Final

December 14, 2022

Midterm Questions

1 Which of the following techniques does not help with vanishing gradient problem?

1. ReLU non-linearity
2. Auxiliary losses
3. Skip-connections
4. **Drop-out [1 Mark]**

2 Which one of the following is incorrect?

1. Both convolutional and fully-connected networks are feed-forward.
2. **There is one bias term per channel of a convolution kernel. [1 Mark]**
3. Padding helps convolution to pay attention to border pixels as well.
4. A convolutional layer is a fully-connected layer applied to neighboring patches of an image.

3 Which one of the following is not a regularization technique?

1. Drop-out
2. **Non-linear activation functions [1 Mark]**
3. Adding some noise to gradients
4. Weight decay

4 Suppose the output of a convolutional layer is a feature map of size 64x64 with depth of 128. If we apply global average pooling to this network, what is the output size?

1. 64x64
2. **128x1 [1 Mark]**
3. 64x1
4. 128x128

5 Which one of the following is incorrect about CNNs?

1. The weight sharing occurs across all spatial dimensions.

2. As we go deeper, the feature map resolution is decreased.
3. As we go deeper, the dimension of channels is increased.
4. **None of the above. [1 Mark]**

6 Which one of the following is incorrect about data augmentations?

1. Augmentations preserve the semantic content.
2. Augmentations help with generalization.
3. Augmentations increase the data diversity.
4. **Augmentations can change the class. [1 Mark]**

7 Which of the following is correct?

1. In batch normalization, the summary statistics is computed for each sample independently.
2. Batch normalization is compatible with stochastic gradient descent.
3. We need moving averages at inference time for layer normalization.
4. **Layer normalization is a non-parametric method. [1 Mark]**

8 Which one of the following is incorrect about optimizers?

1. Adam optimizer uses a memory at least twice the number of parameters.
2. Adam optimizer is able to handle the ravines in loss landscape.
3. **Optimizers must be gradient-based. [1 Mark]**
4. **SGD with momentum uses a memory at least twice the number of parameters. [1 Mark]**

9 Which one of the following is incorrect?

1. The derivative of an output with respect to its skip connection is 1.
2. Saddle points are more likely than local optima in loss landscape.
3. In a fully-connected network, the layers before the final layer learn linearly-separable embeddings.
4. **Kernels at the final layers of a convolutional network access more local information. [1 Mark]**

10 Which of the following is incorrect?

1. Derivative of the loss function with respect to a weight indicates how much the weight is changing the loss.
2. Derivative of the loss function with respect to a weight indicates if the weight is increasing or decreasing the loss.
3. Derivative of the loss function with respect to an input feature indicates how much the network is paying attention to that feature.
4. **Derivative of the loss function with respect to an input feature can help to train the network faster. [1 Mark]**

11 Suppose we want to design an image search engine where given a query image we retrieve top 20 images from our image dataset ordered by their similarity to the query image. Also suppose that we have access to a pretrained ResNet model. Explain in details how you would implement this search engine.

1. Need to compute the embeddings for the query image, the image database, compute the similarity and sort by similarity.

12 Example: 1-layer ANN with MSE and Gradient Descent

[ ]:
```
# data (first column is the bias term)
x = [[1, 0.1,-0.2],
     [1,-0.1, 0.9],
     [1, 1.2, 0.1],
     [1, 1.1, 1.5]]
# labels (desired output)
t = [0, 0, 0, 1]
# initial weights
w = [1, -1, 1]

iterations = 50
learning = 10

def simple_ann_MSE(x, w, t, iterations, learning):

    E = []

    #iterate over epochs
    for ii in range(iterations):
        err = []
        y = []
        #iterate over all the samples x
        for n in range(len(x)):
            v = 0
            # compute w.x
            for p in range(len(x[0])):
                v = v + x[n][p]*w[p]
            #sigmoidal activation
            y.append(1 / (1 + math.e**(-v)))
            #MSE classification error
            err.append((y[n]-t[n])**2)
            #gradient descent to compute new weights
            for p in range(len(w)):
                d = x[n][p]*(y[n]-t[n])*(1-y[n])*(y[n])
                w[p] = w[p] - learning*d

        #sum up classification error
        E.append(sum(err)/len(x))

    return (y, w, E)

(y, w, E) = simple_ann_MSE(x, w, t, iterations, learning)
```

13 Example: 1-layer ANN with Cross-Entropy and Gradient Descent

[ ]:
```
def simple_ann_CE(x, w, t, iterations, learning):
    E = []
    #iterate over epochs
    for ii in range(iterations):
        err = []
        y = []
        #iterate over all the samples x
        for n in range(len(x)):
            v = 0
            #compute w.x
            for p in range(len(x[0])):
                v = v + x[n][p]*w[p]

            #sigmoidal activation
            y.append(1 / (1 + math.e**(-v)))
            #cross-entropy classification error
            err.append(-t[n]*math.log(y[n]+ 0.000001) - (1-t[n])*math.
 ↪log(1-y[n]+ 0.000001))
            #gradient descent to compute new weights
            for p in range(len(w)):
                d = x[n][p]*(y[n]-t[n]) #cross_entropy
                w[p] = w[p] - learning*d
        #sum up classification error
        E.append(sum(err))

    return (y, w, E)

(y, w, E) = simple_ann_CE(x, w, t, iterations, learning)
```

14 Simple training function

[ ]:
```
def train(model, data, batch_size=64, num_epochs=1 , print_stat = 1):
    train_loader = torch.utils.data.DataLoader(data, batch_size=batch_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    iters, losses, train_acc, val_acc = [], [], [], []

    # training
    n = 0 # the number of iterations
    for epoch in range(num_epochs):
        for imgs, labels in iter(train_loader):
            out = model(imgs)             # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()               # backward pass (compute parameter␣
 ↪updates)
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch

            # save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)             # compute␣
 ↪*average* loss
            train_acc.append(get_accuracy(model, train=True)) # compute␣
 ↪training accuracy
            val_acc.append(get_accuracy(model, train=False))  # compute␣
 ↪validation accuracy
            n += 1
```

15 ANN vs CNN arhitectures for image classification

[ ]:
```
class ANN_MNISTClassifier(nn.Module):
    def __init__(self):
        super(ANN_MNISTClassifier, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 50)
        self.fc2 = nn.Linear(50, 20)
        self.fc3 = nn.Linear(20, 10)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = F.relu(self.fc1(flattened))
        activation2 = F.relu(self.fc2(activation1))
        output = self.fc3(activation2)
        return output

    print('Artificial Neural Network Architecture (aka MLP) Done')

#Convolutional Neural Network Architecture
class CNN_MNISTClassifier(nn.Module):
    def __init__(self):
        super(CNN_MNISTClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 5, 5) #in_channels, out_chanels, kernel_size
        self.pool = nn.MaxPool2d(2, 2) #kernel_size, stride
        self.conv2 = nn.Conv2d(5, 10, 5) #in_channels, out_chanels, kernel_size
        self.fc1 = nn.Linear(160, 32)
        self.fc2 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 160)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    print('Convolutional Neural Network Architecture Done')
```

16 The general formula is this if you are interested: $[(W - K + 2P)/S] + 1$.

- W is the input size
- K is the Kernel size
- P is the padding
- S is the stride

==========

- 28×28 (1ch) => conv1 => 24×24 (5ch) — (28-5+1)
- 24×24 (5ch) => pool => 12×12 (5ch)
- 12×12 (5ch) => conv2 => 8×8 (10ch)
- 8×8 (10ch) => pool => 4×4 (10ch)
- 4×4 (10ch) => Flat => $4 \times 4 \times 10 = 160$

17 Transfer learning (AlexNet)

17.1 Applying AlexNet on a Dataset

In order to use transfer learning with AlexNet on a new dataset we will have to keep in mind how AlexNet was trained. AlexNet was trained on images of 3 x 224 x 224 images from the ImageNet dataset. These images are of higher resolution than what we have seen until now and are in colour. Hence, it would take significant effort to apply AlexNet to MNIST data, instead we will use another dataset.

```
# confirm output from AlexNet feature extraction
alexNet = torchvision.models.alexnet(pretrained=True)
features = alexNet.features(images)
features.shape
```

17.2 Data Normalization

Data normalization means to scale the input features of a neural network, so that all features are scaled similarly (similar means and standard deviations). Although data normalization does not directly prevent overfitting, normalizing your data makes the training problem easier.

Data normalization is less of an issues for input data – like images – where all input features have similar interpretations. All features of an image are pixel intensities, all of which are scaled the same way. However, if we were performing prediction of, say, housing prices based on a house's number of bedrooms, square footage, etc., we would want each of the features to be scaled similarly. A scale of mean 0 and standard deviation 1 is one approach. Another approach is to scale each feature so that they are in the range `[0, 1]`.

The PyTorch transform `transforms.ToTensor()` automatically scales each pixel intensity to the range `[0, 1]`. In your lab 2 code, we used the following transform:

This transform subtracts 0.5 from each pixel, and divides the result by 0.5. So, each pixel intensity will be in the range `[-1, 1]`. In general, having both positive and negative input values helps the network trains quickly (because of the way weights are initialized). Sticking with each pixel being in the range `[0, 1]` is usually fine.

17.3 Data Augmentation

While it is often expensive to gather more data, we can often programmatically *generate* more data points from our existing data set. We can make small alterations to our training set to obtain slightly different input data, but that is still valid. Common ways of obtaining new (image) data include:

- Flipping each image horizontally or vertically (won't work for digit recognition, but might for other tasks)
- Shifting each pixel a little to the left or right
- Rotating the images a little
- Adding noise to the image

… or even a combination of the above. For demonstration purposes, let's randomly rotate our digits a little to get new training samples.

17.4 Weight Decay

A more interesting technique that prevents overfitting is the idea of weight decay. The idea is to **penalize large weights**. We avoid large weights, because large weights mean that the prediction relies a lot on the content of one pixel, or on one unit. Intuitively, it does not make sense that the classification of an image should depend heavily on the content of one pixel, or even a few pixels.

Mathematically, we penalize large weights by adding an extra term to the loss function, the term can look like the following:

- L^1 regularization: $\sum_k |w_k|$
  - Mathematically, this term encourages weights to be exactly 0
- L^2 regularization: $\sum_k w_k^2$
  - Mathematically, in each iteration the weight is pushed towards 0
- Combination of L^1 and L^2 regularization: add a term $\sum_k |w_k| + w_k^2$ to the loss function.

In PyTorch, weight decay can also be done automatically inside an optimizer. The parameter `weight_decay` of `optim.SGD` and most other optimizers uses L^2 regularization for weight decay. The value of the `weight_decay` parameter is another tunable hyperparameter.

17.5 Dropout

Yet another way to prevent overfitting is to build **many** models, then average their predictions at test time. Each model might have a different set of initial weights.

We won't show an example of model averaging here. Instead, we will show another idea that sounds drastically different on the surface.

This idea is called **dropout**: we will randomly "drop out", "zero out", or "remove" a portion of neurons from each training iteration.

In different iterations of training, we will drop out a different set of neurons.

The technique has an effect of preventing weights from being overly dependent on each other: for example for one weight to be unnecessarily large to compensate for another unnecessarily large weight with the opposite sign. Weights are encouraged to be "more independent" of one another.

During test time though, we will not drop out any neurons; instead we will use the entire set of weights. This means that our training time and test time behaviour of dropout layers are *different*. In the code for the function `train` and `get_accuracy`, we use `model.train()` and `model.eval()` to flag whether we want the model's training behaviour, or test time behaviour.

While unintuitive, using all connections is a form of model averaging! We are effectively averaging over many different networks of various connectivity structures.

[ ]:
```
class MNISTClassifierWithDropout(nn.Module):
    def __init__(self):
        super(MNISTClassifierWithDropout, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 50)
        self.layer2 = nn.Linear(50, 20)
        self.layer3 = nn.Linear(20, 10)
        self.dropout1 = nn.Dropout(0.4) # drop out layer with 20% dropped out␣
 ↪neuron
        self.dropout2 = nn.Dropout(0.4)
        self.dropout3 = nn.Dropout(0.4)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = F.relu(self.layer1(self.dropout1(flattened)))
        activation2 = F.relu(self.layer2(self.dropout2(activation1)))
        output = self.layer3(self.dropout3(activation2))
        return output

model = MNISTClassifierWithDropout()
train(model, mnist_train, mnist_val, num_iters=500)
```

18 Autoencoder

[ ]:
```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        encoding_dim = 32
        # encoder
        self.fc1 = nn.Linear(28 * 28, encoding_dim)
        # decoder
        self.fc2 = nn.Linear(encoding_dim, 28*28)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        x = F.relu(self.fc1(flattened))
        # sigmoid for scaling output from 0 to 1
        x = F.sigmoid(self.fc2(x))
        return x

def train(model, num_epochs=5, batch_size=64, learning_rate=1e-3):
    torch.manual_seed(42)
    criterion = nn.MSELoss() # mean square error loss
    optimizer = torch.optim.Adam(model.parameters(),
                                 lr=learning_rate,
                                 weight_decay=1e-5) # <--
    train_loader = torch.utils.data.DataLoader(mnist_data,
                                               batch_size=batch_size,
                                               shuffle=True)
    outputs = []
    for epoch in range(num_epochs):
        for data in train_loader:
            img, _ = data
            recon = model(img)
            img = img.view(-1, 28 * 28)
            loss = criterion(recon, img)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        print('Epoch:{}, Loss:{:.4f}'.format(epoch+1, float(loss)))
        outputs.append((epoch, img, recon),)
    return outputs
```

19 Convolutional Autoencoder

When working with image data it is often better to use a convolutional neural network and take advantage of the spatial relationships. The architecture for the encoder stage of a convolutional autoencoder will consist of standard convolutional layers that we have seen in our previous architectures. The decoder step will be a bit more tricky since we need a way to increase the resolution.

We need something akin to convolution, but that goes in the *opposite* direction. We will use something called a **transpose convolution**. Transpose convolutions were first called *deconvolutions*, since it is the "inverse'' of a convolution operation. However, the terminology was confusing since it has nothing to do with the mathematical notion of deconvolution.

19.1 Implementation of a Convolutional Autoencoder

To demonstrate the use of convolution transpose operations, we will build a **convolutional autoencoder**. Below is an example of a *convolutional* autoencoder that uses solely convolutional layers:

[ ]:
```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential( # like the Composition layer you built
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7))

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1,␣
 ↪output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid())

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

def train(model, num_epochs=5, batch_size=64, learning_rate=1e-3):
    torch.manual_seed(42)
    criterion = nn.MSELoss() # mean square error loss
    optimizer = torch.optim.Adam(model.parameters(),
                                 lr=learning_rate,
                                 weight_decay=1e-5)
```

```
    train_loader = torch.utils.data.DataLoader(mnist_data,
                                               batch_size=batch_size,
                                               shuffle=True)
    outputs = []
    for epoch in range(num_epochs):
        for data in train_loader:
            img, _ = data
            recon = model(img)
            loss = criterion(recon, img)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        print('Epoch:{}, Loss:{:.4f}'.format(epoch+1, float(loss)))
        outputs.append((epoch, img, recon),)
    return outputs
```

19.2 Variational Autoencoder

To allow us to sample from the embedding space and generate new images, we add a constraint on the encoding network that forces it to generate latent vectors that roughly follow a unit Gaussian distribution. This constraint is what separates a variational autoencoder from the ones we've seen up until now.

Now generating new images requires that we sample a latent vector from the unit Gaussian and pass it into the decoder.

As shown in the figure, we will have encoding and decoding networks similar to what we used before, whether with fully-connected or convolutional layers. Then we add two additional linear layers to hold the mean and standard deviation vectors of the embedding space. We will need some way to generate a sampled latent space which will act as input to the decoding network.

We will also need to update our loss function to use Kullback-Leibler divergence to constrain the embedding space to follow a unit Gaussian distribution. You will not be required to know the math behind this.

A demonstration of the variational autoencoder is provided below.

[ ]:
```
# dimensions of latent space
zdim = 25

# Variational Autoencoder
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()

        # encoder
        self.fc1 = nn.Linear(28 * 28, 350)
        self.relu = nn.ReLU()
        self.fc2m = nn.Linear(350, zdim)  # mu layer
        self.fc2s = nn.Linear(350, zdim)  # sd layer

        # decoder
        self.fc3 = nn.Linear(zdim, 350)
        self.fc4 = nn.Linear(350, 28 * 28)
        self.sigmoid = nn.Sigmoid()

    def encode(self, x):
        h1 = self.relu(self.fc1(x))
        return self.fc2m(h1), self.fc2s(h1)

    # reparameterize
    def reparameterize(self, mu, logvar):
        if self.training:
            std = logvar.mul(0.5).exp_()
            eps = std.data.new(std.size()).normal_()
            return eps.mul(std).add_(mu)
        else:
            return mu

    def decode(self, z):
        h3 = self.relu(self.fc3(z))
        return self.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 28 * 28))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```

20 Training Variational Autoencoder

[ ]:
```
# loss function for VAE are unique and use Kullback-Leibler
# divergence measure to force distribution to match unit Gaussian
def loss_function(recon_x, x, mu, logvar):
    bce = F.binary_cross_entropy(recon_x, x.view(-1, 28 * 28))
    kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    kld /= batch_size * 28 * 28
    return bce + kld

def train(model, num_epochs = 1, batch_size = 64, learning_rate = 1e-3):
    model.train() #train mode so that we do reparameterization
    torch.manual_seed(42)

    train_loader = torch.utils.data.DataLoader(datasets.MNIST('data',
               train=True, download=True, transform=transforms.ToTensor()),
               batch_size = batch_size, shuffle = True)

    optimizer = optim.Adam(model.parameters(), learning_rate)

    for epoch in range(num_epochs):
      for data in train_loader:  # load batch
          img, _ = data

          recon, mu, logvar = model(img)
          loss = loss_function(recon, img, mu, logvar) # calculate loss
          loss.backward()
          optimizer.step()
          optimizer.zero_grad()

      print('Epoch:{}, Loss:{:.4f}'.format(epoch+1, float(loss)))
```

21 Recurrent Neural Networks

21.1 word2vec models

A word2vec model learns embedding of words using the following architecture:

- **Encoder**: word -> embedding
- **Decoder**: embedding -> nearby words (context)

Specific word2vec models differ in the which "nearby words" is predicted using the decoder: is it the 3 context words that appeared *before* the input word? Is it the 3 words that appeared *after*? Or is it a combination of the two words that appeared before and two words that appeared after the input word?

These models are trained using a large corpus of text: for example the whole of Wikipedia or a large collection of news articles. We won't train our own word2vec models in this course, so we won't talk about the many considerations involved in training a word2vec model.

Instead, we will use a set of pre-trained word embeddings. These are embeddings that someone else took the time and computational power to train. One of the most commonly-used pre-trained word embeddings are the **GloVe embeddings**.

GloVe is a variation of a word2vec model. Again, the specifics of the algorithm and its training will be beyond the scope of this course. You should think of **GloVe embeddings** similarly to pre-trained AlexNet weights in that they "may" provide improvements to the representation of data.

Unlike AlexNet, there are several variations of GloVe embeddings. They differ in the corpus used to train the embedding, and the *size* of the embeddings.

21.2 GloVe Embeddings

To load pre-trained GloVe embeddings, we'll use a package called `torchtext`. The package `torchtext` contains other useful tools for working with text that we will see later in the course.

We'll begin by loading a set of GloVe embeddings. The first time you run the code below, Python will download a large file (862MB) containing the pre-trained embeddings.

It is a torch tensor with dimension `(50,)`. It is difficult to determine what each number in this embedding means, if anything. However, we know that there is structure in this embedding space. That is, distances in this embedding space is meaningful.

21.3 Measuring Distance

To explore the structure of the embedding space, it is necessary to introduce a notion of *distance*. You are probably already familiar with the notion of the **Euclidean distance**. The Euclidean distance of two vectors $x = [x_1, x_2, ...x_n]$ and $y = [y_1, y_2, ...y_n]$ is just the 2-norm of their difference $x - y$. We can compute the Euclidean distance between x and y: $\sqrt{\sum_i (x_i - y_i)^2}$

An alternative measure of distance is the **Cosine Similarity**. The cosine similarity measures the *angle* between two vectors, and has the property that it only considers the *direction* of the vectors, not their magnitudes.

21.4 Analogies

One surprising aspect of GloVe vectors is that the *directions* in the embedding space can be meaningful. The structure of the GloVe vectors certain analogy-like relationship like this tend to hold:

$$king - man + woman \approx queen$$

The

$$doctor - man + woman \approx nurse$$

analogy is very concerning. Just to verify, the same result does not appear if we flip the gender terms:

21.5 Sentiment analysis

The columns we care about is the first one and the last one. The first column is the label (the label `0` means "sad" tweet, `4` means "happy" tweet), and the last column contains the tweet. Our task is to predict the sentiment of the tweet given the text.

The approach today is as follows, for each tweet:

1. We will split the text into words. We will do so by splitting at all whitespace characters. There are better ways to perform the split, but let's keep our dependencies light.
2. We will look up the GloVe embedding of each word. Words that do not have a GloVe vector will be ignored.
3. We will sum up all the embeddings to get an embedding for an entire tweet.
4. Finally, we will use a fully-connected neural network to predict whether the tweet has positive or negative sentiment.

First, let's sanity check that there are enough words for us to work with.

```
[ ]: mymodel = nn.Sequential(nn.Linear(50, 30),
                             nn.ReLU(),
                             nn.Linear(30, 10),
                             nn.ReLU(),
                             nn.Linear(10, 2))
train_network(mymodel, train_loader, valid_loader, num_epochs=100,␣
 ↪learning_rate=1e-4)

def train_network(model, train_loader, valid_loader, num_epochs=5,␣
 ↪learning_rate=1e-5):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    losses, train_acc, valid_acc = [], [], []
    epochs = []
    for epoch in range(num_epochs):
        for tweets, labels in train_loader:
            optimizer.zero_grad()
            pred = model(tweets)
            loss = criterion(pred, labels)
            loss.backward()
            optimizer.step()

        losses.append(float(loss))
        if epoch % 5 == 4:
            epochs.append(epoch)
            train_acc.append(get_accuracy(model, train_loader))
            valid_acc.append(get_accuracy(model, valid_loader))
            print("Epoch %d; Loss %f; Train Acc %f; Val Acc %f" % (
                epoch+1, loss, train_acc[-1], valid_acc[-1]))

def get_accuracy(model, data_loader):
    correct, total = 0, 0
    for tweets, labels in data_loader:
        output = model(tweets)
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += labels.shape[0]
    return correct / total
```

21.6 Recurrent Neural Network Module

PyTorch has variations of recurrent neural network modules. These modules computes the following:

$$hidden = updatefn(hidden, input)$$

$$output = outputfn(hidden)$$

These modules are more complex and less intuitive than the usual neural network layers, so let's take a look:

[ ]:
```
rnn_layer = nn.RNN(input_size=50,    # dimension of the input repr
                   hidden_size=50,   # dimension of the hidden units
                   batch_first=True) # input format is [batch_size, seq_len,␣
 ↪repr_dim]
```

Now, let's try and run this untrained `rnn_layer` on `tweet_emb`. We will need to add an extra dimension to `tweet_emb` to account for batching. We will also need to initialize a set of hidden units of size `[batch_size, 1, repr_dim]`, to be used for the *first* set of computations.

[ ]:
```
tweet_input = tweet_emb.unsqueeze(0) # add the batch_size dimension
h0 = torch.zeros(1, 1, 50)           # initial hidden state
out, last_hidden = rnn_layer(tweet_input, h0)
```

We don't technically have to explictly provide the initial hidden state, if we want to use an initial state of zeros. Just for today, we will be explicit about the hidden states that we provide. out2, last_hidden2 = rnn_layer(tweet_input) Now, let's look at the output and hidden dimensions that we have:

[ ]:
```
print(out.shape)
print(last_hidden.shape)
# torch.Size([1, 31, 50])
# torch.Size([1, 1, 50])
```

21.7 RNN Model

[ ]:
```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Look up the embedding
        x = self.emb(x)
        # Set an initial hidden state
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, _ = self.rnn(x, h0)
        # Pass the output of the last time step to the classifier
        out = self.fc(out[:, -1, :])
        return out

model = TweetRNN(50, 50, 2)
```

So, we will need a different way of batching.

One strategy is to **pad shorter sequences with zero inputs**, so that every sequence is the same length. The following PyTorch utilities are helpful.

- `torch.nn.utils.rnn.pad_sequence`
- `torch.nn.utils.rnn.pad_packed_sequence`
- `torch.nn.utils.rnn.pack_sequence`
- `torch.nn.utils.rnn.pack_padded_sequence`

(Actually, there are more powerful helpers in the `torchtext` module that we will use in Lab 5. We'll stick to these in this demo, so that you can see what's actually going on under the hood.)

[ ]:
```
from torch.nn.utils.rnn import pad_sequence

tweet_padded = pad_sequence([tweet for tweet, label in train[:10]],
                            batch_first=True)
print(tweet_padded.shape)
print(tweet_padded[0:2])
```

21.8 TweetBatcher

One issue we overlooked was that in our `TweetRNN` model, we always take the **last output unit** as input to the final classifier. Now that we are padding the input sequences, we should really be using the output at a previous time step. Recurrent neural networks therefore require much more record keeping than ANNs or even CNNs.

There is yet another problem: the longest tweet has many, many more words than the shortest. Padding tweets so that every tweet has the same length as the longest tweet is impractical. Padding tweets in a mini-batch, however, is much more reasonable.

In practice, practitioners will batch together tweets with the same length. For simplicity, we will do the same. We will implement a (more or less) straightforward way to batch tweets.

[ ]:
```
import random

class TweetBatcher:
    def __init__(self, tweets, batch_size=32, drop_last=False):
        # store tweets by length
        self.tweets_by_length = {}
        for words, label in tweets:
            # compute the length of the tweet
            wlen = words.shape[0]
            # put the tweet in the correct key inside self.tweet_by_length
            if wlen not in self.tweets_by_length:
                self.tweets_by_length[wlen] = []
            self.tweets_by_length[wlen].append((words, label),)

        #  create a DataLoader for each set of tweets of the same length
        self.loaders = {wlen : torch.utils.data.DataLoader(
                                    tweets,
                                    batch_size=batch_size,
                                    shuffle=True,
                                    drop_last=drop_last) # omit last batch if␣
 ↪smaller than batch_size
            for wlen, tweets in self.tweets_by_length.items()}

    def __iter__(self): # called by Python to create an iterator
        # make an iterator for every tweet length
        iters = [iter(loader) for loader in self.loaders.values()]
        while iters:
            # pick an iterator (a length)
            im = random.choice(iters)
            try:
                yield next(im)
            except StopIteration:
                # no more elements in the iterator, remove it
                iters.remove(im)
```

21.9 TweetLSTM

[ ]:
```
class TweetLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(TweetLSTM, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Look up the embedding
        x = self.emb(x)
        # Set an initial hidden state and cell state
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the LSTM
        out, _ = self.rnn(x, (h0, c0))
        # Pass the output of the last time step to the classifier
        out = self.fc(out[:, -1, :])
        return out

model = TweetLSTM(50, 50, 2)
train_rnn_network(model, train_loader, valid_loader, num_epochs=20,␣
 ↪learning_rate=2e-5)
get_accuracy(model, test_loader)
```

22 Generative Recurrent Neural Networks (GNN)

Last time we discussed using recurrent neural networks to make predictions about sequences. In particular, we treated tweets as a **sequence** of words. Since tweets can have a variable number of words, we needed an architecture that can take variable-sized sequences as input.

This time, we will use recurrent neural networks to **generate** sequences. Generating sequences is more involved compared to making predictions about sequences. However, it is a very interesting task, and many students chose sequence-generation tasks for their projects.

1. We need to be able to generate the *next* token, given the current hidden state. In practice, we get a probability distribution over the next token, and sample from that probability distribution.
2. We need to be able to update the hidden state somehow. To do so, we need two pieces of information: the old hidden state, and the actual token that was generated in the previous step. The actual token generated will inform the subsequent tokens.

We will repeat both functions until a special "END OF SEQUENCE" token is generated.

Note that there are several tricky things that we will have to figure out. For example, how do we actually sample the actual token from the probability distribution over tokens? What would we do during training, and how might that be different from during testing/evaluation? We will answer those questions as we implement the RNN.

First, we will need to encode this tweet using a one-hot encoding. We'll build dictionary mappings from the character to the index of that character (a unique integer identifier), and from the index to the character. We'll use the same naming scheme that `torchtext` uses (`stoi` and `itos`).

For simplicity, we'll work with a limited vocabulary containing just the characters in `tweet[100]`, plus two special tokens:

- `<EOS>` represents "End of String", which we'll append to the end of our tweet. Since tweets are variable-length, this is a way for the RNN to signal that the entire sequence has been generated.
- `<BOS>` represents "Beginning of String", which we'll prepend to the beginning of our tweet. This is the first token that we will feed into the RNN.

The way we use these special tokens will become more clear as we build the model.

```
vocab = list(set(tweet)) + ["<BOS>", "<EOS>"]
vocab_stoi = {s: i for i, s in enumerate(vocab)}
vocab_itos = {i: s for i, s in enumerate(vocab)}
vocab_size = len(vocab)
```

22.1 TextGenerator

[ ]:
```
class TextGenerator(nn.Module):
    def __init__(self, vocab_size, hidden_size, n_layers=1):
        super(TextGenerator, self).__init__()

        # identiy matrix for generating one-hot vectors
        self.ident = torch.eye(vocab_size)

        # recurrent neural network
        self.rnn = nn.GRU(vocab_size, hidden_size, n_layers, batch_first=True)

        # a fully-connect layer that outputs a distribution over
        # the next token, given the RNN output
        self.decoder = nn.Linear(hidden_size, vocab_size)

    def forward(self, inp, hidden=None):
        inp = self.ident[inp]                   # generate one-hot vectors of␣
 ↪input
        output, hidden = self.rnn(inp, hidden) # get the next output and hidden␣
 ↪state
        output = self.decoder(output)           # predict distribution over next␣
 ↪tokens
        return output, hidden

model = TextGenerator(vocab_size, 64)
```

22.2 Training with Teacher Forcing

At a very high level, we want our RNN model to have a high probability of generating the tweet. An RNN model generates text one character at a time based on the hidden state value. At each time step, we will check whether the model generated the correct character. That is, at each time step, we are trying to select the correct next character out of all the characters in our vocabulary. Recall that this problem is a multi-class classification problem, and we can use Cross-Entropy loss to train our network to become better at this type of problem.

```
criterion = nn.CrossEntropyLoss()
```

However, we don't just have a single multi-class classification problem. Instead, we have **one classification problem per time-step** (per token)! So, how do we predict the first token in the sequence? How do we predict the second token in the sequence? To help you understand what happens durign RNN training, we'll start with inefficient training code that shows you what happens step-by-step. We'll start with computing the loss for the first token generated, then the second token, and so on. So, let's start with the first classification problem: the problem of generating the **first** token (`tweet[0]`). To generate the first token, we'll feed the RNN network (with an initial, empty hidden state) the "" token. Then, the output

```
bos_input = torch.Tensor([vocab_stoi["<BOS>"]])
print(bos_input.shape, type(bos_input))
bos_input = bos_input.long()
print(bos_input.shape, type(bos_input))
bos_input = bos_input.unsqueeze(0)
print(bos_input.shape, type(bos_input))
output, hidden = model(bos_input, hidden=None)
output # distribution over the first token
```

We can compute the loss using `criterion`. Since the model is untrained, the loss is expected to be high. (For now, we won't do anything with this loss, and omit the backward pass.)

```
target = torch.Tensor([vocab_stoi[tweet[0]]]).long().unsqueeze(0)
criterion(output.reshape(-1, vocab_size), # reshape to 2D tensor
          target.reshape(-1))             # reshape to 1D tensor
```

Now, we need to update the hidden state and generate a prediction for the next token. To do so, we need to provide the current token to the RNN. We already said that during test time, we'll need to sample from the predicted probabilty over tokens that the neural network just generated.

Right now, we can do something better: we can **use the ground-truth, actual target token**. This technique is called **teacher-forcing**, and generally speeds up training. The reason is that right now, since our model does not perform well, the predicted probability distribution is pretty far from the ground truth. So, it is very, very difficult for the neural network to get back on track given bad input data.

```
# Use teacher-forcing: we pass in the ground truth `target`,
# rather than using the NN predicted distribution
output, hidden = model(target, hidden)
output # distribution over the second token
```

Finally, with our final token, we should expect to output the "" token, so that our RNN learns when to stop generating characters.

[ ]:
```
output, hidden = model(target, hidden)
target = torch.Tensor([vocab_stoi["<EOS>"]]).long().unsqueeze(0)
loss = criterion(output.reshape(-1, vocab_size), # reshape to 2D tensor
                 target.reshape(-1))             # reshape to 1D tensor
print(i, output, loss)
```

In practice, we don't really need a loop. Recall that in a predictive RNN, the `nn.RNN` module can take an entire sequence as input. We can do the same thing here:

[ ]:
```
tweet_ch = ["<BOS>"] + list(tweet) + ["<EOS>"]
tweet_indices = [vocab_stoi[ch] for ch in tweet_ch]
tweet_tensor = torch.Tensor(tweet_indices).long().unsqueeze(0)

print(tweet_tensor.shape)

output, hidden = model(tweet_tensor[:,:-1]) # <EOS> is never an input token
target = tweet_tensor[:,1:]                 # <BOS> is never a target token
loss = criterion(output.reshape(-1, vocab_size), # reshape to 2D tensor
                 target.reshape(-1))             # reshape to 1D tensor
```

[ ]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
for it in range(500):
    optimizer.zero_grad()
    output, _ = model(tweet_tensor[:,:-1])
    loss = criterion(output.reshape(-1, vocab_size),
                     target.reshape(-1))
    loss.backward()
    optimizer.step()

    if (it+1) % 100 == 0:
        print("[Iter %d] Loss %f" % (it+1, float(loss)))
```

22.3 Generating a Token

At this point, we want to see whether our model is actually learning something. So, we need to talk about how to actually use the RNN model to generate text. If we can generate text, we can make a qualitative asssessment of how well our RNN is performing.

The main difference between training and test-time (generation time) is that we don't have the ground-truth tokens to feed as inputs to the RNN. Instead, we need to actually **sample** a token based on the neural network's prediction distribution.

But how can we sample a token from a distribution?

On one extreme, we can always take the token with the largest probability (argmax). This has been our go-to technique in other classification tasks. However, this idea will fail here. The reason is that in practice, **we want to be able to generate a variety of different sequences from the same model**. An RNN that can only generate a single new Trump Tweet is fairly useless.

In short, we want some randomness. We can do so by using the logit outputs from our model to construct a multinomial distribution over the tokens and then sample a random token from that multinomial distribution.

One natural multinomial distribution we can choose is the distribution we get after applying the softmax on the outputs. However, we will do one more thing: we will add a **temperature** parameter to manipulate the softmax outputs. We can set a **higher temperature** to make the probability of each token **more even** (more random), or a **lower temperature** to assign more probability to the tokens with a higher logit (output). A **higher temperature** means that we will get a more diverse sample, with potentially more mistakes. A **lower temperature** means that we may see repetitions of the same high probability sequence.

[ ]:
```
def sample_sequence(model, max_len=100, temperature=0.8):
    generated_sequence = ""

    inp = torch.Tensor([vocab_stoi["<BOS>"]]).long()
    hidden = None
    for p in range(max_len):
        output, hidden = model(inp.unsqueeze(0), hidden)
        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = int(torch.multinomial(output_dist, 1)[0])
        # Add predicted character to string and use as next input
        predicted_char = vocab_itos[top_i]

        if predicted_char == "<EOS>":
            break
        generated_sequence += predicted_char
        inp = torch.Tensor([top_i]).long()
    return generated_sequence

print(sample_sequence(model, temperature=0.8))
print(sample_sequence(model, temperature=1.0))
```

```
print(sample_sequence(model, temperature=1.5))
print(sample_sequence(model, temperature=2.0))
print(sample_sequence(model, temperature=5.0))

# God Bless the people of Venezuela!
# God Bless the people of Venezuela!
# God Blessthth peoplefof VenezuelalaG
# GolsBless the people of VenezfeuVa
# h
```

22.4 Generative RNN using GPU

Training a generative RNN can be a slow process. Here's a sample GPU implementation to speed up the training. The changes required to enable GPU are provided in the comments below.

[ ]:
```
def sample_sequence_cuda(model, max_len=100, temperature=0.8):
    generated_sequence = ""

    inp = torch.Tensor([vocab_stoi["<BOS>"]]).long().cuda()    # <----- GPU
    hidden = None
    for p in range(max_len):
        output, hidden = model(inp.unsqueeze(0), hidden)
        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp().cpu()
        top_i = int(torch.multinomial(output_dist, 1)[0])
        # Add predicted character to string and use as next input
        predicted_char = vocab_itos[top_i]

        if predicted_char == "<EOS>":
            break
        generated_sequence += predicted_char
        inp = torch.Tensor([top_i]).long().cuda()    # <----- GPU
    return generated_sequence


def train_cuda(model, data, batch_size=1, num_epochs=1, lr=0.001,␣
 ↪print_every=100):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    it = 0
    data_iter = torchtext.legacy.data.BucketIterator(data,
                                                     batch_size=batch_size,
                                                     sort_key=lambda x: len(x.text),
                                                     sort_within_batch=True)
    for e in range(num_epochs):
        # get training set
        avg_loss = 0
```

```
        for (tweet, lengths), label in data_iter:
            target = tweet[:, 1:].cuda()               # <------- GPU
            inp = tweet[:, :-1].cuda()                 # <------- GPU
            # cleanup
            optimizer.zero_grad()
            # forward pass
            output, _ = model(inp)
            loss = criterion(output.reshape(-1, vocab_size), target.reshape(-1))
            # backward pass
            loss.backward()
            optimizer.step()

            avg_loss += loss
            it += 1 # increment iteration count
            if it % print_every == 0:
                print("[Iter %d] Loss %f" % (it+1, float(avg_loss/print_every)))
                print("    " + sample_sequence_cuda(model, 140, 0.8))
                avg_loss = 0

model = TextGenerator(vocab_size, 64)
model = model.cuda()
model.ident = model.ident.cuda()
train_cuda(model, trump_tweets, batch_size=32, num_epochs=1, lr=0.004,␣
 ↪print_every=100)
```

23 Tutorial 7a - Generative Adversarial Networks

Thus far, we have discussed several **generative** models. A generative model learns the *structure* of a set of input data. In doing so, the model learns to *generate* new data that it has never seen before in the training data. The generative models we discussed were:

- an autoencoder
- an RNN used to generate text

A Generative Adversarial Network (GAN) is yet another example of a generative model. To motivate the GAN, let's first discuss the drawbacks of an autoencoder.

[ ]:
```
class Discriminator(nn.Module):

    def __init__(self):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 1)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = x.view(-1, 28*28) # flatten image
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = self.dropout(x)
        out = self.fc4(x)
        return out

class Generator(nn.Module):

    def __init__(self):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(100, 32)
        self.fc2 = nn.Linear(32, 64)
        self.fc3 = nn.Linear(64, 128)
        self.fc4 = nn.Linear(128, 28*28)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x), 0.2)
```

```
        x = self.dropout(x)
        out = F.tanh(self.fc4(x))
        return out

D = Discriminator()
G = Generator()
```

For now, both the Discriminator and Generator are fully-connected networks. One difference between these models and the previous models we've built is that we are using a `nn.LeakyReLU` activation.

Leaky ReLU is a variation of the ReLU activation that lets some information through, even when its input is less than 0. The layer `nn.LeakyReLU(0.2)` performs the computation: `x if x > 0 else 0.2 * x`.

But what loss function should we optimize? Consider the following quantity:

`P(D correctly identifies real image) + P(D correctly identifies image generated by G)`

A good **discriminator** would want to *maximize* the above quanity by altering its parameters.

Likewise, a good **generator** would want to *minimize* the above quanity. Actually, the only term that the generator controls is `P(D correctly identifies image generated by G)` So, the best thing for the generator to do is alter its parameters to generate images that can fool D.

Since we are looking at class probabilities, we will use binary cross entropy loss.

Here is a training loop to train a GAN. For every minibatch of data, we train the discriminator for one iteration, and then we train the generator for one iteration.

For the discriminator, we use the label `0` to represent a **fake** image, and `1` to represent a real image.

[ ]:
```
def train(G, D, lr=0.002, batch_size=64, num_epochs=20):

    rand_size = 100;

    # optimizers for generator and discriminator
    d_optimizer = optim.Adam(D.parameters(), lr)
    g_optimizer = optim.Adam(G.parameters(), lr)

    # define loss function
    criterion = nn.BCEWithLogitsLoss()

    # get the training datasets
    train_data = datasets.MNIST('data', train=True, download=True,␣
 ↪transform=transforms.ToTensor())

    # prepare data loader
    train_loader = torch.utils.data.DataLoader(train_data,␣
 ↪batch_size=batch_size, shuffle=True)
```

```
# keep track of loss and generated, "fake" samples
samples = []
losses = []

# fixed data for testing
sample_size=16
test_noise = np.random.uniform(-1, 1, size=(sample_size, rand_size))
test_noise = torch.from_numpy(test_noise).float()

for epoch in range(num_epochs):
    D.train()
    G.train()

    for batch_i, (real_images, _) in enumerate(train_loader):

        batch_size = real_images.size(0)

        # rescale images to range -1 to 1
        real_images = real_images*2 - 1

        # === Train the Discriminator ===

        d_optimizer.zero_grad()

        # discriminator losses on real images
        D_real = D(real_images)
        labels = torch.ones(batch_size)
        d_real_loss = criterion(D_real.squeeze(), labels)

        # discriminator losses on fake images
        z = np.random.uniform(-1, 1, size=(batch_size, rand_size))
        z = torch.from_numpy(z).float()
        fake_images = G(z)

        D_fake = D(fake_images)
        labels = torch.zeros(batch_size) # fake labels = 0
        d_fake_loss = criterion(D_fake.squeeze(), labels)

        # add up losses and update parameters
        d_loss = d_real_loss + d_fake_loss
        d_loss.backward()
        d_optimizer.step()


        # === Train the Generator ===
        g_optimizer.zero_grad()
```

```
            # generator losses on fake images
            z = np.random.uniform(-1, 1, size=(batch_size, rand_size))
            z = torch.from_numpy(z).float()
            fake_images = G(z)

            D_fake = D(fake_images)
            labels = torch.ones(batch_size) #flipped labels

            # compute loss and update parameters
            g_loss = criterion(D_fake.squeeze(), labels)
            g_loss.backward()
            g_optimizer.step()

        # print loss
        print('Epoch [%d/%d], d_loss: %.4f, g_loss: %.4f, '
              % (epoch + 1, num_epochs, d_loss.item(), g_loss.item()))

        # append discriminator loss and generator loss
        losses.append((d_loss.item(), g_loss.item()))

        # plot images
        G.eval()
        D.eval()
        test_images = G(test_noise)

        plt.figure(figsize=(9, 3))
        for k in range(16):
            plt.subplot(2, 8, k+1)
            plt.imshow(test_images[k,:].data.numpy().reshape(28, 28),␣
↪cmap='Greys')
        plt.show()

    return losses
```

GANs are notoriously difficult to train. One difficulty is that a training curve is no longer as helpful as it was for a supervised learning problem! The generator and discriminator losses tend to bounce up and down, since both the generator and discriminator are changing over time. Tuning hyperparameters is also much more difficult because we don't have the training curve to guide us. Newer GAN models like Wasserstein GAN tries to alleviate some of these issues, but are beyond the scope of this course.

To compound the difficulty of hyperparameter tuning, GANs also take a long time to train. It is tempting to stop training early, but the effects of hyperparameters may not be noticable until later on in training.

You might have noticed in the images generated by our simple GAN that the model seems to only output a small number of digit types. This phenomenon is called **mode collapse**. A generator can optimize `P(D correctly identifies image generated by G)` by learning to generate one type of input (e.g. one digit) really well, and not learning how to generate any other digits at all!

To prevent mode collapse, newer variations of GANs provides the discriminator with a *small set* of either real or fake data, rather than one at a time. A discriminator would therefore be able to use the variety of the generated data as a feature to determine whether the entire small set of data is real or fake.

24 Adversarial Examples

This notebook demonstrates how easy it is to create adversarial examples. Let's start by training some models to classify the digits in the MNIST data set. We'll work with one fully-connected neural network and one convolutional network to show the generality of our approach.

[ ]:
```
class FCNet(nn.Module):
    def __init__(self):
        super(FCNet, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 50)
        self.layer2 = nn.Linear(50, 20)
        self.layer3 = nn.Linear(20, 10)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = F.relu(self.layer1(flattened))
        activation2 = F.relu(self.layer2(activation1))
        output = self.layer3(activation2)
        return output

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 5, 5, padding=2)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(5, 10, 5, padding=2)
        self.fc1 = nn.Linear(10 * 7 * 7, 32)
        self.fc2 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 10 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = x.squeeze(1)
        return x
```

[ ]:
```
def train(model, data, batch_size=64, lr=0.001, num_iters=1000,␣
↪print_every=100):
    train_loader = torch.utils.data.DataLoader(data, batch_size=batch_size)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    total_loss = 0
    n = 0

    while True:
        for imgs, labels in iter(train_loader):
            out = model(imgs)
            loss = criterion(out, labels)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            total_loss += loss.item()
            n += 1

            if n % print_every == 0:
                print("Iter %d. Avg.Loss: %f" % (n, total_loss/print_every))
                total_loss = 0
            if n > num_iters:
                return
```

[ ]:
```
fc_model = FCNet()
train(fc_model, mnist_images, num_iters=1000)
```

24.1 Targetted Adversarial Attack

The purpose of an adversarial attack is to perturb an input (usually an image x) so that a neural network f misclassifies the perturbed image $x + \epsilon$. In a targeted attack, we want the network f to misclassify the perturbed image into a class of our choosing.

Let's begin with this image. We will perturb the image so our model thinks that the image is of the digit 3, when in fact it is of the digit 5.

[ ]:
```
noise = torch.randn(1, 28, 28) * 0.01
noise.requires_grad = True
```

[ ]:
```
optimizer = optim.Adam([noise], lr=0.01, weight_decay=1)
criterion = nn.CrossEntropyLoss()

for i in range(1000):
    adv_image = torch.clamp(image + noise, 0, 1)
    out = model(adv_image.unsqueeze(0))
    loss = criterion(out, torch.Tensor([target_label]).long())
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

[ ]:
```
def create_adversarial_example(model, image, target_label):
    noise = torch.randn(1, 28, 28)
    noise.requires_grad = True

    optimizer = optim.Adam([noise], lr=0.01, weight_decay=1)
    criterion = nn.CrossEntropyLoss()

    for i in range(1000):
        adv_image = torch.clamp(image + noise, 0, 1)
        out = model(adv_image.unsqueeze(0))
        loss = criterion(out, torch.Tensor([target_label]).long())
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```