size = 3)
    solver = Solver(model, data,
                    num_epochs=2, batch_size=500,
                    update_rule='adam',
                    optim_config={
                    'learning_rate': 1e-3,
                    },
                    lr_decay=0.9,
                    verbose=True, print_every=15)

    solver.train()

    # print out the validation and test accuracy
    y_val_max.append(np.argmax(model.loss(data['X_val']), axis=1))
    y_test_max.append(np.argmax(model.loss(data['X_test']), axis=1))
    print('Validation set accuracy {} filters: {}'.format(param,np.
 ↪mean(y_val_max == data['y_val'])))
    print('Test set accuracy{} filters: {}'.format(param,np.mean(y_test_max ==␣
 ↪data['y_test'])))

plt.plot(param_values,y_val_max)
plt.plot(param_values,y_test_max)
plt.grid()
```

```
(Iteration 1 / 196) loss: 2.303588
(Epoch 0 / 2) train acc: 0.118000; val_acc: 0.113000
(Iteration 16 / 196) loss: 1.806035
(Iteration 31 / 196) loss: 1.707018
(Iteration 46 / 196) loss: 1.403167
(Iteration 61 / 196) loss: 1.429196
(Iteration 76 / 196) loss: 1.305658
(Iteration 91 / 196) loss: 1.369181
```

```
(Epoch 1 / 2) train acc: 0.516000; val_acc: 0.547000
(Iteration 106 / 196) loss: 1.378973
(Iteration 121 / 196) loss: 1.235189
(Iteration 136 / 196) loss: 1.242587
(Iteration 151 / 196) loss: 1.278554
(Iteration 166 / 196) loss: 1.119325
(Iteration 181 / 196) loss: 1.214951
(Iteration 196 / 196) loss: 1.206243
(Epoch 2 / 2) train acc: 0.595000; val_acc: 0.595000
Validation set accuracy 16 filters: 0.595
Test set accuracy16 filters: 0.591
(Iteration 1 / 196) loss: 2.304672
(Epoch 0 / 2) train acc: 0.154000; val_acc: 0.133000
(Iteration 16 / 196) loss: 1.790985
(Iteration 31 / 196) loss: 1.634762
(Iteration 46 / 196) loss: 1.607491
(Iteration 61 / 196) loss: 1.455025
(Iteration 76 / 196) loss: 1.468206
(Iteration 91 / 196) loss: 1.302375
(Epoch 1 / 2) train acc: 0.542000; val_acc: 0.532000
(Iteration 106 / 196) loss: 1.320351
(Iteration 121 / 196) loss: 1.340267
(Iteration 136 / 196) loss: 1.195271
(Iteration 151 / 196) loss: 1.211796
(Iteration 166 / 196) loss: 1.183481
(Iteration 181 / 196) loss: 1.123983
(Iteration 196 / 196) loss: 1.190672
(Epoch 2 / 2) train acc: 0.617000; val_acc: 0.585000
Validation set accuracy 32 filters: 0.59
Test set accuracy32 filters: 0.5965
(Iteration 1 / 196) loss: 2.306647
(Epoch 0 / 2) train acc: 0.107000; val_acc: 0.098000
(Iteration 16 / 196) loss: 1.917913
(Iteration 31 / 196) loss: 1.732582
(Iteration 46 / 196) loss: 1.594063
(Iteration 61 / 196) loss: 1.528596
(Iteration 76 / 196) loss: 1.422646
(Iteration 91 / 196) loss: 1.317119
(Epoch 1 / 2) train acc: 0.565000; val_acc: 0.529000
(Iteration 106 / 196) loss: 1.317187
(Iteration 121 / 196) loss: 1.171796
(Iteration 136 / 196) loss: 1.193191
(Iteration 151 / 196) loss: 1.201704
(Iteration 166 / 196) loss: 1.156940
(Iteration 181 / 196) loss: 1.192673
(Iteration 196 / 196) loss: 1.186760
(Epoch 2 / 2) train acc: 0.615000; val_acc: 0.585000
Validation set accuracy 64 filters: 0.5883333333333334
```

```
Test set accuracy64 filters: 0.5946666666666667
(Iteration 1 / 196) loss: 2.310781
(Epoch 0 / 2) train acc: 0.153000; val_acc: 0.139000
(Iteration 16 / 196) loss: 2.000824
(Iteration 31 / 196) loss: 1.777481
(Iteration 46 / 196) loss: 1.679931
(Iteration 61 / 196) loss: 1.463985
(Iteration 76 / 196) loss: 1.475998
(Iteration 91 / 196) loss: 1.472990
(Epoch 1 / 2) train acc: 0.507000; val_acc: 0.531000
(Iteration 106 / 196) loss: 1.392754
(Iteration 121 / 196) loss: 1.333520
(Iteration 136 / 196) loss: 1.308036
(Iteration 151 / 196) loss: 1.224769
(Iteration 166 / 196) loss: 1.178209
(Iteration 181 / 196) loss: 1.279558
(Iteration 196 / 196) loss: 1.155914
(Epoch 2 / 2) train acc: 0.609000; val_acc: 0.601000
Validation set accuracy 128 filters: 0.5915
Test set accuracy128 filters: 0.59325
```



[22]:
```
y_val_max_2 =[]
y_test_max_2 = []
for i in range(4):
    y_val_max_2.append(np.mean(y_val_max[i] == data['y_val']))
    y_test_max_2.append(np.mean(y_test_max[i] == data['y_test']))

plt.plot(param_values,y_val_max_2)
plt.plot(param_values,y_test_max_2)
plt.legend(['val','test'])
plt.grid()
```