Chapter 2. Fundamentals

In Chapter 1, I described the major conceptual building block for understanding deep learning: nested, continuous, differentiable functions. I showed how to represent these functions as computational graphs, with each node in a graph representing a single, simple function. In particular, I demonstrated that such a representation showed easily how to calculate the derivative of the output of the nested function with respect to its input: we simply take the derivatives of all the constituent functions, evaluate these derivatives at the input that these functions received, and then multiply all of the results together; this will result in a correct derivative for the nested function because of the chain rule. I illustrated that this does in fact work with some simple examples, with functions that took NumPy's `ndarray`s as inputs and produced `ndarray`s as outputs.

I showed that this method of computing derivatives works even when the function takes in multiple `ndarray`s as inputs and combines them via a *matrix multiplication* operation, which, unlike the other operations we saw, changes the shape of its inputs. Specifically, if one input to this operation—call the input X—is a B × N `ndarray`, and another input to this operation, W, is an N × M `ndarray`, then its output P is a B × M `ndarray`. While it isn't clear what the derivative of such an operation would be, I showed that when a matrix multiplication $v(X, W)$ is included as a "constituent operation" in a nested function, we can still use a simple expression *in place of* its derivative to compute the derivatives of its inputs: specifically, the role of $\frac{\partial v}{\partial u}(W)$ can be filled by X^T, and the role of $\frac{\partial v}{\partial u}(X)$ can be played by W^T.

In this chapter, we'll start translating these concepts into real-world applications, Specifically, we will:

1. Express linear regression in terms of these building blocks
2. Show that the reasoning around derivatives that we did in Chapter 1 allows us to train this linear regression model
3. Extend this model (still using our building blocks) to a one-layer neural network

Then, in Chapter 3, it will be straightforward to use these same building blocks to

Before we dive into all this, though, let's give an overview of *supervised learning*, the subset of machine learning that we'll focus on as we see how to use neural networks to solve problems.

Supervised Learning Overview

At a high level, machine learning can be described as building algorithms that can uncover or "learn" *relationships* in data; supervised learning can be described as the subset of machine learning dedicated to finding relationships *between characteristics of the data that have already been measured*.[1]

In this chapter, we'll deal with a typical supervised learning problem that you might encounter in the real world: finding the relationship between characteristics of a house and the value of the house. Clearly, there is some relationship between characteristics such as the number of rooms, the square footage, or the proximity to schools and how desirable a house is to live in or own. At a high level, the aim of supervised learning is to uncover these relationships, given that we've *already measured* these characteristics.

By "measure," I mean that each characteristic has been defined precisely and represented as a number. Many characteristics of a house, such as the number of bedrooms, the square footage, and so on, naturally lend themselves to being represented as numbers, but if we had other, different kinds of information, such as natural language descriptions of the house's neighborhood from TripAdvisor, this part of the problem would be much less straightforward, and doing the translation of this less-structured data into numbers in a reasonable way could make or break our ability to uncover relationships. In addition, for any concept that is ambiguously defined, such as the value of a house, we simply have to pick a single number to describe it; here, an obvious choice is to use the price of the house.[2]

Once we've translated our "characteristics" into numbers, we have to decide what structure to use to represent these numbers. One that is nearly universal across machine learning and turns out to make computations easy is to represent each set of numbers for a single observation—for example, a single house—as a *row* of data, and then stack these rows on top of each other to form "batches" of data that will get fed into our models as two-dimensional `ndarray`s. Our models will then return predictions as output `ndarray`s with each prediction in a row, similarly stacked on top of each other, with one prediction for each observation in the batch.

Now for some definitions: we say that the length of each row in this `ndarray` is the number of *features* of our data. In general, a single characteristic can map to many features, a classic example being a characteristic that describes our data as belonging to one of several *categories*, such as being a red brick house, a tan brick house, or a slate house;[3] in this specific case we might describe this single characteristic with three features. The process of mapping what we informally think of as characteristics of our observations into features is called *feature engineering*. I won't spend much time discussing this process in this book; indeed, in this chapter we'll deal with a problem in which we have 13 characteristics of each observation, and we simply represent each characteristic with a single numeric feature.

I said that the goal of supervised learning is ultimately to uncover relationships between characteristics of data. In practice, we do this by choosing one characteristic that we want to predict from the others; we call this characteristic our *target*. The choice of which characteristic to use as the target is completely arbitrary and depends on the problem you are trying to solve. For example, if your goal is just to *describe* the relationship between the prices of houses and the number of rooms they have, you could do this by training a model with the prices of houses as the target and the number of rooms as a feature, or vice versa; either way, the resulting model will indeed contain a description of the relationship between these two characteristics, allowing you to say, for example, a higher number of rooms in a house is associated with higher prices. On the other hand,

available, you have to choose the price as your target, so that you can ultimately feed the other information into your model once it is trained.

Figure 2-1 shows this hierarchy of descriptions of supervised learning, from the highest-level description of finding relationships in data, to the lowest level of quantifying those relationships by training models to uncover numerical representations between the features and the target.

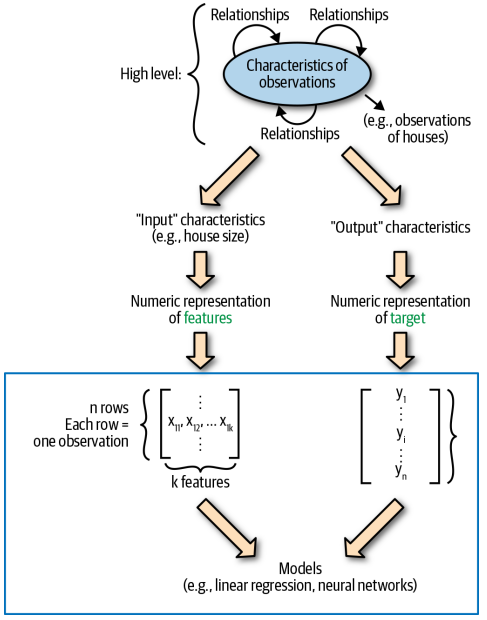


Figure 2-1. Supervised learning overview

As mentioned, we'll spend almost all our time on the level highlighted at the bottom of Figure 2-1; nevertheless, in many problems, getting the parts at the top correct—collecting the right data, defining the problem you are trying to solve, and doing feature engineering—is much harder than the actual modeling. Still, since this book is focused on modeling—specifically, on understanding how deep learning models work—let's return to that subject.

Supervised Learning Models

Now we know at a high level what supervised learning models are trying to do—and as I alluded to earlier in the chapter, such models are just nested, mathematical functions. We spent the last chapter seeing how to represent such functions in terms of diagrams, math, and code, so now I can state the goal of supervised learning more precisely in terms of both math and code (I'll show plenty of diagrams later): the goal is to *find* (a mathematical function) / (a function that takes an `ndarray` as input and produces an `ndarray` as output) that can (map characteristics of observations to the target) / (given an input `ndarray` containing the features we created, produce an output `ndarray` whose values are "close to" the `ndarray` containing the target).

Specifically, our data will be represented in a matrix X with n rows, each of which represents an observation with k features, all of which are numbers. Each row observation will be a vector, as in $x_i = [x_{i1} \quad x_{i2} \quad x_{i3} \quad \dots \quad x_{ik}]$, and these observations will be stacked on top of one another to form a batch. For example, a batch of size 3 would look like:

For each batch of observations, we will have a corresponding batch of *targets*, each element of which is the target number for the corresponding observation. We can represent these in a one-dimensional vector:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

In terms of these arrays, our goal with supervised learning will be to use the tools I described in the last chapter to build a function that can take as input batches of observations with the structure of X_{batch} and produce vectors of values p_i—which we'll interpret as "predictions"—that (for data in our particular dataset X, at least) are "close to the target values" y_i for some reasonable measure of closeness.

Finally, we are ready to make all of this concrete and start building our first model for a real-world dataset. We'll start with a straightforward model—*linear regression*—and show how to express it in terms of the building blocks from the prior chapter.

Linear Regression

Linear regression is often shown as:

$$y_i = \beta_0 + \beta_1 \times x_1 + ... + \beta_n \times x_k + \epsilon$$

This representation describes mathematically our belief that the numeric value of each target is a linear combination of the k features of X, plus the β_0 term to adjust the "baseline" value of the prediction (specifically, the prediction that will be made when the value of all of the features is 0).

This, of course, doesn't give us much insight into how we would code this up so that we could "train" such a model. To do that, we have to translate this model into the language of the functions we saw in Chapter 1; the best place to start is with a diagram.

Linear Regression: A Diagram

How can we represent linear regression as a computational graph? We *could* break it down all the way to the individual elements, with each x_i being multiplied by another element w_i and then the results being added together, as in Figure 2-2.

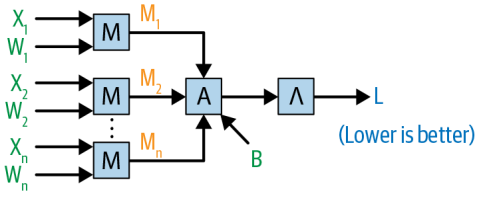


Figure 2-2. The operations of a linear regression shown at the level of individual multiplications and additions

But again, as we saw in Chapter 1, if we can represent these operations as just a matrix multiplication, we'll be able to write the function more concisely while still being able to correctly calculate the derivative of the output with respect to the input, which will allow us to train the model.

How can we do this? First, let's handle the simpler scenario in which we don't have an intercept term (β_0, shown previously). Note that we can represent the

$$W = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix}$$

Our prediction would then simply be:

$$p_i = x_i \times W = w_1 \times x_{i1} + w_2 \times x_{i2} + ... + w_k \times x_{ik}$$

So, we can represent "generating the predictions" for a linear regression using a single operation: the dot product.

Furthermore, when we want to make predictions using linear regression with a batch of observations, we can use another, single operation: the matrix multiplication. If we have a batch of size 3, for example:

$$X_{batch} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & ... & x_{1k} \\ x_{21} & x_{22} & x_{23} & ... & x_{2k} \\ x_{31} & x_{32} & x_{33} & ... & x_{3k} \end{bmatrix}$$

then performing the *matrix multiplication* of this batch X_{batch} with W gives a vector of predictions for the batch, as desired:

$$p_{batch} = X_{batch} \times W = \begin{bmatrix} x_{11} & x_{12} & x_{13} & ... & x_{1k} \\ x_{21} & x_{22} & x_{23} & ... & x_{2k} \\ x_{31} & x_{32} & x_{33} & ... & x_{3k} \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix} = \begin{bmatrix} x_{11} \times w_1 + x_{12} \times w_2 + x_{13} \times w_3 + ... + x_{1k} \times w_k \\ x_{21} \times w_1 + x_{22} \times w_2 + x_{23} \times w_3 + ... + x_{2k} \times w_k \\ x_{31} \times w_1 + x_{32} \times w_2 + x_{33} \times w_3 + ... + x_{3k} \times w_k \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

So generating predictions for a batch of observations in a linear regression can be done with a matrix multiplication. Next, I'll show how to use this fact, along with the reasoning about derivatives from the prior chapter, to train this model.

"TRAINING" THIS MODEL

What does it mean to "train" a model? At a high level, models[4] take in data, combine them with *parameters* in some way, and produce predictions. For example, the linear regression model shown earlier takes in data X and parameters W and produces the predictions p_{batch} using a matrix multiplication:

$$p_{batch} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

To train our model, however, we need another crucial piece of information: whether or not these predictions are good. To learn this, we bring in the vector of *targets* y_{batch} associated with the batch of observations X_{batch} fed into the function, and we compute a *single number* that is a function of y_{batch} and p_{batch} and that represents the model's "penalty" for making the predictions that it did. A reasonable choice is *mean squared error*, which is simply the average squared value that our model's predictions "missed" by:

$$MSE(p_{batch}, y_{batch}) = MSE\left(\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}\right) = \frac{(y_1 - p_1)^2 + (y_2 - p_2)^2 + (y_3 - p_3)^2}{3}$$

Getting to this number, which we can call L, is key: once we have it, we can use all the techniques we saw in Chapter 1 to compute the *gradient* of this number with respect to each element of W. Then *we can use these derivatives to update each element of W in the direction that would cause L to decrease*. Repeating this procedure many times, we hope, will "train" our model; in this chapter, we'll see that this can indeed work in practice. To see clearly how to compute these gradients, we'll complete the process of representing linear regression as a computational graph.

Figure 2-3 shows how to represent linear regression in terms of the diagrams from the last chapter.

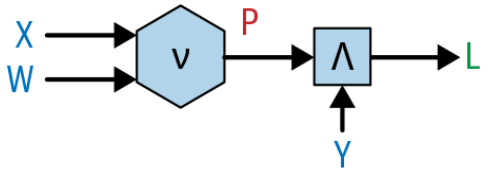


Figure 2-3. The linear regression equations expressed as a computational graph—the dark blue letters are the data inputs to the function, and the light blue W denotes the weights

Finally, to reinforce that we're still representing a nested mathematical function with this diagram, we could represent the loss value L that we ultimately compute as:

$$L = \Lambda((\nu(X,W),Y)$$

Adding in the Intercept

Representing models as diagrams shows us conceptually how we can add an intercept to the model. We simply add an extra step at the end that involves adding a "bias," as shown in Figure 2-4.

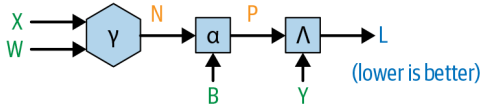


Figure 2-4. The computational graph of linear regression, with the addition of a bias term at the end

Here, though, we should reason mathematically about what is going on before moving on to the code; with the bias added, each element of our model's prediction p_i will be the dot product described earlier with the quantity b added to it:

$$p_{batch\_with\_bias} = x_i \mathrm{dot} W + b = \begin{bmatrix} x_{11} \times w_1 + x_{12} \times w_2 + x_{13} \times w_3 + \ldots + & x_{1k} \times w_k + b \\ x_{21} \times w_1 + x_{22} \times w_2 + x_{23} \times w_3 + \ldots + & x_{2k} \times w_k + b \\ x_{31} \times w_1 + x_{32} \times w_2 + x_{33} \times w_3 + \ldots + & x_{3k} \times w_k + b \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Note that because the intercept in linear regression should be just a single number rather than being different for each observation, the *same number* should get added to each observation of the input to the bias operation that is passed in; we'll discuss what this means for computing the derivatives in a later section of this chapter.

Linear Regression: The Code

We'll now tie things together and code up the function that makes predictions and computes losses given batches of observations X_{batch} and their corresponding targets y_{batch}. Recall that computing derivatives for nested functions using the chain rule involves two sets of steps: first, we perform a "forward pass," passing the input successively forward through a series of operations and saving the

The following code does this, saving the quantities computed on the forward pass in a dictionary; furthermore, to differentiate between the quantities computed on the forward pass and the parameters themselves (which we'll also need for the backward pass), our function will expect to receive a dictionary containing the parameters:

```
def forward_linear_regression(X_batch: ndarray,
                              y_batch: ndarray,
                              weights: Dict[str, ndarray])
                              -> Tuple[float, Dict[str, ndarray]]:
    '''
    Forward pass for the step-by-step linear regression.
    '''
    # assert batch sizes of X and y are equal
    assert X_batch.shape[0] == y_batch.shape[0]

    # assert that matrix multiplication can work
    assert X_batch.shape[1] == weights['W'].shape[0]

    # assert that B is simply a 1x1 ndarray
    assert weights['B'].shape[0] == weights['B'].shape[1] == 1

    # compute the operations on the forward pass
    N = np.dot(X_batch, weights['W'])

    P = N + weights['B']

    loss = np.mean(np.power(y_batch - P, 2))

    # save the information computed on the forward pass
    forward_info: Dict[str, ndarray] = {}
    forward_info['X'] = X_batch
    forward_info['N'] = N
    forward_info['P'] = P
    forward_info['y'] = y_batch

    return loss, forward_info
```

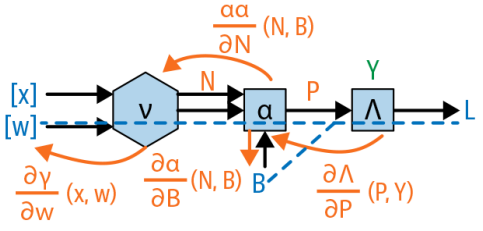
Now we have all the pieces in place to start "training" this model. Next, we'll cover exactly what this means and how we'll do it.

Training the Model

We are now going to use all the tools we learned in the last chapter to compute $\frac{\partial L}{\partial w_i}$ for every w_i in W, as well as $\frac{\partial L}{\partial b}$. How? Well, since the "forward pass" of this function was passing the input through a series of nested functions, the backward pass will simply involve computing the partial derivatives of each function, evaluating those derivatives at the functions' inputs, and multiplying them together—and even though a matrix multiplication is involved, we'll be able to handle this using the reasoning we covered in the last chapter.

Calculating the Gradients: A Diagram

Conceptually, we want something like what is depicted in Figure 2-5.



We simply step backward, computing the derivative of each constituent function and evaluating those derivatives at the inputs that those functions received on the forward pass, and then multiplying these derivatives together at the end. This is straightforward enough, so let's get into the details.

Calculating the Gradients: The Math (and Some Code)

From Figure 2-5, we can see that the derivative product that we ultimately want to compute is:

$$\frac{\partial \Lambda}{\partial P}(P,Y) \times \frac{\partial \alpha}{\partial N}(N,B) \times \frac{\partial \nu}{\partial W}(X,W)$$

There are three components here; let's compute each of them in turn.

First up: $\frac{\partial \Lambda}{\partial P}(P,Y)$. Since $\Lambda(P,Y) = (Y-P)^2$ for each element in Y and P:

$$\frac{\partial \Lambda}{\partial P}(P,Y) = -1 \times (2 \times (Y-P))$$

We're jumping ahead of ourselves a bit, but note that coding this up would simply be:

```
dLdP = -2 * (Y - P)
```

Next, we have an expression involving matrices: $\frac{\partial \alpha}{\partial N}(N,B)$. But since α is just addition, the same logic that we reasoned through with numbers in the prior chapter applies here: increasing any element of N by one unit will increase $P = \alpha(N,B) = N + B$ by one unit. Thus, $\frac{\partial \alpha}{\partial N}(N,B)$ is just a matrix of +1+s, of the same shape as N.

Coding *this* expression, therefore, would simply be:

```
dPdN = np.ones_like(N)
```

Finally, we have $\frac{\partial \nu}{\partial W}(X,W)$. As we discussed in detail in the last chapter, when computing derivatives of nested functions where one of the constituent functions is a matrix multiplication, we can act *as if*:

$$\frac{\partial \nu}{\partial W}(X,W) = X^T$$

which in code is simply:

```
dNdW = np.transpose(X, (1, 0))
```

We'll do the same for the intercept term; since we are just adding it, the partial derivative of the intercept term with respect to the output is simply 1:

```
dPdB = np.ones_like(weights['B'])
```

The last step is to simply multiply these together, making sure we use the correct order for the matrix multiplications involving `dNdW` and `dNdX` based on what we reasoned through at the end of the last chapter.

Calculating the Gradients: The (Full) Code

Recall that our goal is to take everything computed on or inputed into the forward pass—which, from the diagram in Figure 2-5, will include X, W, N, B, P, and y—and compute $\frac{\partial \Lambda}{\partial W}$ and $\frac{\partial \Lambda}{\partial B}$. The following code does that, receiving W and B as inputs in a dictionary called `weights` and the rest of the quantities in a dictionary called `forward_info`:

```
    '''
    Compute dLdW and dLdB for the step-by-step linear regression model.
    '''
    batch_size = forward_info['X'].shape[0]

    dLdP = -2 * (forward_info['y'] - forward_info['P'])

    dPdN = np.ones_like(forward_info['N'])

    dPdB = np.ones_like(weights['B'])

    dLdN = dLdP * dPdN

    dNdW = np.transpose(forward_info['X'], (1, 0))

    # need to use matrix multiplication here,
    # with dNdW on the left (see note at the end of last chapter)
    dLdW = np.dot(dNdW, dLdN)

    # need to sum along dimension representing the batch size
    # (see note near the end of this chapter)
    dLdB = (dLdP * dPdB).sum(axis=0)

    loss_gradients: Dict[str, ndarray] = {}
    loss_gradients['W'] = dLdW
    loss_gradients['B'] = dLdB

    return loss_gradients
```

As you can see, we simply compute the derivatives with respect to each operation and successively multiply them together, taking care that we do the matrix multiplication in the right order.[5] As we'll see shortly, this actually works—and after the intuition we built up around the chain rule in the last chapter, this shouldn't be too surprising.

> **NOTE**
>
> An implementation detail about those loss gradients: we're storing them as a dictionary, with the names of the weights as keys and the amounts that increasing the weights affect the losses as values. The `weights` dictionary is structured the same way. Therefore, we'll iterate through the weights in our model in the following way:
>
> ```
> for key in weights.keys():
>     weights[key] -= learning_rate * loss_grads[key]
> ```
>
> There is nothing special about storing them in this way; if we stored them differently, we would simply iterate through them and refer to them differently.

Using These Gradients to Train the Model

Now we'll simply run the following procedure over and over again:

1. Select a batch of data.
2. Run the forward pass of the model.
3. Run the backward pass of the model using the info computed on the forward pass.

The Jupyter Notebook for this chapter of the book includes a `train` function that codes this up. It isn't too interesting; it simply implements the preceding steps and adds a few sensible things such as shuffling the data to ensure that it is fed through in a random order. The key lines, which get repeated inside of a `for` loop, are these:

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)

for key in weights.keys():  # 'weights' and 'Loss_grads' have the same
    weights[key] -= learning_rate * loss_grads[key]
```

Then we run the `train` function for a certain number of *epochs*, or cycles through the entire training dataset, as follows:

```
train_info = train(X_train, y_train,
                   learning_rate = 0.001,
                   batch_size=23,
                   return_weights=True,
                   seed=80718)
```

The `train` function returns `train_info`, a `Tuple`, one element of which is the parameters or *weights* that represent what the model has learned.

> **NOTE**
>
> The terms "parameters" and "weights" are used interchangeably throughout deep learning, so we will use them interchangeably in this book.

Assessing Our Model: Training Set Versus Testing Set

To understand whether our model uncovered relationships in our data, we have to introduce some terms and ways of thinking from statistics. We think of any dataset received as being a *sample* from a *population*. Our goal is always to find a model that uncovers relationships in the population, despite us seeing only a sample.

There is always a danger that we build a model that picks up relationships that exist in the sample but not in the population. For example, it might be the case in our sample that yellow slate houses with three bathrooms are relatively inexpensive, and a complicated neural network model we build could pick up on this relationship even though it may not exist in the population. This is a problem known as *overfitting*. How can we detect whether a model structure we use is likely to have this problem?

The solution is to split our sample into a *training set* and a *testing set*. We use the training data to train the model (that is, to iteratively update the weights), and then we evaluate the model on the testing set to estimate its performance.

The full logic here is that if our model was able to successfully pick up on relationships that generalize from the *training set* to *the rest of the sample* (our whole dataset), then it is likely that the same "model structure" will generalize from our *sample*—which, again, is our entire dataset—to the *population*, which is what we want.

Assessing Our Model: The Code

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]):
    '''
    Generate predictions from the step-by-step linear regression model.
    '''
    N = np.dot(X, weights['W'])

    return N + weights['B']
```

Then we simply use the weights returned earlier from the `train` function and write:

```
preds = predict(X_test, weights)  # weights = train_info[0]
```

How good are these predictions? Keep in mind that at this point we haven't validated our seemingly strange approach of defining models as a series of operations, and training them by iteratively adjusting the parameters involved using the partial derivatives of the loss calculated with respect to the parameters using the chain rule; thus, we should be pleased if this approach works at all.

The first thing we can do to see whether our model worked is to make a plot with the model's predictions on the x-axis and the actual values on the y-axis. If every point fell exactly on the 45-degree line, the model would be perfect. Figure 2-6 shows a plot of our model's predicted and actual values.

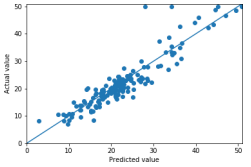


Figure 2-6. Predicted versus actual values for our custom linear regression model

Our plot looks pretty good, but let's quantify how good the model is. There are a couple of common ways to do that:

- Calculate the mean distance, in absolute value, between our model's predictions and the actual values, a metric called *mean absolute error*:

```
def mae(preds: ndarray, actuals: ndarray):
    '''
    Compute mean absolute error.
    '''
    return np.mean(np.abs(preds - actuals))
```

- Calculate the mean squared distance between our model's predictions and the actual values, a metric known as *root mean squared error*:

```
def rmse(preds: ndarray, actuals: ndarray):
    '''
    Compute root mean squared error.
    '''
    return np.sqrt(np.mean(np.power(preds - actuals, 2)))
```

The values for this particular model are:

Root mean squared error is a particularly common metric since it is on the same scale as the target. If we divide this number by the mean value of the target, we can get a measure of how far off a prediction is, on average, from its actual value. Since the mean value of `y_test` is `22.0776`, we see that this model's predictions of house prices are off by 5.0508 / 22.0776 ≅ 22.9% on average.

So are these numbers any good? In the Jupyter Notebook containing the code for this chapter, I show that performing a linear regression on this dataset using the most popular Python library for machine learning, Sci-Kit Learn, results in a mean absolute error and root mean squared error of `3.5666` and `5.0482`, respectively, which are virtually identical to what we calculated in our "first-principles-based" linear regression previously. This should give you confidence that the approach we've been taking so far in this book is in fact a valid approach for reasoning about and training models! Both later in this chapter, and in the next chapter we'll extend this approach to neural networks and deep learning models.

Analyzing the Most Important Feature

Before beginning modeling, we scaled each feature of our data to have mean 0 and standard deviation 1; this has computational advantages that we'll discuss in more detail in Chapter 4. A benefit of doing this that is specific to linear regression is that we can interpret the absolute values of the coefficients as corresponding to the importance of the different features to the model; a larger coefficient means that the feature is more important. Here are the coefficients:

```
np.round(weights['W'].reshape(-1), 4)
```

```
array([-1.0084,  0.7097,  0.2731,  0.7161, -2.2163,  2.3737,  0.7
       -2.6609,  2.629 , -1.8113, -2.3347,  0.8541, -4.2003])
```

The fact that the last coefficient is largest means that the last feature in the dataset is the most important one.

In Figure 2-7, we plot this feature against our target.

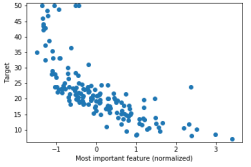


Figure 2-7. Most important feature versus target in custom linear regression

We see that this feature is indeed strongly correlated with the target: as this feature increases, the value of the target decreases, and vice versa. However, this relationship is *not* linear. The expected amount that the target changes as the feature changes from –2 to –1 is *not* the same amount that it changes as the feature changes from 1 to 2. We'll come back to this later.

In Figure 2-8, we overlay onto this plot the relationship between this feature and the *model predictions*. We'll generate this by feeding the following data through our trained model:

- The values of all features set equal to their mean

data

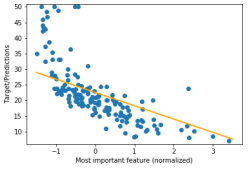


Figure 2-8. Most important feature versus target and predictions in custom linear regression

This figure shows (literally) a limitation of linear regression: despite the fact that there is a visually clear and "model-able" *non*linear relationship between this feature and the target, our model is only able to "learn" a linear relationship because of its intrinsic structure.

To have our model learn a more complex, nonlinear relationship between our features and our target, we're going to have to build a more complicated model than linear regression. But how? The answer will lead us, in a principles-based way, to building a neural network.

Neural Networks from Scratch

We've just seen how to build and train a linear regression model from first principles. How can we extend this chain of reasoning to design a more complex model that can learn nonlinear relationships? The central idea is that we'll first do *many* linear regressions, then feed the results through a nonlinear function, and finally do one last linear regression that ultimately makes the predictions. As it will turn out, we can reason through how to compute the gradients for this more complicated model in the same way we did for the linear regression model.

Step 1: A Bunch of Linear Regressions

What does it mean to do "a bunch of linear regressions"? Well, doing one linear regression involved doing a matrix multiplication with a set of parameters: if our data X had dimensions `[batch_size, num_features]`, then we multiplied it by a weight matrix W with dimensions `[num_features, 1]` to get an output of dimension `[batch_size, 1]`; this output is, for each observation in the batch, simply a *weighted sum* of the original features. To do multiple linear regressions, we'll simply multiply our input by a weight matrix with dimensions `[num_features, num_outputs]`, resulting in an output of dimensions `[batch_size, num_outputs]`; now, *for each observation*, we have `num_outputs` different weighted sums of the original features.

What are these weighted sums? We should think of each of them as a "learned feature"—a combination of the original features that, once the network is trained, will represent its attempt to learn combinations of features that help it accurately predict house prices. How many learned features should we create? Let's create 13 of them, since we created 13 original features.

Step 2: A Nonlinear Function

Next, we'll feed each of these weighted sums through a *non*linear function; the first function we'll try is the `sigmoid` function that was mentioned in Chapter 1. As a refresher, Figure 2-9 plots the `sigmoid` function.

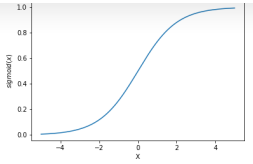


Figure 2-9. Sigmoid function plotted from x = –5 to x = 5

Why is using this nonlinear function a good idea? Why not the `square` function $f(x) = x^2$, for example? There are a couple of reasons. First, we want the function we use here to be *monotonic* so that it "preserves" information about the numbers that were fed in. Let's say that, given the date that was fed in, two of our linear regressions produced values of –3 and 3, respectively. Feeding these through the `square` function would then produce a value of 9 for each, so that any function that receives these numbers as inputs after they were fed through the `square` function would "lose" the information that one of them was originally –3 and the other was 3.

The second reason, of course, is that the function is nonlinear; this nonlinearity will enable our neural network to model the inherently nonlinear relationship between the features and the target.

Finally, the `sigmoid` function has the nice property that its derivative can be expressed in terms of the function itself:

$$\frac{\partial \sigma}{\partial u}(x) = \sigma(x) \times (1 - \sigma(x))$$

We'll make use of this shortly when we use the `sigmoid` function in the backward pass of our neural network.

Step 3: Another Linear Regression

Finally, we'll take the resulting 13 elements—each of which is a combination of the original features, fed through the `sigmoid` function so that they all have values between 0 and 1—and feed them into a regular linear regression, using them the same way we used our original features previously.

Then, we'll try training the *entire* resulting function in the same way we trained the standard linear regression earlier in this chapter: we'll feed data through the model, use the chain rule to figure out how much increasing the weights would increase (or decrease) the loss, and then update the weights in the direction that decreases the loss at each iteration. Over time (we hope) we'll end up with a more accurate model than before, one that has "learned" the inherent nonlinearity of the relationship between our features and our target.

It might be tough to wrap your mind around what's going on based on this description, so let's look at an illustration.

Diagrams

Figure 2-10 is a diagram of what our more complicated model now looks like.

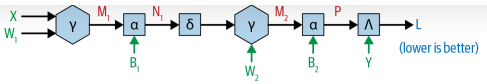


Figure 2-10. Steps 1–3 translated into a computational graph of the kind we saw in Chapter 1

You'll see that we start with matrix multiplication and matrix addition, as before. Now let's formalize some terminology that was mentioned previously: when we apply these operations in the course of a nested function, we'll call the first matrix that we use to transform the input features the *weight* matrix, and we'll call the second matrix, the one that is added to each resulting set of features, the *bias*. That's why we'll denote these as W_1 and B_1.

After applying these operations, we'll feed the results through a sigmoid function and then repeat the process again with *another* set of weights and biases—now called W_2 and B_2—to get our final prediction, P.

ANOTHER DIAGRAM?

Does representing things in terms of these individual steps give you intuition for what is going on? This question gets at a key theme of this book: to fully understand neural networks, we have to see multiple representations, each one of which highlights a different aspect of how neural networks work. The representation in Figure 2-10 doesn't give much intuition about the "structure" of the network, but it does indicate clearly how to train such a model: on the backward pass, we'll compute the partial derivative of each constituent function, evaluated at the input to that function, and then calculate the gradients of the loss with respect to each of the weights by simply multiplying all of these derivatives together—just as we saw in the simple chain rule examples from Chapter 1.

Nevertheless, there is another, more standard way to represent a neural network like this: we could represent each of our original features as circles. Since we have 13 features, we need 13 circles. Then we need 13 more circles to represent the 13 outputs of the "linear regression-sigmoid" operation we're doing. In addition, each of these circles is a function of all 13 of our original features, so we'll need lines connecting all of the first set of 13 circles to all of the second set.[6]

Finally, all of these 13 outputs are used to make a single final prediction, so we'll draw one more circle to represent the final prediction and 13 lines showing that these "intermediate outputs" are "connected" to this final prediction.

Figure 2-11 shows the final diagram.[7]

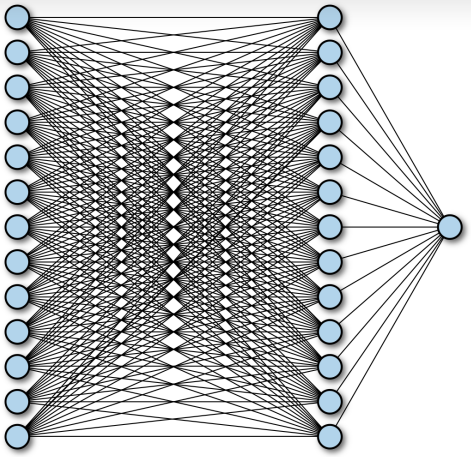


Figure 2-11. A more common (but in many ways less helpful) visual representation of a neural network

If you've read anything about neural networks before, you may have seen them represented like the diagram in Figure 2-11: as circles with lines connecting them. While this representation does have some advantages—it lets you see "at a glance" what kind of neural network this is, how many layers it has, and so on—it doesn't give any indication of the actual calculations involved, or of how such a network might be trained. Therefore, while this diagram is extremely important for you to see because you'll see it in other places, it's included here primarily so you can see the *connection* between it and the primary way we are representing neural networks: as boxes with lines connecting them, where each box represents a function that defines both what should happen on the forward pass for the model to make predictions and what should happen on the backward pass for the model to learn. We'll see in the next chapter how to translate even more directly between these diagrams and code by coding each function as a Python class inheriting from a base `Operation` class—and speaking of code, let's cover that next.

Code

In coding this up, we follow the same function structure as in the simpler linear regression function from earlier in the chapter—taking in `weights` as a dictionary and returning both the loss value and the `forward_info` dictionary, while replacing the internals with the operations specified in Figure 2-10:

```
def forward_loss(X: ndarray,
                 y: ndarray,
                 weights: Dict[str, ndarray]
                 ) -> Tuple[Dict[str, ndarray], float]:
    '''
    Compute the forward pass and the loss for the step-by-step
    neural network model.
    '''
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']

    O1 = sigmoid(N1)

    M2 = np.dot(O1, weights['W2'])

    P = M2 + weights['B2']
```

```
    forward_info['M1'] = M1
    forward_info['N1'] = N1
    forward_info['O1'] = O1
    forward_info['M2'] = M2
    forward_info['P'] = P
    forward_info['y'] = y

    return forward_info, loss
```

Even though we're now dealing with a more complicated diagram, we're still just going step by step through each operation, doing the appropriate computation, and saving the results in `forward_info` as we go.

Neural Networks: The Backward Pass

The backward pass works the same way as in the simpler linear regression model from earlier in the chapter, just with more steps.

DIAGRAM

The steps, as a reminder, are:

1. Compute the derivative of each operation and evaluate it at its input.
2. Multiply the results together.

As we'll see yet again, this will work because of the chain rule. Figure 2-12 shows all the partial derivatives we have to compute.

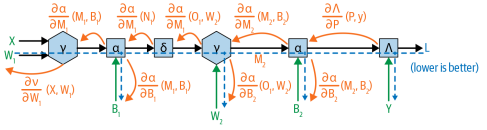


Figure 2-12. The partial derivatives associated with each operation in the neural network that will be multiplied together on the backward pass

Conceptually, we want to compute all these partial derivatives, tracing backward through our function, and then multiply them together to get the gradients of the loss with respect to each of the weights, just as we did for the linear regression model.

MATH (AND CODE)

Table 2-1 lists these partial derivatives and the lines in the code that correspond to each one.

Table 2-1. Derivative table for neural network

Derivative	Code
$\frac{\partial \Lambda}{\partial P}(P, y)$	`dLdP = -(forward_info[y] - forward_info[P])`
$\frac{\partial o_2}{\partial M_2}(M_2, B_2)$	`np.ones_like(forward_info[M2])`
$\frac{\partial o_2}{\partial B_2}(M_2, B_2)$	`np.ones_like(weights[B2])`
$\frac{\partial o_2}{\partial W_2}(O_1, W_2)$	`dM2dW2 = np.transpose(forward_info[O1], (1, 0))`
$\frac{\partial o_2}{\partial O_1}(O_1, W_2)$	`dM2dO1 = np.transpose(weights[W2], (1, 0))`
$\frac{\partial \sigma}{\partial u}(N_1)$	`dO1dN1 = sigmoid(forward_info[N1] × (1 - sigmoid(forward_info[N1])`
$\frac{\partial o_1}{\partial M_1}(M_1, B_1)$	`dN1dM1 = np.ones_like(forward_info[M1])`
$\frac{\partial o_1}{\partial B_1}(M_1, B_1)$	`dN1dB1 = np.ones_like(weights[B1])`
$\frac{\partial o_1}{\partial W_1}(X, W_1)$	`dM1dW1 = np.transpose(forward_info[X], (1, 0))`

> **NOTE**
>
> The expressions we compute for the gradient of the loss with respect to the bias terms, `dLdB1` and `dLdB2`, will have to be summed along the rows to account for the fact that the same bias element is added to each row in the batch of data passed through. See "Gradient of the Loss with Respect to the Bias Terms" for details.

THE OVERALL LOSS GRADIENT

You can see the full `loss_gradients` function in the Jupyter Notebook for this chapter on the book's GitHub page. This function computes each of the partial derivatives in Table 2-1 and multiplies them together to get the gradients of the loss with respect to each of the `ndarray`s containing the weights:

- `dLdW2`

- `dLdB1`

The only caveat is that we sum the expressions we compute for `dLdB1` and `dLdB2` along `axis = 0`, as described in "Gradient of the Loss with Respect to the Bias Terms".

We've finally built our first neural network from scratch! Let's see if it is in fact any better than our linear regression model.

Training and Assessing Our First Neural Network

Just as the forward and backward passes worked the same for our neural network as for the linear regression model from earlier in the chapter, so too are training and evaluation the same: for each iteration of data, we pass the input forward through the function on the forward pass, compute the gradients of loss with respect to the weights on the backward pass, and then use these gradients to update the weights. In fact, we can use the following identical code inside the training loop:

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)

for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

The difference simply lies in the internals of the `forward_loss` and `loss_gradients` functions, and in the `weights` dictionary, which now has four keys (`W1`, `B1`, `W2`, and `B2`) instead of two. Indeed, this is a major takeaway from this book: even for very complex architectures, the mathematical principles and high-level training procedures are the same as for simple models.

We also get predictions from this model in the same way:

```
preds = predict(X_test, weights)
```

The difference again is simply in the internals of the `predict` function:

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]) -> ndarray:
    '''
    Generate predictions from the step-by-step neural network model.
    '''
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']

    O1 = sigmoid(N1)

    M2 = np.dot(O1, weights['W2'])

    P = M2 + weights['B2']

    return P
```

Using these predictions, we can calculate the mean absolute error and root mean squared error on the validation set, as before:

```
Mean absolute error: 2.5289
Root mean squared error: 3.6775
```

Both values are significantly lower than the prior model! Looking at the plot of predictions versus actuals in Figure 2-13 shows similar improvements.

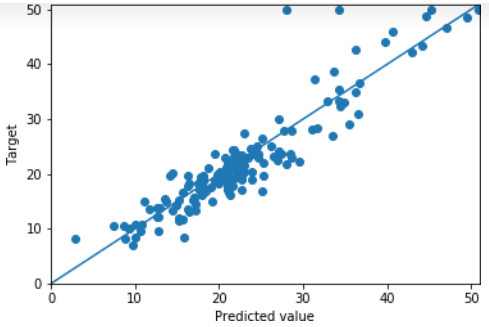


Figure 2-13. Predicted value versus target, in neural network regression

Visually, the points look closer to the 45-degree line than in Figure 2-6. I encourage you to step through the Jupyter Notebook on the book's GitHub page and run the code yourself!

Two Reasons Why This Is Happening

Why does this model appear to be performing better than the model before? Recall that there was a *nonlinear* relationship between the most important feature of our earlier model and our target; nevertheless, our model was constrained to learn only *linear* relationships between individual features and our target. I claim that, by adding a nonlinear function into the mix, we have allowed our model to learn the proper, nonlinear relationship between our features and our target.

Let's visualize this. Figure 2-14 shows the same plot we showed in the linear regression section, plotting the normalized values of the most important feature from our model along with both the values of the target and the *predictions* that would result from feeding the mean values of the other features while varying the values of the most important feature from –3.5 to 1.5, as before.

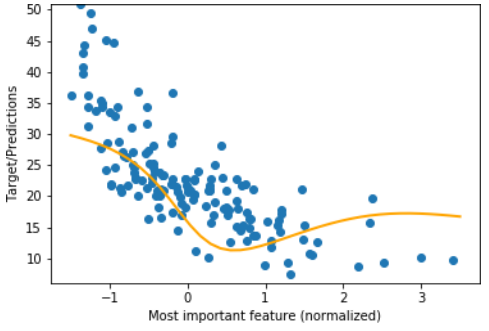


Figure 2-14. Most important feature versus target and predictions, neural network regression

We can see that the relationship shown (a) is now nonlinear and (b) more closely matches the relationship between this feature and the target (represented by the points), as desired. So adding the nonlinear function to our model allowed it to learn, via iteratively updating the weights using the training, the nonlinear relationship that existed between the inputs and the outputs.

That's the first reason why our neural network performed better than a

target, as opposed to just individual features. This is because the neural network uses a matrix multiplication to create 13 "learned features," each of which is a combination of all the original features, and then essentially applies another linear regression on top of these learned features. For example, doing some exploratory analysis that is shared on the book's website, we can see that the most important combinations of the 13 original features that the model has learned are:

$$-4.44 \times \text{feature}_6 - 2.77 \times \text{feature}_1 - 2.07 \times \text{feature}_7 + \ldots$$

and:

$$4.43 \times \text{feature}_2 - 3.39 \times \text{feature}_4 - 2.39 \times \text{feature}_1 + \ldots$$

These will then be included, along with 11 other learned features, in a linear regression in the last two layers of the neural network.

These two things—learning *nonlinear* relationships between individual features and our target, and learning relationships between *combinations* of features and our target—are what allow neural networks to often work better than straightforward regressions on real-world problems.

Conclusion

In this chapter, you learned how to use the building blocks and mental models from Chapter 1 to understand, build, and train two standard machine learning models to solve real problems. I started by showing how to represent a simple machine learning model from classical statistics—linear regression—using a computational graph. This representation allowed us to compute the gradients of the loss from this model with respect to the model's parameters and thus train the model by continually feeding in data from the training set and updating the model's parameters in the direction that would decrease the loss.

Then we saw a limitation of this model: it can only learn *linear* relationships between the features and target; this motivated us to try building a model that could learn *nonlinear* relationships between the features and target, which led us to build our first neural network. You learned how neural networks work by building one from scratch, and you also learned how to train them using the same high-level procedure we used to train our linear regression models. You then saw empirically that the neural network performed better than the simple linear regression model and learned two key reasons why: the neural network was able to learn *nonlinear* relationships between the features and the target and also to learn relationships between *combinations* of features and the target.

Of course, there's a reason we ended this chapter still covering a relatively simple model: defining neural networks in this way is an extremely manual process. Defining the forward pass involved 6 individually coded operations, and the backward pass involved 17. However, discerning readers will have noticed that there is a lot of repetition in these steps, and by properly defining abstractions, we can move from defining models in terms of individual operations (as in this chapter) to defining models in terms of these abstractions. This will allow us to build more complex models, including deep learning models, while deepening our understanding of how these models work. That is what we'll begin to do in the next chapter. Onward!

---

1 The other kind of machine learning, *un*supervised learning, can be thought of as finding relationships between things you have measured and things that have not been measured yet.

2 Though in a real-world problem, even how to choose a price isn't obvious: would it be the price the house last sold for? What about a house that hasn't been on the market for a long time? In this book, we'll focus on examples in which the numeric representation of the data is obvious or has

3 Most of you probably know that these are called “categorical” features.

4 At least the ones we’ll see in this book.

5 In addition, we have to sum `dLdB` along axis 0; we explain this step in more detail later in this chapter.

6 This highlights an interesting idea: we *could* have outputs that are connected to only *some* of our original features; this is in fact what convolutional neural networks do.

7 Well, not quite: we haven’t drawn *all* of the 169 lines we would need to show all the connections between the first two “layers” of features, but we have drawn enough of them so that you get the idea.

PREV
1. Foundations

NEXT
3. Deep Learning from Scratch