Dropout

In this notebook, you will implement dropout. Then we will ask you to train a network with batchnorm and dropout, and acheive over 55% accuracy 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, and their layer structure. 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).

In [2]:

```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from nndl.layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_grad
ient_array
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-ipyt
hon
%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 [3]:

```
# 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,)
```

Dropout forward pass

Implement the training and test time dropout forward pass, `dropout_forward`, in `nndl/layers.py`. After that, test your implementation by running the following cell.

In [4]:

```
x = np.random.randn(500, 500) + 10

for p in [0.3, 0.6, 0.75]:
    out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
    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()
)
```

```
Running tests with p =  0.3
Mean of input:  9.99866118222753
Mean of train-time output:  10.020458173747361
Mean of test-time output:  9.99866118222753
Fraction of train-time output set to zero:  0.699364
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.99866118222753
Mean of train-time output:  10.015853911502186
Mean of test-time output:  9.99866118222753
Fraction of train-time output set to zero:  0.398892
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  9.99866118222753
Mean of train-time output:  10.02481785217837
Mean of test-time output:  9.99866118222753
Fraction of train-time output set to zero:  0.24802
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

Implement the backward pass, `dropout_backward`, in `nndl/layers.py`. After that, test your gradients by running the following cell:

In [5]:

```
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.8, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_
param)[0], x, dout)

print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560956332445e-11
```

Implement a fully connected neural network with dropout layers

Modify the `FullyConnectedNet()` class in `nndl/fc_net.py` to incorporate dropout. A dropout layer should be incorporated after every ReLU layer. Concretely, there shouldn't be a dropout at the output layer since there is no ReLU at the output layer. You will need to modify the class in the following areas:

(1) In the forward pass, you will need to incorporate a dropout layer after every relu layer.

(2) In the backward pass, you will need to incorporate a dropout backward pass layer.

Check your implementation by running the following code. Our W1 gradient relative error is on the order of 1e-6 (the largest of all the relative errors).

In [6]:

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for dropout in [0.5, 0.75, 1.0]:
    print('Running check with dropout = ', dropout)
    model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                              weight_scale=5e-2, dtype=np.float64,
                              dropout=dropout, seed=123)

    loss, grads = model.loss(X, y)
    print('Initial loss: ', loss)

    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name], verbose=Fals
e, h=1e-5)
        print('{} relative error: {}'.format(name, rel_error(grad_num, grads[n
ame])))
    print('\n')
```

```
Running check with dropout =  0.5
Initial loss:  2.309771209610118
W1 relative error: 2.694274363733021e-07
W2 relative error: 7.439246147919978e-08
W3 relative error: 1.910371122296728e-08
b1 relative error: 4.112891126518e-09
b2 relative error: 5.756217724722137e-10
b3 relative error: 1.3204470857080166e-10


Running check with dropout =  0.75
Initial loss:  2.306133548427975
W1 relative error: 8.72986097970181e-08
W2 relative error: 2.9777307885797295e-07
W3 relative error: 1.8832780806174298e-08
b1 relative error: 5.379486003985169e-08
b2 relative error: 3.6529949080385546e-09
b3 relative error: 9.987242764516995e-11


Running check with dropout =  1.0
Initial loss:  2.3053332250963194
W1 relative error: 1.2744095365229032e-06
W2 relative error: 4.678743300473988e-07
W3 relative error: 4.331673892536035e-08
b1 relative error: 4.0853539035931665e-08
b2 relative error: 1.951342257912746e-09
b3 relative error: 9.387142701440351e-11
```

Dropout as a regularizer

In class, we claimed that dropout acts as a regularizer by effectively bagging. To check this, we will train two small networks, one with dropout and one without dropout.

In [7]:

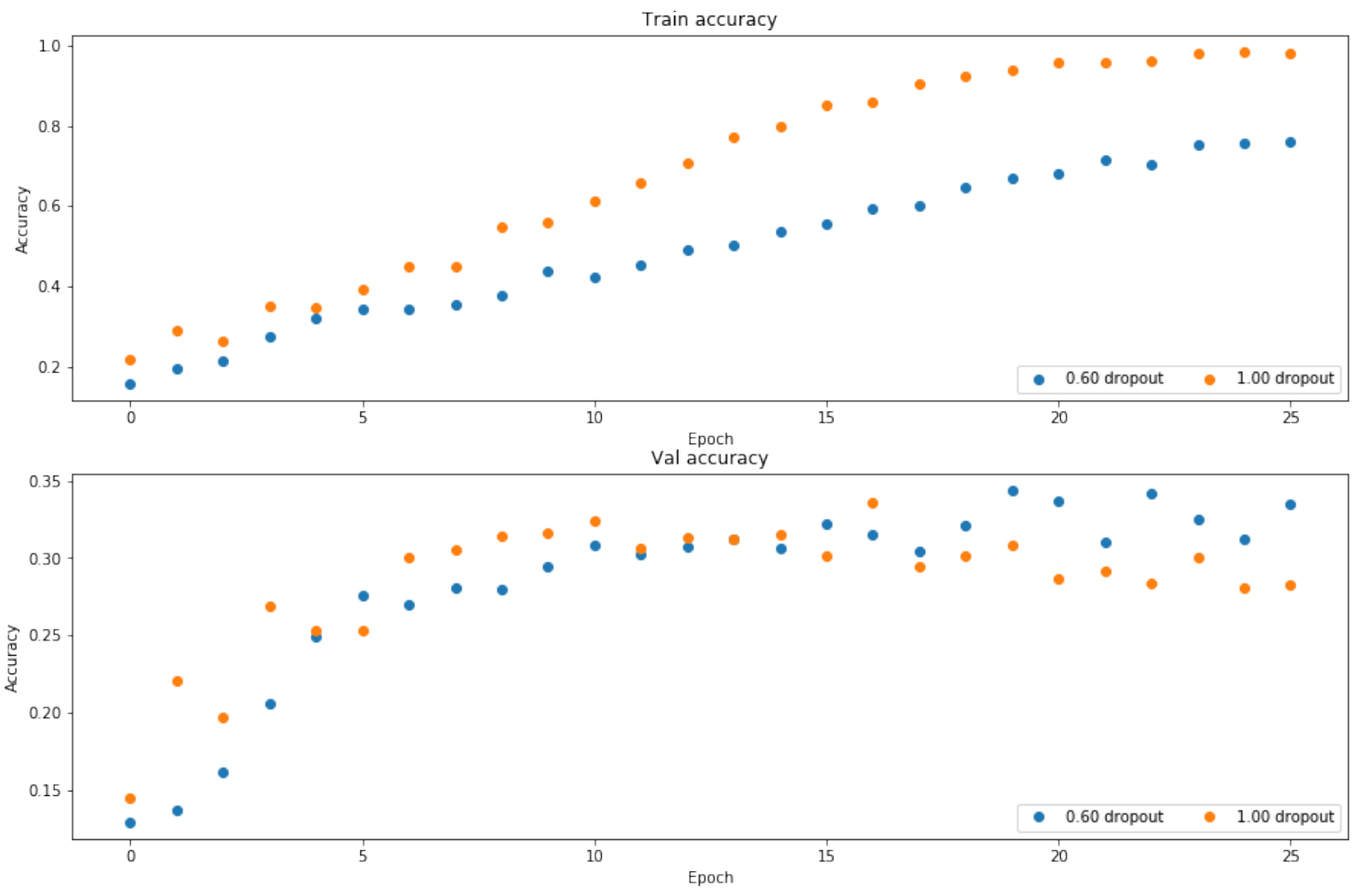
```
# Train two identical nets, one with dropout and one without

num_train = 500
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [0.6, 1.0]
for dropout in dropout_choices:
    model = FullyConnectedNet([100, 100, 100], dropout=dropout)

    solver = Solver(model, small_data,
                    num_epochs=25, batch_size=100,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': 5e-4,
                    },
                    verbose=True, print_every=100)
    solver.train()
    solvers[dropout] = solver
```

```
(Iteration 1 / 125) loss: 2.296919
(Epoch 0 / 25) train acc: 0.156000; val_acc: 0.129000
(Epoch 1 / 25) train acc: 0.194000; val_acc: 0.137000
(Epoch 2 / 25) train acc: 0.214000; val_acc: 0.161000
(Epoch 3 / 25) train acc: 0.276000; val_acc: 0.206000
(Epoch 4 / 25) train acc: 0.320000; val_acc: 0.249000
(Epoch 5 / 25) train acc: 0.342000; val_acc: 0.276000
(Epoch 6 / 25) train acc: 0.344000; val_acc: 0.270000
(Epoch 7 / 25) train acc: 0.354000; val_acc: 0.281000
(Epoch 8 / 25) train acc: 0.376000; val_acc: 0.280000
(Epoch 9 / 25) train acc: 0.436000; val_acc: 0.295000
(Epoch 10 / 25) train acc: 0.422000; val_acc: 0.308000
(Epoch 11 / 25) train acc: 0.454000; val_acc: 0.302000
(Epoch 12 / 25) train acc: 0.490000; val_acc: 0.307000
(Epoch 13 / 25) train acc: 0.502000; val_acc: 0.312000
(Epoch 14 / 25) train acc: 0.538000; val_acc: 0.306000
(Epoch 15 / 25) train acc: 0.556000; val_acc: 0.322000
(Epoch 16 / 25) train acc: 0.594000; val_acc: 0.315000
(Epoch 17 / 25) train acc: 0.600000; val_acc: 0.304000
(Epoch 18 / 25) train acc: 0.646000; val_acc: 0.321000
(Epoch 19 / 25) train acc: 0.668000; val_acc: 0.344000
(Epoch 20 / 25) train acc: 0.682000; val_acc: 0.337000
(Iteration 101 / 125) loss: 1.253072
(Epoch 21 / 25) train acc: 0.714000; val_acc: 0.310000
(Epoch 22 / 25) train acc: 0.704000; val_acc: 0.342000
(Epoch 23 / 25) train acc: 0.752000; val_acc: 0.325000
(Epoch 24 / 25) train acc: 0.758000; val_acc: 0.312000
(Epoch 25 / 25) train acc: 0.762000; val_acc: 0.335000
```

```
(Iteration 1 / 125) loss: 2.301009
(Epoch 0 / 25) train acc: 0.218000; val_acc: 0.145000
(Epoch 1 / 25) train acc: 0.288000; val_acc: 0.221000
(Epoch 2 / 25) train acc: 0.262000; val_acc: 0.197000
(Epoch 3 / 25) train acc: 0.350000; val_acc: 0.269000
(Epoch 4 / 25) train acc: 0.346000; val_acc: 0.253000
(Epoch 5 / 25) train acc: 0.392000; val_acc: 0.253000
(Epoch 6 / 25) train acc: 0.448000; val_acc: 0.300000
(Epoch 7 / 25) train acc: 0.450000; val_acc: 0.305000
(Epoch 8 / 25) train acc: 0.546000; val_acc: 0.314000
(Epoch 9 / 25) train acc: 0.560000; val_acc: 0.316000
(Epoch 10 / 25) train acc: 0.614000; val_acc: 0.324000
(Epoch 11 / 25) train acc: 0.658000; val_acc: 0.306000
(Epoch 12 / 25) train acc: 0.708000; val_acc: 0.313000
(Epoch 13 / 25) train acc: 0.770000; val_acc: 0.312000
(Epoch 14 / 25) train acc: 0.798000; val_acc: 0.315000
(Epoch 15 / 25) train acc: 0.850000; val_acc: 0.301000
(Epoch 16 / 25) train acc: 0.860000; val_acc: 0.336000
(Epoch 17 / 25) train acc: 0.904000; val_acc: 0.295000
(Epoch 18 / 25) train acc: 0.922000; val_acc: 0.301000
(Epoch 19 / 25) train acc: 0.940000; val_acc: 0.308000
(Epoch 20 / 25) train acc: 0.958000; val_acc: 0.287000
(Iteration 101 / 125) loss: 0.149086
(Epoch 21 / 25) train acc: 0.958000; val_acc: 0.292000
(Epoch 22 / 25) train acc: 0.960000; val_acc: 0.284000
(Epoch 23 / 25) train acc: 0.980000; val_acc: 0.300000
(Epoch 24 / 25) train acc: 0.984000; val_acc: 0.281000
(Epoch 25 / 25) train acc: 0.982000; val_acc: 0.283000
```

In [8]:

```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
    solver = solvers[dropout]
    train_accs.append(solver.train_acc_history[-1])
    val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout in dropout_choices:
    plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' % d
ropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
    plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dro
pout)
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()
```



Question

Based off the results of this experiment, is dropout performing regularization? Explain your answer.

Answer:

From the first plot, we can observe that train accuracy approaches to 100% when we don't perform dropout.

Please note that we've implemented an inverted dropout so when $p = 1.00$, effectively, we do not perform dropout.

If we look at the case when $p = 0.6$, we can see that training accuracy barely hits %80 and this would suggest that we are actually performing regularization due to the fact that we are not overfitting. On the other hand, we can see that validation accuracy for the dropout case is actually higher. This also suggests that dropout is performing regularization. In other words, the validation accuracy is getting higher while the training accuracy is regularized around a region or more importantly we are not overfitting the training set.

Final part of the assignment

Get over 55% validation accuracy on CIFAR-10 by using the layers you have implemented. You will be graded according to the following equation:

min(floor((X - 32%)) / 23%, 1) where if you get 55% or higher validation accuracy, you get full points.

In [9]:

```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #
data = get_CIFAR10_data()
data_training = {
    'X_train': data['X_train'],
    'y_train': data['y_train'],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}


model = FullyConnectedNet([500, 500, 500], dropout=0.5, weight_scale= 1e-2, us
e_batchnorm=True, reg=0.0)

solver = Solver(model, data_training,
                num_epochs=20, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay = 0.9,
                verbose=True, print_every=100)
solver.train()


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

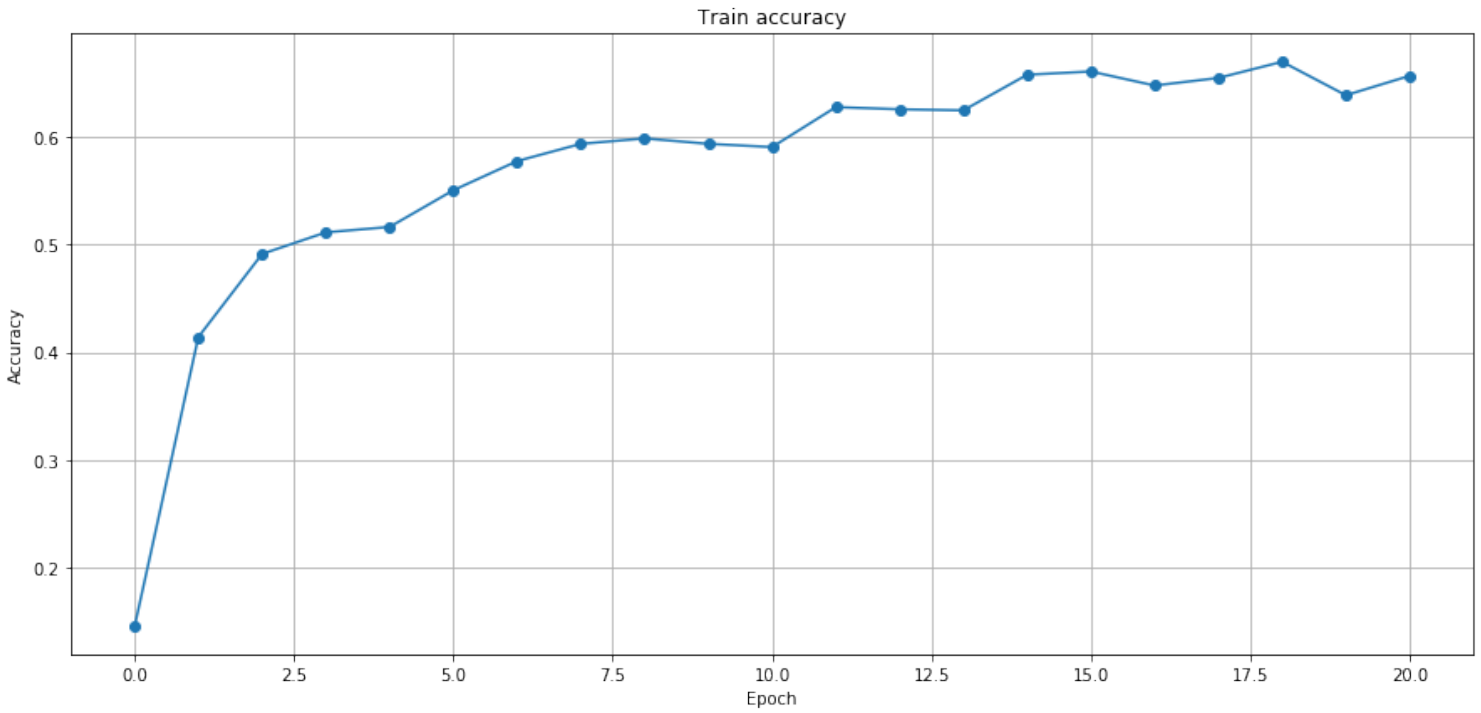
```
(Iteration 1 / 9800) loss: 2.345271
(Epoch 0 / 20) train acc: 0.146000; val_acc: 0.133000
(Iteration 101 / 9800) loss: 1.836871
(Iteration 201 / 9800) loss: 1.865793
(Iteration 301 / 9800) loss: 1.826321
(Iteration 401 / 9800) loss: 1.788521
(Epoch 1 / 20) train acc: 0.414000; val_acc: 0.460000
(Iteration 501 / 9800) loss: 1.757053
(Iteration 601 / 9800) loss: 1.556654
(Iteration 701 / 9800) loss: 1.760806
(Iteration 801 / 9800) loss: 1.406088
(Iteration 901 / 9800) loss: 1.640498
(Epoch 2 / 20) train acc: 0.491000; val_acc: 0.492000
(Iteration 1001 / 9800) loss: 1.544485
(Iteration 1101 / 9800) loss: 1.425994
(Iteration 1201 / 9800) loss: 1.721044
(Iteration 1301 / 9800) loss: 1.723351
(Iteration 1401 / 9800) loss: 1.517958
(Epoch 3 / 20) train acc: 0.511000; val_acc: 0.527000
(Iteration 1501 / 9800) loss: 1.316956
(Iteration 1601 / 9800) loss: 1.476159
(Iteration 1701 / 9800) loss: 1.417122
(Iteration 1801 / 9800) loss: 1.571144
```

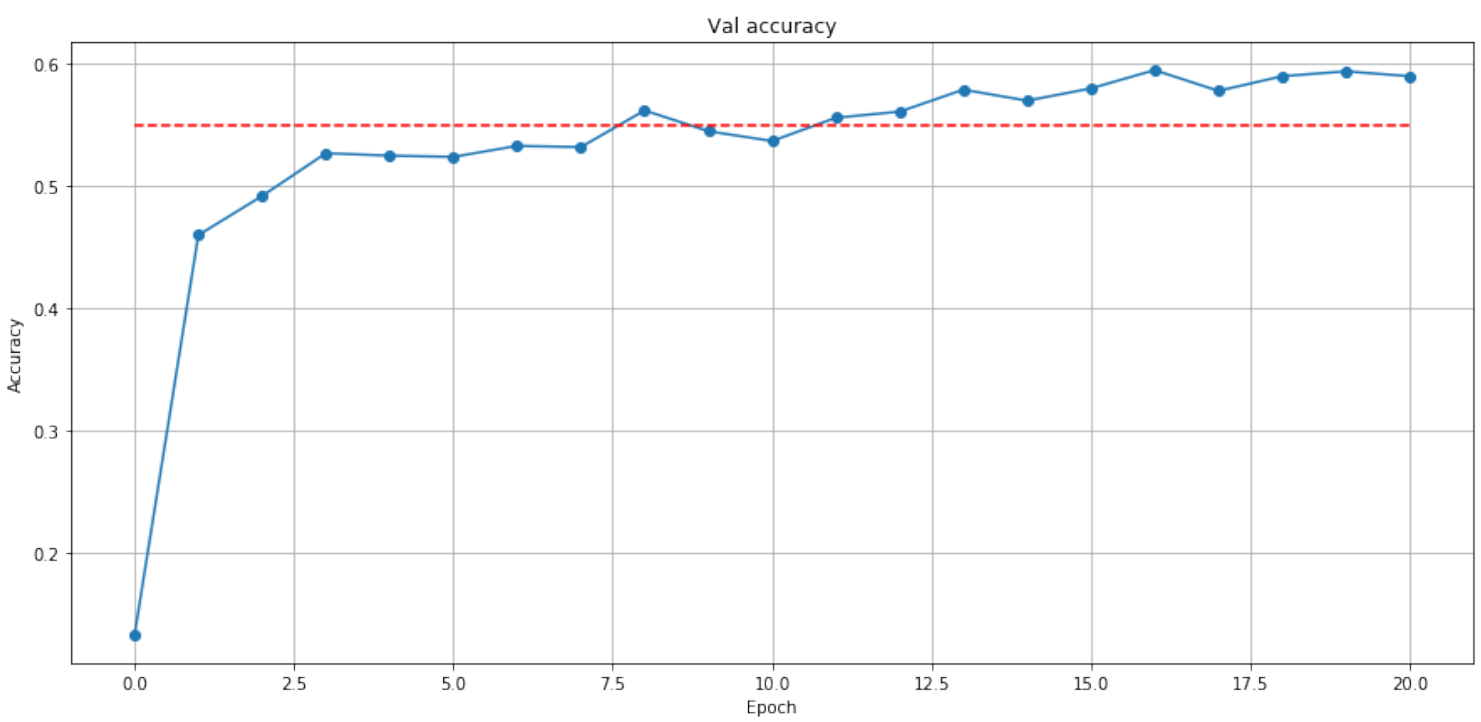
```
(Iteration 1901 / 9800) loss: 1.506970
(Epoch 4 / 20) train acc: 0.516000; val_acc: 0.525000
(Iteration 2001 / 9800) loss: 1.393406
(Iteration 2101 / 9800) loss: 1.344309
(Iteration 2201 / 9800) loss: 1.532873
(Iteration 2301 / 9800) loss: 1.510269
(Iteration 2401 / 9800) loss: 1.238353
(Epoch 5 / 20) train acc: 0.550000; val_acc: 0.524000
(Iteration 2501 / 9800) loss: 1.427981
(Iteration 2601 / 9800) loss: 1.249954
(Iteration 2701 / 9800) loss: 1.430426
(Iteration 2801 / 9800) loss: 1.444717
(Iteration 2901 / 9800) loss: 1.287213
(Epoch 6 / 20) train acc: 0.577000; val_acc: 0.533000
(Iteration 3001 / 9800) loss: 1.322541
(Iteration 3101 / 9800) loss: 1.368296
(Iteration 3201 / 9800) loss: 1.371087
(Iteration 3301 / 9800) loss: 1.374292
(Iteration 3401 / 9800) loss: 1.487730
(Epoch 7 / 20) train acc: 0.593000; val_acc: 0.532000
(Iteration 3501 / 9800) loss: 1.346944
(Iteration 3601 / 9800) loss: 1.187196
(Iteration 3701 / 9800) loss: 1.294232
(Iteration 3801 / 9800) loss: 1.350984
(Iteration 3901 / 9800) loss: 1.742692
(Epoch 8 / 20) train acc: 0.598000; val_acc: 0.562000
(Iteration 4001 / 9800) loss: 1.412027
(Iteration 4101 / 9800) loss: 1.275940
(Iteration 4201 / 9800) loss: 1.235680
(Iteration 4301 / 9800) loss: 1.308844
(Iteration 4401 / 9800) loss: 1.197585
(Epoch 9 / 20) train acc: 0.593000; val_acc: 0.545000
(Iteration 4501 / 9800) loss: 1.403956
(Iteration 4601 / 9800) loss: 1.247077
(Iteration 4701 / 9800) loss: 1.301475
(Iteration 4801 / 9800) loss: 1.237536
(Epoch 10 / 20) train acc: 0.590000; val_acc: 0.537000
(Iteration 4901 / 9800) loss: 1.037384
(Iteration 5001 / 9800) loss: 1.185195
(Iteration 5101 / 9800) loss: 1.433539
(Iteration 5201 / 9800) loss: 1.296107
(Iteration 5301 / 9800) loss: 1.269018
(Epoch 11 / 20) train acc: 0.627000; val_acc: 0.556000
(Iteration 5401 / 9800) loss: 1.184150
(Iteration 5501 / 9800) loss: 1.238093
(Iteration 5601 / 9800) loss: 1.409913
(Iteration 5701 / 9800) loss: 1.184747
(Iteration 5801 / 9800) loss: 1.296892
(Epoch 12 / 20) train acc: 0.625000; val_acc: 0.561000
(Iteration 5901 / 9800) loss: 1.241959
(Iteration 6001 / 9800) loss: 1.414284
(Iteration 6101 / 9800) loss: 1.178617
(Iteration 6201 / 9800) loss: 1.399899
(Iteration 6301 / 9800) loss: 1.073929
(Epoch 13 / 20) train acc: 0.624000; val_acc: 0.579000
(Iteration 6401 / 9800) loss: 1.277826
(Iteration 6501 / 9800) loss: 1.184416
```

```
(Iteration 6601 / 9800) loss: 1.185851
(Iteration 6701 / 9800) loss: 1.316677
(Iteration 6801 / 9800) loss: 1.086234
(Epoch 14 / 20) train acc: 0.657000; val_acc: 0.570000
(Iteration 6901 / 9800) loss: 1.147629
(Iteration 7001 / 9800) loss: 1.167673
(Iteration 7101 / 9800) loss: 0.999595
(Iteration 7201 / 9800) loss: 1.068292
(Iteration 7301 / 9800) loss: 1.195786
(Epoch 15 / 20) train acc: 0.660000; val_acc: 0.580000
(Iteration 7401 / 9800) loss: 1.129483
(Iteration 7501 / 9800) loss: 1.181328
(Iteration 7601 / 9800) loss: 1.306452
(Iteration 7701 / 9800) loss: 1.160219
(Iteration 7801 / 9800) loss: 1.053766
(Epoch 16 / 20) train acc: 0.647000; val_acc: 0.595000
(Iteration 7901 / 9800) loss: 1.250405
(Iteration 8001 / 9800) loss: 1.361988
(Iteration 8101 / 9800) loss: 1.127705
(Iteration 8201 / 9800) loss: 1.060643
(Iteration 8301 / 9800) loss: 1.300719
(Epoch 17 / 20) train acc: 0.654000; val_acc: 0.578000
(Iteration 8401 / 9800) loss: 1.095596
(Iteration 8501 / 9800) loss: 1.005118
(Iteration 8601 / 9800) loss: 1.215252
(Iteration 8701 / 9800) loss: 1.063952
(Iteration 8801 / 9800) loss: 1.108920
(Epoch 18 / 20) train acc: 0.669000; val_acc: 0.590000
(Iteration 8901 / 9800) loss: 1.124983
(Iteration 9001 / 9800) loss: 1.202398
(Iteration 9101 / 9800) loss: 1.424639
(Iteration 9201 / 9800) loss: 1.207969
(Iteration 9301 / 9800) loss: 1.105221
(Epoch 19 / 20) train acc: 0.638000; val_acc: 0.594000
(Iteration 9401 / 9800) loss: 1.118681
(Iteration 9501 / 9800) loss: 1.266613
(Iteration 9601 / 9800) loss: 1.134120
(Iteration 9701 / 9800) loss: 1.160204
(Epoch 20 / 20) train acc: 0.656000; val_acc: 0.590000
```

In [11]:

```
# Plot train and validation accuracies of the model:

train_accs = []
val_accs = []
train_accs = (solver.train_acc_history[-1])
val_accs = (solver.val_acc_history[-1])

plt.subplot(2, 1, 1)
plt.plot(solver.train_acc_history, '-o')
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid()

plt.subplot(2, 1, 2)
plt.plot(solver.val_acc_history, '-o')
horiz_line_data = np.array([0.55 for i in range(len(solver.val_acc_history))])
plt.plot(horiz_line_data, 'r--')
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid()

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





In [12]:

```
y_test_pred = np.argmax(model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(model.loss(data['X_val']), axis=1)
print('Validation set accuracy: {}'.format(np.mean(y_val_pred == data['y_val']
)))
print('Test set accuracy: {}'.format(np.mean(y_test_pred == data['y_test'])))
```

```
Validation set accuracy: 0.59
Test set accuracy: 0.575
```