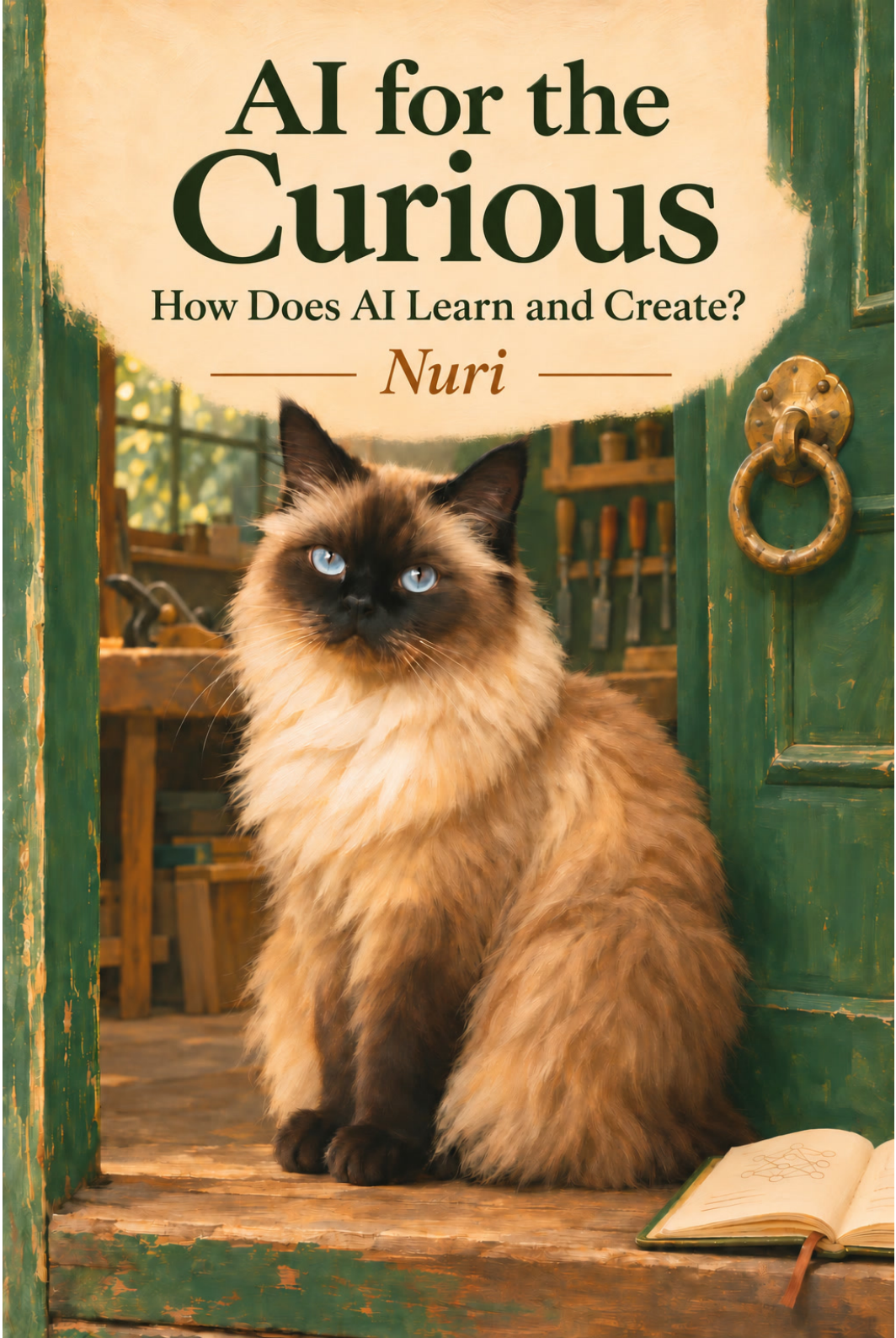
AI for the Curious

How Does AI Learn and Create?

Nuri



AI for the Curious

How Does AI Learn and Create?

Nuri

Version 1.0 | November 23, 2026

This book is dedicated to stray animals and to the people who feed them and show them kindness.

Publication Information

AI for the Curious
How Does AI Learn and Create?

Author: Nuri
Contact: exforge@x386.org

Editorial development and content support: Charlie, the name used here for OpenAI's AI system

About this edition

This first edition includes all thirteen chapters, the introductory sections, closing reflections, and a glossary. Reader feedback and corrections will be considered for future versions.

If you're new to AI assistants, *AI for Everyone* offers practical examples of how to ask a question, check an answer, and protect your personal information.

Version: 1.0
Date: November 23, 2026

License

The written content of this book is available under the **CC BY-SA 4.0** license. You may copy, distribute, and adapt it, including for commercial purposes. You must credit the author, link to the license, indicate any changes you have made, and share adaptations under the same license.

License summary and full terms

This brief description does not replace the legal text of the license.

Unless otherwise indicated, original software developed to produce the book will be released under **GPL-3.0-or-later**.

Third-party fonts and other assets are subject to their respective licenses.

Contents

Introduction 1
Who Is This Book For? 3
How to Use This Book 4
Learning Roadmap 5
Chapter 1 · Model training, parameters, and backpropagation 8
Chapter 2 · Next-token prediction and cross-entropy in language models 16
Chapter 3 · Token vectors, attention, and softmax 22
Chapter 4 · Roles, context, and answer generation in chat models 29
Chapter 5 · Tool calls and data freshness 34
Chapter 6 · Searching documents and answering from sources (RAG) 38
Chapter 7 · Image perception, pixels, and visual features 42
Chapter 8 · Generating images from text with diffusion 48
Chapter 9 · Generating images from a reference and editing with a mask 54
Chapter 10 · Sound perception, waveforms, and spectrograms 58
Chapter 11 · Speech recognition and text-to-speech synthesis 63
Chapter 12 · Perceiving motion in video and generating video 68
Chapter 13 · Sound and picture in sync, event order, and causality in video 73
Looking Back and Closing Reflections 78
Glossary 80

Introduction

Say “The door is open” into your phone and those four words appear on the screen. Now whisper them. The transcript may stay exactly the same, though anyone listening to the two recordings would hear the difference at once. How does the phone find the same words when your voice has changed so much? Now imagine going the other way. If all we have is the line “The door is open,” how does a system reading it aloud decide where to pause or which word to stress?

In our first book, *AI for Everyone*, we looked at when the systems we encounter in daily life work well and how far we can trust their answers. In this book we’ll explore how those answers come about. We’ll spend some time in a small workshop with a machinist and an apprentice. On the first day, all we’ll have is an order book. Later we’ll take a sentence, a photograph, a short audio recording, and a few seconds of video from that same workshop and examine each in turn.

The order book tells us how many parts went into each past order and how long the job took. Suppose we estimate the time for a new job by allowing a certain number of minutes per part. If our estimates fall five minutes short on small orders and large ones alike, perhaps we need to allow for setup at the start of each job. A calculation like this will be our first **model**. We’ll change the time it allows for setup and for each part, then see how far its estimates fall from the times in the book. For this model, **learning** from examples means adjusting the numbers in the calculation to reduce prediction error. After training, we can use that same calculation for an order whose duration we don’t yet know.

A language model also has numbers learned from examples. What it predicts is how a piece of text will continue. “The machinist walked up to the workshop door and…” could continue with “opened it” or “locked it.” If you’ve just read that the apprentices are waiting outside, the first may seem more likely. We’ll see how the model draws on earlier words and how each new piece it selects affects the next choice. Then we’ll have the apprentice ask the phone a harder question. “Is the workshop door locked right now?” A sentence about the machinist locking up that morning and a reading just taken from the lock give us different grounds for answering.

Describe the green workshop door and an image model can draw it, brass knocker and all. The machinist’s own knocker, though, has a

small notch in its rim. What changes if we give the model a photograph so that the notch can appear in the drawing, too? We'll look at how a photograph is represented by tiny points of color, how a model uses the relationships among them, and which details can change when it generates a new picture.

With sound and video, time enters the calculation. The silence between two knocks helps us count them. Look at a single photograph of a door standing ajar and it's hard to say whether the door is opening or closing. Watch several pictures in the order they were taken and the direction becomes clear. Throughout the book we'll use small experiments like these to see what a model can infer from the clues available to it and which details it fills in when it generates a new sound or video.

Who Is This Book For?

We wrote this book for readers who look at an AI model's answer and wonder, "How did it get there?" You might recognize your own cat in a photograph even if half its face is in shadow. What relationships among the light and dark points in that same picture could a model learn that would help it tell a cat from another animal? Curiosity about something we do with ease is a good starting point for the technical questions we'll explore here.

Whether you're meeting AI for the first time or already use it in your studies or work, you may recognize situations from your own experience. Think of a researcher who has an interview transcribed. Getting the words right matters. If the subject of the research is hesitation, the pause between two words matters, too. The researcher needs to know which information survives in the transcript and which details can be recovered only by listening to the recording. Throughout the book we'll consider what a model produces alongside the information it had to work with.

You can follow the opening examples if you can compare two rows in a table, average a few numbers, and see how far an estimate falls from a measurement. No programming experience or advanced mathematics is required. If you enjoy the mathematics, the same examples let you follow the calculation further. You can work out, for instance, why changing two settings by the same amount can have quite different effects on prediction error. We introduce each technical term in the example where it first comes up. The glossary makes it easy to find your way back to a concept as you read on.

How to Use This Book

If this is your first look at how AI learns, we suggest reading the chapters in order. The story starts with the machinist's order book, and each new question gives us a chance to use something from the chapter before. In Chapter 1, we compare a time estimate with the time the machinist measured. In Chapter 2, we ask the same question about a language model. If a sentence has several sensible continuations, what do we compare its prediction with?

When someone asks a question in the workshop, pause before you read the answer. Once you know how long the ten-part and twenty-part orders took, for instance, write down your own estimate for thirty parts. Add a sentence explaining how you arrived at it. When the machinist records how long the new order took, go back to your note. If your estimate differs from the measurement, you can now ask which assumption needs another look.

When you come to a table or a formula, try following one example all the way through. Pick an order from the table and start with the units. How do two minutes *per part* and five minutes *per order* enter the calculation? Then change just the part count and make the estimate again. If mathematical notation is unfamiliar, being able to describe that relationship in your own words is enough for a first reading. You can return to the arithmetic once you know what question it answers.

You can also repeat the small experiments in an AI application. Give it the full story and ask why the machinist locked the door. Then take out the sentence that explains the reason and ask the same question in a fresh chat, so the original passage won't supply a clue through the conversation history. For each answer, ask yourself, "Can I point to a sentence in the passage I supplied that supports this explanation?"

Use the chapter titles in the Contents to find a topic again. If you're wondering why a detail in a photograph changed during an edit, for example, go straight to Chapter 9. The question in italics beneath each chapter title takes you into the relevant workshop scene. For a short definition, turn to the Glossary at the back. To see how the topics fit together, turn to the Learning Roadmap.

Learning Roadmap

The thirteen chapters follow four stages. First we'll look at how a model learns from past examples. Then we'll see how it uses what it has learned to answer a question and ask what information supports that answer. As we turn to images, sound, and video, we can ask those questions again. This time our clues might be the pixels in a photograph or the time between two knocks.

1. Learning from examples and predicting text · Chapters 1–3

Chapter 1 begins with a time estimate for an order. We calculate the difference between the estimate and the measured time, then see how that difference responds when we change each of the model's settings. **Chapter 2** takes the same idea into language. For a model predicting how a sentence will continue, the written text itself supplies the training example. We judge a prediction by how much probability the model assigned to the piece of text the author actually used.

Chapter 3 shows how that probability is calculated. We give different openings to two passages that end in exactly the same words. By tracing how the model combines information from earlier words, we can explain why a small change at the beginning can affect the prediction at the end.

2. Chat and the evidence behind an answer · Chapters 4–6

In **Chapter 4** we move from completing an unfinished sentence to answering a question. We follow the answer by looking at who said what in the conversation and what information came before the question. **Chapter 5** adds a complication. We know the machinist locked the door at lunchtime. Is it still locked that evening? The time of the reading from the smart lock changes what we can tell the apprentice.

Chapter 6 investigates something that happened in the past. We follow the assistant as it retrieves a maintenance entry and puts it in

front of the model. By the end of these three chapters, you can take a claim in an answer and identify the sentence in the conversation, the device reading, or the dated entry you'd need to check it.

3. Reading, generating, and editing images · Chapters 7–9

Chapter 7 begins with a thin strip of light coming in under the door. We compare the brightness of a few pixels to see how an edge can stand out in an image. **Chapter 8** takes us a step further. With no photograph of the door to go on, a written description guides a model as it generates a new image starting from noise.

For **Chapter 9**, we put a photograph of the machinist's own knocker on the workbench. Using it as a reference and cutting the knocker out of it to place on the door give different results. To change a particular area of the door, we mark that area with a mask. By the end of this stage, we can discuss which methods can preserve the tiny notch on the knocker and why.

4. Following sound, motion, and the order of events · Chapters 10–13

Chapter 10 introduces change over time through sound. We pick out two knocks in a recording and look for the features that carry the ringing of the brass. **Chapter 11** applies this distinction to speech. When two people say the same sentence, their recordings can sound different and still produce the same transcript. Reading that transcript aloud calls for choices about stress and delivery.

Chapter 12 compares successive pictures to make sense of motion. When a door stands ajar, the order of the frames tells us which way it's moving. Generating video also means keeping the door's appearance and movement consistent across frames. **Chapter 13** brings sound and picture together in one recording. We can hear the knocker strike and see the door open shortly afterward. When whoever opens it stays out of shot, we know the sequence of events, yet the person's identity remains an open question.

At the end of the roadmap

When you finish the book, three questions can help you examine a new example on your own. What might the model have compared

its prediction with during training? What information does it have now? Which detail in its answer can you point to in the text, image, or recording at hand? You can take the habit of asking questions that began with the machinist’s notebook into your own encounters with AI.

Chapter 1 · Model training, parameters, and backpropagation

What changes inside a model as it learns?

An order for thirty parts comes into the workshop. The customer wants to know how long the job will take. The machinist opens the workshop notebook. The last time they made ten of these parts, the job took 25 minutes. An order for twenty took 45 minutes.

What would you tell the customer?

The extra ten parts took 20 minutes, which works out to 2 minutes per part. But the first order leaves us with a small puzzle. Ten parts at 2 minutes each account for 20 minutes. Where did the other 5 go? Perhaps the machinist has to set up the machine at the start of every order. On that assumption, our answer for thirty parts is **5 + 2 × 30 = 65 minutes**.

Look again at the notebook. Neither entry says how long setup took. We've come up with a relationship that could explain both observations. Let's give the computer the form of that relationship and see how it can use the measurements to find the two numbers in it.

Improving a prediction

Our model fits on one line:

$$\text{predicted time} = a + b \times \text{number of parts}$$

Here a is the time we add once per order, and b is the time we add for each part. We call numbers whose values we can adjust during training **parameters**. To keep the arithmetic easy to follow, let's start with $a = 0$ and $b = 2$. We'll take the error to be the prediction minus the measurement, so a minus sign means we underestimated the time.

Parts	Measured time	Initial prediction	Error
10	25 minutes	20 minutes	−5 minutes

Parts	Measured time	Initial prediction	Error
20	45 minutes	40 minutes	−5 minutes

Both predictions fall five minutes short. You might look at the table and say, “Just add five minutes for setup and you’re done.” We’d like to give the computer a method it can use with hundreds of orders. Changing the parameters might improve some predictions and make others worse. How do we decide whether the new settings are better overall?

One way is to square each error and take the average. Squaring does two things for us. If one estimate is five minutes short and another is five minutes over, the errors won’t cancel out. Larger errors also carry more weight. A 2-minute error contributes 4 to the sum, whereas a 4-minute error contributes 16.

The quantity we try to reduce during training is called the **loss**. Let’s calculate our starting loss:

$$\frac{(-5)^2 + (-5)^2}{2} = \frac{25 + 25}{2} = 25$$

The unit here is minutes squared. We were five minutes off on each order. **25** is the score our chosen measure assigns to those two errors. After changing the parameters, we can do the same calculation and see whether the score went down.

Which parameter should we change, and in which direction?

Let’s try a small experiment. Keep b where it is and raise a from 0 to 0.01. Since setup time is added once per order, both predictions go up by 0.01 minutes. Both errors go from −5 to −4.99. The new loss is 4.99^2, or 24.9001. At our starting point, increasing a a little helps.

Now go back to the original settings and try the same experiment with b alone. Adding 0.01 minutes **per part** raises the prediction for ten parts by 0.10 minutes and the prediction for twenty by 0.20 minutes. The errors are now −4.90 and −4.80. The new loss is $(4.90^2 + 4.80^2)/2 = 23.525$.

We nudged each parameter by the same amount, yet the loss changed by different amounts. There are two things to notice here. One is **direction**. When we increase the parameter, does the loss

rise or fall? The other is **sensitivity**. How much difference does a small change make?

The **derivative** gives the rate at which the loss changes at that point as we vary one parameter and hold the others fixed. Put those derivatives in order, one per parameter, and we have the **gradient**. Ours has two numbers. If you'd like to see where they come from, let's work through one training step.

> **The arithmetic of a training step**
>
> Start with a single order. When the error is -5, increasing the prediction by a small amount d gives a new error of $-5 + d$. Let's expand its square:
>
> $$(-5 + d)^2 = 25 - 10d + d^2$$
>
> For a small step, the last term is tiny. With $d = 0.01$, for instance, the middle term is -0.1 and the last term is just 0.0001. Near our starting point, we can estimate the change in squared error as **-10 × the increase in the prediction**. That -10 is the slope.
>
> Both orders start with the same error. A change in a carries over one for one to both predictions, so the slope of the loss with respect to a is $(-10 - 10)/2 = \mathbf{-10}$.
>
> The first prediction multiplies b by 10 and the second multiplies it by 20. The slope of the loss with respect to b is therefore $[(-10) \times 10 + (-10) \times 20]/2 = \mathbf{-150}$.
>
> Our gradient is **(−10, −150)**. Those are the slopes where we're standing. To decide how big a step to take, we choose a number called the **learning rate**. Let's use 0.001. From each parameter we subtract 0.001 times its own slope:
>
> $$a_{\text{new}} = 0 - 0.001 \times (-10) = 0.01$$
>
> $$b_{\text{new}} = 2 - 0.001 \times (-150) = 2.15$$
>
> Both slopes were negative. Subtracting a negative number increased each parameter. Our new predictions are **21.51 minutes** for ten parts and **43.01 minutes** for twenty. The loss is now about **8.07**.

What changed inside the computer after one training step? We still have the same formula and the same orders. The two numbers

we store in the formula have changed. That's why the next calculation gives us different predictions.

There's a surprise here. We worked out from the notebook that each part took 2 minutes. The algorithm started at exactly that value and moved to 2.15. How did the loss go down?

We started with zero setup time, so both estimates were low. At that point, increasing the time per part made up some of the shortfall. On the next step the algorithm will calculate the slopes again at its new position. The direction one parameter should move depends in part on the current value of the other. That's one reason we use a learning rate to limit the size of the step. A slope that points downhill here tells us little about how far we can keep going in that direction.

When the order for thirty parts is finished

The machinist adds a line to the notebook. The thirty-part order took **95 minutes**. Our 65-minute estimate, which followed the first two entries perfectly, fell half an hour short. Where did we go wrong?

A visit to the workshop gives us the answer. The first twenty parts come off the machine ready to ship. Every part beyond the twentieth in the same order needs another three minutes of finishing by hand. Ten of the thirty parts needed that extra work, accounting for the extra 30 minutes.

Could we have deduced that rule from the first two orders? Neither needed any hand finishing. What's more, our formula assumed that *each additional part adds the same amount of time*. Whatever values we choose for a and b, a single straight line can't pass through all three points. The new order revealed a limitation we couldn't see in the training records. A model's ability to apply what it has learned to new examples is called **generalization**.

To fix the calculation, let's answer one question first. **How many parts will need finishing by hand?**

Order size	Parts minus 20	Parts to finish by hand	Extra time
18 parts	−2	0	0 minutes
20 parts	0	0	0 minutes
23 parts	3	3	9 minutes

One small operation takes us from the second column to the third. We can't finish minus two parts, so we use zero whenever the result

is negative. If it's positive, we use it as it stands. The rule is simply **take the larger of zero and the number you have**.

Who chooses the activation function?

The operation we just used also has a name in neural networks: **ReLU**. Its formula is short:

$$\mathrm{ReLU}(z) = \max(0, z)$$

Here z is the number passed to the function. In our example, z = number of parts − 20. For an order of 18, ReLU receives −2 and returns 0. For an order of 23, it receives 3 and returns 3. The next operation multiplies the result by three minutes.

A unit in a neural network can work much the same way. First it multiplies its inputs by weights, adds up the products, and adds a constant. Then it passes the resulting number through an **activation function**. ReLU is one such function. The word "activation" might make you picture an on-off switch. Look at the table, though. The output also tells us *how much*. Three extra parts and ten extra parts pass different values on to the next calculation.

Did the model discover ReLU during training? **Typically, the people who design the model choose the type of activation function. The rule for standard ReLU stays fixed throughout training.** What can be learned are the numbers around it. Our workshop calculation lets us see where three separate choices come in:

- **ReLU** takes the larger of its input and zero.
- **20** is the threshold. We subtract it from the part count before applying ReLU.
- **3 minutes** is the extra time each part adds once we're past the threshold. We multiply the output of ReLU by that amount.

The machinist told us the threshold and the extra time. If all we had were order records, we could put two learnable parameters in place of 20 and 3. With enough examples, particularly on both sides of the threshold, we could adjust those values during training. Changing the threshold would move the bend in the graph. Changing the extra time per part would change the slope beyond the bend. Some other activation functions also have learnable parameters of their own.

To see what this small operation buys us, consider two successive increases in order size. Going from 19 parts to 20 adds 2 minutes. Going from 20 to 21 adds 2 minutes of machine time and 3 minutes

of hand finishing, for a total of 5. **Adding one more part can now have different effects depending on where we start.** We can express the nonlinear relationship we found in the workshop.

We know the outcome. How can the first layer learn?

Suppose the only time we measure in the workshop is the total. We have no separate timings for setup, production, or hand finishing. We can tell when the model gets the final answer wrong. How can we work out the contribution of a weight at the very beginning?

For a moment, reduce the calculation to two multiplications. Our input is 3. The first layer multiplies it by its weight of 2 to produce **6**. The second layer multiplies that 6 by its own weight of 4 to produce **24**. The answer we wanted was 20. We've overshot by four.

Imagine increasing the first weight from 2 to 2.001. Before doing the arithmetic, can you tell which way the final prediction will move? The first layer's output will rise a little. The second layer will multiply that increase by four. A prediction that was already high will go a little higher.

Follow the numbers. Increasing the weight by 0.001 takes the intermediate value from 6 to 6.003. The final prediction goes from 24 to 24.012. We've followed the effect of a small change in the first weight through two operations.

Now read the same calculation from the other end. **The final prediction is too high. Which intermediate value affected it?** The input to the second layer. **By how much?** The second layer multiplied that input by four. **Which weight affected that intermediate value?** The weight in the first layer. Its effect in turn depended on the input, 3. **Backpropagation** starts at the loss and works backward, combining these local effects to calculate the gradient of the loss with respect to each weight. The training algorithm then uses those gradients to update the weights.

At no point did we tell the first layer, "Here's the intermediate value you should have produced." Knowing the final target and the operations along the way was enough to calculate how a small change in that weight would affect the loss.

In this example we could have rolled the two layers into one instruction: multiply by 8. With nonlinear operations such as our workshop ReLU between layers, the network can express a wider range of relationships. A feature that one layer picks out becomes an input the

next can use. To see why that matters, let’s try a different job in the workshop.

From the workshop notebook to a photograph of a bicycle

This time the machinist wants to answer a different question: **“Will this part pass inspection?”** Suppose that for every part inspected in the past, the machinist measured the longest scratch on its surface and the amount of burr left on its edge. We could put those measurements alongside the inspection results and try to predict the outcome. Our answer now has two possibilities, pass or fail. The recorded inspection result plays the role the clock played when we were predicting production time.

Now suppose the machinist sets the measuring tools aside and records only **a photograph of the part and the inspection result**. We’re asking the same question, but nobody gives the model a scratch length anymore. Is there something in the photograph that carries that information? Perhaps a thin, dark line across an otherwise smooth surface. Starting from pixels, the model also has to learn which features can help it reach a decision.

From here it’s a short step to recognizing a bicycle in a photograph. Imagine a collection of photos, some with bicycles and some without. Each has just one label, “present” or “absent.” We haven’t marked the wheels or described the shape of the handlebars.

The early layers of an image network can develop filters that respond to light-to-dark transitions in particular directions. Later layers can use the ways those transitions come together in the image. Suppose the model gives a low probability of “bicycle present” to a photo with a bicycle in it. We know the label, so we can calculate a loss. Through backpropagation, that loss can influence adjustments all the way back to the filters that first processed the image.

That’s how we can teach a model to recognize bicycles without marking the position of every wheel rim. In the workshop, we gave an intermediate value a precise name: “number of parts to finish by hand.” With an image network, we can’t always point to a learned feature and name it “wheel rim” with the same confidence. We can still understand how it came to be learned. The loss measured at the end of processing a photo helps adjust weights at the very beginning.

In the workshop we had a clock. In the bicycle photo we had a label. Now imagine a language model reading this line in its training text: “Early that morning, the machinist walked up to the workshop door and…” Several continuations would make sense. **When a language model learns, what does it compare its prediction with?**

Chapter 2 · Next-token prediction and cross-entropy in language models

What does a language model predict, and how do we measure its error?

"Early that morning, the machinist walked up to the workshop door and…"

Which word would you write next? If you pictured the machinist arriving for work, you might have chosen "opened." Let's see what the author actually wrote:

> Early that morning, the machinist walked up to the workshop door and locked it. The machine the job called for that day was at the customer's factory.

The second sentence explains why the machinist locked the door first thing in the morning. When we stopped at "and," we didn't have that information. "Opened" was a perfectly reasonable guess. If we used this passage for training, how would we judge a language model's prediction?

In the previous chapter, the machinist checked the clock and wrote 25 minutes in the notebook. We compared our prediction with that measurement. **When a language model is trained to predict the next piece of text, the text itself supplies the target.** Here the author wrote "locked." Given only what the model could read up to that point, how much probability did it assign to that continuation?

How many questions can we get from one sentence?

Suppose we give "lock," "locks," and "locking" a whole entry each in the model's vocabulary. What happens tomorrow when we come across "unlocking"? If we can represent text in smaller pieces, we can reuse familiar pieces in new words.

Each piece is called a *token*. A *tokenizer* uses a vocabulary and a set of rules to split text into pieces and turn each piece into its vocabulary ID. Let's make up a tiny vocabulary in which "lock" has ID 17

and “ed” has ID 42. If our rules split “locked” into those two pieces, the input to the model contains 17 followed by 42. The IDs act as addresses in the vocabulary. Another tokenizer might keep “locked” in one piece or split it elsewhere. Token boundaries needn’t follow the boundary between a stem and an ending.

The split matters for prediction, too. With our tiny vocabulary, if the model produces 17 and then 42, it has taken two prediction steps to write what the reader sees as a single word, “locked.”

To keep the next calculation easy to follow on paper, let’s switch to a simpler rule. For the rest of this example, count each whole word as one token and each punctuation mark as a token of its own. Write the sentence on a sheet of paper and cover the right-hand end with your hand. Each time you uncover one more token, the same sentence gives you a new question. Here are the last few:

Context the model can see	Next token
Early that morning, the machinist walked up to the	workshop
Early that morning, the machinist walked up to the workshop	door
Early that morning, the machinist walked up to the workshop door	and
Early that morning, the machinist walked up to the workshop door and	locked
Early that morning, the machinist walked up to the workshop door and locked	it
Early that morning, the machinist walked up to the workshop door and locked it	.

The left side of each row is the **context** and the right side is the **target**. “Locked” is the target in one row and part of the context in the next. The training software can get both from the same text. Nobody has to sit down and write a separate question and answer for each example. Learning from targets obtained from the data itself in this way is called **self-supervised learning**.

If the training software has the whole sentence, doesn’t the model get to see the answer? Look at the boundary between the columns.

Even when many positions are processed together, the calculation at each position can use only the tokens up to that point. The position predicting what follows “and” can’t see the author’s choice of “locked.” The training software can use that choice to score the prediction.

If both models guess “opened,” did they do equally well?

At this point the model produces a **probability distribution** over the tokens in its vocabulary. Imagine two models reading the same unfinished sentence. We’ll use made-up percentages so we can compare them.

Next token	Model A	Model B
opened	60%	50%
locked	10%	40%
All other tokens combined	30%	10%
Total	**100%**	**100%**

Ask each model for its single most likely word and both will give you “opened.” If all we recorded was whether that word matched the author’s, we’d score them equally. Yet Model B allowed a substantial chance that the machinist locked the door. Model A considered that much less likely.

Now lift your hand from the page. The author wrote “locked,” so the row that counts for this step is **locked**. Model A assigned it 10% probability and Model B assigned it 40%. **Model B should have the lower loss on this example.** We didn’t have to ask either model to choose a word. We knew the target, so we could look up the probability each model assigned to it.

Try changing “locked” to “opened” in the author’s sentence. Leave the models’ percentages alone. Which model gets the lower loss now? Model A, because we’re reading a different row. One sentence tells us which prediction scored better on that example. To compare the models more generally, we’d repeat the calculation at many positions in texts held out of training.

How do we turn a percentage into a loss?

In Chapter 1 we squared the error in a time estimate. Here the number we have is a probability assigned to the target token. We want a small loss when that probability is high and a large loss when it's very low. Let's see what a common way of doing that gives us for our two models.

> **Cross-entropy with two numbers**
>
> Call the probability assigned to the target token p. The **cross-entropy loss** at this position is $-\ln(p)$. A logarithm tells us what power we have to raise a base to in order to get a given number. For instance, since $10^{-2} = 0.01$, the base-10 logarithm of 0.01 is -2. The natural logarithm, ln, asks the same question using base e (about 2.718). For a probability strictly between 0 and 1, the exponent is negative. The minus sign in the formula gives us a positive loss. Enter the probability in your calculator as a decimal.
>
> - Model A assigned $p = 0.10$. Its loss is about **2.30**.
> - Model B assigned $p = 0.40$. Its loss is about **0.92**.
>
> One end of the scale is easy to check. Give the target a probability of 1 and the loss is zero, since $\ln(1) = 0$. At the other end, the loss grows as the target probability approaches zero. Giving the word the author actually used almost no chance carries a steep penalty.

Logarithms help in another way. Suppose the model gives "locked" a probability of 40%, then, once it sees "locked," gives "it" a probability of 50%. The probability it assigns to those two tokens occurring in that order is $0.40 \times 0.50 = 0.20$. A logarithm turns multiplication into addition, so we can work out the losses for the two tokens separately and add them: $0.92 + 0.69 \approx 1.61$. Calculating $-\ln(0.20)$ directly gives the same rounded result.

A loss of 0.92 for Model B doesn't mean it was 92% wrong. The loss is a score that lets us compare predictions on a common scale. During training, we calculate these scores for many target tokens and usually take their average. Then the mechanism we saw in Chapter 1 comes into play. Backpropagation computes the gradients of the loss with respect to the weights, and the training algorithm updates the weights.

Will the model learn to put "locked" in every story about arriving at the workshop in the morning? Another author might write exactly the same opening and follow it with "opened." On that example, the

probability assigned to “opened” determines the loss. The same weights are used across many training examples. “Early that morning” might be one clue. An earlier line of dialogue, “I’m working at the customer’s place today,” might supply a different one. Assigning probability to several continuations lets the model reflect something familiar about language: even in the same context, several things can make sense next.

What if we ask the model to write the rest?

Close the training text. Suppose we give the trained model just “Early that morning, the machinist walked up to the workshop door and” and ask it to continue. This time there’s no author’s next word waiting under our hand. The model still produces probabilities for the next token. Now we use that distribution to **choose what to write**.

Go back to Model A. If we choose the most likely token at every step, our first choice is “opened.” If we sample according to the probabilities, we might get “locked” instead. Suppose we get “opened.” Here’s the text the model now has in front of it:

> Early that morning, the machinist walked up to the workshop door and opened

What could follow? “It” would let the machinist open the door. “The” might take us somewhere else entirely: perhaps the machinist opened the toolbox first. Looking at the runner-up in the old table won’t tell us. That table was for a context ending in “and.” **The chosen token joins the context, and the model calculates fresh probabilities for the new context.** That’s how the text grows.

Think of the same sentence at three moments in a model’s life. **Before training from scratch begins**, a tokenizer can already split the sentence into pieces, and the model can compute a probability distribution. Its initial weights have yet to learn from English text what tends to follow an opening like this one. **During training**, we know how the written sentence continues. We calculate a loss from the probability assigned to “locked” and use gradients from many examples to update the weights. **After training**, we use the learned weights to complete the sentence. When the model writes “opened,” its next prediction changes because “opened” is now part of the context. That one-word step requires no weight update.

We now know how to score the percentages in the table and how to use them to generate text. So far, though, we’ve supplied the percentages ourselves. When a real model reads our sentence, **what**

calculations connect “early that morning” with “the workshop door” and turn those clues into a probability for “opened”?

Chapter 3 · Token vectors, attention, and softmax

Why might we expect different continuations after exactly the same words?

> **A.** The apprentices were waiting outside. The machinist walked up to the workshop door and…
>
> **B.** Everyone had left, and the lights were out. The machinist walked up to the workshop door and…

If the machinist opened the door in the first story and locked it in the second, most of us would read on without a second thought. Now cover the first sentence of each passage with your hand. From the words you can still see, could you tell which story should continue with “locked”?

In Chapter 2 we supplied percentages for “opened” and “locked.” Now let’s look at what happens in between. How does the fact that the apprentices are waiting outside reach a prediction made several words later?

We’ll explore that question through a neural network architecture called the **Transformer**. An architecture specifies how the parts that do the computing are arranged and connected. In the language model we’ll follow, each token occupies a position in the text. At each position, a group of numbers represents the information available there. A layer takes those numbers, combines information from different positions, processes the result, and passes it to the next layer. After several layers, the numbers at the final “and” carry information about what came before it. The model uses that last position to score the candidates for the next token.

Let’s start by seeing what numbers a single position gets to work with. For our prediction example, we’ll treat “and,” “opened,” and “locked” as one token each.

What’s in row 17?

Imagine that the machinist records three measurements for every metal part in the notebook: *length*, *width*, *and thickness*, in that order. The entry for one part might read (40, 10, 2), in millimeters.

We're using three numbers together, and their order matters. Anyone looking for the width knows to read the second number. In this chapter we'll use **vector** to mean an ordered list of numbers like this one. Each number is called a *component*.

Remember the little vocabulary in Chapter 2 that split "locked" into "lock" and "ed"? Let's use it briefly to see how a lookup works. Suppose the tokenizer encounters "locked and bolted." For the "locked and" part, it returns the IDs 17, 42, and 58, in that order. The model uses each ID as a row address. Here's a table with three components per row. We've chosen the numbers so we can follow the calculation.

Token	Row number	First component	Second component	Third component
lock	17	0.6	−0.2	0.4
ed	42	−0.1	0.7	0.2
and	58	0.3	0.5	−0.4

Start with 17. Run your finger along that row and you get the vector (0.6, −0.2, 0.4). Those three numbers enter the calculation at the position occupied by "lock." Row 42 gives us a different set of three, so "ed" has a starting representation of its own. Finally, row 58 gives us the vector for "and." **Three tokens went in, and we got one vector for each position.** The starting vector we look up for a token is its *embedding*.

> **How long is a row in a real model?** We ran a finger along row 17 and read off three numbers. In the small version of GPT-2, we'd read off 768. For a passage of 20 tokens, that gives the model 20 starting vectors, each with 768 components. GPT-2 model configuration

In the machinist's notebook we knew what the second column measured. What does −0.2 measure in the row for "lock"? Nobody assigned the second column a meaning such as "security" or "metalwork." During training, the loss on next-token predictions provides gradients for adjusting the numbers in the table, too. Information about how a token is used can spread across many components. What we read from a row is a set of numbers the later layers have learned to use together.

Go back to the two passages and find the final "and" in each. Both lookups take us to row 58. That gives us a starting point determined by the identity of the token. How can we also take into account *where* it occurs?

Word order carries information

There's a note in the workshop window: "First measure, then cut." You could use exactly the same words to write "First cut, then measure." Follow the second instruction and, by the time you discover you've cut the part too short, there's little you can do about it. Yet an inventory of the words would look identical.

The model also takes account of each token's position in the sequence. In particular, "measure" stands in a different relation to "first" and "then" in the two notes. Transformer models have several ways of incorporating position. The idea we'll need here is that when the model calculates a relationship between two positions, the result can depend both on which tokens occupy them and on where they are relative to one another.

How does information arrive from the previous sentence?

We're at the final "and." As we update the numbers at this position, we'd like to draw on earlier positions. One simple approach would be to take their average. But must every position get an equal say in every sentence? **Attention** gives us a way to calculate how much information to take from each position according to the context at hand.

Try a small experiment on paper. Pick two positions that could supply information to "and." To keep things manageable, suppose "apprentices" and "waiting" are each a single token, and let only those two positions contribute to our experiment.

First, prepare the numbers each position can contribute. The *dimension* of a vector is the number of components it contains. Our little model carries three numbers per position through its main calculation. Inside attention, let's use two numbers for an intermediate calculation. Both widths are choices made when designing the model. A simple rule will show us how three numbers can give us two.

Call the three numbers entering this layer x, y, and z. For the first intermediate number, calculate $x + z$. For the second, calculate $y + z$. Two calculations give us two results, and the third input takes part in both. A real model learns the coefficients in sums like these during training. We've chosen the coefficients and the inputs below so we can do the arithmetic by hand.

Position	Three numbers entering the layer	Two numbers available from this position
apprentices	(1, 3, 1)	(1 + 1, 3 + 1) = **(2, 4)**
waiting	(6, 2, 2)	(6 + 2, 2 + 2) = **(8, 4)**

We still have two distinct positions. We've expressed what each can contribute as a vector with two components. For the moment, keep your eye on **2 and 8**, the first components of the vectors in the last column.

Next, decide how much to take from each position. With equal shares, the first component would be (2 + 8) / 2 = 5. Let's give "apprentices" a 75% share and "waiting" a 25% share. Then swap just the shares and do the same calculation again.

Share from "apprentices"	Share from "waiting"	First component passed on
75%	25%	0.75 × 2 + 0.25 × 8 = **3.5**
25%	75%	0.25 × 2 + 0.75 × 8 = **6.5**

Now put the vector back together. We do the same calculation for the second component. It happens to be 4 at both source positions, so either choice of shares gives us 4. In the first trial, then, the final "and" receives (3.5, 4). In the second, it receives (6.5, 4). This is information the last position can use to update its own representation.

We chose the shares in that experiment so we could see their effect. When the model reads an actual sentence, how does it work out what share each position gets?

Who decides on 75%?

At each position, the model applies three learned transformations to its representation to produce three vectors: a *query*, a *key*, and a *value*. Follow their roles from the final "and." The **query** at that position is compared with the **key** at every position it can see. Each comparison gives a score for how well the two match. An operation called *softmax*, which we'll work through shortly, turns the scores

into shares. In our two-position experiment, suppose the scores gave us 3 and 1 after exponentiation. The shares would be 3/4 and 1/4. That's one way to arrive at 75% and 25%. The vectors we then multiply by those shares, (2, 4) and (8, 4), are the **values**.

What the model learns during training are the transformations that produce the queries, keys, and values. When we give the trained model a new sentence, those transformations stay the same, but their inputs change. The model therefore recalculates the attention weights and the values it passes along. In the next-token model we're following, each position can attend to itself and to positions on its left. That's how information from "apprentices" can reach the final "and."

Let's check the dimensions. The main representation at "and" has three numbers. The contribution from attention in our first trial has two: (3.5, 4). Before we can add the contribution component by component, we need to express it as three numbers. That's the job of the **output projection** in attention. For example, call the two inputs u and v, and let the three outputs be u, v, and $v - u$. Our vector (3.5, 4) becomes (3.5, 4, 0.5). Now we can add each component to its counterpart at "and." Once again we've chosen coefficients that make the arithmetic easy. The projection puts the information gathered by attention into a form the main representation can use. A feed-forward network in the layer further transforms the representation. The next layer works with those processed numbers, so it can use a clue from the previous sentence together with another clue. **Both passages started with the same row for "and." By the time we approach the output, we have a vector specific to each passage.**

From a vector to a probability for "opened"

We have the vector at the last position. Which token in the vocabulary is likely to follow? For each candidate, the output layer multiplies the components of the final vector by learned coefficients and sums the products to get a score. This raw score is called a *logit*. When the final vector changes, the same coefficients can give different scores.

Let's keep the main representation three components wide. To examine the output calculation on its own, suppose the final vector after all the layers is (1, 0, 1) for passage A and (0, 1, 1) for passage B. We've chosen these example values for the last layer.

Imagine a small output layer that scores just two candidate tokens. To score "opened," let it add the first and third components. To score

“locked,” let it add the second and third. Apply the same rules to both passages.

Passage	Vector at the last position	Score for “opened”	Score for “locked”
A	(1, 0, 1)	1 + 1 = **2**	0 + 1 = **1**
B	(0, 1, 1)	0 + 1 = **1**	1 + 1 = **2**

Three numbers give us two scores here because we’re scoring two candidates. If we evaluated a hundred candidates with the same three-component representation, we’d calculate a hundred scores, using each candidate’s own set of coefficients.

What does softmax do with those scores?

For passage A our scores are 2 and 1. At first glance, dividing each score by their sum might seem to do the job. But the model can produce negative scores as well. If the scores were 1 and −1, their sum would be zero. We need a way to turn any score into a positive number while keeping the higher score ahead.

Softmax first raises e to the power of each score. We met e, approximately 2.718, in Chapter 2 when we used the natural logarithm. Even if the exponent is negative, the result is positive. For example, e^{-1} is about 0.37. Next we add up the positive numbers and divide each one by that total.

Candidate	Raw score	After exponentiation	Share of the total
opened	2	$e^2 \approx 7.39$	7.39 / 10.11 ≈ **73%**
locked	1	$e^1 \approx 2.72$	2.72 / 10.11 ≈ **27%**

You can trace the 10.11 with your finger. Add the two numbers in the third column, then use that same total for both divisions in the last column. The higher score gets the larger share, and the shares add up to 100%. In the full model the sum includes every candidate in the vocabulary.

Try changing the numbers. If both scores were 1, each candidate would get 50%. Now raise only the score for “opened” to 2. Its exponential grows by a factor of e. The raw score for “locked” is still 1, yet its share drops from 50% to 27%. **A candidate’s probability depends on the scores of the other candidates, too.** In passage B the scores trade places, so “locked” gets the 73% share.

Now put two percentages from this chapter side by side. The **75%** in our attention calculation told us how much weight to give the value from one input position. The **73%** at the output is the probability assigned to “opened” as the next token. Attention distributed shares among **positions in the text**. The output distributed them among **possible continuations in the vocabulary**.

Go back to the two passages and uncover their first sentences. We read the same final words and expect different continuations. We can now see how that can happen in the model. Changing the first sentence still leaves the final “and” starting from the same vocabulary row, but it changes the information that can reach that position. When different numbers reach the output layer, the scores for “opened” and “locked” can change. Behind the percentages we supplied by hand in Chapter 2, we now have a calculation we can follow.

Chapter 4 · Roles, context, and answer generation in chat models

How does a model that predicts one token at a time answer a question?

> "I'm working at the customer's factory this afternoon," said the machinist. "I don't want to leave the workshop unlocked while I'm away." The machinist hung up an apron, walked up to the workshop door and locked it, then set off.

In Chapter 3 we stopped the story at "walked up to the workshop door and" and saw how the model could give "locked" a high score as the next word. This time, let's give it the whole story and write a question underneath.

> Why did the machinist lock the door?

"Locked" is already on the page. What we want now is an answer that brings two details together. The machinist will spend the afternoon somewhere else and wants the workshop locked in the meantime. How can a system that picks just one token at a time put that answer together?

An answer is one way to continue

Imagine writing "Why did the machinist lock the door?" on a slip of paper and handing it to someone. They'd probably try to answer the question. Now imagine they saw the same line in a Spanish workbook under the instruction "Translate the following questions into Spanish." They might write "¿Por qué cerró el maquinista la puerta con llave?" The question is the same. A few words at the top of the page changed the job they were being asked to do.

In the texts a language model encounters during pretraining, a question can have many kinds of continuation. In an interview, an answer might follow. On an exam paper, it might be another question. In a novel, the next words might be "asked the apprentice." The model can learn from all of these. Answering a question is one of the things it can do by predicting the next token. In a chat application we have

a more specific expectation. When we ask a question, we want the model to answer *as the assistant*.

How does the model learn what to do on the assistant's turn?

One common approach is to show a pretrained model user requests paired with example responses. This is called **instruction tuning**. Here's what one training example might look like.

> **User:** "The apprentice put on a glove to protect their hand from the hot metal." Why did the apprentice put on a glove?
> **Assistant:** To protect their hand from the hot metal.

Think back to our question in Chapter 2. What do we compare the prediction with? Suppose the first token of the example response is "To." The model sees the user's message and a marker indicating that it's the assistant's turn. It calculates probabilities for the next token. We know which token to score because we have the example response. At the next position, the model is also given the actual opening, "To," and the token that follows it becomes the target. Each piece of the example response supplies a training target in turn.

The loss calculation we used in Chapter 2 works here as well. Assigning a low probability to a token in the example response incurs a larger loss at that position. A training step adjusts the parameters in a direction intended to reduce the loss. Across many examples, one user asks for a reason, another wants a translation, and a third wants a shorter version of a passage. From this variety the model can learn to produce the kind of response a new request calls for. Our particular story about the machinist need never have appeared in the training examples.

The question on the screen is part of the model's input

Back at the workshop, you type the story and the question into a chat box and press Send. The application also tells the model who said what. Here's a readable version of the input it might prepare. The actual markers for roles and message boundaries vary from model to model.

Role	Message
System	Answer the reader's question in one sentence, using the story provided.
User	"I'm working at the customer's factory this afternoon," said the machinist. "I don't want to leave the workshop unlocked while I'm away." The machinist hung up an apron, walked up to the workshop door and locked it, then set off. Why did the machinist lock the door?
Assistant	*Generation starts here.*

The system instruction sets two expectations. The story should supply the information, and the answer should fit in one sentence. The user message contains both the passage to work with and the question about it. The final role marker indicates whose words the upcoming text will be. The information available to the model at that point is its **context**.

Try a small experiment. Keep every word of the story and change the question to "Where will the machinist be working this afternoon?" Now "At the customer's factory" is a complete answer. In the first question, the same detail was part of a reason. The attention mechanism from Chapter 3 is at work here, too. Changing "why" to "where" changes the input. That can change what information is carried forward from earlier positions and how the candidate tokens score as the answer begins.

From one token to a sentence

Suppose the model gives this answer to the original question.

> Since the machinist was heading to the customer's factory, they locked up to avoid leaving the workshop unsecured.

We read that sentence as a whole. Let's pause at a few points and see what the model has to work with. For readability, the table shows stretches of text that may span several tokens.

Where we pause	What's available when the next token is predicted?
The answer hasn't started	The instruction, story, question, and marker for the assistant's turn. "Since" and "The" are both possible openings.
The answer so far is "Since the machinist was heading to the customer's"	Those words have joined the original context. The phrase "the customer's factory" in the story gives the model a clue to what might follow "customer's."
The answer so far is "Since the machinist was heading to the customer's factory,"	The opening clause gives a reason and leaves a thought to complete. The machinist's words about leaving the workshop unlocked are still in the context.

At each step the model recalculates the probabilities and the generation method selects a token. That token becomes part of the input for the next step. If the first word had been "The," the answer might instead have read "The machinist locked up to keep the workshop secure while working at the customer's factory." The same reason can be expressed in several ways. When a special end-of-response token is selected, the application can stop generation.

Here we can see a small but important difference between training and use. During instruction tuning, we could take the actual continuation from the example response. When you ask a question, the application has no answer key against which to check each step. The model continues from the words it has produced so far. Change the destination in the story and the answer can change, too, with the same model parameters. **What changes is the information available to the model as it works out that answer.**

What happens if we take a line out of the story?

Let's take out the machinist's own words and leave just this:

> The machinist hung up an apron, locked the workshop door, and set off.
> Why did the machinist lock the door?

What would you say this time? “The machinist was leaving the workshop” is fair enough. “To keep the tools from being stolen” is a plausible guess. But who would think to say “The machinist was off to the customer’s factory”? We remember that detail because we’ve read the longer story. If you try this experiment in a chat application, give it the short passage **in a fresh chat, without the earlier story**. In the same conversation, the application may include the first story in the context as well.

An answer that sticks to the short passage might read, “The machinist locked the door when leaving. The story doesn’t give a specific reason.” A model might still answer, “To protect the tools from thieves.” That explanation fits everyday experience. But go back to the passage and ask, “Which words told us that?” There’s nothing we can point to. A high probability for the next token tells us the model finds it a likely continuation in context. Whether events actually happened that way is a separate question.

Now ask one more question. **Is the workshop door locked right now?** Even with the full story back on the page, all we know is what the machinist did on the way out. Perhaps the apprentice came by five minutes later and unlocked the door. To find out how things stand now, we’d have to ask someone who can check the door or read the latest status from a smart lock. How can a chat application reach an outside source like that, and how does the information it gets back find its way into the model’s answer? In the next chapter we’ll open the workshop door to the outside world.

Chapter 5 · Tool calls and data freshness

Is the workshop door locked right now?

It's 3:20 p.m. The machinist is working at the customer's factory. We know the machinist locked the workshop door when leaving at lunchtime. Now a message comes in from the apprentice's phone.

> Is the workshop door locked right now?

Put yourself in the shoes of the assistant on the phone. You have the story from Chapter 4 in front of you. You can read it as many times as you like before answering. But what if, while you're reading, the machinist's business partner unlocks the door with a spare key? On the page in front of you, an afternoon when the partner stops by looks exactly like an afternoon when nobody comes. It takes an observation from outside the story to tell the two apart.

Let's try the simplest route first. The apprentice walks over to the door, checks the lock, and types a message into the chat. "I checked at 3:20 p.m. The door is locked." The model can now answer, "According to the apprentice's 3:20 check, the door is locked." We've seen this happen before. A new message joined the conversation and gave the model something to base its answer on. How could we get the same information without sending the apprentice all the way to the door?

How does the model know what it can ask for?

Suppose the workshop has a smart lock with a sensor that can read the position of the bolt. The developer of the chat application has written a small function that asks for a reading. Give the function the name of a door and it returns the state reported by the device on that door, along with the time of the reading. A function that an application makes available to the model in this way is called a **tool**.

Imagine someone telling you, "You can call this number to find out whether the main workshop door is locked." You now know whom to call. To learn anything about the door, you still need an answer from the other end of the line. The tool description supplied to the model serves much the same purpose.

Tool name	What does it do?	What input does it need?
`read_lock_status`	Returns the lock status of the specified door and the time of the reading.	Door name

The application gives the model this description along with the apprentice's question. In our example the workshop has just one entrance, and the application lists it as "main workshop door." The model can see in its input both the operation it needs and the name of the door to ask about.

What does "Let's ask the lock" look like as model output?

In Chapter 4 we saw the model start to write an answer when it was the assistant's turn. A model that can use tools can also use that turn to produce a request for the application. We call this a **tool call**. Here's the request in a form that makes its two parts easy to see.

> **Requested operation:** `read_lock_status`
> **Door:** main workshop door

The model draws on the context to produce the request. "Right now" suggests we need a fresh observation. "Door" points to the lock tool as a possible source. The tool description tells the model how to put the request together. Pause here for a moment. If you asked "Well, is it locked?" at this point, all we'd have to go on would still be what the machinist did at lunchtime. Can you find anything in those two lines that tells you the state of the lock?

The **application** receives the request, runs the named function, and waits for the device to respond. In our example, the device reads the position of the bolt. Here's what the function returns.

> **Door:** main workshop door
> **Lock status:** locked
> **Time of reading:** today at 3:20 p.m.

This is the first time "locked at 3:20 p.m." appears in the exchange. The application adds the result to the conversation as a *tool message* and calls the model again. Now the model can see the apprentice's

question, its own request, and the result together. The apprentice might see this answer on the phone:

> Yes. The smart lock on the main workshop door reported that it was locked at 3:20 p.m.

The apprentice sees a question and an answer. In between, the model ran twice. On the first pass it wrote the request specifying what to check. On the second it wrote the answer using a context that now included the reading. That's how a fact the model didn't have a few seconds ago made its way into the sentence. It took no further training.

What if we change just the result?

Suppose the application's clock still reads 3:20 p.m. Keep the apprentice's question, the story, and the tool call exactly as they were. Change only the result the tool returns. Cover the right-hand column with your hand. What would you tell the apprentice in each case?

Result returned by the tool	What we can tell the apprentice
Locked. Reading taken today at 3:20 p.m.	The latest reading shows the main workshop door was locked at 3:20 p.m.
Locked. Last reading taken today at 12:05 p.m.	The door was locked at 12:05 p.m. We don't know whether that's changed since then.
Device unreachable. No reading on record.	I couldn't reach the lock, so I can't confirm whether the door is locked right now.

In the second row, the answer arrives on the phone **at 3:20 p.m.** It's easy to assume the lock has just been checked. But the last reading the device reported was **at 12:05 p.m.** Since the apprentice asked about the door *right now*, those three hours and fifteen minutes change what we can say.

In the third row, the connection could have failed with the bolt in either position. Suppose the model answered, "The machinist locked it on the way out, so it's locked." It would be using the story to make a guess where a reading was missing. In Chapter 4 we asked, "Which words in the story support this answer?" We can now take that question a little further. **Which device measured what, and when?**

Suppose the apprentice asks one more question that evening. “Is the door still locked?” The afternoon’s answer is right there on the chat screen. The application may even have included it in the model’s context again. By evening, the 3:20 reading is several hours old. What should the model ask for once more before it answers?

Chapter 6 · Searching documents and answering from sources (RAG)

How does the model find an answer in the maintenance log?

It's early November. At the entrance to the workshop, the apprentice finds the brass knocker from the old door. The machinist had taken it off and set it aside. The apprentice asks the assistant on the phone:

> Why did we replace the main door last month?

The smart lock from Chapter 5 can tell us the position of the bolt right now. To find out why the machinist had a new door fitted in October, we'd do better to open the maintenance log. Let's read it ourselves first. There's one detail worth knowing. Materials are delivered through the main workshop door, so the maintenance crew also calls it the "receiving entrance."

Date	Maintenance log entry
October 3	Lubricated the lock on the main door. Turned the key several times to check it.
October 14	The leaf at the receiving entrance was bent. It wouldn't close properly, so we took it off and fitted a new one.
October 22	Water was getting in under the storeroom door. Replaced the worn seal.

Which entry would you show the apprentice? The first uses the exact words "main door." The third mentions both a door and a replacement. But to answer the apprentice's "why," we can put a finger on "wouldn't close properly" in the October 14 entry. That same entry tells us both what was wrong and what the machinist did about it.

What if the log had thirty thousand entries?

Give the model these three entries along with the question and we're on familiar ground. It can write an answer based on the text in front of it. Now imagine that the workshop has kept records for ten years and there are thirty thousand entries in the archive. Sending the whole archive with every question takes time and computing power. There's also a limit to how much text the model can process at once. Which page should we open?

Imagine a computer searching for the exact words "main door." It would have no trouble finding the first entry. Yet the entry with our answer contains neither "main" nor "door." As readers, we made two connections. We know where the receiving entrance is in this particular workshop. We also understand that taking something off and fitting a new one describes a replacement. We'd like our search system to find an account of the job even when the writer used different words.

One way to do that uses the **vectors** we met in Chapter 3. There we started with a list of numbers for a single token. Here the item we want to find is a whole maintenance entry. We can use an *embedding model* that represents each short entry as a single vector. During training for retrieval, the model can be adjusted to produce vectors that are close together for a question and a passage that answers it. We can calculate how close they are. For example, we can give two vectors a score based on how closely their directions align.

Before and after the question arrives

When preparing the archive, the application computes a vector once for each entry. Alongside it, the application stores the date and a record ID it can use to look up the original text. The vector for October 14 can be ready long before anyone knows what the apprentice will ask.

When the question arrives, the application computes a vector for the question as well. In our example, it can narrow the search to October and use the workshop's list of locations to add "receiving entrance" to the query. That little detail matters. An embedding model might connect "fitted a new one" with "replaced." How would it know that these two names refer to the same door in *our* workshop? Using the list of locations gives the search system that piece of local knowledge.

The application compares the question vector with the vectors for the candidate entries. It looks up the **original text** of a few entries that score highly and adds those passages, with their dates, to the chat model's context. Notice where the score's job ends. It helps us decide which page is worth opening. What lets us tell a reader that the door was bent is the sentence we find on that page.

With the October 14 entry in front of it, the chat model might answer:

> According to the October 14 maintenance entry, the leaf of the main door was bent and wouldn't close properly. The machinist took it off and fitted a new one.

The apprentice said, "we replaced the main door." The log gives us a finer detail. The machinist specifically recorded replacing the *leaf*, the part that swings in the frame. The date gives the apprentice a way to turn back to that entry and check the detail firsthand.

This method is called **retrieval-augmented generation** (**RAG**). *Retrieval* is the search that brings back relevant entries. *Generation* is the writing of the answer using a context that now includes those entries. Vector search is one way to retrieve them. In a well-kept log, a simple lookup by date and part number may do the job just as well. Either way, if a new entry is added tomorrow and made searchable, the chat model can use it without further training.

What if the wrong page turns up?

Try a small experiment. Cover the second and third entries in the table with your hand. Suppose the search returned only the October 3 entry about lubricating the lock. The apprentice's question is the same. What answer could you give from the one line you can see?

"They replaced the door because the lock was faulty" sounds plausible. Read the entry again. Someone lubricated the lock and then turned the key. Did the writer say the lock was faulty? Did they say anything that connects that work to the replacement later in the month?

Here's an answer the evidence would support. "The October 3 entry I have says the lock was lubricated. It doesn't say why the door was replaced." Now lift your hand. **We could say, "This entry gives no reason." If we'd said, "There's no reason in the archive," we'd have been wrong.** In a case like this, searching again for "receiving entrance" may do far more good than writing a longer answer from the same incomplete evidence.

We found the right entry. How much of it did we bring back?

This time the search finds the right page. But suppose that when we put the log on the computer, we stored each sentence as a separate passage. The search returns just this piece of the October 14 entry:

> It wouldn't close properly, so we took it off and fitted a new one.

Write that sentence on a slip of paper and hand it to someone who's never seen the workshop. They can tell you why something was taken off. But how would they know *what* was taken off, or why it wouldn't close properly? The apprentice's question mentions the main door, so it's tempting to fill in the gap. The very same sentence could have come from a maintenance entry about a cabinet.

Now put the preceding sentence back at the top of the slip. "The leaf at the receiving entrance was bent." With those few words restored, we can point both to the part that was removed and to the cause of the trouble. That's why **where we split a document** matters when we prepare it for search. In our log, keeping each dated entry together makes sense. In another log, where an entry might run to several pages, it might be more useful to retrieve a matching paragraph along with its heading and a neighboring paragraph.

When an assistant that uses RAG gives you a surprising answer, think back to this experiment. First ask, "Which lines from the log did the model see?" Then read those lines and ask, "Could I have drawn the same conclusion from them?" The first question tells you what the search turned up. The second helps you judge what the model did with it.

Let's close the log. On the phone, the apprentice finds a photograph taken on the morning of October 14. The top edge of the old door leaf is up against the frame. At the bottom corner there's a gap with light coming through. In the log, the machinist told us in words what had happened. Now all we have are pixels. How could a model work out from this image how the door fits in its frame? And could the morning photograph alone tell it that the leaf was replaced that afternoon?

Chapter 7 · Image perception, pixels, and visual features

How does a model notice light coming through under a door?

The apprentice opens the photograph taken on the morning of October 14. With two fingers, the apprentice zooms in until a single pale square fills the phone screen. “Which part of the door am I looking at?”

The machinist shrugs. That square could be a glint on the brass knocker or daylight coming in under the door. The apprentice zooms out a little and more pale squares appear beside the first. A little farther out, we can see a thin strip of light running along the bottom edge of the door and widening toward the right-hand corner. The original square is still there. Seeing what surrounds it changes how we read it.

Suppose the apprentice uploads the same photo to a chat application and asks, “Where is light coming through around the door?” How could a model use the relationship between that square and its surroundings?

Four pixels on paper

Each pixel in a photograph has a position and a set of numbers describing the color at that position. In a color image that keeps the red, green, and blue components separately, we can use three numbers per pixel. To make the arithmetic easy to follow by hand, let’s work with a grayscale copy of the photo. Give each pixel a single number from 0 to 255, with 0 for black and 255 for white. The larger the number, the lighter that spot appears.

Pick a patch near the bottom edge of the door, two pixels wide and two pixels high. Let’s give it these values:

	Left pixel	Right pixel
Top row	20	20
Bottom row	220	220

We have two dark pixels in the top row and two light ones in the bottom row. Let's answer a question with a single number. *How much brighter is the bottom half of the patch than the top half?* The bottom row averages 220 and the top row averages 20. Subtract the top average from the bottom average and we get **220 − 20 = 200**.

Imagine turning the paper a quarter turn. Both light pixels are now in the same column. Each row has one light pixel and one dark one, so both rows average 120. The same calculation now gives **120 − 120 = 0**. The four pixels have exactly the same total brightness as before. All we changed was their arrangement.

How we combine the four numbers determines which difference we pick up. Our first calculation responds to a horizontal edge, where brightness changes as we move from one row to the next. A second calculation comparing the columns would respond to a vertical edge. Write the two results side by side and we have a small **vector** of the kind we met in Chapter 3. Both numbers come from the same four pixels, yet each tells us something different about the patch.

How does the model learn that calculation?

Let's slow down the arithmetic. To average the top row, we added its two pixel values and divided by two. We could just as well take half of each value and add the halves. Each bottom pixel also contributes half its value to its row average. Where does the minus sign come in? We want the *bottom average minus the top average*. So we subtract the two halves from the top and add the two halves from the bottom.

Pixel	Brightness	Weight in the calculation	Contribution
Top left	20	−½	−10
Top right	20	−½	−10
Bottom left	220	+½	+110
Bottom right	220	+½	+110

Add the last column and we get −10 − 10 + 110 + 110 = **200**. The numbers in the third column are the **weights** we met in Chapter 1. A minus sign tells us to subtract brightness from that position, however bright or dark the pixel happens to be. Suppose all four pixels had a value of 220. The −220 from the top and the +220 from the bottom would cancel. We'd have a very bright patch with zero change in brightness from top to bottom.

We chose the weights because we knew which question we wanted to answer. In an image network, weights like these can be learned. Suppose we have lots of labeled photographs of doors, some with a gap along the bottom edge and some without. The network makes a prediction for each photo. The loss measures how well that prediction fits the label, and backpropagation supplies gradients for adjusting the weights in the layers that process the image. The training objective is to predict the labels correctly. Which weights prove useful depends on the examples and the task.

Our 200 tells us only so much. We might get exactly the same result at the edge of a white line painted on a wall. To say anything about a gap under a door, let's look at the rest of the photo.

How does one patch get information from another?

A **Vision Transformer** (ViT) applies the attention mechanism we explored with text in Chapter 3 to patches of an image. Imagine laying a square grid over the photograph. The part of the image inside each square is a *patch*. To keep the calculation small, let each patch again be two pixels wide and two pixels high. Read every patch in the same order: top left, top right, bottom left, bottom right. We can now write our patch as the list **(20, 20, 220, 220)**.

What's in the patch?

Let's calculate two numbers for each patch. For the first, use *bottom average minus top average*, as before. For the second, use *right average minus left average*. In our original patch, the right and left columns both average 120, so the second number is zero. Four brightness values have given us the vector **(200, 0)**. We can apply the same two calculations to other patches.

Pattern in the patch	Four pixel values	Resulting vector
Light bottom row	(20, 20, 220, 220)	(200, 0)
Light right column	(20, 220, 20, 220)	(0, 200)
Dark throughout	(20, 20, 20, 20)	(0, 0)

In a ViT working with a color photograph, all the color components of the patch enter the calculation. Different learned weighted sums produce the components of the vector representing that patch. We

can name our two components easily because we chose the calculations ourselves. What a component in a trained model has picked up can be much harder to put into words.

Where is the patch?

Imagine we see the same dark-to-light transition in two places, once near the knocker and once at the threshold. If the four brightness values are identical, our calculation gives **(200, 0)** for both patches. But the apprentice asked *where* the light was coming through. Where will that information come from?

In the original ViT design, each patch position has its own **position vector**, the same length as the patch vector. The values in these position vectors are learned during training. Let's choose some example numbers for our two positions so we can see how the addition works.

Patch position	Vector from the pixels	Position vector	Sum, component by component
Square near the knocker	(200, 0)	(1, 3)	(201, 3)
Square at the threshold	(200, 0)	(4, 2)	(204, 2)

In the first row we added 200 to 1 and 0 to 3. What does **(1, 3)** stand for here? Once we've located the square in the photograph, we look up the list of numbers assigned to that position. Reading those two numbers as "row one, column three" would miss that lookup step. If tomorrow's photograph has a dark patch in the same spot, its pixels will give us (0, 0) and we'll add the same position vector to it. Changing the content of a patch or moving that content to a new position can both change the numbers that reach the next layer.

So far the square at the threshold has drawn information only from its own pixels and position. In an **attention** layer, its representation is used to compare it with other squares. As with a token in Chapter 3, the model assigns different weights to the information it takes from different positions. Suppose light edges in neighboring squares line up. Information from those squares can help distinguish a long strip of light from a single glint. The vector passed to the next layer at the threshold position now carries information about its neighbors, too. Over several layers, information from small patches can come together to show how the door fits in its frame.

Who writes the words “at the bottom right”?

What an image model produces depends on the task we train it to do. Our simple classifier trained on door photos might give a score to each of two options, “gap” and “no gap.” The apprentice wants something more. The answer needs to say *where* the gap is, in an English sentence.

Imagine a chat model that processes the photo and the question together. Give it an image encoder, a language model, and a connection between the two whose weights can be learned. This is one way to bring images and text together. Let’s follow the apprentice’s question through it.

First the image encoder processes the photograph. The steps we’ve just followed produce a sequence of vectors. Some carry information about the change in brightness at the bottom of the door and how that change continues across neighboring squares. At this stage, nobody has to put any of it into English.

Next the visual information crosses into the language model. Remember our example vector with two numbers. Suppose the language model expects three numbers at each position. We could put a small layer in between that computes three different weighted sums of the two inputs. That gives us a list of the required length for each patch. Getting the dimensions to match is the first step. Training also has to shape *which visual details those three numbers carry, and how*, so the language model can use them.

Consider a single training example. A photo comes with the question “Where is the light coming from?” and the example answer “From the bottom right corner of the door.” If the language model assigns low probabilities to the tokens in that answer, we can calculate the loss we met in Chapter 2. Backpropagation carries the gradient back to the connecting layer, helping adjust its weights. Across many examples with light coming through in different places, a transformation that preserves location makes the correct answers easier to produce. That’s one way the two parts can learn to work together.

When the apprentice uploads the workshop photo, the language model can use both the vectors from the question’s tokens and those passed on from the photograph. As it answers “Where?”, some of its attention may fall on patches carrying location information. That information can then contribute to the scores for the next token. Token by token, the model might write, “Light is coming in under the door, and the gap looks wider at the right-hand corner.” We can now put a finger on what passes between the two parts of this model: lists of numbers computed from the photograph.

Can we fast-forward a photograph?

Try one last experiment. Show two people the whole photograph. Ask one, “Where does the strip of light get wider?” Ask the other, “Did the machinist replace the door leaf that afternoon?” The first person can point to the answer in the photo. The second can zoom in as far as they like and still won’t see what happened that afternoon.

We know the leaf was replaced because we read the maintenance log in Chapter 6. Imagine laying that entry beside the photograph. **Where was the gap?** We can point to the photograph. **What was done afterward?** We can point to the dated entry. When a model uses both in the same conversation, we can check its answer in much the same way. For each thing it tells us, which input would we turn back to?

Chapter 8 · Generating images from text with diffusion

How does a model draw a door from a description?

The machinist sets the brass knocker from the old door down on the workbench. “I wonder how it would look against dark green.” The apprentice takes out the phone. This time, there’s no photo, just a request:

> Draw a wooden workshop door painted dark green, with an old brass knocker on it.

The apprentice hasn’t said how many boards to use or what shape the knocker should be. Yet the finished picture will show the boards in some particular arrangement, and the knocker will have a particular shape. Where will those details come from?

One way to generate an image is through **diffusion**. There’s something curious about how we train a model to make a good picture this way. We start by spoiling pictures we already have. To see what that buys us, let’s bring back the four pixels from Chapter 7.

What’s left when we spoil the picture a little?

The top two pixels each had a brightness of 20, and the bottom two each had a brightness of 220. Let the training program generate a random number for each pixel, some positive and some negative. When we add these numbers to the clean values, some pixels get a little lighter and others a little darker. These random changes are what we mean here by **noise**. We’ll use simple addition in the table so we can follow the arithmetic. In diffusion training, the proportions of image and noise in the mixture depend on the chosen noise level.

Pixel	Clean value	Noise added by the program	Value the model sees
Top left	20	+8	28
Top right	20	−3	17
Bottom left	220	−5	215
Bottom right	220	+6	226

Run your finger along the first row. The model sees 28. Because we did the arithmetic, we know that 28 came from adding 8 to 20. Could the model work that out just by looking at 28? If we'd started with 25 and added 3, it would still see 28.

Now uncover the other pixels. There's a 17 beside the 28 in the top left and a 215 just below it. If the full image shows a dark surface with a bright strip underneath, we have a clue about the kind of region the 28 belongs to. That clue can't establish that we added exactly 8. But a model that has learned how neighboring values vary across many photographs can make better estimates than it could from the 28 alone.

Who wrote the answer key for training?

Let's give the diffusion model we're studying the noisy image, the noise level used to make it, and a written description of the image. Its task is to predict the noise added at each position. Suppose it predicts +6 for the top left pixel. The target the program saved was +8, so the estimate is off by two. If we tried to recover the clean value in our table using that estimate, we'd get 28 − 6 = 22. That's two units away from the original 20.

Now read each column of numbers from top to bottom. Follow the same route each time: top left, top right, bottom left, bottom right. The clean image gives us the vector **(20, 20, 220, 220)**, the added noise gives us **(+8, −3, −5, +6)**, and the image the model sees is **(28, 17, 215, 226)**. Each vector has four dimensions. All that means here is that we're working with four values, one for each of four positions. When adding the vectors, we add the first value to the first, the second to the second, and so on. It's the same calculation we did row by row in the table.

The model also predicts four noise values for this example. Suppose its prediction is **(+6, −3, −5, +4)**. Put it beside the target and you can see that the middle two predictions are exact, while the first and last are each off by two. It's important to compare each prediction with the target at its own position. If the model applies the correction for the first pixel to the last one, the bright and dark spots can end up in the wrong places even if the total brightness comes out right.

In Chapter 1, when we predicted how long a job would take, we had to check the clock to learn the answer. Here, nobody has to go through thousands of photographs labeling each pixel by hand. **The program generated the noise itself, so it already has the target**

it needs to score the prediction. We can calculate a loss from the prediction errors across all positions and use backpropagation to adjust the model's weights.

> **A closer look · How do four pixels give us one loss?**
>
> Take the calculation we used for order times in Chapter 1 and apply it to our noise predictions. We added +8 to the top left pixel and the model predicted +6. Taking the error as prediction minus target gives $6 - 8 = -2$. Its square is 4. The two middle predictions were exact. For the last pixel, the model predicted +4 instead of +6. The four squared errors, in order, are 4, 0, 0, and 4. Their average gives us the loss for this example: $(4 + 0 + 0 + 4)/4 = 2$.
>
> Now change only the first prediction. What would the loss be if the model predicted +7 there? The squared error for the first pixel would fall from 4 to 1. The last pixel still contributes 4, bringing the new average to $(1 + 0 + 0 + 4)/4 = 1.25$. That gives us a way to compare two predictions for the same noisy image. During training, we can work out how a small change in the weights would affect this average across many examples.

For one training example, use so little noise that the door is easy to make out. For another, turn down the contribution from the image and turn up the noise until even the edges are hard to see. That's why the model also receives the noise level as an input. The clues available for the next correction in a slightly grainy patch of wood are different from the clues in an image where we can't yet make out the boards.

Where does the machinist's description come in?

Suppose the pictures in the training collection have captions. Some say "brass knocker" and others say "black iron ring." The part of the model that processes text turns those words into vectors. When the image network estimates the noise, it can draw on information in those vectors as well. In Chapter 7 we followed attention from one image patch to another. Here, attention can also connect a region of the image with relevant words in the description.

Imagine a small bright spot that's barely visible against a dark surface. If the caption says "old brass knocker," a yellowish metallic glint would make sense in that region. If using such relationships be-

tween words and images helps the model predict the noise more accurately during training, the loss provides a signal for learning them. Later, when we generate a picture, the word “brass” can influence the colors and textures that emerge.

How can we take the first step without a clean picture?

When the apprentice taps Draw, the program generates a fresh array of random numbers. Displayed on a screen, it would look like static. This time there’s no photograph of a door that we started with and added noise to. What use is the noise prediction the model learned during training?

Think back to the training examples in which noise dominated the mixture. Even then, the model tried to make an estimate using the description and the visual patterns it had learned. It can now do the same calculation with the starting numbers and the machinist’s description. The program running the generation process uses the prediction together with the noise level for that step to calculate new numbers corresponding to a slightly lower noise level. **The result of the first step becomes the input to the second.** The model looks at the new numbers and makes its estimate again.

In our four-pixel example, we can follow a step with two lists. One holds the four numbers representing the current image. The other holds the model’s noise estimates for those four positions. Using the two vectors and the noise level for that step, the program calculates another four-number image vector. On the next pass, the model looks at that updated vector and predicts a **new noise vector**. Remember the target column we put beside the prediction during training? There’s no such column during generation. We know the random numbers we started with, but we have no ready-made list of corrections that will take us from those numbers to one particular door matching the description.

Why take several steps? At the start, there isn’t even a recognizable door surface to give the knocker a setting. After a few updates, a surface that looks like a door may start to emerge. The next prediction can draw on where that surface is. The model has the same description at every step, but the image on which it bases its prediction keeps changing. The weights learned during training stay fixed throughout. What we update are the numbers representing the picture we’re making.

Some systems carry out these steps on a smaller numerical representation of the image, called a *latent representation*. At the end, a decoder turns that representation into the color pixels we see on screen. Working directly with brightness in our four-pixel example lets us follow the same learning idea on paper.

What does “high-dimensional” mean at the scale of a real picture? A color image 512 pixels wide and 512 pixels tall, with red, green, and blue values for every pixel, holds **512 × 512 × 3 = 786,432** numbers. If we put them all in one list, each image would be a vector with 786,432 components. A program can work with those numbers while keeping their row, column, and color channel structure. In a model that generates in a latent representation, the numbers being updated carry image features encoded on a smaller grid. At either scale, the relationship to remember from our four pixels is the same: each component of the predicted correction corresponds to a component of the representation being updated.

If the same prompt gives us two doors, which one is right?

Imagine hanging three drawings side by side. We’ll use the same model and generation settings for all three. To make the starting noise, the program needs a great many random values. A computer can produce them in a repeatable sequence from a starting number called a **seed**. Choosing 17 as the seed doesn’t fill every pixel with 17. It starts the number generator at a particular point. The different values the generator produces in succession make up the initial noise vector. Start the same generator again with 17 under the same conditions and we get the same sequence.

That’s handy for the apprentice. If we change both the prompt and the random starting point, it’s hard to tell which change accounts for a difference between two drawings. For our experiment, assume that using the same seed lets us repeat both the initial noise and any random choices made in later steps.

Drawing	Prompt	Seed
A	Dark green wooden workshop door, old brass knocker	17
B	Dark green wooden workshop door, old brass knocker	42
C	Dark green wooden workshop door, black iron ring	17

First compare **A and B**. Suppose both satisfy the prompt. If one shows three wide boards and the other shows six narrow ones, which would you mark wrong? The machinist never specified a number of boards. Updates from different starting points can lead to two different doors that both fit the description.

Now compare **A and C**. The starting point is the same, but we've changed the prompt. The very first noise prediction draws on different words. Later steps carry forward the effects of that new prediction. Along with the material of the ring, the shadow it casts on the door or even the arrangement of the boards may change.

The machinist holds the old knocker beside the drawing. The one in the picture is brass, too, but the real one has a small notch in its rim from a fall years ago. The apprentice never mentioned that notch. With the photograph in Chapter 7, we could ask, "Where can you see that detail?" If we wanted the notch to appear in the right place in this drawing, what would we add to the description? Perhaps the simplest thing would be to include a photograph of the knocker on the workbench.

Chapter 9 · Generating images from a reference and editing with a mask

If we supply a photo of the knocker, will its notch survive?

The apprentice takes a photograph of the brass knocker on the workbench. The machinist has picked one of the green doors they drew in Chapter 8. The request sounds simple enough. “Show me how this knocker would look mounted on that door. I want to be able to tell it’s ours.”

There’s a small notch in the outer rim. On the first try, the apprentice gives the program the reference photo and the drawing of the door. The brass comes out the right color, the ring is the right size, and its shadow falls across the green wood. The machinist zooms in. “What happened to our notch?”

This time the apprentice supplied the missing detail. You can see the notch in the photograph. How does a detail get from a photo into the finished picture? Let’s follow three things the apprentice can try on the computer.

What if we cut the knocker out of the photo and place it on the door?

The apprentice opens an ordinary image editor, selects the knocker, and moves the selection onto the picture of the door. Imagine doing this with a printed photograph and a pair of scissors. If you cut only around the outer rim, you’ll still see the workbench inside the ring. Cut out the middle as well and the green door will show through. The apprentice takes the same care on screen, following the outline all the way into the notch. Once the knocker is in place, a sliver of green wood shows through that tiny indentation, too. Suppose the apprentice keeps the knocker the same size and simply slides it to the right. A dark spot on its surface moves just as far to the right as the notch does. We could measure the distance between them, pixel by pixel, in both the source photo and the result.

That leaves us with another problem. On the workbench, the light fell on the knocker from the right. In the drawing, the door is lit from the left. When the apprentice moved the brass, the bright reflection came along with it. The machinist recognizes the notch, but the knocker looks like a sticker on the door. To make it look at home in this light, some pixels will have to change.

How does a reference photo guide generation?

Now imagine using a system that can generate an image with a reference photo to guide it. When the apprentice typed just “brass knocker,” all sorts of shapes fit the description. The photograph gives us more to go on. Suppose this particular ring is thicker at the bottom than at the top. A bright band along its lower edge gives us a clue to how the metal curves there.

In our example, the part that processes the photo is called an **image encoder**. It uses numbers to describe how colors, edges, and textures come together in different regions. The information those numbers carry together is the photo’s *numerical representation*. As readers, we look at the image and say, “The ring is thicker at the bottom.” The model can use that shape without first putting it into words. In Chapter 8, the model estimated a correction at each step using the current noisy image and the written description. Here, that estimate also depends on features taken from the reference photograph.

What difference does that make to the picture? Pick a pixel on the right-hand edge of the knocker as it appears on the door. What determines its color? When we cut the knocker out and moved it, we could point to that pixel’s address in the original photo. In a generated picture, the color at that position takes shape over a series of steps. The model can draw the curved brass surface in a way that suits the new lighting. Those same calculations can also give the rim a smooth, unbroken curve.

If the notch is especially small, there are two places we might lose track of it. Suppose the input photo is reduced to a quarter of its original width before it reaches the encoder. A notch two pixels wide in the original now spans half a pixel. There’s no separate half-pixel square in which to show it. As the photo shrinks, the dark color inside the notch and the light brass beside it may both contribute to the color of a single pixel. Enlarge the little picture again and all you might see is a faint spot. It’s hard to tell from that color whether the indentation was sharp or rounded. There’s a second possibility. The notch may be quite distinct in the input, yet the model may draw a

knocker without it. Looking only at the finished picture, could we tell where the detail was lost? Either route could give us the same smooth rim.

What if we redraw just part of the door?

The machinist lays a printout of the door on the workbench, places a sheet of tracing paper over it, and shades in the old knocker with a pencil. “Let’s draw ours here. I like that knot in the wood up in the corner. Let’s keep it.” Lift the tracing paper and the pencil marks come away with it. Lay it back down and the shading shows exactly which part of the door is to change.

The apprentice does much the same thing on screen, brushing over the old knocker and its shadow. The program shows the marked area in translucent red. If the brush strays over the knot, the apprentice can erase that part of the mark. The separate layer that keeps track of the selected area is called a **mask**. The red helps the apprentice see the selection. What to draw there is another question, answered by the machinist’s description and the reference photo.

How might the program store that selection? Imagine a cell in the top layer for every little square of color in the door image, one cell per pixel. In our example, put a 1 wherever the brush passed and a 0 everywhere else. A 1 means “let this part be redrawn,” and a 0 means “keep the existing picture here.” When filling the marked area, the model can draw on the surrounding wood. If the new knocker is smaller than the old one, for instance, it can continue the grain across the newly exposed area, following the lines on either side. Filling a selected region of an image using clues from its surroundings is called *inpainting*.

Suppose the program uses the mask to combine the original picture with the newly generated one. Let’s follow the brightness of four neighboring pixels in a single row of the door. A larger number means a lighter pixel.

	Pixel 1	Pixel 2	Pixel 3	Pixel 4
Original door image	30	34	32	29
Generated image	35	90	72	24
Mask	0	1	1	0
Combined result	30	90	72	29

The new image had 35 at the first pixel. We can point to the reason we still see 30 in the result. At that position, the program took the value from the original door. For the two middle pixels, it used the generated values. What if the apprentice chooses a soft-edged brush? At a pixel along the boundary, we could ask the program to blend the old and new values in equal proportions. At the second pixel in our table, that would give us the average of 34 and 90, or 62. A gradual transition like that can help where a shadow fades across the green wood.

Now consider the machinist’s two requests in light of the table. If the knot in the upper corner falls outside the mask, combining the images in this way keeps the original knot intact. The notch in the knocker, on the other hand, falls inside the area to be redrawn. Where in the mask could the model find the shape of that notch? For that detail, we still depend on the reference photograph and how the system uses it during generation.

The apprentice tries once more. This time the first step is to place the cutout of the real knocker on the door by hand. Then the apprentice marks only the area where its shadow will fall and the green wood around it for redrawing. The machinist zooms in and finds the notch. And the old highlight from the light on the right? That’s still there, too. Keeping the knocker’s surface preserved both details.

Now the machinist wants to change just the highlight on the brass. If you had the brush, which part of the knocker would you include in the mask, and which would you leave outside it? When you check the result, look in two places. Does the outline of the notch still match the original photo? Does the new position of the highlight agree with the shadow on the door?

Chapter 10 · Sound perception, waveforms, and spectrograms

How can a model tell that someone knocked twice?

The machinist mounts the brass knocker on the green door. The apprentice starts the first recording on the phone. The machinist lifts the ring and lets it fall. Clack! A short pause, then another knock. The apprentice waits for the faint ringing of the brass to die away before stopping the recording.

For the second recording, the machinist moves over to the wooden workbench. The apprentice taps Record again. This time the machinist raps the bench twice with a knuckle, leaving the same pause between knocks. Two short, dull taps. The apprentice stops the recording and hands over the phone. The machinist picks one of the recordings and plays it. “The knocker,” says the apprentice. “You can still hear it ringing at the end.” The machinist plays the other one. “I tapped out the same rhythm both times. What did the phone record that lets you hear the difference now?”

How does one knock become thousands of numbers?

When the knocker strikes the door, both the knocker and the door vibrate. The air nearby compresses and expands a little in turn. Those small pressure changes travel to the phone’s microphone, moving a thin diaphragm inside it back and forth. The microphone converts that movement into a changing electrical signal.

If it were your job to trace that signal on paper, you could move a pen from left to right at a steady speed, moving the tip up as the signal rose and down as it fell. The phone’s recording circuit can measure the signal at equally spaced instants and store each reading as a number. One such measurement is a **sample**. The number of samples taken per second is the **sample rate**. Let’s say our single microphone records at 16,000 samples per second. Whether the machinist knocks once in that second or three times, we get 16,000 numbers. What tells the knocks apart is how those numbers change over time.

What does a single sample of zero tell us? At that instant, the signal was at its baseline. A rapidly vibrating diaphragm passes through its resting position on the way out and again on the way back. A lone zero gives us little grounds for declaring the workshop quiet. If the readings on either side also stay close to zero, then we have evidence that this part of the recording was quiet.

The apprentice zooms in on the recording. Imagine plotting each sample as a dot at the instant it was taken. The shape we see as we follow the dots from left to right is the **waveform**. The line swings to either side of the baseline as the air pressure rises and falls. A single strike of the knocker can send that line up and down many times. If the apprentice counted every peak as a knock, we'd think the machinist had been waiting at the door for quite a while.

Try it with a ruler. Hold one end down on a table, pull the overhanging end down, and let go. You released it once, but the tip went up and down many times. A strike of the knocker starts a vibration in much the same way. Zoom in on the recording and you see the oscillations within a single knock. Look at a longer stretch and you can see how those oscillations start up and die away together.

Where can we see the two knocks?

Let's zoom out a little. At two points in the recording, the apprentice sees the trace suddenly spread into a wide band and then narrow again. The rapid vibrations we saw up close now fit inside those two swells. Suppose the first begins 0.20 seconds into the recording and the second begins at 0.70 seconds. The machinist struck again half a second after the first knock.

> **A closer look · How can we measure the time between knocks from the samples?**
>
> Could the apprentice work out the same interval if the screen showed sample numbers instead of seconds? Number the first sample in the recording 0. At 16,000 samples per second, the first knock starts at sample $0.20 \times 16{,}000 = 3{,}200$. The second begins at 0.70 seconds, or sample 11,200. There are $11{,}200 - 3{,}200 = 8{,}000$ sample intervals between the two onsets. To convert that count to seconds, divide by the sample rate. We get $8{,}000 / 16{,}000 = 0.50$ seconds.
>
> What if a second phone recorded the same two knocks at 32,000 samples per second? We'd count 16,000 sample intervals between the onsets. The elapsed time would still

be $16{,}000/32{,}000 = 0.50$ seconds. The machinist kept the same rhythm. The second phone took measurements twice as often during that half second.

How could a program pick up this change? Divide the recording into short stretches of time. In each one, find the average distance of the readings from the baseline. Four units above the line and four units below it both count as a distance of four, so the two directions of a strong vibration won't cancel out. We get one number summarizing the strength of the vibration over that short stretch.

Let's work through four readings. Suppose we measure +4, −4, +6, and −6 in succession during a knock. Their sum is zero. What interests us here is how far each reading went from the baseline. The distances are 4, 4, 6, and 6. They add up to 20 and average 5. In a quiet stretch, readings of +1, 0, −1, and 0 would give an average distance of 0.5. By repeating this calculation for successive stretches, a program can track where the vibration gets stronger. At 16,000 samples per second, even a hundredth of a second holds 160 readings.

In our clean recording, the average starts small, suddenly rises, and soon drops back down. Half a second later, it rises again. We've found two onsets. But suppose the machinist had struck the knocker the first time and the bench the second. We might still find two rises. From the timing alone, the apprentice couldn't explain why the second sound was different.

Where is the ringing in the recording?

Imagine tapping a table once with your finger, then lightly tapping the rim of a drinking glass. With the glass, you may hear a high note that you can follow even after you've lifted your finger. That's the kind of detail the apprentice picked up when listening to the two workshop recordings. Once a knock begins, which vibrations can we hear, and how long do they last?

A vibration that repeats a hundred times a second has a **frequency** of 100 hertz. A vibration that repeats a thousand times a second has a higher frequency and, heard on its own, a higher pitch. A strike of the knocker sets off many vibrations at different frequencies at once. What reaches the microphone is their combination. Some die away quickly, while others linger.

We've now met two different rates. The 16,000 samples per second tell us how often the phone measures the signal. The 100 hertz tell us how many complete cycles the vibration makes each second. With just that 100-hertz tone playing, the phone takes 160 samples in one

cycle. If we record a 1,000-hertz tone on the same phone, we get 16 samples per cycle. The phone's sampling clock keeps ticking at the same rate. The higher tone fits more cycles into the same time.

The apprentice takes out a sheet of graph paper. Let each column stand for a short stretch of the recording. Give the lower rows to slower vibrations and use higher rows for faster ones. A program works out how much energy falls around each frequency in that stretch. The apprentice shades the cells with more energy darker. This kind of display is called a **spectrogram**. Move to the right across the page and you can follow the sound through time. Read up a single column and you can see which frequencies are strongest during that stretch.

How can readings from one microphone give us several rows to shade in the same column? The program can take a short stretch of sound and work out how much of it can be accounted for by regular oscillations at different rates. Think of playing two piano keys together. A single pressure wave reaches your ear, yet you can hear both the lower note and the higher one in it. The program likewise obtains separate values for the slower and faster oscillations in that short stretch. Moving on to the next stretch gives us the next column.

In our recording of the knocker, many rows may darken at the instant of impact. In the columns that follow, most of those rows grow pale, but a few cells high on the page may stay dark for a little longer. The apprentice heard that horizontal streak as a faint, high ringing sound. Now imagine looking at the same stretch in the recording of the bench and seeing the streak fade sooner. The time between knocks is the same in both recordings, but the two pages now look different.

Who tells the model that streak belongs to the knocker?

The apprentice sets up a small experiment, recording many strikes from different distances, some hard and some gentle. Knowing what was struck each time, the apprentice can write "knocker" or "workbench" beside each recording. That word is the **label** used in training. A model that works with spectrograms takes the grid of numbers for a recording as input and calculates a score for each label. If it gives "knocker" a high score on a recording of the bench, the mismatch between its prediction and the apprentice's label helps guide

an adjustment to its weights. The learning idea from Chapter 1 is at work again.

What if, on the first day, the apprentice recorded every strike of the knocker up close and every tap on the workbench from across the room? The model could get many examples right simply by calling louder sounds “knocker.” The label tells it which object was struck. It leaves open which clue the model will use to arrive at its answer. So the apprentice records both sounds from near and far, then checks the predictions on recordings held out of training. A knocker recorded from across the room and a workbench recorded up close are particularly good tests of whether the model relies too heavily on loudness.

The machinist is curious and tries something. Holding on to the brass ring, the machinist strikes the door twice with it. Contact with the hand quickly damps the vibration. The apprentice can still count two knocks, but the familiar ringing is gone. If every knocker the model heard during training rang freely, it might call this recording “workbench.” To test that idea, the machinist makes one more recording, this time letting the ring go. Same knocker, same door, same rhythm. Comparing the two predictions can give us a clue to how much the ringing matters to the model.

The apprentice has left the recording running. At the end we hear the machinist’s voice: “The door is open.” The phone recorded those words in the same way, as a sequence of numbers. When the apprentice says “The door is open,” most of the numbers will change. How can a model transcribe both recordings and come up with the same words?

Chapter 11 · Speech recognition and text-to-speech synthesis

What changes when two people say the same sentence?

The apprentice plays the end of yesterday's recording, the one that was left running. The machinist's voice comes through the speaker. "The door is open." The apprentice starts a new recording and says the same words. Then the phone transcribes both recordings. Two lines appear on the screen, one below the other. Both read "The door is open."

The machinist mutes the phone and points to one of the lines. "Which of us said that?" The apprentice laughs. Without remembering the order, who could tell? A moment ago the apprentice recognized both voices from the very first word. Some details fell away on the way from recording to text. Let's see how the two voices came to produce the same line on the screen. Then we'll give the phone just that line and ask it to speak.

What stays the same inside a word?

The machinist says "open" slowly. The apprentice repeats it in a hurry. Now try it yourself. As you say "open," notice when your lips touch. You can keep the opening "oh" sound going with your lips apart. They come together for the "p," then part again as you finish the word. Try drawing it out. You can spend longer on "oh," but your lips still close and reopen at the same point in the sequence.

The phone's microphone can't see your lips. It can pick up the trace their closing and opening leave in the sound. In Chapter 10 we laid short stretches of time side by side in a grid showing which frequencies were strong in each stretch. Let's use that same spectrogram here. While you sustain "oh," energy stands out in certain frequency bands. As the lips close, that energy drops. There can be a brief burst as the lips part, followed by the pattern of sounds that finishes the word. In the apprentice's faster recording, those changes fit into fewer columns.

It would be hard to lay the two grids on top of each other and expect exactly the same cells to darken. The durations differ, and so do the speakers' voices. Yet **the order in which the changes occur** is a clue in both recordings. A lip closure alone won't identify a "p." Your lips also close for "m." Say "happy" and "hammy" aloud. In "hammy," listen for the hum you can keep going through your nose while your lips are together. To distinguish two movements that look alike, it helps to hear what happens just before, during, and just after the closure.

Where does the space go in a recording?

This time the apprentice says "The door is open," deliberately pausing before "open." The gap is easy to spot on the phone. Then the apprentice says the sentence in one breath, as if calling out to a friend at the door. There's still a space before "open" on the page. A neat stretch of silence in the corresponding spot on the recording is harder to find. And we've just heard the sound grow briefly quieter at the "p" *inside* a word. Treat every quiet patch as a word boundary and we might split "open" in two.

Consider how a **speech recognition** model could learn. Suppose we have recordings from many speakers, each paired with a line transcribing what was said. During training, the model processes short stretches of audio. It draws longer sound patterns from adjacent stretches and uses those patterns to predict a sequence of text. The text can be produced in small pieces, each a *token* of the kind we met in earlier chapters. A token can cover several letters, and the sound that supports it can extend across many time slices in the recording.

Suppose the model has written "The door was open" so far. At the end of this particular recording there's also a brief "d" sound. The training transcript reads "The door was opened." If our example tokenizer treats the ending "ed" as a separate token, assigning it a low probability contributes to the loss. As the weights are adjusted, predictions that take that final sound into account can gain ground. In another recording the speaker might actually say "The door was open." This time the right thing to do is stop there. Working with many recordings and transcripts gives the model a chance to learn both where the speech continues and which pieces of text go with what it hears. The apprentice needn't mark the instant when every letter begins.

To test the idea, the apprentice makes three short recordings. In the first, the machinist says “hopper.” In the second, the apprentice says the same word in a higher voice. In the third, the apprentice speaks again, this time saying “hippo.” Going from the first recording to the second, we’d like the transcript to keep “hopper.” Going from the second to the third, we’d like it to pick up the change in both vowels and give us “hippo.” In all three recordings the lips close for “p.” That clue alone leaves the choice of word wide open.

Try singing “hee” and “hoo” on the same note. Even if you keep the pitch steady, you can feel your lips round as you move into “hoo.” Changing the shape of your mouth reinforces some frequencies more strongly than others. On a spectrogram, the regions where energy is concentrated shift. Now say “hee” once in a low voice and once in a high voice. The pitch changes, yet you can hear the same vowel both times. Training examples from different speakers give a model a chance to learn which changes in the sound tend to go with changes on the page. A quick “hopper” and a slow “hopper” can yield the same text, while “hippo,” spoken by the same person at the same speed, can yield a different word.

What if a hammer falls on the bench just as the machinist finishes the sentence? “The door is” might come through clearly while the end is buried in noise. Drawing on word sequences it encountered during training, the model might still write “open.” Reading the tidy sentence on the screen, the apprentice might never suspect how hard the last syllable was to hear. You can try this with a friend. Play just the final syllable from the noisy recording, then play the recording from the beginning. After hearing the earlier words, your friend may have a much stronger hunch about that same indistinct sound.

Now give the phone just the text

The apprentice types “The door is open.” on a new page and taps **Read Aloud**. This time the program has neither person’s recording. Still, a voice comes through the speaker. It says “open” at a particular pace, and its pitch falls toward the end of the sentence. The machinist looks at the page. “Where does it say how long to hold that sound?”

Try reading the line two ways. First, answer someone calling from the street, “Can I come in?” Then imagine arriving at the workshop in the morning and finding the door you locked last night standing ajar. Say the same four words again. The second time, you may linger a little on “open.” The duration of syllables, the rise and fall

of pitch, and the placement of stress are all aspects of **prosody**. Punctuation gives us a clue, but two readings that end with the same period can sound quite different.

Here's one route a **speech synthesis** system can take. The first component works out which speech sounds should follow one another and predicts how they might sound as the sentence unfolds. Picture its output on the graph paper from Chapter 10. This time we're filling in the grid for a sentence that has yet to be spoken, predicting when each frequency band will grow stronger. If the "oh" in "open" is drawn out, its pattern will run across more columns. A choice of speaker or reading style, if the system takes one as input, can also influence the prediction.

How could the system learn from a written line how many columns to allow for a syllable? Imagine a training recording in which the speaker draws out that "oh." In the grid computed from the recording, the vowel leaves a long trace. If the grid predicted from the text moves on to "p" too early, it will already show the change associated with the lips closing while the vowel is still sounding in the actual recording. The difference between the grids can be used to adjust the model's weights. Across many examples, words, punctuation, and delivery vary together, giving the model a chance to learn patterns of timing and stress that suit different texts.

The machinist chooses two reading voices on the phone and listens to the same sentence in each. Even with the reading speeds matched, the voices are easy to tell apart. In everyday speech we might call that the **color of a voice**. In music the term is **timbre**. Even when two people hold the same vowel on the same note, the higher-frequency vibrations accompanying the fundamental can differ in strength. If a synthesis system trains on multiple voices and the speaker is identified in each recording, the model can learn to use an input indicating who is speaking. When the apprentice selects another voice from the menu, "The door is open" stays on the page, but the energy distribution in the predicted grid can change. Drawing out "open" changes something else again. Try it in your own voice. Say the word briefly, then stretch it out in surprise.

The second component's job is to turn that grid into an audio waveform that can drive a speaker. The component that does this is called a *vocoder*. Each column on our graph paper summarized a short stretch of time. To move the speaker, we need the individual waveform values that follow one another throughout that stretch. A vocoder trained on real recordings paired with their grids can learn to generate a waveform consistent with a given grid. When the

phone converts those values into an electrical signal and sends it to the speaker, the air begins to vibrate again.

The machinist asks for one last experiment. The apprentice taps **Play** on the machinist's original recording. A faint squeak from a chair in the background sounds again. Then the apprentice taps the line transcribed from that recording, "The door is open," and chooses **Read Aloud**. The words come back. When should the chair squeak? If that line were all the apprentice had left, could it tell us whether the squeak came before or after the word "open"?

Chapter 12 · Perceiving motion in video and generating video

Is the door opening or closing?

The apprentice props the phone against the workbench. The green door is closed, and the camera faces it squarely. As the apprentice starts recording, the machinist slowly opens the door a little. Two seconds later, the apprentice stops the recording and picks a still from the middle to show the machinist. There's a gap about a handspan wide between the door and its frame. "You remember filming this. Could someone who saw only this picture tell which way the door was moving?"

The apprentice plays the whole clip with the sound off. The gap gets wider. Then the apprentice plays the same clip backward. Now the door is closing. Pause in the middle and they're looking at the same handspan of space. The still gave them a clue to how far open the door was. Seeing what came just before and after it lets them answer a different question: which way was it moving?

Where is the motion in the pictures?

Draw a little door in the bottom right corner of a page in a notebook. On the next page, draw it slightly ajar. On the page after that, open it a little farther. Run your thumb along the edges of the pages and the door moves. Tear out one page and lay it on the table. You still have the shape of the door at that instant. To tell which way it was moving, you need to know how that page relates to the others.

Each individual picture in a video is called a **frame**. If the apprentice's phone records 25 frames per second, a two-second clip contains 50 frames. The phone captures successive frames one twenty-fifth of a second, or 40 milliseconds, apart. Play those same 50 frames at 50 frames per second and the machinist opens the door in one second on screen. The order of the frames affects the direction of the motion we see. The time between frames at playback affects how fast it looks.

Let's pick three frames at equal intervals from the 50. The phone stays still. Choose a height near the top of the door and measure the width of the bright strip between the free edge of the door and the

doorframe at that height in each picture. We'll use the following values to make the clip easy to follow on paper.

Playback order	First picture	Second picture	Third picture
Width of the strip going forward	4 pixels	10 pixels	16 pixels
Width of the strip going backward	16 pixels	10 pixels	4 pixels

Average the three values in either row and you get 10. Yet in the first row the strip gains 6 pixels at each step, while in the second it loses 6. If all we give a system is the average, we've discarded the information it needs to find the direction. Even a calculation as simple as subtracting the first measurement from the second works here because it takes the order into account.

> **A closer look · Which way is the width changing, and how fast?**
>
> Suppose the apprentice chose the three pictures at 0.40, 0.60, and 0.80 seconds into the recording. Between the first two, the strip widens from 4 pixels to 10. The change is $10 - 4 = +6$ pixels. It took 0.20 seconds, so the average rate of widening over that interval was $6 / 0.20 = 30$ pixels per second. In the next interval the width goes from 10 pixels to 16 in the same amount of time.
>
> Now play the recording backward at the same speed. In the first interval the width falls from 16 pixels to 10, giving us $10 - 16 = -6$ pixels. Again, 0.20 seconds have passed on screen, so the rate is −30 pixels per second. The plus sign tells us the strip is widening and the minus sign tells us it's narrowing. What we've measured is the rate of change in the width of the bright strip *in the picture*. To work out how many degrees the door actually turned, we'd need more information about the positions of the camera and the door.

How does the model find the edge to follow?

The apprentice knew which strip to measure. In the images we give the model, there's no arrow pointing out the edge of the door. What's more, the brass knocker changes position on screen as the

door opens. Look in the second frame at the spot where the knocker was in the first and you might find a plain expanse of green wood. Keeping your eye on the same spot can mean losing track of the object.

Consider one approach a video model can take. It divides each frame into small regions and uses a group of numbers to represent how colors, edges, and textures fit together in each region. That's the idea we used for a single image in Chapter 7. This time, alongside each region's location in the picture, the model also takes into account **which frame it came from**. It can relate the region containing the ring in one frame to a similar region slightly to one side in the next. The doorframe stays in the same place in both pictures, giving the model a reference for how the ring moves relative to it.

How does the model learn which relationships are useful? Imagine the apprentice collecting short clips of doors, some opening and some closing, and labeling each clip with the action it shows. During training, errors in predicting those labels guide adjustments to the model's weights. Suppose the apprentice includes the same recording played forward and backward, each with its correct label. The green paint, the shape of the knocker, and the back wall are the same in both examples. Even so, the expected answer changes. The pair shows why it pays to attend to changes across frames. To see whether the model has learned to do that, we could play a clip of a different door in both directions and compare its predictions.

The apprentice tries one more experiment. This time the machinist opens the door and closes it again, all within a second. If we select just the first and last frames for the model to examine, both show a closed door. Could the model tell for sure what happened in between? Give it one more frame from the middle, showing the door ajar, and it has new evidence. How many frames the phone **records** per second and which frames the program **examines** are two separate decisions. The apprentice can have the whole action on the phone and still leave it out of the pictures the model sees.

If we draw the same door fifty times, do we have a film?

The machinist closes the door and sits down at the workbench. The apprentice types, "The green workshop door slowly opens." In Chapter 8 we saw how to generate a picture of a door from a description. What if the apprentice generated each frame separately, asking each time for the door to stand a little farther open? There'd be fifty

pictures to play in sequence. But a knot in the wood might appear above the knocker in the third picture and below it in the fourth. Pause on either picture and it looks perfectly plausible. Press Play and the knot jumps.

Let’s adapt the method we used in Chapter 8 to generate an image from noise. This time the numbers in front of the model represent a short sequence of frames. During training, we can take that representation of a real clip and add random noise whose values we know. When we ask the model to predict the added noise, it can draw on neighboring frames to make sense of an indistinct region in one frame. If the knocker is blurred, its position in the frames just before and after gives a clue to where the ring might have passed in between. Working with many clips gives the model a chance to learn how shapes change in appearance as they move.

The apprentice singles out one word in the prompt: “slowly.” Suppose the training clips come with captions describing the action. In two examples, the door starts closed and ends up a handspan open. In one it takes the full two seconds to get there. In the other it opens at once and stays there. The first and last frames may look alike, but the frames in between differ. If the model also draws on the text when predicting a correction, examples like these can teach it what kind of change goes with the words “slowly opens.” The apprentice needn’t specify the angle of the door in degrees for every frame.

During generation, we have the apprentice’s prompt and some random numbers to start with. The real clip we used for training is gone. At each step, the model predicts corrections using the current state of several frames together. By the end of one step, we’ve refined the numerical representation of the whole short sequence a little. The first frame is still a work in progress, just like the others. This is one way to generate video. Letting information pass between frames helps the knocker turn with the door as it swings open.

Picture the two sequences involved here. The machinist lays three sheets of paper side by side on the table. The one on the left represents the beginning of the clip, the middle one a moment halfway through, and the one on the right the end. The apprentice works on all three sheets in one round of corrections, then returns to the same three in the next round. **Move from left to right across the table and you move through time in the clip. Return to the same sheet one round later and you’re refining the drawing of the same instant.** Once the work is done, playing the pictures in order from left to right gives us the motion on screen.

Now imagine the first sheet is a photograph the machinist took of the closed door. The machinist supplies it as the starting frame and asks to see the door open a little. With the door shut, we can't see so much as a shelf in the workshop behind it. If the end of a blue shelf appears through the narrow gap in the middle frame, what would you expect in the last one? As the door opens farther, more of the shelf might come into view. If a red curtain suddenly appears in that spot, the change will catch your eye. The model has two jobs here. It has to generate something for a place hidden in the original photograph, then keep track of that detail as the gap widens. Even if the shelf is beautifully drawn in every frame where it appears, the machinist might say, "There's no shelf like that in my workshop." The photograph never showed the model what was actually behind the door.

This time the apprentice also supplies the photograph of the notched knocker from Chapter 9. The machinist pauses the generated clip. In the first frame, the notch is on the outer rim of the ring. As the door turns, the ring looks narrower and the notch shifts sideways on screen. The machinist zooms in on the last frame. To find the notch, should the machinist put a finger on the **same spot on the screen** as before, or look for the **same part of the rim** on the ring now that it has turned? If that part of the ring has passed out of sight, we'd expect to lose sight of the notch, too. But if that part is still visible and the notch suddenly appears on the opposite edge, we've caught a change between two well-drawn frames that the motion can't explain.

Chapter 13 · Sound and picture in sync, event order, and causality in video

Two knocks. Was it the machinist who opened the door?

At the start of the four-second recording, the green door is closed. A hand reaches in from outside, lifts the brass ring, and lets it fall. The ring strikes the metal plate. Clack. The door stays shut. The same hand lifts the ring again and lets go. A second clack. The hand withdraws. After a short pause, the door begins to swing inward. The camera shows only the upper half of the door. Whoever opened it, neither their hand nor their face comes into view.

The apprentice wants to enter what happened in the maintenance log and brings the phone to the machinist. “Listen to it first. Then we’ll watch it and decide what to write.” The machinist takes out a pen. The apprentice rewinds the clip and turns the phone face down.

First, just listen

Clack. A short pause. Clack. A faint ringing lingers after the second sound. Then the recording ends.

What would you write in the log? “I heard two knocks,” the machinist begins. The apprentice waits. “Sounds like our brass knocker.” How could we check those two statements? For the first, we can replay the clip and count the knocks together. For the second, we could listen to the knocker and workbench recordings from Chapter 10 and compare the ringing. A blow to another piece of metal nearby might have made a similar sound. The machinist leaves room for that possibility with the words “sounds like.”

Now mute the phone and turn it face up. A hand reaches up from below, lifts the ring, and lets it fall. The ring swings back against the plate. The hand repeats the movement and slips out of view. A moment later, a narrow gap appears between the door and its frame and begins to widen. Pause the clip and we still can’t see who’s inside.

“They lifted it twice,” says the apprentice. This time there are two moments to point to on the screen. But if all we’d seen was this silent clip, could we tell how long the knocker rang after the second strike? Seeing the ring reach the plate makes it easy to imagine the sound. To measure how long the ringing lasted, we’d still need the audio.

What do we learn by putting sound and picture together?

The apprentice turns the sound back on. We hear the first clack just as the ring swings back against the plate. The same goes for the second. The door starts to open after the second knock. Now we have more than two separate counts of two. **We can tell which sound coincides with which movement.**

Try a small editing experiment. On a copy of the recording, have the apprentice delay just the audio by one second. The hand on screen still lifts the ring twice. Listen to the audio on its own and we still count two knocks. Play them together, though, and the ring might touch the plate in silence. A clack might come after the hand has withdrawn. A system could summarize each track correctly on its own and still miss that difference if all it did was put the two summaries side by side.

A model that works with several kinds of information together is called **multimodal**. Here’s one way such a model might handle our clip. The part that processes images uses a group of numbers to represent the ring approaching the plate across successive frames. The part that processes sound uses numbers to represent the sudden onset of the impact and the vibration that follows it over a short stretch of time. At the sample rate we saw in Chapter 10, many audio samples can fit into the duration of a single video frame. To relate what the two parts pick up, it helps to keep track of *which stretch of the recording* each set of numbers came from.

What could teach the model to pay attention to that relationship? Imagine a training exercise. The apprentice leaves the audio in place for some examples and shifts it for others, as in our experiment. Each time, the model has to predict whether the sound and the movement are in sync. Both versions have the same door, the same hand, and the same ringing. Recognizing the door or counting the knocks would do little to tell them apart. A prediction that takes account of when the impact begins relative to the ring’s return to the plate could do better. The apprentice knows which clips were shifted,

so the error can be measured. The learning idea from Chapter 1 is at work again. There we adjusted weights using the error in a predicted order time. Here the error comes from predicting whether two events line up in time.

How can a model learn to recognize a door opening and generate the same movement?

The machinist points to the gap on screen. “Could a model still learn to tell that the door is opening if we turned the sound off completely?” The apprentice makes two short clips. The first contains just the last part of the recording, where the door moves. The second plays that part backward. Both clips show the same green paint, the same knocker, the same light. In one the door opens. In the other it closes.

For training, let’s label each clip with the action it shows. If the model answers “opening” for both, it will get one wrong. Imagine a calculation that relates the doorframe to the edge of the door across several successive frames. The strip of light between them goes from narrow to wide in one clip and from wide to narrow in the other. Taking that difference into account can reduce the prediction error. As the weights are adjusted in small steps during training, the model can learn to use clues like this one. To test it, the apprentice films a different door from another angle. Even if the strip of light is hidden in the new shot, the turn of the door leaf relative to the frame may still be visible. If the model struggles here, the apprentice has a reason to investigate what its success on the earlier examples depended on.

Now the apprentice shows the machinist the first frame, with the door shut. “What if I give it this picture and ask, ‘Have the knocker strike twice, then let the door open’?” The answer to this request will run for four seconds. The ring has to rise again between the first impact and the second. The door has to stay against its frame through both knocks and start moving only afterward. Every individual picture could be beautifully drawn, yet if the door opens at the first knock, the clip misses the prompt.

Let’s apply the generation method from Chapter 12. During training, the program takes the numbers representing the frames of a clip with a written description and adds random noise. It knows the values it added, so it can measure the error in the model’s noise prediction. Suppose the ring’s lower edge is hard to make out through the noise in one frame. Where might that edge lie? The frames on ei-

ther side give different clues depending on whether the ring is rising or falling back toward the plate. By drawing on those neighboring frames and the description, the model can reduce its prediction error.

To generate the new clip, the apprentice supplies the starting photo and the prompt. For the remaining moments, the system starts with random numbers and brings the short sequence into focus step by step. The machinist asks for two takes. In one, the door opens immediately after the second knock. In the other, the ring swings a little before the door opens. Both can fit the prompt. The apprentice pauses the second clip. “If we could see your apron through that gap, would we write in the log that you opened the door?” In a generated film, the apron might be a detail that fits the story. To decide what to put in the maintenance log, we need to go back to the original recording on the phone.

What’s the difference between “after” and “because”?

Suppose the apprentice gives the original recording to a model that can process sound and picture together. It answers:

> Hearing the two knocks, the machinist opens the door.

The machinist holds the pen poised over the log. “At what point in the recording could you tell I’d heard them?”

Let’s go through the sentence. For the two knocks, we can point to the movement of the ring and play the sounds of the impacts. To see the door opening afterward, we can go to the final frames. But where could we pause to show that the machinist heard the knocks? There’s no reply from inside, either. We never see the person who opens the door. With the shot framed this tightly, we can say very little about the person whose hand we see outside.

Why does the model’s sentence sound so natural? Someone knocks at a workshop door and the machinist inside answers it. It’s a familiar story. The model’s guess might even be right this time. Try a small test. Suppose someone else inside happened to be walking toward the door just then and opened it to go out. Which sound or picture on the phone would *have* to change in those four seconds? If both stories fit the same recording, we’d need more information to choose between them. A second camera facing into the workshop might show who opened the door. To find out whether that person heard the knocks, we’d do better to ask them.

In the end, the apprentice writes this in the log:

> The knocker struck twice. Shortly after the second knock, the door opened a little. The person who opened it was out of shot.

The machinist turns back to the first page of the notebook. Ten parts, 25 minutes. Twenty parts, 45 minutes. That day the apprentice had worked out the difference and allowed five minutes per order for setup. “Did setup really take five minutes?” the machinist asks. The apprentice looks again at the old entries. All the notebook gives is the total time for each job.

Just then, a new order arrives at the door. As the machinist steps up to the machine, the apprentice checks the clock. This time there’s room in the notebook for two times. The apprentice will enter the first when the machinist starts setting up, and the second when work begins on the first part.

Looking Back and Closing Reflections

From two orders in the machinist’s notebook, we worked out a setup time of five minutes. The calculation fit both entries. When we returned to that page at the end of the book, we noticed what was missing. The machinist had never recorded when setup ended. It was our assumption that setup accounted for those five minutes. Understanding how a model learns starts with noticing a distinction like this one.

During training, the model makes a prediction. We choose a measure for judging it and adjust the model’s learnable numbers according to that measure. For the orders, we used the difference between the estimated time and the measured time. For the language model, we knew how the unfinished sentence continued in the text. The less probability the model gave that continuation, the greater the loss. In both cases, the question we asked helped determine what relationship could be learned, just as the examples did.

When we used a trained model, we saw a different kind of change. Removing the sentence that said where the machinist was going changed the answer we could give to “Why did the machinist lock the door?” No further training was needed. What changed was the information available for answering. Two questions about the workshop door brought this out especially clearly. When investigating why the door leaf had been replaced, we checked whether the maintenance entry referred to the right entrance. When asking “Is the door locked right now?”, even after finding the right device we needed the time of its reading.

In the chapters on images, sound, and video, we saw how a model can draw on what it has learned to produce new material. Many different pictures would fit the prompt “Put a brass knocker on the green door.” If the notch on the machinist’s knocker matters, we can open the photograph and compare that exact spot. Try a similar experiment with sound. Two people saying “The door is open” can produce identical transcripts. Read the line aloud again and the words can survive. To hear where the original speaker paused, we need the recording. In generating a picture or a voice, the system gives form to details the prompt left open.

Pick an example from your own work and try changing just one thing in the input. Write down how you'd expect the answer to change. When you play the door clip backward, for instance, opening becomes closing. The color of the door stays the same. Now look at the model's answer. You've turned a broad question like "Did it understand the video?" into two questions you can check with your own eyes.

The apprentice did something similar on the last page. It would have been easy to write a longer explanation for the next order. Instead, the apprentice decided to measure the part of the old calculation that still raised a question. The next time a detail in a model's answer gives you pause, you might ask yourself the same thing. *What would I have needed to record in my notebook to find out?*

Glossary

Each entry points to the chapter or chapters where the term appears. Turn to the chapter listed if you'd like to revisit the definition through its workshop example.

A

Activation function (Chapter 1) — A rule that transforms the value computed in a layer before passing it on. ReLU, for example, turns negative values into zero.

Attention (Chapters 3 and 7) — The mechanism that determines how much weight information from different positions receives in the calculation at the current position.

B

Backpropagation (Chapter 1) — A method that uses the chain rule to work backward from the loss and calculate how sensitive it is to parameters in earlier layers.

C

Context (Chapters 3–6) — The inputs a model can draw on when producing an answer. These can include the question, earlier messages, and retrieved records.

D

Diffusion (Chapters 8 and 12) — In this book's examples, a way of generating an image or video by applying corrections step by step to a noisy representation.

E

Embedding (Chapters 3 and 6) — A learned numerical representation of an item such as a token or a passage of text.

F

Frame (Chapter 12) — An individual picture from a particular instant in a video. Frame order carries information about the direction of motion.

Frequency (Chapter 10) — The number of complete cycles of a vibration per second, measured in hertz.

L

Label (Chapter 10) — The target answer supplied for a training example. A recording of a knock might have the label “knocker.”

Latent representation (Chapter 8) — An intermediate numerical representation that encodes features of an input such as an image.

Loss (Chapters 1–2) — A measure that gives a numerical score to the mismatch between a prediction and its target during training.

M

Mask (Chapter 9) — A selection that specifies which positions in an image take part in an edit.

Multimodal (Chapter 13) — Describes a system that can work with more than one kind of information, such as text, sound, and images.

P

Parameter (Chapter 1) — A number in a model that can be adjusted during training. Weights are one example.

Pixel (Chapters 7–9) — An element that holds the color or brightness at one position in a digital image.

Prosody (Chapter 11) — The patterns of stress, duration, rhythm, and pitch in speech.

R

RAG (*retrieval-augmented generation*, Chapter 6) — An approach that retrieves relevant documents or passages and supplies them as context to the model generating an answer.

S

Sample rate (*sampling rate*, Chapter 10) — The number of measurements taken per second in an audio recording.

Seed (Chapter 8) — A value that sets the starting state of a pseudo-random number generator.

Softmax (Chapter 3) — An operation that converts raw scores into positive shares that add up to one.

Spectrogram (Chapters 10–11) — A grid showing the strength of different frequencies over successive time intervals.

T

Token (Chapters 2–4) — A unit of text processed by a language model. It can represent a word, part of a word, or a punctuation mark.

Tool call (Chapter 5) — A request a model produces for an application to run a known operation with specified inputs. In the workshop, the application reads the lock and returns the result to the model.

Training (Chapters 1–3) — The process of adjusting a model's learnable parameters using errors measured on examples.

V

Vector (Chapters 3 and 7) — An ordered list of numbers. It can represent features of a token or an image patch, for example.

Vocoder (Chapter 11) — In the book's speech synthesis example, the component that turns an acoustic representation into a waveform that can be played through a speaker.

W

Weight (Chapter 1) — A learnable number that adjusts an input's contribution to a calculation. During training it can be updated in response to prediction error.