Data Generation

In [5]:
```
%matplotlib inline
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt

dim_theta = 10
data_num = 1000
scale = .1

theta_true = np.ones((dim_theta,1))
print('True theta:', theta_true.reshape(-1))

A = np.random.uniform(low=-1.0, high=1.0, size=(data_num,dim_theta))
y_data = A @ theta_true + np.random.normal(loc=0.0, scale=scale, size=(data_num, 1))

A_test = np.random.uniform(low=-1.0, high=1.0, size=(50, dim_theta))
y_test = A_test @ theta_true + np.random.normal(loc=0.0, scale=scale, size=(50, 1))
```

```
True theta: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Solving for the exact mean squared loss (solving Ax = b)

In [7]:
```
'''
Hints:
1. See the least squares solution to Ax = b (when it is covered in lecture).

2. Use Numpy functions like Numpy's linear algebra functions to solve for x in Ax = b.
In fact, the linear algebra module is already imported with ```import numpy.linalg as la```.

3. Use the defined variable A in Ax = b. Use y_data as b. Use theta_pred as x.
'''
theta_pred = la.inv(A.T @ A) @ (A.T @ y_data)

print('Empirical theta', theta_pred.reshape(-1))
```

```
Empirical theta [0.99218671 0.99592763 0.9993016  0.99963533 1.00368735 1.00186409
 1.00490434 0.99861362 0.99307294 0.99822247]
```

SGD Variants Noisy Function

In [25]:
```
batch_size = 1
max_iter = 1000
lr = 0.001
theta_init = np.random.random((10,1)) * 0.1
```

In [94]:
```
def noisy_val_grad(theta_hat, data_, label_, deg_=2.):
    gradient = np.zeros_like(theta_hat)
    loss = 0

    for i in range(data_.shape[0]):
        x_ = data_[i, :].reshape(-1,1)
        y_ = label_[i, 0]
        err = np.sum(x_ * theta_hat) - y_

        '''
        Hints:
        1. Find the gradient and loss for each data point x_.
        2. For grad, you need err, deg_, and x_.
        3. For l, you need err and deg_ only.
        4. Checkout the writeup for more hints.
        '''
        grad = deg_ * (np.abs(err) ** (deg_ - 1)) * np.sign(err) * x_
        l = np.abs(err) ** deg_

        loss += l / data_.shape[0]
        gradient += grad / data_.shape[0]

    return loss, gradient
```

Running SGD Variants

In [110…
```
#@title Parameters
```

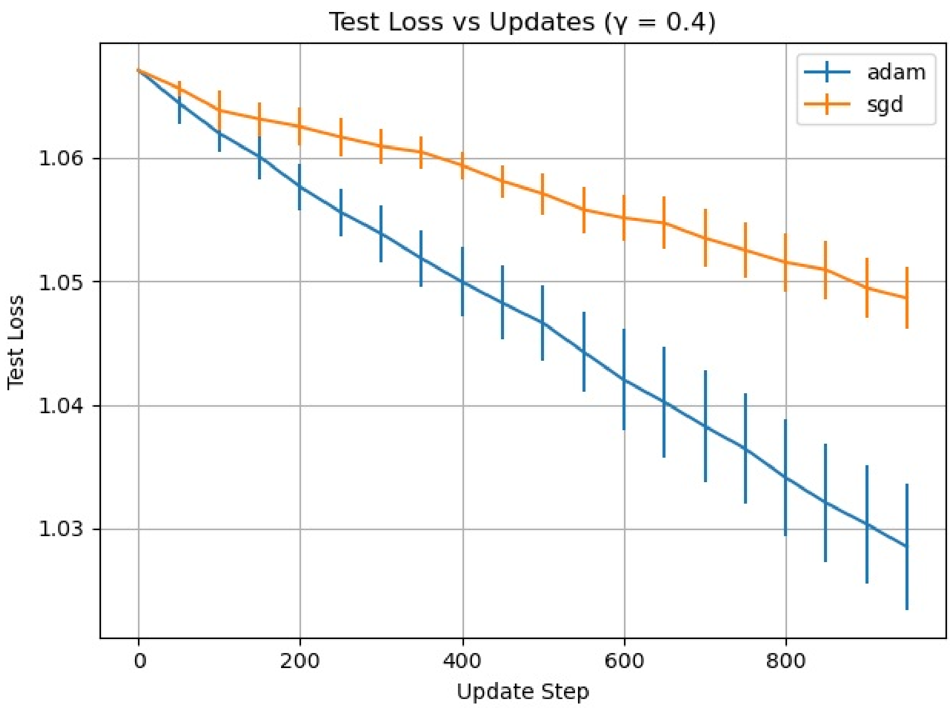
```python
deg_ = 2. #@param {type: "number"}
num_rep = 10 #@param {type: "integer"}
max_iter = 1000 #@param {type: "integer"}
fig, ax = plt.subplots(figsize=(10,10))
best_vals = {}
test_exp_interval = 50 #@param {type: "integer"}
grad_artificial_normal_noise_scale = 0. #@param {type: "number"}
```

In [113…

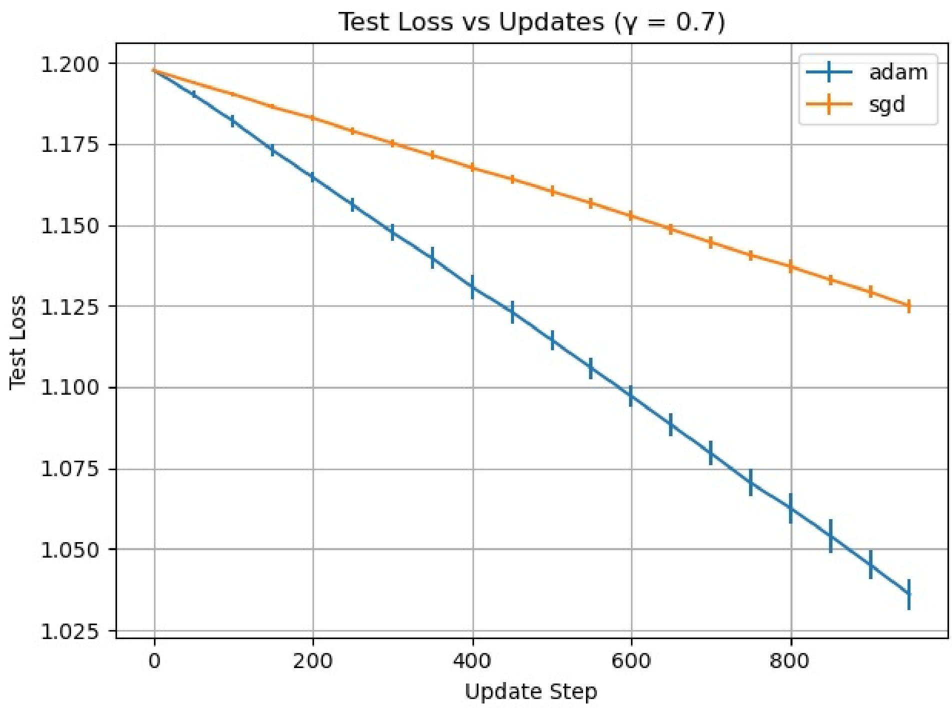
```python
for gamma_val in [0.4, 0.7, 1, 2, 3, 5]:
    deg_ = gamma_val
    print(f'Running experiments for γ = {deg_}')
    for method_idx, method in enumerate(['adam', 'sgd']):
        test_loss_mat = []
        train_loss_mat = []

        for replicate in range(num_rep):
            if replicate % 20 == 0:
                print(method, replicate)

            if method == 'adam':
                beta_1 = 0.9
                beta_2 = 0.999
                epsilon = 1e-8
                m = np.zeros_like(theta_init)
                v = np.zeros_like(theta_init)

            if method == 'adagrad':
                print('Adagrad Not implemented.')
                epsilon = NotImplemented # TODO: Initialize parameters
                squared_sum = NotImplemented

            theta_hat = theta_init.copy()
            test_loss_list = []
            train_loss_list = []

            for t in range(max_iter):
                idx = np.random.choice(data_num, batch_size) # Split data
                train_loss, gradient = noisy_val_grad(theta_hat, A[idx,:], y_data[idx,:], deg_=deg_)
                artificial_grad_noise = np.random.randn(10, 1) * grad_artificial_normal_noise_scale + np.sign(n
                gradient = gradient + artificial_grad_noise
                train_loss_list.append(train_loss)

                if t % test_exp_interval == 0:
                    test_loss, _ = noisy_val_grad(theta_hat, A_test[:,:], y_test[:,:], deg_=deg_)
                    test_loss_list.append(test_loss)

                if method == 'adam':
                    m = beta_1 * m + (1 - beta_1) * gradient
                    v = beta_2 * v + (1 - beta_2) * (gradient ** 2)

                    m_hat = m / (1 - beta_1**(t + 1))
                    v_hat = v / (1 - beta_2**(t + 1))

                    theta_hat = theta_hat - lr * m_hat / (np.sqrt(v_hat) + epsilon)


                elif method == 'adagrad':
                    print('Adagrad Not implemented.')
                    squared_sum = squared_sum + NotImplemented # TODO: Implement Adagrad
                    theta_hat = theta_hat - lr * NotImplemented

                elif method == 'sgd':
                    theta_hat = theta_hat - lr * gradient

            test_loss_mat.append(test_loss_list)
            train_loss_mat.append(train_loss_list)

        print(method, 'done')
        x_axis = np.arange(max_iter)[::test_exp_interval]

        print('test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in 
        print('The elements of test_loss_np are the test loss values computed in each replicate and training st
        test_loss_np = np.array(test_loss_mat)

        '''
        Hints:
        1. Use test_loss_np in np.mean() with axis = 0
        '''
        test_loss_mean = np.mean(test_loss_np, axis=0)

        '''
        Hints:
        1. Use test_loss_np in np.std() with axis = 0
```

```
        2. Divide by np.sqrt() using num_rep as a parameter
        '''
        test_loss_se = np.std(test_loss_np, axis=0) / np.sqrt(num_rep)

        plt.errorbar(x_axis, test_loss_mean, yerr=2.5*test_loss_se, label=method)
        best_vals[method] = min(test_loss_mean)
        print(f'Final parameters for method "{method}":\n{theta_hat.reshape(-1)}\n')

    plt.xlabel("Update Step")
    plt.ylabel("Test Loss")
    plt.title(f'Test Loss vs Updates (γ = {deg_})')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

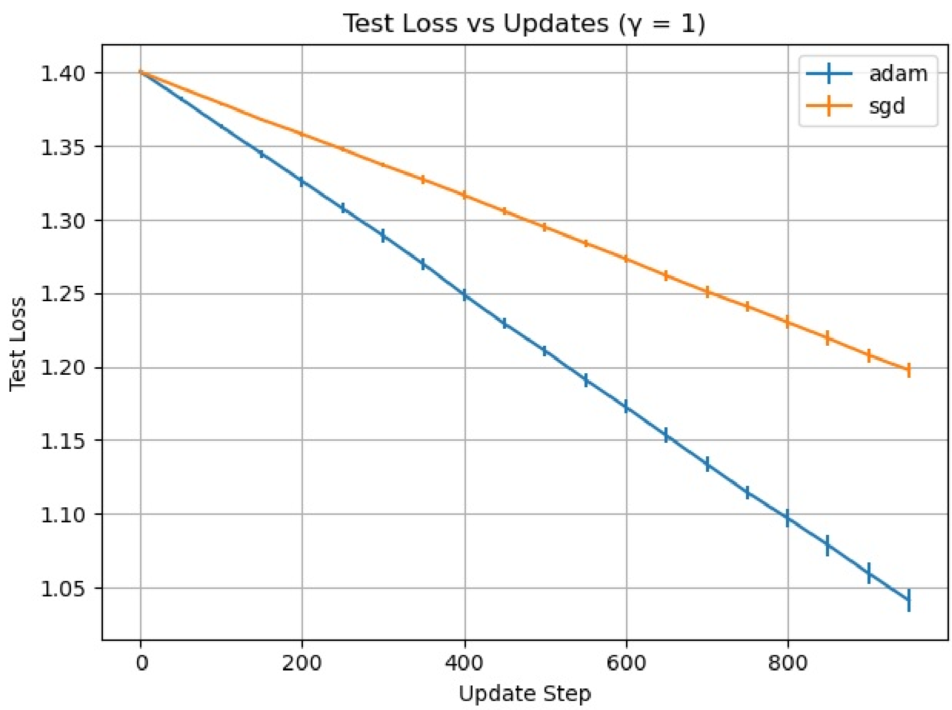
```
Running experiments for γ = 0.4
adam 0
adam done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "adam":
[0.10675621 0.13306561 0.22496032 0.10094938 0.18651719 0.14231369
 0.14785469 0.08840414 0.30632364 0.13131128]

sgd 0
sgd done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "sgd":
[0.04865604 0.12794786 0.17395863 0.07027743 0.14997399 0.03025955
 0.14304976 0.03519987 0.18897177 0.02844775]
```



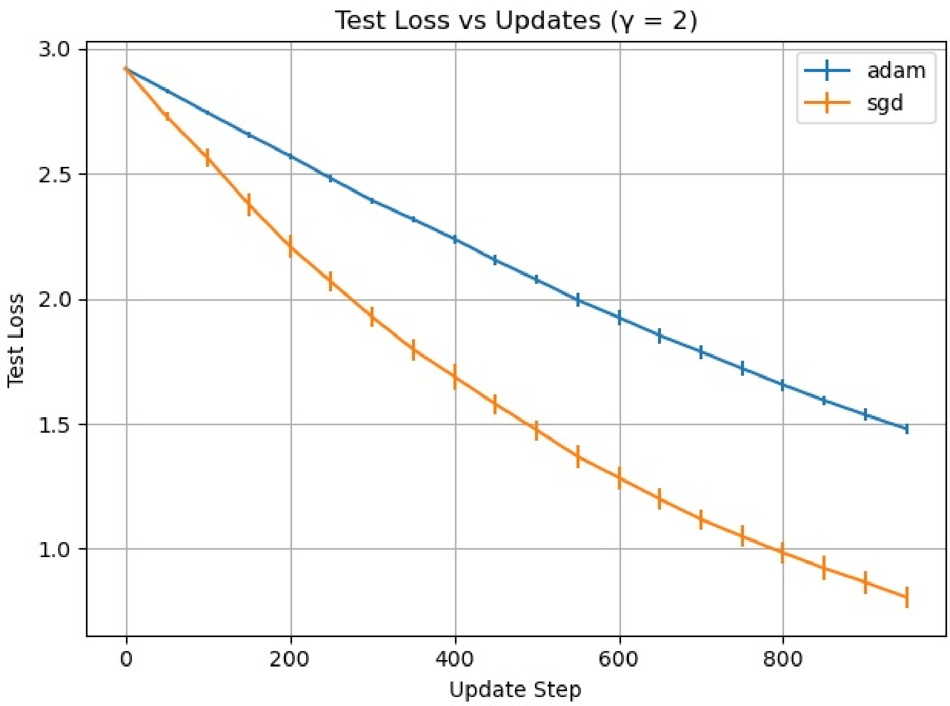

```
Running experiments for γ = 0.7
adam 0
adam done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "adam":
[0.21138336 0.25611736 0.33999842 0.18436075 0.27973139 0.2775759
 0.26417712 0.2610319  0.29572431 0.25085275]

sgd 0
sgd done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "sgd":
[0.12270384 0.16269365 0.19887691 0.10335078 0.14877666 0.13488393
 0.12730259 0.12570428 0.16097615 0.18247377]
```



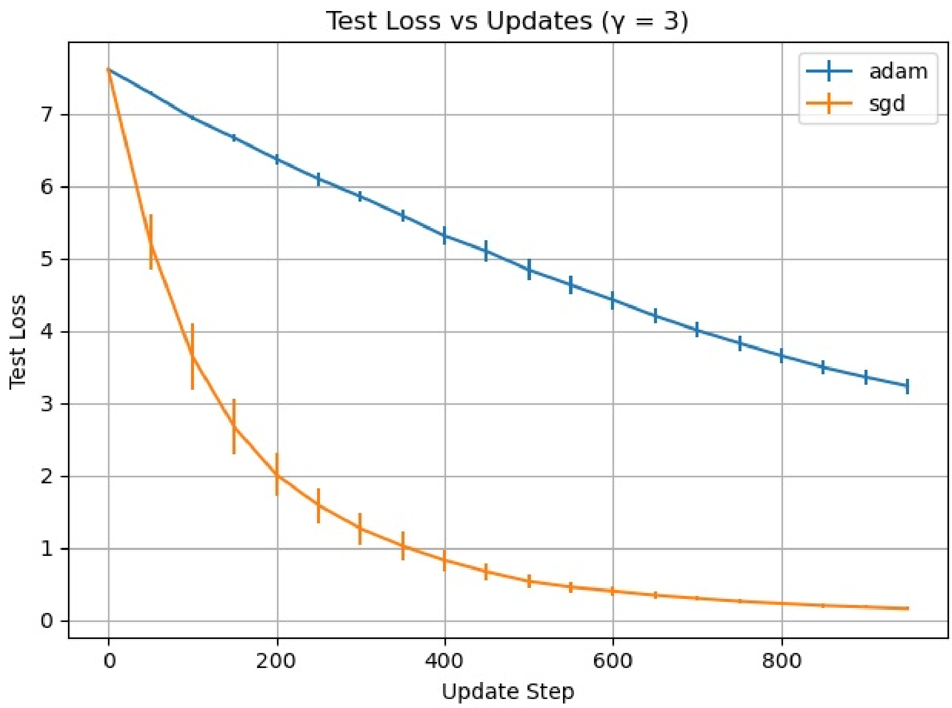

```
Running experiments for γ = 1
adam 0
adam done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "adam":
[0.27632435 0.30497584 0.38177757 0.27352123 0.339567   0.29960823
 0.33396765 0.3480048  0.35326908 0.26729357]

sgd 0
sgd done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "sgd":
[0.16909786 0.17034048 0.26476039 0.13554004 0.22585654 0.2036784
 0.20631278 0.19936605 0.29641202 0.20573266]
```



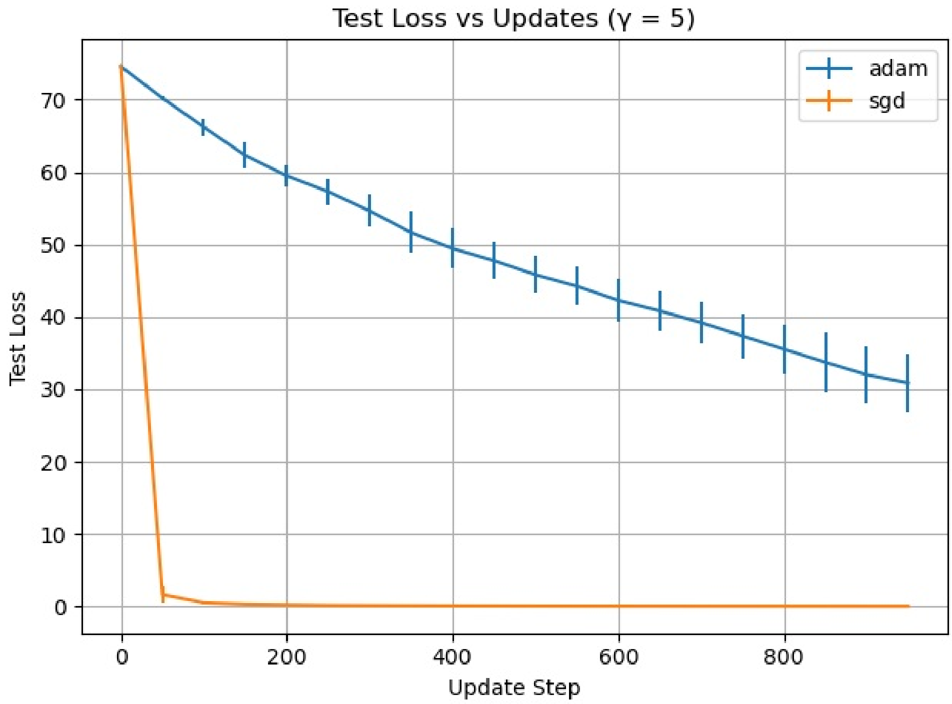
```
Running experiments for γ = 2
adam 0
adam done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training

The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "adam":
[0.27417634 0.34360829 0.40276927 0.28214233 0.40483774 0.36351695
 0.34171235 0.32246308 0.38679239 0.36323049]

sgd 0
sgd done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "sgd":
[0.57508718 0.59779443 0.60955804 0.43432368 0.55146614 0.48872879
 0.59873052 0.52675437 0.54217826 0.54655387]
```




```
Running experiments for γ = 3
adam 0
adam done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "adam":
[0.25624687 0.31877116 0.34075927 0.23898033 0.34050586 0.29671767
 0.25507838 0.28910636 0.33825251 0.31997232]

sgd 0
sgd done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "sgd":
[0.69075776 0.77598346 0.74513576 0.70377153 0.80302679 0.74187181
 0.65412221 0.70737926 0.76404636 0.79229806]
```




```
Running experiments for γ = 5
adam 0
adam done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "adam":
[0.20628413 0.26385046 0.27084552 0.14417176 0.27692224 0.22025362
 0.22579816 0.22747377 0.25572015 0.26423421]

sgd 0
sgd done
test_loss_np is a 2d array with num_rep rows and each column denotes a specific update stage in training
The elements of test_loss_np are the test loss values computed in each replicate and training stage.
Final parameters for method "sgd":
[0.95223817 1.07675152 0.87585526 0.63918599 0.97985196 1.09275952
 1.11038456 0.94355939 0.55686714 1.03764575]
```



In [115…

```
best_vals = { k: int(v * 1000) / 1000. for k,v in best_vals.items() } # A weird way to round numbers
plt.title(f'Test Loss \n(objective degree: {deg_},  best values: {best_vals})')
plt.ylabel('Test Loss')
plt.legend()
plt.xlabel('Updates')
```

```
C:\Users\jeffr\AppData\Local\Temp\ipykernel_9016\2927815680.py:4: UserWarning: No artists with labels found to p
ut in legend.  Note that artists whose label start with an underscore are ignored when legend() is called with n
o argument.
  plt.legend()
```

Out[115…

```
Text(0.5, 0, 'Updates')
```

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Imports for Python libraries

In [74]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from time import time
from torchvision import datasets, transforms
from torch import nn
from torch import optim
```

Set up the mini-batch size

In [75]:
```
#@title Batch Size
mini_batch_size = 64 #@param {type: "integer"}
```

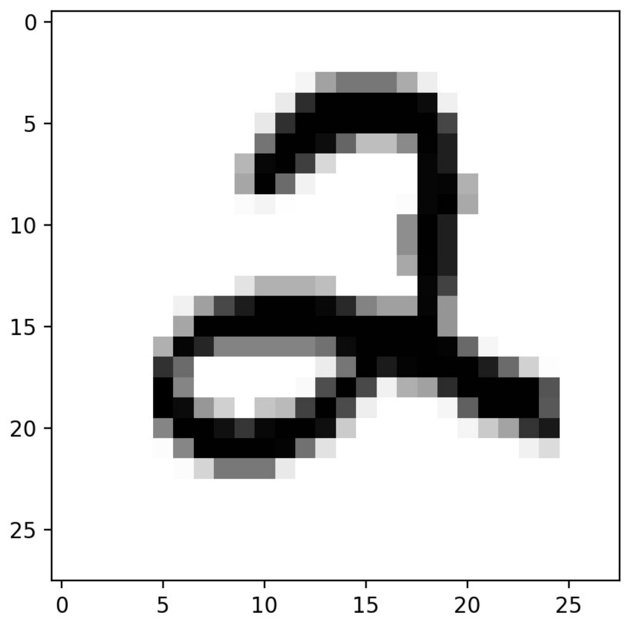
Download the dataset, pre-process, and divide into mini-batches

In [76]:
```
### 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('MNIST_data/', download=True, train=True, transform=transform)
valset = datasets.MNIST('MNIST_data/', download=True, train=False, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=mini_batch_size, shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=mini_batch_size, shuffle=True)
dataiter = iter(trainloader)
images, labels = next(dataiter)
print(type(images))
print(images.shape)
print(labels.shape)
```

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

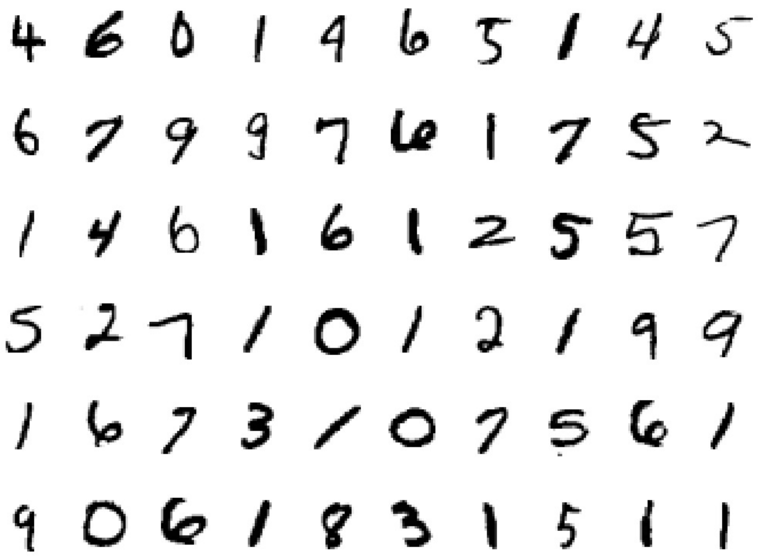
Explore the processed data

In [77]:
```
plt.imshow(images[0].numpy().squeeze(), cmap='gray_r'); # Change the index of images[] to get different numbers
```



In [78]:
```
figure = plt.figure()
num_of_images = 60
for index in range(1, num_of_images + 1):
    plt.subplot(6, 10, index)
```

```python
    plt.axis('off')
    plt.imshow(images[index].numpy().squeeze(), cmap='gray_r')
```



Set up the neural network

In [79]:
```python
# Please change the runtime to GPU if you'd like to have some speed-up on Colab
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

### 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]), # Fully Connected Layer
    nn.ReLU(), # Activation
    nn.Linear(hidden_sizes[0], hidden_sizes[1]), # Fully Connected Layer
    nn.ReLU(), # Activation
    nn.Linear(hidden_sizes[1], output_size), # Fully Connected Layer
    nn.LogSoftmax(dim=1) # (Log) Softmax Layer: Output a probability distribution and apply log
)
print(model)
model.to(device)
```

```
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)
)
```

Out[79]:
```
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)
)
```

Set up the optimization model

In [80]:
```python
#@title Optimizer
lr = 0.003 #@param {type: "number"}
optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # Feel free to try out other optimizers as you see f
```

Set up the loss function to optimize over

In [81]:
```python
time0 = time()
epochs = 15
criterion = nn.NLLLoss() # Negative log likelihood loss function is used
images, labels = next(iter(trainloader))
```

```
images = images.view(images.shape[0], -1).to(device)

logps = model(images) # Model spits out the log probability of image belonging to different classes
loss = criterion(logps, labels.to(device))
```

Train the neural network

In [82]:
```
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).to(device)
        labels = labels.to(device)

        # Training pass
        optimizer.zero_grad()

        output = model(images).to(device)
        loss = criterion(output, labels)

        # backpropagation: calculate the gradient of the loss function w.r.t model parameters
        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: 0.38967667361185243
Epoch 1 - Training loss: 0.18968197613207896
Epoch 2 - Training loss: 0.14011508118865618
Epoch 3 - Training loss: 0.11321266755751615
Epoch 4 - Training loss: 0.0965055028971499
Epoch 5 - Training loss: 0.0817778741539732
Epoch 6 - Training loss: 0.07242025381975821
Epoch 7 - Training loss: 0.06685666090745264
Epoch 8 - Training loss: 0.06066104276803261
Epoch 9 - Training loss: 0.05491418143345127
Epoch 10 - Training loss: 0.05058346516029514
Epoch 11 - Training loss: 0.048102074209526224
Epoch 12 - Training loss: 0.04394178601107927
Epoch 13 - Training loss: 0.04050959063049005
Epoch 14 - Training loss: 0.038537531182747525

Training Time (in minutes) = 3.2236938953399656
```

Evaluate the trained neural network

In [83]:
```
correct_count, all_count = 0, 0
for images, labels in valloader:
    for i in range(len(labels)):
        img = images[i].view(1, 784).to(device)
        labels = labels.to(device)
        # Forward pass only during evaluation
        with torch.no_grad():
            logps = model(img)

        # 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.cpu().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 = 0.9717
```

Predict using the trained neural network

In [84]:

```
def view_classify(img, ps):
    """ Function for viewing an image and it's predicted classes."""
    ps = ps.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 [85]:

```
images, labels = next(iter(valloader))

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

# 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.cpu().view(1, 28, 28), ps.cpu())
```

```
Predicted Digit = 4
```