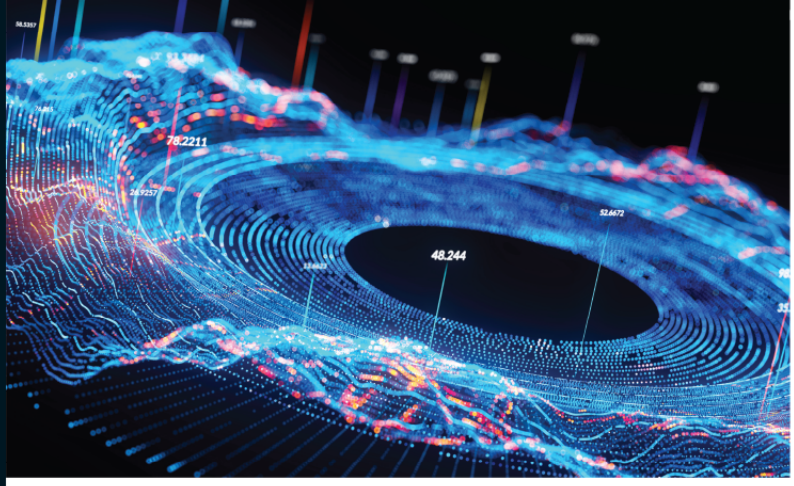
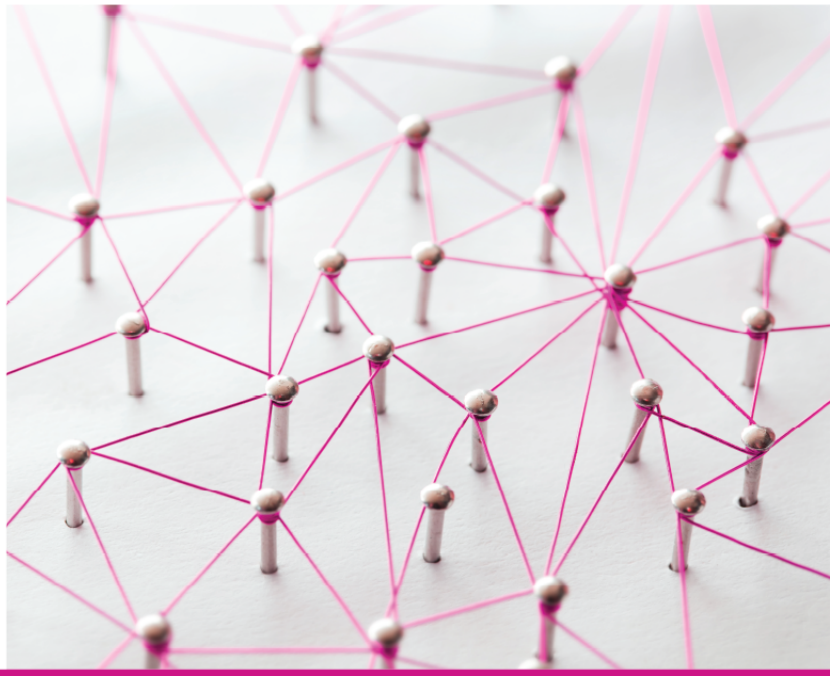



```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

# Generate dummy data
np.random.seed(42)
X = np.random.randn(100, 1)  # 100 sar
true_params = np.array([2.5, 3.7])  # Tru
y = true_params[0] + true_params[1] *

# Shuffle the data
indices = np.random.permutation(len(X
X = X[indices]
y = y[indices]
```

Machine Le
Deep Learn
Neural Netw
Gradient D
Classifica
Regulariza
Convolutio



Keras 3

The Comprehensive Guide to Deep Learning with the Keras API and Python

Mohammad Nauman

Rheinwerk Computing

Contents

Chapter 6
Regularization Techniques

In this chapter, we'll explore the critical challenge that every machine learning practitioner faces: creating models that generalize well beyond their training data. You'll learn how to identify when our model is too complex or too simple. By the end of this chapter, you'll have acquired essential tools for building neural networks that capture meaningful patterns without being led astray by noise in the training data.

As we've explored in previous chapters, machine learning models can capture increasingly complex patterns in data. We've moved from simple linear models to powerful neural networks, each time expanding our ability to represent sophisticated relationships. But this growing power comes with a subtle danger. Just as a sharp knife requires careful handling, powerful models demand proper control to avoid cutting ourselves. In this chapter, we tackle one of the most fundamental challenges in machine learning: how to create models that are powerful enough to capture real patterns in our data without being so flexible that they memorize the noise and peculiarities of our specific training examples. This balancing act lies at the heart of successful machine learning.

When models are too simple, they miss important patterns—such as attempting to explain quantum physics using only words a five-year-old would understand. Imagine trying to describe the rich, multifaceted flavor profile of a gourmet meal using only the words "yummy" and "yucky"—you'd capture the basic sentiment but miss all the nuance that makes the experience special. This is what happens with underfit models; they capture the general direction but miss the subtleties that matter. This problem, called underfitting, limits the usefulness of our predictions, leaving us with a crude approximation where precision is needed.

Conversely, when models are too complex relative to the amount of training data available, they can achieve perfect performance on training examples while failing spectacularly on new data. It's the equivalent of a student who memorizes exam answers without understanding the underlying principles—they might ace the practice test but freeze when the real exam presents questions with even slight variations. Think of an overly complex model as a gossip who creates elaborate narratives based on coincidences: "John always wears blue on days when the stock market goes up!" Such a pattern might perfectly explain past observations while being utterly useless for predictions. This problem, called overfitting, often betrays itself through a growing gap

between training and testing performance—a warning sign that our model is becoming too fixated on the particularities of our training examples rather than learning generalizable patterns.

How to Approach This Chapter

The content of this chapter might seem mathematically dense, but it's perfectly fine if you don't fully grasp every detail right away. I encourage you to read through the content even when concepts feel challenging. We'll revisit key takeaways at the end, and I promise these techniques will make more sense once you see them in action. In practice, they are really easy to use.

In this chapter, we'll see the tools that allow us to navigate between these extremes. These tools let us use sophisticated, high-capacity models while restraining their tendency to overfit. They act like guardrails, keeping our models on the road to good generalization rather than veering off into the ditch of memorization. The general term for this balancing act is regularization. Here, we'll explore several powerful regularization approaches. We'll begin with the L2 regularization method, which constrains our model weights to prevent them from growing too large. We'll then examine dropout, which makes regularization very easy in practice. Along the way, we'll develop an intuitive understanding of the bias-variance trade-off that underlies these techniques and learn practical skills for implementing regularization in Keras. By the end of this chapter, you'll possess the essential tools to build models that not only perform well on training data but generalize effectively to new, unseen examples—the true measure of machine learning success. Let's begin working on this plan by revisiting our old friend—the housing prices dataset.

6.1 An Overview of Overfitting and Underfitting: Do You Need More Data?

In Chapter 3, we built a simple housing price model with just one feature. We assumed a *linear* relationship and thus drew a straight line through our data points, capturing the basic relationship between house size and price. This linear approach served us well as an introduction, but it's a bit like trying to describe a complex landscape using only straight paths—sometimes, you need curves to capture the real terrain. We later introduced neural networks as one way to capture nonlinear relationships, giving our models the flexibility to bend and curve in response to the data. But there's another, often simpler approach we can take: *polynomial features*. In the following sections, we'll take a look at how we can add flexibility to our models. We'll also discuss the problems that arise as we try to do this and then introduce generalized solutions to these problems.

6.1.1 From Lines to Curves: Adding Polynomial Features

Let's consider what happens if the relationship between house size and price isn't a straight line but a curve. Think about it intuitively: Perhaps each additional square meter becomes more valuable as the house gets larger. A jump from 1,500 to 2,000 square meters might add more value than a jump from 1,000 to 1,500 square meters. If the ground truth is that the relationship between inputs and outputs is *quadratic*, but our model assumes linearity, we may get some data points right, but the rest might not fit that well. Figure 6.1 shows what this model will look like. While not that bad for these few data points, it will break down very quickly for larger x-axis values. (For the following discussion, try not to focus on the actual y-axis values. Try to look at the bigger picture by following the trend lines.)

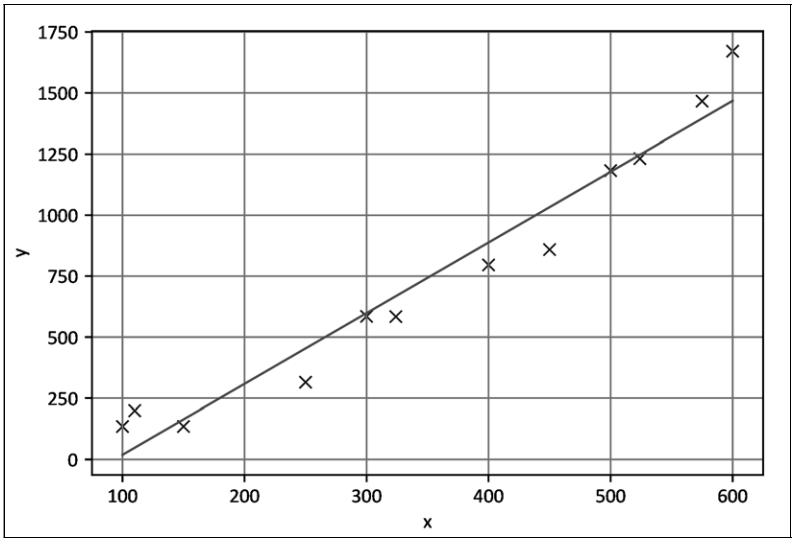


Figure 6.1 Linear Model for a Quadratic Relationship

We can model the underlying relationship between the area and the price as quadratic rather than linear. So, the ground truth is going to look like Figure 6.2. Compare these two models, and you'll notice that the quadratic line follows the data more closely, especially as the input values increase.

Notice how the squared values grow much faster than the original areas. As the house size increases linearly, the squared term increases quadratically. This gives our model the power to capture accelerating relationships—known as *polynomial regression*. This means that, instead of just using the original feature (house size), we create additional features by raising it to different powers. If our original model looked like this:

$$\text{price} = \theta_1 \times \textit{area} + \theta_0$$

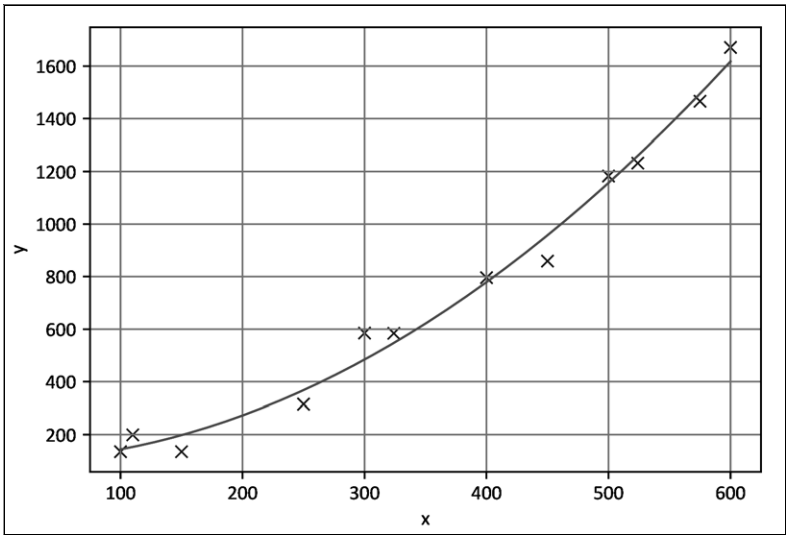


Figure 6.2 Quadratic Model for the Quadratic Relationship

A quadratic model would look like this:

$$\text{price} = \theta_1 \times area + \theta_2 \times area^2 + \theta_0$$

To incorporate this in our gradient descent algorithm, we simply have to add a new column containing the squared values of the original feature, as shown in Table 6.1.

Area	Area^2	Price
145	21,025	808
202	40,804	993
305	93,025	727
365	133,225	582
450	202,500	1612

Table 6.1 Values of Features Squared

Adding this column is quite easy to implement in code, especially because this has to be done only once. We can augment our original dataset with these new pseudo-features and then save the modified dataset to the disk. After this step is done once, we only need to consider the modified dataset for polynomial regression. This can be easily achieved through Scikit-learn (sklearn) as well, as shown in Listing 6.1. Here, the `PolynomialFeatures` function acts like a feature multiplication workshop. When we set `degree=2`, we're telling it to create all possible combinations of our features raised to powers up to 2. Because we only have one feature (house size), it simply gives us that feature and its square. The `include_bias=False` parameter tells it not to add a constant column of 1s

because we'll do that in our model ourselves. (You can run the code through the **06-01-polynomial-features** notebook in the book resources on *https://recluze.net/keras-book* and on the book's official web page.)

When we call `poly.fit_transform(X)`, the function analyzes our data to understand its structure and then creates the new polynomial features. The result, `X_poly`, now has two columns: the original house sizes and those same values squared.

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Original feature - house sizes
X = np.array([[1000], [1500], [2000], [2500], [3000]])

# Transform into polynomial features
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X)

print("Original features:")
print(X)
print("\nPolynomial features (degree 2):")
print(X_poly)
# Output --------------
# Original features:
# [[1000]
#  [1500]
#  [2000]
#  [2500]
#  [3000]]
#
# Polynomial features (degree 2):
# [[1.00e+03 1.00e+06]
#  [1.50e+03 2.25e+06]
#  [2.00e+03 4.00e+06]
#  [2.50e+03 6.25e+06]
#  [3.00e+03 9.00e+06]]
```

Listing 6.1 Adding Polynomial Features Using Sklearn

We've just transformed our single feature into two features—the original area and the area squared. This gives our model more flexibility to capture the curve in our data.

The beauty of this approach is that we can continue using our existing gradient descent algorithm from the previous chapters without any modifications. From the perspective of our gradient descent code, we're just adding another feature—it doesn't know or care

that this new feature is derived from the original one. If we call our original feature x and our new squared feature v (where $v = x^2$), our model becomes

$$\text{price} = \theta_1 \times x + \theta_2 \times v + \theta_0$$

All our gradient descent equations from Chapter 3 still apply perfectly. We simply have one more weight (θ_2) to learn, but the fundamental process remains unchanged. The partial derivatives, weight updates, and convergence criteria all work exactly as before. In fact, we could add any number of transformed features (x^3, x^4, $log(x)$, etc.), and our gradient descent algorithm would still work without modification. This is one of the most powerful aspects of machine learning algorithms: once you understand the core principles, you can easily extend them to more complex scenarios by transforming your data rather than rewriting your algorithms.

6.1.2 Using Increasingly Complex Models

When we fit a linear model to our housing data, it can only draw a straight line. But when we add polynomial features, our model can fit curves of increasing complexity. Assume now that we don't know what the underlying relationship is. So, we go a step further and add 8 degrees of polynomial to our dataset instead of just the quadratic term. The relationship learned as a result will look like Figure 6.3. As you can see, the line follows even closer to the dataset but still misses out some points.

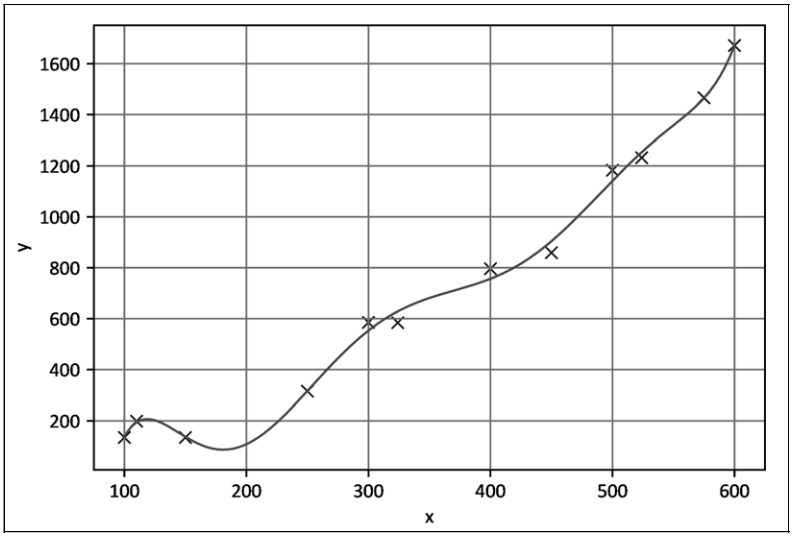


Figure 6.3 8th Order Polynomial Fitting the Same Data

Take this to an extreme and instead try to see what happens if we try to model a relationship that is a 12^{th} degree polynomial. This leads to a curve fitting our data perfectly, as shown in Figure 6.4.

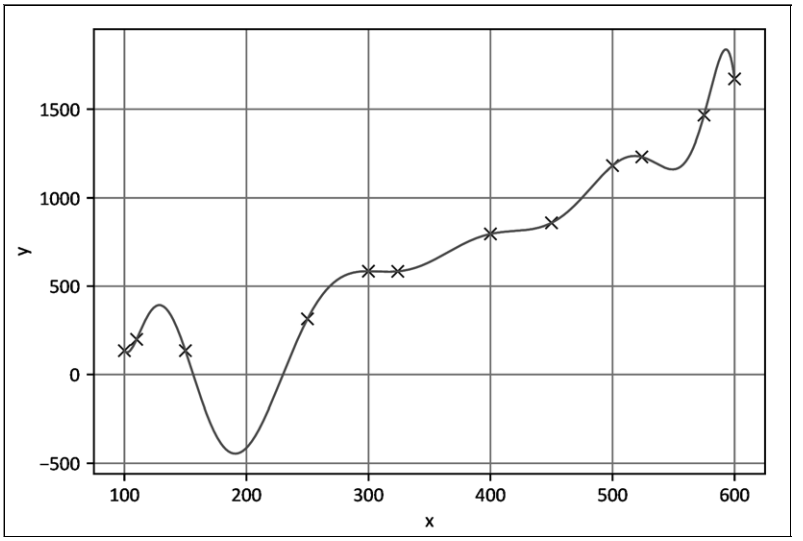


Figure 6.4 12th Order Polynomial Fitting the Same Data

The degree-1 model is just a straight line, while higher-degree models can capture increasingly complex patterns. As we increase the degree, our model becomes more flexible:

- Degree 1: A straight line that misses most points
- Degree 2: A gentle curve that follows the general trend
- Degree 8: A slightly more flexible curve that gets closer to all points
- Degree 12: A wildly oscillating curve that touches every point perfectly

Stop here and think about it! This flexibility is a double-edged sword. The degree-12 model fits our training data perfectly, but it's learned a complex pattern that's unlikely to generalize well to new houses. It's like memorizing the exact answers to the practice problems without understanding the underlying principles—you'll ace the practice test but struggle with new problems.

6.1.3 The Balance of Complexity

Adding more polynomial terms is like giving our model increasingly sophisticated vocabulary—but as with human language, more words don't always lead to clearer communication. Let's see what might go wrong if we try to use the 12^{th} degree polynomial we modeled previously. To truly understand how our models perform, we need to look beyond the training data. That's where our test set comes in. These are data points our model hasn't seen during training—the equivalent of a pop quiz with new questions. Let's visualize how models of different complexities perform on both training and test data in Figure 6.5.

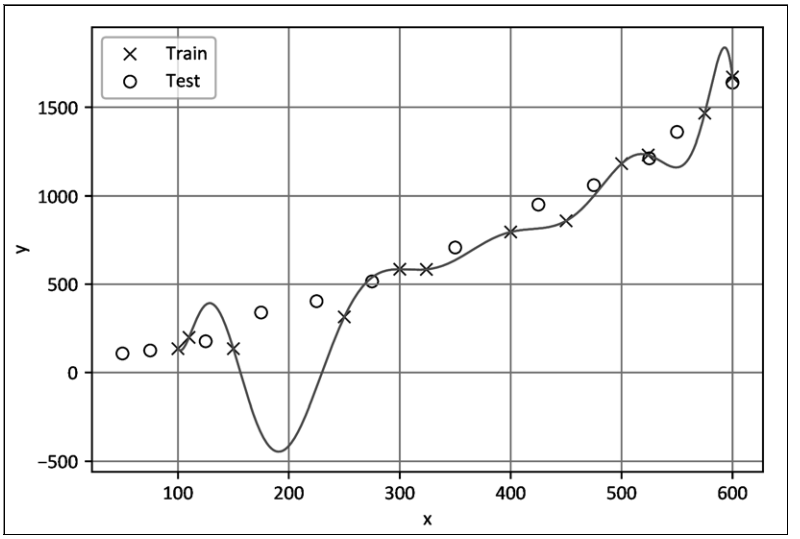


Figure 6.5 Introducing the Test Set into the Mix

Notice how the curve passes precisely through each *x* (training points), creating an elaborate, winding path. At first glance, this might seem impressive, like a student who can recite every example from the textbook perfectly. But the true test of learning isn't memorization; it's the ability to apply knowledge to new situations. When we look at how this model performs on the test data (the circles), we see concerning discrepancies. The most dramatic example occurs around *x = 200*, where our model takes a dramatic plunge to predict a value of -500. But the actual test point in that region shows a value closer to +400! That's a staggering 900-unit error.

Interactive Demo for Polynomial Fits

You can play with an interactive demo we've created to show how the fit changes based on the order of the polynomial. You can find it on the book resources page at *https://recluze.net/keras-book* as **06-04-polynomial-fit-demo**.

Our model has become so fixated on perfectly fitting every training point that it's created an unnecessarily complex explanation. Between *x = 150* and *x = 250*, the model noticed there weren't many training points, so it took a wild dive downward before shooting back up. The algorithm didn't "decide" to do this out of malice or confusion; it simply found that this rollercoaster curve was mathematically the best way to hit all the training points perfectly. So, there are two extremes that we can face while deciding on model complexity. We'll discuss these in the following sections and provide guidance for finding the sweet spot in between.

When Models Are Too Simple: Underfitting

Looking at our degree-1 polynomial (the straight line), we notice it performs poorly on both the training and test data. The model fails to capture the curvature that clearly exists in our housing price relationship. The training error is high, and the test error is similarly high.

As mentioned earlier, this problem is called *underfitting* or *high bias*. It's like trying to explain quantum mechanics using only elementary school vocabulary—the tools are simply inadequate for the task at hand. Our model has made an overly simplistic assumption about the world (that housing prices are a linear function of area), and this bias prevents it from learning the true pattern. This assumption is built into the very structure of our linear model. By restricting ourselves to a straight line, we're essentially declaring that each additional square meter adds exactly the same value to a house, regardless of the home's size. This ignores economic realities such as premium pricing for larger homes, the diminishing utility of extra space beyond certain thresholds, or the way different size brackets might appeal to different market segments.

This is what high bias means in machine learning: The model has strong preconceptions about what the underlying function should look like, and these preconceptions are wrong. No amount of additional training data will help a linear model fit a quadratic relationship—it simply doesn't have the capacity to represent the curve. In practice, it's usually easy to recognize a situation in which a model is underfitting. This is usually when we have very high training loss. This means that our model can't even memorize the data points given to it, let alone predict unseen data points.

The Term High Bias

Underfitting is called high bias because the model is biased toward its own simplistic understanding rather than adapting to the true complexity of the data. It doesn't want to learn from its mistakes.

When Models Are Too Complex: Overfitting

On the other end, look at the degree-12 polynomial. It creates a wildly oscillating curve that passes almost perfectly through each training point. The training error is nearly zero—our model has essentially memorized the training data! But look what happens with the test points: The model's predictions are way off. The test error is enormous compared to the training error. Our model has learned the peculiarities of our specific training examples rather than the general relationship between house size and price.

To understand this distinction, we need to recognize that any dataset contains two components: the *underlying pattern* we want to learn and the *random noise* or peculiarities specific to our sample. The underlying pattern is the true relationship that generalizes across all houses—perhaps house prices increase with area following a gentle

quadratic curve due to fundamental market dynamics. This pattern applies to houses we haven't seen yet and represents the predictive knowledge we're actually seeking. The peculiarities, on the other hand, are random fluctuations specific to our training examples—perhaps one house in our training set sold for slightly more because it had a particularly nice view, or another sold for less because the owner needed to move quickly. These factors aren't captured in our features and appear as random noise in our data. When we use a degree-12 polynomial to fit just a handful of training points, we're giving our model enough flexibility to memorize not just the general trend but also these random fluctuations. It's like creating an elaborate theory to explain why one student got a 93% and another got a 91% on a test, when the difference might just be that one student guessed correctly on one more question.

As mentioned at the beginning of the chapter, this problem is called *overfitting* or *high variance* and is recognizable through a key indicator: very low error on training data along with significantly higher error on test data that leads to a large performance gap between training and test results. Overfitting occurs when our model is so flexible that it captures not only the underlying pattern but also the random noise in our training data.

The Term High Variance

Overfitting is called high variance because the parameters the model learns are going to vary a lot depending on which data points are used to train it. If we pick odd-numbered data points, the model will learn their peculiarities, which will be quite different from the specific noise in even-numbered data points.

Finding the Sweet Spot

The degree-2 polynomial seems to strike a good balance. It's flexible enough to capture the curvature in our data without going overboard with unnecessary complexity. Both the training and test errors are relatively low, and they're similar to each other—a sign that our model is generalizing well.

This illustrates a fundamental principle in machine learning: the *bias-variance tradeoff*. As we increase model complexity, the following happens:

- Bias tends to decrease (the model can represent more complex patterns).
- Variance tends to increase (the model becomes more sensitive to the specific training examples).

Our goal is to find the sweet spot that balances these two sources of error. It's like stretching a rubber band to wrap around a package—not stretching enough leaves it too loose to hold anything together (underfitting), while stretching it too far causes it to snap (overfitting). We need just the right amount of tension that secures the package without breaking the band.

Important Issues to Keep in Mind

Remember that our algorithms don't "see" the data the way we do. When we perform gradient descent, the machine isn't visualizing curves or making aesthetic judgments about whether or not the line fitting the points is visually appealing. It's simply calculating a loss value and following the gradient downward, regardless of whether the resulting function is sensibly smooth or wildly oscillating between data points. This presents a fundamental challenge that becomes even more pronounced with high-dimensional data. In our housing example, we could visualize the relationship between house size and price in a simple 2D plot. But what if our model includes dozens or hundreds of features? We lose our ability to eyeball the complexity of the relationship, and even experts can't intuitively judge when a model with 50 features is using a relationship that's too complex.

We have two broad approaches to address this challenge. The first, which we touched on in Chapter 3, is to simply make an assumption about the appropriate complexity for our problem. We might decide based on domain knowledge that a quadratic relationship makes sense for housing prices or that certain features should interact while others shouldn't. This approach works well when we have strong *prior knowledge*, but it's not always available.

The second approach—which we'll focus on throughout the rest of this chapter—is to develop automated methods that help us determine the right level of complexity. We need mechanisms that allow our models to find that balance themselves without requiring us to manually specify the perfect polynomial degree or feature interactions.

Another critical point to remember is that we must make these decisions without peeking at the test set. Using test data to choose our model's complexity is like getting access to exam questions before a test—it defeats the purpose of having an independent evaluation. This form of *data leakage* can give us an overly optimistic view of how well our model will perform on truly new data when deployed in the real world.

Always Keep the Test Set Separate

When writing code, always make sure that you aren't using your test data points during training. While this might lead to good laboratory results, the model's real success—that in the real world—will become unpredictable!

These techniques are guardrails that keep our models on the right path. Rather than manually deciding when to stop adding complexity, we'll develop systems that naturally prefer simpler explanations unless there's strong evidence for complexity. As we explore these techniques, remember that finding the right complexity isn't simply a theoretical concern. It's fundamental to creating models that work in the real world. A model that perfectly memorizes training data but fails on new examples is like a student who can recite textbook pages verbatim but can't apply the concepts to new problems.

Our goal is to build models that truly learn the underlying patterns. They shouldn't be limited to the specific examples we've shown them.

But how do we come up with complex models that capture intricate patterns in our data without overfitting? That's where regularization comes in—a technique that allows us to use complex models while preventing them from overfitting. We'll explore this powerful approach in the next section.

6.1.4 Regularization Term

Now that we've identified the problem of overfitting, let's discuss a powerful solution that doesn't require us to manually restrict our model's complexity. Rather than deciding in advance which polynomial degree to use, what if we could use a high-degree polynomial but somehow encourage it to behave more reasonably? This is where *regularization* enters the picture. Regularization works by modifying our loss function—the very compass that guides our model during training. Let's recall the loss function we used in Chapter 3 for our housing price prediction:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

This function only cares about one thing: how well our predictions ($\hat{y}_i$) match the actual values (y_i) in our training data. It's like a teacher who grades solely on getting the correct final answers without considering how students arrive at those answers or whether their methods would work on different problems.

We're going to add a term to this loss function. It might seem strange at first but we'll explain the "why" of this in a minute. This is called the *L2 regularization term* and is given as

$$\lambda\sum_{j=1}^{n}\theta_j^2$$

where λ (lambda) is a new hyperparameter that controls complexity, and θ_j represents each parameter in our model. Our complete regularized loss function becomes

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}i - y_i)^2 + \lambda\sum_{j=1}^{n}\theta_j^2$$

We can modify the value of this parameter at will. First, let's consider what happens when we set $\lambda = 0$. In this case, our loss function simplifies to

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}i - y_i)^2 \quad + \quad 0 \cdot \sum_{j=1}^{n}\theta_j^2 \quad = \quad \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

We're right back to our original mean squared error (MSE)! The regularization term completely disappears, and our model is free to use any parameter values it wants, how-

ever large, to fit the training data perfectly. This is the world we were living in before, where our polynomial models could create those wild oscillations to pass through every training point. See the solid lined contour plot in Figure 6.6 for what the loss function landscape will look like in this case. In this example, we get $\theta_1 = 10, \theta_2 = 5$ for our model parameter values that minimize the loss. If you want a refresher on contour plots, please refer to Chapter 3, Section 3.2.1.

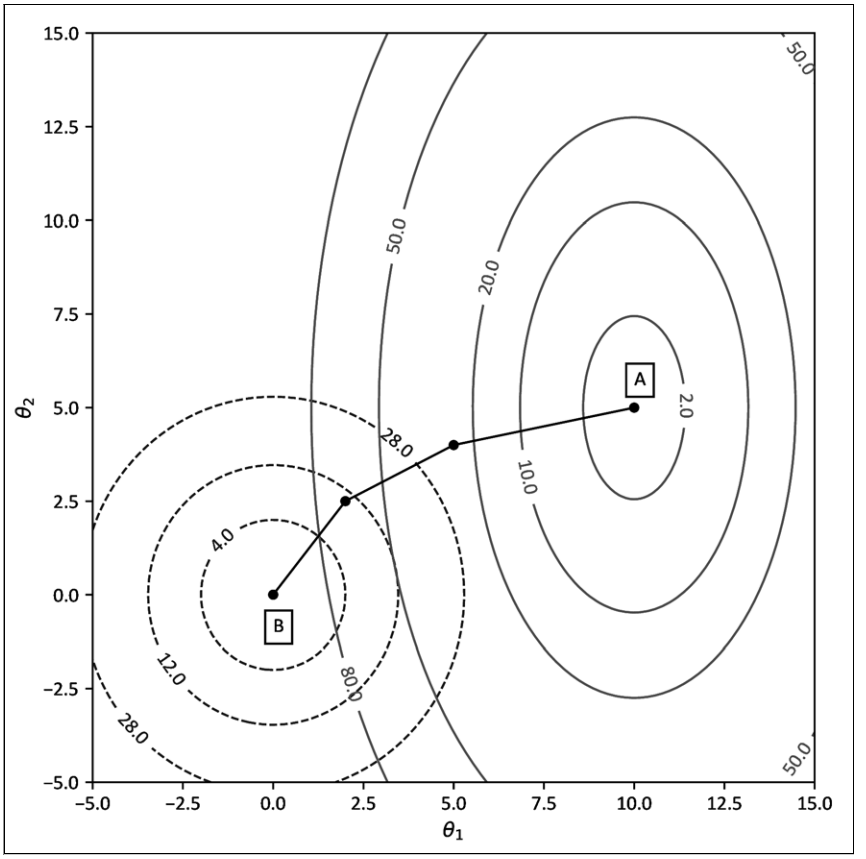


Figure 6.6 Effect of Lambda on Contour Plots and Minimizing Values

But what happens at the other extreme? Let's say we set λ to a very large value, such as *1000*. Now our loss function becomes overwhelmingly dominated by the regularization term. The whole MSE loss function doesn't matter much at all because the regularization term has all the say in the final value of the overall loss.

Effect of Weights: A Simple Example

To fully internalize what λ is doing, consider

$p = q + W \cdot r$

where *W* (our hyperparameter) is a very large number such as *1000*. If *q = 10* and *r = 0.01*, then

$$p = 10 + 1000 \cdot 0.01 = 10 + 10 = 20$$

Now, if we change *q* by adding *2* to it, *p* becomes

$$p = 12 + 1000 \cdot 0.01 = 12 + 10 = 22$$

On the other side, if we merely change *r* a tiny bit to *0.015*, then

$$p = 10 + 1000 \cdot 0.015 = 10 + 15 = 25$$

Even though we made a much smaller relative change to *r*, its effect on the final result is more significant than a large change in *q* because it's being multiplied by that large weight *W*.

When λ is very large, the MSE term becomes almost irrelevant compared to even tiny changes in the regularization term. Our gradient descent algorithm will focus almost entirely on minimizing the regularization term, with little regard for how well the model actually fits the training data.

So, what does minimizing the regularization term mean? Let's look at it again:

$$\lambda \sum_{j=1}^{n} \theta_j^2$$

When we're trying to minimize this expression and λ is fixed (even if it's large), the only way to make it smaller is to reduce the sum of squared parameters. And the absolute minimum value this sum can take is *0*, which happens when all parameters θ_j are exactly *0*.

This is a critical insight: When λ is extremely large, our model will push all parameters toward *0*, effectively ignoring the actual relationships in the data. Even increasing one parameter to a small value like *0.1* would add $0.1^2 = 0.01$ to our sum, which gets multiplied by our large $\lambda = 1000$ to add *10* units to our loss—a huge penalty compared to the minor improvement it might make to the MSE term. Refer back to Figure 6.6. In this case, when λ is very large, the only thing that affects the loss is the regularization term, which is forcing the loss to have a minimum when both parameter values are *0* (or extremely close to it). Remember, the MSE term is what's looking at our data points and measuring how well our predictions match the actual house prices. It's the part of our loss function that connects our model to reality. If we ignore it by setting λ too high, we're essentially telling our model, "Don't worry about predicting house prices accurately—just make sure all of your parameters are as close to 0 as possible!"

With all parameters at or near *0*, our model would predict virtually the same value for every house, regardless of its size or any other features. This is the opposite extreme of overfitting—we've now created a model that's too simple to capture any meaningful patterns in our data, which is a classic case of underfitting.

To summarize, consider what this means: With a large value of λ, we're saying that all values of thetas should be *0*. This feels like a mathematical quirk, but it's a fundamental reshaping of our model's underlying behavior.

In turn, we're saying that none of the inputs have any relation to the output, so we completely disconnected the inputs from the predicted output. That is obviously not what we want. Imagine trying to predict house prices without considering size, location, or any features at all! It would be like a real estate agent who gives the same price estimate for every house, regardless of whether it's a mansion or a studio apartment. But now, we have two extremes: when λ is 0, the regularization term has no effect, but when it's very large, the regularization term dominates and says that *none* of the inputs have any effect on the output. We've mapped out two boundaries of a spectrum—on one end, our model is free to create wild, complex relationships that overfit our training data, and on the other, it's constrained to the point of uselessness, ignoring all input features entirely.

What we want is to somehow control this by slowly turning the knob of λ. Think of λ as a sensitivity dial on a sophisticated instrument. Turn it too low, and the readings become chaotic and unreliable—our model overfits. Turn it too high, and the instrument becomes unresponsive—our model underfits. Somewhere in the middle lies the sweet spot where our model is responsive to genuine patterns in the data without being overly sensitive to noise.

6.1.5 Adjusting the Complexity Knob

Refer back to Figure 6.6 one last time. When we change the λ from *0* to a very high value, this takes us slowly from point A in the figure to point B. During this movement, some of the inputs are ignored less, and some are ignored more. Point A sits far from the origin, with large values for both θ_1 and θ_2. This is where our model lands when λ equals *0*—we're only concerned with minimizing the MSE, and we've found parameters that fit our training data exceptionally well. But these large parameter values suggest a complex model that might be overfitting.

Point B, in stark contrast, sits much closer to the origin. As λ increases, our model is pulled toward this simpler configuration with smaller parameter values. It's like watching a ball rolling downhill, but the landscape itself is changing as we adjust λ, creating a stronger gravitational pull toward the origin.

The black solid line connecting A and B traces the path our model takes as we gradually increase λ. Notice how the path isn't simply a straight line toward the origin—it follows a nuanced trajectory influenced by both the shape of our error surface (solid contours) and the regularization penalty (dashed circles).

This visualization reveals something profound about regularization: It doesn't simply shrink all parameters equally. As we move from A toward B, some parameters decrease

more rapidly than others. The model is making strategic sacrifices, reducing the influence of some inputs while preserving others that provide more explanatory power per unit of regularization cost.

It's similar to how a company might respond to budget cuts—not by reducing every department equally, but by strategically preserving critical functions while scaling back areas that deliver less value relative to their cost. Our model likewise preserves the most efficient parameters while penalizing those that contribute less to predictive accuracy relative to their size. When this happens, the model might decide to make the weights associated with 4+ degree polynomials 0, thus effectively reducing the model complexity. We have a way of adjusting complexity by just changing one value—the λ hyperparameter! This movement between A and B represents the fundamental trade-off at the heart of regularization—finding the sweet spot where our model is complex enough to capture true patterns in the data but simple enough to avoid fitting noise.

But how do we figure out where to stop? How do we know which value of λ gives us that perfect balance between simplicity and accuracy? Here's the plan: We already know that we can't trust our training loss. So, we're going to set λ to a particular value, run the whole training/testing, and come up with a final test loss value for this particular λ value. Then, we'll change the λ value and rerun the whole thing again—this time getting a different overall test loss. Remember, training loss isn't important. That is always going to be minimum for an overfitting model.

So, we end up with this experimental process:

1. Choose a specific λ value.
2. Train a model using this λ value on your training data.
3. Evaluate the model's performance on test data to get a test loss.
4. Record the λ value and corresponding training and test losses.
5. Select a different λ value.
6. Repeat steps 2–5 for multiple λ values.
7. Plot the relationship between λ values and training and test losses.
8. Identify the λ value that produces the lowest test loss.

Each experiment gives us a point on a curve showing the relationship between regularization strength and generalization performance. It's like a scientist methodically testing different conditions to find the optimal formula. A typical example of this plot is shown in Figure 6.7.

The figure illustrates the dance between underfitting and overfitting as we adjust our regularization strength. Take a careful look at the x-axis—it's actually plotted in reverse, with high λ values on the left (*1.0*) decreasing to *0* on the right. This reverse scaling helps us visualize the journey from simple to complex models as we move from left to right.

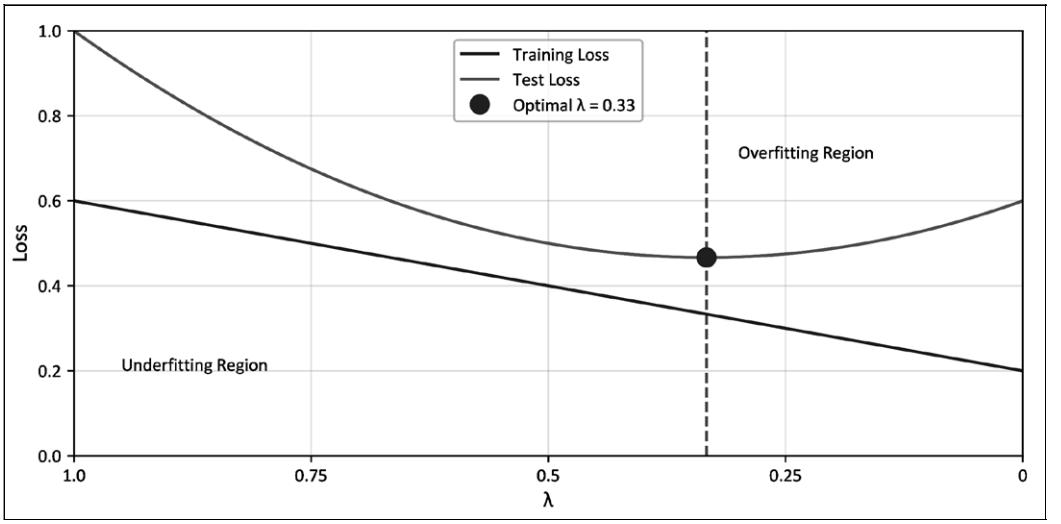


Figure 6.7 Effect of Regularization on Training and Testing Losses

When we start on the left side with a high λ value, we're firmly in the underfitting region. Here, our model is playing it too safe—most parameters are essentially *0*, and the model is largely ignoring the patterns in our data. The training loss is high because our model isn't capturing much of anything, and the test loss is equally disappointing.

As we begin to decrease λ, moving rightward on the graph, both training and test loss start to decrease. This is the sweet spot of learning—our model is gradually embracing more complexity, discovering genuine patterns in the data rather than just playing it safe. The model is learning something meaningful from the data, and this learning generalizes well to new examples. But notice what happens as we continue moving right, further decreasing λ below *0.33*. The training loss continues its downward trajectory—after all, with less regularization, our model can fit the training data more and more precisely. It's like a student who memorizes the textbook examples perfectly.

However, the test loss reaches its minimum at $\lambda = 0.33$ (marked by the dot and vertical dashed line) and then begins to climb back up. This inflection point tells us something crucial: our model has started to overfit. It's no longer just learning meaningful patterns; it's starting to memorize the quirks and noise in our training data. Like an actor who rehearses so rigidly that they can't adapt to unexpected circumstances, our model is losing its ability to generalize.

The Goldilocks Zone

The visualization in Figure 6.7 perfectly captures the Goldilocks principle in machine learning: With high λ, our model is too simple (underfitting); with low λ, it's too complex (overfitting); but at $\lambda = 0.33$, it's just right—complex enough to learn meaningful patterns but not so complex that it gets distracted by noise.

This vertical line marks this optimal value—the point where we should stop reducing λ and declare this as our best model. It has just the right amount of complexity, balancing the need to fit the training data against the risk of memorizing noise. It's found the sweet spot between ignoring the data (underfitting) and believing everything it sees (overfitting). What's particularly elegant about this approach is that we didn't need to manually specify how complex our model should be. Instead, by systematically exploring different λ values and measuring their effect on test performance, we let the data itself tell us the right level of complexity. This data-driven approach to model selection is at the heart of modern machine learning.

6.1.6 Do I Need More Data

Now we're in the position to answer that difficult question in machine learning practice: Should I get more data to make my machine learn better. The answer is "not always." It's a bit like wondering if you need more ingredients to improve your cooking—sometimes the secret lies not in more stuff but in better technique—or in our case, complexity. Data collection is expensive, especially *labeled data*. Getting it in good quality is more difficult. Think of labeled data like handcrafted furniture—each piece requires careful human attention, making it valuable and time-consuming to produce. So, we need to make sure our model actually needs more data before spending the time and effort to collect it.

Before rushing to gather more examples, you first need to use whatever data is available and plot your complexity curves, as we saw in the previous subsection. This diagnostic step is like a doctor running tests before prescribing medication—we need to understand the nature of our model's struggles before we can properly address them. Then, figure out if you're in the overfitting or underfitting zone. If you're underfitting, first increase the complexity of the model by adding higher order polynomials or add complexity in other ways. This is like upgrading from a bicycle to a car when you're consistently arriving late—sometimes, the tool itself lacks the necessary capacity.

In neural networks, adding more complexity means adding more layers or more neurons. Each layer in a neural network acts like a team of specialists that processes information in increasingly sophisticated ways. The first layer might identify simple patterns such as edges or colors, while deeper layers combine these basic observations into complex concepts such as "this is a face" or "this sentence expresses disappointment." Adding more neurons is like hiring more specialists within each team, allowing for more nuanced processing.

For example, if you're building an image recognition system that struggles to distinguish between cats and dogs, adding more layers might help it recognize hierarchical features—moving from identifying basic shapes to understanding fur patterns to recognizing distinctive facial structures. Similarly, adding more neurons might help it become sensitive to subtle distinctions within these categories. When you do this,

you'll get a better understanding of which zone you're in. It's like turning up the volume on your model's capabilities to see if that solves the problem.

If you're underfitting, first make your model complex enough. Don't rush to collect more data if your current model isn't even capable of capturing the patterns in your existing dataset.

If you're overfitting, then more data can usually help because as you get more data, there are more patterns and complexity, and your model can actually find these patterns and decrease the loss. Think of it like teaching someone to recognize bird species—if you only show them three examples of eagles, they might fixate on irrelevant details such as the background color. But show them hundreds of eagles in different positions, lighting conditions, and environments, and they'll start to focus on the truly distinctive features. Additional data in the overfitting zone works because it forces your model to find generalizable patterns rather than memorizing specific examples. It's the difference between a student who has memorized a few specific math problems versus one who has solved enough variations to understand the underlying principles that apply to all problems of that type.

This strategic approach to model improvement—diagnosing the problem first, then addressing it appropriately—saves you from the costly mistake of gathering unnecessary data or the frustration of trying to force a too-simple model to perform complex tasks. It's about working smarter, not just collecting more. Once all of this is done and we have the best value for our hyperparameter λ, one final test for our model remains.

6.1.7 Reporting the Final Results: Validation Set

Hold on a minute—we've just uncovered a subtle but critical issue in our approach. Remember how we used our test set to pick our hyperparameter value λ? There's a hidden danger here that we need to address.

Think about what we've been doing so far. We used our training data to learn the model parameters, which makes perfect sense. Then, we used what we've been calling our "test set" to find the best λ value. But here's the catch: By using this test data to make decisions about our model structure, we've actually leaked information from the test data into our design process. Like peeking at part of the final exam while studying, it compromises the integrity of our evaluation.

Just as we used the training set to pick the model parameters and then used the test set to ensure that these parameters were correct, we need to somehow ensure that the λ picked using the test set is also correct. We need a truly independent evaluation that hasn't influenced any of our choices. For this, we're going to need some more data, but we used up all our data during training and testing (*hyperparameter optimization*). What now? Don't worry. We can still keep the same process but start slightly differently. In step 1 of our machine learning process, we won't split the data into just training and

testing. We'll split it in three parts: *training*, *validation*, and *test* sets. The training part is self-evident—it's the data our model learns from directly, adjusting its parameters to minimize the loss function. The second set that we use to select the best hyperparameters—what we've been calling the test set until now—will be called the validation set. It's like a practice exam that helps us fine-tune our study strategy before the real thing.

Once the hyperparameter best value is decided, we'll use this singular value to run the whole model on the third set, which we'll now call the test set as it's doing the final testing of our model. This test set is our genuine, untouched evaluation data—it's like the sealed final exam that only gets opened when everything else is ready and once you take that, you can no longer retake it.

Whatever result is achieved by the model on the test set, we report that. If it's not good, our model didn't learn well. This lets us ensure that when the model is deployed in the real world, it will likely perform just the same as it did on the test set. By maintaining this discipline, we build models that don't just perform well on data they've seen before but are truly prepared for the challenges of the real world.

The Three-Way Split of Data

This three-way split creates a clean separation of concerns:

- Training data teaches our model parameters.
- Validation data helps us choose the best hyperparameters.
- Test data gives us a true evaluation of our final model's performance.

This is the scientific method for machine learning: We form hypotheses with the training data, refine them with the validation data, and then conduct the final experiment with the test data.

To summarize this section, regularization gives us a powerful way to manage complexity, acting like a judicious editor that simplifies our model by gradually silencing unnecessary parameters—encouraging our model to generalize from patterns rather than memorize specific examples. However, this approach does come with its own challenges, particularly in computational efficiency. To calculate the best value of λ, we have to run the entire training multiple times in a loop, which might not be feasible for large-scale modern machine models.

In the next section, we'll explore an alternative technique that maintains this core philosophy of controlled complexity while addressing this limitation. The concepts we've learned until now will still be useful, but we'll find an easier way to reduce unnecessary complexity.

6.2 Dropout: Concept and Implementation

Take a look at Figure 6.8. Even though a large part of the object in front of the keyboard is obstructed, you can tell that it's unmistakably a banana. Despite this obstruction, your brain can still identify it easily. You've learned the concept of "banana-ness" so well that you don't need to see the entire fruit to recognize it even if it's surrounded by other unrelated objects.

Figure 6.8 An Obstructed Image

Neural networks, however, often struggle with this kind of robust recognition because they tend to fall into a trap known as co-adaptation.

6.2.1 The Problem: Co-Adaptation and Overfitting

Co-adaptation occurs when neurons in a network become overly reliant on one another during training. Recall that in neural networks, neurons are all connected to each other in adjacent layers, with each neuron feeding its learned features forward to neurons in the next layer. This interconnected structure is powerful, but it creates an environment where *co-dependencies* can easily form. Instead of learning independent, robust features, neurons develop intricate dependencies. This is like a study group where, instead of each student developing their own understanding of the material, they become completely dependent on each other's notes and can't function independently. In our neural network, this means that specific neurons start to compensate for the mistakes or

biases of others. Rather than learning complementary features that contribute to overall understanding, they form a fragile ecosystem where each neuron's output critically depends on precisely what other neurons are doing.

Think about our banana image again. A robust network should recognize bananas based on multiple independent features: color, shape, texture, and size. But a *co-adapted network* might develop strange dependencies where one neuron only activates if another specific neuron has activated in a particular way. Remove or alter any piece of this intricate puzzle, and the whole recognition system falls apart.

6.2.2 The Road to Memorization

This co-adaptation leads directly to overfitting, which is perhaps the most persistent challenge in machine learning. Instead of learning generalizable patterns from the training data, the network essentially memorizes the training examples. When this happens, our network performs brilliantly on training data but fails miserably when faced with new examples. The network has learned the peculiarities and even the noise in the training set rather than the true underlying patterns that would allow it to generalize to new situations.

The consequences of this co-adaptation and overfitting are far from academic. In real-world applications, they manifest as follows:

- Medical diagnosis systems that work perfectly in the lab but fail with patients from different demographics
- Facial recognition systems that can't handle varying lighting conditions
- Recommendation engines that can't adapt to shifting user preferences
- Autonomous vehicles that struggle with unfamiliar road conditions

In each case, the network has learned to rely too heavily on specific patterns in the training data rather than developing robust, generalizable features.

The elegant concept of *dropout* was designed to solve this very issue. The strategy is pretty simple: Randomly ignore some neurons during training. That's it—that's the whole strategy. By randomly "dropping out" neurons during training, we force the network to build *redundancy* and *resilience*. Each neuron can no longer rely on any other specific neuron being present, so it must learn features that are robust even when parts of the network are missing—just like how you can still recognize a banana even when most of it is covered by a white rectangle.

Let's consider how this simple concept cleverly simulates an *ensemble* of different networks to combat this problem of co-adaptation and overfitting.

6.2.3 The Ensemble Intuition Behind Dropout

Picture yourself assembling a team for an important project. Would you rather have one brilliant but temperamental expert who might not show up on the day of your presentation, or would you prefer a diverse group of solid performers who can cover for each other if someone falls ill? Most of us would choose the reliable team. This is precisely the wisdom behind dropout, as we'll discuss in the following sections.

Training an Ensemble in Disguise

At its heart, dropout is a clever illusion. While it appears we're training a single neural network, we're actually training thousands of different networks simultaneously. It's like having a theater company where, for each rehearsal, some actors randomly call in sick, forcing the remaining cast to adapt and cover their roles. When we apply dropout to a neural network with 100 neurons and set our dropout rate to 0.5 (meaning each neuron has a 50% chance of being temporarily disabled), we're effectively creating 2^{100} different possible network configurations. That's more possible networks than there are atoms in the observable universe! Each training iteration randomly samples one of these possible networks, trains it for that batch, and then moves on to another random configuration.

This is the magic of dropout: instead of training one massive, co-dependent network, we're training an implicit ensemble of thinner, more robust networks that are forced to work independently.

Connection to Other Ensemble Methods

The concept behind dropout is part of a broader class of machine learning techniques known as *ensemble methods*. In traditional machine learning, algorithms such as *random forests* improve on basic *decision trees* by training on different subsets of features and examples to create a more robust predictor.

If you're interested in exploring these connections further, looking into traditional ensemble methods such as *bagging* and *boosting* can be illuminating. These approaches explicitly train multiple complete models and then combine their predictions. What makes dropout special is that it achieves a similar ensemble effect implicitly within a single model, making it computationally efficient while still capturing the wisdom of the crowd.

The key insight that connects dropout to classical ensemble methods is this: *Diversity* in learning leads to *robustness* in prediction. By forcing different parts of the network to function independently, we create a system where errors tend to cancel out rather than compound. This ensemble intuition helps explain why, counterintuitively, deliberately handicapping our network during training by randomly shutting off neurons

actually leads to better generalization. It's the neural network equivalent of the old saying, "What doesn't kill you makes you stronger."

In the next section, we'll dive into the nuts and bolts of how dropout is actually implemented, including the subtle but important scaling adjustments needed to make the technique work properly.

6.2.4 Dropout Mechanics

Now that we understand the "why" behind dropout, let's roll up our sleeves and explore the "how." Like many brilliant ideas, dropout's implementation is surprisingly straightforward, though there are some subtle but crucial details that make all the difference, as we'll discuss in the following sections.

Two Phases: Training vs. Inference

Think of dropout as wearing two different hats—one during training and another during testing (or inference). This dual personality is key to understanding how dropout works in practice. During training, dropout behaves like that one friend who's always canceling plans at the last minute. As your network processes each batch of data, dropout randomly selects neurons and says, "Sorry, can't make it today!" These neurons are temporarily removed from the network, their outputs are set to zero, and they don't contribute to the forward pass or receive updates during backpropagation.

When it's time for the real show—the inference phase where your model makes predictions on new data—dropout suddenly becomes completely reliable. All neurons show up for work with no exceptions. This Jekyll and Hyde behavior serves a purpose. During training, the random dropout forces the network to build redundancy and resilience. During inference, we want our model to use all of its resources to make the best possible predictions.

The Mathematics of Scaling: Balancing the Books

There's a mathematical wrinkle in this that we need to iron out though. If we're turning off, say, 50% of our neurons during training but keeping them all on during inference, won't this create a mismatch in the scale of activations? This is a crucial detail that the original dropout paper addresses. Let's walk through it with a simple example.

Consider the case where you have a layer with 4 neurons, each outputting the value 2. The total output sum is 8. Now, if we apply dropout with a rate of 0.5, we randomly disable 2 of these neurons, leaving us with a sum of 4 instead of 8. That's a significant reduction!

To compensate for this reduction during training, we scale up the remaining activations by dividing by *(1 – dropoutRate)*. In our example, we'd multiply the output of each

remaining neuron by *1/(1 – 0.5) = 2*. So, our 2 remaining neurons now output 4 each, giving us a total of 8 again—matching the expected magnitude of the full network.

Inverted Dropout: The Modern Approach

In early implementations, the scaling correction was applied during inference—all activations were multiplied by *(1 – dropoutRate)*. However, this approach has a disadvantage: You need to apply different operations depending on whether you're in training or inference mode.

Modern implementations use what's called *inverted dropout*, which flips this approach. With inverted dropout, the following occurs:

- During training, we scale up the remaining activations immediately.
- During inference, no scaling is needed at all.

This shift simplifies deployment by ensuring the inference-time computation is exactly what you'd expect without dropout. In code, inverted dropout looks like that given in Listing 6.2.

```
def inverted_dropout(x, dropout_rate):
    if training:
        # Generate binary dropout mask
        mask = np.random.binomial(1, 1-dropout_rate, size=x.shape)
        # Apply mask and scale
        return (x * mask) / (1 - dropout_rate)
    else:
        # During inference, no changes
        return x
```

Listing 6.2 Inverted Dropout in Python

6.2.5 Finding the Sweet Spot: Dropout Rates in Practice

Not all layers are created equal when it comes to dropout. Through years of experimentation, the deep learning community has developed some rules of thumb for dropout rates:

- **Input layers**
  Light dropout (0.1–0.2) or none at all.
- **Hidden layers**
  Moderate dropout (0.3–0.5).
- **Very deep networks**
  Increasing dropout for deeper layers.
- **Convolutional layers**
  Lower dropout rates than fully connected layers.

- **Recurrent layers**
  Apply dropout carefully, often with specialized techniques.
- **Transformers**
  Typically use dropout rates around 0.1.

We'll get to the *convolution* and *transformer* layers in Chapter 7 and Chapter 9, respectively. The intuition here is that earlier layers learn more general features that are less prone to overfitting, while deeper layers learn more specialized features that benefit more from regularization.

It's worth noting that dropout isn't a one-size-fits-all solution. Some architectures (particularly very deep ones or those with skip connections) may benefit from different dropout strategies or alternative regularization techniques. Like any good recipe, the key is experimentation and adaptation to your specific ingredients.

In the next section, we'll put theory into practice by visualizing exactly what happens to your network's internal representations when dropout is applied. Seeing is believing, and these visualizations will give you a concrete understanding of how dropout reshapes your model's learning process.

6.2.6 Implementing Dropout in Pure Python

Let's roll up our sleeves and build dropout from the ground up. There's something magical about implementing an algorithm yourself—it transforms an abstract concept into something tangible that you can poke, prod, and truly understand. Like learning to bake bread from scratch instead of buying it at the store, coding dropout yourself gives you insights you'd never get from just importing a library. So, let's create a simple Dropout layer in pure Python using Listing 6.3. The complete code for this section can be seen in the book resources page on *https://recluze.net/keras-book* in the notebook **06-02-dropout-scratch** and on the book's official web page.

```
import numpy as np

class Dropout:
    def __init__(self, dropout_rate=0.5):
        self.dropout_rate = dropout_rate
        self.mask = None
        self.training = True

    def forward(self, inputs):
        if not self.training:
            return inputs
```

```
        # Create a binary mask: 1 for keep, 0 for drop
        self.mask = np.random.binomial(1, 1 - \
                    self.dropout_rate, size=inputs.shape)

        # Apply the mask and scale by the keep probability
        # This is inverted dropout
        outputs = (inputs * self.mask) / (1 - self.dropout_rate)

        return outputs

    def backward(self, grad_outputs):
        # Apply the same mask to the gradient
        return (grad_outputs * self.mask) / (1 - self.dropout_rate)

    def set_mode(self, training):
        self.training = training
```

Listing 6.3 Dropout Layer in Pure Python

Let's take a moment to digest what's happening here. Our `Dropout` class is like a gatekeeper that neurons must pass through. During training, this gatekeeper randomly stops some neurons from passing (setting them to 0) and gives a boost to the ones that make it through (scaling them up). During inference, the gatekeeper steps aside and lets everyone through without any interference.

To really understand what's happening, let's create a toy example and see dropout in action through Listing 6.4. Consider having a small layer with just 10 neurons, and we're applying a dropout rate of 0.5, meaning roughly half the neurons will be deactivated.

```
# Let's create a toy example
np.random.seed(42)  # For reproducibility

# Create an input with 10 neurons, each with activation 1.0
inputs = np.ones(10)
print("Original inputs:", inputs)

# Apply dropout during training
dropout = Dropout(dropout_rate=0.5)
dropout.set_mode(training=True)
outputs_training = dropout.forward(inputs)
print("Mask:", dropout.mask)
print("Training outputs:", outputs_training)
```

```
# Now, let's see what happens during inference
dropout.set_mode(training=False)
outputs_inference = dropout.forward(inputs)
print("Inference outputs:", outputs_inference)
# Outputs ---------
Original inputs: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Mask: [1 0 1 0 1 1 0 1 1 0]
Training outputs: [2. 0. 2. 0. 2. 2. 0. 2. 2. 0.]
Inference outputs: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Listing 6.4 Using the Custom Dropout Layer

Notice how during training, some outputs become 0 (where the mask is 0), while others are scaled up to 2.0 (because our `dropout_rate` is 0.5, we divide by 0.5, which doubles the value). But during inference, all outputs remain at their original values of 1.0. This is inverted dropout in action—scale during training, and do nothing during inference. Keras uses something very similar to this layer in its implementation. So, you can use Keras's implementation but be aware of some common pitfalls that you might encounter when using it, as we'll discuss next.

6.2.7 Common Pitfalls and Debugging Tips

When implementing dropout, there are a few common pitfalls to watch out for:

- **Forgetting to switch modes**
  The most common mistake is forgetting to set `training=False` during inference. This would apply random dropout to your production predictions! Always double-check your mode transitions.
- **Incorrect scaling**
  If you scale during inference instead of training (noninverted dropout), you might accidentally apply the scaling factor twice. This manifests as predictions that are consistently too small by a factor of *(1 – dropoutRate)*.
- **Too much dropout**
  Applying too high a dropout rate, especially in smaller networks, can prevent the model from learning anything useful. If your model is underfitting, try reducing the dropout rate.
- **Memory leaks**
  In our simple implementation, we store the dropout mask for backpropagation. In a production system with many layers, these masks can consume significant memory. Consider clearing them after backpropagation if you're rolling out your own `Dropout` code.

- **Uneven distribution**
  Random dropout should follow a *binomial distribution*. If you implement it with a simple threshold on uniform random values, check that you're not introducing subtle biases.

To debug dropout issues, try these approaches:

- Temporarily set the dropout rate to 0 and see if your model converges. If it does, gradually reintroduce dropout.
- Print the proportion of 0s in your dropout masks to verify it matches your intended dropout rate.
- Look at activations before and after dropout to ensure the scaling is working correctly.
- Check gradients flowing through Dropout layers during backpropagation to confirm they're properly masked and scaled.

Now that you've built dropout from scratch, you'll have a much deeper appreciation for what's happening under the hood when you use high-level libraries such as Keras or TensorFlow. Speaking of which, in the next section, we'll see how to implement dropout in Keras with just a few lines of code, using all the insights we've gained so far.

6.3 Other Regularization Methods: L1 and L2 Regularization

So far, we've explored dropout as a powerful technique to combat overfitting. Think of dropout as randomly silencing neurons during training, forcing the network to build redundancy and more robust feature detection. But dropout isn't the only tool in our regularization toolkit.

In this section, we'll turn our attention to two other important regularization techniques: L1 and L2 regularization. These approaches tackle overfitting from a different angle—instead of randomly shutting down neurons, they put constraints on the magnitude of the weights themselves. At their core, both L1 and L2 regularization add an extra term to our loss function that penalizes large weights. The mathematical approach is straightforward. If our original loss function is denoted as $L_{original}$, our regularized loss becomes

$$L_{regularized} = L_{original} + \lambda \times \text{penalty}$$

where λ controls how much we care about the penalty compared to our original objective. A larger *lambda* means we're more concerned about keeping weights small than minimizing the original loss.

6.3.1 L1 Regularization (Lasso)

L1 regularization, also known as *Least Absolute Shrinkage and Selection Operator* (*Lasso*), adds a penalty equal to the absolute value of the weights. Mathematically, we add this term to our loss function:

$$L_{L1} = L_{original} + \lambda \sum_{i=1}^{n} |w_i|$$

This simple modification creates some interesting and highly useful effects on how our model learns. The most notable characteristic of L1 regularization is its tendency to push weights exactly to 0, effectively removing certain features from consideration. While L2 regularization reduced the weights of less important features, L1 regularization is like a harsh judge who says, "If that feature isn't absolutely necessary, don't use it at all."

Consider a situation where you're trying to predict house prices, and you have dozens of potential factors—number of bedrooms, location, square footage, age of the house, and even more specific details like the color of the front door or the type of doorknobs. L1 regularization helps you identify which features actually matter by setting the weights for irrelevant features (perhaps doorknob type) to exactly 0, while maintaining nonzero weights for important features such as location and square footage. This *feature selection* property makes L1 particularly valuable when you suspect many of your inputs might be irrelevant or redundant. It leads to *sparse models*—models that only use a subset of available features—which are often easier to interpret and can be more efficient to deploy.

On the other hand, L2 regularization also has a nice mathematical property: It works particularly well when you have correlated features. For example, if both "square footage of living space" and "number of rooms" help predict house prices (and they're obviously related), L2 will distribute the importance between them rather than arbitrarily picking one over the other.

6.3.2 Elastic Net: Combining L1 and L2

What if we want the best of both worlds—the feature selection properties of L1 and the stability benefits of L2? Enter *Elastic Net*, which simply combines both penalties:

$$L_{Elastic} = L_{original} + \lambda_1 \sum_{i=1}^{n} |w_i| + \lambda_2 \sum_{i=1}^{n} w_i^2$$

Elastic Net gives us fine-grained control over regularization by letting us adjust the relative importance of L1 versus L2 penalties. It's like having two different teachers in the classroom—one focused on removing distractions entirely and another on making sure everyone participates without anyone being too dominant.

This hybrid approach shines in scenarios where you have many correlated features but suspect only some are truly relevant. For instance, in genomic studies where thousands of genes might be measured but only a small subset actually influence the trait being studied, Elastic Net can identify groups of related important genes rather than arbitrarily selecting one representative from each correlated group (as L1 might do) or spreading importance too thinly across all of them (as L2 might do).

6.3.3 When Not to Use Regularization

Now that we've explored the powerful regularization techniques in our arsenal, let's talk about when to put these tools back on the shelf. Regularization isn't always the answer, and knowing when to avoid it is just as important as knowing how to apply it. The first and most obvious scenario is when you're already facing underfitting. Adding regularization to a model that's struggling to capture the underlying patterns in your data is like giving low calorie diet advice to someone who's already underweight. If your training loss is high and your model can't even fit the training data well, adding regularization will only make the problem worse.

Very small datasets present another case where regularization might do more harm than good. When data is scarce, your model needs to squeeze every bit of information from the limited examples available. Regularization, by design, restricts the model's capacity to fit the data perfectly. With very small datasets, the risk of overfitting is naturally reduced because there's simply not enough data to memorize. In these cases, you might find better performance by letting your model use its full capacity without regularization constraints.

There are also situations where *interpretability* is your primary concern. L1 regularization can be helpful here because it creates sparse models, but sometimes you need to retain all features to understand their relationships properly. For instance, in medical research, removing certain variables through aggressive regularization might eliminate factors that doctors need to see, even if they only have a small effect on the prediction. It's like a detective needing to consider all the evidence, not just the most compelling pieces.

Interpretability is a very active area in modern deep learning research. Modern models work so well that they are being used in almost all domains you can think of. The issue though is that researchers and engineers aren't really sure why and how the results are so accurate. *Interpretable machine learning* is an active area of research that aims to create tools and methods for figuring out why the models are working and what different learned weights mean, as well as to improve the models in the process.

Interpretable Machine Learning

I find the area of interpretability to be highly interesting. While it's beyond the scope of this book, I highly recommend you check out the work by Chris Olah (*https://colah.github.io*) as an introduction to this area.

Finally, consider your *domain knowledge*. If you have strong prior information that all of your features are relevant, regularization might be unnecessarily discarding valuable information. For example, if you've carefully crafted a set of financial indicators for stock prediction based on years of economic theory, you probably don't want regularization to arbitrarily zero out some of these hard-won features.

6.3.4 Practical Considerations: Dropout vs. L1/L2 in Neural Networks

While understanding L1 and L2 regularization is essential for your machine learning toolkit, it's worth noting that dropout has become the dominant regularization technique for deep neural networks in practice. The reason lies in computational efficiency and convenience. Both L1 and L2 require careful tuning of the regularization strength parameter λ. Finding the optimal λ often involves running multiple training sessions with different values, which becomes prohibitively expensive for large neural networks that might take days or weeks to train once, let alone multiple times.

Dropout, by contrast, typically works well with default settings (dropping 20%–50% of neurons), making it more of a plug-and-play solution. It also integrates naturally into the forward and backward passes of neural network training without requiring additional hyperparameter searches.

That said, in smaller models or when you have specific goals such as feature selection, L1 and L2 regularization remain valuable tools. Many practitioners even combine techniques—using dropout between layers while also applying a small L2 penalty on the weights. It's important to understand the strengths and limitations of each approach, allowing you to make informed choices based on your specific needs rather than blindly applying the same technique to every problem. Regularization is both art and science—it requires not just mathematical understanding but also intuition developed through practice. This intuition can be developed by repeatedly using it in actual code. So, let's do such an implementation in the next section.

6.4 Applying Regularization in Keras

In this section, we'll take a look at how both L2 regularization and dropout can be applied in Keras. Adding these powerful regularization techniques to your models is surprisingly straightforward.

6.4.1 Implementing L2 Regularization in Keras

In Keras, adding L2 regularization is refreshingly easy, as shown in Listing 6.5.

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.regularizers import l2

# Adding L2 regularization to a Dense layer
x = Dense(units=128,
          activation='relu',
          kernel_regularizer=l2(0.001))(inputs)
```

Listing 6.5 L2 Regularization in Keras

That's it! One parameter, and you've transformed your model. The `kernel_regularizer=l2(0.001)` argument tells Keras to apply L2 regularization to the layer's weights (but not biases) with a regularization strength of 0.001. You can visit the official documentation of Keras to take a look at other options that are available for regularization here: *https://keras.io/api/layers/regularizers*.

6.4.2 Dropout in Keras

Finally, let's see how we can implement dropout in Keras with just a few lines of code. The beauty of Keras is that it handles all the complex machinery we explored earlier, wrapping it in a clean, intuitive interface that lets us focus on designing our architecture rather than managing implementation details.

In Keras, adding dropout to your model is remarkably straightforward. It's akin to installing a quality control checkpoint between assembly lines in a factory. Each checkpoint randomly inspects some products and lets others pass through without scrutiny. Take a look at how we can create a simple neural network with Dropout layers in Listing 6.6.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
import numpy as np

# Create a simple model with Dropout
model = Sequential([
    # Input layer
    Dense(128, activation='relu', input_shape=(784,)),
    # Dropout layer with 30% rate
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
```

```
])

# Compile the model
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Listing 6.6 Dropout Usage in Keras

That's it! With these few lines, we've created a network that automatically applies dropout during training. Notice how we place the `Dropout` layers directly after the `Dense` layers we want to regularize. The parameter we pass to `Dropout(0.3)` indicates the fraction of neurons that will be randomly deactivated during each training pass—in this case, 30% of the neurons in the first hidden layer. When we train this model, Keras handles all the complexity behind the scenes. During each training batch, it randomly selects different neurons to deactivate, forcing the network to learn redundant representations. But here's the really clever part—when it comes time to make predictions, Keras automatically switches to using all neurons with appropriately scaled weights. You don't need to write any additional code to handle this transition between training and inference behaviors.

There are a few key options for the Dropout layer that are worth knowing:

- `rate`
  This is the fraction of units to drop, typically between 0.2 and 0.5. Higher values mean more aggressive regularization but might slow down learning.
- `seed`
  If you want reproducible results, you can set a random seed to ensure the same neurons are dropped in each run.
- `noise_shape`
  This allows you to specify the shape of the binary dropout mask, giving you finer control over which dimensions get dropped.

For most applications, you'll only need to adjust the dropout rate. Finding the right rate often requires experimentation. Too little dropout, and your model might still overfit. Too much, and it might struggle to learn patterns at all. One common pattern is to use higher dropout rates for larger layers (those with more neurons) and lower rates for smaller layers. You can also experiment with gradually increasing dropout rates as you go deeper into the network.

Let's see a complete example of training our model with dropout in Listing 6.7.

```
# Generate some dummy data
# 1000 examples, 784 features
x_train = np.random.random((1000, 784))
# 10 classes
y_train = np.random.randint(10, size=1000)

# Train the model
history = model.fit(
    x_train, y_train,
    epochs=10,
    batch_size=32,
    validation_split=0.2
)
```

Listing 6.7 Using Dropout with Dummy Data

When you run this code, you'll notice something interesting in the training metrics. Initially, the training accuracy might be lower than you'd expect for such a simple dataset. That's dropout at work! During training, the network is deliberately handicapped by having some neurons turned off. Sort of like if you ask a football team to play with only 7 players instead of 11, performance will suffer.

But when evaluation time comes on the validation set, Keras automatically switches to using all neurons with scaled weights. You'll often see a significant gap between training and validation performance, with validation surprisingly outperforming training. This isn't a bug—it's a sign that dropout is working as intended. Your model is learning to be robust.

Remember, the whole point of regularization isn't to achieve the highest possible training accuracy. It's to build a model that generalizes well to new, unseen data. Dropout helps us achieve that by preventing our network from becoming too reliant on any specific neuron.

In the next section, we'll explore other forms of regularization available in Keras and discuss strategies for combining them effectively.

6.4.3 Beyond Basic Dropout: Specialized Variants

While standard dropout has proven remarkably effective across many architectures, it sometimes introduces unexpected complications in specific network designs. As we continue to refine our understanding of neural networks, researchers have developed specialized dropout variants to address these unique challenges, as we'll discuss in the following sections.

The Challenge with Self-Normalizing Networks

Try to picture a beautiful fountain with multiple levels, where each basin perfectly controls the water flow to the next. The water pressure at each level stays consistent, creating a harmonious system. Now, what happens if you randomly block some of the water spouts? The carefully balanced pressure throughout your fountain system becomes disrupted.

This is similar to what happens in *self-normalizing neural networks* that use *Scaled Exponential Linear Unit* (*SELU*) activation functions. These networks are designed to maintain a specific statistical distribution of activations across layers—essentially keeping the "water pressure" steady throughout the network. Standard dropout disrupts this carefully balanced system by randomly zeroing activations.

AlphaDropout: Preserving the Statistical Character

AlphaDropout offers an elegant solution to this problem. Instead of simply setting values to 0, which shifts the mean and variance of the layer, AlphaDropout replaces dropped values with noise that maintains the layer's statistical properties. Let's take a look at a model using AlphaDropout in Listing 6.8.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, AlphaDropout

# Create a self-normalizing neural network with AlphaDropout
model = Sequential([
    Dense(128, activation='selu', input_shape=(784,)),
    AlphaDropout(0.2),  # Maintains mean and variance of SELU activations
    Dense(64, activation='selu'),
    AlphaDropout(0.3),
    Dense(10, activation='softmax')
])
```

Listing 6.8 AlphaDropout in Keras

What makes AlphaDropout special is its mathematical design. When using SELU activations, our networks develop a particular statistical property—activations tend to have a mean of 0 and variance of 1 (what statisticians call the *standard normal distribution*). It's specifically designed for networks using SELU activation functions. If you're building a self-normalizing neural network with SELU activations, AlphaDropout should be your regularization method of choice. Its parameter options are virtually identical to standard dropout as shown in Listing 6.9.

```
AlphaDropout(
    rate=0.2,  # Fraction of units to drop
    seed=None  # Optional random seed for reproducibility
)
```

Listing 6.9 AlphaDropout Settings

Like standard dropout, AlphaDropout operates only during training and is automatically disabled during evaluation and prediction phases.

Other Specialized Dropout Variant

The Keras ecosystem offers several other specialized dropout variants worth knowing about:

- **SpatialDropout 1D/2D/3D**
  These variants drop entire feature maps rather than individual neurons, particularly useful in convolutional networks where adjacent pixels have high correlation. We'll return to these in the next chapter after we've covered another type of layer.
- **GaussianDropout**
  Instead of binary dropout, this applies multiplicative Gaussian noise to the inputs, achieving a similar regularizing effect with different statistical properties.
- **GaussianNoise**
  This adds zero-centered Gaussian noise to the inputs, serving as a form of data augmentation rather than dropout.

Each variant addresses specific challenges in neural network architectures, giving you a powerful toolkit for regularization. As we explore more complex architectures in later chapters, we'll see how these specialized techniques can be vital for optimizing advanced models.

The power of Keras is that it makes these sophisticated techniques accessible through simple, consistent interfaces. Just like how modern cameras abstract away the complex physics of photography, Keras allows you to focus on the architecture of your network while it handles the mathematical intricacies under the hood. One issue remains though: How do we figure out the dropout rate when we're actually applying dropout?

6.4.4 Finding the Perfect Dropout Rate: A Systematic Approach

So far, we've talked about dropout rates as if they were values you should just intuitively know. "Use 0.2 here, maybe 0.5 there"—but where do these numbers come from? The truth is, finding the ideal dropout rate is a bit like finding the perfect amount of spice in a recipe. Too little, and your model might still overfit. Too much, and your network might struggle to learn anything useful. In practice, the optimal dropout rate depends on many factors:

- The complexity of your dataset
- The depth and width of your network
- The amount of training data available
- The specific patterns your network needs to learn

Rather than relying on rules of thumb or intuition, we can let systematic experimentation find the answer for us. This is where *Keras Tuner* comes to our rescue, allowing us to try multiple dropout rates and see which one yields the best performance.

Let's see how we can systematically search for the best dropout rate using Keras Tuner. You might have to install the Tuner using the following command:

```
pip install keras-tuner
```

If you're in Google Colab or in a Jupyter Notebook, you can issue the magic command:

```
!pip install keras-tuner
```

The code begins by loading the MNIST dataset, as shown in Listing 6.10, which contains images of handwritten digits. We've done this a few times already, so we'll go through it quickly. The complete code for this section can be seen in the book resources page on *https://recluze.net/keras-book* in the notebook **06-03-keras-tuner**.

```
import keras
import keras_tuner as kt
from keras import layers
from keras.datasets import mnist
from sklearn.model_selection import train_test_split

(X, y), (_, _) = mnist.load_data()
X = X.reshape(X.shape[0], -1) / 255.0
X_train, X_test, y_train, y_test = \
      train_test_split(X, y, test_size=0.2)
```

Listing 6.10 Loading and Shaping the Data

This section loads the MNIST dataset, normalizes pixel values to [0,1] by dividing by 255, and splits data into 80% training and 20% testing sets with a fixed random seed for reproducibility. Then, we get to the heart of *hyperparameter tuning*, which is defining which aspects of the model should be adjustable. The `build_model` function shown in Listing 6.11 creates a neural network with *tunable* hyperparameters.

```
# A model-building function that takes a hyperparameter object
def build_model(hp):
    model = keras.Sequential()
```

```
    # Add our first layer with tunable units
    model.add(layers.Dense(
        units=hp.Int('units_1', min_value=32, \
              max_value=128, step=32),
        activation='relu',
        input_shape=(784,)
    ))

    # Add our first Dropout layer with a tunable rate
    model.add(layers.Dropout(
        rate=hp.Float('dropout_1', min_value=0.2, \
                               max_value=0.4, step=0.1)
    ))

    # Output layer
    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model
    model.compile(
        optimizer=keras.optimizers.Adam(),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model
```

Listing 6.11 Building the Model for Parameter Tuning

This function defines a neural network with the following:

- A tunable first `Dense` layer where the number of neurons (units) can vary from 32 to 128 in steps of 32
- A tunable `Dropout` layer where the dropout rate can range from 0.2 to 0.4 in steps of 0.1
- A fixed output layer with 10 neurons (one for each digit) and softmax activation

The `hp` parameter is a hyperparameter object provided by Keras Tuner that allows us to define searchable parameters. For each parameter, we specify the following:

- A unique name (e.g., `units_1` or `dropout_1`)
- The range of values to search (`min_value` to `max_value`)
- The *step size* between values

The model uses Rectified Linear Unit (ReLU) activation for the hidden layer and softmax for the output layer, which is standard for classification tasks. It's compiled with the

Adam optimizer and sparse categorical cross-entropy loss function that we've already seen previously.

Once we've defined our model structure and tunable parameters, we set up and run the hyperparameter tuning process, as shown in Listing 6.12.

```
# Create a tuner with our model builder
tuner = kt.Hyperband(
    build_model,
    objective='val_accuracy',
    max_epochs=10,
    factor=3,
    directory='dropout_tuning',
    project_name='mnist_dropout'
)

# Start the search
tuner.search(X_train, y_train,
             epochs=30,
             validation_split=0.2)
```

Listing 6.12 Keras Tuner Execution

This sets up a *hyperband tuner*, which will find the optimum values for us, with these specifications:

- Uses our `build_model` function as a template
- Aims to maximize validation accuracy (`val_accuracy`)
- Trains models for a maximum of 10 epochs in its final round
- Prunes underperforming models at a level of aggression determined by the factor of 3

The results of optimum hyperparameter search are stored in a directory called `dropout_tuning` under the project name `mnist_dropout`. The search method initiates the hyperparameter search by following these steps:

- It trains on our training data (`X_train`, `y_train`).
- Each model can train for up to 30 epochs.
- Of the training data, 20% is set aside for validation during training.

After the search completes, we retrieve and use the best hyperparameters through the code shown in Listing 6.13.

```
# Get the best model and hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"Best dropout rate for first layer: {best_hps.get('dropout_1')}")
print(f"Best units for first layer: {best_hps.get('units_1')}")
```

Listing 6.13 Getting Keras Tuner Results

The code extracts the best hyperparameters based on validation accuracy. The `get_best_hyperparameters` method returns an ordered list of hyperparameter sets, with the best-performing configuration first. By specifying `num_trials=1`, we retrieve only the single best result. The code then prints both the optimal dropout rate and the optimal number of units (neurons) found for the first layer. The output of the tuner when I ran it is shown in Listing 6.14. For this run, the best dropout rate it found was 0.2 and the number of units was 128. These values represent the hyperparameter combination that produced the highest validation accuracy during the tuning process.

```
Trial 3 Complete [00h 00m 10s]
val_accuracy: 0.9292708039283752

Best val_accuracy So Far: 0.9512500166893005
Total elapsed time: 00h 00m 35s

Search: Running Trial #4

Value             |Best Value So Far |Hyperparameter
128               |128               |units_1
0.3               |0.2               |dropout_1

Best dropout rate for first layer: 0.2
Best units for first layer: 128
```

Listing 6.14 Results of the Keras Tuner

We can use these optimal values to construct our final model, knowing it has been specifically tuned for this dataset with the ideal dropout rate and neuron count in the first layer. As you can see, we can optimize any parameter value, including the number of neurons in the layer. This systematic approach to *hyperparameter optimization* gives us confidence that we're using dropout effectively, neither under-regularizing nor over-regularizing our model.

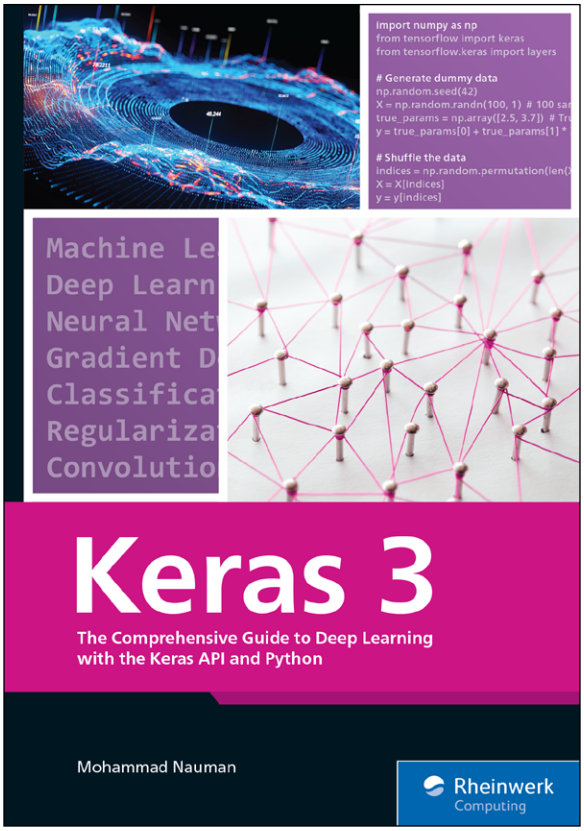
Hyperparameter Tuning in Production

Once the best values of hyperparameters are found, the model is updated to just use the parameter values instead of the `hp` parameter. For instance, the preceding model will be replaced with the following:

```
model.add(layers.Dense(128, activation='relu', \
                       input_shape=(784, )))
model.add(layers.Dropout(0.2))
```

6.5 Summary

This chapter examined regularization techniques to solve overfitting in machine learning models. It began by explaining overfitting and underfitting through examples of polynomial features and model complexity, introducing regularization terms as a solution. The chapter then explored dropout, presenting its concept, mechanics, and implementation details to prevent neurons from co-adapting. It covered L1 and L2 regularization methods, explaining their mathematical foundations and appropriate use cases. The final section demonstrated practical implementations of these techniques in Keras, including code examples for L2 regularization, Dropout layers, and specialized dropout variants. The chapter concluded with systematic approaches for finding optimal dropout rates for different model architectures. We're now in the position to move on to more advanced models and techniques, starting with a powerful type of model in the next chapter that's especially good at working with images. All the fundamentals we've built until now will be reused time and again to support us.



Mohammad Nauman

Keras 3

The Comprehensive Guide to Deep Learning with the Keras API and Python

- Learn to use Keras for deep learning
- Work with techniques such as gradient descent, classification, regularization, and more
- Build and train convolutional neural networks, transformers, and autoencoders

www.sap-press.com/6142

Dr. Mohammad Nauman is a seasoned machine learning expert with more than 20 years of teaching experience. He has taught 40,000+ students globally through online learning platforms. He holds a PhD in computer science and was a post-doctoral fellow at the Max Planck Institute for Software Systems in Germany.

ISBN 978-1-4932-2739-6 • 629 pages • 11/2025

E-book: $54.99 • **Print book:** $59.95 • **Bundle:** $69.99

Rheinwerk Publishing