In [ ]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'assignment4'
FOLDERNAME = 'CS6353/Assignments/assignment4/assignment4/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.
mount("/content/drive", force_remount=True).
/content/drive/My Drive/CS6353/Assignments/assignment4/assignment4/cs6353/datasets
--2024-11-03 05:03:49--  https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... connecte
d.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  31.2MB/s    in 5.9s

2024-11-03 05:03:55 (27.7 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/17049807
1]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/CS6353/Assignments/assignment4/assignment4
```

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

In [ ]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.fc_net import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient, eval_numerical_gradient_
from cs6353.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))))

# You can ignore the message that asks you to run Python script for now.
# It will be required in the second part of the assignment.
```

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

In [ ]:
```
# 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 forward pass

In the file `cs6353/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 [ ]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
  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())
  print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

In the file `cs6353/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

In [ ]:
```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, '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
```

```
# Error should be around e-10 or less
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560814873387e-11
```

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:

If we don't divide the values by p during training with dropout, we're using vanilla dropout instead of inverted dropout. This means at test time, we would need to scale the outputs by p to ensure the activations match the expected values from training.

Without scaling, the test-time activations would be too large since, during training, neurons see only a fraction of their inputs due to dropout, while at test time, all neurons are active. This discrepancy could lead to unstable predictions and exploding gradients.

Inverted dropout solves this by scaling the activations during training (multiplying by 1 / p). This keeps the expected activations consistent between training and testing, eliminating the need for test-time scaling and simplifying the deployment process.

Inverted dropout is preferred because it ensures consistent behavior and avoids the need for adjustments during inference.

Fully-connected nets with Dropout

In the file `cs6353/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a value that is not 1 for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

In [ ]:
```
np.random.seed(231)
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 [1, 0.75, 0.5]:
  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)
```

```
  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  2.300479089768492
W1 relative error: 1.03e-07
W2 relative error: 2.21e-05
W3 relative error: 4.56e-07
b1 relative error: 4.66e-09
b2 relative error: 2.09e-09
b3 relative error: 1.69e-10

Running check with dropout =  0.75
Initial loss:  2.3028487331533216
W1 relative error: 7.87e-07
W2 relative error: 1.12e-07
W3 relative error: 1.47e-07
b1 relative error: 1.83e-08
b2 relative error: 4.32e-09
b3 relative error: 1.75e-10

Running check with dropout =  0.5
Initial loss:  2.30427592207859
W1 relative error: 3.11e-07
W2 relative error: 2.48e-08
W3 relative error: 6.43e-08
b1 relative error: 5.37e-09
b2 relative error: 1.91e-09
b3 relative error: 1.85e-10
```

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 [ ]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
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 = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(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
```

```
1
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.350000; val_acc: 0.227000
(Epoch 2 / 25) train acc: 0.500000; val_acc: 0.230000
(Epoch 3 / 25) train acc: 0.610000; val_acc: 0.269000
(Epoch 4 / 25) train acc: 0.704000; val_acc: 0.291000
(Epoch 5 / 25) train acc: 0.778000; val_acc: 0.284000
(Epoch 6 / 25) train acc: 0.820000; val_acc: 0.271000
(Epoch 7 / 25) train acc: 0.824000; val_acc: 0.281000
(Epoch 8 / 25) train acc: 0.876000; val_acc: 0.253000
(Epoch 9 / 25) train acc: 0.920000; val_acc: 0.278000
(Epoch 10 / 25) train acc: 0.896000; val_acc: 0.267000
(Epoch 11 / 25) train acc: 0.912000; val_acc: 0.266000
(Epoch 12 / 25) train acc: 0.894000; val_acc: 0.298000
(Epoch 13 / 25) train acc: 0.914000; val_acc: 0.277000
(Epoch 14 / 25) train acc: 0.938000; val_acc: 0.260000
(Epoch 15 / 25) train acc: 0.954000; val_acc: 0.282000
(Epoch 16 / 25) train acc: 0.984000; val_acc: 0.278000
(Epoch 17 / 25) train acc: 0.966000; val_acc: 0.275000
(Epoch 18 / 25) train acc: 0.992000; val_acc: 0.274000
(Epoch 19 / 25) train acc: 0.986000; val_acc: 0.278000
(Epoch 20 / 25) train acc: 0.988000; val_acc: 0.266000
(Iteration 101 / 125) loss: 0.068826
(Epoch 21 / 25) train acc: 0.996000; val_acc: 0.267000
(Epoch 22 / 25) train acc: 0.996000; val_acc: 0.274000
(Epoch 23 / 25) train acc: 0.982000; val_acc: 0.264000
(Epoch 24 / 25) train acc: 0.974000; val_acc: 0.262000
(Epoch 25 / 25) train acc: 0.996000; val_acc: 0.270000
0.25
(Iteration 1 / 125) loss: 17.045781
(Epoch 0 / 25) train acc: 0.202000; val_acc: 0.192000
(Epoch 1 / 25) train acc: 0.358000; val_acc: 0.211000
(Epoch 2 / 25) train acc: 0.444000; val_acc: 0.280000
(Epoch 3 / 25) train acc: 0.488000; val_acc: 0.273000
(Epoch 4 / 25) train acc: 0.530000; val_acc: 0.280000
(Epoch 5 / 25) train acc: 0.574000; val_acc: 0.295000
(Epoch 6 / 25) train acc: 0.644000; val_acc: 0.318000
(Epoch 7 / 25) train acc: 0.714000; val_acc: 0.317000
(Epoch 8 / 25) train acc: 0.736000; val_acc: 0.320000
(Epoch 9 / 25) train acc: 0.760000; val_acc: 0.302000
(Epoch 10 / 25) train acc: 0.776000; val_acc: 0.307000
(Epoch 11 / 25) train acc: 0.770000; val_acc: 0.295000
(Epoch 12 / 25) train acc: 0.790000; val_acc: 0.302000
(Epoch 13 / 25) train acc: 0.804000; val_acc: 0.302000
(Epoch 14 / 25) train acc: 0.790000; val_acc: 0.301000
(Epoch 15 / 25) train acc: 0.824000; val_acc: 0.311000
(Epoch 16 / 25) train acc: 0.860000; val_acc: 0.323000
(Epoch 17 / 25) train acc: 0.832000; val_acc: 0.300000
(Epoch 18 / 25) train acc: 0.858000; val_acc: 0.283000
(Epoch 19 / 25) train acc: 0.858000; val_acc: 0.291000
(Epoch 20 / 25) train acc: 0.900000; val_acc: 0.324000
(Iteration 101 / 125) loss: 5.281125
(Epoch 21 / 25) train acc: 0.906000; val_acc: 0.300000
(Epoch 22 / 25) train acc: 0.910000; val_acc: 0.282000
(Epoch 23 / 25) train acc: 0.926000; val_acc: 0.310000
```

```
(Epoch 24 / 25) train acc: 0.928000; val_acc: 0.308000
(Epoch 25 / 25) train acc: 0.938000; val_acc: 0.305000
```

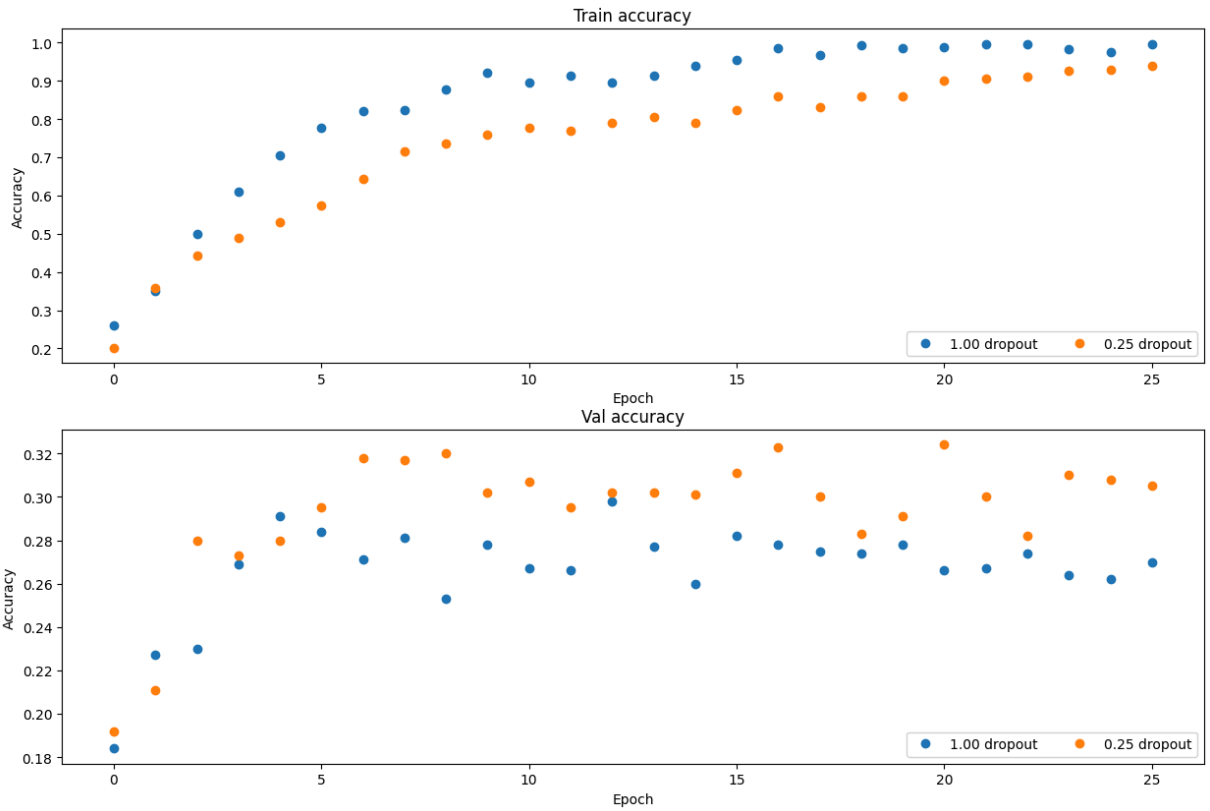
In [ ]:
```
# Plot train and validation accuracies of the two models

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' % dropout)
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' % dropout)
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:

When comparing training and validation accuracies with and without dropout, the results highlight the role of dropout as a regularizer, particularly in reducing overfitting.

Without Dropout: The model shows clear signs of overfitting. Training accuracy is extremely high (99.6% at epoch 25), while validation accuracy is much lower (27% at epoch 25). This large gap indicates that the model is fitting the training data too well but struggling to generalize to new, unseen data.

With Dropout: Dropout reduces the training accuracy slightly (93.8% at epoch 25), but the validation accuracy improves (30.5% at epoch 25). This suggests that dropout regularizes the model, preventing it from overfitting and helping it generalize better to the validation set.

Conclusion:

Dropout effectively narrows the gap between training and validation performance. While it may reduce training accuracy slightly, it leads to better validation accuracy, indicating improved generalization. This supports the conclusion that dropout helps control model complexity, reducing overfitting and improving performance on unseen data.

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 you reduce the size of the hidden layers in a fully-connected network with dropout (parameterized by keep probability p), you don't need to adjust p for overfitting concerns. Dropout adjusts proportionally, meaning the number of neurons dropped is scaled according to the layer's size.

For example, let's say a hidden layer has n = 512 neurons and you use n = 512 and p = 0.25. Then number of dropped neurons would be p * n = 128, leaving 384 neurons active. Now, if

you reduce the hidden layer size to 256 neurons and keep p = 0.25. Then number of dropped neurons would be p * n = 64, leaving 192 neurons active.

Since dropout scales the number of dropped neurons proportionally with the layer size, there's no need to modify p when decreasing the number of neurons in the hidden layers.