EE460J Lab 4

Lab Group Members: Tatsushi Matsumoto, Nick Taylor, Matthew Withey

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
from sklearn.linear_model import LogisticRegression as LR
from sklearn.model_selection import train_test_split as TTS
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn import ensemble
from sklearn.datasets import fetch_openml
```

Problem 1

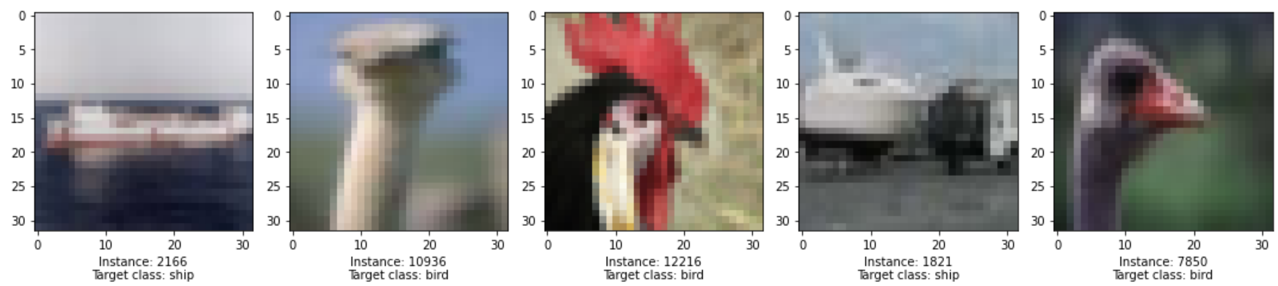
In [2]:
```
cifar = fetch_openml('CIFAR_10_Small')
cifar.data
```

Out[2]:

	a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	...	a3062	a3063	a3064	a3065	a3066	a3067	a306
0	59.0	43.0	50.0	68.0	98.0	119.0	139.0	145.0	149.0	149.0	...	59.0	58.0	65.0	59.0	46.0	57.0	104
1	154.0	126.0	105.0	102.0	125.0	155.0	172.0	180.0	142.0	111.0	...	22.0	42.0	67.0	101.0	122.0	133.0	136
2	255.0	253.0	253.0	253.0	253.0	253.0	253.0	253.0	253.0	253.0	...	78.0	83.0	80.0	69.0	66.0	72.0	79
3	28.0	37.0	38.0	42.0	44.0	40.0	40.0	24.0	32.0	43.0	...	53.0	39.0	59.0	42.0	44.0	48.0	38
4	170.0	168.0	177.0	183.0	181.0	177.0	181.0	184.0	189.0	189.0	...	92.0	88.0	85.0	82.0	83.0	79.0	78
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
19995	76.0	76.0	77.0	76.0	75.0	76.0	76.0	76.0	76.0	78.0	...	228.0	185.0	177.0	223.0	239.0	239.0	235
19996	81.0	91.0	98.0	106.0	108.0	110.0	80.0	84.0	88.0	90.0	...	126.0	107.0	143.0	155.0	156.0	160.0	173
19997	20.0	19.0	15.0	15.0	14.0	13.0	12.0	11.0	10.0	9.0	...	114.0	112.0	68.0	50.0	52.0	52.0	51
19998	25.0	15.0	23.0	17.0	23.0	51.0	74.0	91.0	114.0	137.0	...	87.0	84.0	83.0	84.0	79.0	78.0	78
19999	73.0	98.0	99.0	77.0	59.0	146.0	214.0	176.0	125.0	218.0	...	84.0	89.0	88.0	85.0	93.0	93.0	90

20000 rows × 3072 columns

In [3]:
```
# prints five random images from cifar_10_small
def print_five_images():
    classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
    plt.figure(figsize=(15,10))
    for i in range(5):
        index = random.randint(0, 20000)
        im = cifar.data.loc[index,:]
        im = np.array(im).astype(int)
        im = np.reshape(im, (3, 32, 32))
        im = im.T
        im = np.fliplr(im)
        im = np.rot90(im)
        plt.subplot(1,5, i+1)
        plt.xlabel('Instance: ' + str(index) + '\nTarget class: ' + classes[int(cifar.target.loc[index])]
        plt.imshow(im)
    plt.tight_layout()
    plt.show()

print_five_images()
```



In [4]:
```
train_img, test_img, train_lbl, test_lbl = TTS(cifar.data, cifar.target, test_size=0.25, random_state=0)
print(train_img.shape)
print(train_lbl.shape)
print(test_img.shape)
print(test_lbl.shape)
```

```
(15000, 3072)
(15000,)
(5000, 3072)
(5000,)
```

Logistic Regression, L1

In [5]:
```
def find_best_C_and_Accuracy_L1():
    C_values = [10000, 1000, 100, 10, 1, 0.1, 0.01, 0.001, 0.0001]
    grid = dict(C =C_values, multi_class=['multinomial'], solver=['saga'], tol=[0.1], penalty=['l1'], ver
    cv = KFold(n_splits=4)
    clf = LR()
    grid_search = GridSearchCV(estimator=clf, param_grid=grid, n_jobs=-1, cv=cv)
    results = grid_search.fit(train_img, train_lbl)
    print('Best for L1 penalty: accuracy of ' + str(np.round(results.best_score_, 5)) + ' using regulariz
    return results

best_L1 = find_best_C_and_Accuracy_L1()
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

convergence after 6 epochs took 31 seconds
Best for L1 penalty: accuracy of 0.38653 using regularization coefficient 10

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:   30.5s finished
```

In [6]:
```
L1_scores = pd.DataFrame.from_dict(best_L1.cv_results_)
L1_scores.drop(L1_scores.columns.difference(['param_C', 'mean_test_score']), 1, inplace=True)
L1_scores
```

```
C:\Users\mgwit\AppData\Local\Temp/ipykernel_29888/2927872455.py:2: FutureWarning: In a future version o
f pandas all arguments of DataFrame.drop except for the argument 'labels' will be keyword-only
  L1_scores.drop(L1_scores.columns.difference(['param_C', 'mean_test_score']), 1, inplace=True)
```

Out[6]:

	param_C	mean_test_score
0	10000	0.384800
1	1000	0.384333
2	100	0.384600
3	10	0.386533
4	1	0.385800
5	0.1	0.384133
6	0.01	0.385067
7	0.001	0.384400
8	0.0001	0.324733

Logistic Regression, L2

In [7]:
```
def find_best_C_and_Accuracy_L2():
    C_values = [10000, 1000, 100, 10, 1, 0.1, 0.01, 0.001, 0.0001]
    grid = dict(C =C_values, multi_class=['multinomial'], solver=['saga'], tol=[0.1], penalty=['l2'], ver
    cv = KFold(n_splits=4)
    clf = LR()
    grid_search = GridSearchCV(estimator=clf, param_grid=grid, n_jobs=-1, cv=cv)
    results = grid_search.fit(train_img, train_lbl)
    print('Best for L2 penalty: accuracy of ' + str(np.round(results.best_score_, 5)) + ' using regulariz
    return results

best_L2 = find_best_C_and_Accuracy_L2()
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

convergence after 5 epochs took 9 seconds
Best for L2 penalty: accuracy of 0.38673 using regularization coefficient 0.0001

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    9.0s finished
```

In [8]:
```
L2_scores = pd.DataFrame.from_dict(best_L2.cv_results_)
L2_scores.drop(L2_scores.columns.difference(['param_C', 'mean_test_score']), 1, inplace=True)
L2_scores
```

```
C:\Users\mgwit\AppData\Local\Temp/ipykernel_29888/2948191753.py:2: FutureWarning: In a future version o
f pandas all arguments of DataFrame.drop except for the argument 'labels' will be keyword-only
  L2_scores.drop(L2_scores.columns.difference(['param_C', 'mean_test_score']), 1, inplace=True)
```

Out[8]:

	param_C	mean_test_score
0	10000	0.384867
1	1000	0.382800
2	100	0.385200
3	10	0.384067
4	1	0.385533
5	0.1	0.382867
6	0.01	0.384933
7	0.001	0.384400
8	0.0001	0.386733

In [9]:
```
def get_loss():
    loss = np.zeros((20000, 2))
    prob_L1 = best_L1.predict_proba(cifar.data)
    prob_L2 = best_L2.predict_proba(cifar.data)
    y = cifar.target
    for i in range(20000):
        loss[i][0] = -1*np.log2(prob_L1[i, int(y[i])])
        loss[i][1] = -1*np.log2(prob_L2[i, int(y[i])])

    return pd.DataFrame(loss, columns=['Loss L1', 'Loss L2'])

loss = get_loss()
loss
```

Out[9]:

	Loss L1	Loss L2
0	1.256518	1.319383
1	4.885300	4.741637
2	0.824077	0.864509
3	3.114602	3.068512
4	3.222210	3.446920
...	...	...
19995	2.386389	2.404187
19996	2.082095	2.246407
19997	1.900496	1.785703
19998	3.938400	4.046385
19999	1.550887	1.718308

20000 rows × 2 columns

In [10]:
```
def get_non_zeroes(w, threshold):
    count = 0
    for i in range(len(w)):
        if w[i] > threshold :
            count = count + 1
    return count

def show_sparsity():
    sparsity = np.zeros((10,3))
    threshold = [0,0.1,0.01,0.001,0.0001,0.00001,0.000001,0.0000001,0.00000001,0.000000001]
    w_L1 = best_L1.best_estimator_.coef_.ravel()
    w_L2 = best_L2.best_estimator_.coef_.ravel()
    for i in range(10):
        w_L1_non_zeros = get_non_zeroes(w_L1, threshold[i])
        w_L2_non_zeros = get_non_zeroes(w_L2, threshold[i])
        sparsity[i][0] = threshold[i]
        sparsity[i][1] = 1 - (w_L1_non_zeros/len(w_L1))
        sparsity[i][2] = 1 - (w_L2_non_zeros/len(w_L2))

    sparsity_df = pd.DataFrame(sparsity, columns=['Threshold', 'sparsity of L1', 'sparsity of L2'])
    return sparsity_df

sparsity = show_sparsity()
sparsity
```

Out[10]:

	Threshold	sparsity of L1	sparsity of L2
0	0.000000e+00	0.499447	0.500651
1	1.000000e-01	1.000000	1.000000
2	1.000000e-02	1.000000	1.000000
3	1.000000e-03	1.000000	1.000000
4	1.000000e-04	0.890137	0.902474
5	1.000000e-05	0.548600	0.554232
6	1.000000e-06	0.504199	0.506055
7	1.000000e-07	0.499870	0.501432
8	1.000000e-08	0.499479	0.500684
9	1.000000e-09	0.499447	0.500651

Problem 4

In [11]:
```
def tune_random_forest_cifar_10():
    grid = {
        'n_estimators': [100, 200, 300],
        'max_depth': [10, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 5],
        'verbose': [1],
    }
    cv = KFold(n_splits=4)
    rf = ensemble.RandomForestClassifier()
    grid_search = RandomizedSearchCV(estimator=rf, param_distributions=grid, n_jobs=-1, cv=cv, verbose=2)
    results = grid_search.fit(train_img, train_lbl)
    print('Best accuracy of random forest is ' + str(np.round(results.best_score_, 5)) + ' using paramete
    return results

best_random_forest_cifar = tune_random_forest_cifar_10()
```

```
Fitting 4 folds for each of 10 candidates, totalling 40 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

Best accuracy of random forest is 0.43487 using parameters {'verbose': 1, 'n_estimators': 300, 'min_sam
ples_split': 10, 'min_samples_leaf': 2, 'max_depth': 20}

[Parallel(n_jobs=1)]: Done 300 out of 300 | elapsed:  2.0min finished
```

In [12]:
```
def tune_gb_cifar_10():
    grid = {
        'n_estimators': [100, 200, 300],
        'learning_rate': [0.1, 0.5, 1],
        'max_depth': [10, 20, 30],
        'verbose': [3],
    }
    cv = KFold(n_splits=4)
    gb = ensemble.GradientBoostingClassifier()
    grid_search = RandomizedSearchCV(estimator=gb, param_distributions=grid, n_jobs=-1, cv=cv, verbose=2)
    results = grid_search.fit(train_img, train_lbl)
    print('Best accuracy of gradient boosting is ' + str(np.round(results.best_score_, 5)) + ' using para
    return results

#best_gb_cifar = tune_gb_cifar_10()
```

Gradient Boosting was taking hours to complete and I was not able to produce an accuracy score or best parameters

EE460J Lab 4

Lab Group Members: Tatsushi Matsumoto, Nick Taylor, Matthew Withey

In [15]:
```
#Imports
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.datasets
import sklearn.metrics
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegressionCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
```

Problem 1

In [ ]:

Problem 2

In [7]:
```
#Get MNIST dataset and create data matrix and target vector

mnist = sklearn.datasets.fetch_openml("mnist_784")

X = mnist.data
y = mnist.target
```

In [ ]:
```
#Create train-test split of dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.60)

#Run multiclass logistic regression on split with l2 regularizer
logistic_Reg = LogisticRegressionCV(solver='saga' ,multi_class='multinomial')
logistic_Reg.fit(X_train, y_train)

result = logistic_Reg.predict(X_test)
log_reg_accuracy = logistic_Reg.score(result, y_test)

print("Split Accuracy = " + str(log_reg_accuracy))

#Run parameters on full dataset
full_result = logistic_Reg.predict(X)
full_accuracy = logistic_Reg.score(full_result, y)

print("Full Accuracy = " + str(full_accuracy))
```

In [ ]:
```
#Run multiclass logistic regression on split with l1 regularizer
logistic_Reg_l1 = LogisticRegressionCV(solver='saga' ,multi_class='multinomial' ,penalty='l1')
logistic_Reg_l1.fit(X_train, y_train)

result_l1 = logistic_Reg_l1.predict(X_test)
log_reg_accuracy_l1 = logistic_Reg_l1.score(result_l1, y_test)

print("Split Accuracy for l1 = " + str(log_reg_accuracy_l1))

#Run parameters on full dataset
full_result_l1 = logistic_Reg_l1.predict(X)
full_accuracy_l1 = logistic_Reg_l1.score(full_result, y)

print("Full Accuracy for l1 = " + str(full_accuracy_l1))

#Display coefficients image
```

Problem 3

In [13]:
```
#Random Forests MNIST

random_Forest = RandomForestClassifier(n_estimators=500)

random_Forest.fit(X, y)

generalization_error = np.mean(cross_val_score(random_Forest, X, y, cv = 10))
print("Generalization Error: " + str(generalization_error))

accuracy = 1 - generalization_error
print("Accuracy : " + str(accuracy))

print("\nBest n_estimator parameter: 500")
```

```
Generalization Error: 0.9705428571428574
Accuracy : 0.029457142857142626

Best n_estimator parameter: 500
```

In [ ]:
```
#Gradient Boosting MNIST

gradient_Boost = GradientBoostingClassifier()

gradient_Boost.fit(X, y)

gen_error_boost = np.mean(cross_val_score(gradient_Boost, X, y, cv = 10))
print("Generalization Error: " + str(gen_error_boost))

accuracy_boost = 1 - generalization_error
print("Accuracy : " + str(accuracy_boost))

print("\nBest n_estimator parameter: 100")
```

Problem 4

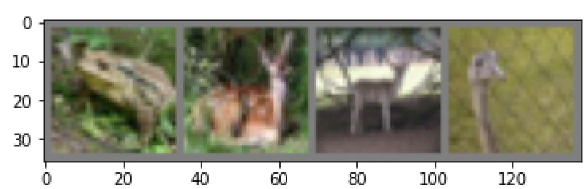
In [ ]:
```
```

In [43]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

In [44]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
Files already downloaded and verified
```

In [45]:
```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(batch_size)))
```




```
 frog  deer  deer  bird
```

In [46]:
```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

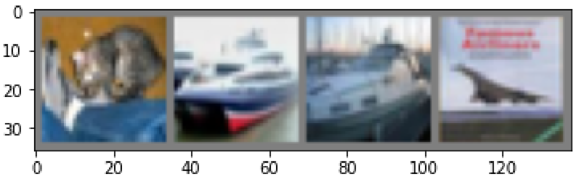
In [47]:
```
def train(data, trainloader, optimizer):

  for epoch in range(2):  # loop over the dataset multiple times

      running_loss = 0.0
      for i, data in enumerate(trainloader, 0):
          # get the inputs; data is a list of [inputs, labels]
          inputs, labels = data

          # zero the parameter gradients
          optimizer.zero_grad()

          # forward + backward + optimize
          outputs = net(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          optimizer.step()

          # print statistics
          running_loss += loss.item()
          if i % 5000 == 4999:    # print every 5000 mini-batches
              print('Epoch %d loss: %.3f' %
                    (epoch + 1, running_loss / 5000))
              running_loss = 0.0
```

In [48]:
```
def test(testloader):
  correct = 0
  total = 0
  # since we're not training, we don't need to calculate the gradients for our outputs
  with torch.no_grad():
      for data in testloader:
          images, labels = data
          # calculate outputs by running images through the network
          outputs = net(images)
          # the class with the highest energy is what we choose as prediction
          _, predicted = torch.max(outputs.data, 1)
          total += labels.size(0)
          correct += (predicted == labels).sum().item()

  print('Accuracy of the network on the 10000 test images: %d %%' % (
      100 * correct / total))
```

In [49]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
rates = [0.001, 0.0001]
momentums = [0.9, 0.95]
for rate in rates:
  for momentum in momentums:
    print('Learning Rate: %.4f Momentum: %.2f' %(rate, momentum))
    optimizer = optim.SGD(net.parameters(), lr=rate, momentum=momentum)
    train(data, trainloader, optimizer)
    test(testloader)
```

```
Learning Rate: 0.0010 Momentum: 0.90
Epoch 1 loss: 1.982
Epoch 1 loss: 1.583
Epoch 2 loss: 1.394
Epoch 2 loss: 1.331
Accuracy of the network on the 10000 test images: 54 %
Learning Rate: 0.0010 Momentum: 0.95
Epoch 1 loss: 1.230
Epoch 1 loss: 1.199
Epoch 2 loss: 1.116
Epoch 2 loss: 1.124
Accuracy of the network on the 10000 test images: 58 %
Learning Rate: 0.0001 Momentum: 0.90
Epoch 1 loss: 1.038
Epoch 1 loss: 1.047
Epoch 2 loss: 0.974
Epoch 2 loss: 1.002
Accuracy of the network on the 10000 test images: 61 %
Learning Rate: 0.0001 Momentum: 0.95
Epoch 1 loss: 0.938
Epoch 1 loss: 0.953
Epoch 2 loss: 0.893
Epoch 2 loss: 0.912
Accuracy of the network on the 10000 test images: 61 %
```

In [50]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```



```
GroundTruth:    cat  ship  ship plane
```

In [51]:
```
outputs = net(images)
```

In [52]:
```
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
Predicted:    dog   car   car  ship
```

In [53]:
```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print("Accuracy for class {:5s} is: {:.1f} %".format(classname,
                                                         accuracy))
```

```
Accuracy for class plane is: 61.2 %
Accuracy for class car   is: 84.6 %
Accuracy for class bird  is: 54.1 %
Accuracy for class cat   is: 40.9 %
Accuracy for class deer  is: 51.0 %
Accuracy for class dog   is: 48.6 %
Accuracy for class frog  is: 66.6 %
Accuracy for class horse is: 73.7 %
Accuracy for class ship  is: 75.2 %
Accuracy for class truck is: 63.9 %
```

In [10]:
```
# Import necessary packages
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from time import time
```

In [11]:
```
import os
from google.colab import drive
```

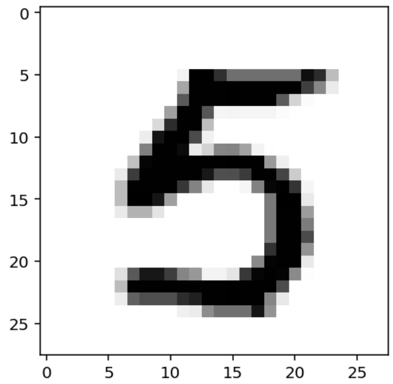
In [12]:
```
### Run this cell

from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((0.5,), (0.5,)),
                              ])

# Download and load the training data
trainset = datasets.MNIST('drive/My Drive/mnist/MNIST_data/', download=True, train=True, transform=transf
valset = datasets.MNIST('drive/My Drive/mnist/MNIST_data/', download=True, train=False, transform=transfo
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz (http://yann.lecun.com/exdb/mni
st/train-images-idx3-ubyte.gz)
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz (http://yann.lecun.com/exdb/mni
st/train-images-idx3-ubyte.gz) to drive/My Drive/mnist/MNIST_data/MNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting drive/My Drive/mnist/MNIST_data/MNIST/raw/train-images-idx3-ubyte.gz to drive/My Drive/mnis
t/MNIST_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz (http://yann.lecun.com/exdb/mni
st/train-labels-idx1-ubyte.gz)
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz (http://yann.lecun.com/exdb/mni
st/train-labels-idx1-ubyte.gz) to drive/My Drive/mnist/MNIST_data/MNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting drive/My Drive/mnist/MNIST_data/MNIST/raw/train-labels-idx1-ubyte.gz to drive/My Drive/mnis
t/MNIST_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz (http://yann.lecun.com/exdb/mnis
t/t10k-images-idx3-ubyte.gz)
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz (http://yann.lecun.com/exdb/mnis
t/t10k-images-idx3-ubyte.gz) to drive/My Drive/mnist/MNIST_data/MNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting drive/My Drive/mnist/MNIST_data/MNIST/raw/t10k-images-idx3-ubyte.gz to drive/My Drive/mnist/
MNIST_data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz (http://yann.lecun.com/exdb/mnis
t/t10k-labels-idx1-ubyte.gz)
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz (http://yann.lecun.com/exdb/mnis
t/t10k-labels-idx1-ubyte.gz) to drive/My Drive/mnist/MNIST_data/MNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/4542 [00:00<?, ?it/s]

Extracting drive/My Drive/mnist/MNIST_data/MNIST/raw/t10k-labels-idx1-ubyte.gz to drive/My Drive/mnist/
MNIST_data/MNIST/raw
```

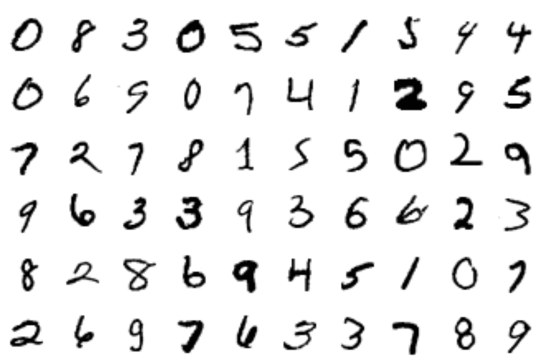
In [13]:
```
dataiter = iter(trainloader)
images, labels = dataiter.next()
print(type(images))
print(images.shape)
print(labels.shape)
```

```
<class 'torch.Tensor'>
torch.Size([64, 1, 28, 28])
torch.Size([64])
```

In [14]:
```
plt.imshow(images[0].numpy().squeeze(), cmap='gray_r');
```



In [15]:
```
figure = plt.figure()
num_of_images = 60
for index in range(1, num_of_images + 1):
    plt.subplot(6, 10, index)
    plt.axis('off')
    plt.imshow(images[index].numpy().squeeze(), cmap='gray_r')
```



In [16]:
```
from torch import nn

# Layer details for the neural network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

# Build a feed-forward network
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.LogSoftmax(dim=1))
print(model)
```

```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax(dim=1)
)
```

In [17]:
```

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
model.to(device)
```

```
cuda
```

Out[17]:
```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax(dim=1)
)
```

In [18]:
```
criterion = nn.NLLLoss()
images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1)

logps = model(images.cuda())
loss = criterion(logps, labels.cuda())
```

In [19]:
```
print('Before backward pass: \n', model[0].weight.grad)
loss.backward()
print('After backward pass: \n', model[0].weight.grad)
```

```
Before backward pass: 
 None
After backward pass: 
 tensor([[-2.2657e-03, -2.2657e-03, -2.2657e-03,  ..., -2.2657e-03,
         -2.2657e-03, -2.2657e-03],
        [-2.1346e-03, -2.1346e-03, -2.1346e-03,  ..., -2.1346e-03,
         -2.1346e-03, -2.1346e-03],
        [ 3.6523e-05,  3.6523e-05,  3.6523e-05,  ...,  3.6523e-05,
          3.6523e-05,  3.6523e-05],
        ...,
        [ 3.3214e-03,  3.3214e-03,  3.3214e-03,  ...,  3.3214e-03,
          3.3214e-03,  3.3214e-03],
        [ 6.1648e-04,  6.1648e-04,  6.1648e-04,  ...,  6.1648e-04,
          6.1648e-04,  6.1648e-04],
        [ 3.4574e-03,  3.4574e-03,  3.4574e-03,  ...,  3.4574e-03,
          3.4574e-03,  3.4574e-03]], device='cuda:0')
```

In [20]:
```
# Optimizers require the parameters to optimize and a learning rate
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

In [22]:
```

print('Initial weights - ', model[0].weight)

images, labels = next(iter(trainloader))
images.resize_(64, 784)

# Clear the gradients, do this because gradients are accumulated
optimizer.zero_grad()

# Forward pass, then backward pass, then update weights
output = model(images.cuda())
loss = criterion(output, labels.cuda())
loss.backward()
print('Gradient -', model[0].weight.grad)
```

```
Initial weights -  Parameter containing:
tensor([[-0.0273, -0.0145, -0.0089,  ..., -0.0027, -0.0260,  0.0335],
        [ 0.0090, -0.0083, -0.0304,  ..., -0.0202, -0.0314, -0.0164],
        [ 0.0097,  0.0004,  0.0194,  ..., -0.0197,  0.0193, -0.0249],
        ...,
        [-0.0089, -0.0201,  0.0191,  ...,  0.0270,  0.0075,  0.0266],
        [-0.0050, -0.0084, -0.0184,  ...,  0.0147,  0.0211,  0.0279],
        [-0.0343,  0.0088, -0.0013,  ..., -0.0325, -0.0139, -0.0025]],
       device='cuda:0', requires_grad=True)
Gradient - tensor([[-0.0022, -0.0022, -0.0022,  ..., -0.0022, -0.0022, -0.0022],
        [-0.0005, -0.0005, -0.0005,  ..., -0.0005, -0.0005, -0.0005],
        [ 0.0006,  0.0006,  0.0006,  ...,  0.0006,  0.0006,  0.0006],
        ...,
        [-0.0017, -0.0017, -0.0017,  ..., -0.0017, -0.0017, -0.0017],
        [ 0.0012,  0.0012,  0.0012,  ...,  0.0012,  0.0012,  0.0012],
        [ 0.0013,  0.0013,  0.0013,  ...,  0.0013,  0.0013,  0.0013]],
       device='cuda:0')
```

In [23]:
```
# Take an update step and few the new weights
optimizer.step()
print('Updated weights - ', model[0].weight)
```

```
Updated weights -  Parameter containing:
tensor([[-0.0273, -0.0145, -0.0089,  ..., -0.0027, -0.0260,  0.0335],
        [ 0.0090, -0.0083, -0.0304,  ..., -0.0202, -0.0314, -0.0164],
        [ 0.0097,  0.0004,  0.0194,  ..., -0.0197,  0.0193, -0.0249],
        ...,
        [-0.0089, -0.0201,  0.0191,  ...,  0.0270,  0.0075,  0.0266],
        [-0.0050, -0.0084, -0.0184,  ...,  0.0147,  0.0211,  0.0279],
        [-0.0343,  0.0088, -0.0014,  ..., -0.0325, -0.0139, -0.0025]],
       device='cuda:0', requires_grad=True)
```

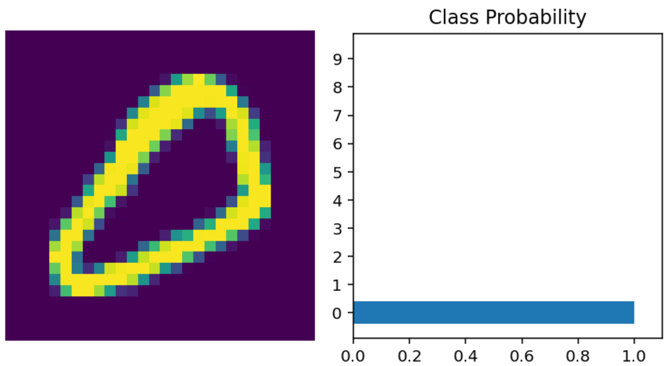
In [24]:
```
optimizer = optim.SGD(model.parameters(), lr=0.003, momentum=0.9)
time0 = time()
epochs = 15
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # Training pass
        optimizer.zero_grad()

        output = model(images.cuda())
        loss = criterion(output, labels.cuda())

        #This is where the model learns by backpropagating
        loss.backward()

        #And optimizes its weights here
        optimizer.step()

        running_loss += loss.item()
    else:
        print("Epoch {} - Training loss: {}".format(e, running_loss/len(trainloader)))
print("\nTraining Time (in minutes) =",(time()-time0)/60)
```

```
Epoch 0 - Training loss: 0.6425788553793039
Epoch 1 - Training loss: 0.282724671327928
Epoch 2 - Training loss: 0.2230572267008552
Epoch 3 - Training loss: 0.18020208306840932
Epoch 4 - Training loss: 0.15242658956234517
Epoch 5 - Training loss: 0.12949277689930663
Epoch 6 - Training loss: 0.11337902554686168
Epoch 7 - Training loss: 0.10130069432045415
Epoch 8 - Training loss: 0.09172150385834134
Epoch 9 - Training loss: 0.08139755662931784
Epoch 10 - Training loss: 0.075294104856012
Epoch 11 - Training loss: 0.06842541152428684
Epoch 12 - Training loss: 0.06307056332344631
Epoch 13 - Training loss: 0.05893389996327857
Epoch 14 - Training loss: 0.054264334936701714

Training Time (in minutes) = 4.007928570111592
```

In [25]:
```
def view_classify(img, ps):
    ''' Function for viewing an image and it's predicted classes.
    '''
    ps = ps.cpu().data.numpy().squeeze()

    fig, (ax1, ax2) = plt.subplots(figsize=(6,9), ncols=2)
    ax1.imshow(img.resize_(1, 28, 28).numpy().squeeze())
    ax1.axis('off')
    ax2.barh(np.arange(10), ps)
    ax2.set_aspect(0.1)
    ax2.set_yticks(np.arange(10))
    ax2.set_yticklabels(np.arange(10))
    ax2.set_title('Class Probability')
    ax2.set_xlim(0, 1.1)
    plt.tight_layout()
```

In [26]:
```
images, labels = next(iter(valloader))

img = images[0].view(1, 784)
# Turn off gradients to speed up this part
with torch.no_grad():
    logps = model(img.cuda())

# Output of the network are log-probabilities, need to take exponential for probabilities
ps = torch.exp(logps)
probab = list(ps.cpu().numpy()[0])
print("Predicted Digit =", probab.index(max(probab)))
view_classify(img.view(1, 28, 28), ps)
```

```
Predicted Digit = 0
```



In [27]:
```
correct_count, all_count = 0, 0
for images,labels in valloader:
  for i in range(len(labels)):
    img = images[i].view(1, 784)
    # Turn off gradients to speed up this part
    with torch.no_grad():
        logps = model(img.cuda())

    # Output of the network are log-probabilities, need to take exponential for probabilities
    ps = torch.exp(logps)
    probab = list(ps.cpu().numpy()[0])
    pred_label = probab.index(max(probab))
    true_label = labels.numpy()[i]
    if(true_label == pred_label):
      correct_count += 1
    all_count += 1

print("Number Of Images Tested =", all_count)
print("\nModel Accuracy =", (correct_count/all_count))
```

```
Number Of Images Tested = 10000

Model Accuracy = 0.9708
```