7 Autoencoding and self-supervision

This chapter covers

- Training without labels
- Autoencoding to project data
- Constraining networks with bottlenecks
- Adding noise to improve performance
- Predicting the next item to make generative models

You now know several approaches to specifying a neural network for classification and regression problems. These are the classic machine learning problems, where for each data point $\boldsymbol{x}$ (e.g., a picture of a fruit), we have an associated answer y (e.g., fresh or rotten). But what if we do not have a label y? Is there any useful way for us to learn? You should recognize this as an *unsupervised* learning scenario.

People are interested in self-supervision because *labels are expensive*. It is often much easier to get lots of data, but knowing *what* each data point is requires a lot of work. Think about a sentiment classification problem where you try to predict if a sentence is conveying a positive notion (e.g., "I love this deep learning book I'm reading.") or a negative one (e.g., "The author of this book is bad at making jokes."). It's not *hard* to read the sentence, make a determination, and save that information. But if you want to build a *good* sentiment classifier, you might want to label hundreds of thousands to millions of sentences. Do you really want to spend days or

weeks labeling so many sentences? If we could somehow learn without needing these labels, it would make our lives much easier.

One strategy for unsupervised learning that has become increasingly common in deep learning is called *self-supervision*. The idea behind self-supervision is that we use a regression or classification loss function ℓ to do the learning, and we predict something about the input data $\boldsymbol{x}$ itself. In these cases, the labels come *implicitly with the data* and allow us to use the same tools we have already learned to use. Coming up with clever ways to get implicit labels is the trick to self-supervision.

Figure 7.1 shows three of the many ways self-supervision can be done: *in-painting*, where you obscure part of the input and then try to predict what was hidden; *image sorting*, where you break the image into multiple parts, shuffle them, and then try to put them back in the right order; and *autoencoding*, where you're given the input image and predict the input. Using self-supervision, we can train models without labels and then use what the model has learned to do data clustering, perform tasks like identifying noisy/bad data, or build useful models with less data (the latter will be demonstrated in chapter 13).

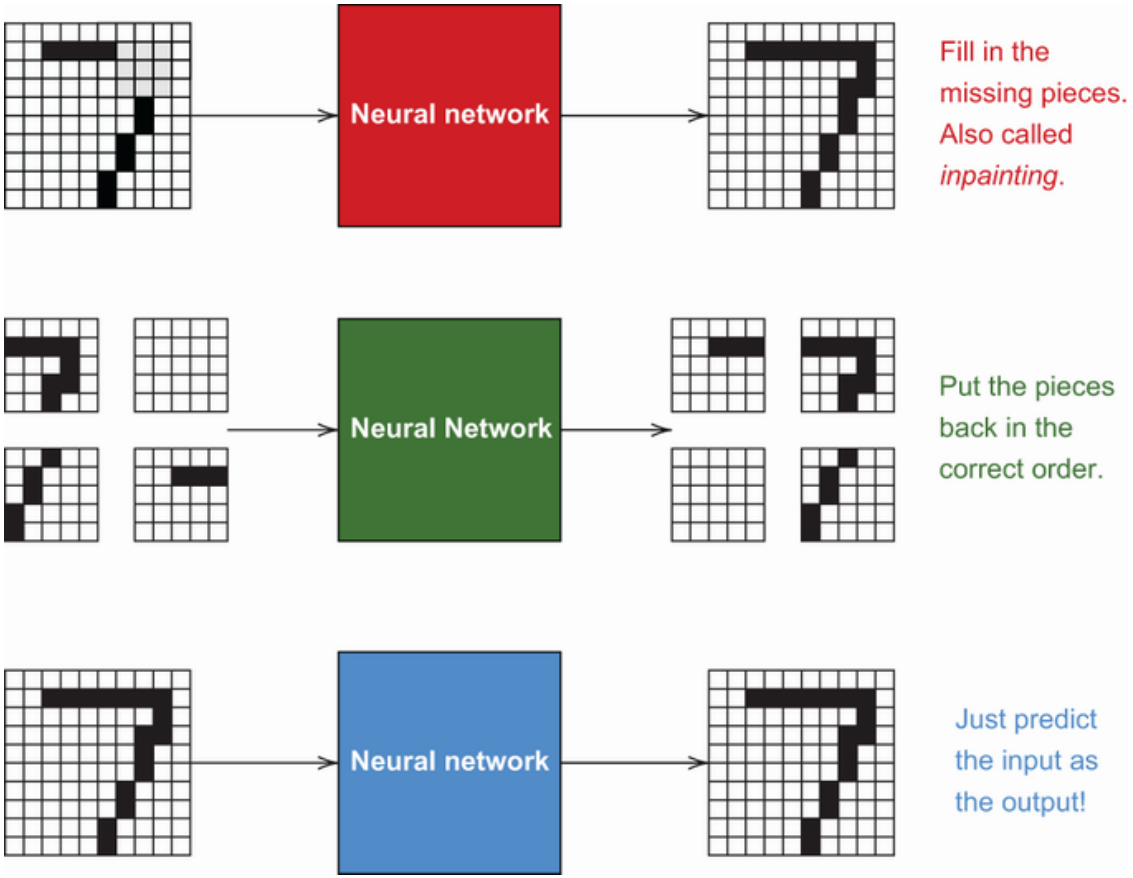


Figure 7.1 Three different types of self-supervised problems: inpainting, image sorting, and autoencoding. In each case, we do not need to know *what* the image is of because the network will try to predict the original image no matter what. In the first case (red), we randomly block out a portion of the image and ask the network to fill in the missing piece. In the second case, we break the image into pieces, shuffle them, and ask the network to put them back in the correct order. In the last case, we simply ask the network to predict the original image from the original image.

There are numerous ways to create a self-supervised problem, and researchers are coming up with new approaches all the time. In this chapter, we focus on one specific kind of self-supervision called *autoencoding*, the third example in figure 7.1, because the key to autoencoding is *predicting the input from the input*. This may seem like an insane idea at first. Surely it is a trivial problem for the network to learn to return the input as it was given. This would be like defining a function as

```
def superUsefulFunction(x):
    return x
```

This `superUsefulFunction` would implement a *perfect* autoencoder. So if the problem is so easy, how can it be useful? That's what we learn in this chapter. The trick is to *constrain* the network, giving it a handicap so that it is unable to learn the trivial solution. Think of it like a test in school, where the teacher has given you 100 questions on an open book exam. If you had all the time in the world, you could simply read the book, find the answers, and write them down. But if you are *constrained* to complete the exam in just an hour, you don't have time to search the book for everything. Instead, you are forced to learn the *underlying concepts* that help you reconstruct the answers to all the questions. The constraints help encourage learning and understanding. The same idea is at work with autoencoders: with the trivial solution off the table, the network is forced to learn something more useful (underlying concepts) to solve the problem.

This chapter explains the concept of autoencoding further and shows that bread-and-butter principal component analysis (PCA) works by secretly being an autoencoder. We'll make small changes to a PyTorch version of PCA to change it into a fully fledged autoencoding neural network. As we make an autoencoding network larger, it becomes more important to constrain it well, which we demonstrate with the *denoising* strategy. Finally we apply these concepts to sequential models like RNNs, which gives the *autoregressive* model.

7.1 How autoencoding works

Let's first describe the idea of autoencoding in a little more detail. Autoencoding means we generally learn *two* functions/networks: first a function $f^{in}(\boldsymbol{x}) = \boldsymbol{z}$, which transforms the input $\boldsymbol{x}$ into a new representation $\boldsymbol{z} \in \mathbb{R}^{D'}$, and then a function $f^{out}(\boldsymbol{z}) = \boldsymbol{x}$, which converts the new representation $\boldsymbol{z}$ back into the original representation. We call these two functions the *encoder* f^{in} and the *decoder* f^{out}, and the process is summarized in figure 7.2. Without any labels, the new representation $\boldsymbol{z}$ is learned in such a way that it captures useful information about the structure of the data in a compact form that is friendly to ML algorithms.

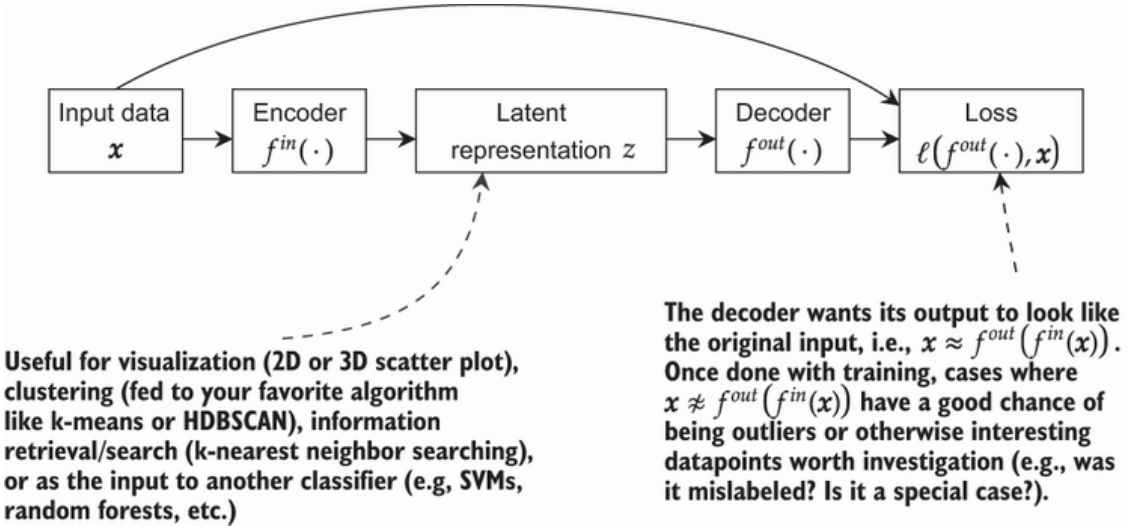


Figure 7.2 The process of autoencoding. The input goes through an encoder that produces a new representation of the data $\boldsymbol{z}$. The representation $\boldsymbol{z}$ can be useful for lots of tasks like clustering, search, and even classification. The decoder attempts to reconstruct the original input $\boldsymbol{x}$ from the encoding $\boldsymbol{z}$.

This assumes that there is a representation $\boldsymbol{z}$ to be found that somehow captures information about the data $\boldsymbol{x}$ but is unspecified. We don't know

what z should look like because we have never observed it. For this reason, we call it a *latent* representation of the data because it is not visible to us but emerges from training the model.[1]

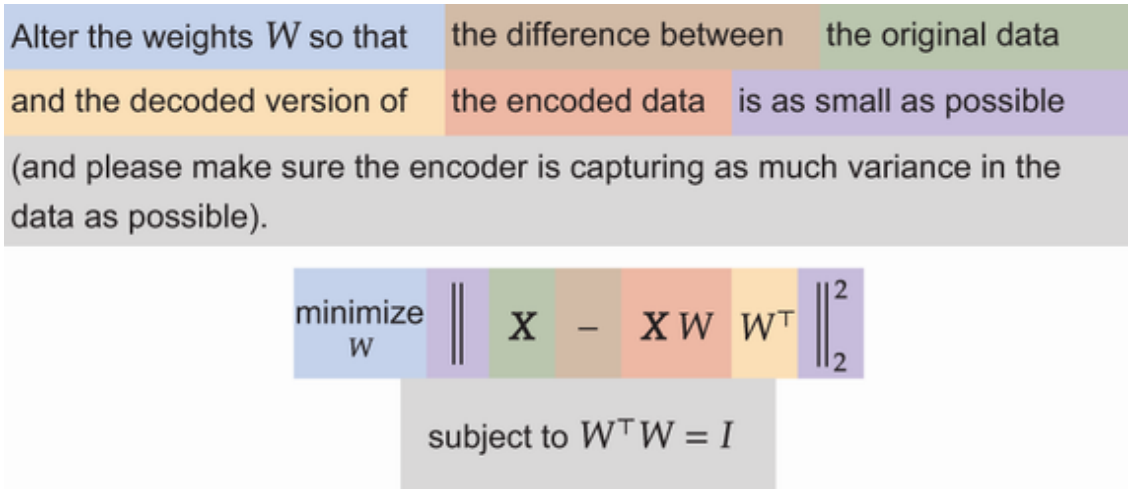
This may seem very silly at first glance. Couldn't we just learn a function f^{in} that does not do anything and returns the input as output? That would trivially solve the problem, but it's like taking out a loan from the bank only to *immediately* pay it back. You have *technically* satisfied all the explicit goals, but you didn't accomplish anything by doing it. This silliness is the dangerous shortcut that we want to avoid. The trick is to set up the problem so the network can't cheat like this. There are a number of different approaches to doing this, and we learn about two of them in this chapter: bottlenecks and denoising. Both of these work by constraining the network so that it is impossible to learn the naive solution.

7.1.1 Principle component analysis is a bottleneck autoencoder

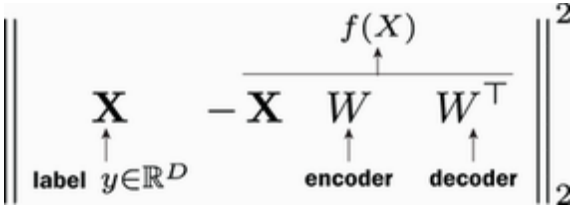
This block of the chapter is a little more challenging to work through, but it's worth it. You'll gain a new perspective on a classic algorithm that will help you further bridge how existing tools that aren't generally thought of as deep learning are actually deeply related.

To exemplify how you can constrain a network so that it can't learn the naive `superUsefulFunction` approach, we first talk about a famous algorithm you probably know about but may not be aware is secretly an autoencoder: the feature engineering and dimensionality reduction technique known as *principle component analysis* (PCA). PCA is used to convert a feature vector in a D-dimensional space down to a lower dimension D', which one might call an *encoder*. PCA also includes a seldom-used

decoder step, where you can convert back (approximately) to the original D dimensional space. The following annotated equation defines the optimization problem that PCA solves:

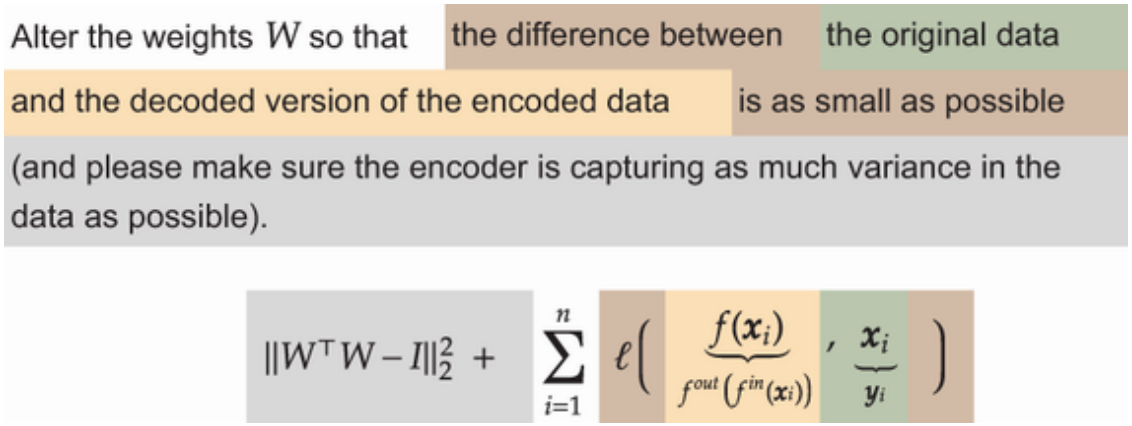


PCA is an important and widely used algorithm, so if you want to use it, you should use one of the existing implementations. But PyTorch is flexible enough for us to implement PCA ourselves. If you look closely at the equation, PCA is a *regression* problem. How? Let's look at the main part of the equation and try to annotate it again with deep learning style equations:



We have one weight matrix W acting as the encoder, and its transpose $W^\top$ is acting as the decoder. That means PCA is using *weight sharing* (remem-

ber that concept from chapters 3 and 4?). The original input is on the left, and we have the 2-norm ($\|\cdot\|_2^2$), which is used by the mean squared error loss. The "subject to" part of the equation is a *constraint* requiring the weight matrix to behave in a particular way. Let's rewrite this equation in the same manner we have been doing for our neural networks:



$\|WW^\top - I\|_2^2$ is a regularization penalty based on the "subject to" constraint. Personally, I think re-expressing PCA as an autoencoder using a loss function helps make it easier to understand it. It's also more explicit that we are using $f(\cdot)$ as a single network that encompasses the sequence of the encoder $f^{in}(\cdot)$ and decoder $f^{out}(\cdot)$.

Now we have written our PCA as a loss function over all our data points. We know PCA works, and if PCA is an autoencoder, then the idea of autoencoding can't be as crazy as it first seems. So how does PCA make it work? The insight that PCA provides is that we make the *intermediate representation too small*. Remember that the first thing PCA does it go from D dimensions down to $D' < D$. Imagine if $D = 1,000,000$ and $D' = 2$. There is *no possible way* to save enough information about 1 million features in

just 2 features to be able to reconstruct the input perfectly. So the best that PCA can do is learn the best 2 features possible, which then forces PCA to learn something useful. This is the primary trick to make autoencoding work: push your data into a smaller representation than you started with.

7.1.2 Implementing PCA

Now that we understand how PCA is an autoencoder, let's work on converting it into PyTorch code. We need to define our network function $f(x)$, given by the following equation. But how do we implement the rightmost $W^\top$ part?

The main trick we need to implement this is to reuse the weight from a `nn.Linear` layer in PyTorch. We walk through implementing PCA in PyTorch here to hammer home the fact that *PCA is an autoencoder*. Once we have implemented PCA, we will make a few changes to convert it into a deep autoencoder, similar to how we moved from linear regression into a neural network in chapter 2. First, let's quickly define some constants for the number of features we are working with, how large our hidden layers should be, and other standard items we want:

```
D = 28*28      ❶
n = 2          ❷
```

```
C = 1          ❸
classes = 10 ❹
```

❶ How many values are in the input? 28 * 28 images. We use this to help determine the size of subsequent layers.

❷ Hidden layer size

❸ How many channels are in the input?

❹ How many classes?

Next, let's implement this missing layer to represent $W^\top$. We call this new layer a *transpose layer*. Why? Because the mathematical operation we are using is called a *transpose*. We also add some logic to have a custom bias term for the weight transposed layer because the input layer has a matrix with shape $W \in \mathbb{R}^{D \times D'}$ and a bias vector $\boldsymbol{b} \in \mathbb{R}^{D'}$. That means $W^\top \in \mathbb{R}^{D' \times D}$, but we can't really take a meaningful transpose of $\boldsymbol{b}$. So if someone wants a bias term, it has to be a new separate one.

Our new `TransposeLinear` module follows. This class implements the `Transpose` operation $W^\top$. The matrix to transpose W must be passed in as the `linearLayer` in the constructor. This way, we can share weights between an original `nn.Linear` layer and this transposed version of that layer.

```
class TransposeLinear(nn.Module):                    ❶
     def __init__(self, linearLayer, bias=True):
```

```
        """
        linearLayer: is the layer that we want to use the transpose of to
        ➥ produce the output of this layer. So the Linear layer represents
        ➥ W, and this layer represents W^T. This is accomplished via
        ➥ weight sharing by reusing the weights of linearLayer
        bias: if True, we will create a new bias term b that is learned
        separately from what is in
        linearLayer. If false, we will not use any bias vector.
        """

        super().__init__()
        self.weight = linearLayer.weight                    ❷

        if bias:
            self.bias = nn.Parameter(torch.Tensor(
            ➥ linearLayer.weight.shape[1]))                ❸


        else:
            self.register_parameter('bias', None)           ❹


    def forward(self, x):                                   ❺
        return F.linear(x, self.weight.t(), self.bias)      ❻
```

❶ Our class extends nn.Module. All PyTorch layers must extend this.

❷ Creates a new variable weight to store a reference to the original weight term

❸ Creates a new bias vector. By default, PyTorch knows how to update Modules and Parameters. Since tensors are neither, the Parameter class

wraps the Tensor class so PyTorch knows that the values in this tensor need to be updated by gradient descent.

❹ The Parameter class can't take None as an input. So if we want the bias term to exist but be potentially unused, we can use the register_parameter function to create it. The important thing here is that PyTorch always sees the same parameters regardless of the arguments for the Module.

❺ The forward function is the code that takes an input and produces an output.

❻ The F directory of PyTorch contains many functions used by Modules. For example, the linear function performs a linear transform when given an input (we use the transpose of our weights) and a bias (if None, it knows to not do anything).

Now that we have our `TransposeLinear` layer completed, we can implement PCA. First is the architecture, which we break into the encoder and decoder portions. Because PyTorch `Module`s are also built from `Module`s, we can define the encoder and decoder as separate parts and use both as components in a final `Module`.

Note that because the input comes in as an image with shape $(\boldsymbol{B},1,28,28)$, and we are using linear layers, we first need to flatten the input into a vector of shape $(\boldsymbol{B},28*28)$. But in the decode step, we want to have the same shape as the original data. We can use the `View` layer I've provided to convert it back. It works just like the `.view` and `.reshape` functions on tensors, except as a `Module`, for convenience:

```
linearLayer = nn.Linear(D, n, bias=False) ❶

pca_encoder =
   nn.Sequential(                          ❷
   nn.Flatten(), linearLayer,
)

pca_decoder = nn.Sequential(               ❸

   TransposeLinear(linearLayer, bias=False),
   View(-1, 1, 28, 28)                     ❹
)
pca_model = nn.Sequential(                 ❺
   pca_encoder,
   pca_decoder
)
```

❶ Since we will share the weights of the linear layer, let's define it separately.

❷ The encoder flattens and then uses the linear layer.

❸ The decoder uses our TransposeLinear layer and the now-shared linearLayer object.

❹ Shapes the data back to its original form

❺ Defines a final PCA model that has the sequence of an encoder followed by a decoder

We have everything we need to train up this autoencoder. But to make it *truly* PCA, we need to add the $WW^\top = I$ constraint. This constraint has a name: *orthogonality*. We won't go into the derivation of *why* PCA has this, but we will include it as a good exercise. We start our model in the right place by giving it an initial *random* set of orthogonal weights using the `nn.init.orthogonal_` function. That just takes one line of code:

```
nn.init.orthogonal_(linearLayer.weight)
```

We aren't going to strictly enforce orthogonality during training because the code to do that would be a little uglier than I want. Instead, we take a common and simple approach to *encourage* orthogonality but not *require* it.[2] This is done by converting the equality $W^TW = I$ into a *penalty* or *regularizer* $\|WW^\top - I\|_2^2$. This works because if the penalty is 0, then W is orthogonal; and if the penalty is nonzero, it will increase the loss, and thus gradient decent will try to make W more orthogonal.

It is not hard to implement this. We are using the mean square error (MSE) loss function $\ell_{MSE}(f(\boldsymbol{x}),\boldsymbol{x})$ to train the self-supervision part. We can just augment this loss function with the loss over the penalty:

$$\underbrace{\ell_{MSE}(f(\boldsymbol{x}),\boldsymbol{x})}_{\text{Please learn to be a good autoencoder}} + \underbrace{\ell_{MSE}(WW^\top,\boldsymbol{I})}_{\text{and try to keep your weights orthogonal}}$$

The following block of code does this. As an additional step, we decrease the strength of the regularizer by a factor of 0.1 to reinforce that the au-

toencoding portion is more important than the orthogonality portion:

```
mse_loss = nn.MSELoss()                                                    ❶

def mseWithOrthoLoss(x, y):                                                ❷
    W = linearLayer.weight                                                 ❸
    I = torch.eye(W.shape[0]).to(device)                                   ❹


    normal_loss = mse_loss(x, y)                                           ❺

    regularization_loss = 0.1*mse_loss(torch.mm(W, W.t()), I)              ❻

    return normal_loss + regularization_loss                               ❼
```

❶ Original loss function

❷ Our PCA loss function

❸ Grabs W from the linearLayer object we saved earlier

❹ The identity matrix that is the target for regularization

❺ Computes the original loss $\ell_{MSE}(f(\boldsymbol{x}),\boldsymbol{x})$

❻ Computes the regularizer penalty $\ell_{MSE}(W^\top W,\boldsymbol{I})$

❼ Returns the sum of the two losses

7.1.3 Implementing PCA with PyTorch

Now we want to create a wrapper for the MNIST datasets. Why? Because the default MNIST dataset will return data in pairs (x,y) for the input and label, respectively. But in our case, the *input is the label* because we are trying to predict the output from the input. So we extend the PyTorch `Dataset` class to take the original tuple x, y and instead return a tuple x, x. This way, our code keeps the convention that the first item in the tuple is the input, and the second item is the desired output/label:

```
class AutoEncodeDataset(Dataset):
    """Takes a dataset with (x, y) label pairs and converts it to (x, x) pairs.
    This makes it easy to reuse other code"""

    def __init__(self, dataset):
        self.dataset = dataset

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):
        x, y = self.dataset.__getitem__(idx)
        return x, x                                    ❶
```

❶ Throws away the original label

NOTE If you were implementing an autoencoder for a real-world problem, you would have code that looks more like `x = self.dataset.__getitem__(idx)` ,because you wouldn't know the label `y` . Then you could `return x, x` .

With this `AutoEncodeDataset` wrapper in hand, we can load the original MNIST dataset and wrap it with `AutoEncodeDataset`, and we'll be ready to start training:

```
train_data = AutoEncodeDataset(torchvision.datasets.MNIST("./", train=True,
➥ transform=transforms.ToTensor(), download=True))
test_data_xy = torchvision.datasets.MNIST("./", train=False,
➥ transform=transforms.ToTensor(), download=True)
test_data_xx = AutoEncodeDataset(test_data_xy)

train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data_xx, batch_size=128)
```

Now we can train this PCA model the same way we have been training other neural networks. The `AutoEncodeDataset` makes the input also act as the label, `pca_model` combines the sequence of encoding and decoding the data, and `mseWithOrthoLoss` implements a PCA-specific loss function that combines: 1) making the output look like the input $\ell_{MSE}(f(x), x)$, and 2) maintaining the orthogonal weights that PCA desires ($\|W^\top W - I\|_2^2 = 0$):

```
train_network(pca_model, mseWithOrthoLoss, train_loader,
➥ test_loader=test_loader, epochs=10, device=device)
```

7.1.4 Visualizing PCA results

You may have noticed that we used the hidden layer size $n = 2$. This was intentional because it lets us *plot* the results and build some good visual intuition about how autoencoders work. This is because we can use PCA

to visualize our data in two dimensions when $n = 2$. This is a very common use case for PCA. Even if we used a larger target dimension, projecting the data down can make it faster and/or more accurate to search for similar data. So it is useful to have a function that will take a dataset and encode it all to the lower-dimensional space. The following function does that and copies the labels so that we can look at our results compared to the ground truth of the MNIST test data:

```
def encode_batch(encoder, dataset_to_encode):
    """
    encoder: the PyTorch network that takes in a dataset and converts it to
    ➥ a new dimension
    dataset_to_encode: a PyTorch 'Dataset' object that we want to convert.

    Returns a tuple (projected, labels) where 'projected' is the encoded
    ➥ version of the dataset, and 'labels' are the original labels
    ➥ provided by the 'dataset_to_encode'
    """
    projected = []                                                    ❶
    labels = []

    encoder = encoder.eval()                                          ❷
    encoder = encoder.cpu()                                           ❸


    with torch.no_grad():                                             ❹
        for x, y in DataLoader(dataset_to_encode, batch_size=128):
            z = encoder(x.cpu())                                      ❺
            projected.append( z.numpy() )                             ❻
            labels.append( y.cpu().numpy().ravel() )
```

```
    projected = np.vstack(projected)                          ❼

    labels = np.hstack(labels)
    return projected, labels                                  ❽
projected, labels = encode_batch(pca_encoder, test_data_xy)   ❾
```

❶ Creates space to store the results

❷ Switches to eval mode

❸ Switches to CPU mode for simplicity, but you don't have to

❹ We don't want to train, so torch.no_grad!

❺ Encodes the original data

❻ Stores the encoded version and label

❼ Turns the results into single large NumPy arrays

❽ Returns the results

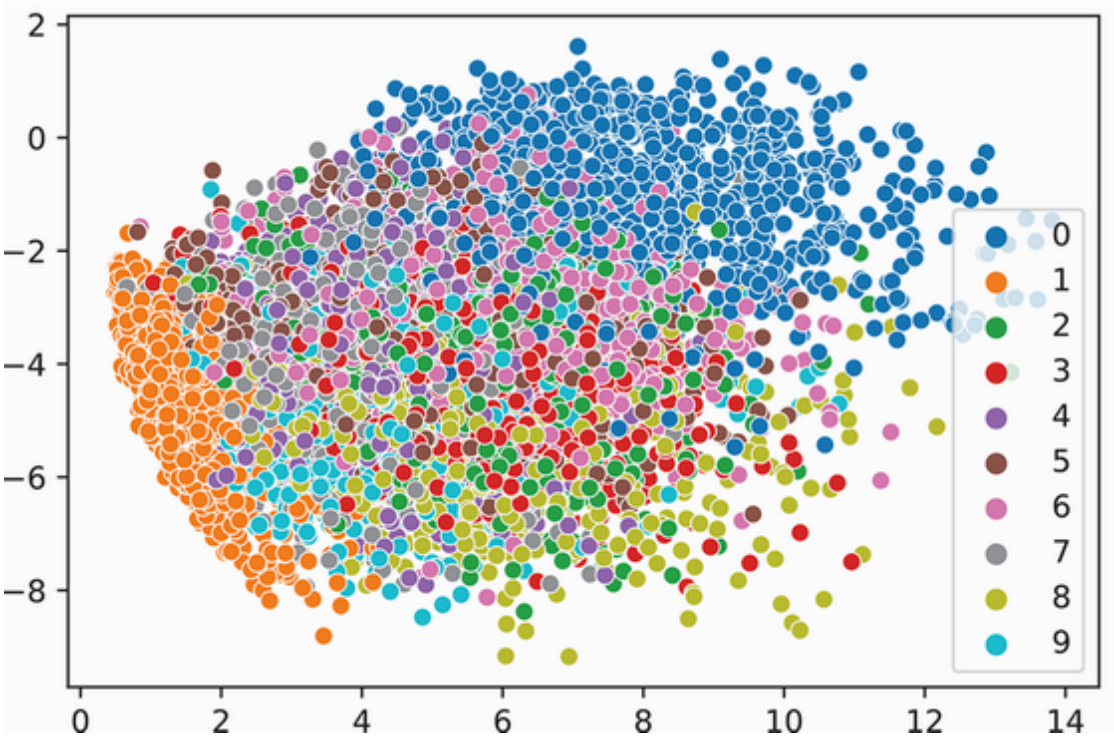
❾ Projects our data

Using the `encode_batch` function, we have now applied PCA to the dataset, and we can plot the results with seaborn. This should look like a very familiar PCA plot: some classes have decent separation from the others, while some are clumped together. The following code has this odd bit: `hue=[str(l) for l in labels], hue_order=[str(i) for i in range(10)]`, which is included to make the plot

easier to read. If we used `hue=labels`, the code would work fine, but seaborn would give all the digits similar colors and that would be hard to read. By making the labels strings (`hue=[str(l) for l in labels]`), we get seaborn to give each class a more distinct color, and we use the `huge_order` to make seaborn plot the classes in the order we expect:

```
sns.scatterplot(x=projected[:,0], y=projected[:,1],
➥ hue=[str(l) for l in labels],
➥ hue_order=[str(i) for i in range(10)], legend="full")
```

```
[15]: <AxesSubplot:>
```



From this plot, we can get some ideas about the quality of the encoding. For example, it's probably easy to differentiate the 0 and 1 classes from

all the others. Some of the others might be much harder, though; and in a truly unsupervised scenario, where we don't know the true labels, we won't be able to easily discover the distinct concepts. Another thing we can use to help judge this is the encode/decode process. If we did a good job, the output should be the same as the input. First, we define a simple helper function to plot the original input x on the left and the encoded-decoded version on the right:

```
def showEncodeDecode(encode_decode, x):
    """
    encode_decode: the PyTorch Module that does the encoding and decoding
    ➥ steps at once
    x: the input to plot as is, and after encoding & decoding it
    """

    encode_decode = encode_decode.eval() ❶
    encode_decode = encode_decode.cpu()  ❷
    with torch.no_grad():                ❸
        x_recon = encode_decode(x.cpu())
    f, axarr = plt.subplots(1,2)         ❹
    axarr[0].imshow(x.numpy()[0,:])
    axarr[1].imshow(x_recon.numpy()[0,0,:])
```

❶ Switches to eval mode

❷ Moves things to the CPU so we don't have to think about what device anything is on and because this function is not performance-sensitive

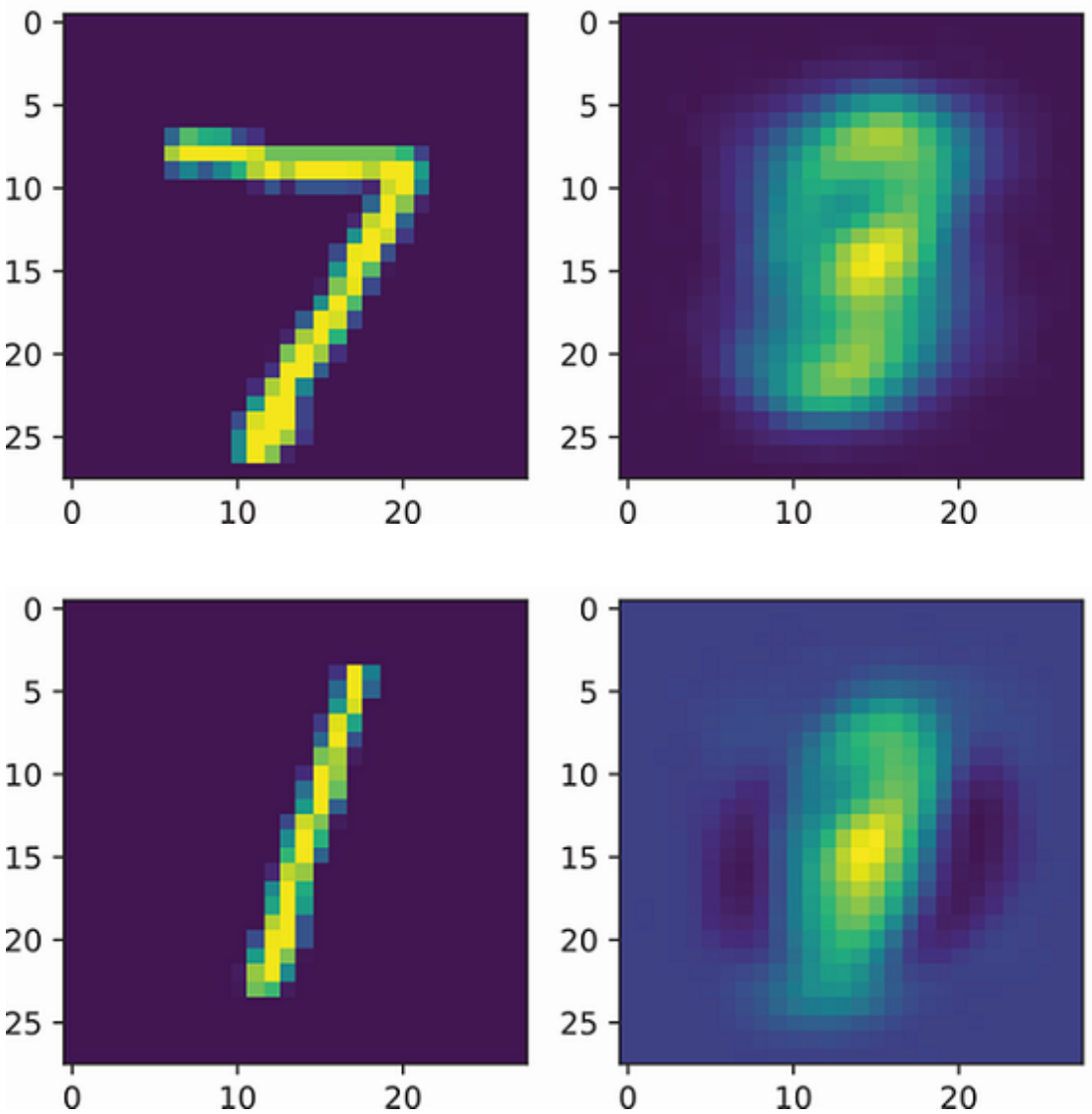
❸ Always no_grad if you are not training

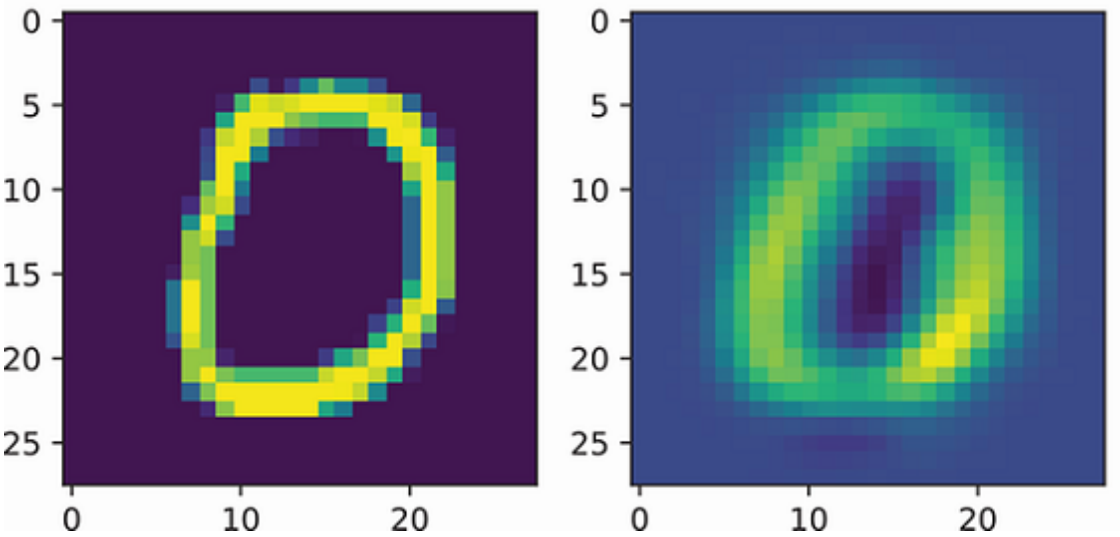
❹ Uses Matplotlib to create a side-by-side plot with the original on the left

We reuse this function throughout this chapter. First let's look at some input-output combinations for a few different digits:

```
showEncodeDecode(pca_model, test_data_xy[0][0])
showEncodeDecode(pca_model, test_data_xy[2][0])
showEncodeDecode(pca_model, test_data_xy[10][0]) ❶
```

❶ Shows the input (left) and output (right) for three data points


0
5
10
15
20
25
0
10
20



These results match what we expected based on the 2D plot. The 0 and 1 classes look kind of like a 1 and 0 after we encode and decode them. The 7 . . . not so much. Why? Well, we are converting 784 dimensions down to 2. That's a *lot* of information compression—much more than we can reasonably expect poor PCA to be able to do.

7.1.5 A simple nonlinear PCA

PCA is one of the simplest autoencoders we could design because it is a completely linear model. What happens if we just add a little of what we have learned about? We can add a single nonlinearity and remove the weight sharing to turn this into a small nonlinear autoencoder. Let's see how that looks:

```
pca_nonlinear_encode = nn.Sequential(   ❶
    nn.Flatten(),
    nn.Linear(D, n),
    nn.Tanh(),                          ❷
```

```
)

pca_nonlinear_decode = nn.Sequential(   ❸
    nn.Linear(n, D),                    ❹
    View(-1, 1, 28, 28)
)
pca_nonlinear = nn.Sequential(          ❺
    pca_nonlinear_encode,
    pca_nonlinear_decode
)
```

❶ Augments the encoder with a Tanh nonlinearity

❷ The only real change: adding a nonlinear operation at the end

❸ The decoder gets its own Linear layer, making it look more like a normal network.

❹ We are no longer tying the weights, for simplicity.

❺ Combines them into the encoder-decoder function $f(\cdot)$

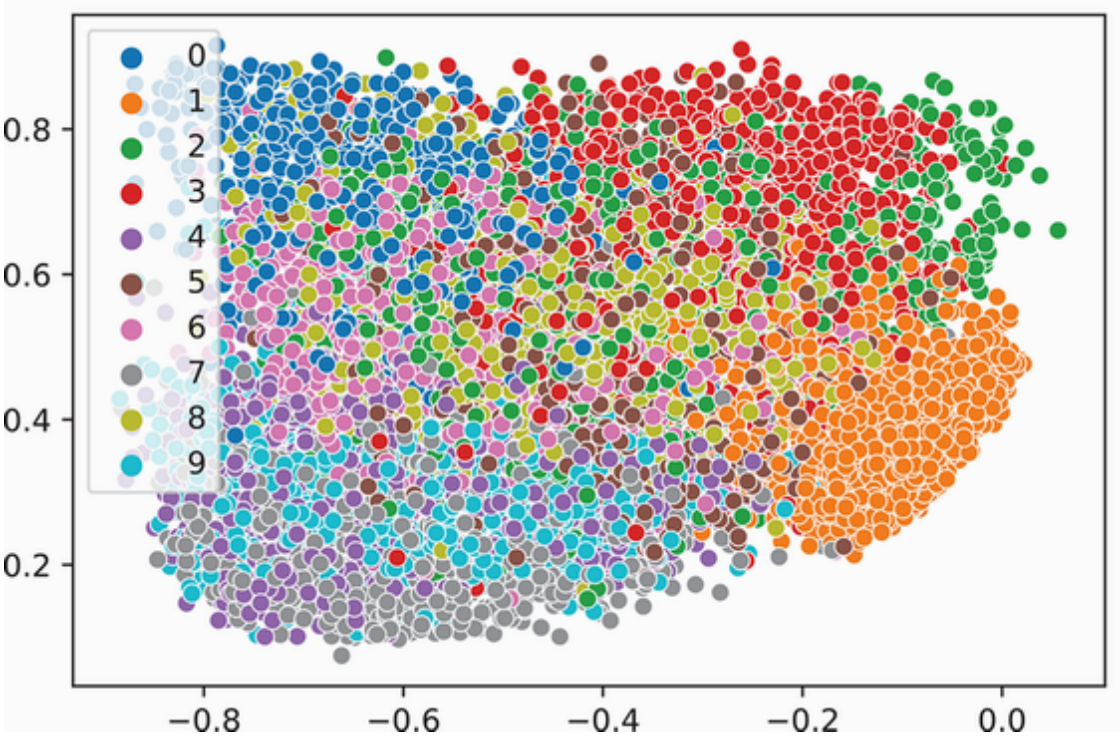
Since we are no longer sharing weights between the encoder and decoder, we do not care if the weights are orthogonal. So when we train this model, we use the normal MSE loss:

```
train_network(pca_nonlinear, mse_loss, train_loader,
➥ test_loader=test_loader, epochs=10, device=device)
```

In the following blocks of code, we again plot all the 2D encodings and our three encoded-decoded images to see visually what has changed. This lets us look subjectively to see if the quality is better:

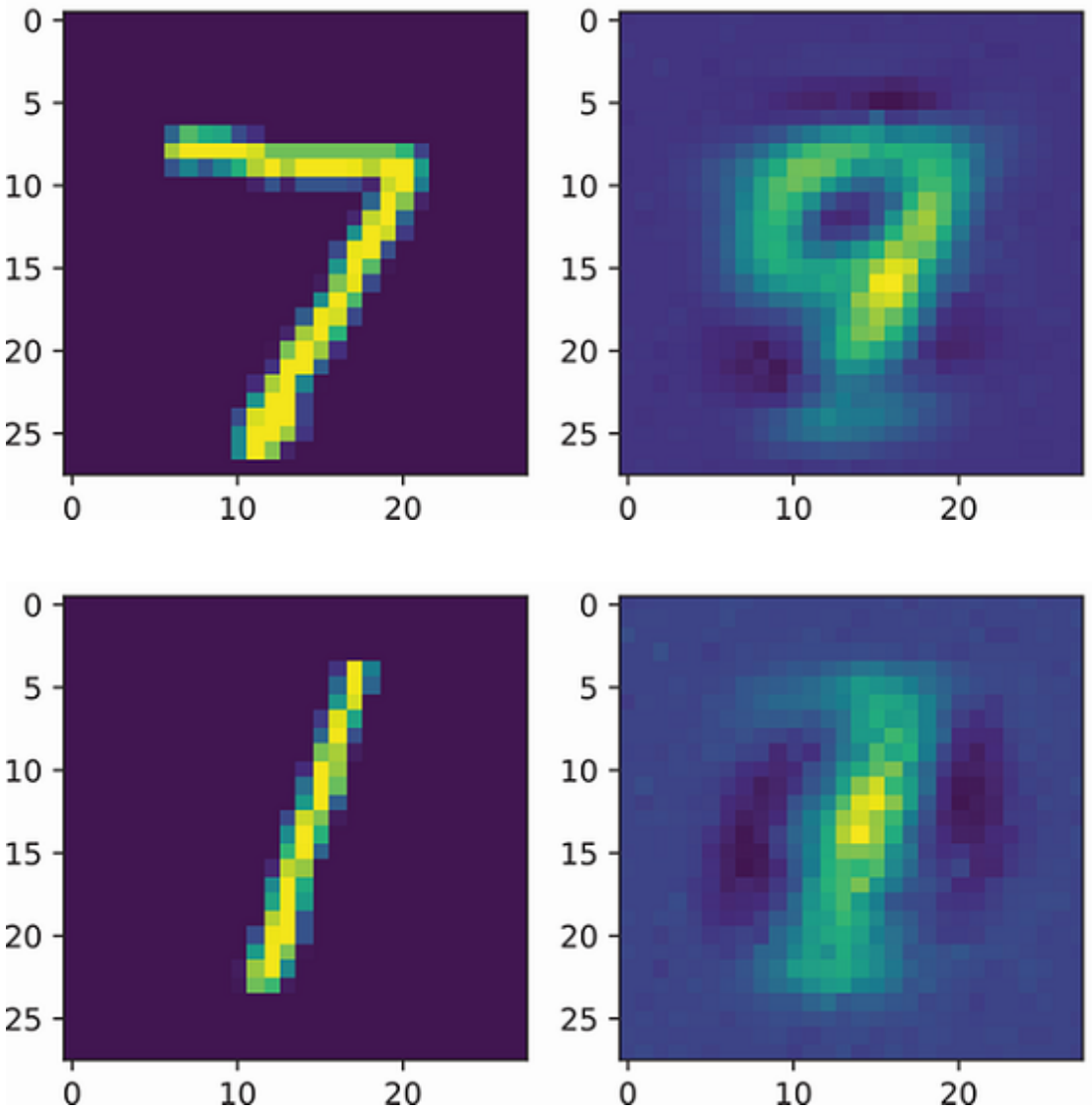
```
projected, labels = encode_batch(pca_nonlinear_encode, test_data_xy)
sns.scatterplot(x=projected[:,0], y=projected[:,1],
➥ hue=[str(l) for l in labels],
➥ hue_order=[str(i) for i in range(10)], legend="full" )
```

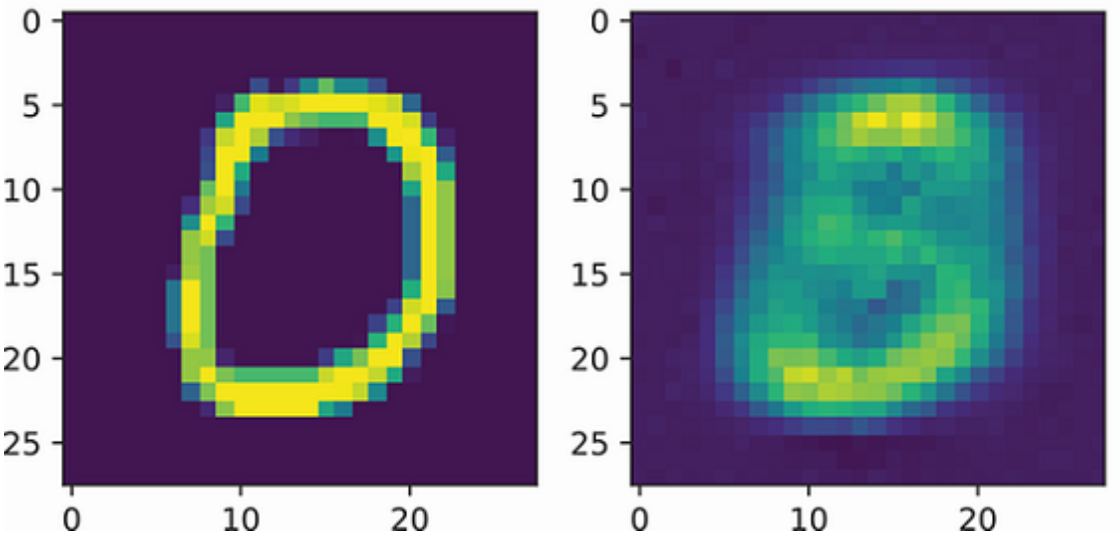
```
[20]: <AxesSubplot:>
```




```
showEncodeDecode(pca_nonlinear, test_data_xy[0][0])
showEncodeDecode(pca_nonlinear, test_data_xy[2][0])
```

```
showEncodeDecode(pca_nonlinear, test_data_xy[10][0])
```


0
5
10
15
20
25
0
10
20
0
5
10
15
20
25
0
10
20
0
5
10
15
20
25
0
10
20



Overall, the change is noticeable but not clearly different. The 2D plot still has a lot of overlap. The encode-decode images show a few artifacts: 0, 1, and 7 are qualitatively similar but with different styles. What was the point? We have turned PCA into an autoencoder with one nonlinearity, so we have modified the PCA algorithm; and because we did so in PyTorch, it trained up and worked out of the box. Now we can try to make bigger changes to get a better result. As a theme of deep learning, if we make this model deeper by adding more layers, we should be able to successfully improve the results.

7.2 Designing autoencoding neural networks

PCA is a very popular method for dimensionality reduction and visualization, and any situation where you would use PCA is one where you may want to instead use an *autoencoding network*. An autoencoding network is the same idea, but we will make the encoder and decoder larger networks with more layers so they can learn more powerful and complex encoders and decoders. Since we have seen that PCA is an autoencoder,

an autoencoding network may be a more accurate choice by being able to learn more complex functions.

Autoencoding networks are also useful for *outlier detection*. You want to detect outliers so that you can have them manually reviewed, because an outlier is unlikely to be handled well by a model. An autoencoder can detect outliers by looking at how well it reconstructed the input. If you can reconstruct the input well, the data probably looks like normal data you've seen. If you *cannot* successfully reconstruct the input, it is probably *unusual* and thus an outlier. This process is summarized in figure 7.3. You can use this approach to find potentially bad data points in your training data or validate user-submitted data (e.g., if someone uploads a picture of their face to an application that looks for ear infections, you should detect the face as an outlier and not make a diagnosis).

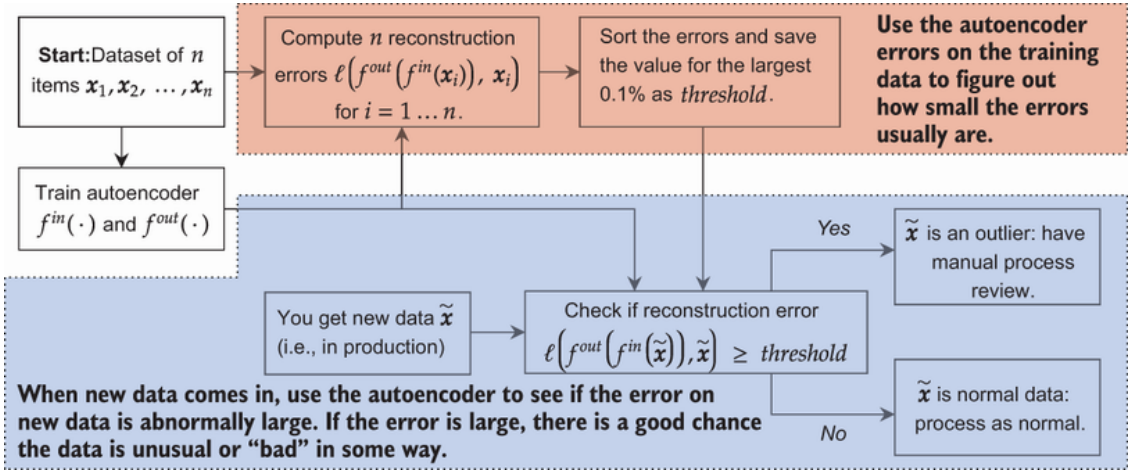


Figure 7.3 Example application of autoencoders to detect outliers. A dataset is used to train the autoencoder, and then you compute all the reconstruction errors. Assuming that 0.1% of the data is abnormal, it should be the most difficult to reconstruct. If you find the threshold for the top 0.1% of errors, you can apply that threshold to new data to detect outliers. Outliers are often ill-behaved, and it may be better to treat them differently or give them an extra review. Outlier detection can be done on training data or new testing data. You can change the 0.1% to match what you believe is happening in your dataset.

Now let's talk about how to set up a deep learning-based autoencoder. The standard approach is to make a symmetric architecture between the encoder and decoder: keep the same number of layers in each and put them in the reverse order (encoder goes from big to small, and decoder goes from small to big). We also use a *bottleneck* style encoder, meaning the layers have progressively fewer neurons. This is shown in figure 7.4.

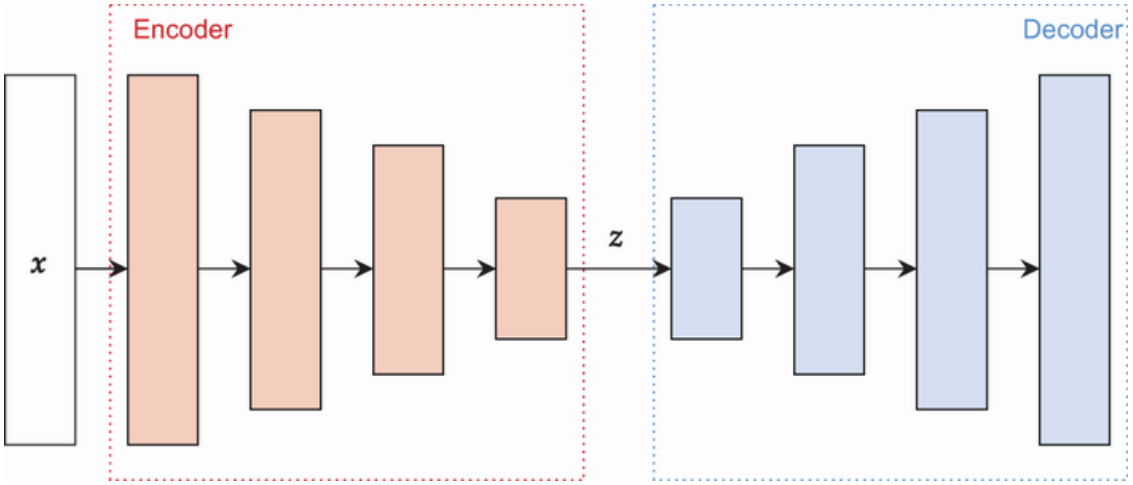


Figure 7.4 Example of a standard autoencoder design. The input comes in on the left. The encoder starts out large and tapers down the size of the hidden layers at each step. Because autoencoders are usually symmetric, our decoder will receive the small representation z and begin expanding it back to the original size.

Autoencoders do not *have* to be symmetric. They work *just fine* if they are not. This is done purely to make it easier to think and reason about the network. This way, you are making half as many decisions about how many layers are in the network, how many neurons are in each layer, and so on.

The bottleneck in the encoder *is* important, though. Just as PCA did, by pushing down to a smaller representation, we make it impossible for the

network to cheat and learn the naive solution of immediately returning the input as the output. Instead, the network must learn to identify high-level concepts like “there is a circle located at the center,” which could be used for encoding (and then decoding) the numbers 6 and 0. By learning multiple high-level concepts, the network is forced to start learning useful representations.

NOTE The autoencoder can also be seen as a way to produce embeddings. The encoder part of the network in particular makes an excellent candidate for an embedding construction. The approach is unsupervised, so you don’t need labels, and the encoder’s lower-dimensional output works better with common tools for visualization and nearest-neighbor search.

7.2.1 Implementing an autoencoder

Since we have gone through the pain of implementing PCA, the autoencoder should be easier and more straightforward when done in this style, primarily because we do not share any weights across layers and no longer need the orthogonal constraint on the weights. For simplicity, we focus on fully connected networks for autoencoders, but the concepts are widely applicable. To start, let’s define another helper function `getLayer` that creates a single hidden layer for us to place in a network, similar to what we did in chapter 6:

```
def getLayer(in_size, out_size):
    """
    in_size: how many neurons/features are coming into this layer
    out_size: how many neurons/outputs this hidden layer should produce
```

```
    """
    return nn.Sequential( ❶
        nn.Linear(in_size, out_size),
        nn.BatchNorm1d(out_size),
        nn.ReLU())
```

❶ Organizes the conceptual “block" of a hidden layer into a Sequential object

With this helper function in hand, the following code shows how easy it is to implement an autoencoder using the more advanced tools we have learned about, like batch normalization and ReLU activations. It uses a simple strategy of decreasing the number of neurons in each hidden layer by a fixed pattern. In this case, we divide the number of neurons by 2, then 3, then 4, and so on until the last layer of the decoder, where we jump straight to the target size D'. The pattern used to decrease the number of layers is not that important, as long as the size of the layers consistently decreases:

```
auto_encoder = nn.Sequential(                ❶
    nn.Flatten(),
    getLayer(D, D//2),                       ❷
    getLayer(D//2, D//3),
    getLayer(D//3, D//4),
    nn.Linear(D//4, n),                      ❸
)

auto_decoder = nn.Sequential(                ❹
    getLayer(n, D//4),                       ❺
    getLayer(D//4, D//3),
```

```
    getLayer(D//3, D//2),

    nn.Linear(D//2, D),
    View(-1, 1, 28, 28)                 ❻
)
auto_encode_decode = nn.Sequential(     ❼
    auto_encoder,
    auto_decoder
)
```

❶ Dividing by 2, 3, 4, etc., is one of many patterns that can be used.

❷ Each layer has a smaller output size than the previous one.

❸ Jumps down to the target dimension

❹ Decoder does the same layers/sizes in reverse to be symmetric

❺ Each layer increases in size because we are in the decoder.

❻ Reshapes to match the original shape

❼ Combines into a deep autoencoder

As always, we can train this network using the exact same function. We stick with the mean squared error, which is very common in autoencoders:

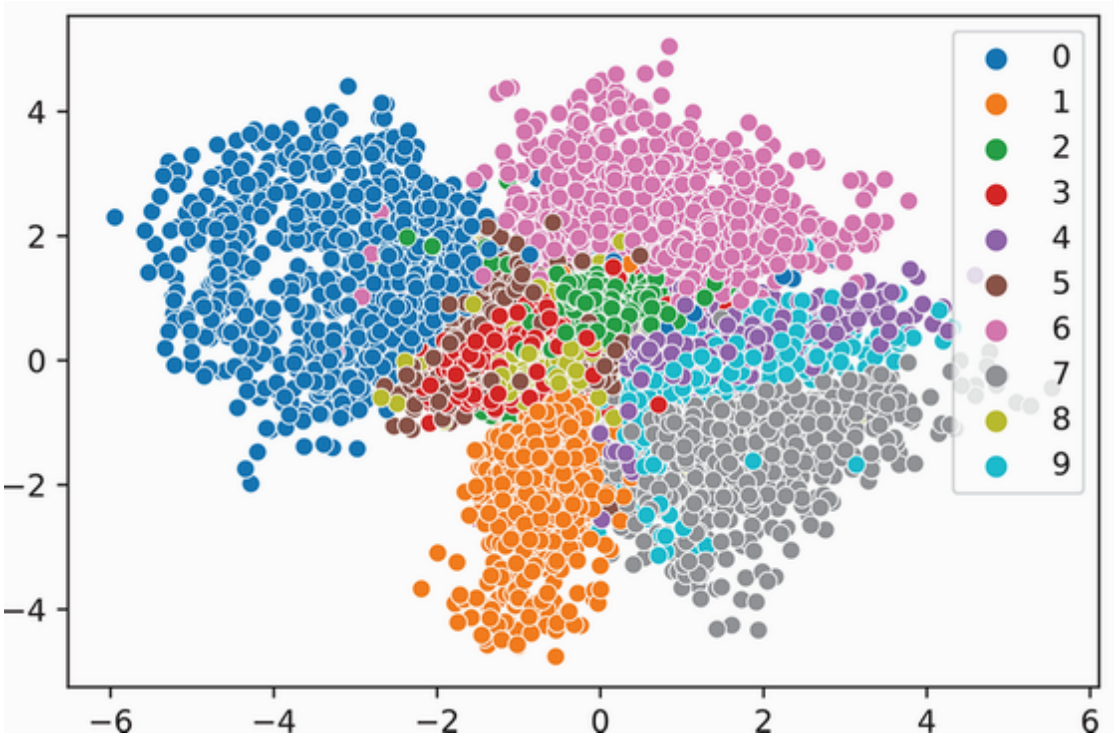
```
train_network(auto_encode_decode, mse_loss, train_loader,
➥ test_loader=test_loader, epochs=10, device=device)
```

7.2.2 Visualizing autoencoder results

How did our new autoencoder do? The 2D plot shows *much* more separation in the projected dimension z. Classes 0, 6, and 3 are *very* well separated from all the others. In addition, the middle area where there are more classes next to each other at least has more continuity and uniformity in the class present. The classes have distinct homes in the middle area, rather than being smeared on top of each other:

```
projected, labels = encode_batch(auto_encoder, test_data_xy)
sns.scatterplot(x=projected[:,0], y=projected[:,1],
➥ hue=[str(l) for l in labels],
➥ hue_order=[str(i) for i in range(10)], legend="full")
```

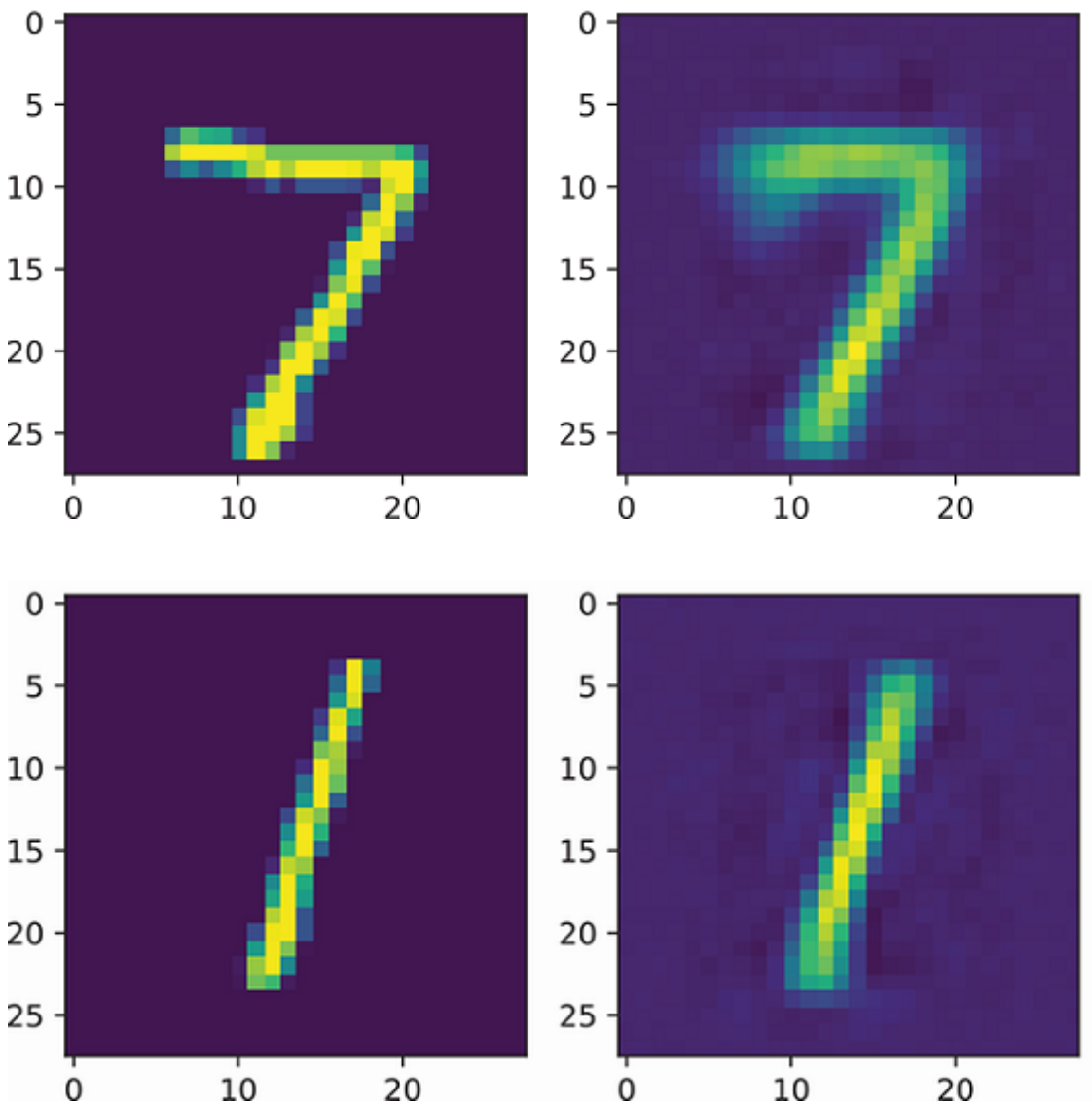
```
[25]: <AxesSubplot:>
```

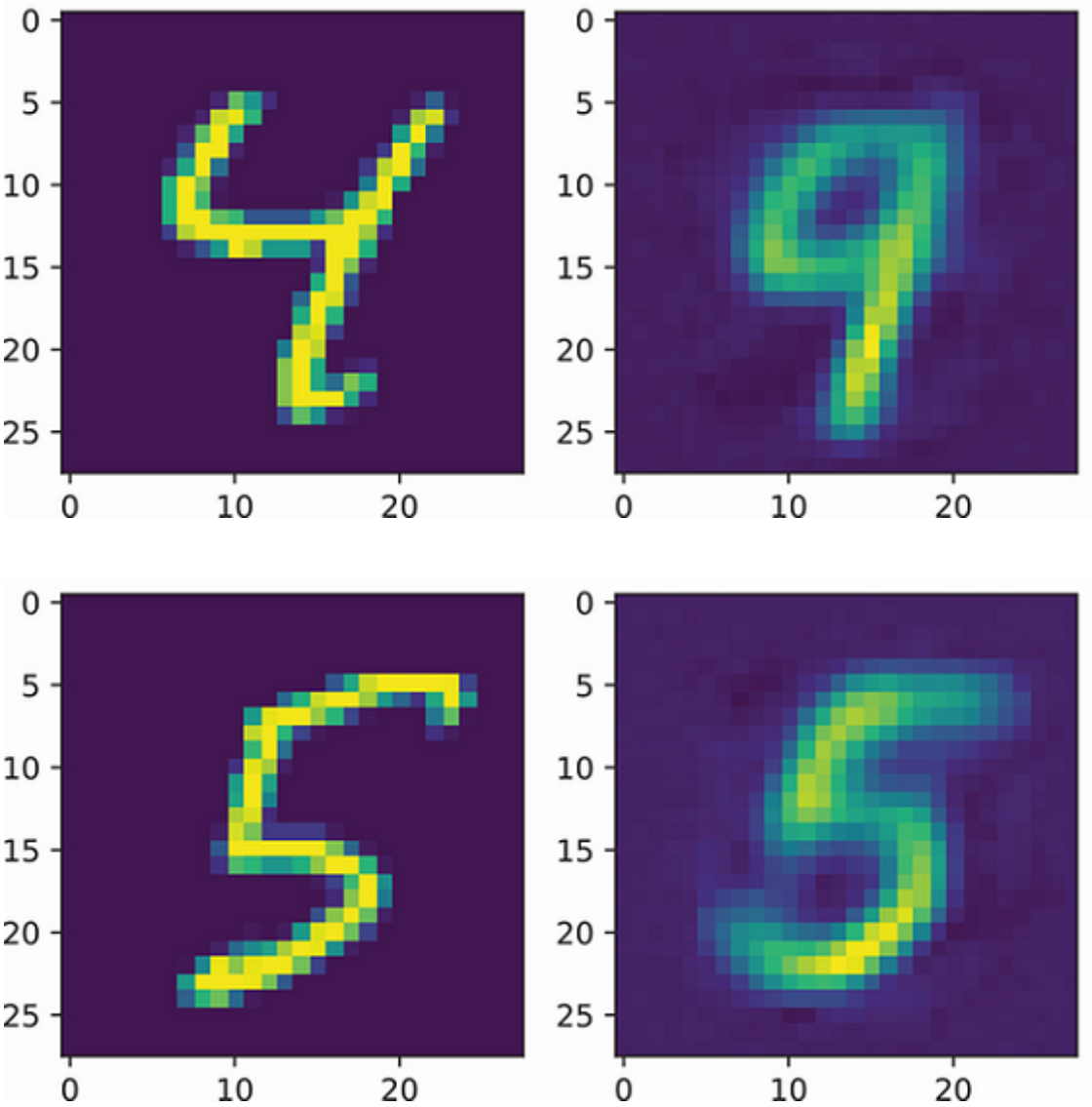


This is also one way to use autoencoders to explore unknown data. If we did not know the class labels, we might have concluded from this projection that there were likely at least two to four different subpopulations in the data.

We can also look at some examples of the encode-decode cycle. Unlike before, the reconstructions are now crisp, with much less blur. But it's not perfect: class 4 is usually hard to separate from others and has low-quality reconstructions:

```
showEncodeDecode(auto_encode_decode, test_data_xy[0][0])
showEncodeDecode(auto_encode_decode, test_data_xy[2][0])
showEncodeDecode(auto_encode_decode, test_data_xy[6][0])
showEncodeDecode(auto_encode_decode, test_data_xy[23][0])
```


0
5
10
15
20
25
0
10
20



Try playing with the code and looking at the results for different data points. If you do, you may start to notice that the reconstructions do not always maintain the *style* of the inputs. This is more obvious with the 5 here: The reconstruction is smoother and more pristine than the original input. Is this a good thing or a bad thing?

From the perspective of how we trained the model, it's a bad thing. The reconstruction *is different* from the input, and the goal was to *exactly* reconstruct the input.

However, our true goal is not to just learn to reconstruct inputs from themselves. We already have the input. Our goal was to learn a useful representation of the data without needing to know the data labels. From this perspective, the behavior is a good thing: it means there are multiple different potential "5"'s that could be the input and would be mapped to the same "5" reconstruction. In this sense, the network has on its own learned that there is a *canonical* or *prototypical* 5, without being explicitly told about the concept of 5 or even that there are distinct numbers present.

But the example of the digit 4 is a bad failure case. The network reconstructed a completely different digit because the restriction is *too strong*: the network was forced down to just two dimensions and couldn't learn all the complexity in the data with such little space. This meant forcing out the concept of a 4. Similar to PCA, if you give the network a larger bottleneck (more features to use), the reconstructions will steadily improve in quality. Using two dimensions is great for visualization in scatterplots, but for other applications, you probably want to use a few more features. (This, like most things in ML, is problem-specific. You should make sure you have a way to test your results to compare them, and then use that test to determine how many features you should use).

7.3 Bigger autoencoders

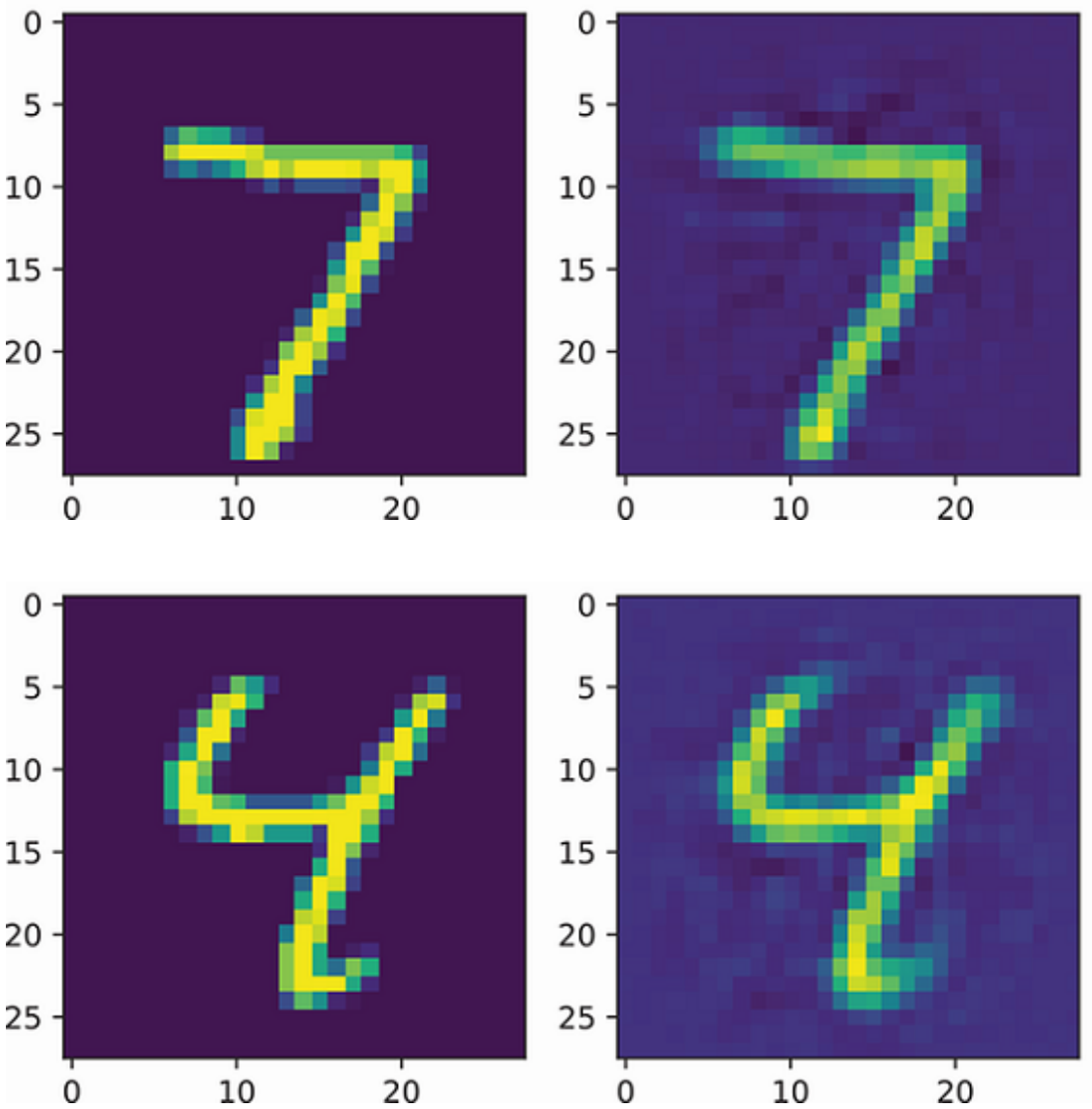
All the autoencoding we have done so far has been based on projecting down to two dimensions, which we have already said makes the problem exceptionally hard. Your intuition should tell you that if we make the target dimension size D' a little larger, our reconstructions should improve in quality. But what if we made the target size *larger* than the original input? Would this work? We can easily modify our autoencoder to try this and see what happens. In the following block of code, we simply jump up to $D' = 2 \cdot D$ right after the first layer of the encoder and stay at that number of neurons for the entire process:

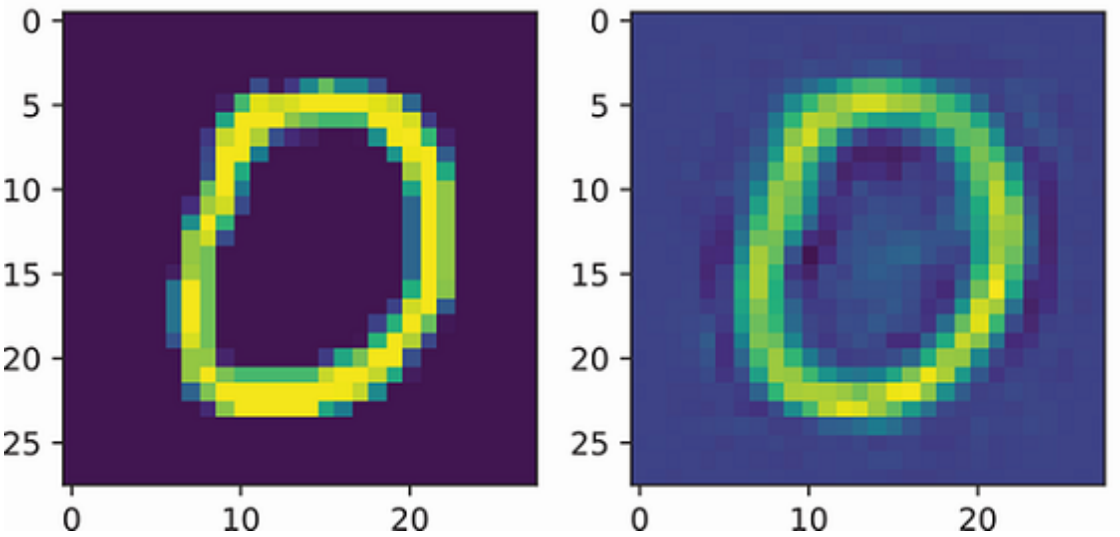
```
auto_encoder_big = nn.Sequential(
    nn.Flatten(),
    getLayer(D, D*2),
    getLayer(D*2, D*2),
    getLayer(D*2, D*2),
    nn.Linear(D*2, D*2),
)

auto_decoder_big = nn.Sequential(
    getLayer(D*2, D*2),
    getLayer(D*2, D*2),
    getLayer(D*2, D*2),
    nn.Linear(D*2, D),
    View(-1, 1, 28, 28)
)

auto_encode_decode_big = nn.Sequential(
    auto_encoder_big,
    auto_decoder_big
```

```
)

train_network(auto_encode_decode_big, mse_loss, train_loader,
➥ test_loader=test_loader, epochs=10, device=device)
```

We can't make a 2D plot since we have too many dimensions. But we can still do the encode/decode comparisons on the data to see how our new autoencoder performs. If we plot some examples, it becomes clear that we now have *very good* reconstructions, which include even minute details from the original input. For example, the following 7 has a slight uptick at the top left and a slightly thicker ending at the bottom, which are present in the reconstruction. The 4 that was completely mangled previously has a lot of unique curves and styles that are also faithfully preserved:

```
showEncodeDecode(auto_encode_decode_big, test_data_xy[0][0])
showEncodeDecode(auto_encode_decode_big, test_data_xy[6][0])
showEncodeDecode(auto_encode_decode_big, test_data_xy[10][0])
```


0
5
10
15
20
25
0
10
20
0
5
10
15
20
25
0
10
20
0
5
10
15
20
25
0
10
20



So the question is, is this a better autoencoder than the previous one? Have we learned a useful representation? It's a hard question to answer because we are using the input reconstruction as the loss of the network, but it is not what we truly care about. We want the network to learn *useful* representations. This is a variant of the classic unsupervised learning problem: if you don't know what you are looking for, how do you know if you are doing a good job?

7.3.1 Robustness to noise

To help us answer the question about which autoencoder is better, $D' = 2$ or $D' = 2 \cdot D$, we will add some noise to our data. Why? One intuition we can use is that if a representation is good, it should be *robust*. Imagine if the clean data we have been using was like a road, and our model was the car. If the road is pristine and smooth, the car will handle well. But what if there are potholes and cracks (read, noise) in the road? A good car should still be able to drive successfully. Similarly, if we have noisy data, an ideal model will still perform well.

There are *many* different ways we could make our data noisy. One of the easiest is to add noise from a normal distribution. We denote the normal distribution as $N(\mu,\sigma)$, where μ is the mean value returned and σ is the standard deviation. If s is a value sampled from the normal distribution, we denote that as $s \sim N(\mu,\sigma)$.

To make our data noisy, we use PyTorch to construct an object that represents the normal distribution and perturbs the input data such that we get $\tilde{\boldsymbol{x}} = \boldsymbol{x} + s$, where $s \sim N(\mu,\sigma)$. To represent the normal distribution $N(\mu,\sigma)$, PyTorch provides the `torch.distributions.Normal` class:

```
normal = torch.distributions.Normal(0, 0.5) ❶
```

❶ First argument is the mean μ; second is the standard deviation σ

This class has a `sample` method that performs the $s\sim$ step. We use the `sample_shape` argument to tell it that we want a tensor with a shape of `sample_shape` to be filled with random values from this distribution. The following function takes an input $\boldsymbol{x}$ and sample noise that is the same shape as $\boldsymbol{x}$ so we can add it, creating our noisy sample $\tilde{\boldsymbol{x}} = \boldsymbol{x} + s$:

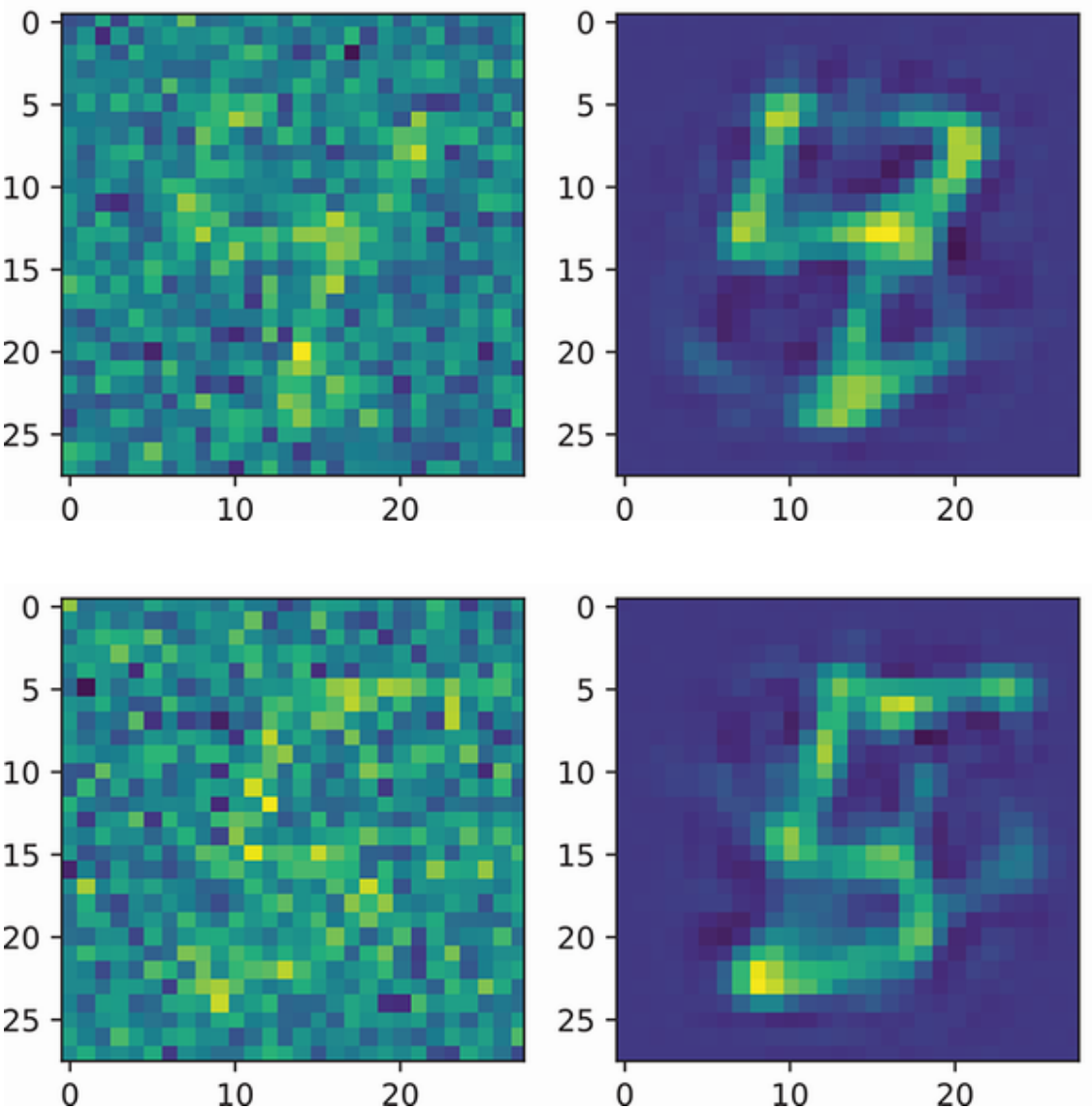
```
def addNoise(x, device='cpu'):
    """
    We will use this helper function to add noise to some data.
    x: the data we want to add noise to
    device: the CPU or GPU that the input is located on.
    """
    return x +
```

```
        normal.sample(sample_shape=
        ➥ torch.Size(x.shape)).to(device) ❶
```

❶ $x + s$

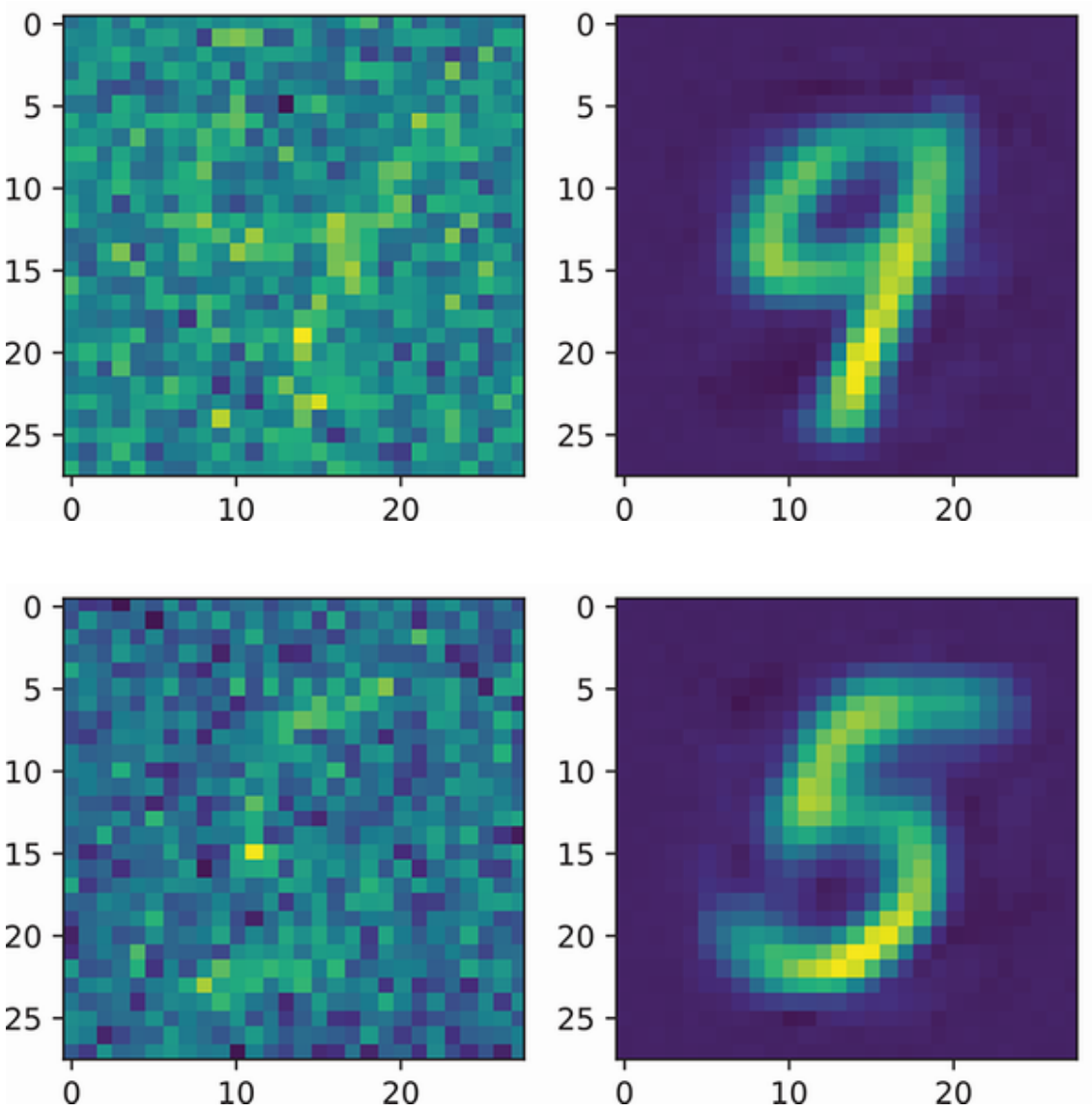
With our simple `addNoise` function in place, we can try it with our big model. We have intentionally set the amount of noise to be fairly large to make the changes and differences between models obvious. For the following input data, you should see that the reconstructions are garbled, with extraneous lines. Since the noise is random, you can run the code multiple times to see different versions:

```
showEncodeDecode(auto_encode_decode_big, addNoise(test_data_xy[6][0]))
showEncodeDecode(auto_encode_decode_big, addNoise(test_data_xy[23][0]))
```



This would seem to indicate that our large autoencoder with $D' = 2 \cdot D$ is not very robust. What happens if we apply the same noisy data to our original autoencoder, which uses $D = 2$? You can see that next. The 5 is reconstructed almost exactly as it was before: a little blurry, but clearly a 5:

```
showEncodeDecode(auto_encode_decode, addNoise(test_data_xy[6][0]))
showEncodeDecode(auto_encode_decode, addNoise(test_data_xy[23][0]))
```



If you run the 4 through multiple times, sometimes you get a 4 back out of the decoder, and sometimes you get something else. This is because the

noise is different every time, and our 2D plot showed that the 4 was being conflated with many other classes.

Based on this experiment, we can see that as we make the encoding dimension D' smaller, the model becomes *more robust*. If we let the encoding dimension become too large, it may be good at performing reconstructions on easy data, but it is not robust to changes and noise. Part of this is because when $D' \geq D$, it becomes easy for the model to learn a simple approach. It has more than enough capacity to copy the input and learn to regurgitate what it was given. By constraining the model with a smaller capacity ($D' \leq D$), the only way it can learn to solve the task is by creating a more compact representation of the input data. Ideally, you try to find a dimension D' that fits a balance between reconstructing the data well but using as small an encoding dimension as possible.

7.4 Denoising autoencoders

It is not easy to balance having D' small enough to be robust but large enough to do well at reconstruction. But there is a trick we can play that will allow us to have large $D' > D$ *and* learn a robust model. The trick is to create what is called a *denoising autoencoder*. A denoising autoencoder adds noise to the encoder's input while still expecting the decoder to produce a clean image. So our math goes from $\ell(f(\boldsymbol{x}), \boldsymbol{x})$ to $\ell(f(\tilde{\boldsymbol{x}}), \boldsymbol{x})$. If we do this, there is no naive solution of just copying the input, because we perturb it before giving it to the network. The network must learn how to remove the noise, or denoise, the input and thus allows us to use $D' > D$ while still obtaining robust representations.

Denoising networks have a lot of practical usages. If you can make synthetic noise that is realistic to the issues you might see in real life, you can create models that remove the noise and improve the accuracy by making the data cleaner. Libraries like scikit-image (**https://scikit-image.org**) are available with many transforms that can be used to make noisy images, and I've personally used this approach to improve fingerprint recognition algorithms.[3] How I used a denoising autoencoder is shown in figure 7.5, which is also a summary of how denoising autoencoders are typically set up. The original (or sometimes very clean) data comes in at the start, and we apply noise-generating processes to it. The more that noise looks like the kinds of issues you see in real data, the better. The noisy/corrupted version of the data is given as the input to the autoencoder, but the loss is computed against the original clean data.

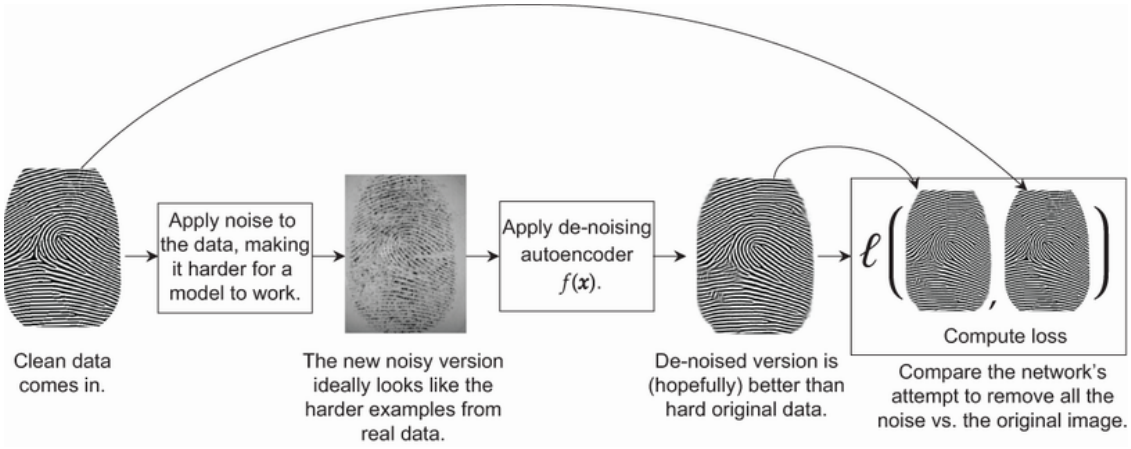


Figure 7.5 The denoising autoencoder processes applied to fingerprint images. This used special software that generates hyper-realistic fingerprint images, with the goal of removing that noise to make fingerprint processing less error prone. You can still get good results with simpler and unrealistic noise.

7.4.1 Denoising with Gaussian noise

We will make only one change to our previous `auto_encoder_big` model: we will add a new layer at the beginning of the encoder subnetwork, which adds noise to the input *only when we are training*. The assumption is usually that our training data is relatively clean and prepared, and we are adding noise to make it more robust. If we are then *using* the model, and no longer training, we want to get the best answer we can—which means we want the cleanest data possible. Adding noise at that stage would make our lives more difficult, and if the input already had noise, we would just compound the problem.

So the first code we need is a new `AdditiveGaussNoise` layer. It takes the input x in. If we are in training mode (denoted by `self.training`), we add noise to the input; otherwise we return it unperturbed:

```
class AdditiveGaussNoise(nn.Module): ❶
    def __init__(self):
    super().__init__()

    def forward(self, x):
        if self.training:            ❷
            return addNoise(x, device=device)
        else:                        ❸
            return x
```

❶ We don't need to do anything in the constructor of this object.

❷ Every PyTorch Module object has a self.training boolean that can be used to check if we are in training (True) or evaluation (False) mode.

❸ Now training: return the data as it was given.

Next, we redefine the same large autoencoder as before, where $D' = 2 \cdot D$. The only difference is that we insert the `AdditiveGaussNoise` layer at the start of the network:

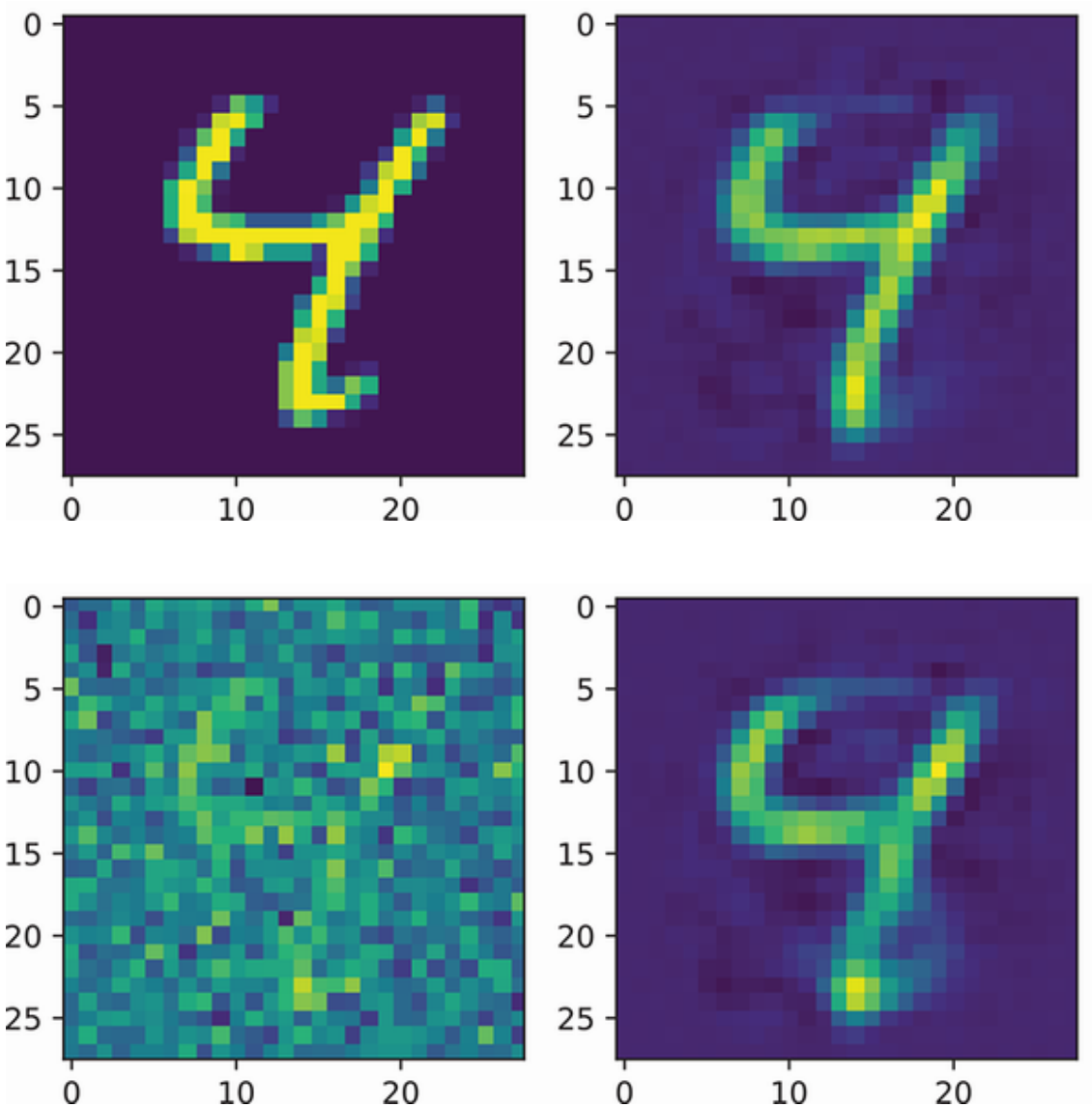
```
dnauto_encoder_big = nn.Sequential(
    nn.Flatten(),
    AdditiveGaussNoise(),                                    ❶
    getLayer(D, D*2),
    getLayer(D*2, D*2),
    getLayer(D*2, D*2),
    nn.Linear(D*2, D*2),
)

dnauto_decoder_big = nn.Sequential(
    getLayer(D*2, D*2),
    getLayer(D*2, D*2),
    getLayer(D*2, D*2),
    nn.Linear(D*2, D),
    View(-1, 1, 28, 28)
)

dnauto_encode_decode_big = nn.Sequential(
    dnauto_encoder_big,
    dnauto_decoder_big
)
train_network(dnauto_encode_decode_big, mse_loss, train_loader,
➥ test_loader=test_loader, epochs=10, device=device) ❷
```

❶ Only addition! We hope inserting noise here helps.

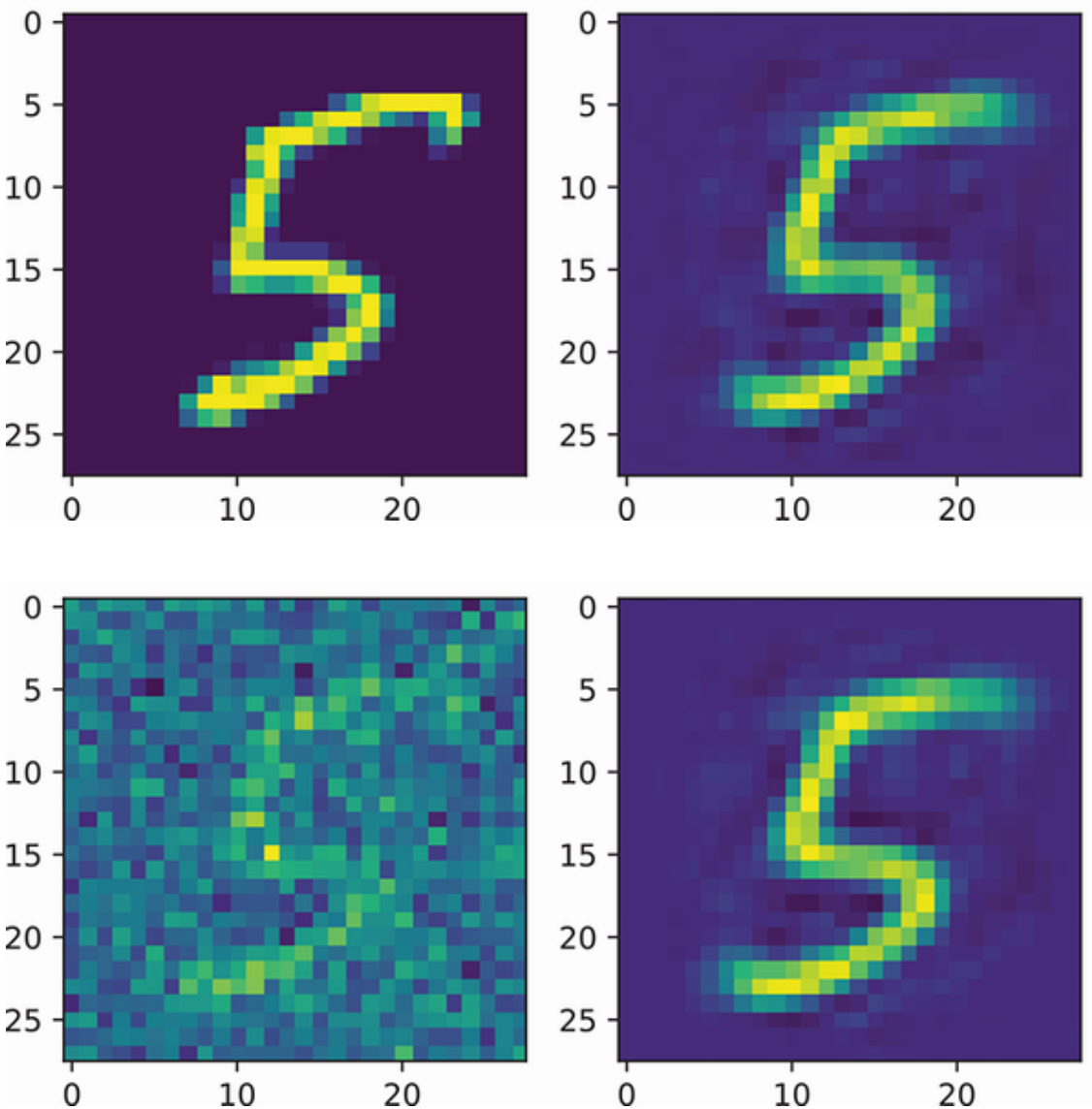
❷ Trains as usual

How well does it do? Following, we can see the same data reconstructed when there is and is not noise. The new denoising model is clearly the best at creating reconstructions of all the models we have developed so far. In both cases, the denoising autoencoder captures most of the style of the individual digits. The denoising approach still misses small details, likely because they are so small that the model isn't sure if they are a real part of the style or part of the noise. For example, the flourishes on the bottom of the 4 and the top of the 5 are missing after reconstruction:

```
showEncodeDecode(dnauto_encode_decode_big, test_data_xy[6][0])
showEncodeDecode(dnauto_encode_decode_big, addNoise(test_data_xy[6][0]))
```




```
showEncodeDecode(dnauto_encode_decode_big, test_data_xy[23][0])
showEncodeDecode(dnauto_encode_decode_big, addNoise(test_data_xy[23][0]))
```



The denoising approach is very popular for training autoencoders, and the trick of introducing your own perturbations into the data is widely used to build more accurate and robust models. As you learn more about deep learning and different applications, you will find many forms and spins on this approach.

Beyond helping learn more robust representations, the denoising approach can itself be a useful model. Noise can naturally occur in many situations. For example, when performing optical character recognition (OCR) to convert an image into searchable text, you can get noise from damage to the camera, damage to the document (e.g., water or coffee stains), changes in lighting, objects casting shadows, and so on. Many OCR systems have been improved by learning to add noise that looks like the noise seen in real life and asking the model to learn in spite of it.

Denoising with dropout

Adding Gaussian noise can be cumbersome because we need to figure out exactly how much noise to add, which can change from dataset to dataset. A second, more popular approach is to use *dropout*.

Dropout is a very simple idea: with some probability p, zero out any given feature value. This forces the network to be robust because it can *never* rely on any specific feature or neuron value, since p% of the time, the feature or value will not be there. Dropout is a very popular regularizer that can be applied to both the *input* of a network and to the *hidden layers*.

The following code block trains up a dropout-based denoising autoencoder. By default, dropout uses $p = 50\%$, which is fine for hidden layers but on the aggressive size for the input. So for the input, we apply only $p = 20\%$:

```
dnauto_encoder_dropout = nn.Sequential(
    nn.Flatten(),
    nn.Dropout(p=0.2),                                    ❶
```

```
    getLayer(D, D*2),
    nn.Dropout(),                                          ❷
    getLayer(D*2, D*2),
    nn.Dropout(),
    getLayer(D*2, D*2), nn.Dropout(), nn.Linear(D*2, D*2)
)

dnauto_decoder_dropout = nn.Sequential(
    getLayer(D*2, D*2),
    nn.Dropout(),
    getLayer(D*2, D*2),
    nn.Dropout(),
    getLayer(D*2, D*2),
    nn.Dropout(),
    nn.Linear(D*2, D),
    View(-1, 1, 28, 28)
)

dnauto_encode_decode_dropout = nn.Sequential(
dnauto_encoder_big,
dnauto_decoder_big
)
train_network(dnauto_encode_decode_dropout, mse_loss, ❸
➥ train_loader, test_loader=test_loader, epochs=10, device=device)
```

❶ For the input, we usually drop only 5 to 20% of the values.

❷ By default, dropout uses 50% probability to zero out values.

❸ Trains as usual

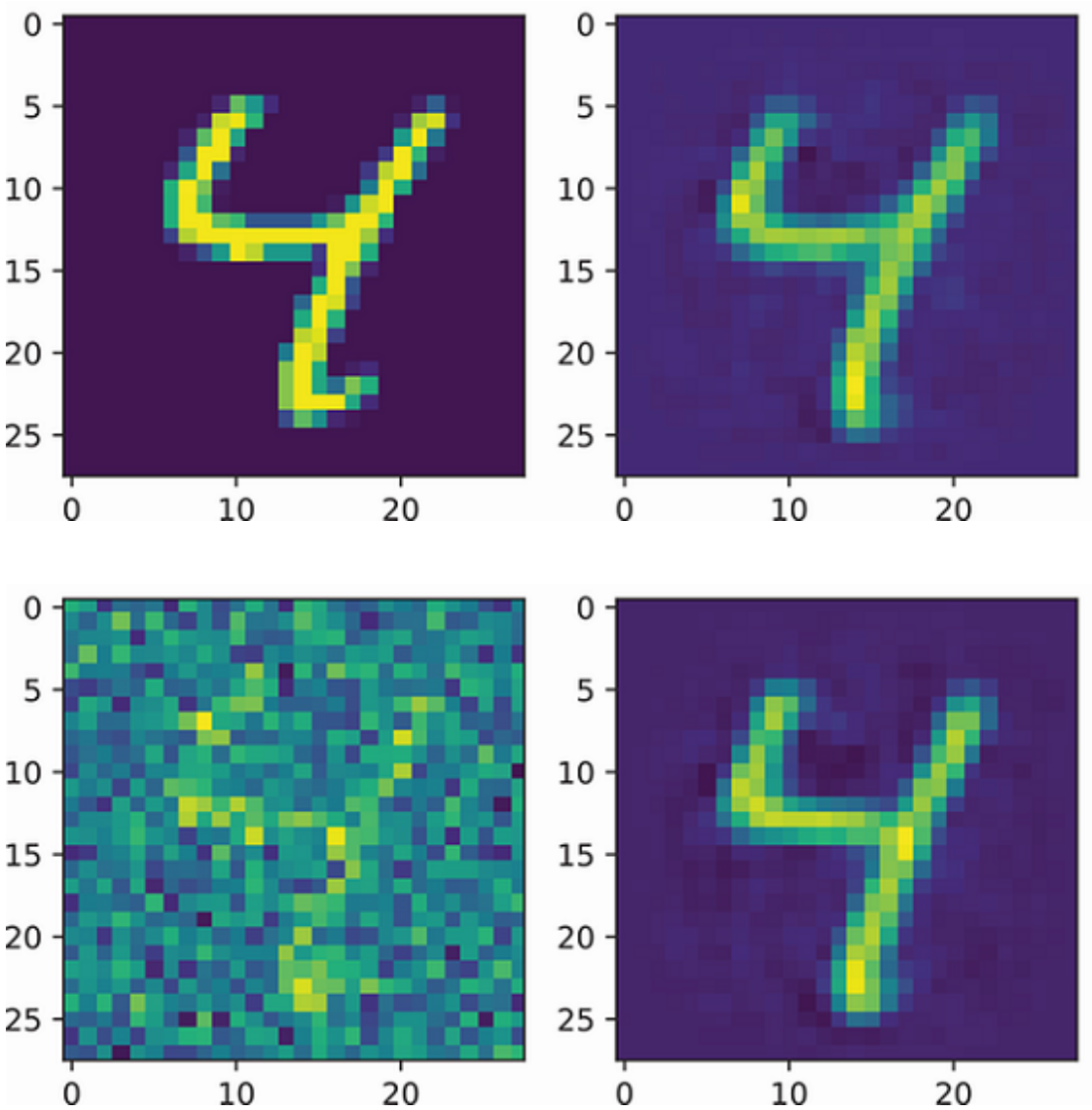
Now that the model is trained, let's apply it to some of the test data. Dropout can encourage a large degree of robustness, which we can show off by applying it to both dropout noise and Gaussian noise. The latter is something the network has never seen before, but that does not stop the autoencoder from faithfully determining an accurate reconstruction:

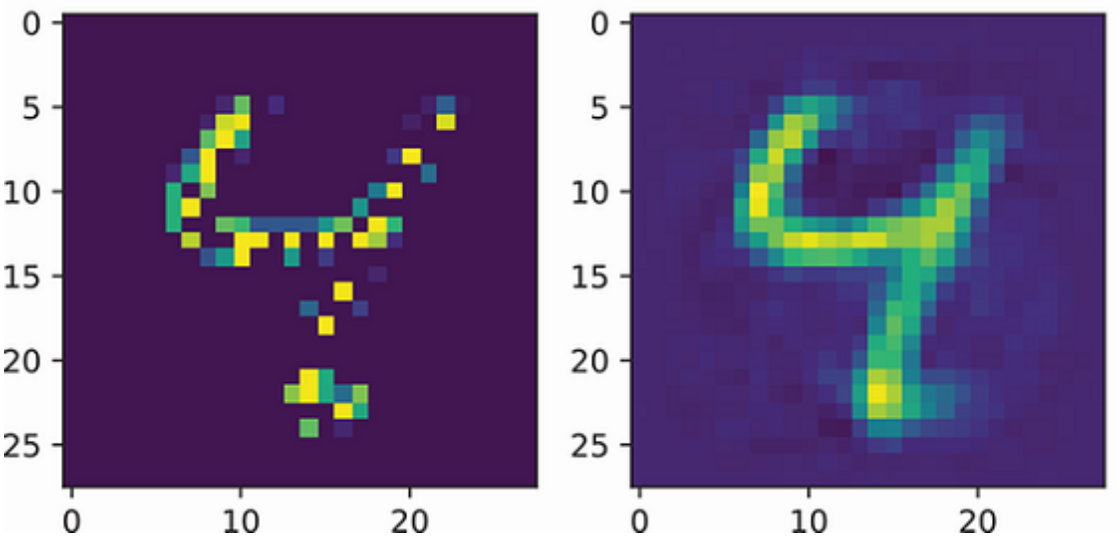
```
showEncodeDecode(dnauto_encode_decode_dropout,
➥ test_data_xy[6][0])                    ❶
showEncodeDecode(dnauto_encode_decode_dropout,
➥ addNoise(test_data_xy[6][0]))          ❷
showEncodeDecode(dnauto_encode_decode_dropout,
➥ nn.Dropout()(test_data_xy[6][0]))      ❸
```

❶ Clean data

❷ Gaussian noise

❸ Dropout noise


0
5
10
15
20
25
0
10
20



The rise and fall of dropout

The origins of dropout began with denoising autoencoders way back in 2008[a] but was applied to only the input. Later it was developed into a more general-purpose regularizer[b] and played a significant role in the re-birth of neural networks as a field and research area.

Like most regularizers, dropout had the goal of improving generalization and reducing overfitting. It was quite good at it and had an attractive and intuitive story about how it worked. For many years, dropout was a critical tool for getting good results, and implementing a network without it was almost impossible. Dropout still works as a regularizer and is useful and used, but it is not ubiquitous as it once was. The tools we have learned thus far, like normalization layers, better optimizers, and residual connections, give us most of the benefits of dropout.

Using dropout is not a *bad* thing, and I'm being hyperbolic by referring to "the fall of dropout." The technique has simply become less popular over

time. My unsubstantiated theory as to why is that first, it is a little slower to train, requiring lots of random numbers and increased memory use when we can get most of its benefits now without those costs. Second, dropout is applied differently at training versus test time. During training, you lose ~50% of your neurons, making the network effectively smaller. But during validation, you get all 100% of your neurons. This can cause the confusing situation where testing performance looks better than training performance because train and test are being evaluated in different ways. (Technically, the same is true of batch normalization, but the inversion isn't as common.) I think people opted for the slightly less expensive and less perplexing results of other methods. That said, dropout is still a great default choice to use as a regularizer for new architectures where you are not sure what does or does not work.

---

[a] P. Vincent, H. Larochelle, Y. Bengio, and P.A. Manzagol, "Extracting and composing robustfeatures with denoising autoencoders," in *Proceedings of the 25th International Conference on Machine Learning*, New York: Association for Computing Machinery, 2008, pp. 1096–1103, **https://doi.org/10.1145/1390156.1390294**.↩

[b] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.↩

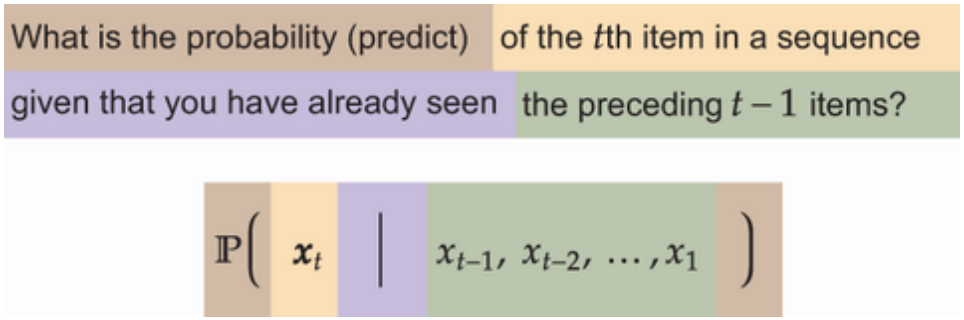
7.5 Autoregressive models for time series and sequences

The autoencoding approach has been very successful for images, signals, and even fully connected models with tabular data. But what if our data

is a sequence problem? Especially if our data is in a language represented by discrete tokens, it's hard to add meaningful noise to things like a letter or word. Instead, we can use an *autoregressive model*, which is an approach specifically designed for time-series problems.

You can use an autoregressive model for basically all the same applications for which you might use an autoencoding one. You can use the representation an autoregressive model learns as the input to another ML algorithm that doesn't understand sequences. For example, you could train an autoregressive model on book reviews of *Inside Deep Learning* and then a clustering algorithm like k-means or HDBSCAN to cluster those reviews.[4] Since these algorithms do not naturally take text as input, the autoregressive model is a nice way to quickly expand the reach of your favorite ML tools.

Let's say you have t steps of your data: $x_1, x_2, \ldots, x_{t-1}, x_t$. The goal of an autoregressive model is to predict x_{t+1} given all the previous items in the sequence. The mathy way to write this would be $\mathbb{P}(x_{t+1}|x_1, x_2, \ldots, x_t)$, which is saying



The autoregressive approach is still a form of self-supervision because the next item in a sequence is a trivial component of having the data in the first place. If you treat the sentence "This is a sentence" as a sequence of characters, you, by definition, know that *T* is the first item, *h* the second, *i* the third, and so on.

Figure 7.6 illustrates how autoregressive models happen at a high level. A sequence-based model is shown in the green blocks and takes in an input x_i. The prediction at the ith step is thus $\hat{x}_i$. We then use the loss function ℓ to compute the loss between the current prediction $\hat{x}_i$ and the *next input* x_{i+1}, $\ell(\hat{x}_i, x_{i+1})$. So for an input with T time steps, we have $T - 1$ loss calculations: the last time step T can't be used as an input because there is no $T + 1$th item to compare it against.

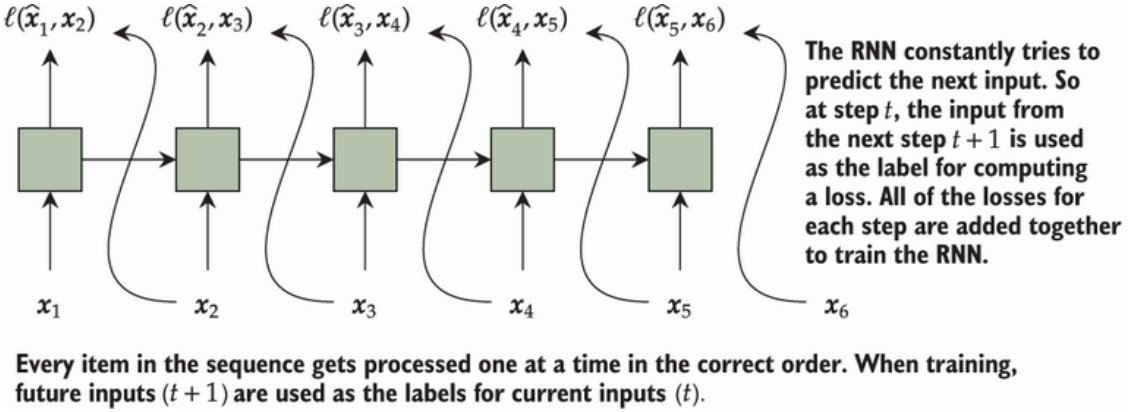


Figure 7.6 Example of an autoregressive setup. The inputs are at the bottom, and the outputs are at the top. For an input x_i, the prediction from the autoregressive model is $\hat{x}_i$, and the *label* $y_i = x_{i+1}$.

You may have guessed from the look of this diagram that we will use a recurrent neural network to implement our autoregressive models. RNNs are great for sequence-based problems like this one. The big change compared to our previous use of RNNs is that we will make a prediction at *every* step, instead of just the *last* step.

A type of autoregressive model popularized by Andrej Karpathy (**http://karpathy.github.io**) is called *char-RNN* (character RNN). This is an autoregressive approach where the inputs/outputs are characters, and we'll show a simple way to implement a char-RNN model on some Shakespeare data.

NOTE While RNNs are an appropriate and common architecture to use for auto- regressive models, bidirectional RNNs are not. This is because an autoregressive model is making predictions about the future. If we used a bidirectional model, we would have information about the future content in the sequence, and knowing the future is cheating! Bi-directional RNNs were useful when we wanted to make a prediction about the

whole sequence, but now that we are making predictions about the input, we need to enforce a no-bidirectional policy to make sure our models do not get to peek at information they should not see.

7.5.1 Implementing the char-RNN autoregressive text model

The first thing we need is our data. Andrej Karpathy has shared some text from Shakespeare online that we will download. There are about 100,000 characters in this text, so we store the data in a variable called `shakespear_100k`. We use this dataset to show the process of training an autoregressive model, as well as its generative capabilities:

```
from io import BytesIO
from zipfile import ZipFile
from urllib.request import urlopen
import re

all_data = [] resp = urlopen(
➥ "https://cs.stanford.edu/people/karpathy/char-rnn/shakespear.txt")
shakespear_100k = resp.read()
shakespear_100k = shakespear_100k.decode('utf-8').lower()
```

Now we will build a vocabulary Σ of all the characters in this dataset. One change you could make is to not use the `lower()` function to convert everything to lowercase. As we are exploring deep learning, these early decisions are important for how our model will eventually be used and how useful it will be. So, you should learn to recognize choices like this *as choices*. I've chosen to use all lowercase data, and as a result, our vocabu-

lary is smaller. This reduces the difficulty of the task but means our model can't learn about capitalization.

Here's the code:

```
vocab2indx = {}                          ❶
for char in shakespear_100k:
    if char not in vocab2indx:           ❷
vocab2indx[char] = len(vocab2indx)       ❸


indx2vocab = {}                          ❹
for k, v in vocab2indx.items():          ❺
    indx2vocab[v] = k
print("Vocab Size: ", len(vocab2indx))
print("Total Characters:", len(shakespear_100k))

Vocab Size: 36
Total Characters: 99993
```

❶ The vocab Σ

❷ Adds every new character to the vocab

❸ Sets the index based on the current vocab size

❹ Useful code to go from the index back to the original characters

❺ Iterates over all key,value pairs and creates a dictionary with the inverse mapping

Next we take a very simple approach to build an autoregressive dataset. We have the original 100,000 characters in one long sequence since they are taken from one of Shakespeare's plays. If we break this sequence into chunks that are sufficiently long, we can almost guarantee that each chunk will contain a few complete sentences. We obtain each chunk by indexing into a position `start` and grabbing a slice of the text `[start:start+chunk_size]`. Since the dataset is autoregressive, our *labels* are the tokens starting one character over. This can be done by grabbing a new slice shifted by one, `[start+1:start+1+chunk_size]`. This is shown in figure 7.7.

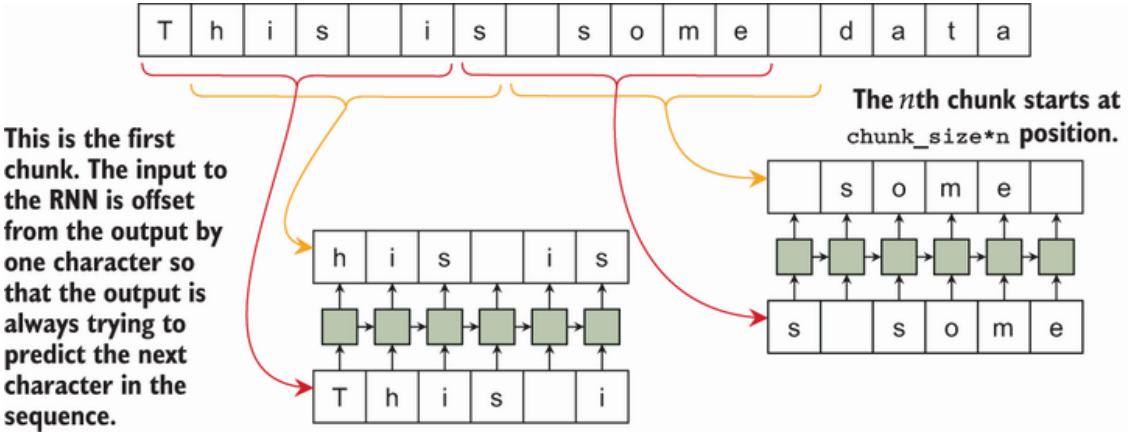


Figure 7.7 Red shows grabbing the input, and yellow the output, using chunks of six characters. This makes it easy to create the dataset for the model where every batch size has the same length, simplifying our code and ensuring maximum GPU utilization (no work done on padded inputs/outputs).

The following code uses this strategy to implement the dataset for autoregressive problems from a large text corpus. We assume the corpus exists as one long string, and it is OK to concatenate multiple files together into one long string since our chunks are smaller than most documents. While we are giving our model the difficulty of having to learn *starting at a ran-*

dom position that is probably part-way into a word, it makes it very easy for us to implement all of the code:

```
class AutoRegressiveDataset(Dataset):
    """
    Creates an autoregressive dataset from one single, long, source
    ➥ sequence by breaking it up into "chunks".
    """

    def __init__(self, large_string, max_chunk=500):
        """
        large_string: the original long source sequence that chunks will
        ➥ be extracted from
        max_chunk: the maximum allowed size of any chunk.
        """
        self.doc = large_string
        self.max_chunk = max_chunk

    def __len__(self):
        return (len(self.doc)-1) // self.max_chunk            ❶

    def __getitem__(self, idx):
        start = idx*self.max_chunk                            ❷

        sub_string = self.doc[start:start+self.max_chunk]     ❸
        x = [vocab2indx[c] for c in sub_string]               ❹
        sub_string = self.doc[start+1:start+self.max_chunk+1] ❺
        y = [vocab2indx[c] for c in sub_string]               ❻
        return torch.tensor(x, dtype=torch.int64), torch.tensor(y,
        ➥ dtype=torch.int64)
```

❶ The number of items is the number of characters divided by chunk size.

❷ Computes the starting position for the idx'th chunk

❸ Grabs the input substring

❹ Converts the substring into integers based on our vocab

❺ Grabs the label substring by shifting over by 1

❻ Converts the label substring into integers based on our vocab

Now comes the tricky part: implementing an autoregressive RNN model. For this we use a gated recurrent unit (GRU) instead of long short-term memory (LSTM), because the code will be a little easier to read since the GRU has only the hidden states $\boldsymbol{h}_t$ and does not have any context states $\boldsymbol{c}_t$. The high-level strategy for our implementation is given in figure 7.8.

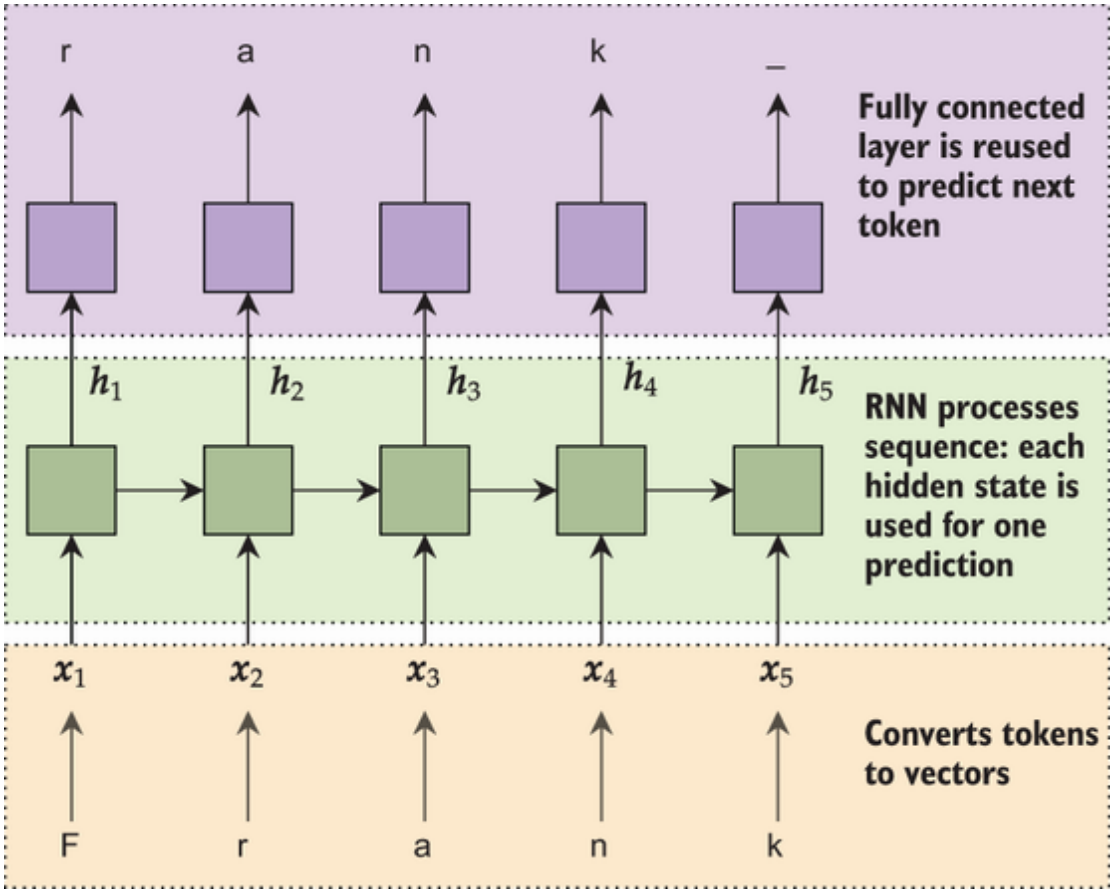


Figure 7.8 Autoregressive RNN design. The input (starting from the bottom) is in yellow, where an `nn.Embedding` layer converts each character into a vector. These vectors are fed into an RNN layer, shown in green, which sequentially processes each character. Then a set of fully connected layers processes each RNN hidden state h_t *independently* (via weight sharing) to make a prediction about the next token.

Defining an autoregressive constructor

Our constructor takes some familiar arguments. We want to know the size of the vocabulary `num_embeddings`, how many dimensions in the embedding layer `embd_size`, the number of neurons in each hidden layer `hidden_size`, and the number of RNN layers `layers=1`:

```
class AutoRegressive(nn.Module):

    def __init__(self, num_embeddings, embd_size, hidden_size, layers=1):
        super(AutoRegressive, self).__init__()
        self.hidden_size = hidden_size
        self.embd = nn.Embedding(num_embeddings, embd_size)
```

Our first major change to our architecture is that we do not use the normal `nn.GRU` module. The normal `nn.RNN`, `nn.LSTM` and `nn.GRU` modules take in *all* the time steps at once and return *all the outputs* at once. You can use these to implement an autoregressive model, but we are instead going to use the `nn.GRUCell` module. The `GRUCell` processes sequences *one item at a time*. This is slower but can make it easier to handle inputs with an unknown and variable length. This approach is summarized in figure 7.9. The `Cell` classes will be useful once we are finished training the model, but I don't want to ruin the surprise—we will come back to *why* we are doing it this way in a moment.

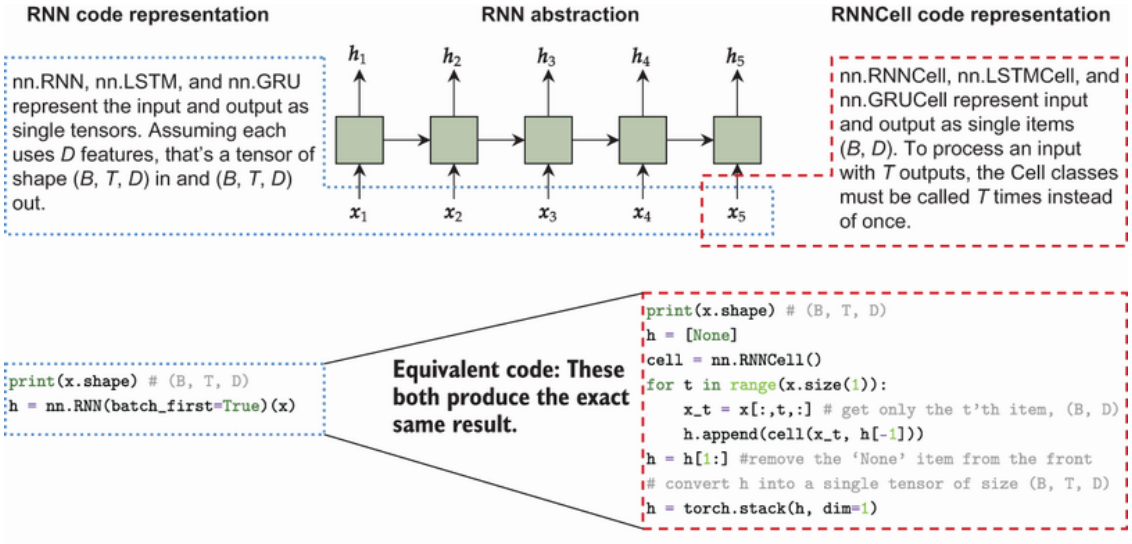


Figure 7.9 Example demonstrating the primary difference between the RNN and cell classes in PyTorch. Left: Normal RNNs process the entire sequence in a single operation, making them faster but requiring all data to be available at once. Right: Cell classes process items one at a time, making them slower but easier to use when you don't have all the inputs already available.

If we want multiple `layers` of an RNN, we have to manually specify and run them ourselves. We can do this by using a `ModuleList` to specify multiple modules in a group. This means our initialization code after `self.embd` looks like this:

```
self.layers = nn.ModuleList(
    [nn.GRUCell(embd_size, hidden_size)] + [nn.GRUCell(hidden_size, hidden_size)
for i in range(layers-1)]) self.norms = nn.ModuleList(
    [nn.LayerNorm(hidden_size) for i in range(layers)])
```

We broke up the specification of `GRUCell` layers into two parts. First is a list of one item for the first layer since it has to go from `embd_size` inputs to `hidden_size` outputs. Second is all remaining layers with `[nn.-GRUCell(hidden_size, hidden_size) for i in range(layers-`

`1)]`, which works because each of these layers has the same input and output size. For fun, I've also included a `LayerNorm` normalization layer for each RNN result.

The last thing we need in our constructor is the purple layers that take in the hidden states h_t and output a prediction for the class. This is done with a small fully connected network:

```
self.pred_class = nn.Sequential(
    nn.Linear(hidden_size, hidden_size),    ❶
    nn.LeakyReLU(),
    nn.LayerNorm(hidden_size),              ❶
    nn.Linear(hidden_size, num_embeddings) ❷
)
```

❶ (B, *, D)

❷ (B, *, D) -> B(B, *, VocabSize)

Notice that we define a component of this module as an entire network. This will help us compartmentalize our design and make our code easier to read. If you ever want to go back and change the subnetwork that goes from hidden RNN states to predictions, you can change just the `pred_class` object, and the rest of the code will work fine.

Implementing the autoregressive forward function

The `forward` function of the module will organize the work done by two other helper functions. First we embed the input tokens into their vector

forms, as this can all be done at once. Because we are using a `GRUCell` class, we need to keep track of the hidden states ourselves. So we use a `initHiddenStates(B)` function that creates the initial hidden states $\boldsymbol{h_0} = \vec{0}$ for each GRU layer. Then we use a `for` loop to grab each of the t items and process them one step at a time using a `step` function that takes the inputs $\boldsymbol{x_t}$ and a list of the GRU hidden states `h_prevs`. The GRU hidden states are stored in a list `last_activations` to get the predictions at every time step. Finally, we can return a single tensor by `stack`ing the results together:

```
  def forward(self, input):                                          ❶
      B = input.size(0)                                              ❷
      T = input.size(1)                                              ❸

      x = self.embd(input)                                           ❹

      h_prevs = self.initHiddenStates(B)                             ❺

      last_activations = []
      for t in range(T):
          x_in = x[:,t,:]                                            ❻
          last_activations.append(self.step(x_in, h_prevs))

  last_activations = torch.stack(last_activations, dim=1)            ❼

  return last_activations
```

❶ Input should be (B, T).

❷ What is the batch size?

❸ What is the maximum number of time steps?

❹ (B, T, D)

❺ Initial hidden states

❻ (B, D)

❼ (B, T, D)

`initHiddenStates` is easy to implement. We can use the `torch.zeros` function to create a tensor of all zero values. We just have the argument `B` for how large the batch is, and then we can grab the `hidden_size` and number of `layers` from the object's members:

```
  def initHiddenStates(self, B):
      """
      Creates an initial hidden state list for the RNN layers.

      B: the batch size for the hidden states.
      """
      return [torch.zeros(B, self.hidden_size, device=device)
          for _ in range(len(self.layers))]
```

The `step` function is a little more involved. First we check the shape of the input, and if it has only one dimension, we assume that we need to embed the token values to make vectors. Then we check the hidden states `h_prevs` and, if they are not provided, initialize them using `initHiddenStates`. These are both good defensive code steps to make sure our function can be versatile and avoid errors:

```
    def step(self, x_in, h_prevs=None):
        """
        x_in: the input for this current time step and has shape (B)
        if the values need to be embedded, and (B, D) if they
        have already been embedded.

        h_prevs: a list of hidden state tensors each with shape
        (B, self.hidden_size) for each layer in the network.
        These contain the current hidden state of the RNN layers
        and will be updated by this call.
        """

        if len(x_in.shape) == 1:                                    ❶
            x_in = self.embd(x_in)                                  ❷

        if h_prevs is None:
            h_prevs = self.initHiddenStates(x_in.shape[0])

        for l in range(len(self.layers)):                           ❸
            h_prev = h_prevs[l]
            h = self.norms[l](self.layers[l](x_in, h_prev))         ❹

            h_prevs[l] = h
            x_in = h
        return self.pred_class(x_in)                                ❺
```

❶ Preps all three arguments to be in the final form. First, (B); we need to embed it.

❷ Now (B, D)

❸ Processes the input

❹ Pushes in the current input with previous hidden states

❺ Makes predictions about the token

After those defensive coding steps, we simply loop through the number of layers and process the results. `x_in` is the input to a layer, which is passed into the current layer `self.layers[l]` and then the normalization layer `self.norms[l]`. After that, we do the minor bookkeeping of storing the new hidden state `h_prevs[l] = h` and setting `x_in = h` so that the next layer has its input ready to process. Once that loop is done, `x_in` has the result from the last RNN layer, so we can feed it directly to the `self.pred_class` object to go from the RNN hidden state to a prediction about the next character.

A shortcut for linear layers over time

You may notice a comment in this code about the tensor shapes with `(B, D)`. This is because `nn.Linear` layers have a special trick that allows them to be applied to multiple inputs *independently* at the same time. We have always used a linear model over a tensor with shape (B,D), and the linear model can take in D inputs and return D' outputs. So we would go from $(B,D) \rightarrow (B,D')$. If we have T items in a sequence, we have a tensor of shape (B,T,D). Applying a linear model to *each* time step naively would require a `for` loop and look like this:

```
def applyLinearLayerOverTime(x):
    results = []                                    ❶
```

```
    B, T, D = x.shape
    for t in range(T):
        results.append(linearLayer(x[:,t,:])      ❷
return torch.stack(results, dim=0).view(B, T, -1) ❸
```

❶ Place to store result for each step

❷ Gets the result for each step

❸ Stacks everything into one tensor and shapes it correctly

That is more code than we would like, *and* it will run slower because of the `for` loop. PyTorch has a simple trick that `nn.Linear` layers are applied to the *last* axis of a tensor *regardless of the number of axes*. That means this whole function can be replaced with just `linearLayer` , and you will get the *exact same result*. That is less code *and* faster. This way, any fully connected network can be used on single time steps or groups of time steps, without having to do anything special. Still, it's good to keep comments like `(B, D)` and `(B, T, D)` so that you can remind yourself *how* you are using your networks.

With our model defined, we are almost finished. Next we quickly create our new `AutoRegressiveDataset` with the `shakespear_100k` data as the input, and we make a data loader using a respectable batch size. We also create our `AutoRegressive` model with 32 dimensions for the embedding, 128 hidden neurons, and 2 GRU layers. We include gradient clipping because RNNs are sensitive to that issue:

```
autoRegData = AutoRegressiveDataset(shakespear_100k, max_chunk=250)
autoReg_loader = DataLoader(autoRegData, batch_size=128, shuffle=True)
```

```
autoReg_model = AutoRegressive(len(vocab2indx), 32, 128, layers=2)
autoReg_model = autoReg_model.to(device)

for p in autoReg_model.parameters():
    p.register_hook(lambda grad: torch.clamp(grad, -2, 2))
```

Implementing an autoregressive loss function

The last thing we need is a loss function ℓ. We are making a prediction at every step, so we want to use the `CrossEntropyLoss` that is appropriate for classification problems. *However*, we have *multiple* losses to compute, one for every time step. We can solve this by writing our own loss function `CrossEntLossTime` that computes the cross-entropy for each step. Similar to our `forward` function, we slice each prediction `x[:,t,:]` and a corresponding label `y[:t]` so that we end up with the standard (B,C) and (B) shapes for the prediction and labels, respectively, and can call `CrossEntropyLoss` directly. Then we add the losses from every time step to get a single total loss to return:

```
def CrossEntLossTime(x, y):
    """
    x: output with shape (B, T, V)
    y: labels with shape (B, T)
    """
    cel = nn.CrossEntropyLoss()

    T = x.size(1)

    loss = 0
```

```
        for t in range(T):                    ❶
            loss += cel(x[:,t,:], y[:,t]) ❷

        return loss
```

❶ For every item in the sequence . . .

❷ ... compute the sum of the prediction errors.

Now we can finally train our autoregressive model. We use our same `train_network` function but pass in our new `CrossEntLossTime` function as the loss function ℓ—and everything just works:

```
train_network(autoReg_model, CrossEntLossTime, autoReg_loader, epochs=100,
➥ device=device)
```

7.5.2 Autoregressive models are generative models

We saved one last detail for the end because it's easier to *see* it than it is to explain. Autoregressive models are not only self-supervised; they also fall into a class known as *generative models*. This means they can *generate* new data that looks like the original data it was trained on. To do this, we switch our model to `eval` mode and create a tensor `sampling` that stores our generated output. Any output generated from a model can be called a *sample*, and the process of generating that sample is called *sampling*, if you want to sound cool (and this is good terminology for you to remember):

```
autoReg_model = autoReg_model.eval()
sampling = torch.zeros((1, 500), dtype=torch.int64, device=device)
```

To sample from an autoregressive model, we usually need to give the model a *seed*. This is some original text that the model is *given*; then the model is asked to make predictions about what comes next. The code for setting a seed is shown next, where "EMILIA:" is our initial seed, as if the character Emilia is about to speak in the play:

```
seed = "EMILIA:".lower()
cur_len = len(seed)
sampling[0,0:cur_len] = torch.tensor([vocab2indx[x] for x in seed])
```

The sampling process of an autoregressive model is shown in figure 7.10. Our seed is passed in as the initial inputs to the model, and we *ignore* the predictions being made. This is because our seed is helping to build the hidden states $\boldsymbol{h}$ of the RNN, which contain information about every previous input. Once we have processed the entire seed, we have no more inputs. After the seed has run out of inputs, we use the *previous* output of the model $\hat{\boldsymbol{x}}_t$ as the *input* for the next time step $t + 1$. This is possible because the autoregressive model has *learned to predict what comes next*. If it does a good job at this, its predictions can be used as inputs, and we end up *generating* new sequences in the process.

When using an already trained autoregressive model (i.e., "testing"), current outputs (t) are used as future inputs ($t+1$). The usage is swapped based on if you are training or testing the autoregressive model.

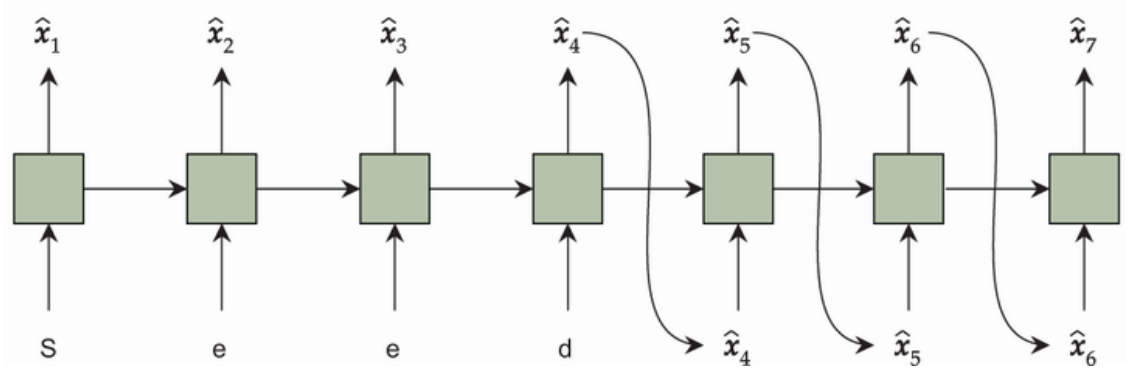


Figure 7.10 A seed is given to the model, and we ignore the predictions being made. Once the seed runs out, we use the predictions at time step t as the input to the next step $t+1$.

But how do we use a prediction as an input? Our model is making a prediction about the probability of seeing *every* different character as the next possible output. But the next input needs to be a *specific* character. This can be done by *sampling* the predictions based on the model's output probabilities. So if the character *a* has a 100% prediction of being next, the model *will* return *a*. If instead we have 80% *a*, 10% *b*, and 10% *c*, we will *probably* select *a* as the next class, but we could also pick *b* or *c*. The following code does just that:

```
for i in tqdm(range(cur_len, sampling.size(1))):
    with torch.no_grad():
        h = autoReg_model(sampling[:,0:i])      ❶
        h = h[:,-1,:]                           ❷
        h = F.softmax(h, dim=1)                 ❸
        next_tokens = torch.multinomial(h, 1)   ❹
        sampling[:,i] = next_tokens             ❺
        cur_len += 1                            ❻
```

❶ Processes all the previous items

❷ Grabs the last time step

❸ Makes probabilities

❹ Samples the next prediction

❺ Sets the next prediction

❻ Increases the length by one

NOTE Just as autoencoders make great embeddings, so do autoregressive models. The hidden state `h` in the previous code can be used as an embedding that summarizes the entire sequence processed thus far. This is a good way to go from word embeddings to sentence or paragraph embeddings.

Now we have a new sequence that we have predicted, but what does it look like? That is why we saved an *inverse mapping* from tokens back to our vocabulary with the `indx2vocab dict`: we can use that to map each integer back to a character, and then `join` them together to create an output. The following code converts our generative sample back into text we can read:

```
s = [indx2vocab[x] for x in sampling.cpu().numpy().flatten()]
print("".join(s))

emilia:
to hen the words tractass of nan wand,
```

```
no bear to the groung, iftink sand'd sack,
i will ciscling
bronino:
if this, so you you may well you and surck, of wife where sooner you.

corforesrale:
where here of his not but rost lighter'd therefore latien ever
un'd
but master your brutures warry:
why,
thou do i mus shooth and,
rity see! more mill of cirfer put,
and her me harrof of that thy restration stucied the bear:
and quicutiand courth, for sillaniages:
so lobate thy trues not very repist
```

You should notice a few things about the output of our generation. While it kind of looks Shakespearian, it quickly devolves. This is because with each step of the data, we get further from *real* data, and our model *will* make unrealistic choices that thus become errors and negatively impact future predictions. So the longer we generate, the lower the quality will become.

7.5.3 Changing samples with temperature

The model rarely gives a probability of *zero* for any token, which means we will eventually select an incorrect or unrealistic next token. If you are 99% sure that the next character should be *a*, why give the model the 1% opportunity to pick something that is likely wrong? To encourage the model to go with the most likely predictions, we can add a *temperature* to the generation process. The `temperature` is a scalar that we divide the

model's prediction by before computing the `softmax` to make probabilities. This is shown in figure 7.11, where you can push the temperature to extreme values like infinity or zero. Something with infinite temperature will result in uniformly random behavior (not what we want), and something with zero temperature is frozen solid and returns the same (most likely) thing over and over again (also not what we want).

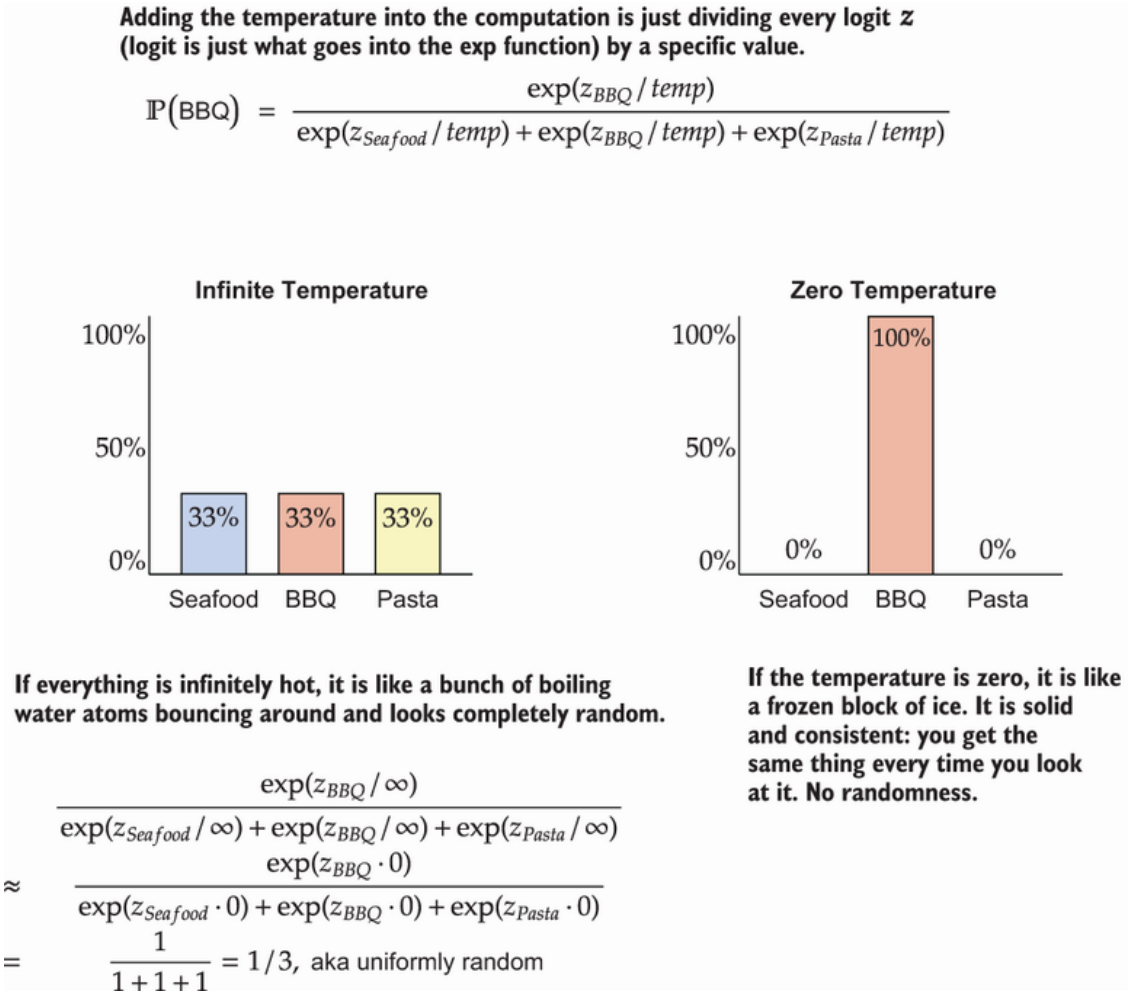


Figure 7.11 What food is Edward going to eat? If the temperature is set very high, Edward's selection will be random, regardless of what the initial probabilities are. If the temperature is at or near zero, Edward will always pick BBQ because it is more likely than any other option. More temperature = more randomness, and no negative temperatures are allowed.

Instead of using these extremes, we can instead focus on adding a small effect by making the temperature a value slightly larger or smaller than `1.0`. The default value of `temperature=1.0` causes no change in the probabilities and so is the same as what we already did: compute the original probabilities and sample a prediction based on those probabilities. But if you use a `temperature < 1`, then items that originally had a larger chance of being selected (like BBQ) will get an even bigger advantage and increase their probability (think, "the rich get richer"). If we make `temperature > 1`, we end up giving lower-probability items more of a chance to be selected, at the cost of the originally higher probabilities. The effect of temperature is summarized in figure 7.12 with an example of picking my next meal.

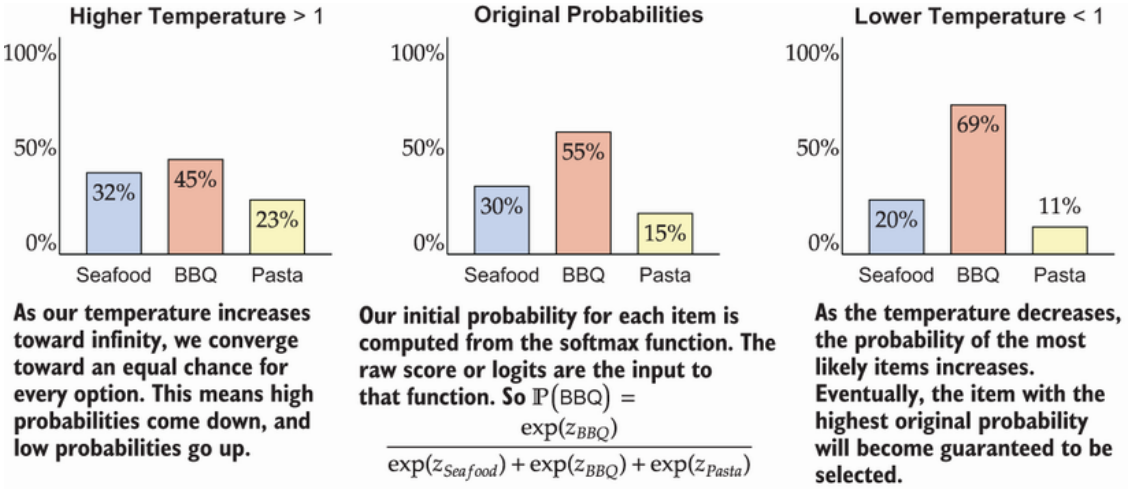


Figure 7.12 Example of how temperature impacts the probability of selecting my next meal. I'm most likely to eat BBQ by default because it is delicious. Raising the temperature encourages diversity, eventually selecting each item at random if we went to the extreme minimum temperature (zero). Lowering temperature reduces diversity, eventually selecting only the most likely original item at the extreme highest possible temperature (toward infinity).

In practice, a value of 0.75 is a good default[5] (I usually see 0.65 at the low end and 0.8 at the high end) because it maintains diversity but avoids

picking items that were more unlikely originally (e.g., BBQ is my favorite food group, but some diversity is good for you and is more realistic). The following code adds the temperature to the sampling process:

```
cur_len = len(seed)
temperature = 0.75                                ❶
for i in tqdm(range(cur_len, sampling.size(1))):
    with torch.no_grad():
        h = autoReg_model(sampling[:,0:i])
        h = h[:,-1,:]                             ❷
        h = F.softmax(h/temperature, dim=1)       ❸
        next_tokens = torch.multinomial(h, 1)
        sampling[:,i] = next_tokens

        cur_len += 1
```

❶ Primary addition: controls the temperature and our sampling behavior

❷ Grabs the last time step

❸ Makes probabilities

Why is temperature called temperature?

The analogy you should go with is a pot of water, and the temperature is literally the temperature of the water. If the temperature is very high, the water begins to boil, and the water's atoms are in random positions as they bounce around. As the temperature decreases, the water begins to freeze, and the atoms stay still in an organized pattern. High temperature

= chaotic (read, random), and low temperature is still (read, no randomness).

If we print out our predictions, we should see something that feels a little more reasonable. It is not perfect, and a bigger model with more epochs of training can help improve that. But adding a `temperature` is a common trick to help control the generation process:

```
s = [indx2vocab[x] for x in sampling.cpu().numpy().flatten()]
print("".join(s))

emilia:
therefore good i am to she prainns siefurers.

king ford:
beor come, sir, i chaed
to me, the was the strong arl and the boy fear mabber lord,,
coull some a clock dightle eyes, agaary must her was flord but the hear fall
the cousion a tarm:
i am a varstiend the her apper the service no that you shall give yet somantion,
and lord, and commind cure, i why had they helbook.

mark ars:
who her throw true in go speect proves of the wrong and further gooland, before
but i am so are berether, i
```

So you may be thinking, why not lower the temperature even further? Should we not *always* want to go with the most likely prediction? This gets to some deep issues about the difficulty of evaluating generative and autoregressive models, especially when the input is something like human text, which is not all that predictable to begin with. If I told you I

could *perfectly* predict what someone was going to say, you would likely be incredulous. How could that be possible? We should always apply that same standard to our model. If we *always* select the most likely prediction, we are assuming the model can *perfectly* predict what a person would say next. We can try that by setting the temperature to a very low value like 0.05:

```
cur_len = len(seed)
temperature = 0.05                              ❶
for i in tqdm(range(cur_len, sampling.size(1))):
    with torch.no_grad():
        h = autoReg_model(sampling[:,0:i])
        h = h[:,-1,:]                           ❷
        h = F.softmax(h/temperature, dim=1)     ❸
        next_tokens = torch.multinomial(h, 1)
        sampling[:,i] = next_tokens

        cur_len += 1
s = [indx2vocab[x] for x in sampling.cpu().numpy().flatten()]
print("".join(s))

emilia:
i will straight the shall shall shall be the with the shall shall shall shall
be the with the shall be the with the shall shall shall be the shall shall
shall she shall shall shall shall shall be the with the shall shall shall
shall be the shall be the shall shall shall shall shall shall shall shall be
the prove the will so see and the shall be the will the shall shall shall
shall shall shall be the with the shall shall shall shall be the shall be the
with the shall shall shall be the wi
```

❶ Very low temp: will almost always pick the most likely items

❷ Grabs the last time step

❸ Makes probabilities

The model has become *very* repetitive. This is what usually happens when you select the most likely token as the next one; the model devolves to selecting a sequence of the most common words/tokens over and over and over again.

Hot alternatives to temperature

Adjusting the temperature is but one of many possible techniques for selecting the generated output in a more realistic-looking manner. Each has pros and cons, but you should be aware of three alternatives: beam search, top-k sampling, and nucleus sampling. The fine folks at Hugging Face have a good blog post that introduces these at a high level (**https://huggingface.co/blog/how-to-generate**).

7.5.4 Faster sampling

You may have noticed that the sampling process takes a surprisingly long time—about 45 to 50 seconds to generate 500 characters—yet we could train over 100,000 characters in just seconds per epoch. This is because each time we make a prediction, we needed to refeed the entire generated sequence far back into the model to get the next prediction. Using Big O notation means we are doing $O(n^2)$ work to generate a sequence of $O(n)$ length.

The `GRUCell` that processes sequences one step at a time makes it easy for us to solve this problem. We break our `for` loop up into two parts, each using the `step` function directly instead of the `forward` function of the module. The first loop pushes the seed into the model, updating an explicitly created set of hidden states `h_prevs` . After that, we can write a new loop that generates the new content and calls the `step` function to update the model after sampling the next character. This process is shown in the following code:

```
seed = "EMILIA:".lower()                                    ❶
cur_len = len(seed)

sampling = torch.zeros((1, 500), dtype=torch.int64, device=device)
sampling[0,0:cur_len] = torch.tensor([vocab2indx[x] for x in seed])

temperature = 0.75                                          ❷
with torch.no_grad():
    h_prevs = autoReg_model.initHiddenStates(1)             ❸
    for i in range(0, cur_len):                             ❹
        h = autoReg_model.step(sampling[:,i], h_prevs=h_prevs)
    for i in tqdm(range(cur_len, sampling.size(1))):        ❺
        h = F.softmax(h/temperature, dim=1)                 ❻
        next_tokens = torch.multinomial(h, 1)
        sampling[:,i] = next_tokens
        cur_len += 1 h = autoReg_model.step(sampling[:,i],
        h_prevs=h_prevs)                                    ❼
```

❶ Sets up our seed and the location to store the generated content

❷ Picks a temperature

❸ Initializes the hidden state to avoid redundant work

❹ Pushes the seed through

❺ Generates new text one character at a time

❻ Makes probabilities

❼ Now pushes only the new sample into the model

This new code runs in less than a second. This is much faster, and it gets faster the longer the sequence we want to generate, because it has a better Big O complexity of $O(n)$ to generate $O(n)$ tokens. Next we print out the generated result, which you can see is qualitatively the same as before:

```
s = [indx2vocab[x] for x in sampling.cpu().numpy().flatten()]
print("".join(s))

emilia:
a-to her hand by hath the stake my pouse of we to more should there had the
would break fot his good, and me in deserved
to not days all the wead his to king; her fair the bear so word blatter with
my hath thy hamber--

king dige:
it the recuse.

mark:
wey, he to o hath a griec, and could would there you honour fail;
have at i would straigh his boy:
coursiener:
```

```
and to refore so marry fords like i seep a party. or thee your honour great
way we the may her with all the more ampiled my porn
```

With that, you now know all the basics of autoencoders and autoregressive models, two related approaches that share the same fundamental idea: use the input data as the target output. These can be especially powerful techniques to mimic/replace expensive simulations (have the network learn to predict what happens next), clean up noisy data (inject realistic noise and learn how to remove it), and generally train useful models without any labeled data. The latter note will become important in chapter 13 when you learn how to do transfer learning, a technique allowing you to use unlabeled data to improve results on a smaller labeled dataset.

Exercises

Share and discuss your solutions on the Manning online platform at Inside Deep Learning Exercises (**https://liveproject.manning.com/project/945**). Once you submit your own answers, you will be able to see the solutions submitted by other readers, and see which ones the author judges to be the best.

1. Create a new version of the MNIST dataset that does not contain the numbers 9 and 5, and train one of the autoencoders on this dataset. Then run the autoencoder on the test dataset, and record the average error (MSE) for each of the 10 classes. Do you see any patterns in the results, and can the autoencoder identify 9 and 5 as outliers?

2. Train the bottleneck-style autoencoder with a target size $D' = 64$ dimensions. Then use k-means (**https://scikit-learn.org/stable/modules/clustering.html#k-means**)to create $k = 10$ clusters on the original version of MNIST and the version encoded using $D' = 64$ dimensions. Use the homogeneity score from scikit-learn (**http://mng.bz/nYQV**) to evaluate these clusters. Which method does best:k-means on the original images or k-means on the encoded representations?
3. Use the denoising approach to implement a denoising convolutional network. This can be done by not having any pooling operations so that the input stays the same size.
4. Sometimes people train deep autoencoders with weight sharing between the encoder and decoder. Try implementing a deep bottleneck autoencoder that uses the `TransposeLinear` layer for all of the decoder's layers. Compare the weight shared versus the non-weight shared network when you have only n=1,024, 8,192, 32,768, and all 60,000 samples from MNIST.
5. **Challenging:** Train an asymmetric denoising autoencoder for MNIST where the encoder is a fully connected network and the decoder is a convolutional network. *Hint:* You will need to end the encoder with a `View` layer that changes the shape from (B,D) to $(B,C,28,28)$, where D is the number of neurons in the last `nn.LinearLayer` of the encoder and $D = C \cdot 28 \cdot 28$. Do the results of this network look better or worse than the fully connected network in the chapter, and how do you think intermixing architectures impacts that result?

6. **Challenging:** Reshape the MNIST dataset as a sequence of pixels, and train an autoregressive model over the pixels. This requires using real-valued inputs and outputs, so you will not use an `nn.Embedding` layer, and you will need to switch to the MSE loss function. After training, try generating multiple digits from this autoregressive pixel model.
7. Convert the `GRUCell`s in the autoregressive model to `LSTMCell`s, and train a new model. Which one do you think has better generated output?
8. The `AutoRegressiveDataset` can start an input in the middle of a sentence since it naively grabs subsequences of the input. Write a new version that will only select the start of a sequence at the start of a new line (i.e., after a carriage return '\n') and then returns the next `max_chunk` characters (it's OK if there is some overlap between chunks). Train a model on this new version of the dataset. Do you think it changes the characteristics of the generated output?
9. After training your autoregressive model on sentences, use the `LastTimeStep` class to extract a feature vector representing each sentence. Then feed these vectors into your favorite clustering algorithm and see if you can find any groups of similar styles or types of sentences. *Note:* You may want to sub-sample a smaller number of sentences to make your clustering algorithm run faster.

Summary

- Self-supervision is a means of training a neural network by using parts of the input *as the labels* we are trying to predict.

- Self-supervision is considered unsupervised because it can be applied to any data and does not require any kind of process or human to manually label the data.
- Autoencoding is one of the most popular forms of self-supervision. It works by having the network predict the input as the output but somehow constrains the network so that it can't naively return the original input.
- Two popular constraints are a bottleneck design that forces the dimension to shrink before expanding back out and a denoising approach where the input is altered before being given to the network, but the network still must predict the unaltered output.
- If we have a sequence problem, we can instead use autoregressive models, which look at every previous input in a sequence to predict the next input.
- Autoregressive approaches have the benefit of being *generative*, which means we can create synthetic data from the model!

---

[1] *Latent* generally means something that isn't (easily) visible but does exist and/or can be enhanced. For example, a latent infection exists but has no symptoms, so you can't see it. But untreated, it will emerge and become more visible. *Latent* in our context is fortunately not as morbid.↩

[2] Technically, that means we aren't learning the PCA algorithm—but we are learning something *very* similar to it. Close enough is good enough.↩

[3] E. Raff, "Neural fingerprint enhancement," in *17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 118–124, https://doi.org/10.1109/ICMLA.2018.00025.↩

[4] You should look up “topic modeling” algorithms if you want to maximize your results with text data. There are deep topic models, but they are a bit beyond what we cover in this book. That said, I have seen people be reasonably successful with this autoregressive approach, which is more flexible with new situations and types of data than topic models are.↩

[5] A good default for text generation, that is. Other tasks may need you to play around with the temperature to select a better value.↩