Introductory Lecture (use MS Sway)

PSV Nataraj

Topics

- Classifier
- ML and DL – briefly
- Samples, features, labels
- Computerised learning strategy
- Generalization, Validation, Training, Retraining
- Learning – good and bad news
- Parameters and hyper-parameters
- ML - categories
    - Supervised learning: Classification and regression
    - Unsupervised Learning – clustering, noise reduction, dim reduction
    - Semi-Supervised learning - Generators
- Reinforced Learning
- ANN - Interpretation (via bank loan example)

What's a classifier ?

- A classifier assigns **a label** to each sample describing which category or class, that sample belongs to.

Example of Classifiers

- If the input is a song, classifier assigns the label as the genre (e.g., rock or classical).
- If it's a photo of an animal, the classifier assigns the label as the name of the animal shown (e.g., a tiger or an elephant).
- In mountain weather for hiking, classifier may label the hiking experience into 3 categories: Lousy, Good, and Great.

Machine Learning Systems

- Beauty of **machine learning systems** is they learn a dataset's **characteristics *automatically***.
- We don't have to tell an algorithm how to recognize a 2 or a 7, because system figures that out for itself.
- But to do that well, ML system often needs a ***lot*** of data. Enormous amounts of data.
- That's a big reason **why machine learning has exploded in popularity and applications in the last few years**.
- The flood of data provided by the Internet has let these tools extract **a lot of meaning from a lot of data.**

Simple example of ML Systems

- **Example**: Online companies (Amazon, Flipkart etc.) make use of every interaction with every customer to accumulate more data.
- They use it as input to machine learning algorithms, getting more information about customers.

Learning from Labeled Data

There are lots of machine learning algorithms. Many are straightforward.

Example: Find the best straight line through a bunch of data points, see Figure 1.5.

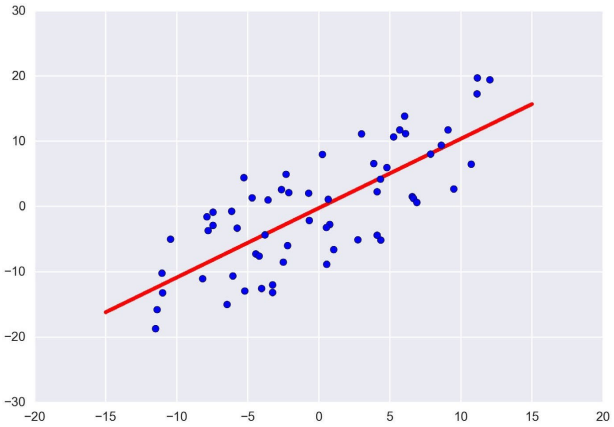


Figure 1.5: Given a set of data points (in blue), we can imagine a straightforward algorithm that computes the best straight line (in red) through those points.

ML- Learning from Labeled Data (Contd,)

- **ML:** Represent a straight line with just a few numbers.
- Use some formula to compute those numbers, given the input data points.
- This is a familiar algorithm, uses analysis to find the best way to solve a problem
- Its implemented in a program that performs that analysis.
- This is a strategy used by many machine learning algorithms.

DL- Learning from Labeled Data (Contd)

- **DL:** By contrast, the strategy used by many deep learning algorithms is less familiar.
- It involves **slowly learning** from examples, a little bit at a time, over and over.
- Each time the program sees a new piece of data, it improves its own parameters.
- Ultimately, it finds a set of values that will do a good job of computing what we want.
- Much more open-ended than the one that fits a straight line.

DL- Learning from Labeled Data (Contd)

- Problem: Don't know how to directly calculate the right answer,
- Solution using DL: Build a system that can figure out how to do that itself.
- Idea: Create an algorithm that can **work out** its own answers, rather than implementing a known procedure/method that directly yields an answer.
- Recent enormous success of **deep learning** algorithms is due to what?
- Due to programs that find their own answers in this way.

Example of Samples, Features, Labels

Weather measurements on a mountain for hiking

- Sample is weather at a given moment.
- Features are measurements: temperature, wind speed, humidity, etc.
- Hand over each sample (with a value for each feature) to a human expert.
- Expert examines features and provides a **label** for that sample.
- Expert's opinion, using a score from 0 to 100, tells how the day's weather would be for **good hiking**.
- Labels can be "Lousy", "Good", "Excellent" (weather for hiking)
- The idea is shown in Figure 1.7.

Example of Samples, Features, Labels (Contd.)

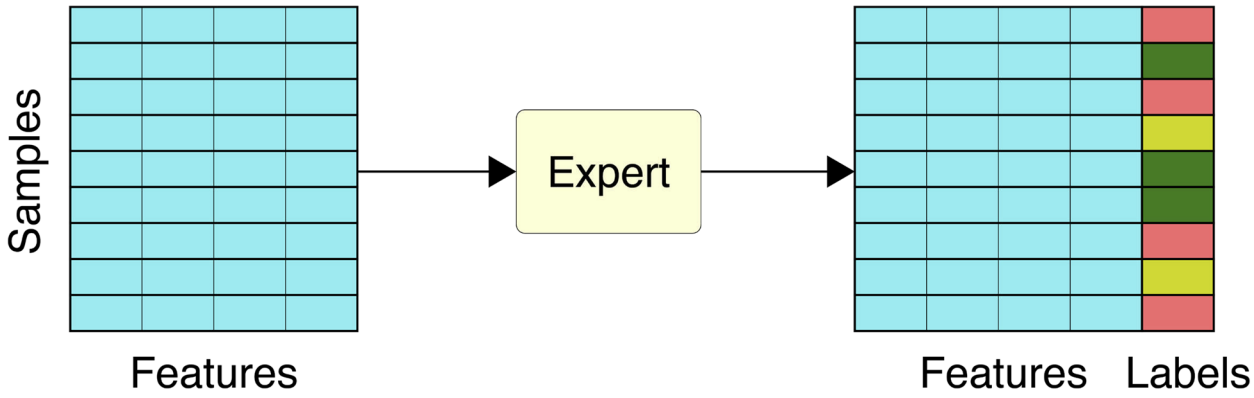


Figure 1.7: To label a dataset, we start with a list of samples, or data items. Each sample is made up of a list of features that describe it. We give the dataset to a human expert, who examines the features of each sample one by one, and assigns a label for that sample.

A Computerized Learning Strategy

- First, collect as much data as possible.
- Call each piece of observed data (say, the weather at a given moment) as **sample**,
- Call the names of the measurements that make it up (the temperature, wind speed, humidity, etc.) as **features.**
- Hand over each sample (with a value for each feature) to a human expert.
- Expert examines features and provides a **label** for that sample.
- Example: if our sample is a **photo**, the label might be the **name of the person** or the type of **animal in the photo.**

A Computerized Learning Strategy (Contd.)

Figure 1.8 shows the idea of a learning strategy

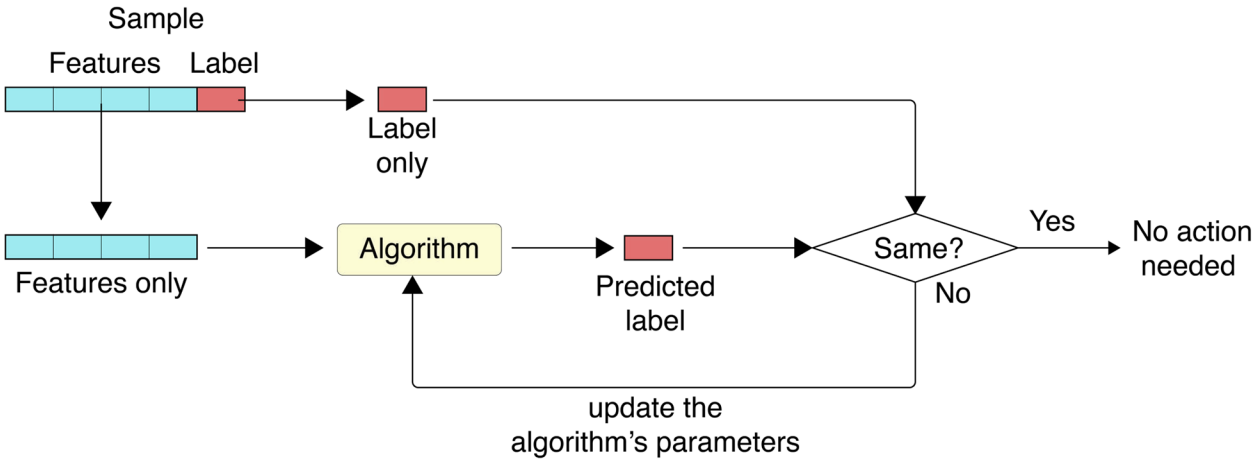


Figure 1.8: One step of training, or learning.

Split the sample's features and its label.

From the features, the algorithm predicts a label.

Compare the prediction with the real label.

If the predicted label matches the label we want, we don't do a thing.

Otherwise, we tell the algorithm to modify, or update, itself so it's less likely to make this mistake again. The process is basically **trial and error**.

Notes on Computerized Learning Strategy

- First, set aside some of these labeled samples for time being (use them later for validation).
- Give remaining labeled data to our computer, and ask it to find a way to come up with the right label for each input.
- We do ***not*** tell it how to do this.
- Instead, we give labelled data to an algorithm with a large number of parameters it can adjust (perhaps even millions of them).
- Different types of learning will use different algorithms.

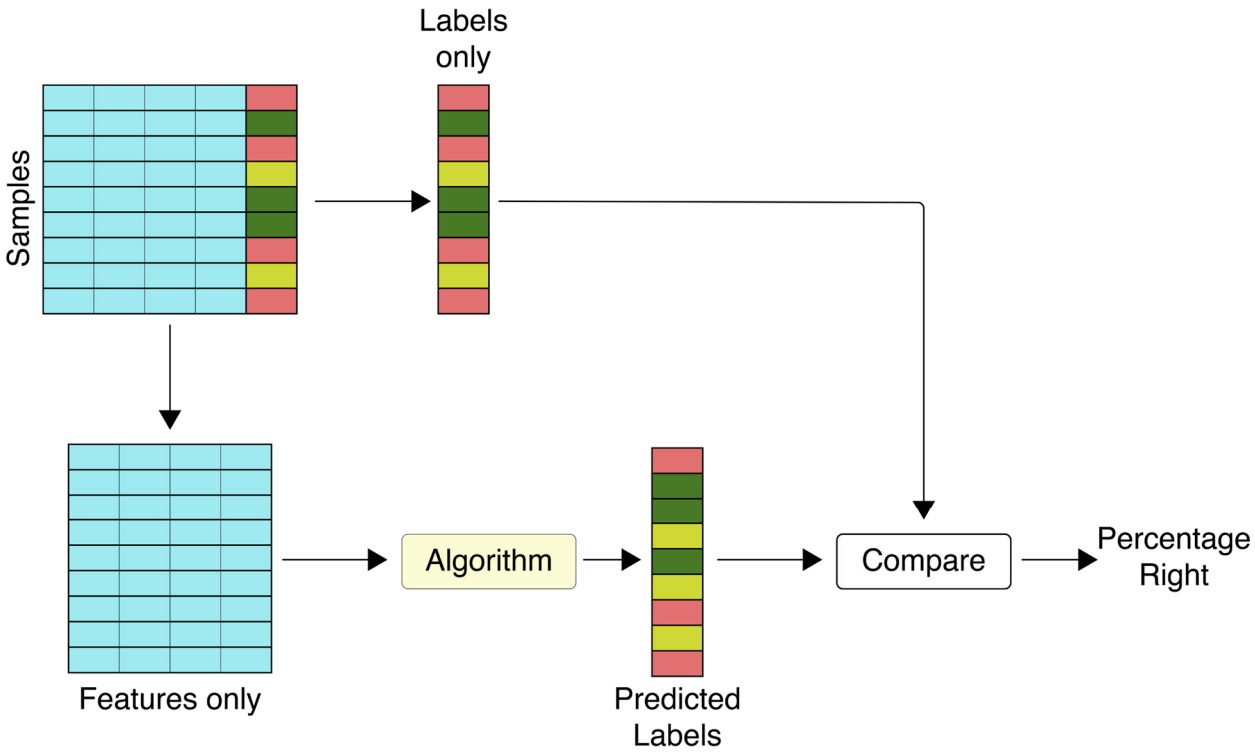
Notes on Computerized Learning Strategy (Contd.)

- Each algorithm **learns by changing** the **internal parameters** it uses to create its predictions.
- **Big change:** risk of changing them so much that it makes **other predictions worse**.
- Small change: Cause learning to run slower.
- We have to find by trial and error for each type of algorithm the **right trade-off between these extremes.**
- We call the amount of updating the **learning rate**,
- A **small** learning rate **is cautious and slow**,
- A **large** learning rate speeds things up but could **backfire**.

Generalization (or Evaluation/Validation)

- We now return to the labeled data **kept aside** in the last section.
- This is called as test data.
- We evaluate how the system can **generalize** what it learned, by showing these samples that it's ***never seen*** before.
- This **test set** shows how the system performs on **new data**.

Generalization or Evaluation - Procedure



- Figure 1.9: The overall process for evaluating a classifier (also called a categorizer).

Generalization - Procedure (Contd.)

- In Figure 1.9, we again split the **test data** (not training data) into features and labels.
- The algorithm assigns, or predicts, a label for each set of features.
- We then compare the predictions with the real labels to get a measurement of accuracy.
- If it's good enough, we deploy the system.
- If the computer's predictions on these brand-new samples are not a sufficiently close match to the expert's labels, then we return to the training step in Figure 1.8. That is, if the results aren't good enough, we go back and train some more.
- Note that unlike training, in this process there is no feedback and no learning.
- Until we return to explicit training, the algorithm doesn't change its parameters, regardless of the quality of its predictions.

Re-training

- Use again **original training set** data. Note that these are the same samples.

    - **Shuffle** this data first - but no new information.
    - Show every sample again, letting it learn along the way again.
    - Computer learns over and over again from the very same data.
- Now, show test data set .
    - Ask algorithm to predict labels for the test set again.
- If the performance isn't good enough, go back to original training set again, and then test again.
- Around and around we go, repeating this process often hundreds of times. letting it learn just a little more each time.
- Computer doesn't get bored or cranky seeing the same data over and over.
- It just learns what it can and gets a little bit better each time it gets another shot at learning from the data.

Learning – Good and Bad News

Bad News:

- No guarantee that there's a successful learning algorithm for every set of data,
- No guarantee that if there is one, we'll find it.
- May not have enough computational resources to find the relationship between the samples and their labels.

Good news:

Even without a mathematical guarantee, in practice we can often find solutions that generalize very well, sometimes doing even better than human experts.

Parameters and hyperparameters

Learning algorithm modifies itself its own parameter values, over time.

Learning algorithm are also controlled by values that we set (such as the learning rate we saw above).

These are called **hyperparameters**.

What's the difference between between parameters and hyperparameters ?

Computer adjusts its own parameter values during the learning process, while **we specify the hyperparameters** when we write and run our program.

When do we deploy the System ?

When the algorithm has learned enough to perform well enough on the test set that we're satisfied, we're ready to **deploy**, or **release**, our algorithm to the world.

Users submit data and our system returns the label it predicts.

That's how pictures of faces are turned into names, sounds are turned into words, and weather measurements are turned into forecasts.

Machine Learning – major categories

- Let's now get a big picture for the field of machine learning.
- See the major categories that make up the majority of today's ML tools.

Supervised Learning (ML)

- Supervised learning (SL) is done for samples with pre-assigned labels.
- Supervision comes from the labels.
- Labels guide the comparison step
- See Figure 1.8, where the algorithm is told if it predicted the correct label or not.
- There are two general types of supervised learning, called classification and regression.

Two Types of SL

Classification: look through a given collection of categories to find the one that best describes a particular input.

Regression: take a set of measurements and **predict some other value**

SL – Classification

- Start training by providing a list of all the labels (or classes, or categories) that we want it to learn.
- Make the list so that it has all the labels for all the samples in the training set, with the duplicates removed.
- Train the system with lots of photos and their labels, until it does a good job of predicting the correct label for each photo.
- Now, turn the system loose on new photos it hasn't seen before.

- For those objects it **saw during training,** It should **properly label images**.
- Caution: For those objects it **did not see during training,** the system will try to pick the best category from those it knows about.
- Figure 1.10 shows the idea.

SL – Classification Example

- **Example**: Sort and label photos of everyday objects.
- We want to sort them : an apple peeler, a salamander, a piano, and so on.
- We want to **classify** or **categorize** these photos.
- The process is called **classification** or **categorization**.

SL – Classification (example contd.)

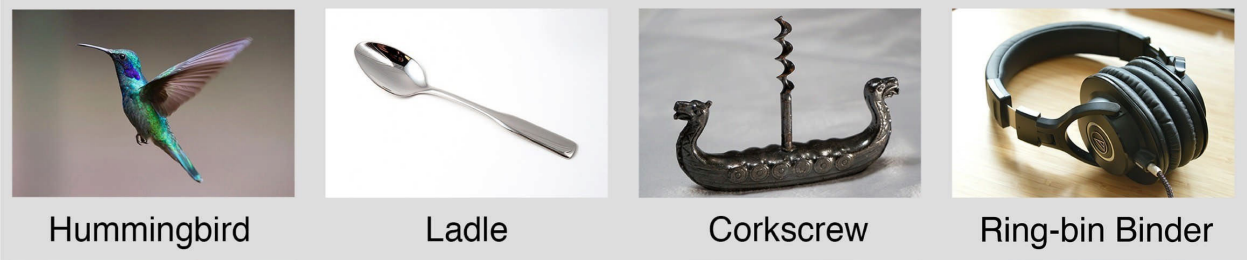


Figure 1.10: When we perform classification, we train a classifier with a set of images, each with an associated label.

After training, we give it new images, and it will try to choose the best label for each image.

SL – Classification (example contd.)

- In Figure 1.10, we used a trained classifier to identify four images never seen before.
- The system had not been trained on metal spoons or headphones,
  - in both cases it found the best match it could.
- To correctly identify those objects, the system needs to see multiple examples of them during training.
- Another way to say: System can only understand what it has learned.
- Traditional classifiers will always do their best to find the closest match for every input, But they can only use the categories that they know about.

SL- Regression (Example Music band attendance)

Problem: we have an incomplete collection of measurements of attendance.

We want to estimate the missing values of attendance.

Example (Music band):

Data: **A**ttendance at a series of concerts at a local arena.

The band gets paid each night, receiving a fraction of the total ticket sales for that concert.

Problem: Unfortunately, we **lost** the **count** for one evening's performance.

We also want to know what tomorrow's attendance is likely to be.

SL Regression (Example Contd.)

- The measurements we have are shown in the left illustration of Figure 1.11.
- Our estimates for the missing values are shown on the right of Figure 1.11.

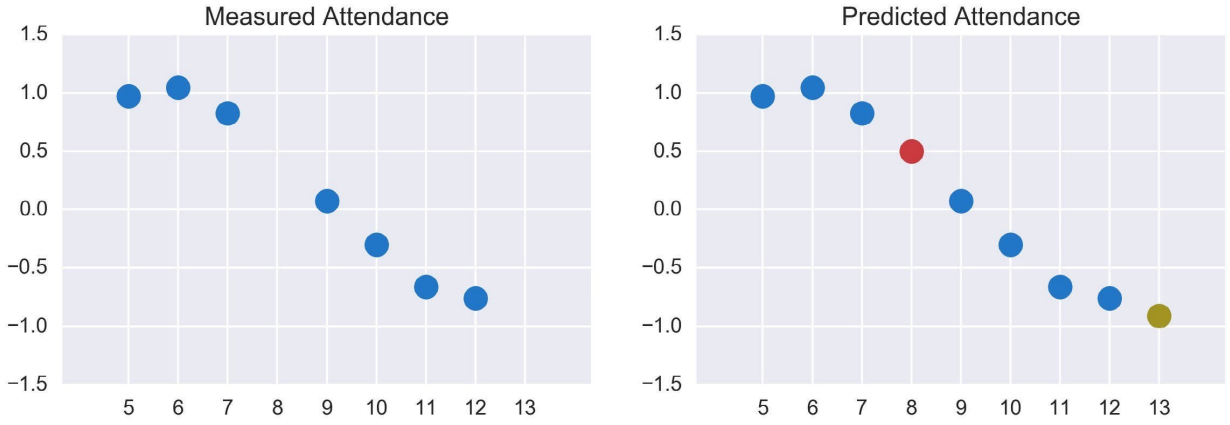


Figure 1.11: In regression, we work with sets of input and output values. Here, the input is the date of a concert from May 5 to May 13, and the output is the attendance.

Left: Our measured data, which is missing a value for May 8th.

Right: The red dot is the estimated value of the missing point at May 8, and the yellow dot is a prediction of the attendance on May 13.

SL Regression

- This process of filling in or predicting data is **regression**.
- "Regression" uses statistical properties of the data to estimate missing or future values.
- The most famous kind of regression is **linear regression**.

- We match our input data with a **straight line**, as in Figure 1.12.

SL- Linear and Nonlinear Regression

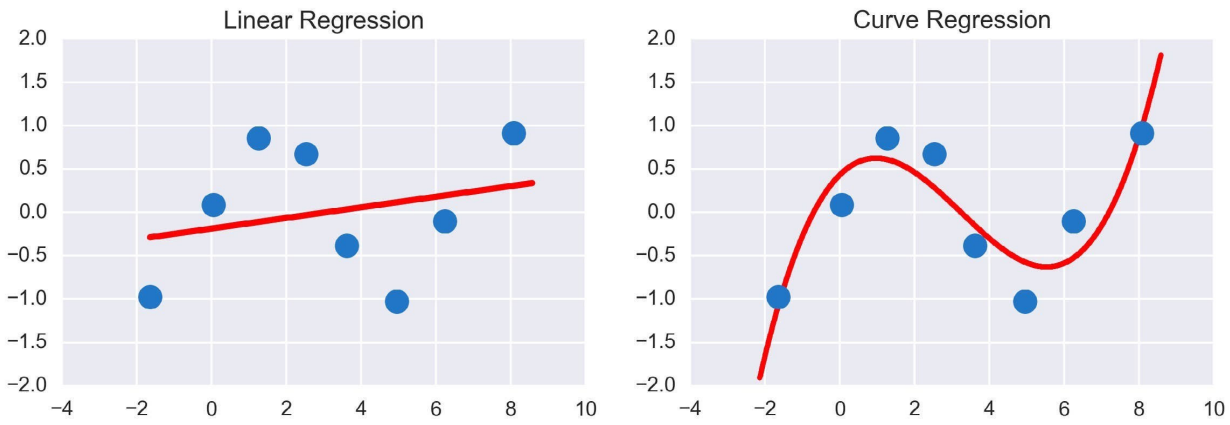


Figure 1.12: Representing points of data with mathematical shapes.

Left: **Linear** regression fits a straight line (red) to the data (blue). The line is not a very good match to the data, but it has the benefit of being simple.

Right: A more complex **nonlinear** regression fits a curve to the same data. This is a better match to the data, but it has a more complicated form and requires more work (and thus more time) when part of a calculation.

SL- Linear and Nonlinear Regression (Contd.)

- We can use more **complex forms of regression** to create more complicated types of curves, as in the right of Figure 1.12.
- These provide a **better fit** to the data.
- Q: Why not go for complex regression?
  - **More computation time is needed**.
  - More data is also needed

Unsupervised Learning (USL) – a form of ML

- What's USL?
  - When input data does not have labels, any algorithm that learns from the data belongs to USL.
  - We are not "supervising" the learning process by offering labels.

  - The system has to figure everything out on its own, with no help from us.
- USL used for **clustering**, **noise reduction**, and **dimension reduction**.
- Let's look at these in turn.

USL for Clustering - Via Pottery Example

Example: (Pottery Markings)

Suppose that we're digging out the foundation for a new house.

Surprise! we find the ground is filled with old clay pots and vases.

Call an archaeologist, who says its a jumbled collection of ancient pottery, from many different places and different times.

The archaeologist doesn't recognize any of the markings and decorations, so she can't declare for sure where each one came from.

Some of the marks look like variations on the same theme, while others look like different symbols.

USL for Clustering - Via Pottery Example (Contd.)

She takes rubbings of the markings, and then tries to sort them into groups.

But there are far too many of them for her to manage.

She turns to a machine learning algorithm.

Why ML?

to automatically group the markings together in a **sensible** way.

USL for Clustering - Via Pottery Example (Contd.)

Figure 1.13 shows her captured marks, and the **groupings that could be found automatically** by an algorithm.

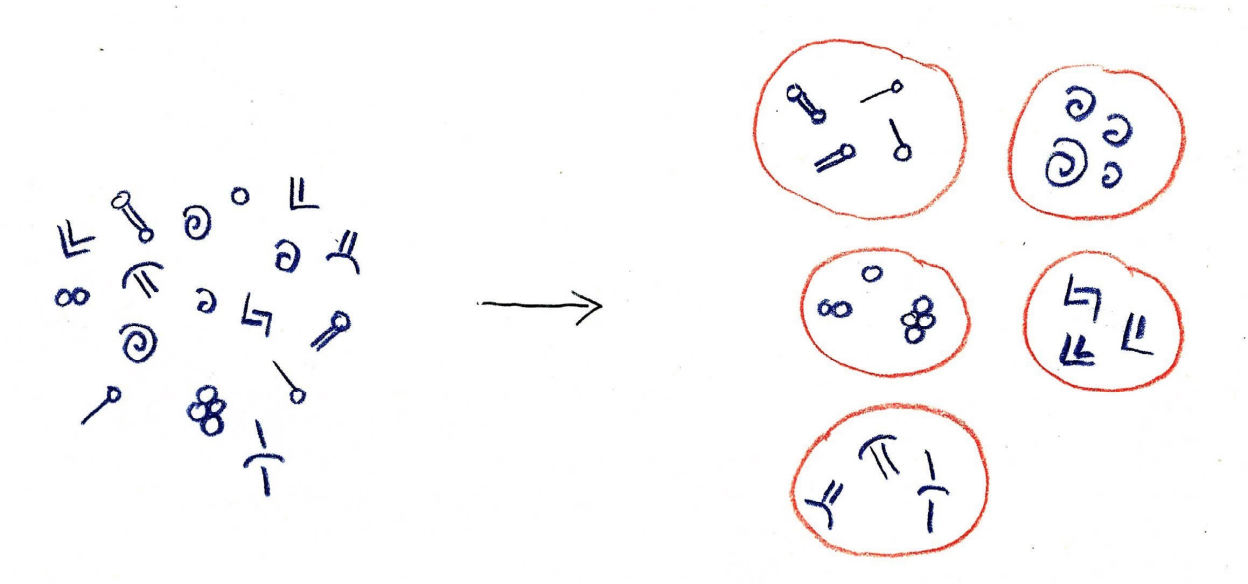


Figure 1.13: Using a clustering algorithm to organize marks on clay pots. Left: The markings from the pots. Right: The marks grouped into similar clusters.

USL for Clustering

- This is a **clustering** problem
- The ML algorithm is a **clustering algorithm**.
- There are many clustering algorithms to choose from.
- **Because our inputs are unlabeled, this archaeologist is performing** clustering, **using an** unsupervised learning **algorithm**.

USL for Noise Reduction – Noisy Image Example

- Sometimes samples are **corrupted** by **noise**.
- Figure 1.14 shows an example of a noisy image, and how a de-noising algorithm might clean it up.

USL for Noise Reduction – Noisy Image Example (Contd.)

Figure 1.14: De-noising, or noise reduction, reduces the effects of random distortions to samples, and can even fill in some missing gaps.

Left: An image of a cow degraded with a lot of noisy pixels, and some missing pixels. Right: The original image. De-noising algorithms can recover this image with different degrees of success.

USL for Noise Reduction

De-noising is a form of unsupervised learning.

Why?

As we don't have labels for our data (for example, in a noisy photo we just have pixels)

The algorithm estimates what part of sample is noise and then removes it.

Important Tip: Apply de-noising algorithms to data before learning from it.

Why? By removing weird **and** missing **values from the input, learning process happens more quickly and smoothly.**

USL for Dimensionality Reduction

- Problem: Sometimes our samples have more features than they need.
- So, simplify the data:
  - Remove uninformative features,
  - Combine redundant features, or
  - Trade off some accuracy for simplicity by combining or otherwise manipulating features.
- For all these tasks, there are unsupervised learning algorithms that can do the job.
- Using unsupervised learning to find a way to reduce the number of values (or dimensions) of our data is called **dimension reduction**.

• The name describes that we're reducing the number of features (also called dimensions) of each sample

USL for Dimensionality Reduction (Contd.)

- We shall see several examples on this.
- In all of these cases, we remove features that aren't contributing to understanding the underlying information.
- By making data simpler, our learning system has to do less work to learn.
- This makes training faster, and perhaps even more accurate.

USL for Dimensionality Reduction Example#1 Weather

- **Data**: Weather samples in the desert at the height of summer.
- Record daily wind speed, wind direction, and rainfall.
- Given the season and locale, the rainfall value will be 0 in every sample.
- If we use these samples in a machine learning system, the computer will need to process and interpret this useless, constant piece of information with every sample.
  - At best **this would** slow **down the analysis.**
  - At worst **it could affect the system's accuracy, because the computer would devote some of its finite resources of time and memory to trying to learn from this unchanging feature.**

USL for Dimensionality Reduction Example#2 Health Clinic

- Sometimes features contain **redundant data**.
- A health clinic might take everyone's weight in kilograms when they walk into the door. Then when a nurse takes them to an examination room, she measures their weight again but this time in pounds.
- Same information repeated twice, but it might be hard to recognize that because the values are different.
- Like the useless rainfall measurements, this redundancy will not work to our benefit

USL for Dimensionality Reduction Example#3 Tree Infection

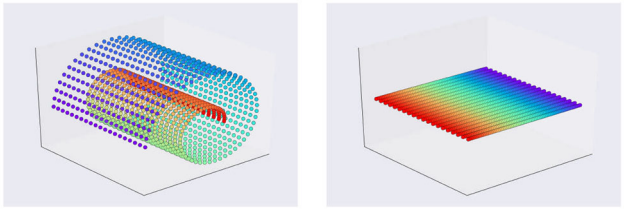


Figure 1.15: Measuring the infection of a tree trunk as it's turned and shaved.

Left: The original data is in 3D.

Right: We can make the problem easier by unrolling the data into 2D.

Now we need only 3 numbers for each data point: 2 for each point's location, and 1 for its value.

Processing is faster.

USL for Dimensionality Reduction Example#4 Highway Traffic

- Monitoring traffic on a curvy, one-way, one-lane road through the mountains.
- We want to make sure that the traffic is safe along one particular segment that we're concerned about.
- So set up some monitoring gear, and periodically take snapshots like Figure 1.16(a).

USL for Dimensionality Reduction Example#4 Highway Traffic (Contd.)

Fig shows where each car is located, which way it's headed, and how fast it's going along the stretch we want to watch.

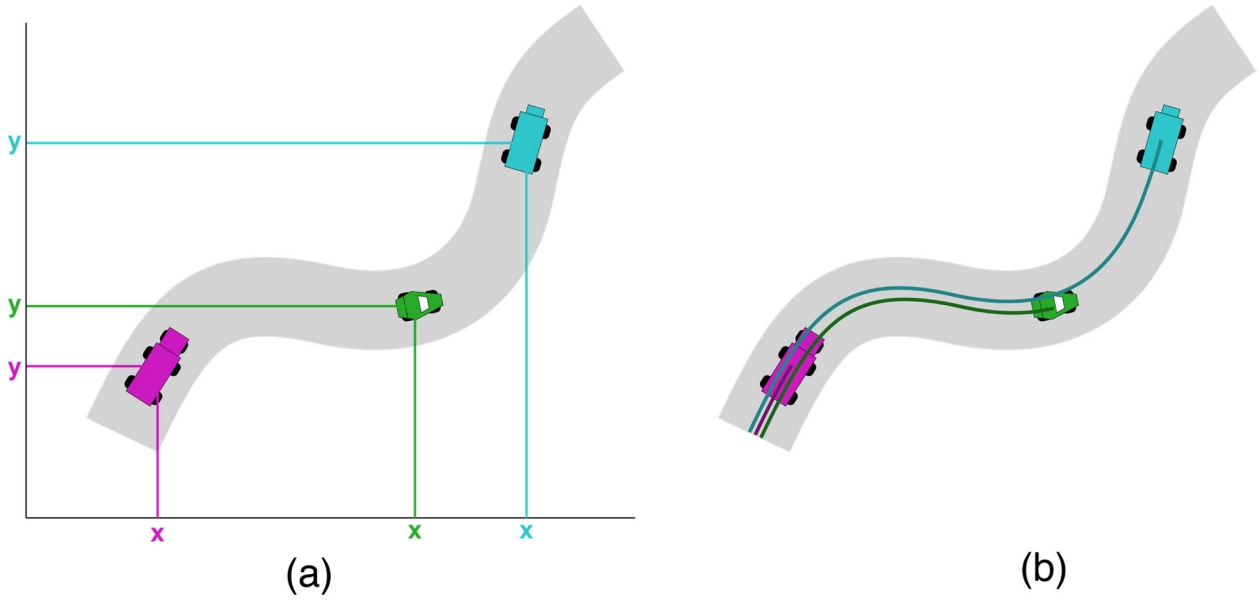


Figure 1.16: Where are the vehicles on this road?

(a) We can measure each car's position using latitude and longitude, requiring two numbers (here marked x and y).

(b) Alternatively, we can measure the distance that each vehicle has travelled along the road. This requires only one number, making our data simpler.

Semi-Supervised Learning (Generators) Example - Persian Carpets in Movie

Shooting a movie inside a Persian carpet warehouse.

Problem: Want hundreds of carpets, all over the warehouse.

Carpets on the floors, carpets on the walls, and carpets in great racks in the middle of the space.

Each carpet to look real, but be different from all the others.

Our budget is nowhere near big enough to buy, or even borrow, hundreds of unique carpets.

So instead, we buy just a few carpets, and then we give them to our props department to make many **fake** carpets.

Semi-Supervised Learning (Generators) Example - Persian Carpets in Movie

Figure 1.18 shows our starting carpet.

Figure 1.18: A Persian carpet that we'd like to generalize.

Semi-Supervised Learning (Generators) Example - Persian Carpets in Movie

Figure 1.19: Some new fake carpets based on the starting image of Figure 1.18.

Made by ML Algorithms.

Semi-Supervised Learning (Generators)

- This process of **data generation** is implemented by ML algorithms called **generators**.
- Train generators with large numbers of examples - so that they can produce new
- versions with lots of variation.
- We don't need labels to train generators, so its unsupervised learning techniques.

- But we do give generators some feedback as they're learning, so they know if they're making good enough fakes for us or not.
- So maybe we should put our generators into the category of supervised learning?
- A generator is in the middle ground. It doesn't have labels, but it is getting some feedback from us. We call this middle ground **semi-supervised learning**.

Reinforcement Learning via Example

- Suppose you are take care of a friend's three-year old daughter.
- You have no idea what the young girl likes to eat.
- First dinner: make Pasta with butter. She likes it!
  - Repeat this dinner for a week. She gets bored.
- Week 2: Add some cheese, and she likes it.
  - Repeat this dinner for week 2. She gets bored.
- Week 3: Try pesto sauce. But girl refuses to take a bite.
  - So make pasta with some marinara sauce instead, and she rejects that too.
  - Frustrated, you make a baked potato with cream. She likes it!
- Weeks 3 & 4 : Try one recipe and one variation after another, trying to develop a menu that the child will enjoy.
- The only feedback: Little girl eats the meal, or she doesn't.

Approach to learning is **Reinforcement Learning** !

Reinforcement Learning - More Examples

RL Example 2:

- Agent: Autonomous car
- Environment: Traffic/people on the street.
- Actions: Driving
- Feedback: Driving okay if following **traffic rules** and **keep everybody safe**.

RL Example 3:

- Agent: DJ at a dance club
- Environment : Dancers
- Feedback: **Like or dislike** the music.

Reinforcement Learning

- **Agent** makes decisions and takes actions (the chef).
- Environment is everything else in the universe (the child).
- Environment gives **feedback** or a **reward signal** to agent after **every action**
- Feedback tells how good or bad that action is.
- The reward signal is often just a single number, where larger positive numbers mean the action was considered better, while more negative numbers can be seen as punishments.
- The reward signal is not a label, nor a pointer to a specific kind of "correct answer."
- Figure 1.20 shows the idea of RL

Reinforcement Learning (Contd.)

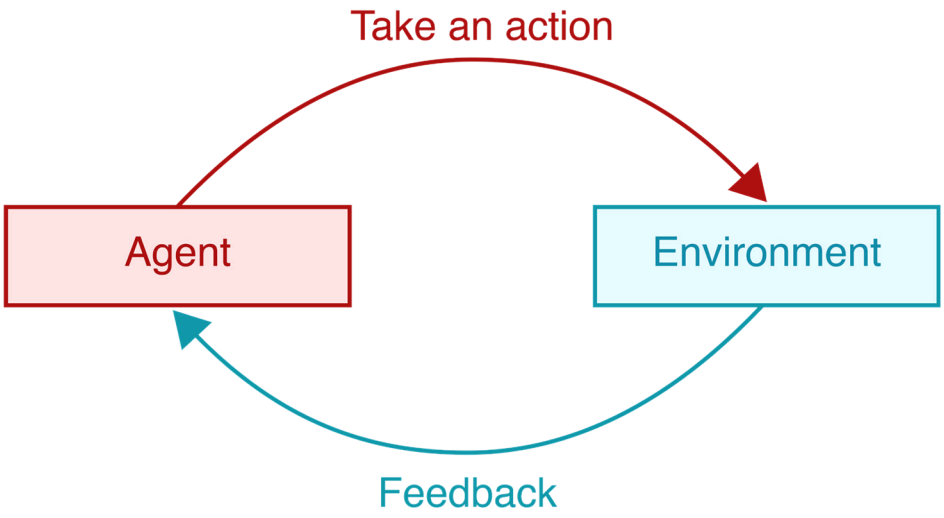


Figure 1.20: In reinforcement learning, we have an agent (who takes actions) and an environment (everything in the universe except the agent). The agent acts, and the environment responds by sending feedback in the form of a reward signal.

How's RL different from SL?

- The general plan of **learning from mistakes** is the same, **but the mechanism** is different.
- **Supervised learning**: system produces a result (typically a category or a predicted value), and then we **compare it to the correct result**, which **we provide**.
- **Reinforcement learning: T*here is no correct result*. The data has no label.**
- **There's just feedback** that tells us how **well** we're doing.
- Feedback tells that our action was **"good" or "bad."**

- In contrast to supervised learning algorithms, the reward signal is not a label and no idea/pointer to “correct answer.”

ANN Interpretation

What’s really going on in Neural Networks?

Can we explain why the network is producing the results it gives us?

Lets see an example.

Example

- Let’s consider the process of getting a loan.
- In practice, any loan application is going to be a complex affair.
- Based on the applications for those loans, they came up with rules that would let them predict whether a loan was likely to end up getting paid back or not.
- This is of course just like what a perceptron does. The inputs are weighted and combined to produce a final score, as in Figure 20.24.

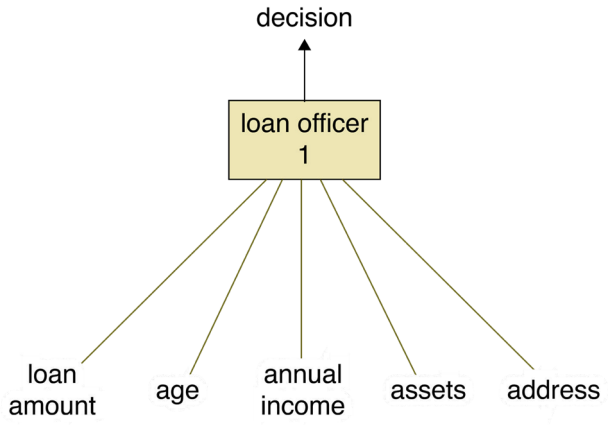


Figure: **To decide whether or not to issue a loan, a loan officer might take in the information on an application form, and weight the various pieces by different amounts to produce a final score.**

The value of that score determines whether or not the loan is granted.

- Say, loan is rejected
- We might try our luck at another bank.

- In this bank, Suppose that there are 5 different loan officers, and each one has developed his own idiosyncratic procedure for evaluating the criteria that go into a loan.
- We submit our application to the bank, and then all 5 officers to evaluate our application in turn.
- Maybe 3 of them say yes, and 2 say no. If it's a simple majority vote, then we'd get the loan.
- We can draw this as a two-layer neural network.
- The input is our application, followed by a layer with 5 neurons, one for each loan officer, each reading the input.
- Each officer's decision is presented at their output.
- Let's say +1 means that they would approve the loan, −1 means they would deny it, and intermediate scores represent more lukewarm conclusions. Figure 20.25 shows this interpretation.

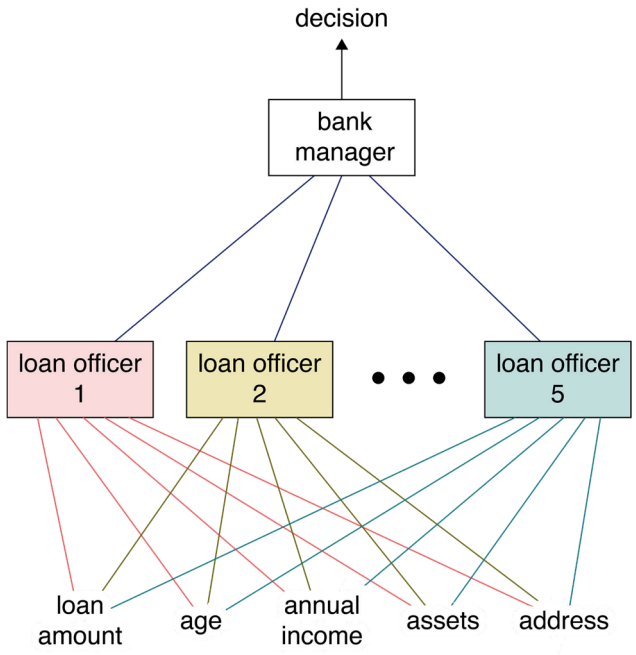


Figure 20.25: Our loan application is given to 5 different loan officers, each of which carries out his own distinct evaluation. Their scores are weighted and then added together by the bank manager to produce a final score, which determines if the loan is granted or not.

- Let's suppose that the bank is now issuing so many loans that the bank manager is getting overwhelmed.
- So he hires a bunch of supervisors to sit between the loan officers and himself.
- Suppose we have 5 loan officers, 8 supervisors, and 1 bank manager. Then we can build a 3-layer neural network to represent this process, as in Figure 20.26.

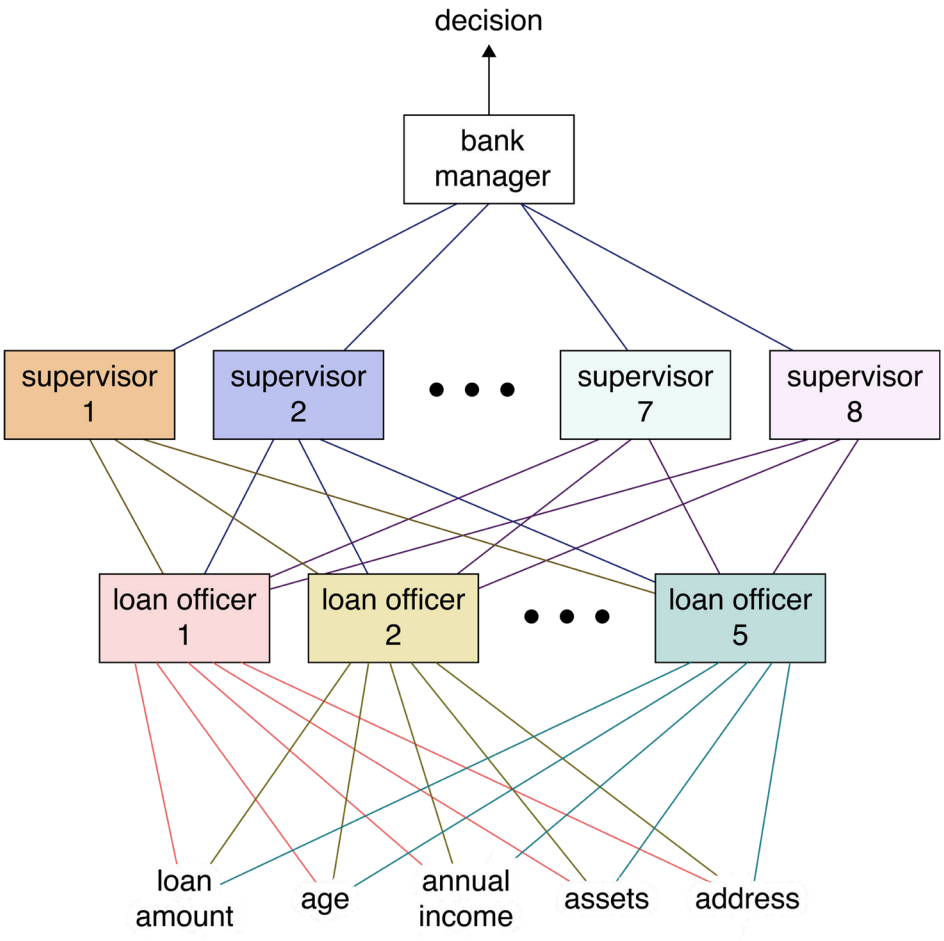


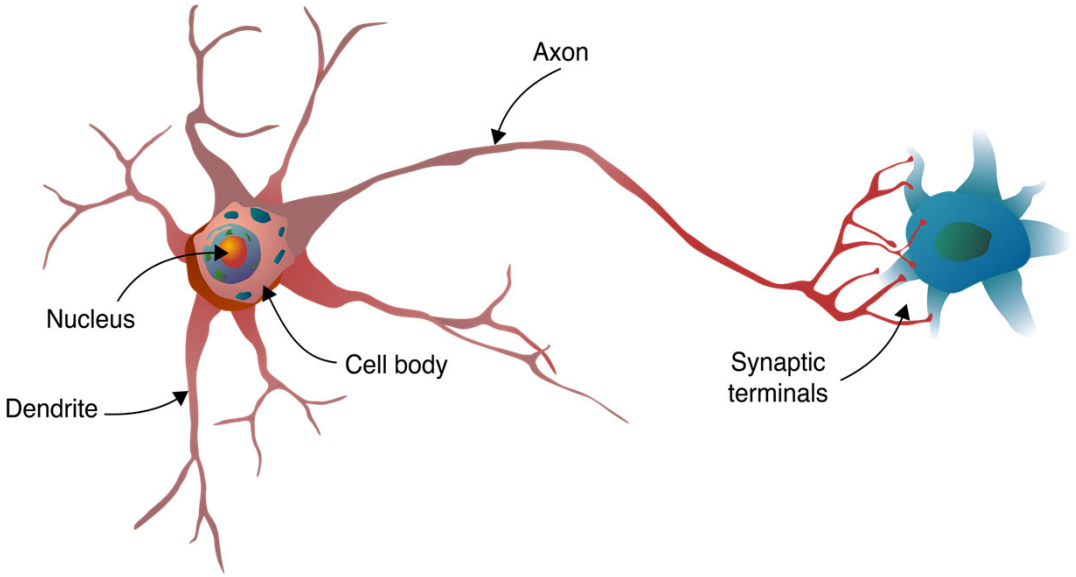
Figure 20.26: Each of our loan officers reports his evaluation of our application to a supervisor. In this example we have 5 loan officers and 8 supervisors. The supervisors report their decisions to the bank manager, who makes the final decision about the loan.

- Supervisors look at the decisions of the loan officers.
- Supervisors combine the results of the loan officers, and then pass their judgements up to the bank manager.
- Bank manager final decision is based on how he chooses to weight the conclusions of the supervisors.
- The bank can keep adding more and more layers of officers, each one reviewing the decisions of the previous layer and looking for statisti- cal patterns

Neurons
Feedforward nets
Activation functions
I/O layers

Neurons

How real biological neurons inspired the artificial neurons we use in machine learning, and how those little bits of processing work alone and in groups.



- Neurons are the nerve cells that make up the brain, used for our cognitive abilities.
- Neurons are information processing machines.

Figure 10.1: **A sketch of a biological neuron (in red) with a few major structures identified.**
This neuron's outputs are communicated to another neuron (in blue), only partially shown.

- Basis of DL algorithms: Building a **network** of connected computational elements.
- Fundamental unit of such networks: a small bundle of computation called an **artificial neuron**, or simply **neuron**.
- **Artificial neuron** was inspired by **human neurons.**
- In this part, we'll look very briefly at look at artificial neurons.

Important note: Not all machine learning algorithms use neurons.

Many ML techniques use traditional analysis and explicit equations, to find a solution. No neurons are used.

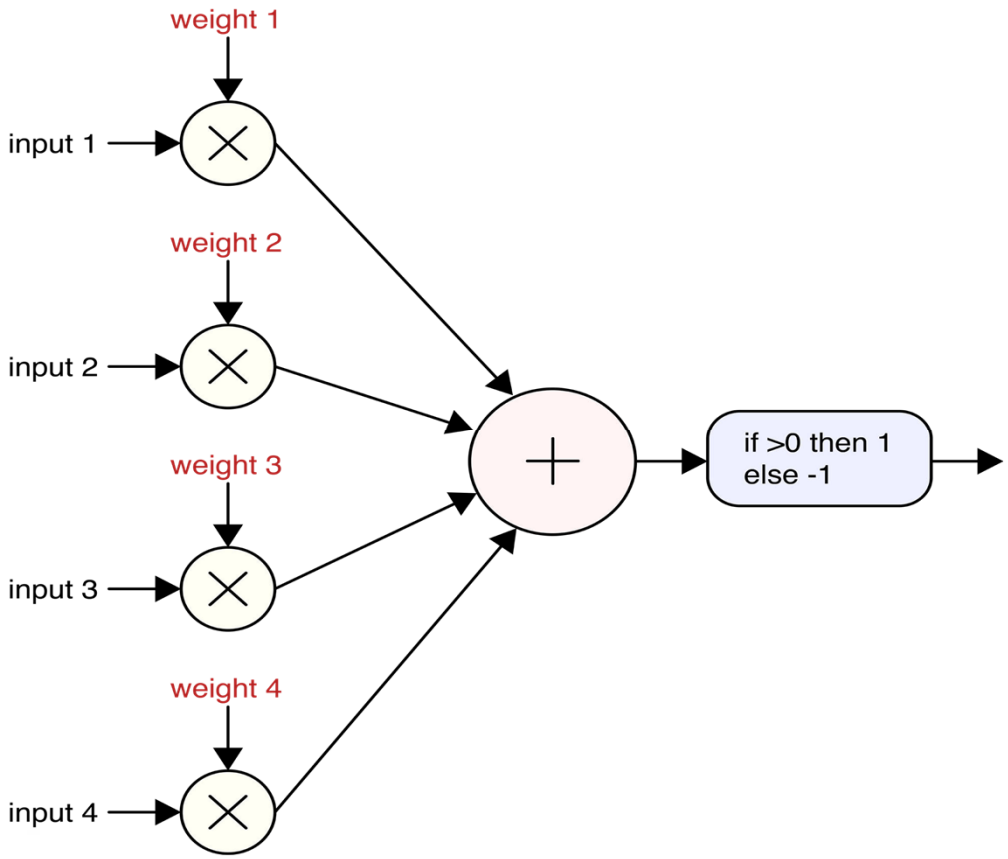
Neurons are in more general learning systems, such as **neural networks** that make up **deep learning** systems.

In those systems, artificial neurons are indispensable.

We should know about these artificial neurons so we can use modern deep learning systems.

Perceptron

- The history of artificial neurons began in 1943.
- The “neurons” we use in machine learning are inspired by real neurons.
- In 1957, the **perceptron** was proposed as a simplified mathematical model of a neuron.
- Figure 10.2 gives a block diagram of a single perceptron with 4 inputs.



Perceptron mimics the functional behavior of **neuron**.

Figure 10.2: **A schematic view of a perceptron. Each input is a single number, and it's multiplied by a corresponding real number called its weight. The results are all added together, and then tested against a threshold. If results are positive, perceptron outputs +1, else −1.**

Modern Artificial Neurons

- **Modern** Neurons are only slightly generalized from the **original** perceptrons.
- Still called "perceptrons" or "**neurons**"
- Two changes to original perceptron: one at the input, and one at the output.
- 1. Input - Provide each neuron with one more input, called **bias**. It's a number that is directly added into the sum of all the weighted inputs. Each neuron has its own bias.
- 2. Output- Replace threshold with an activation function, i.e., a mathematical **function** that takes the sum (including the bias) as input and returns a new floating-point value as output.

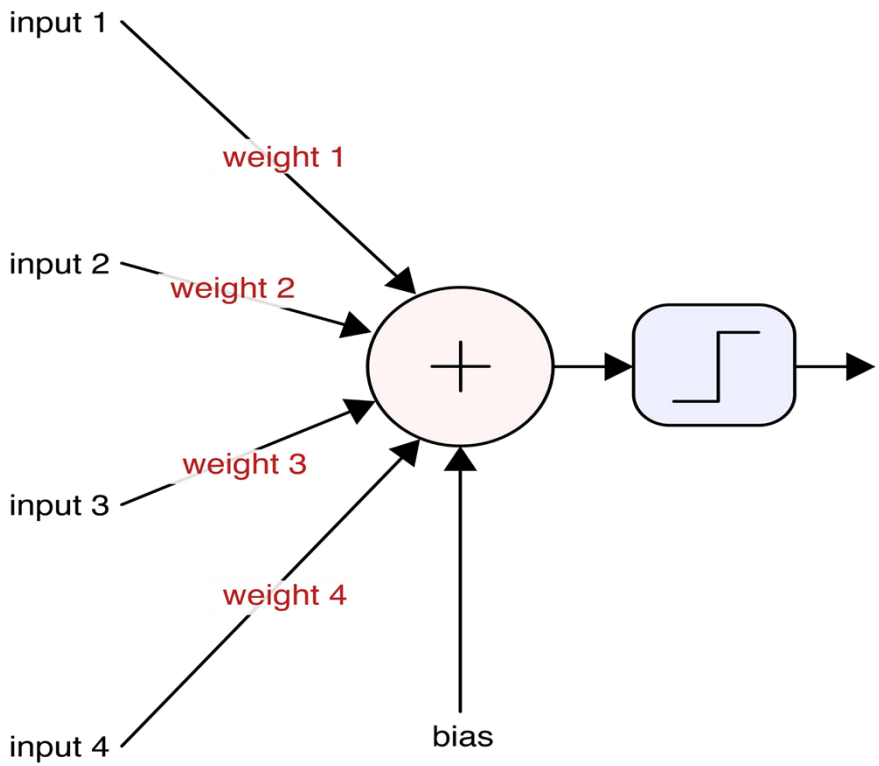


Figure 10.4: **A neuron is often drawn with the weights on the arrows. This "implicit multiplication" is common in machine learning figures. We've also replaced the threshold function with a step, to remind us that any activation function can follow the sum**

Note:

- In neural network diagrams, the weights and the nodes where they multiply the inputs, are not drawn.
- The weights are always there, and they always modify the inputs. They're just not drawn.

- Output of activation function might take on any value.
- Variety of activation functions, each with its own pros and cons.
- We'll run through them later

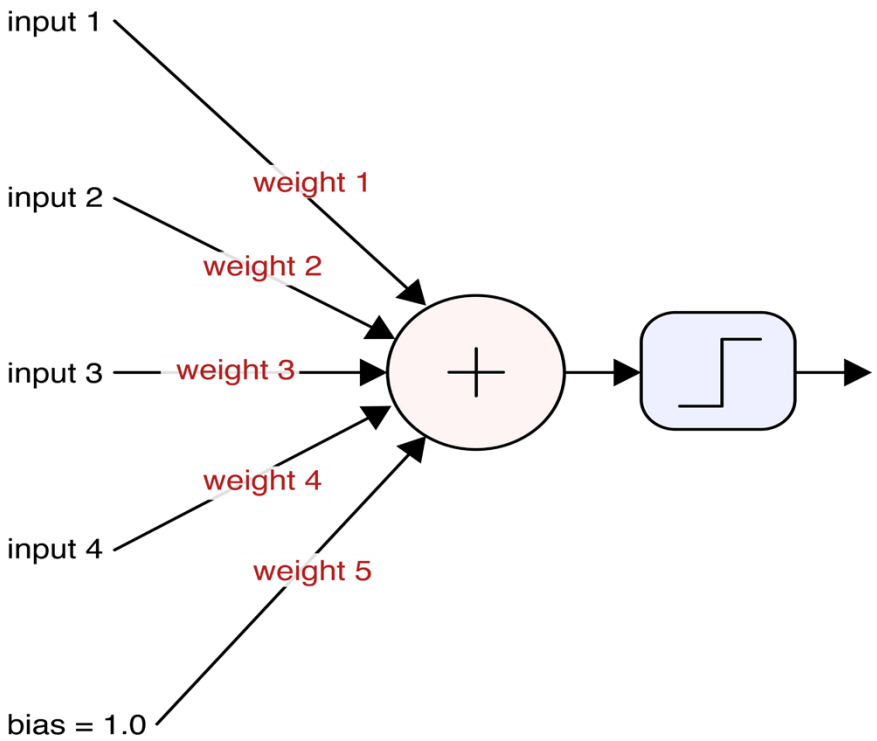


Figure 10.5: **The "bias trick" in action. Rather than show the bias term explicitly, as in Figure 10.4, we pretend it's another input with its own weight.**

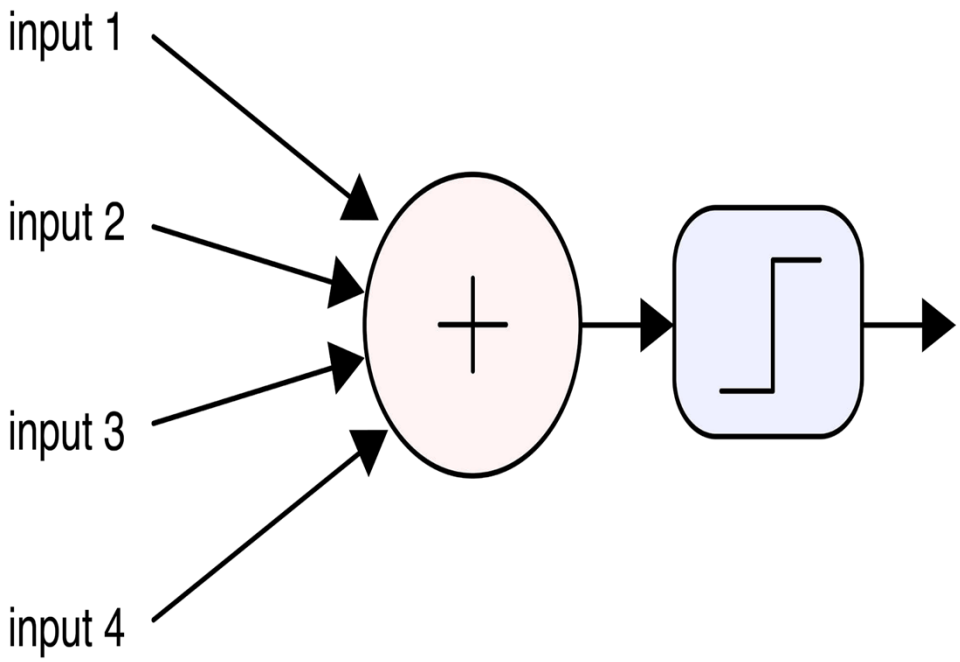


Figure 10.6: **A typical drawing of an artificial neuron. The bias term and the weights are not show, but the bias and weights are in there. We are supposed to add them back in mentally.**

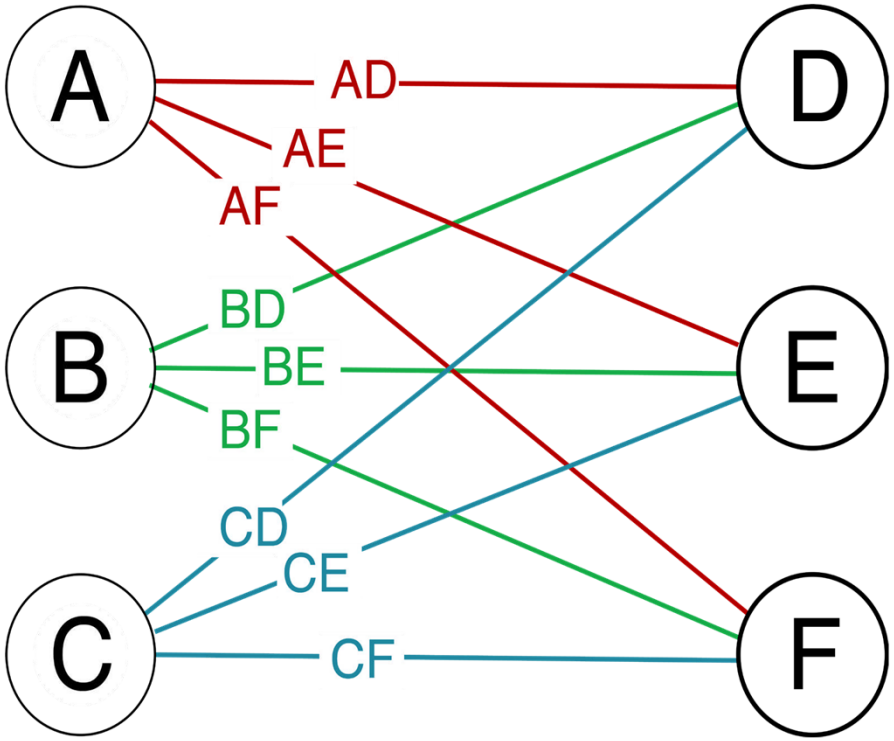


Figure 10.8: **The weights are shown explicitly in this diagram. Following convention, each weight is given a two-letter name formed by combining the names of the neuron that produced the output on that weight's wire, with the name of the neuron that receives the value as input. For example, BF is the weight that multiplies the output of B for use by F.**

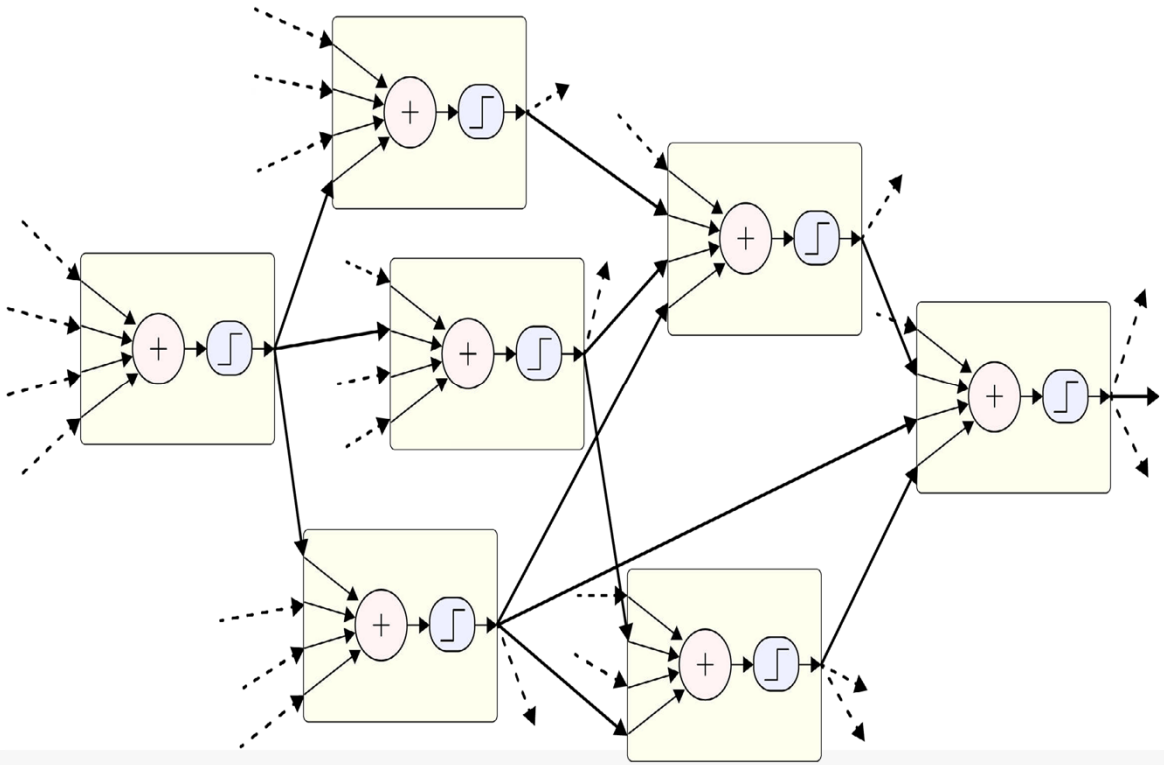
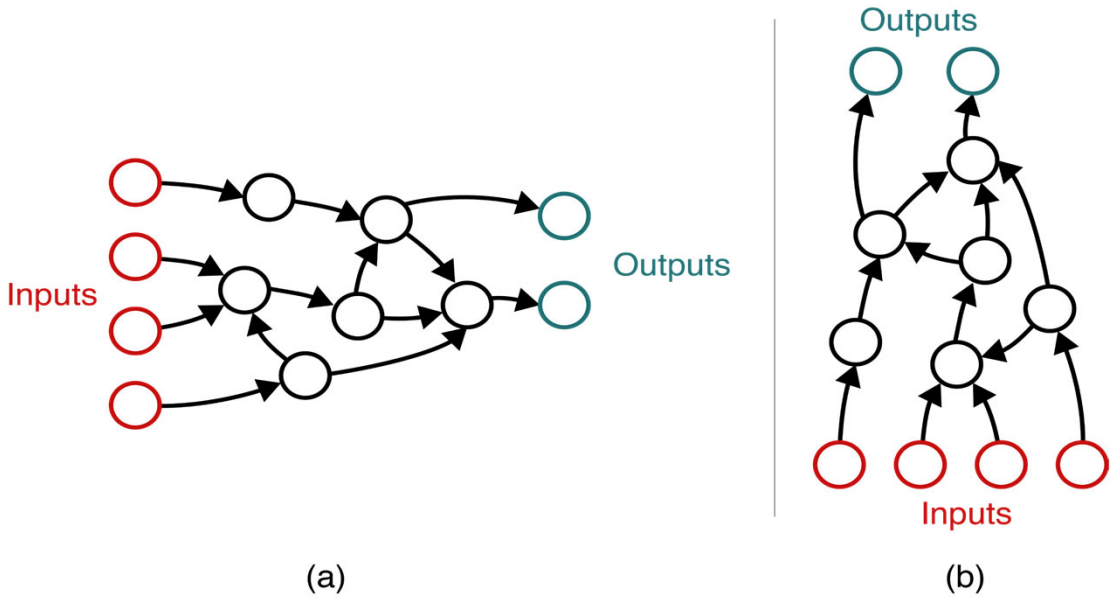


Figure 10.7: **A piece of a larger network of artificial neurons. Each neuron receives its inputs from other neurons The dashed lines show connections coming from outside this little cluster.**

Feedforward Networks

Deep learning systems are structured as networks of neurons.

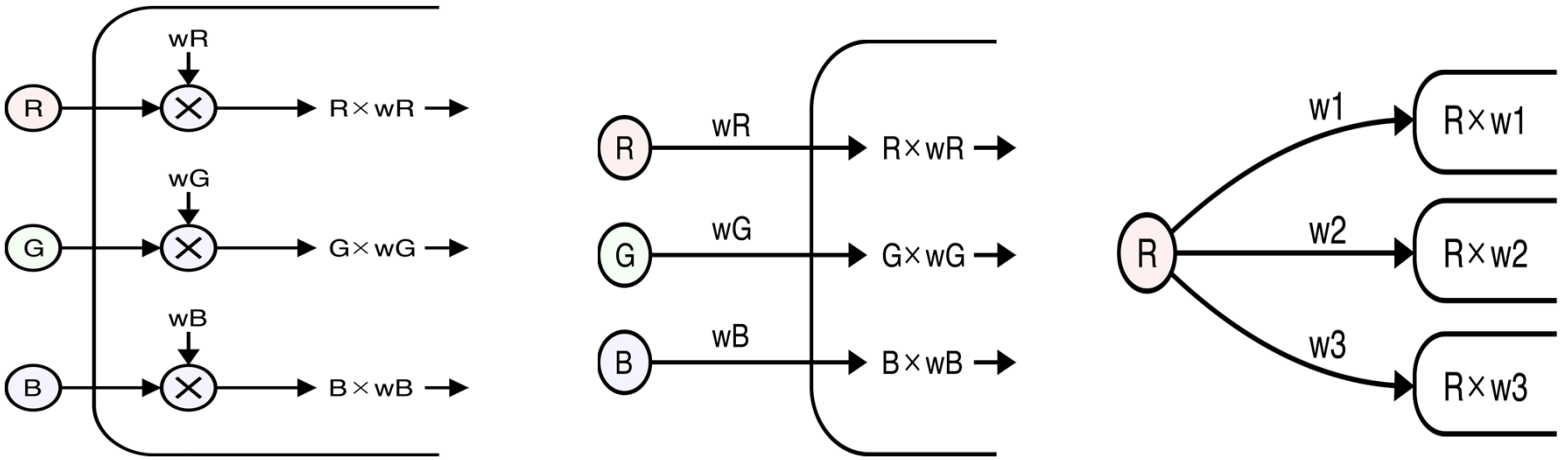
Much of their work is performed as data flows through them in a forward direction, starting at the inputs until it reaches the outputs.



This kind of graph is like a little factory.

Raw materials come in one end, and pass through machines that manipulate and combine them, ultimately producing one or more finished products at the end

Figure 16.1: Two neural networks drawn as graphs. Data flows from node to node along the edges, following the arrows. When the edges are not labeled with an arrow, data usually flows left-to-right or bottom-to-top. No data ever returns to a node once it has left. In other words, information only flows forwards, and there are no loops. (a) Left-to-right flow. (b) Bottom-to-top flow.



Left: Three values, named R, G, and B, are heading into a neuron. As each value enters the neuron, it's weighted, or multiplied by a number associated with that input. The resulting scaled value is then used by the rest of the neuron to compute its output value.

Middle: The weights are written on the wires carrying the values into the neuron.

Right: A single neuron, here labeled R, sends its output to three different neurons, then R's output value will be weighted by three potentially different values, one on each wire.

Activation Functions

The last step in an artificial neuron is to apply an activation function to the value it computes.

Here we'll see a variety of popular activation functions in use today.

Activation function – Step

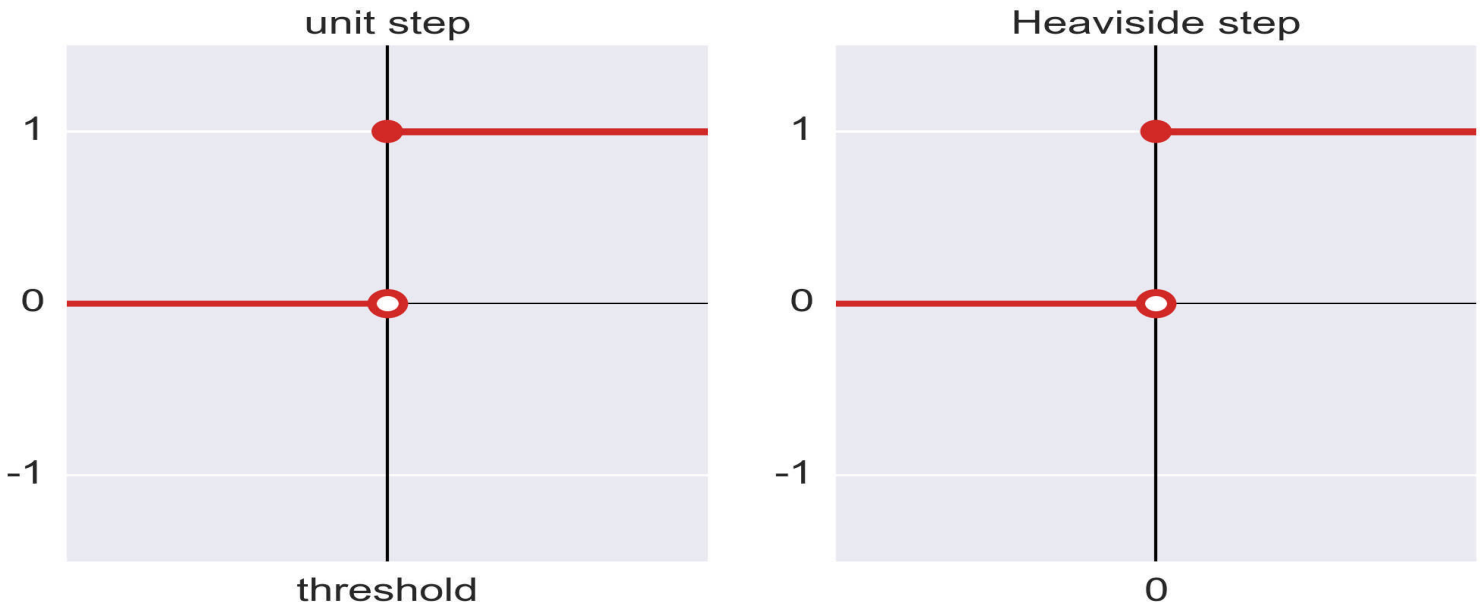


Figure 17.9: **A couple of popular step functions.**

Left: Unit step has a value of 0 to left of the threshold, and 1 to the right.

Right: The Heaviside step is a unit step where the threshold is 0.

Activation function –ReLU

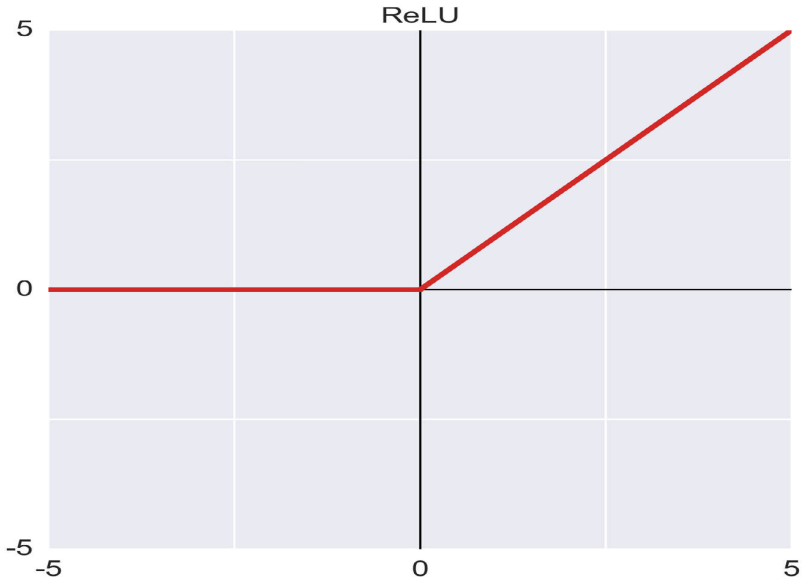


Figure 17.11: The ReLU, or rectified linear unit. Output is 0 for all negative inputs. Else, output = input

- ReLU is *not* a linear function.
- ReLU is popular because it's a simple and fast way to include a non-linearity in our artificial neurons.
- ReLU performs well and is often the first choice of activation function when building a new network.
- There are good mathematical reasons to use ReLU.

Activation Function Gallery

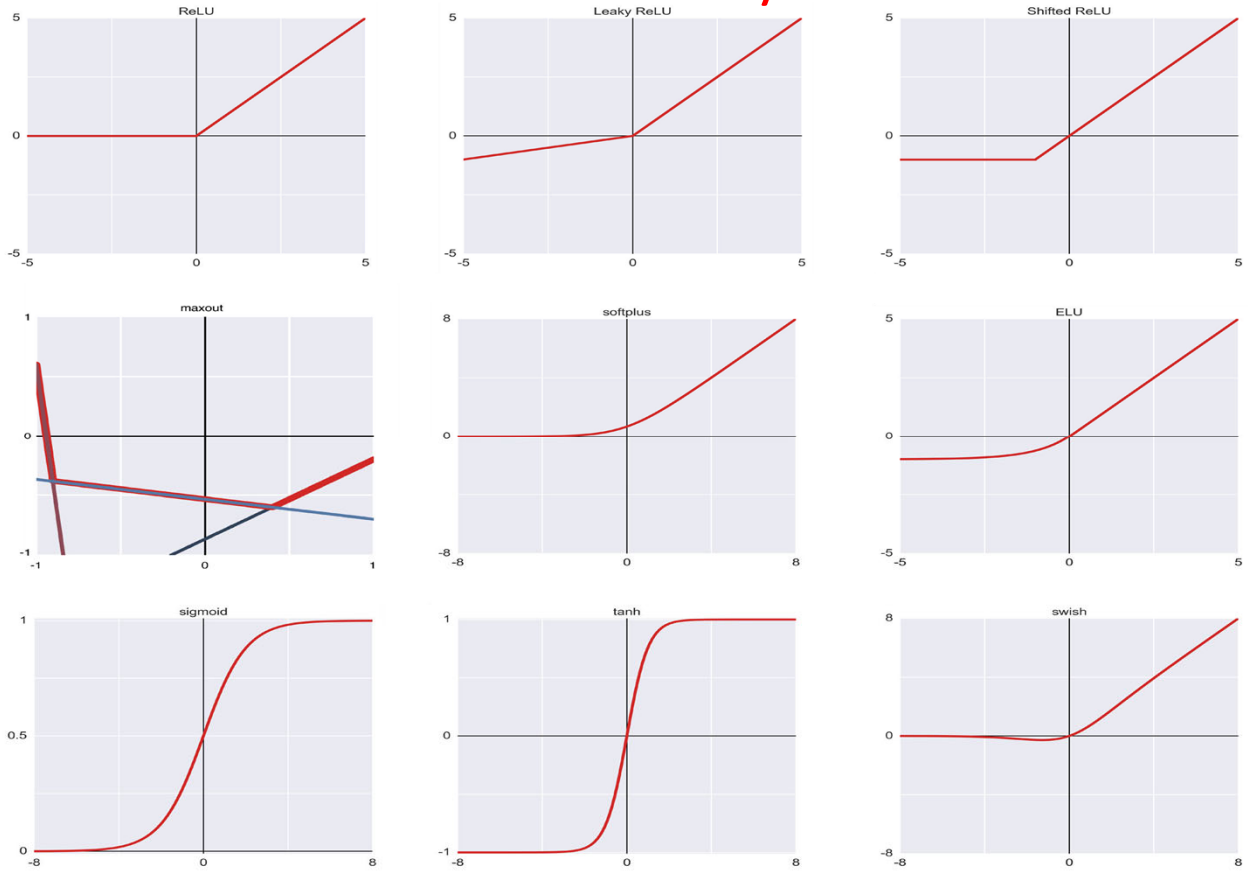
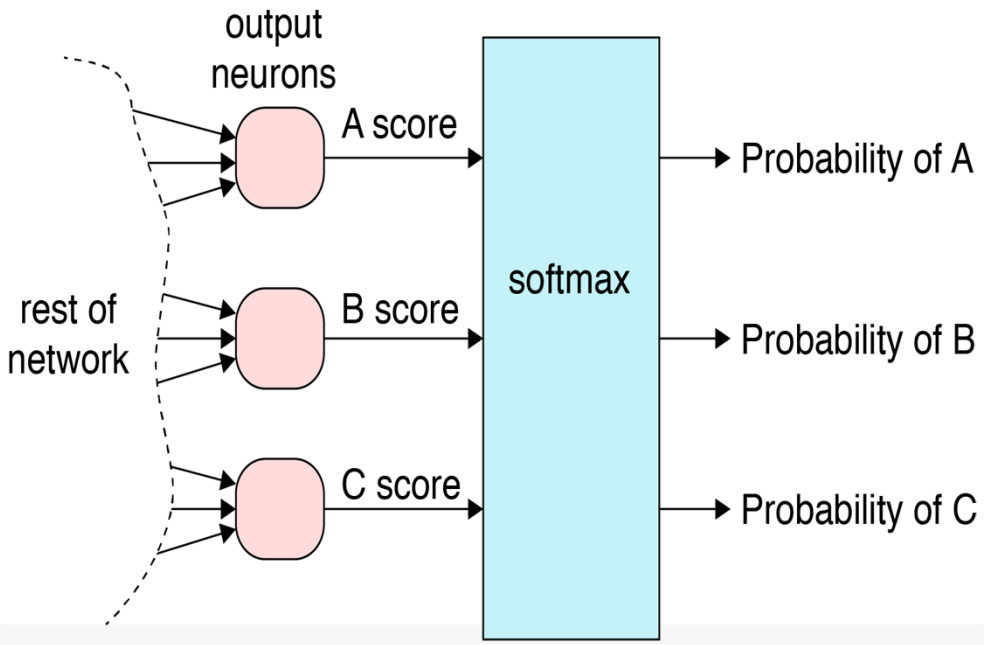


Figure 17.22: A gallery of popular activation functions.
Top row, left to right: ReLU, leaky ReLU, shifted ReLU.
Middle row: maxout, softplus, ELU. Bottom row: sigmoid, tanh, swish.

Softmax Function



- There's another kind of operation that we typically apply only at the output neurons of a classifier neural network,
- Used only if there are 2 or more output neurons.
- It's **not** an activation function.
- Why? Because it applies simultaneously to all the output neurons, not just one.
- The technique is called **softmax**,
- Often used as a final step for classification networks with multiple outputs.
- Softmax turns **numbers** that come out of the network into **probabilities of classes**.

Figure 17.23: **The softmax function takes all the network's outputs and modifies them simultaneously.**

Result is that scores are turned into probabilities.

Softmax Function - Example

Softmax turns our scores into probabilities.

It's widely used at the end of neural networks that are used for classification.

Figure 17.24: Three hypothetical networks, left, middle, and right.
Bottom row shows outputs of softmax
Top row: Scores from a classifier, which are inputs to Softmax.
Bottom row: Results of running the scores in the top row through Softmax.

Softmax operation changes a list of numbers so that they represent probabilities.

Uses a mathematical transformation.

Result : Every output is from 0 to 1, and sum of all outputs =1.

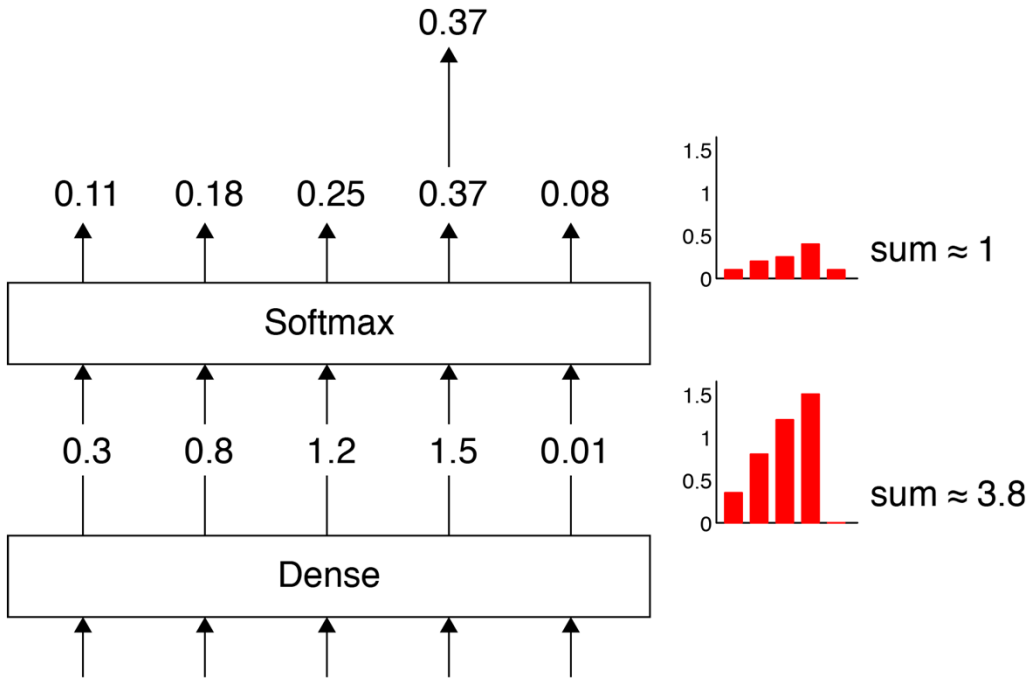


Figure 20.7: **In this example, we have five values coming out of a fully-connected, or dense, layer.**

Values are between 0.01 and 1.5, and they sum up to about 3.8.

After the softmax, the values are from 0 to 1 and add up to 1.

Layers: Input and Output

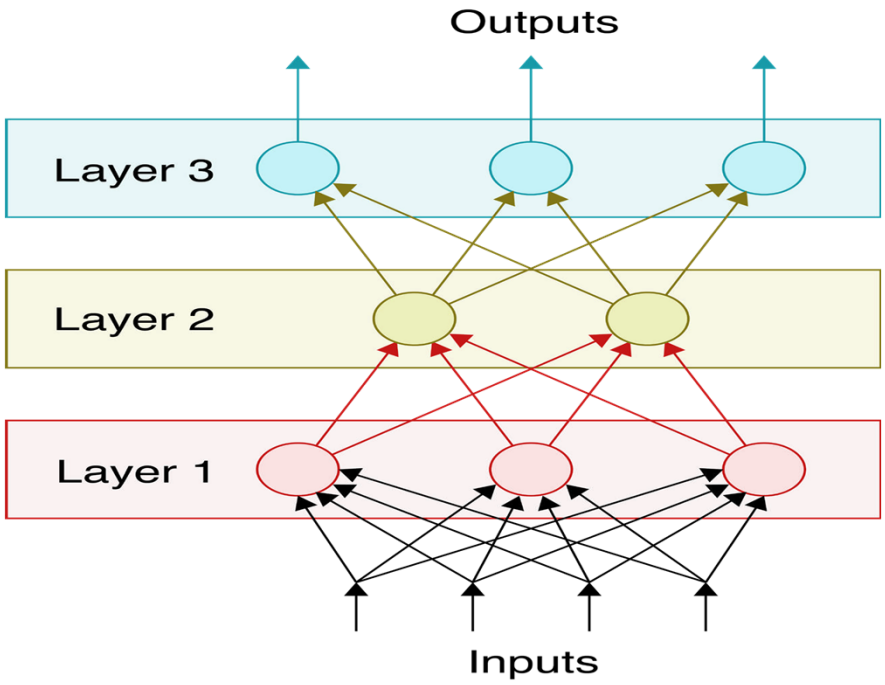


Figure 20.1: A deep learning network, having 4 inputs flowing through 3 layers, creating 3 outputs at the end.

Network is "fully-connected" - every neuron in each layer receives input from every neuron on the previous layer.

- **Output layer**: the topmost layer (layer 3 in Figure 20.1). It contains the final set of neurons.
- **Input layer**: It is NOT Layer 1. It is the row of black arrows at the bottom of Figure 20.1. The input layer refers to the memory that holds the input values.
- These labels simply refer to the position of the layer in the stack: the input is at the start (the bottom, or left) and the output is at the end (the top, or right).
- **Hidden layers**: Layer 1 and Layer 2 in Figure 20.1. These layers are in-between, and are "hidden" from view, and thus are called "hidden layers"
- Someone looking at this network from above or below, can see only the input layer or the output layer. Most networks have a single input layer and a single output layer.

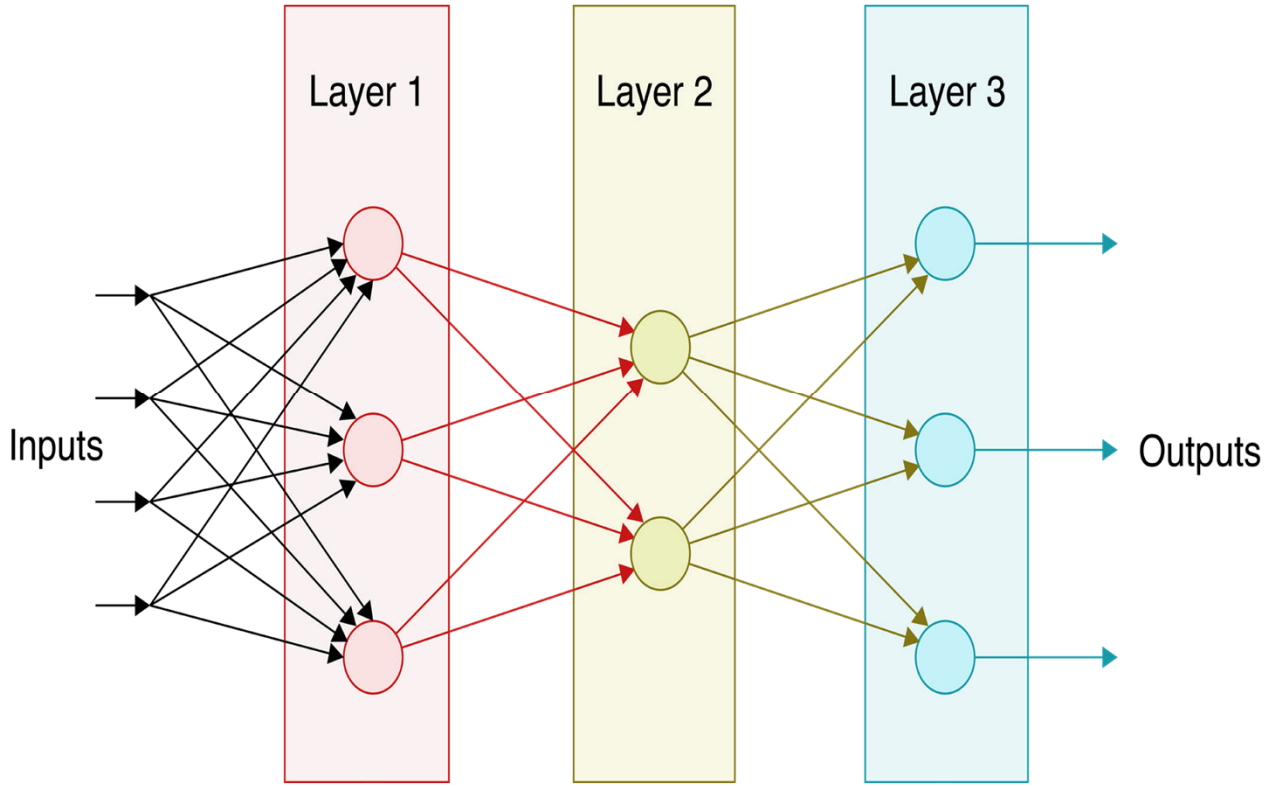
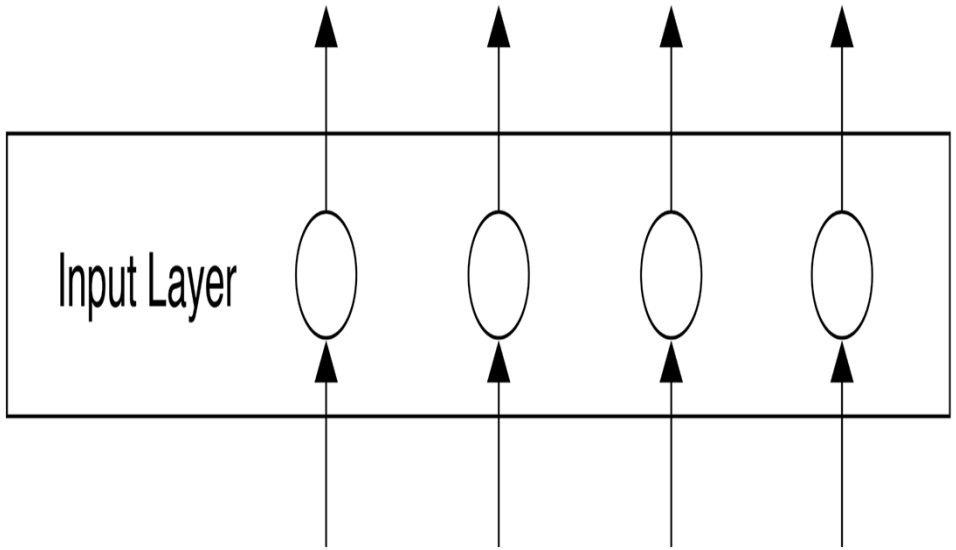


Figure 20.2: **The same deep network of Figure 20.1, but drawn with data flowing left-to-right.**

Input layer – some notes



- The **input layer** is usually **not shown** in deep-learning architecture diagrams.
- It is merely some **memory** that holds the input values.
- Note that these are **not neurons**, since the input layer does no processing.
- It can be thought of as simply as a collection of **chunks of memory**, each capable of holding one number of the input.

Figure 20.4: The input layer is just a placeholder where we can temporarily store the input data.

Ouput Layer – Some notes

- Choose the number of neurons in the output layer to match the type of problem we're trying to solve.
- **Regression problem**:
  - A single numerical output, then there will be just one neuron in the output layer, and that neuron's value is our prediction.
- **Binary classifier**: We have a choice.
  A. Just one neuron with output values from, say, 0 to 1. Values near 0 mean the input is from one class, while values near 1 mean the input is of the other class.
  B. Alternatively, we can have two output neurons, one for each category. We'll typically find which neuron has the larger value, and assign the corresponding category to the input.
- **Multi-class classifier:** Have as many outputs as there are classes.

Output Layer Example

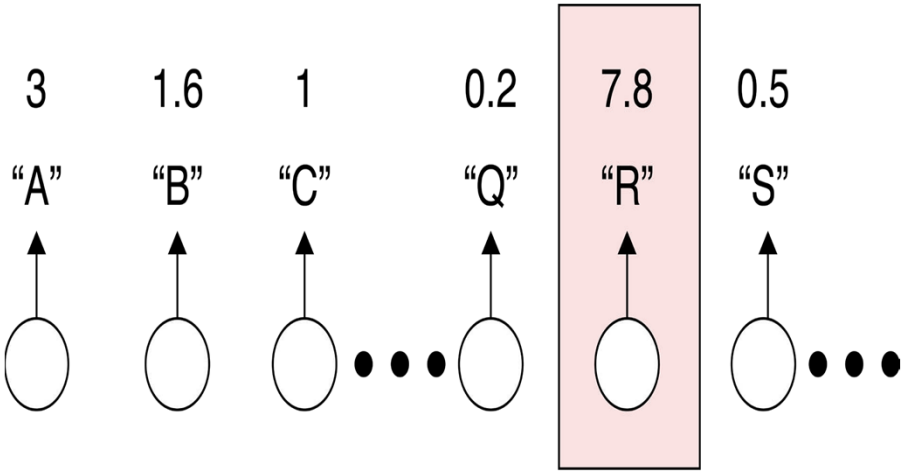


Figure 20.5: **If we're categorizing individual letters, we might have 26 outputs, each one giving us a score for that letter.**

Here, the letter "R" has the largest score shown.

- **Example**: Recognize lower-case letters of the Roman alphabet.
- Have 26 output neurons, one for each letter, providing us with a score for each letter.
- Choose the output with the highest score as the best choice for the category of the input.
- If we want to interpret these outputs as probabilities, we can pass them through a softmax step.
- Figure 20.5 shows the idea.

Fully-Connected Layer

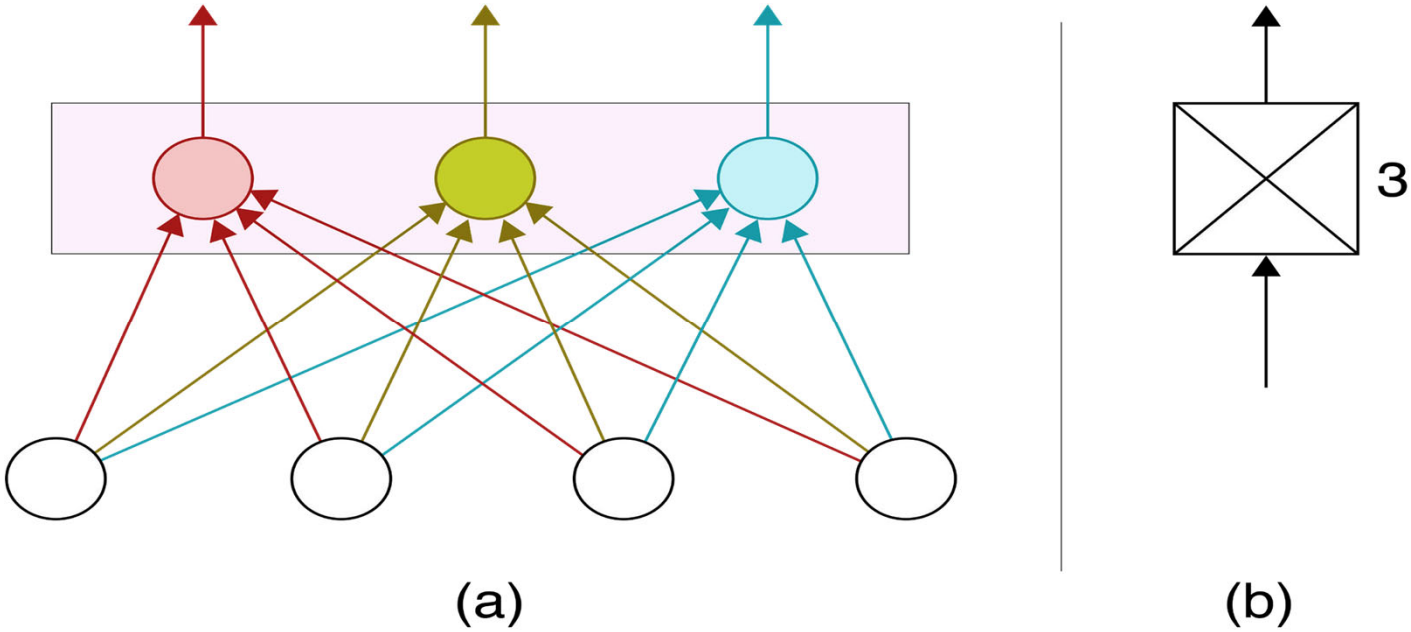


Figure 20.6: **A fully-connected layer, or Multi-Layer Perceptrons (MLPs)**
(a) The colored neurons make up a fully connected layer.
Each of the neurons in the upper layer receives an input from every neuron in the previous layer.
(b) Schematic symbol for a fully-connected layer.

Fully Connected Network

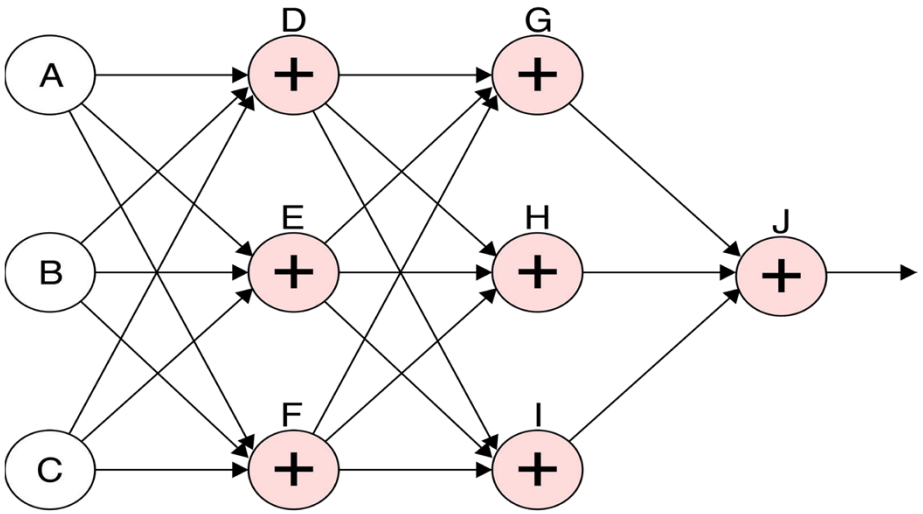


Figure 17.3: **A fully-connected network of 3 inputs, 1 output, and two internal layers of 3 neurons each. Each arrow represents a connection and implicitly has an attached weight.**

Deep Learning

1

Deep learning is solving some machine learning problems using a specific type of approach.

2

Central idea: Build machine learning algorithm out of a series of discrete **layers**.

3

If we stack these up vertically, we say that the result has **depth**, or that the algorithm is **deep**.

4

Many ways to build a deep network, and many different types of layers to choose from when building it.

5

A simple example is in Figure 1.21.

A Deep Layer Network (2 layers)

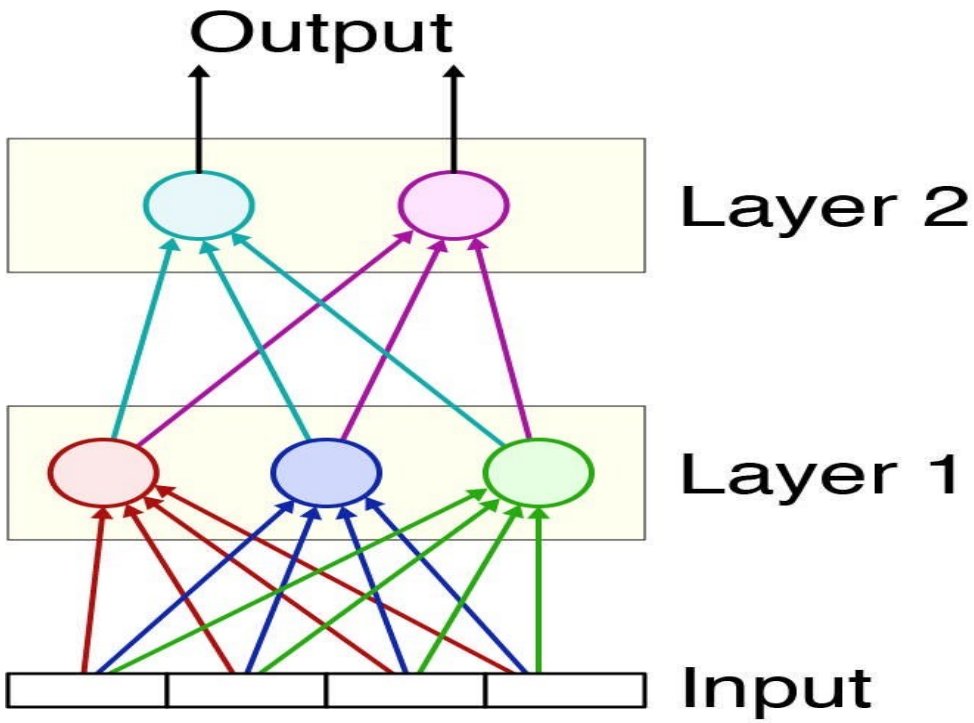


Figure 1.21: A deep network with 2 layers.

Basic Operation of Neurons

This network in Fig. 1.21 has two layers, shown as yellow boxes.

Inside each layer, little pieces of computation called **artificial neurons**, or **perceptrons.**

What do these artificial neurons do ? They

Take in a bunch of numbers,

Combine them in a particular way, and

Pass on that output (a new number) to the next layer.

How “Deep” can Deep Layers be?

Networks can have layers numbering in the hundreds or more.

Instead of having only 2 or 3 neurons in a layer, these can contain hundreds or thousands, arranged in a variety of ways.

How easy is to construct deep layers?

- We can create a full network by calling routines in a deep learning library.
- DL libraries take care of all the construction work.
- Just one short line of code can instantiate a new layer, and make it part of our network.
- The library handles all of the internal and external details for us.
- Because of this structure (stacking) and ease of construction, deep learning is like building something out of Lego blocks.
- We just stack up the layers we want, each specified by a few arguments, name a few more options, and start training.

Is DL really that easy?

Great that building a network is so easy,

But that also means that every decision we make can have **big implications**.

- Picking the right layers,
- Picking their parameters,
- Handling the other options

All need to be done with care.

So, not really that easy!

What can happen with wrong choices?

- If we get something wrong, the network won't **run or learn**.
- We'll just feed it data and it will give us **useless** results.

Application 1: DL for Image Classification

See Figure 1.10 for an example of DL.

Deep network with 16 layers to classify those photos.

See what the DL system returns as output, in Fig. 1.22

Output of DL Image classifier

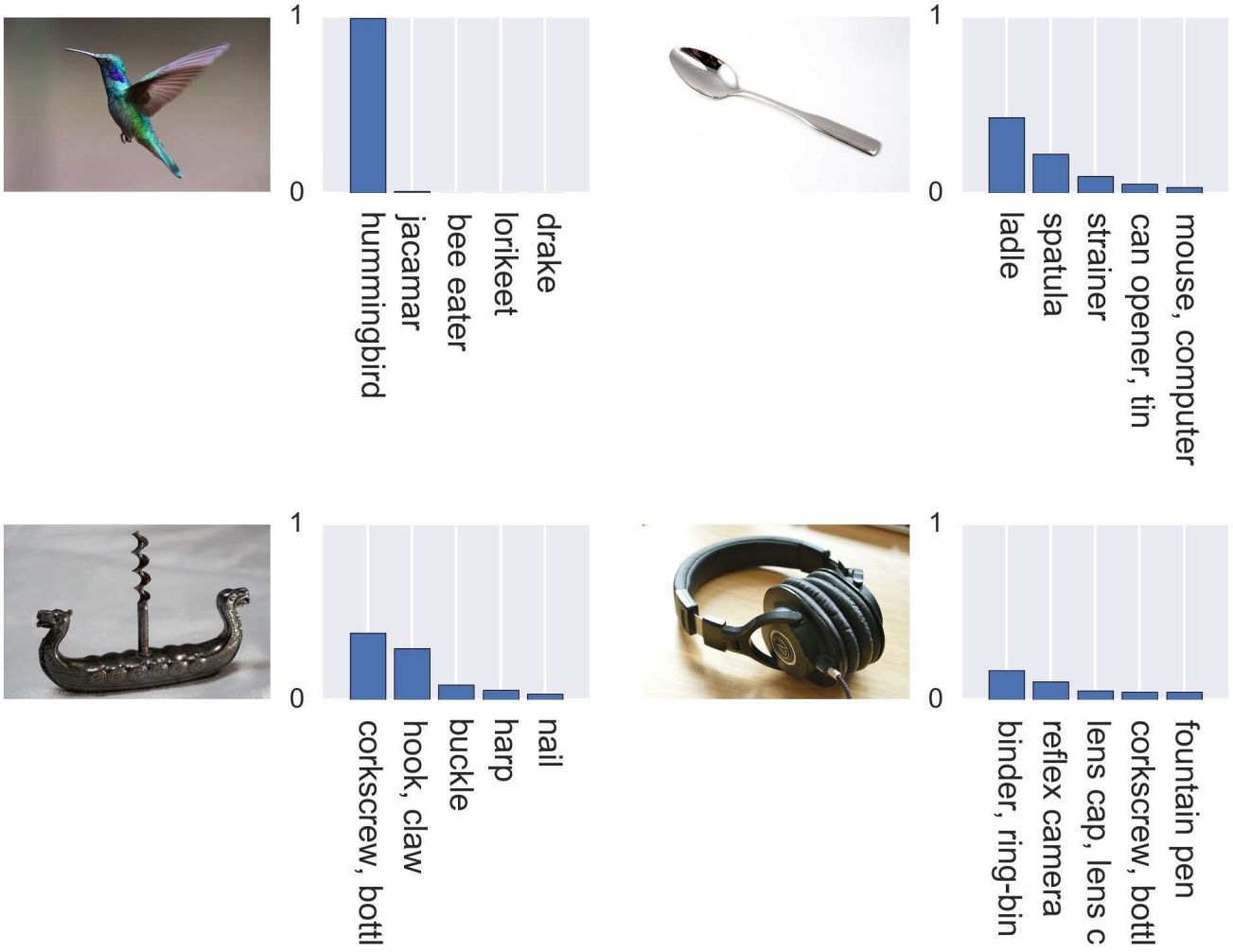


Figure 1.22: Four images and the predictions of our deep learning classifier.

System wasn't trained on spoons or headphones, so it's doing its best to match those images to things it knows about.

Performance of DL Image classifier

- Figure 1.22 shows the four photos along with the categories predicted by the DL network and their relative confidences.
- It recognized the hummingbird **perfectly**.
- It wasn't **very sure** about the corkscrew ship.
- Never saw a spoon or headphones before, it **guessed**.

Application 2: DL Application to Handwritten Numbers

We train our DL system on hand-drawn digits from 0 to 9.

We use a famous data set called **MNIST.**

We will then ask it to classify new images of handwritten digits.

Example 2: DL application to Handwritten digits (Predictions)

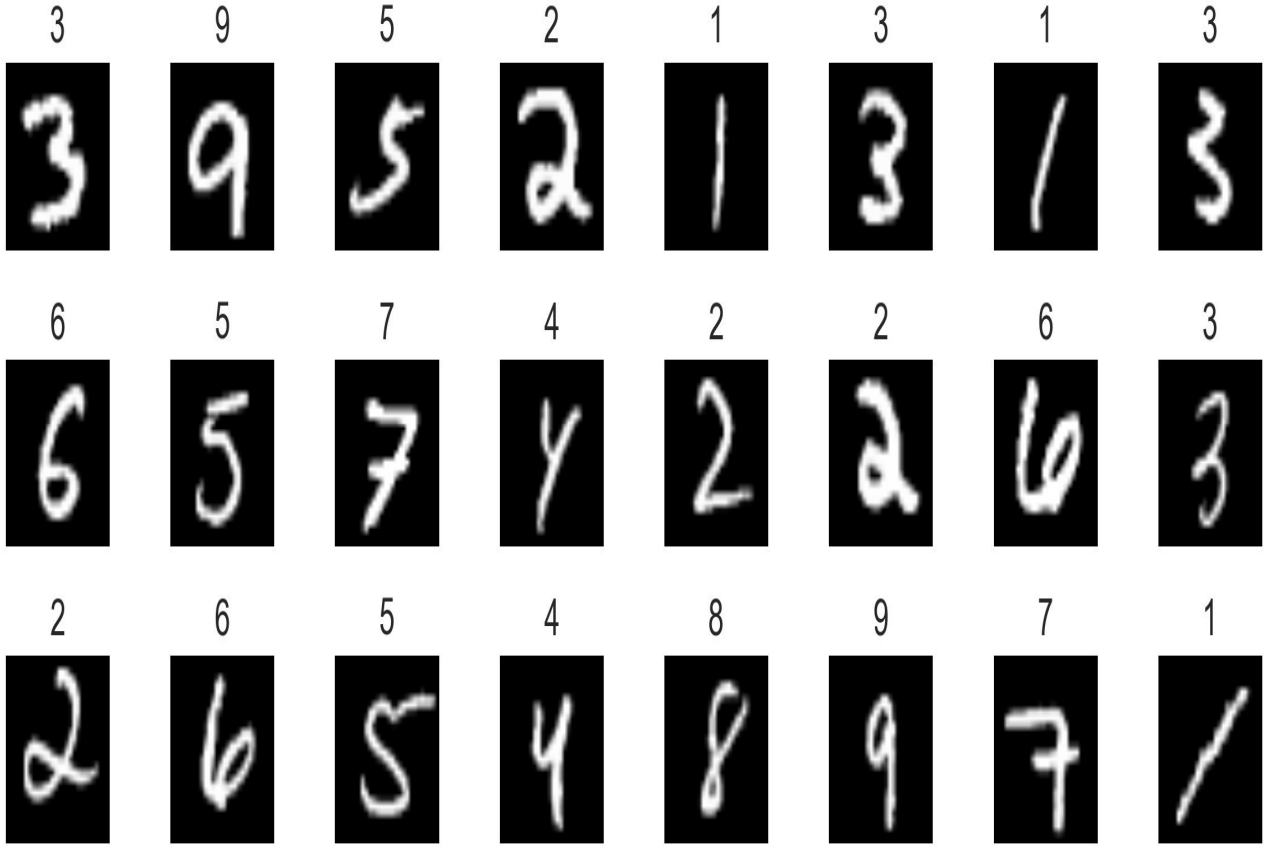


Figure 1.23: A deep learner classifying handwritten digits.

Row 2 see “7”, “4”, “2” and “6”,

Row 3 see “4”, “8” and “1”

Example 2: DL application to Handwritten digits (Performance)

- How's the performance of the classifier?
- Classifier got every one of them correct.
- Modern classifiers have over 99% accuracy.
- Even the little system we built just for this figure correctly classified 9,905 images out of 10,000
- It disagrees with the human expert only 95 times out of 10,000 !
- **So, this is a highly successful application**

Why GPU for Deep Learning ?

In big problems (like identifying which of 1000 different objects is in a photo), deep learning networks can become big.

The network involves thousands or even millions of parameters.

Require a **mountain of data for training**, and **lot of computer time**.

Take help of Graphics Processing Unit (or **GPU**).

What's a GPU?

- A GPU is a specialized chip that sits alongside the **Central Processing Unit** (or **CPU**).
- CPU carries out most of the tasks involved in running a computer, such as talking to peripherals and the network, managing memory, handling files, and so on.
- GPU was originally designed for **complex graphics, such as rendering scenes in 3D games.**

Why's a GPU good for DL?

Operations for training a deep learning system just happen to be a **great match** to the operations that **GPUs** were designed to do, so those training operations can be performed with great speed.

Design of GPUs: operations can also be carried out **in parallel**, so multiple calculations can be performed simultaneously.

New chips designed specifically for **accelerating deep learning calculations** are starting to appear in the market.

Classification and Clustering

PSV Nataraj

How to categorize (or classify) new samples?

1. Classify new samples by breaking up the space into different regions, and then test a point against each region (egg example)

2. Group the training set data itself into **clusters**, or similar chunks.

Classification

Classification

Assign each input data to one or more categories that best describe it

Look at Binary classification

Has only two categories.

.

Example: Training data as the weight and length of eggs

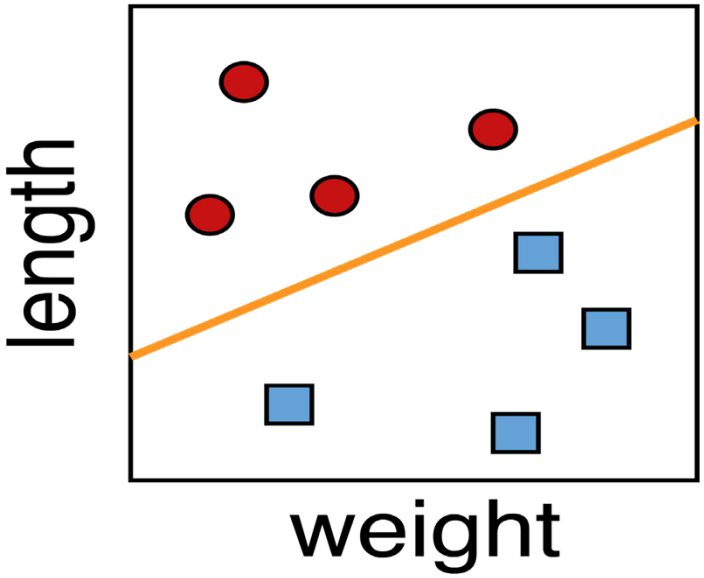


Figure 7.1: **Categorizing eggs. The red circles are fertilized eggs. The blue squares are unfertilized eggs. Each egg is plotted as a point given by its two dimensions of weight and length. The orange line separates the two clusters.**

Draw a straight line between the two groups of eggs.

One side of the line is fertilized eggs, and the other side has unfertilized.

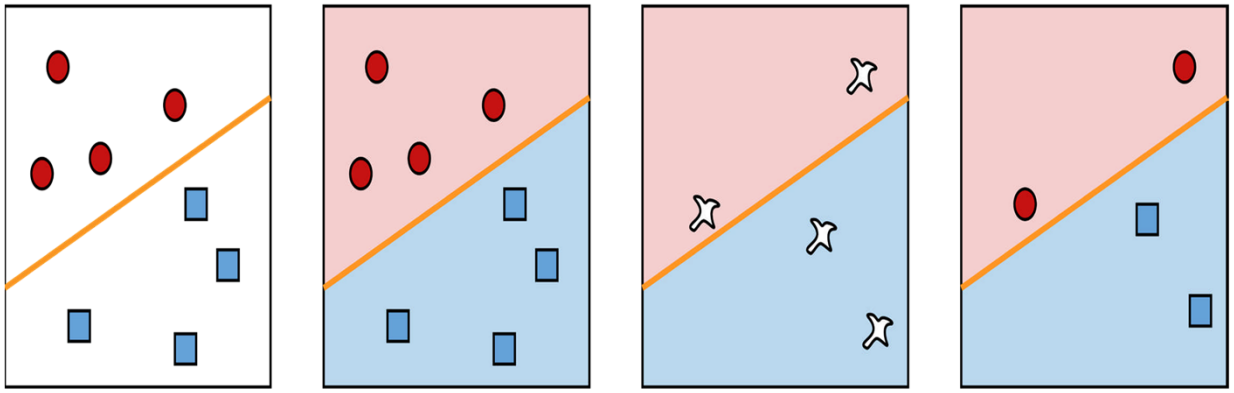
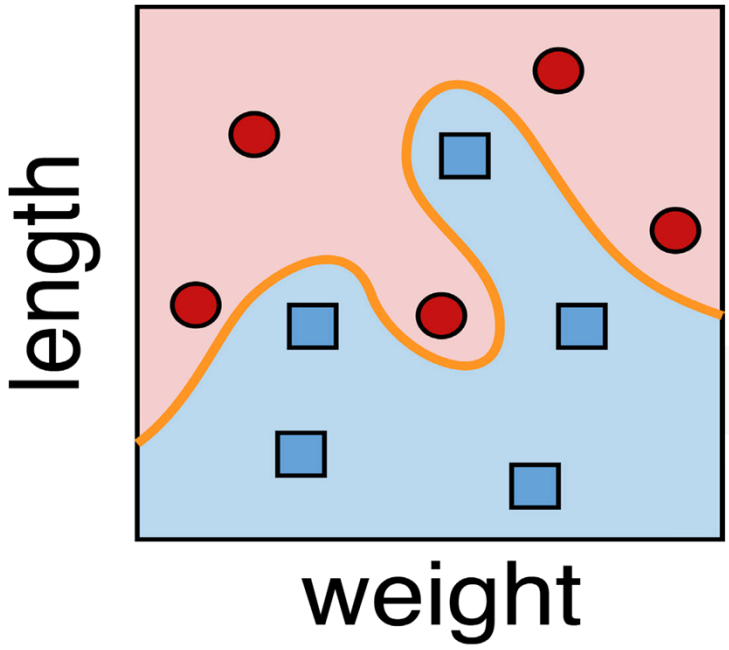


Figure 7.2: **Categorizing new eggs.**
Far left: Our starting data. Second from left: The red and blue regions show how we've decided to split the fertilized and unfertilized eggs.
Third from left: Four new eggs arrive to be categorized.
Far right: We've assigned a category to each of the new eggs.



Sections into which we chop up the plane are **decision regions.**

Lines or curves between them **decision boundaries**

Figure 7.3: **When we add some new types of chickens to our flock, it may get trickier to determine which eggs are fertilized based just on the weight and length.**

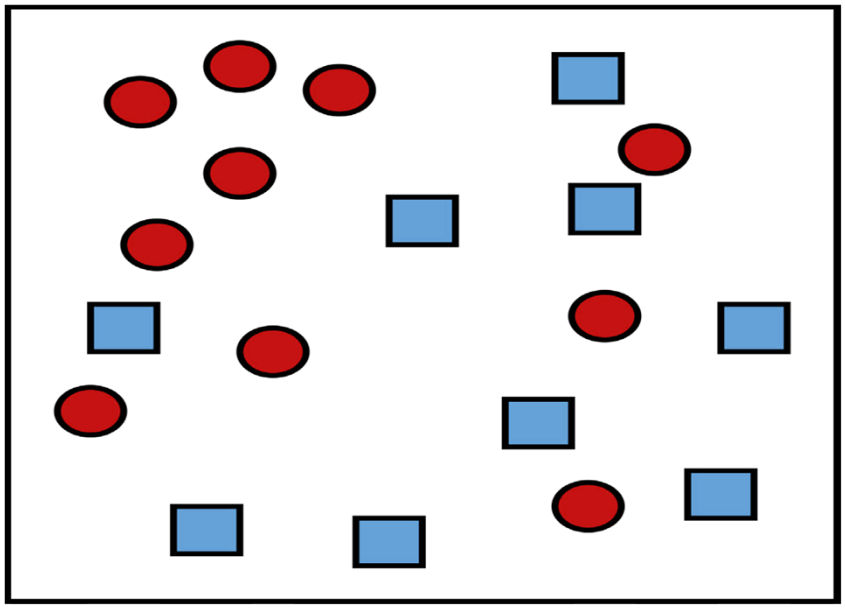


Figure 7.4: **New purchase of chickens has made it much harder to distinguish the fertilized eggs from the unfertilized eggs.**

There's still a mostly-red region and a mostly-blue region, but there's no clear way to draw a line or curve that spearates them.

So a better way to talk about which sample is in which category is to consider **probabilities** rather than certainties.

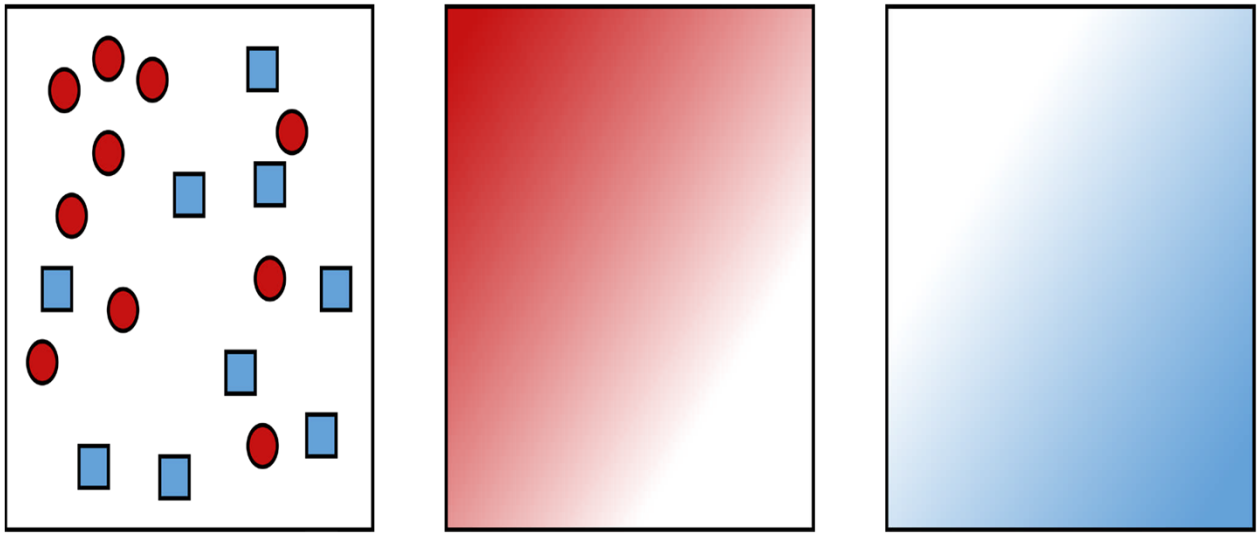


Figure 7.5: **Given the overlapping results shown at the far left:**

We can give every point in our grid a probability of being fertilized, shown in the center where brighter red means the egg is more likely to be unfertilized.

The image at the far right shows a probability for the egg being unfertilized.

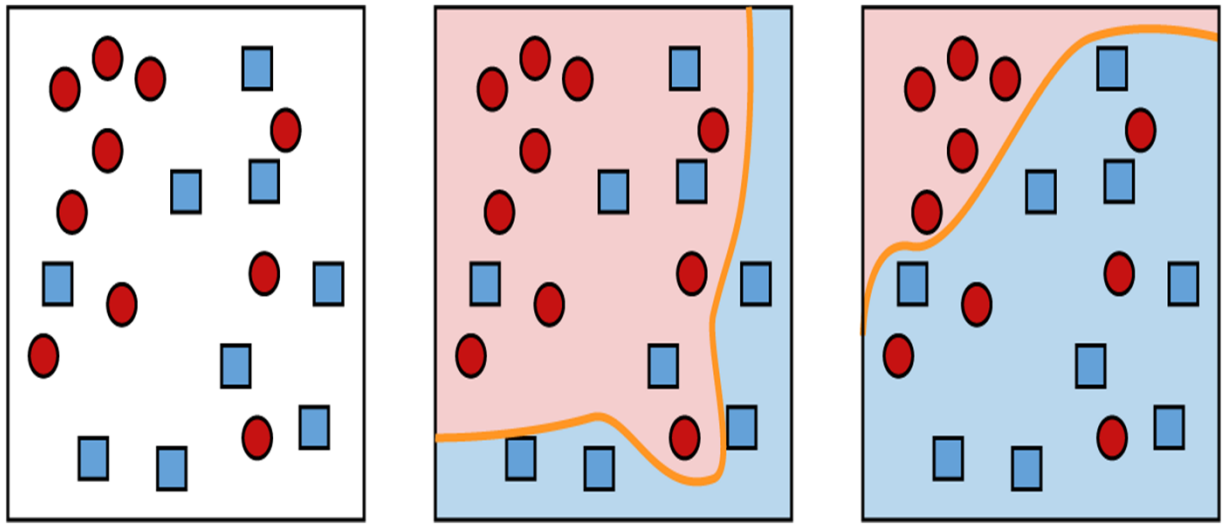


Figure 7.6: **Given the results at the far left, we may choose a policy shown in the center, which accepts some false positives (unfertilized eggs classified as fertilized) to be sure we correctly classify all fertilized eggs.**

Or we may prefer to correctly classify all unfertilized eggs, with the policy shown at the right.

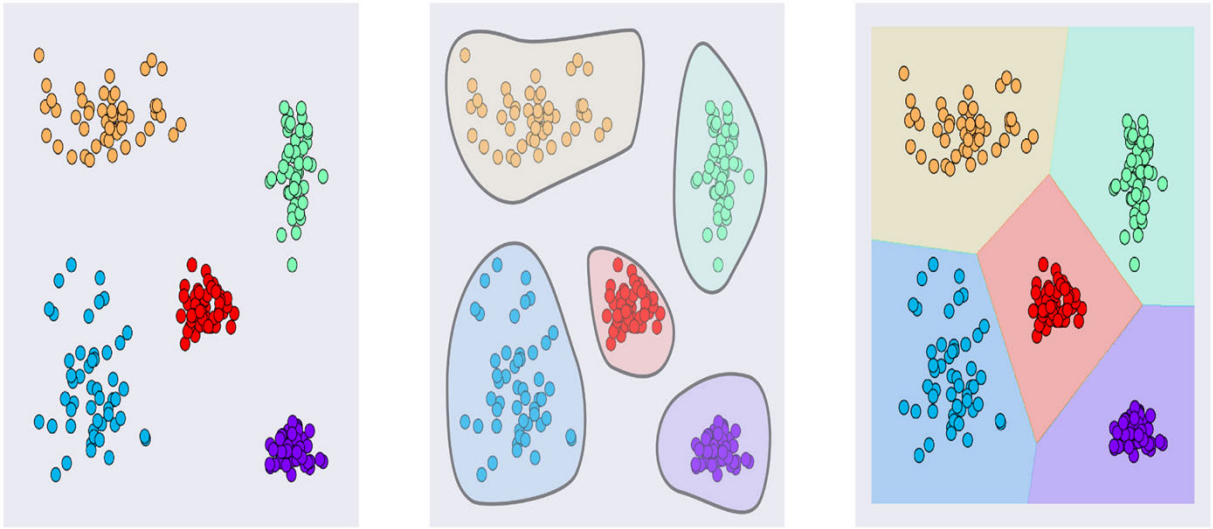


Figure 7.14: Steps of growing clusters by automatic clustering algorithms

Left: Starting data with 5 categories.

Middle: Automatically identifying the 5 groups.

Right: Growing the groups outward so that every point has been assigned to one class.

Clustering Algorithms
(used only for unlabeled data)

k-means clustering

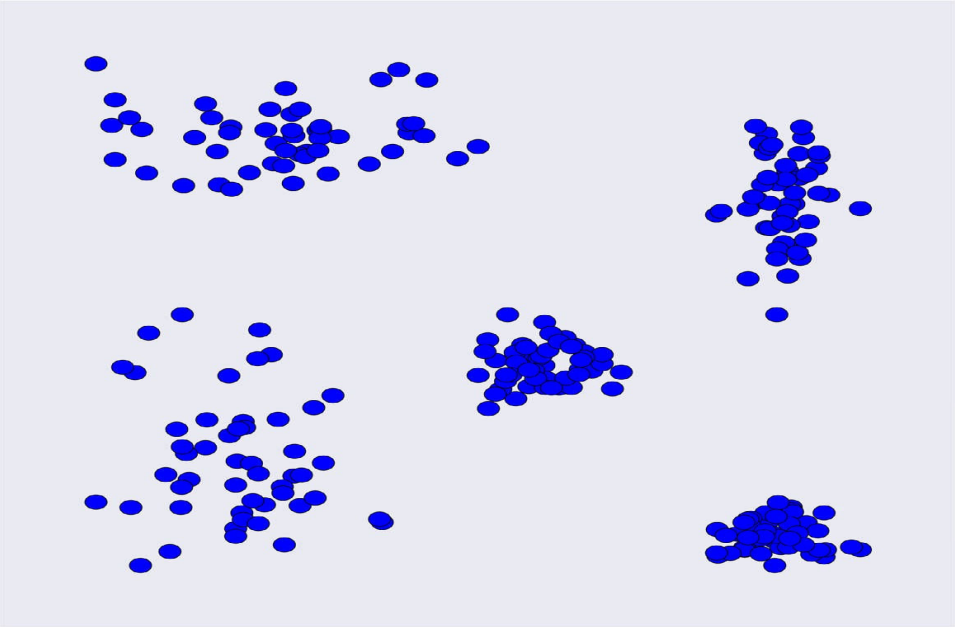
Automatically derives clusters **from unlabeled data**, but we tell how many clusters to find.

"number of clusters" value is denoted as *k*.

k is a **hyperparameter**, or a value that we choose before training our system.

Our chosen value of *k* tells the algorithm **how many regions** to build

Because the algorithm uses the means of groups of points to develop their clusters, it's called k-means clustering



Example: 200 unlabelled Data points

in 5 groups.

Try to automatically find the clusters

Figure 7.15: **A set of 200 unlabeled points.**

They seem to visually fall into 5 groups.

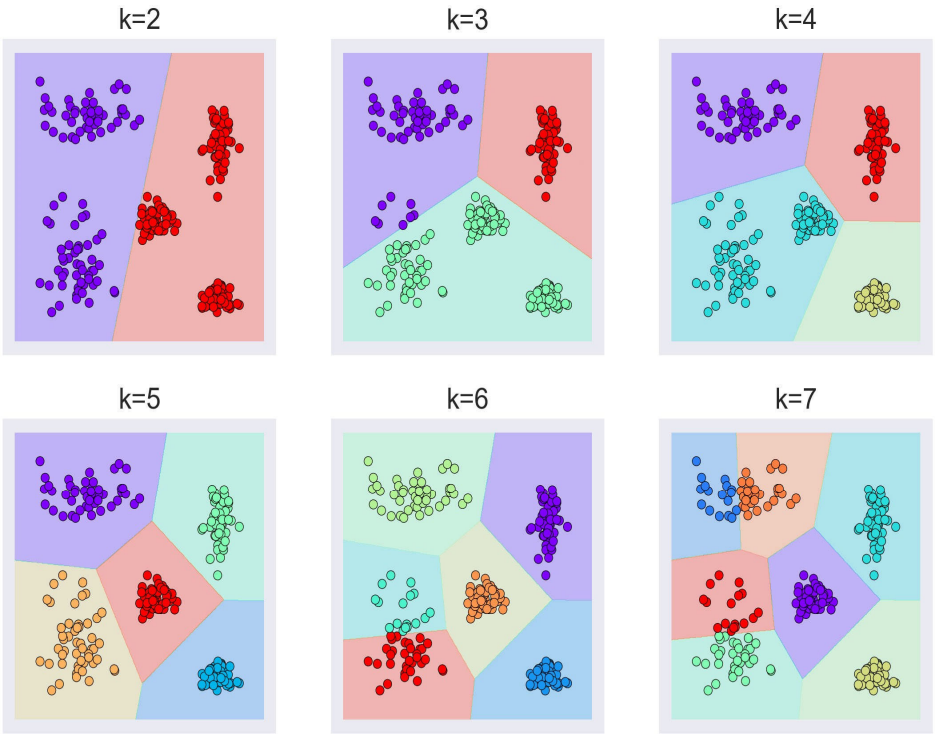


Figure 7.16: **The result of automatically clustering our data in Figure 7.15 for values of k from 2 to 7.**

Unsurprisingly, k=5 produces the best results, but we deliberately made this data easy to separate.

With more complex data, we might not know the best value of k to specify.

Automatically search for a good value of k,

Evaluate the predictions of each choice

Report the value of k that performed the best.

Classification for labeled data

k-nearest neighbours or kNN

Works with labelled data

Finds k-nearest neighbours or kNN

We saw an algorithm called *k-means clustering*.

Despite the similarity in names, *k-means clustering* and *k-nearest neighbor* are very different techniques.

Key difference: **k-means clustering learns from unlabeled data, while kNN works with labeled data**.

They fall into unsupervised and supervised learning, respectively.

k-Nearest Neighbors (KNN)

Non-parametric algorithm called **k-Nearest Neighbors**, or **kNN**.

k is a positive number.

Set k value before the algorithm runs - it's a **hyperparameter**.

kNN is fast to train, because all it does is **save a copy of every incoming sample into a database**.

The interesting part comes when **training is complete**, and a new sample arrives to be categorized.

Central idea of how kNN categorizes a new sample is in Figure 13.1.

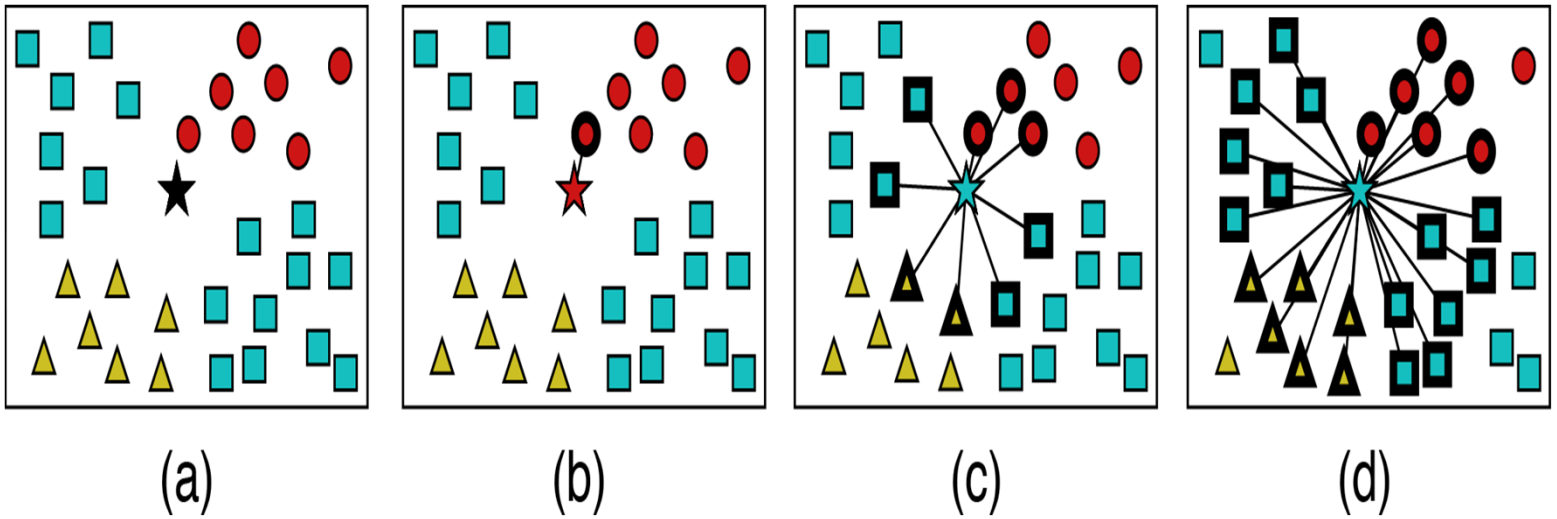


Figure 13.1: To find the class for a new sample, we find the most popular of its k neighbors.
(a) A new sample (the star) surrounded by samples of 3 classes.
(b) When k=1, the 1 nearest neighbor is round, so the category for the star sample is round.
(c) When k=9, we find 3 round samples, 4 squares, and 2 triangles, so the star is assigned the square category.
(d) When k=25, we find 6 round samples, 13 squares, and 6 triangles, so the star is again assigned to the square category.

kNN accepts a new sample to evaluate and a value of *k*.

Finds closest *k* samples to the new sample, and lets them "vote" on the appropriate category.

Whichever category has the most votes wins, and that becomes the category for the new sample.

Let's put kNN to the test. In Figure 13.2 we show a "smile" dataset of 2D data that falls into two categories.

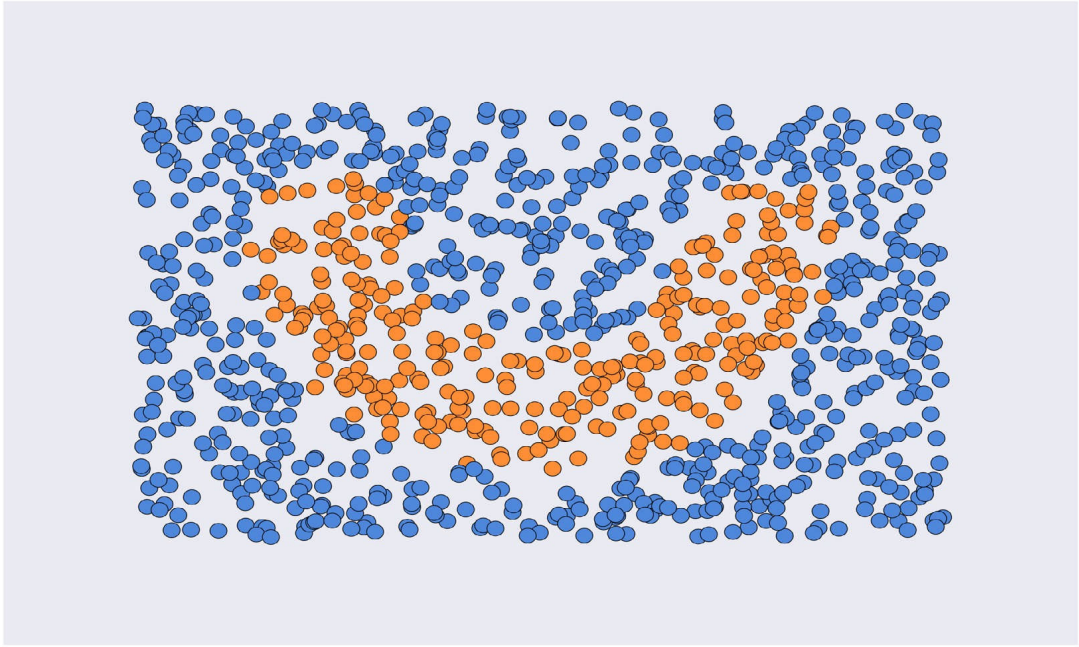


Figure 13.2: A "smile" dataset of 2D points. There are two categories, blue and orange.

Using kNN with different values of k gives us the results in Figure 13.3.

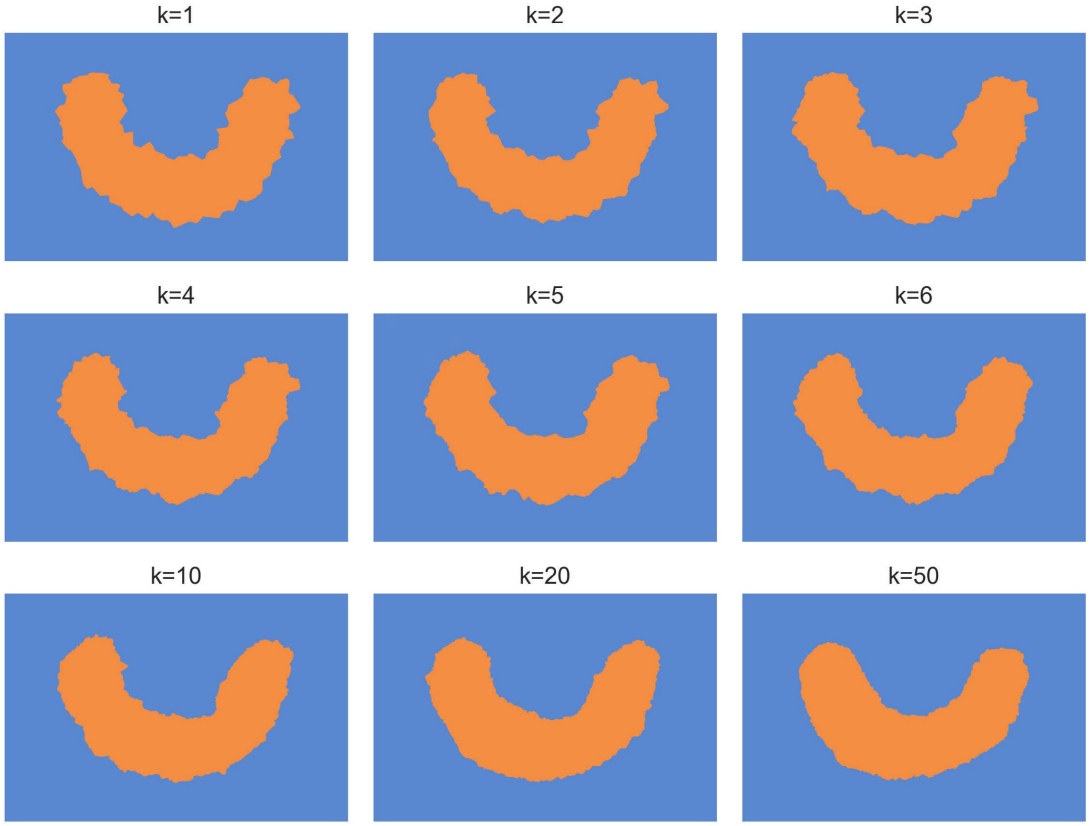


Figure 13.3: **Classifying all the points in the rectangle using kNN for different values of k. Notice how rough the edges are for low values of k, and how they smooth out as k increases. The k values in the top two rows go up by 1 with each image. The bottom row uses much bigger increments.**

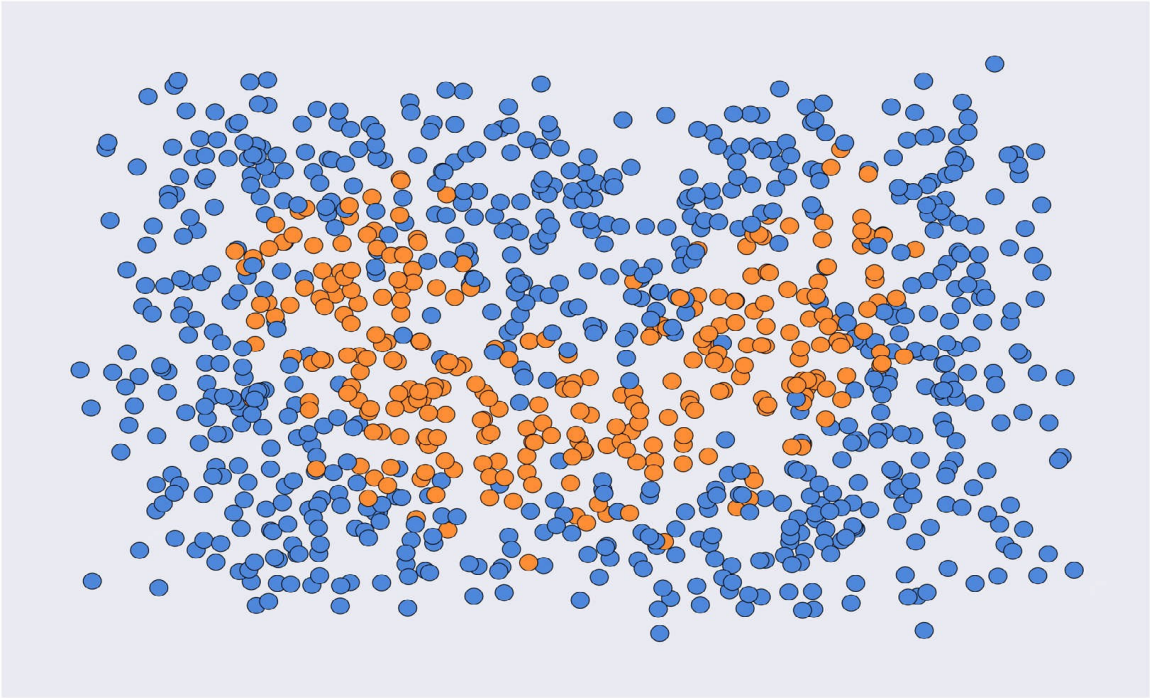


Figure 13.4: **A noisy version of the smile dataset.**

We're using the same number of points as in Figure 13.2, but we've added some 2D noise to jitter the locations of the samples after they were assigned to the blue and orange categories.

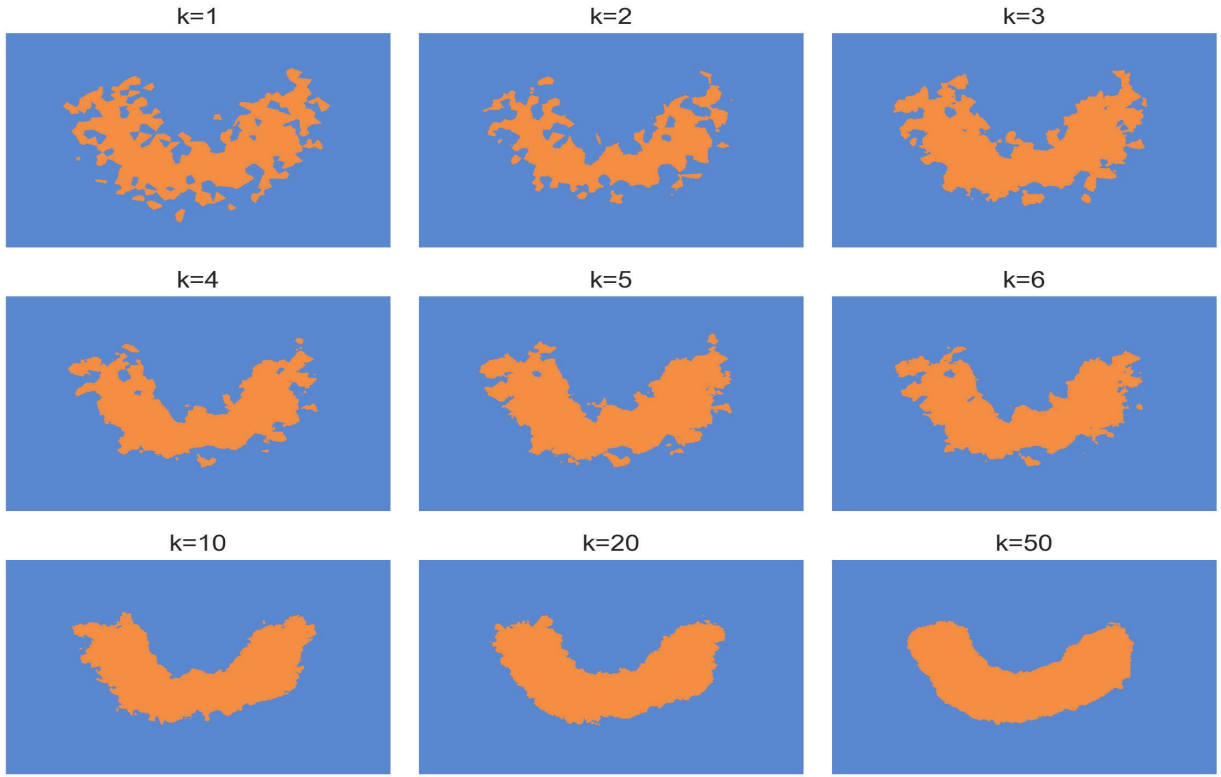


Figure 13.5: **Using kNN to assign a category to points in the plane.**

Note how much the algorithm overfits for low values of k. As the number of neighbors increases, the edges smooth out considerably.

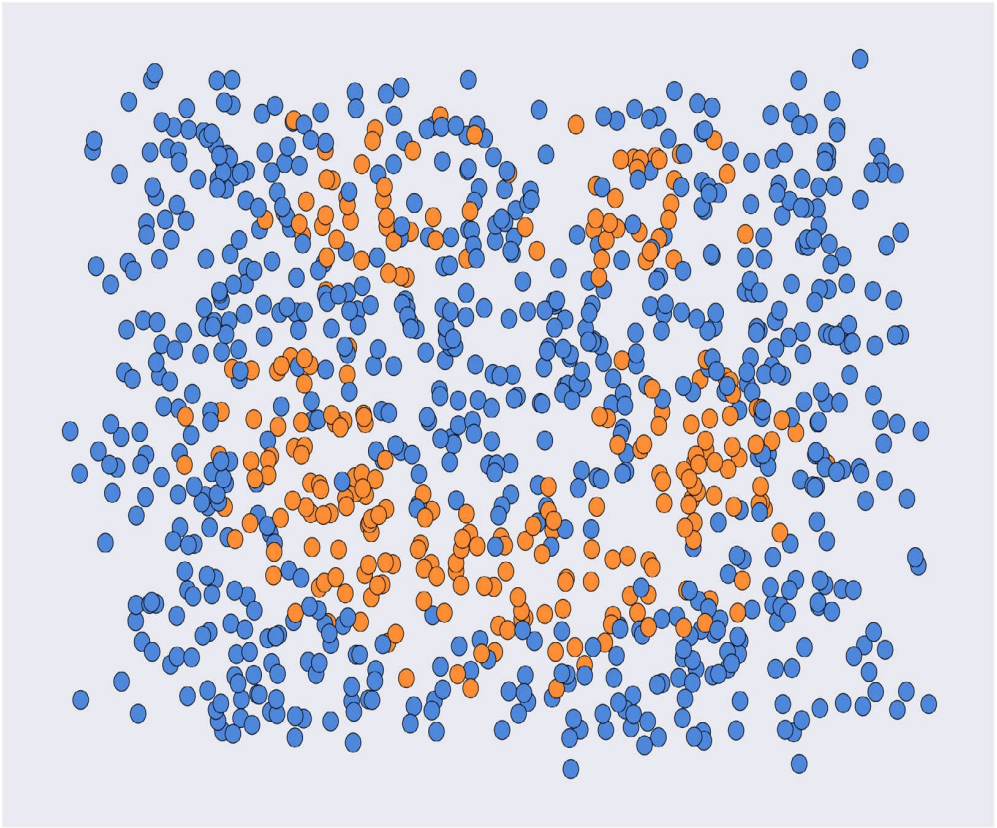


Figure 13.6: **Adding two eyes to our smile database, then adding noise to jitter the sample locations.**

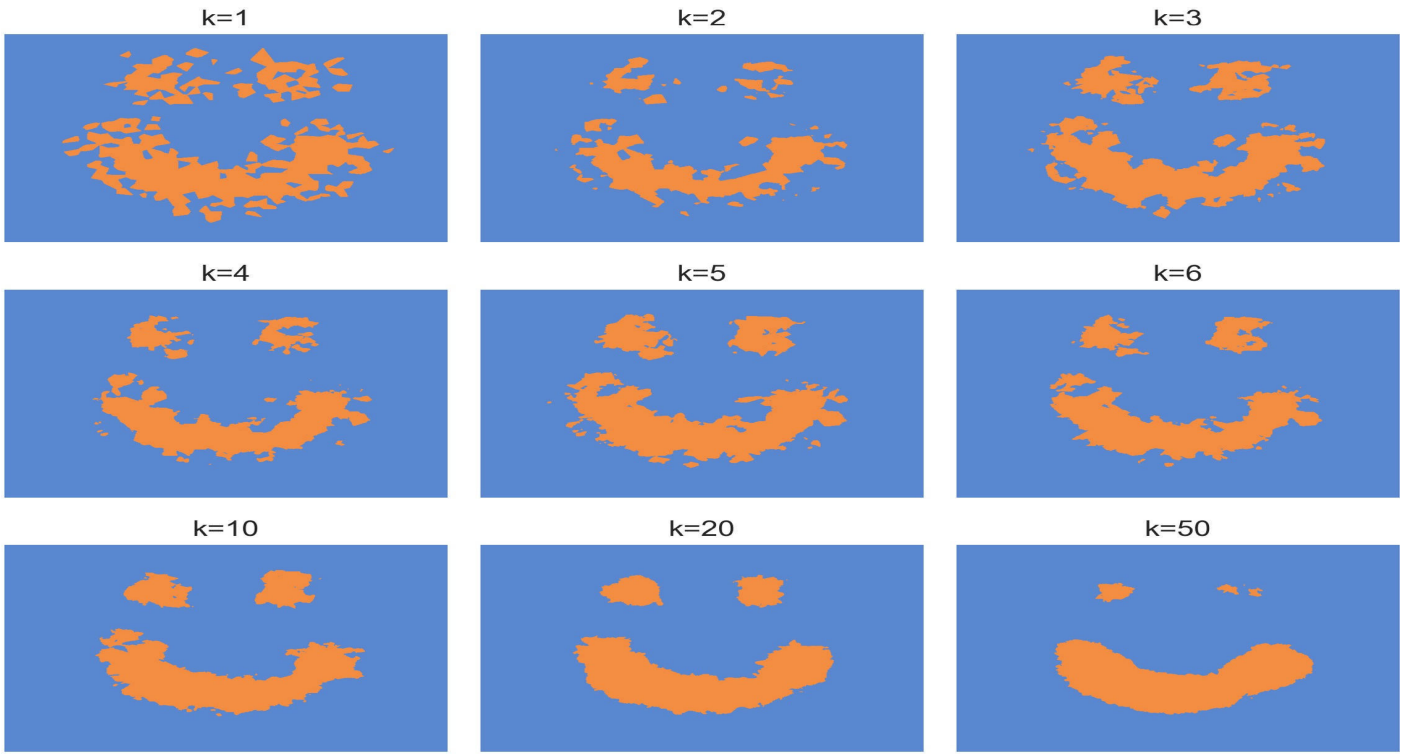


Figure 13.7: **kNN doesn't create boundaries between clusters of samples, so it works even when a class is broken up into pieces.**

Algorithm overfits at low values of k.

When k=50, the eyes are disappearing.

That's because in the neighborhood of the eyes, when we test for a large number of neighbors, the blue points will almost always dominate over orange.

If we increased k further the eyes would disappear completely.

Support Vector Machines (SVMs)

Our next classifier takes a very different approach.

Find an explicit boundary between different sets of samples.

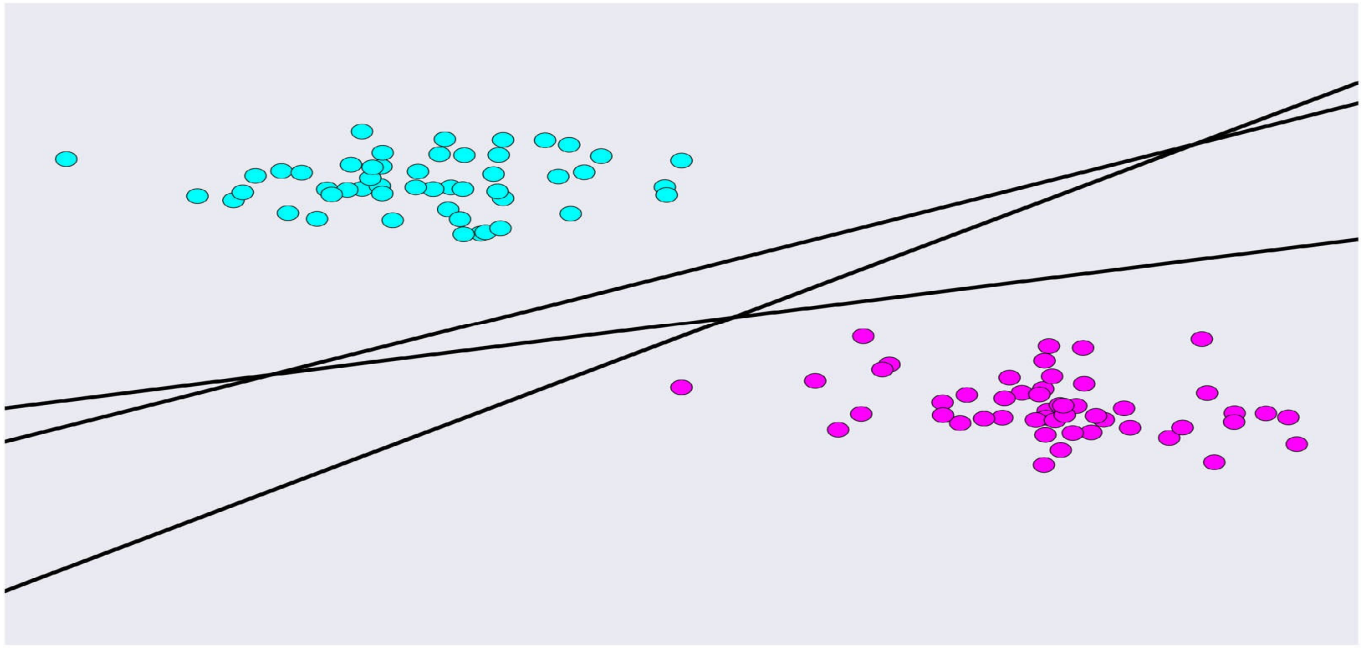
Find a boundary between these clusters.

Use a straight line.

There are lots of lines that will split the two groups.

Three candidates are shown in Figure 13.9.

Figure 13.8: **Our starting dataset consists of two blobs of 2D samples**



Q: Which line is best?

Figure 13.9: **Three of the infinite number of lines that can separate our two clusters of samples.**

Answer: Prefer a line that is as far away as possible from both clusters.

That way any of these new points have their best chance of landing in the proper cluster.

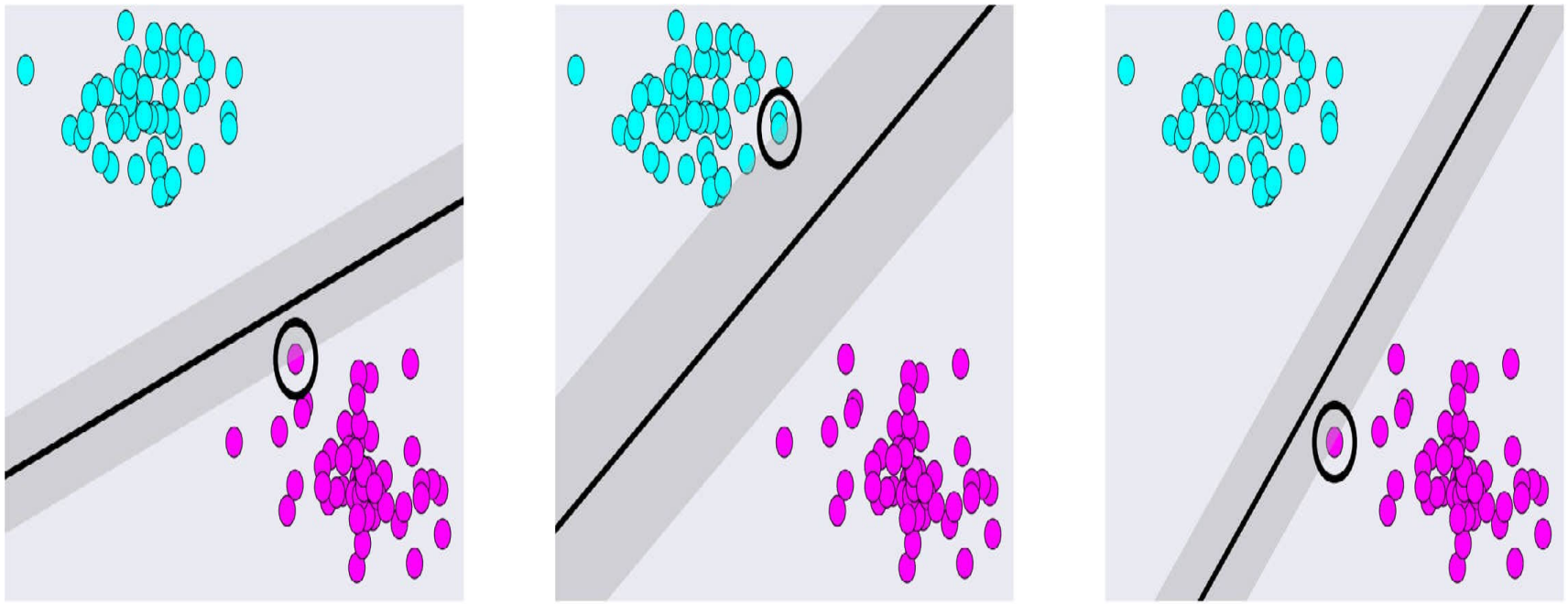


Figure 13.10: **We can assign a quality to each line by finding the distance from that line to the nearest data point.**

Here we've drawn a gray zone symmetrically around each line, at that distance, along with a circle around the point that's closest to the line.

Q: Which is the best line of the three lines?

A: The middle line

Q: Why?

A: Because it is least likely to make an error if a new point comes along that drifts into the zone between the clusters.

The algorithm called the **support vector machine**, or **SVM**, uses this idea

SVM finds the line that is farthest from all the points in both clusters

"machine" is a fanciful synonym of "algorithm."

This best possible line is in Figure 13.11.

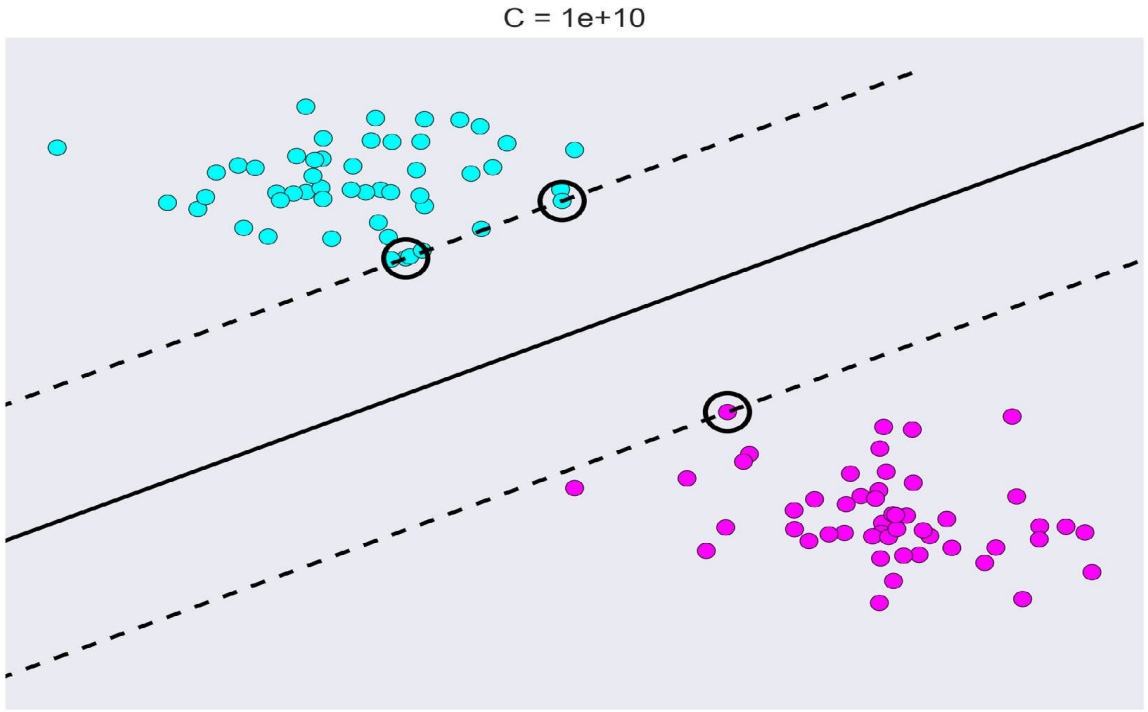


Figure 13.11: **SVM algorithm finds line having greatest distance from all samples.**

That line is shown in black.

Highlighted circles are support vectors,
Circles define both the line and zone around it, shown with dashes.

Figure 13.12: **A new set of data where the blobs overlap. What's the best straight line we can draw in this case?**

SVM algorithm uses a parameter *C*.

C defines how strict the algorithm is about letting points into the region between the margins.

We can think of *C* as "clearance."

Larger the clearance *C*, more the algorithm demands an empty zone around the line.

Smaller the clearance *C*, more points can appear in a zone around the line.

Figure 13.13 shows our overlapping data with a *C* value of 100,000 (or 1e+05 in scientific notation).

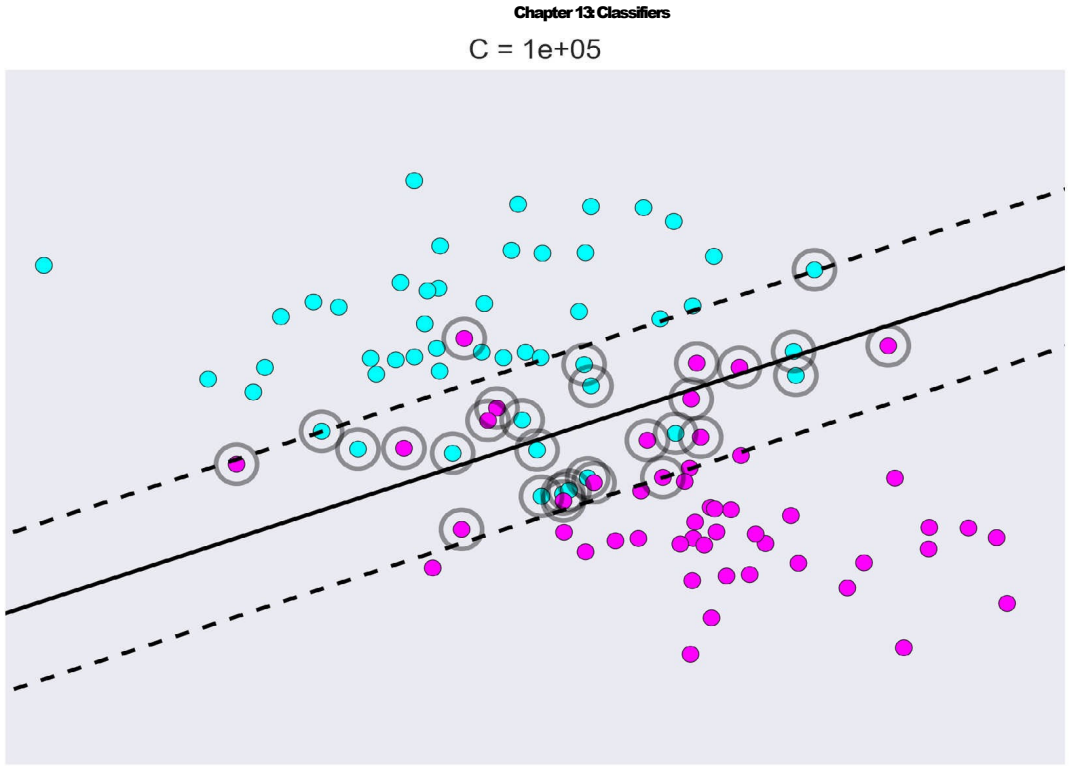


Figure 13.13: C **tells SVM how sensitive to be to points that can "intrude" into the zone around the line that's fit to the data.**

Smaller the value of C**, more points are allowed. Here we set** C **=100,000.**

Let's reduce *C* to 0.01.
Figure 13.14 shows that this lets many more points into region around line.

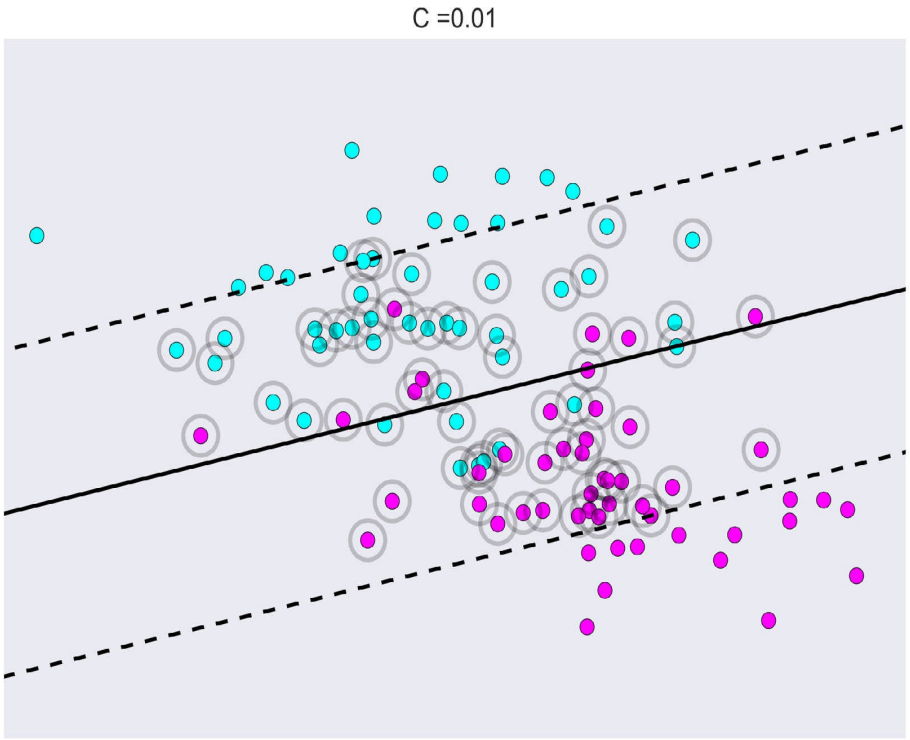
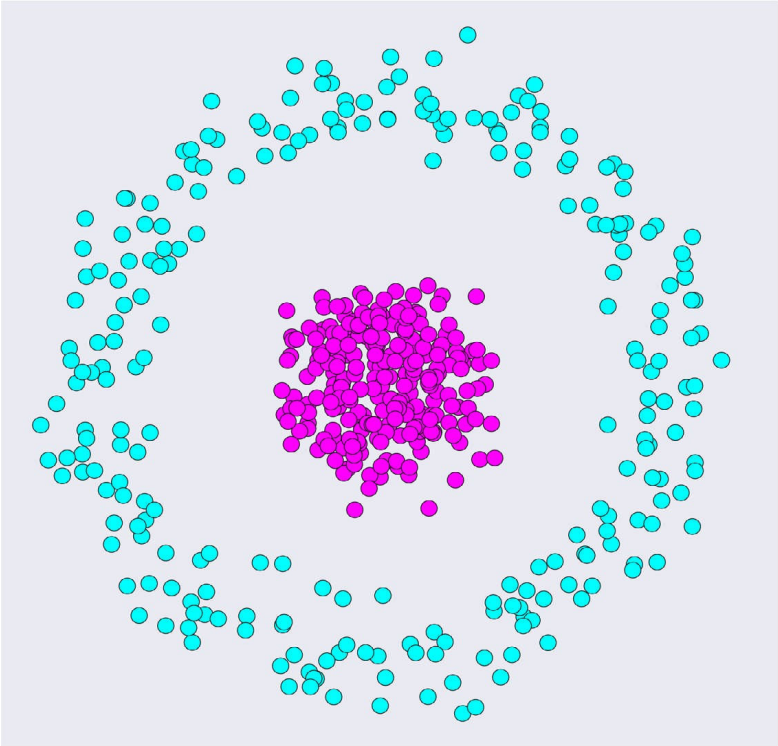


Figure 13.14: Reducing C to 0.01 lets in many more points.



Challenge problem for SVM

Data of Figure 13.15,

A blob of samples of one category surrounded by a ring of samples of another.

No line can separate these two sets.

Figure 13.15: **This set of samples is a challenge for basic SVM, since there's no straight line that can separate them.**

Solution:

Add a third dimension to each point by elevating it by an amount based on that point's distance from the center of the rings

Figure 13.16 shows the idea.

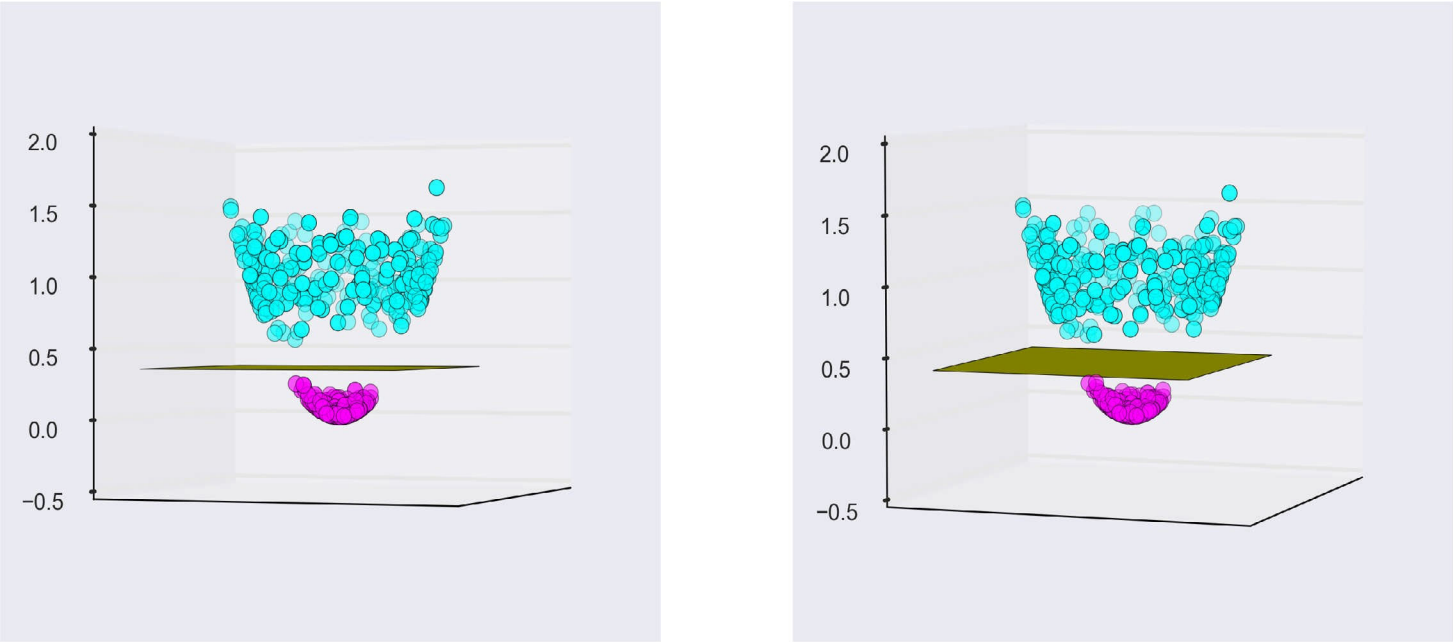


Figure 13.16: **If we push each point in Figure 13.15 upwards by an amount based on its distance from the center of the pink blob, we get two distinct clouds of points.**

It's now easy to place a plane between them.

Just as a line is the linear element in the plane, a plane is the linear element in space, so it's the sort of thing SVM can find.

These two images are two views of the same data, showing the points and the plane between them.

As we see in Figure 13.16, we can now draw a plane (the 2D version of a straight line) between the two sets.

In fact, we can use the very same idea of support vectors and margins as we did before to find the plane.

Figure 13.17 highlights the support vectors for the plane between the two clusters of points.

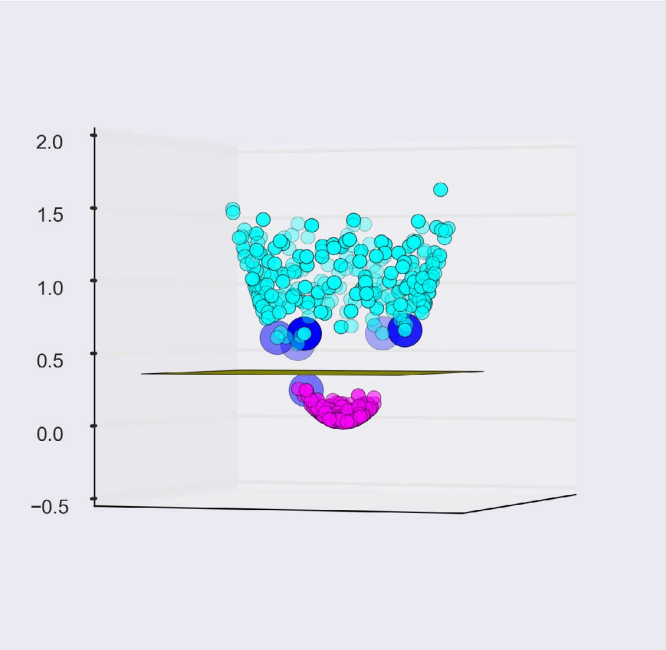


Figure 13.17: Support vectors for the plane

All points above the plane can be placed into one category, and all those below into the other.

Figure 13.18 shows support vectors

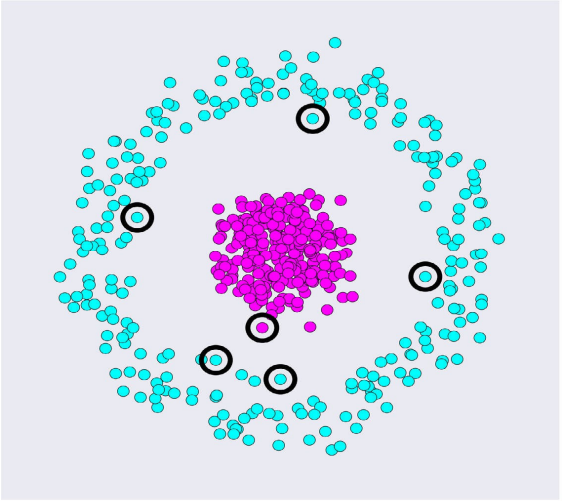


Figure 13.18: Looking down on Figure 13.17.

Figure 13.19 shows boundary created by the plane

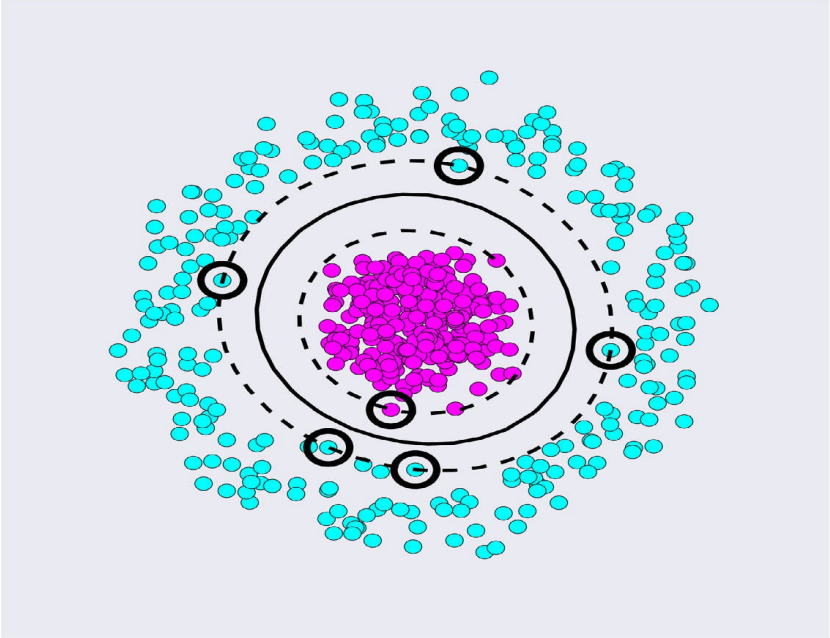


Figure 13.19: Intersection of curved shape we built from our data and the plane we placed in 3D is shown by the solid line.

Here we show the support vectors and the margin they create.

- There's a special way to do the mathematics that speeds things up significantly.
- The technique involves modifying a piece of math called the **kernel,** which forms the heart of the algorithm.
- Rewriting the SVM math to handle this extra work efficiently is called the **kernel trick**.
- The kernel trick finds the distances between transformed points without actually transforming them!
- That's a major efficiency boost.

Decision Trees

We can illustrate the idea with the familiar parlor game called *20 Questions*.

In this game, one player (the chooser) thinks of a specific *target* object, which is often a "person, place, or thing." The other player (the guesser) asks a series of yes/no questions.

If the guesser can correctly identify the target in 20 questions or less, they win.

A typical structure is shown in Figure 13.20

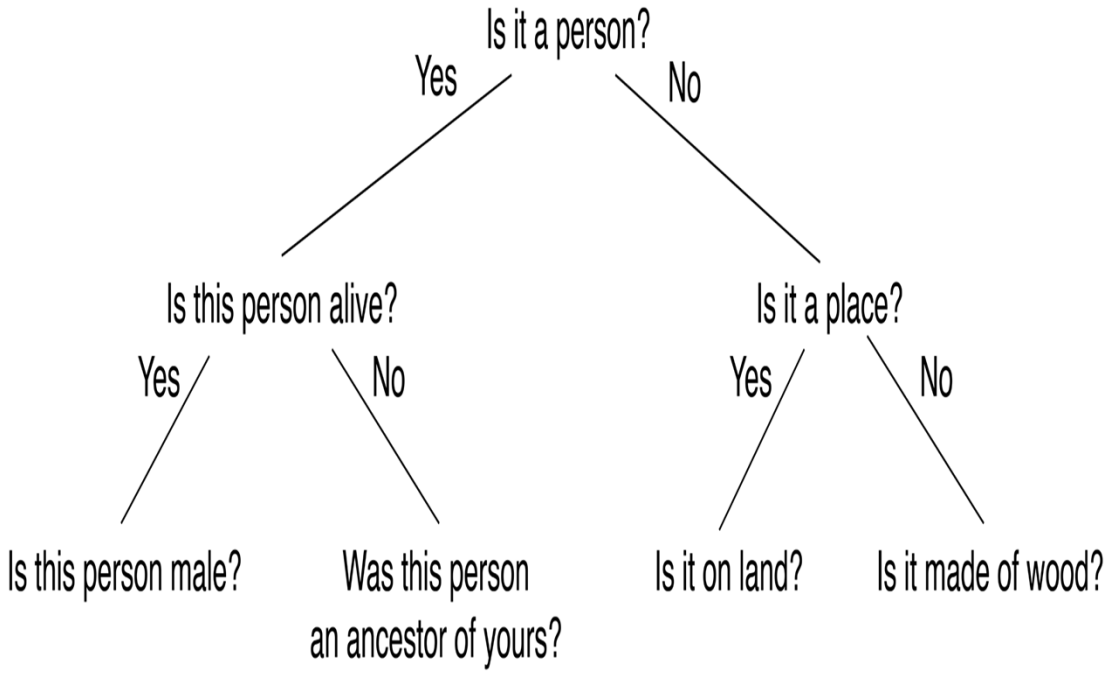


Figure 13.20: A tree for playing "20 questions."

Note that after each decision there are exactly two choices, one each for "yes" and "no."

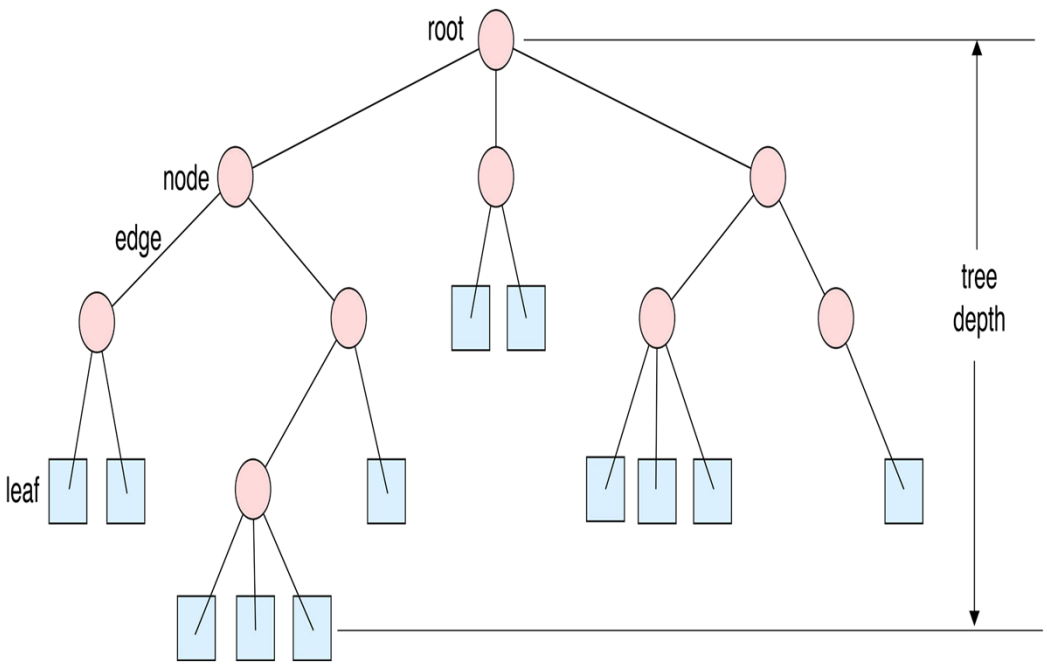


Figure 13.21: Each circle is a node.
Node at the top has a special name: the root.
Lines that join nodes are called edges.
Tree's depth is the number of nodes from its root to its farthest child.
Here, the depth is 4.

Structure in Figure 13.20 a **tree**

Node: Each splitting point in the tree is a **node**, and

Branch: Each line connecting nodes is a **link** or **branch**.

Node at the top is the **root**,

Nodes at the bottom are **leaves**, or **terminal nodes**.

Nodes between the root and the leaves are called **internal nodes** or **decision nodes**.

If a tree has a perfectly symmetrical shape, the tree is **balanced**, otherwise it's **unbalanced.**

Every node has a **depth**, which is a number that gives the smallest number of nodes we must go through to reach the root.

The root has a depth of 0, the nodes immediately below it have a depth of 1, and so on.

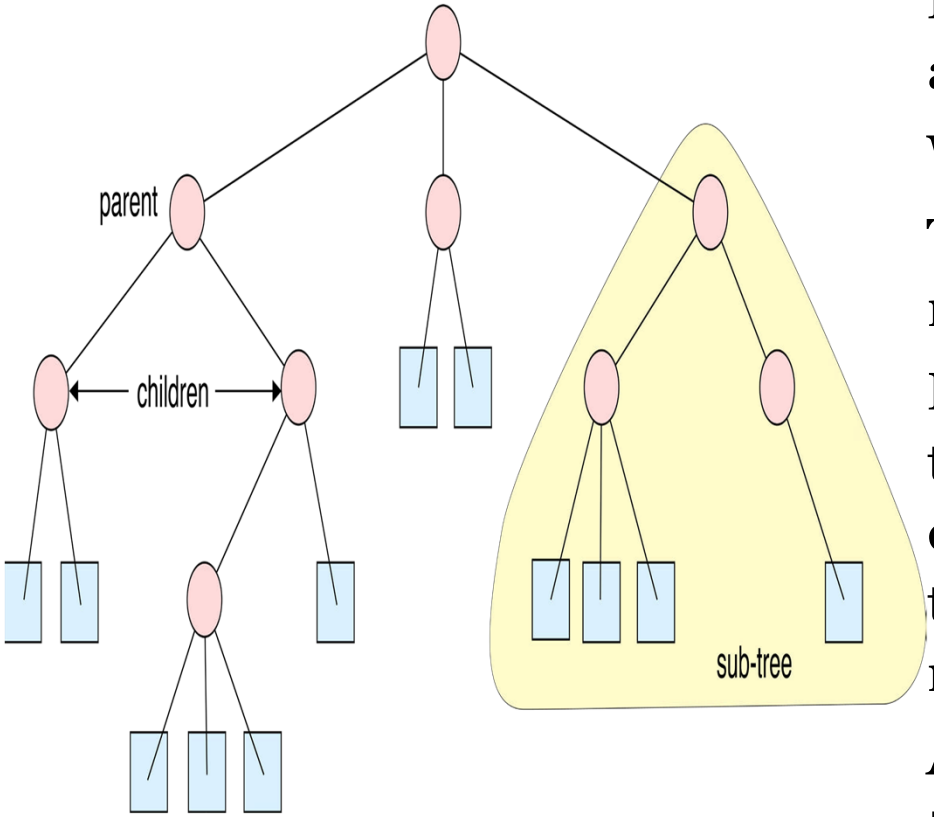


Figure 13.22: **Some more terms applied to trees. A node is a parent if there are nodes immediately below it. Those nodes are called children of that parent. A subtree is a piece of the tree that we want to focus on.**

Every node (except the root) has a node above it.

We call this the **parent** of that node.

The nodes immediately below a parent node are its **children**.

Distinguish between **immediate children** that are directly connected to a parent, and **distant children** that are connected to the parent through a sequence of other nodes.

A node and all of its children taken together are called a **branch** or **sub-tree**. Nodes that share the same immediate parent are called **siblings**.

Figure 13.22 shows some of these ideas graphically.

The *20 Questions* tree is that it is **binary**: Every parent node has exactly two children.

This is a particularly easy kind of tree for a computer to build and use.

If some nodes have more than two children, tree overall is **bushy**.

We can always convert a bushy tree to a binary tree if we want.

An example of a bushy tree that tries to guess the month of someone's birthday is shown in Figure 13.23(a), and the corresponding binary tree is shown in Figure 13.23(b).

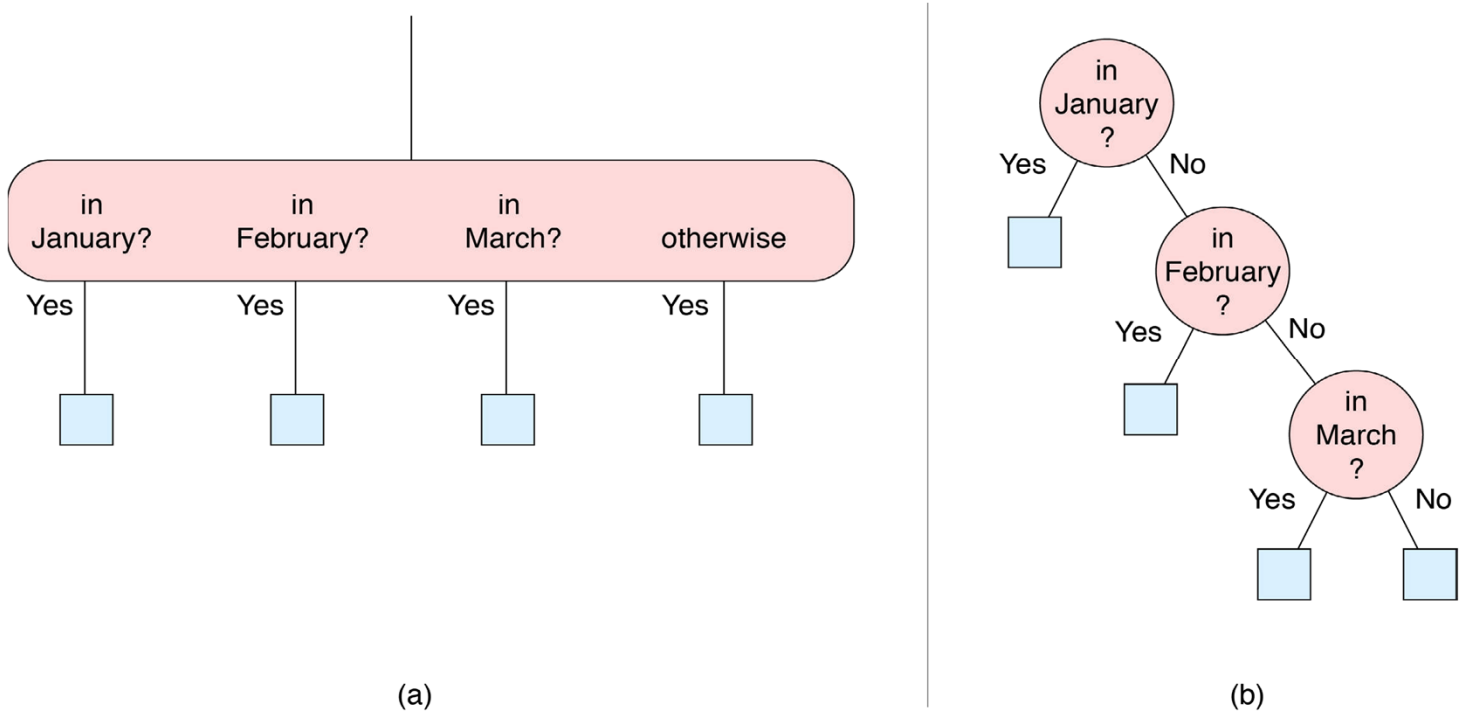


Figure 13.23: Imagine a tree where there are many children after a node, each with its own test.

We can always turn this kind of bushy tree into a binary tree that has just one yes/no question at every node.

We build the tree during training.
Look at the tree building and evaluation process.

The root and all parent nodes contain tests based on the features of a sample.

The leaf nodes contain the training samples themselves.

When a new sample arrives, we start at the root and descend, following the branches based on the tests evaluated on the features of this sample, as in Figure 13.23.

When we reach a leaf node, we test to see if the new training sample has the same category as all the other samples in that leaf.

If so, we add the sample to the leaf and we're done.

Otherwise, we **split** the node, and come up with some kind of test based on the features that will distinguish between this sample and the previous samples in the node.

That test gets saved with the node, and we create at two children, moving each sample into the appropriate child, as in Figure 13.25.

When we're done with training, evaluating new samples is easy.

We just start at the root and work our way down the tree, following branches based on the sample's results when we apply its features to the test at each node.

When we land in a leaf, we report that the sample belongs to the category of the objects in that leaf.

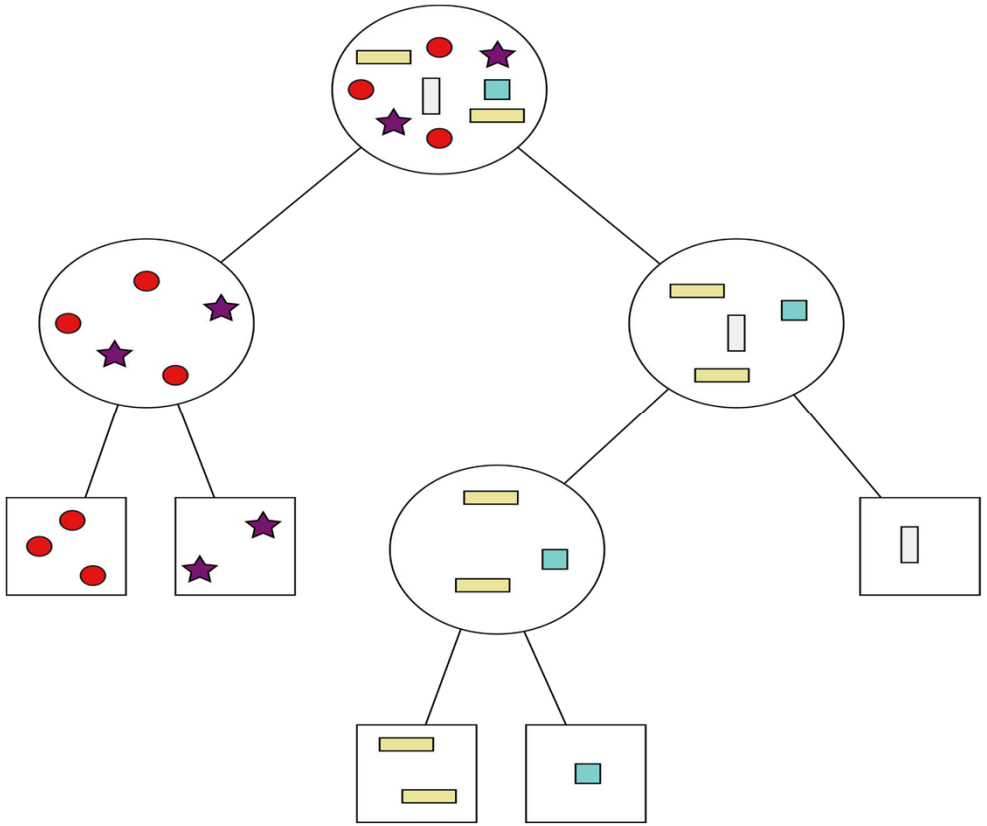


Figure 13.24: **A categorical decision tree. From the many samples at the top, each with its own category, we apply a test at each node until we produce leaves that have only one category of samples each.**

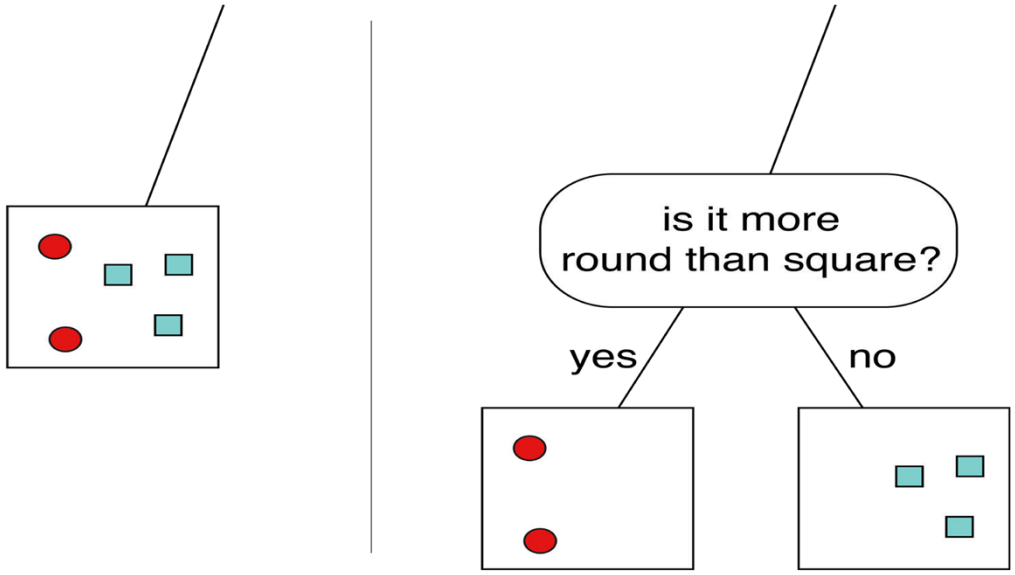


Figure 13.25: **Splitting a node.**

Left, we see a leaf node that has both round and square samples.

Right, we decide to turn that leaf into a regular node by replacing it with a test.

The test creates two leaf nodes, each of which has more uniform contents than the leaf did before it was split.

Recurrent Neural Networks

When we work with sequential data, we need to use special care to preserve and make the best use of the order in which the data arrives.

We'll see a variety of special architectures for just that purpose.

Introduction

Introduction to sequences

- So far, we considered every sample as a standalone entity, unrelated to any other samples.

- This makes sense for things like photographs.
  - Example: If we're classifying an image and decide that we're looking at a cat, it doesn't matter if the image before or after this one was a dog, or squirrel, or airplane. The images are independent of each other.
- Example: If an image is a frame of a movie, then it can be helpful to look at it in the **context** of the other images that precede and follow it.
- Example: Video of a ball flying through the air. We can use a CNN to identify the ball's position, but the CNN can't tell whether the ball is rising or falling.
- That requires knowing what came before.
- In other words, we need some context.
- **Use the RNN of this lecture, which are context based!**

Sequences

Q: What is a sequence?

A: When data is arranged so that each piece has some kind of relationship the pieces that come before and after, we call that a **sequence**.

- In this lecture, we'll look at how we can work with these kinds of **sequences** to derive meaning from them.

Q: Give some examples of sequences:

A: Series of readings of

- Noon time temperature of a specific place over a period of days,
- time of high tide each day
- price of a stock at the close of trading.

Sequences – More Examples

- Lots of interesting **sequences** in the world.
  - Stock prices over a series of days,
  - Notes that make up a song,
  - frames making up a movie, or
  - words from a language.
- We ask questions about all of these sequences, such as
  - What will the stock price be tomorrow?
  - Translation of a string of words into another language, or
  - Is this book written by the same author as is this another book?
- A traditional problem with Neural Networks seen so far is that they have no **memory**
  - So, they're poor at using **context**
  - Context is important to answer questions about sequences.
- **Example**: if we're translating a piece of speech, we might be considering just one word at a time.
- It's context that lets us consider the words that came before and after, so we can make the best translation.

Sequences – Language Example

- An important type of sequence is **language**.
- Language as a stream of letters or words, or larger units like sentences and paragraphs.
- Making sense of these sequences is easier if we can look at the entire collection.
- "He told me I could have one of these strawberries,"
  - Then, we look at previous stream of words to see who is "he"
- "She said to wait over there," requires even more **context**, or **surrounding information** to be understood.

Applications - Sequences

If we can train a network on sequences, we can then generate

- Poem or a story, perhaps even in the voice of a famous writer
- Make new scripts for TV sitcoms.
- Music: We can generate a whole song.
    - Single melody, Polyphonic melody, or a complex song with melody and chords.
- Lyrics: We can create song lyrics.
    - Pop music, folk music, rap lyrics, or country music lyrics
- Speech to text systems
- Generating captions for images and videos.

Interesting websites for RNN - LSTM

General: http://people.idsia.ch/~juergen/rnn.html
Music: http://people.idsia.ch/~juergen/blues/index.html

LSTM Walkthrough:
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Harry Potter: https://medium.com/deep-writing/harry-potter-written-by-artificial-intelligence-8a9431803da6

Lyrics: https://medium.com/deep-writing/hamilton-a-new-song-written-by-ai-daf164d9e616

Rap music – singing with lyrics:
https://soundcloud.com/rapping_neural_network/networks-with-attitude

Let's Start !

What's in this Lecture?

- This lecture presents a **learning architecture** that is explicitly designed to learn from **sequential data**.
- This architecture is called a **recurrent neural network**, or simply an **RNN**.
- RNN is often called **LSTM**, because that specific technique is a widely popular choice of implementation.
- In this lecture, we'll see what makes an RNN special, how to build one, and how to use it.

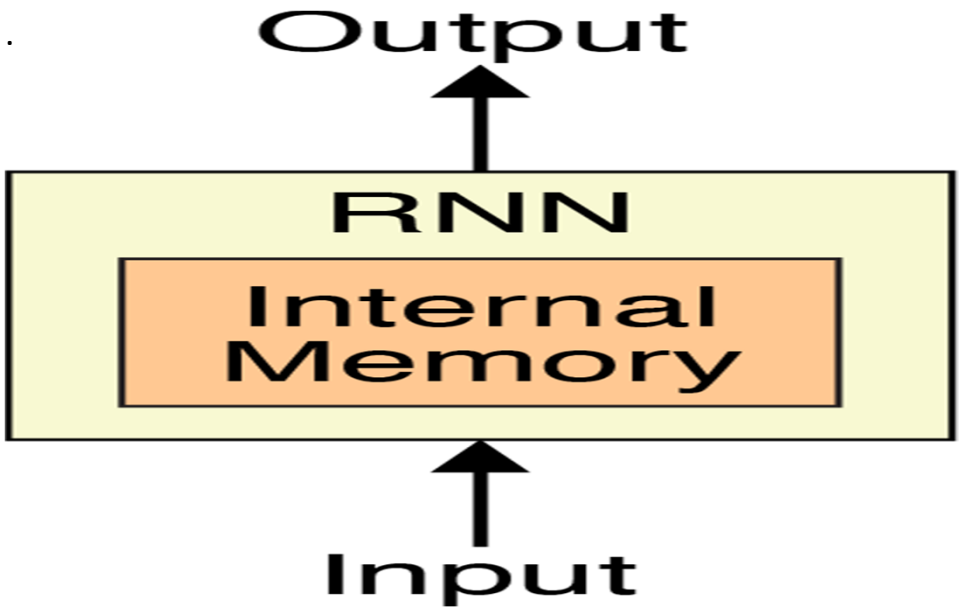
Firstly, What is Recurrence & RNN ?

- Recurrence refers to a repeating or **recurring** action.
- A recurrent cell repeats a given operation over and over, so it's recurring.
- Recurrent networks provide us with a flexible way to add memory and context to our networks.
- RNNs can do all of the things we asked above, and many others.
- They're used for activities ranging from language translation to automatic photo captioning, and even the generation of new prose in the style of known authors.

Recurrent Unit/Cell

A a basic artificial neuron does not have memory.

We can address the lack of memory in Neuron by replacing it with a more complex bit of processing called a **recurrent unit** or **recurrent cell**.



- **Figure 22.5:** A computational unit that is capable of remembering some internal information (state) over time.
- We call the internal information as ***state*** (of unit).
- RNN uses the internal memory to process its input into output.
- The internal memory is able to change with every input, even after training is complete.
- Unit is special as it **saves** information.
- Each time a unit processes an input, it uses some of what it learned from previous inputs.

Recurrent Layers

- RNN unit/cell is mostly just a bunch of neurons and activation functions, but they're wired up in a specific way and treated as a single unit.

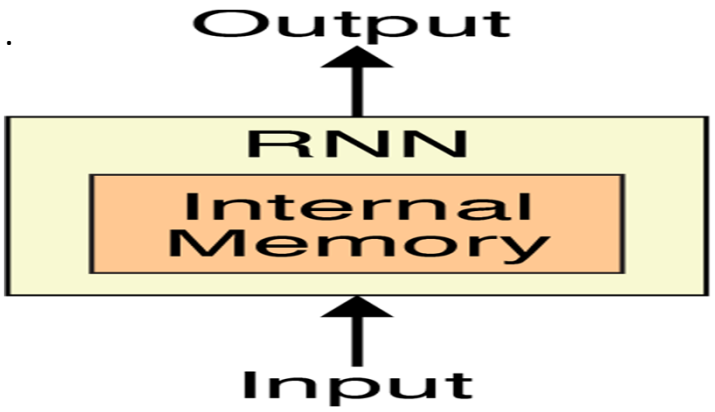
- There's also a bit of memory in there, which we'll see later.

- We just pop this unit/cell into our network as a layer like any other.

- We can build deep-learning networks using a mixture of standard layers, and layers made up of recurrent cells.

- If the recurrent cells are an important part of the network, we call it a **Recurrent Neural Network**, or **RNN**.

What is a "State" ?



- **State** = **internal information** (in the unit's memory)
- State = Information we save and read in this memory.
- "State" = "Remembered data" that describes the situation or configuration.
- We can say "save the state" and "read the state"
- "State" can be written to a file, saved on a USB, or transmitted over a network.
- But in RNNs, we keep the **state** inside the recurrent unit.

Robot Example

A system that reads and writes its state over time

(the way RNN does)

- A robot that fixes cell phones.
- **Robot** sits at a counter with a **parts cabinet** next to it.
- If we place a broken cell phone in front of the robot, it will analyze the phone to work out what's wrong, and then use its tools and the parts in the cabinet to fix it.
- It will take some parts out of the parts supply.
- It might even add some parts back in.
- For instance, if it can replace a complicated bit of the phone with something simpler, then any pieces that were removed but are still functional could be added to the robot's supply of parts.
- The robot's **parts supply** is its **state**, or **persistent information**.
- When it starts each repair, it considers what parts are **available** in **the state** to help decide how to fix the phone.
- When the repair is done, some parts have probably been removed, and others might have been added, producing a **new version of the state**.
- So this one supply of parts changes after each repair, and becomes the starting supply of parts for the next.

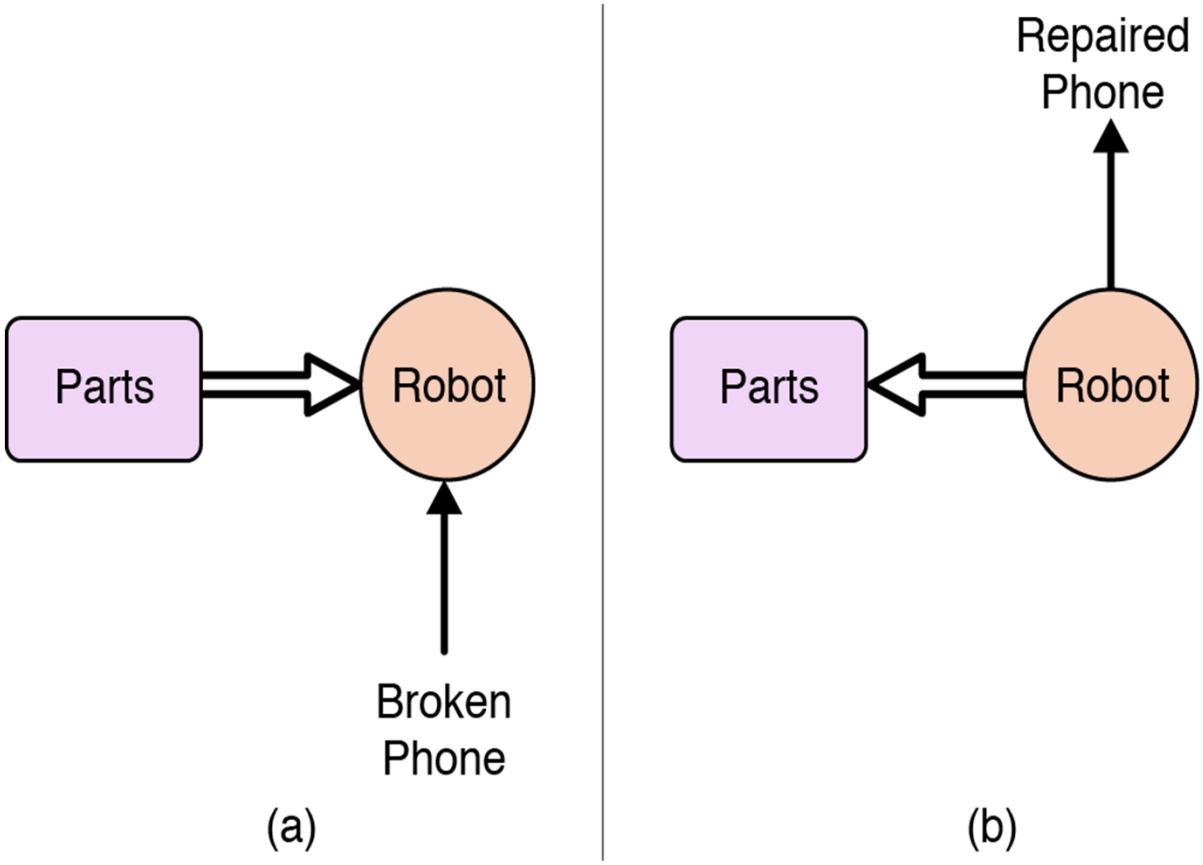


Figure 22.6: Robot that fixes cell phones with the help of a box of parts.

(a) Robot is given a broken phone and a box of parts.

(b) Robot returns the repaired phone, and it updates the contents of the parts box to match the parts that were removed and others that may have been added.

Note for figure:
Single line with arrow - flow of *information* (in this case, information about the phone)
Outline arrow- operations that involve the *state*

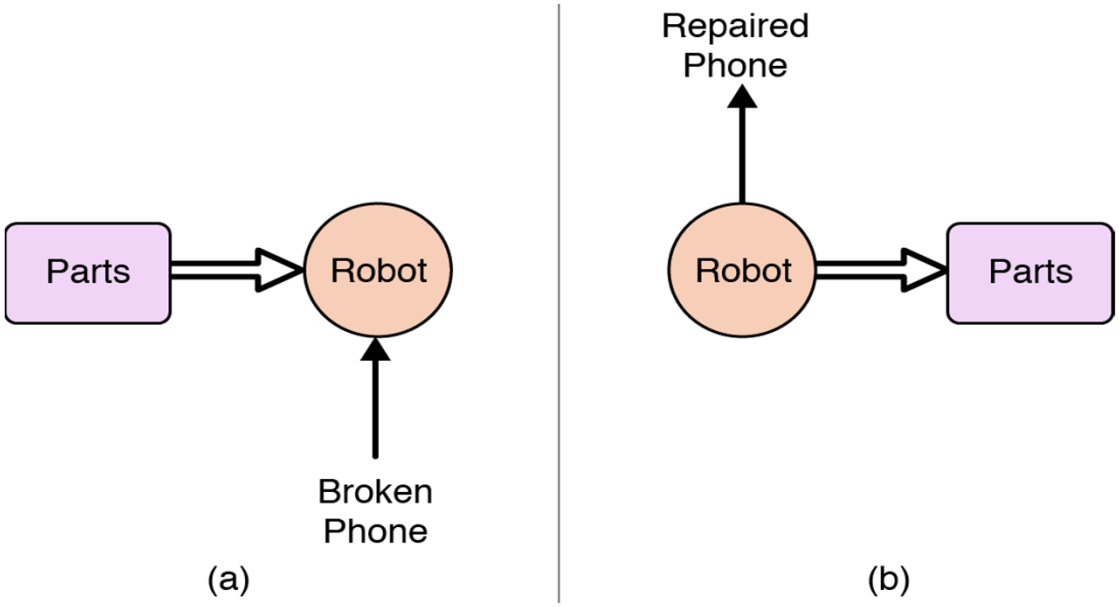


Figure 22.7: A cosmetic change to Figure 22.6 so that all arrows point either up or to the right.

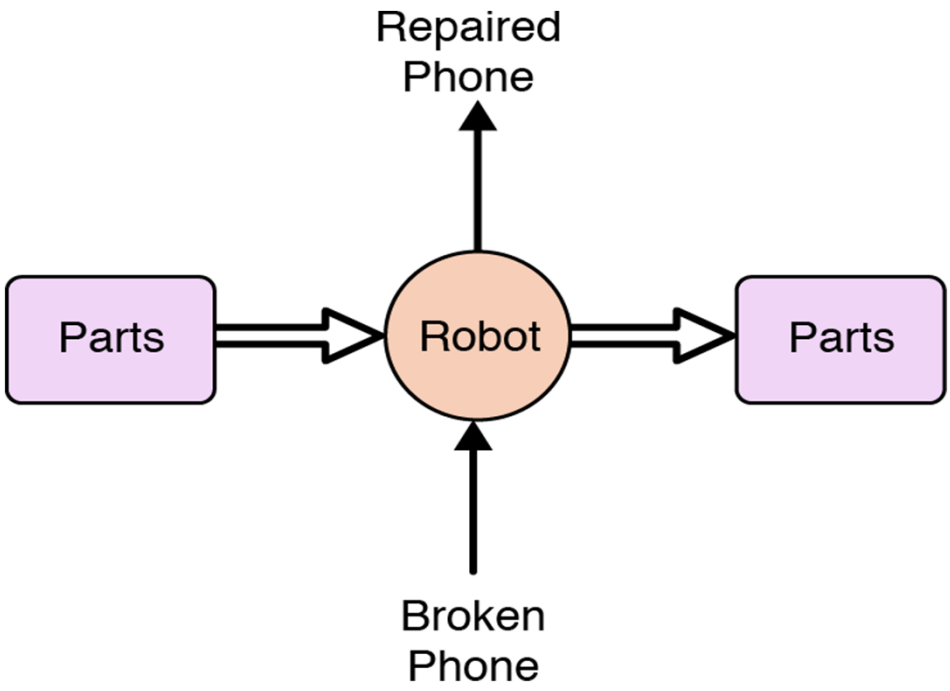


Figure 22.8: Combining the two steps in Figure 22.7 we can make a tighter diagram.

Note: This represents two steps in time, and only one box of parts.

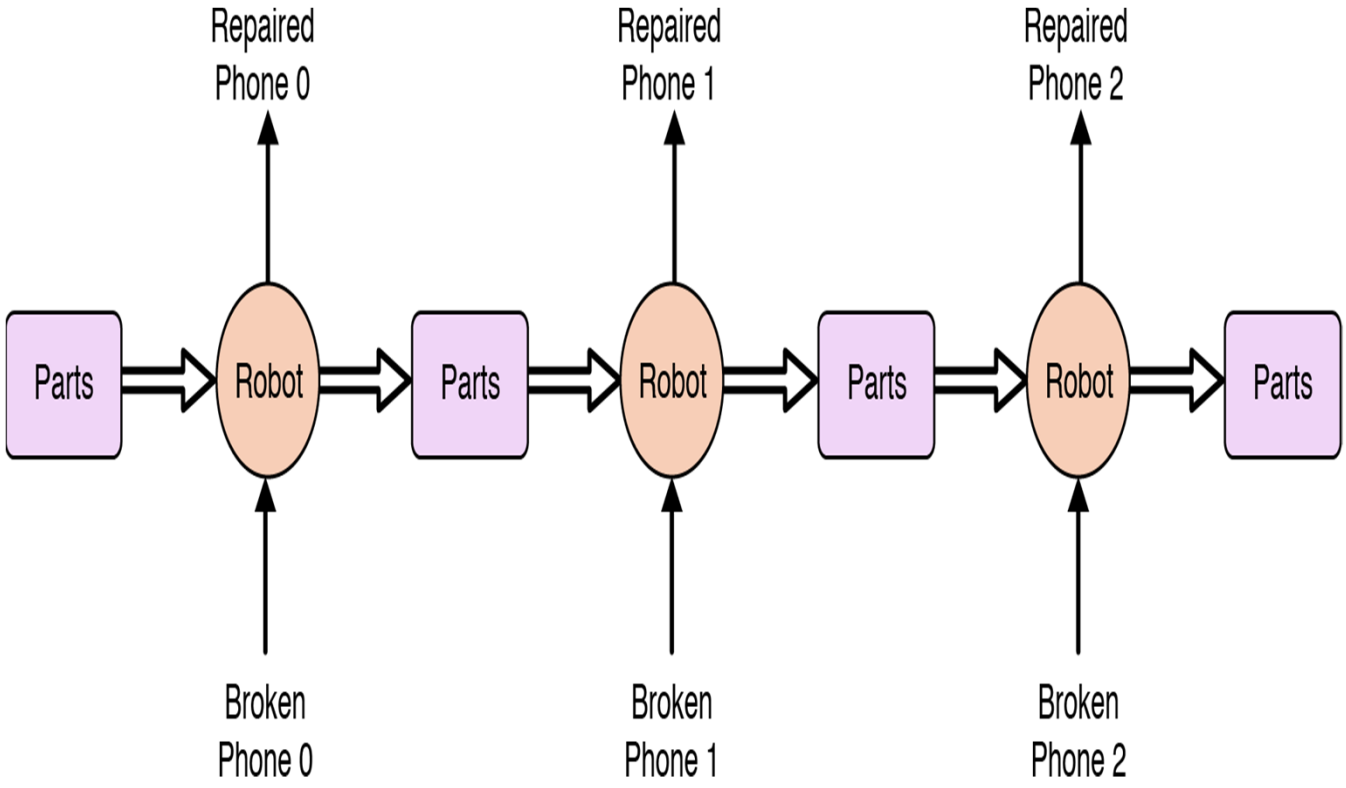


Figure 22.9: **Repairing multiple phones using the diagram of Figure 22.8.**

There's only one robot and only one box of parts, but here we can see that in each sequential repair, the robot uses the parts box in the condition it was in at the end of the previous repair.

Note: We're showing multiple moments of time, like a multiple-exposure snapshot

Analogy between RNN and Robot Example

- We've basically described how an RNN unit works.
- Let's express the process we've just seen using the language of RNNs.
- We can re-draw our repeating repair diagram of Figure 22.9 with this new language in Figure 22.10.

- This is the basic structure of a simple RNN.

- Note that when there's no chance of confusion between individual RNN units/cells and RNN-based networks, we sometimes refer to an RNN unit/Cell as just an RNN.
- Our robot will be replaced by a single RNN **unit**, also called an **RNN cell**.
- This RNN unit or cell is a little bundle of some artificial neurons and memory.
- It accepts an input and provides an output, but most importantly, it has the ability to read and write to its **persistent internal memory** (like the **parts box**).

- This persistent memory is the unit's **state**.

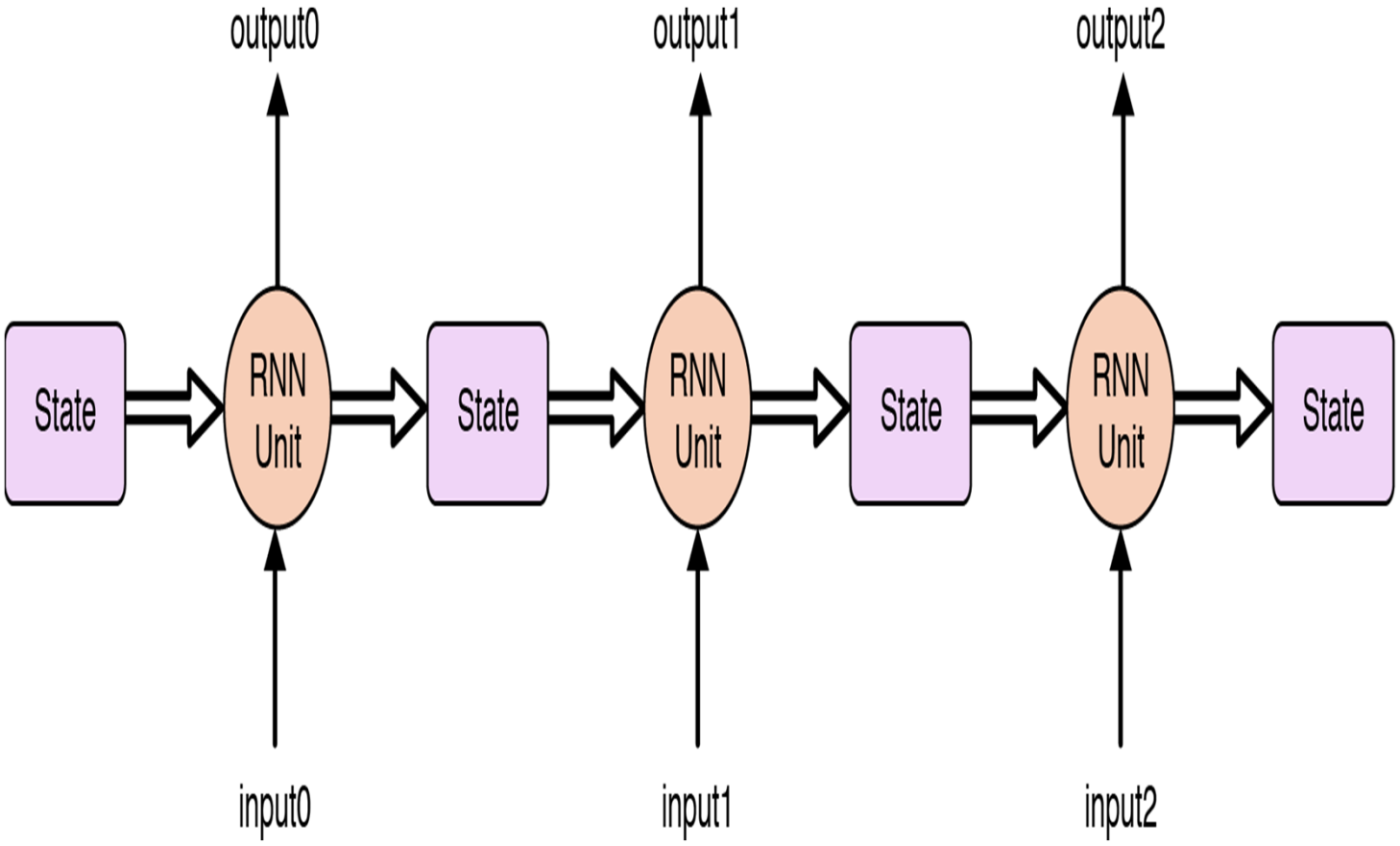


Figure 22.10: A simple RNN.

There is only one RNN Unit, and only one State.

They are re-used by each sequential input, producing sequential outputs.

Managing and Interpreting State Values

- RNN manages the state for us automatically, reading and writing values.
- State information is stored inside the RNN unit itself.
- "State" is just **a list** of floating-point numbers.
- We specify the size of this list when we create the unit.
- We can use lengths of 3 or 5 or lengths of several hundred in larger systems.
- This is a number we pick based on intuition, experience, and usually some experimentation.

A Question: How do you interpret the numbers in the state?

Q: What do these numbers in the state represent? What is exactly remembered and forgotten?

A: The RNN cell "decides" what the state's contents should mean, and how to manage those numbers.

- *So that the network as a whole ultimately produces the answers we're asking it for.*

Most of the time, we don't need to worry much about interpreting these state values.

Organizing Inputs for RNN

Input to RNN unit: Samples.
Each sample is composed of features.
Each feature is made up of **time steps**.

Q: How to organize that data ?

A: Keras Library uses convention of assigning directions (away, down, right) to (samples, time steps, features).

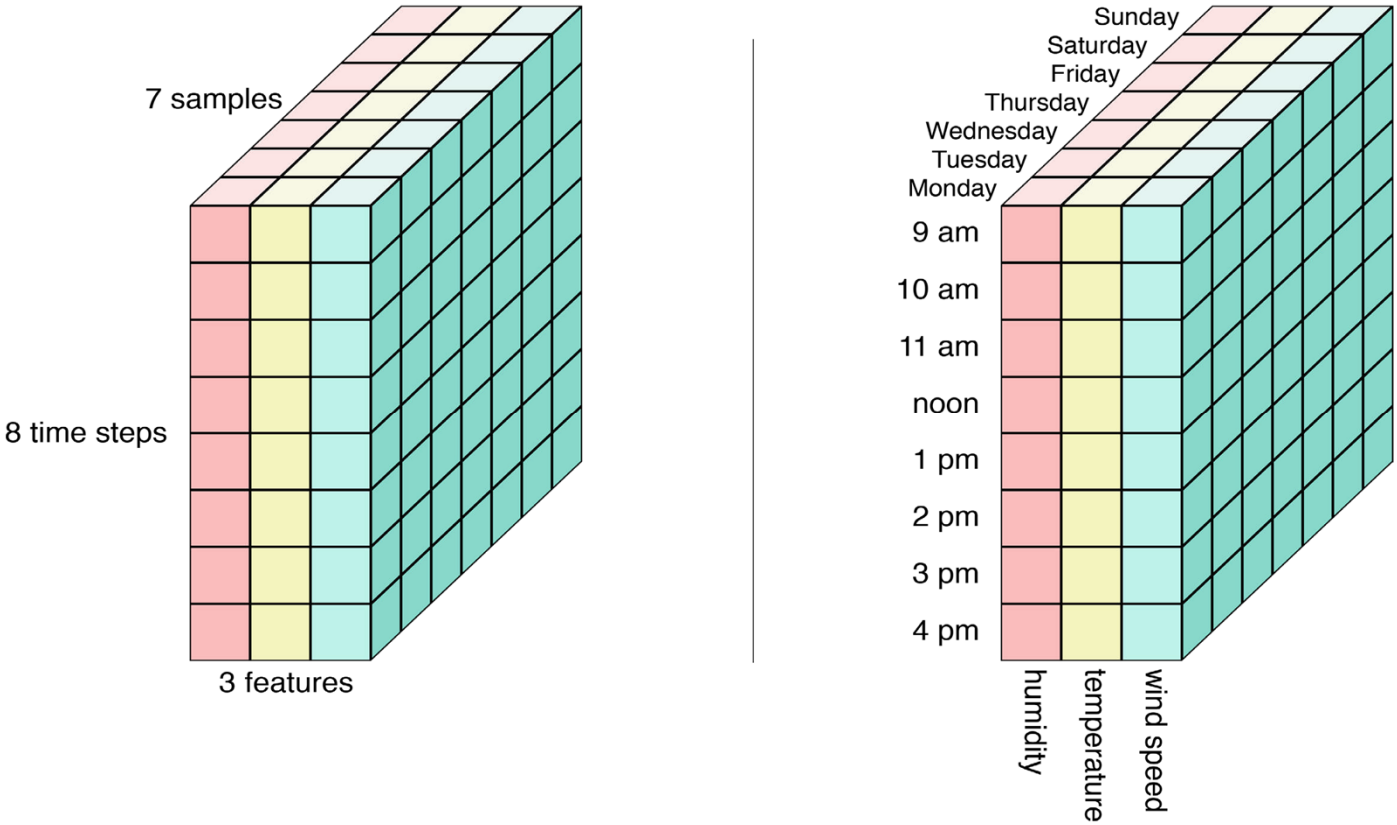


Figure 22.21: Suppose we collect our 8 measurements of 3 types of data on 7 consecutive days.

We can arrange our data as 7 samples, each composed of 3 features, with 8 time steps per feature.

LSTM

- More popular approach uses a kind of RNN cell with the name of **Long Short-Term Memory**, or **LSTM**.

- LSTM is so popular that when people today speak of an RNN unit generally, they often mean an LSTM.

- Let's see now what is an LSTM

Gates

- A key mechanism in LSTM is a **gate**. First, lets understand it.
- Fig 22.24 shows a physical gate controlling flow of water out of a pipe

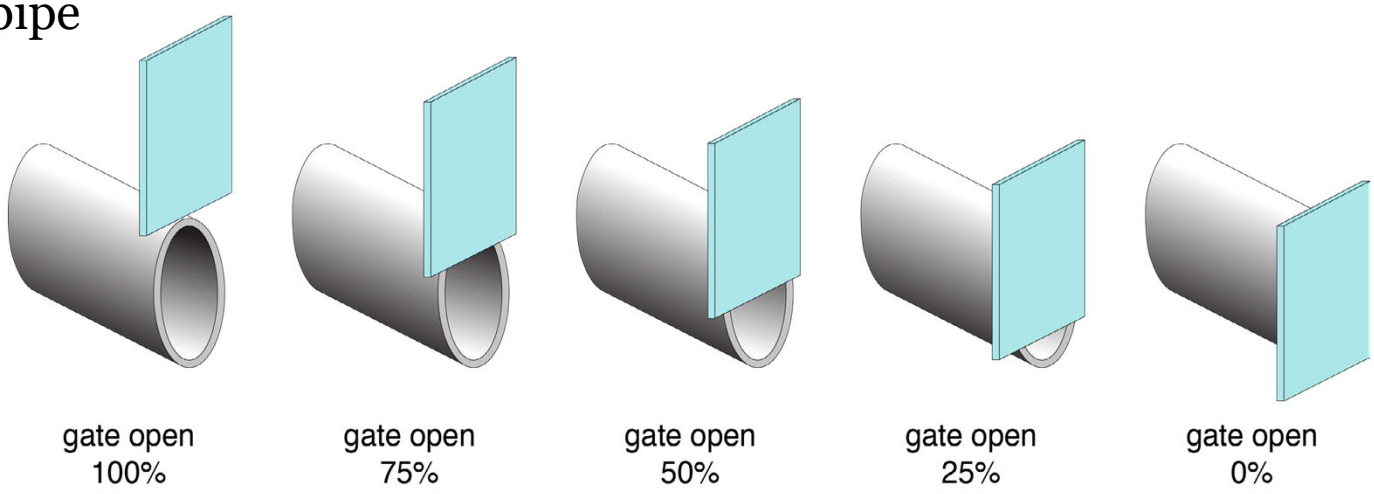


Figure 22.24: Picturing a gate over a water pipe.

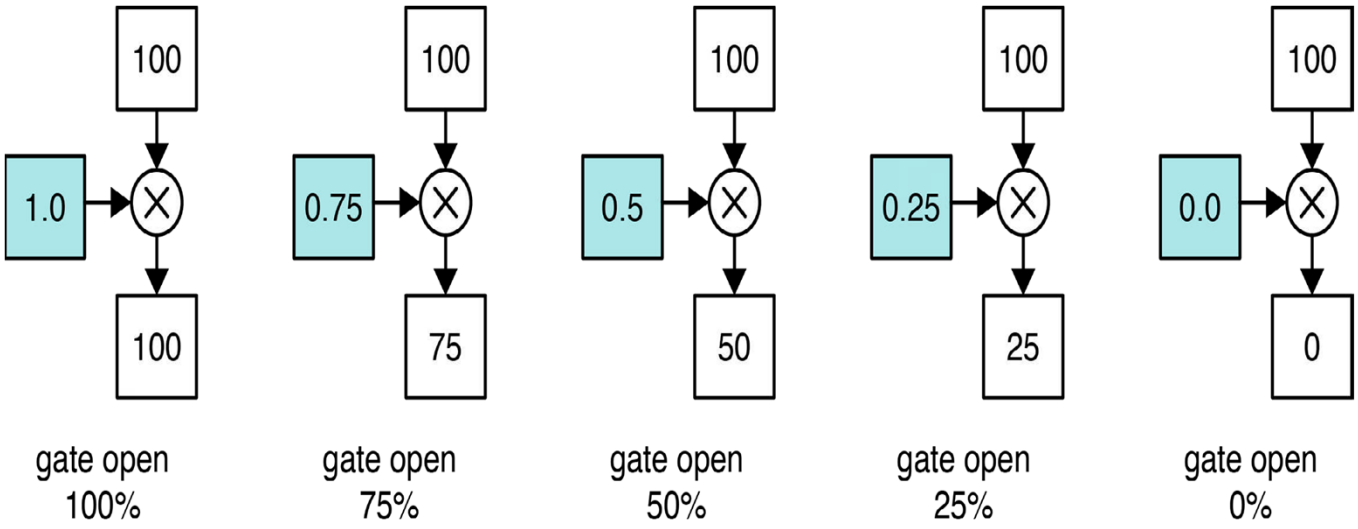


Figure 22.25: Implementing a gate with numbers.

In each diagram, the input value of 100 is shown at the top, the gate value is shown in blue, and the result is in the box at the bottom.

We just multiply the input value by the gate value to get back the gated value.

Forgetting using Gates

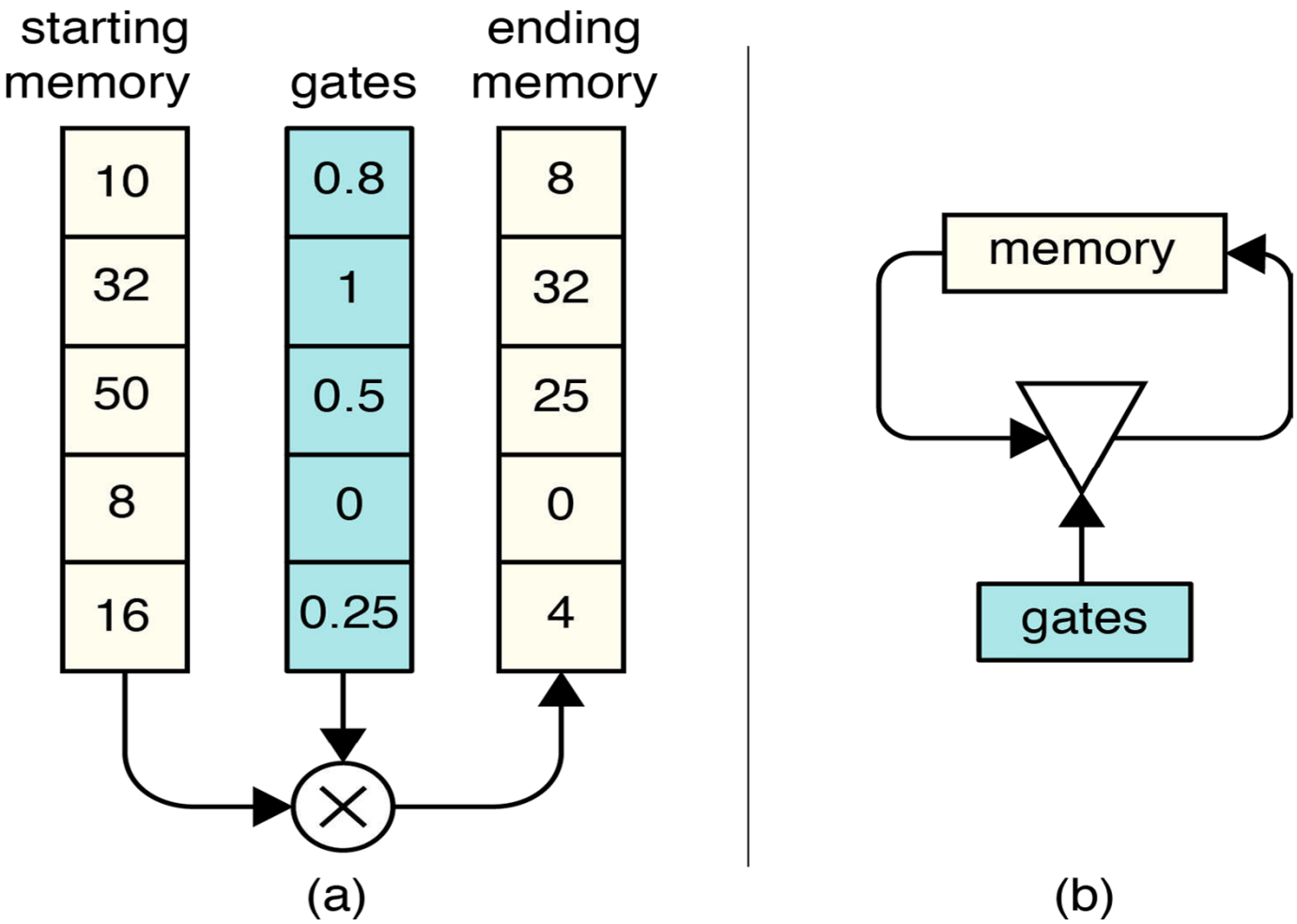


Figure 22.26: Forgetting values in memory.

(a) Each value is multiplied by a gate, and the result is stored back into the memory. Gate values of 1 don't forget anything about their corresponding memory cell, while gate values of 0 cause that value to be completely forgotten. Intermediate values of the gate forget the cell contents by a corresponding amount.

(b) The operation in schematic form.

Remembering using Gates

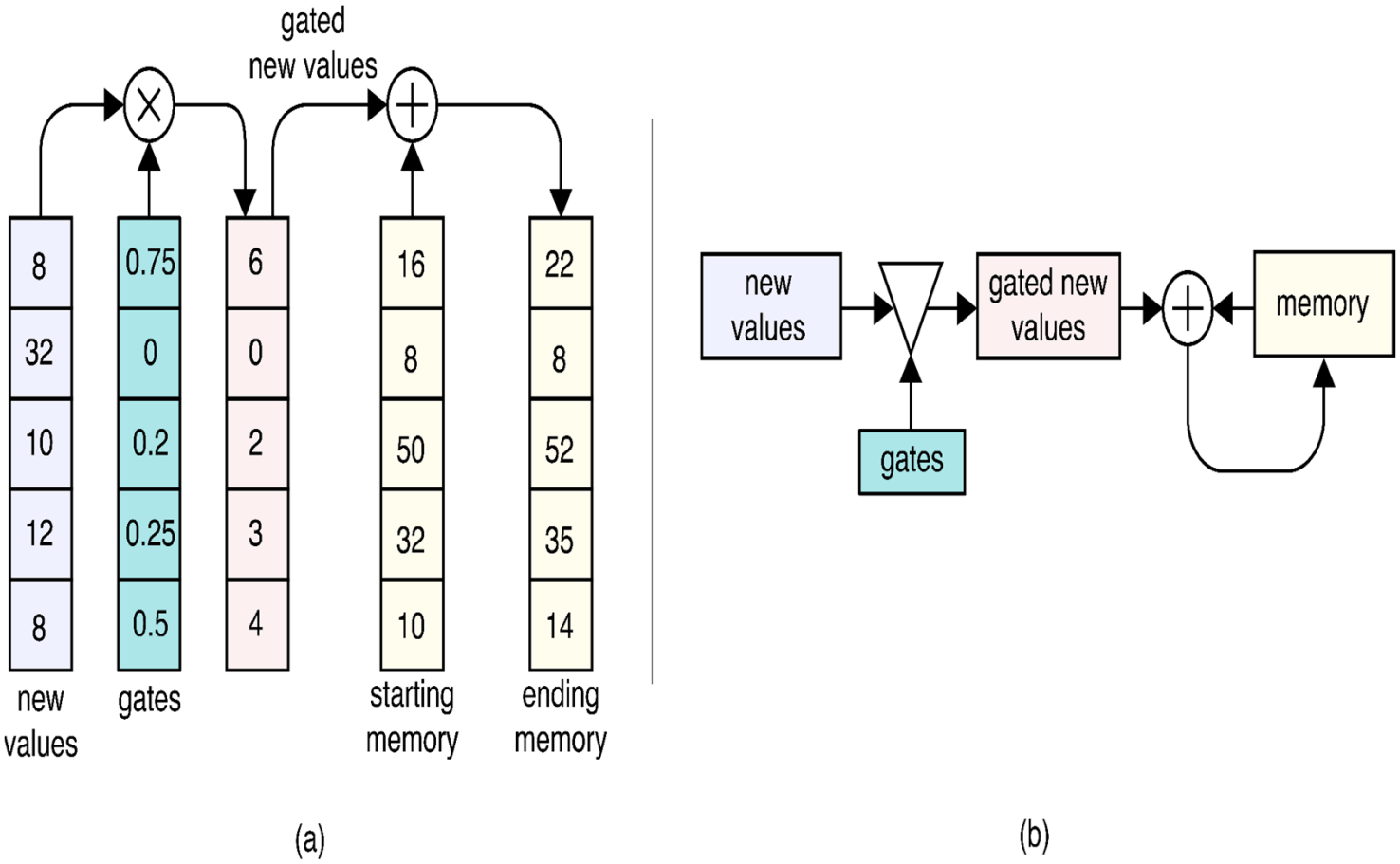


Figure 22.27: The process of remembering. (a) We start with new values to remember, shown on the left. We may not want to remember these at full strength, so we first apply gates.

Then we add the gated values to the existing memory, and save that back into the memory.

(b) The operation in schematic form.

Note: The act of **remembering** involves above **two** steps.

Selection using Gates

To **select** from memory, just determine how much of each element we want to use.

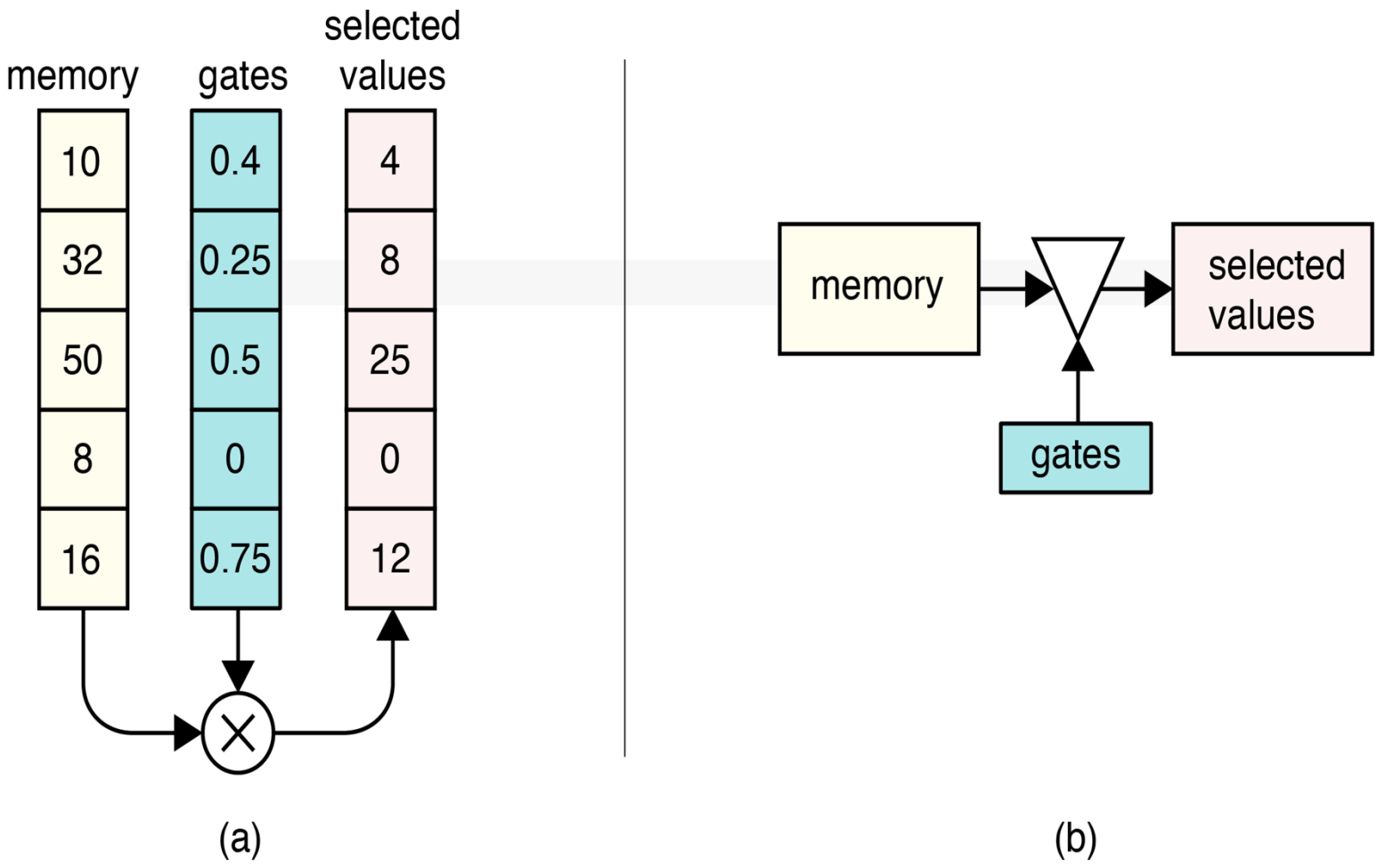


Figure 22.28: Selection using Gates

(a) To select memories, we gate the values in our memory. The gated results are our selections.

Apply gates to the memory elements, and the results are a list of scaled memories.

(b) The schematic version.

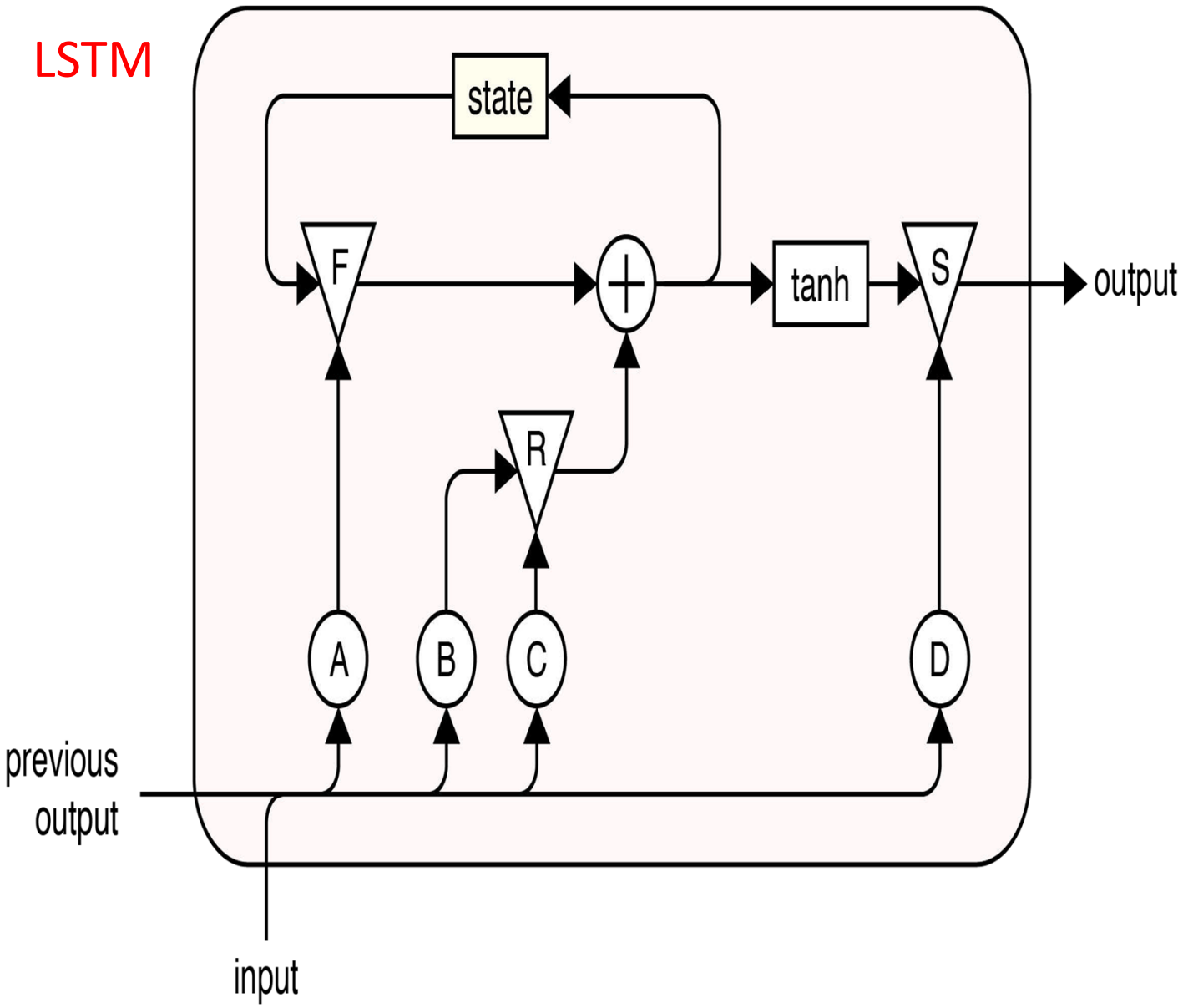


Figure 22.29: The architecture of an LSTM unit.

Inputs: Previous output and a new input

Output: New output on right extreme.

State memory: At top of diagram.

- **Triangles represent gates:** There are three gates, F for forget, R for remember, and S for select.
  - **Gates** manage internal memory and give a lot of control over what cell remembers and forgets over time, so it can manage its internal cell memory in the most effective way.
- **Circles represent sets of neurons,** labeled A through D. The input to these neurons is a single list formed by simply placing the previous output and the new input one atop the other.

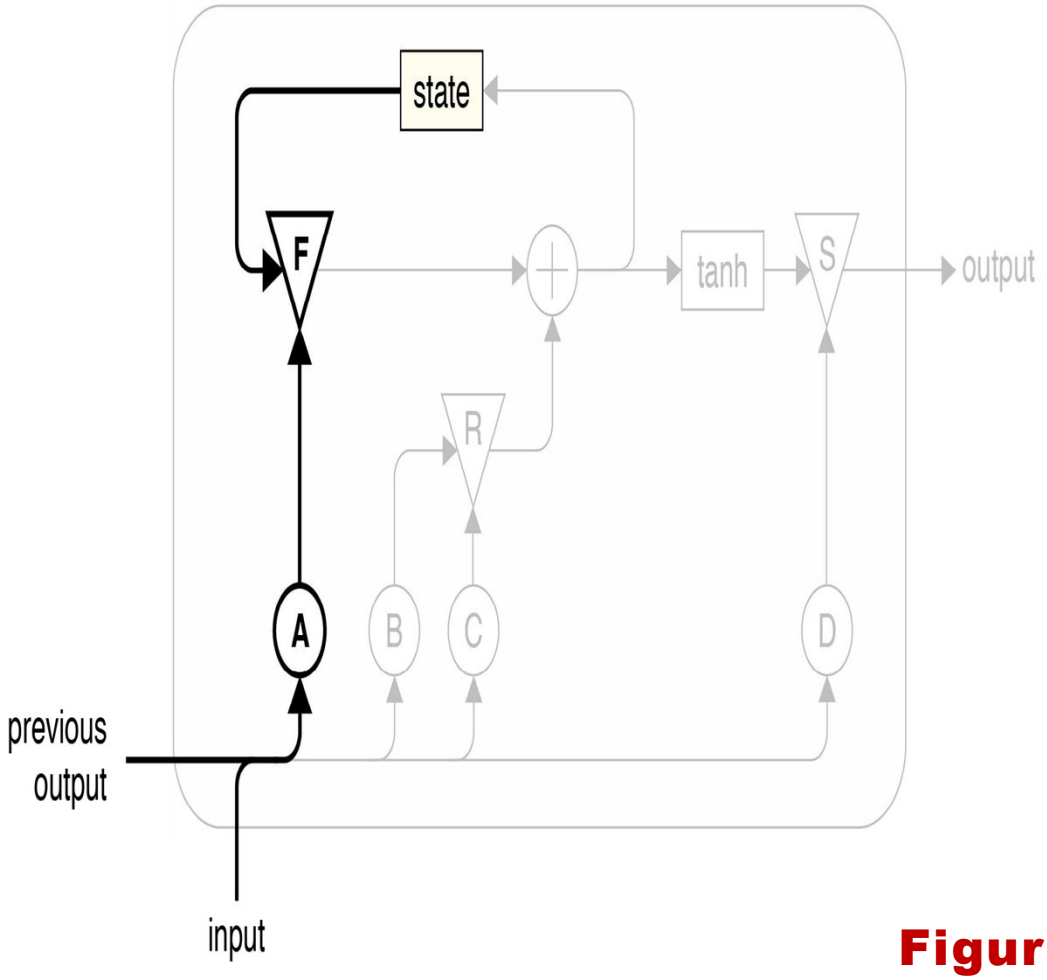


Figure 22.30: Forgetting stage of LSTM.

- **Combined input & previous output are transformed by A neurons into gate signals, which then control F gate & cause some values from State to be forgotten.**

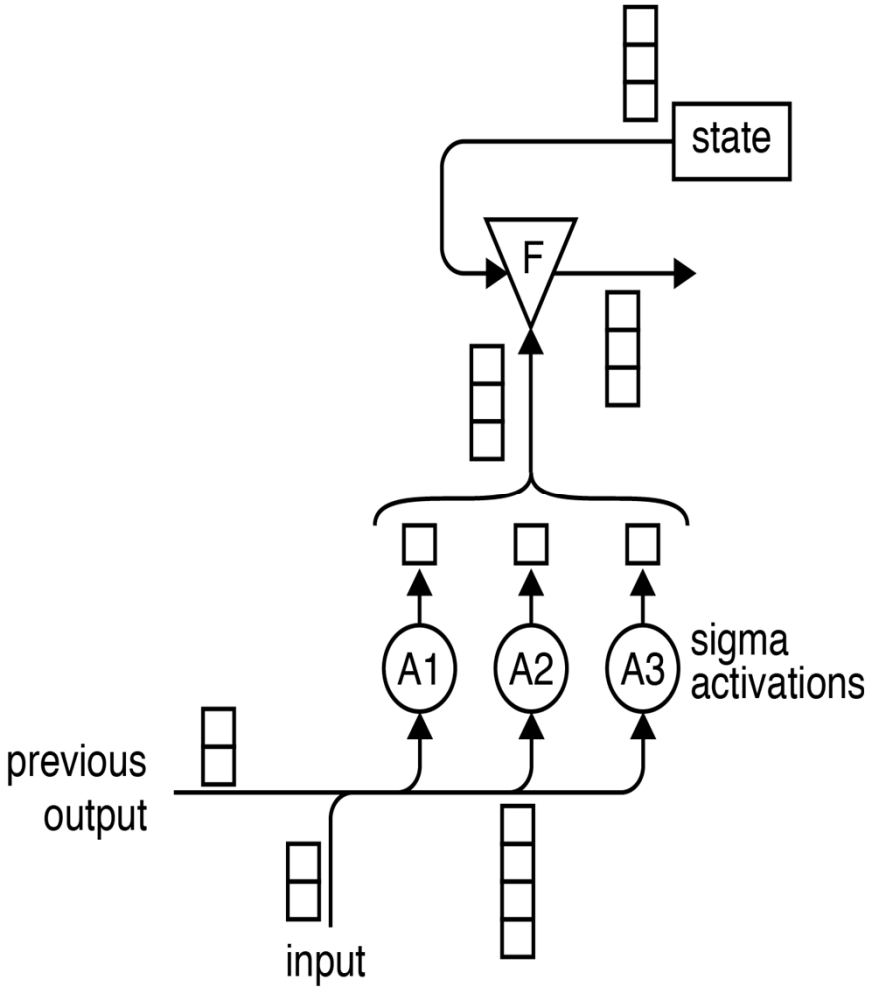


Figure 22.31: Shapes of data moving around

- **Previous output & current input are combined into a single tensor, which is fed into three neurons, each with a sigma activation function**
- **Their three outputs are used to control a gate, whose input is the 3-element state**
- **Output of the gate is state values after being gated by the outputs of "A" neurons.**

Working of forgetting gate

- The new input and the previous output are used by the neurons in A to create a list of gate values.
- Note that a sigma activation function squashes each neuron's output into the range 0 to 1. This makes it appropriate to use as a gate.
- Those are then applied to the current state.
- Each element in the list of values held by the state either stays the same (if its gate value is 1), or else moves towards 0, causing us to "forget" some or all of that value.

- So, at the end of the forgetting stage, we have a temporary copy of the state where we've partially or completely forgotten the elements in the state.

Remembering using gates

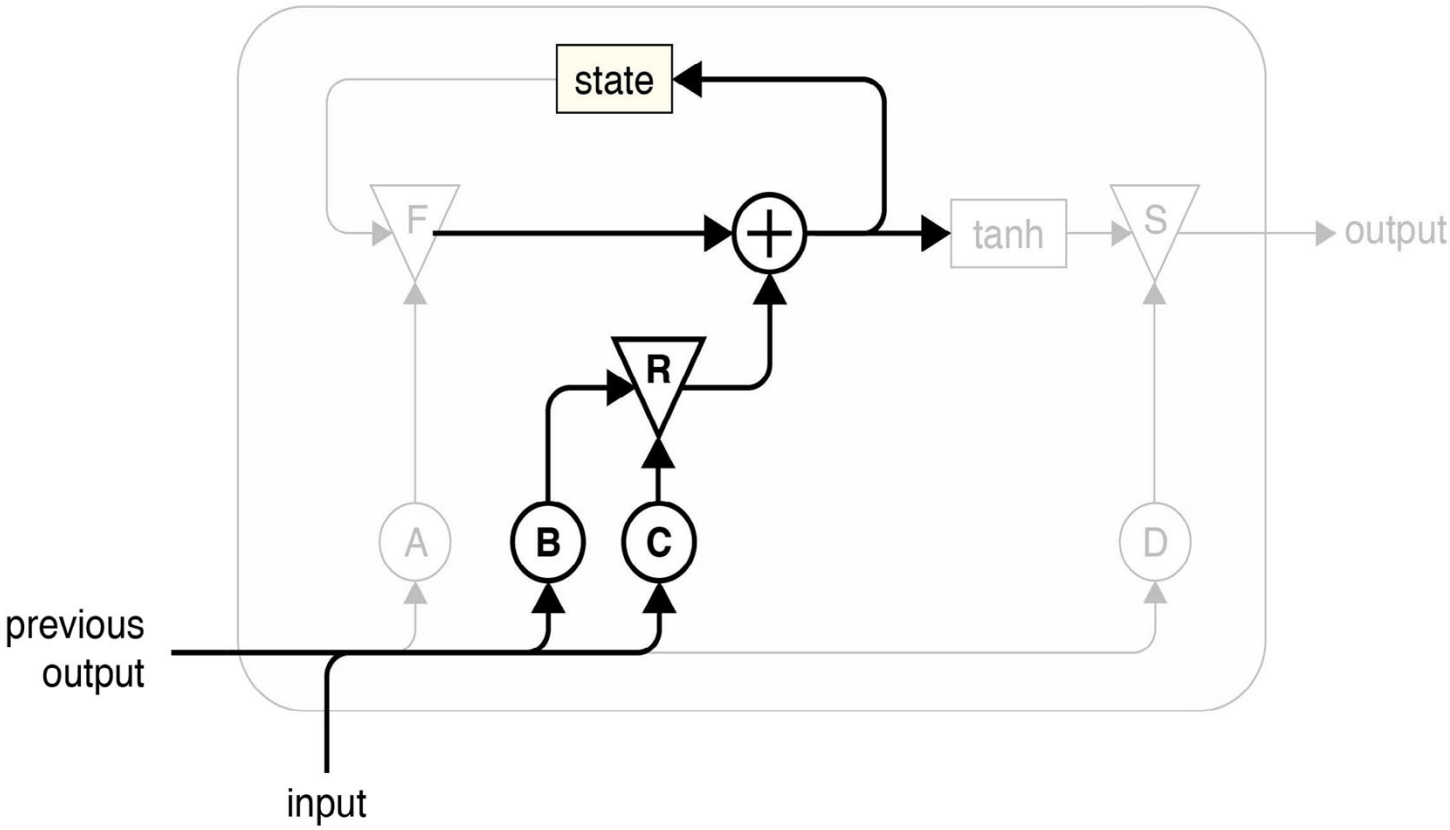


Figure 22.32: **The remember stage of an LSTM.**

- **The combined input and previous output are fed into two sets of neurons, labeled B and C.**
- **The outputs of C are used to control the remember gate R, which adjusts the values coming out of B.**
- **The result is that the input and previous output are scaled down, and then added to the version of the state coming out of the forget gate F.**
- **The result is then written back to the internal state.**

Working of remembering gate

- The next stage is to **remember** something about the new input that's just come in (and something about the previous output as well, if we want).
- As shown in Figure 22.32, we send the combined previous output and input to two sets of neurons, B and C.
- The **C neurons are going to be used as gate values**, so they have a sigma activation function on the end.
- The **outputs of B are the values being gated** (and, ultimately, remembered).
- These gates are used to control **how much of the combined previous output and current input should be remembered** when they pass through the R gate.
- The resulting gated values are added into the state.
- This new value of the state is then written to the state memory, so these values are now remembered

Selecting using Gates

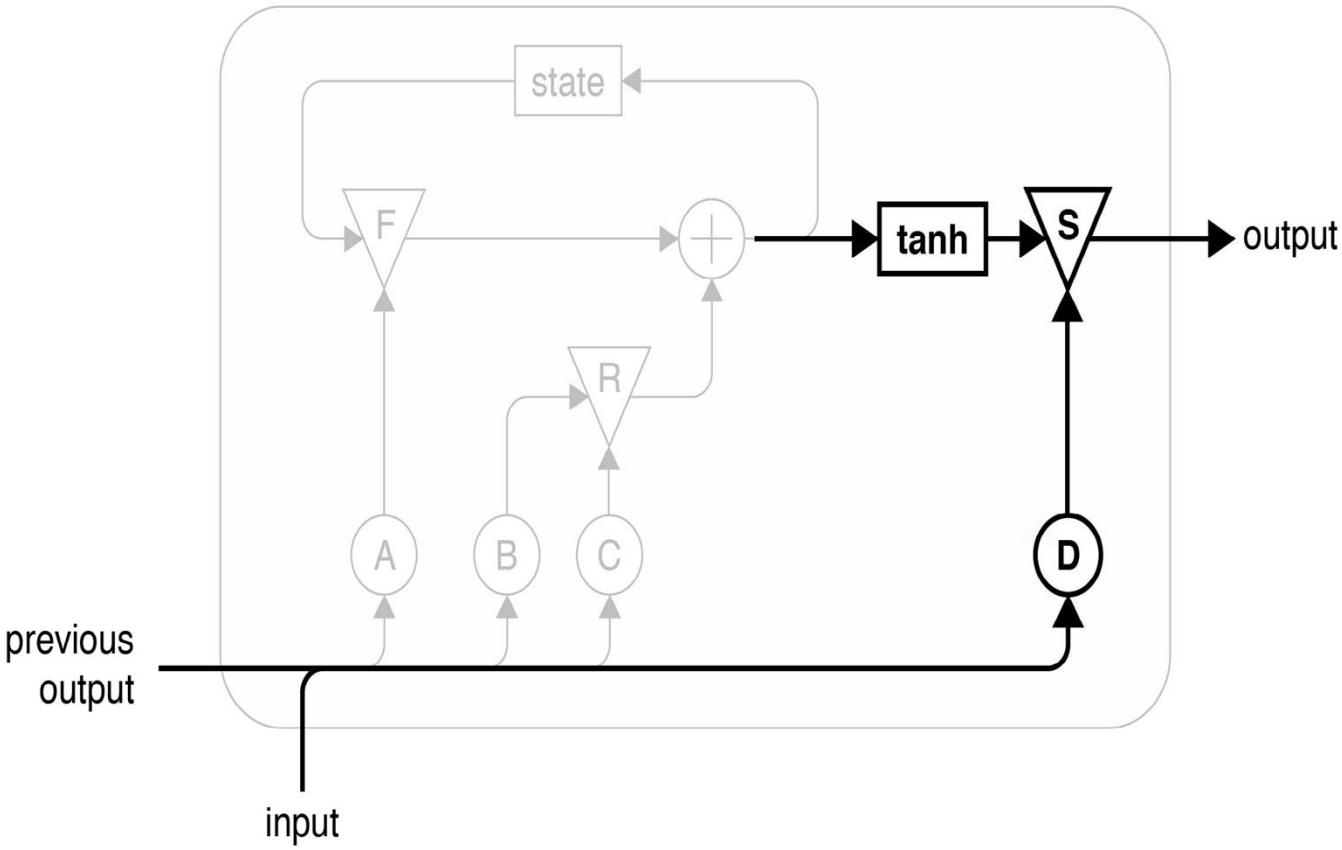


Figure 22.33: The select stage of an LSTM.

- **The combined input and previous output are used to control gate S, which adjusts the new state that was generated by the remember stage.**

- **This is the cell's output.**

Working of selecting gate

- Finally, we **select** some of this newly-computed state for output.
- As shown in Figure 22.33, we run the previous output and current input into another set of neurons marked D, which also have sigma activation functions.
- We use the output of the D neurons as gate values in the gate marked S.
- The input to this gate is the new state we just computed, but first we run that through a *tanh* function.
- The *tanh* function is the same S-shaped function that we saw when discussing activation functions.
- The *tanh* function squashes the state values we just computed, to the range –1 to 1.
- Those values get gated by the S gate, and then become the output of the LSTM cell.

Recap of LSTM

- LSTM takes the previous output and the current input and combines them.

- This combined signal will be used to form gate values, and will also get remembered.
- **The first step** is to **forget** some of the current contents of the state, by running it through the F gate.
- **The second step** is to **remember** some of the new information, by running it through the R gate and then adding that result to the state. The result of that becomes the new state.
- **The third step** is to **select** some of that new state to present as output , by running it through the S gate.

In this way

- LSTM can remember information indefinitely, by not forgetting it and then not adding to it.

OR

- LSTM can completely forget some information and replace it with new values.

OR

- LSTM can partly forget some information, and then partly remember some new values along with them.

How's forgetting, remembering and selecting all done?

- The **four sets of neurons labeled A, B, C, and D** control all this forgetting, remembering, selecting.
- All of their **weights** are learned using gradient descent, like any other weights.
- Eventually the LSTM learns values for the weights that allow it to forget, remember, and thereby

  - Select the right information at the right times to give us useful results.

Some more notes on LSTM

- The name LSTM comes from thinking about what's going on inside an RNN cell.
- RNN cell has some **persistent** or **long-term** memory in the form of its state.
- The state can hang onto its values from one input to the next, **indefinitely**.
- The cell also has some **short-term** memory in the form of the neuron outputs.
- These numbers are temporary, and exist only during the processing of a new input,
  - They are replaced with new values when a new input arrives.
- Goal of LSTM: Take some of those short-term, fleeting values and give them a longer lifespan, allowing them to contribute to future calculations.
- Thus the short-term memory is being given a longer life, resulting in the name "long short-term memory," or LSTM.
- Think of LSTM instead as "persistent short-term memory."

Some more notes on LSTM (Contd.)

- Each RNN cell contains some memory to hold its state.
- The state is usually just a list of numbers.
- The cell memory is initialized with default initial values before the first input arrives.
- Those values change as inputs are received and the cell memory is updated.
- The cell memory can also forget information that is no longer necessary.
- This is all under control of neurons located inside the LSTM.
- The beauty of the whole system is that with enough training, the weights in the neurons inside the LSTM unit learn to adjust themselves.
- The weights adjust in such a way that they control the memory to remember and forget data in just the right ways at the right times.
- Remember that once the weights in these internal networks have been learned by backprop, they don't change.
- But when the unit is evaluating new data, those networks control the cell memory, which *does* change. (So, cell memory or state changes.)

Gated Recurrent Unit (GRU) – A Variant of LSTM

- One of the more famous variants of the LSTM is called the **Gated Recurrent Unit**, or **GRU**.
- The GRU is like an LSTM but with some simplifications.
  - For instance, the Forget and Remembered gates are combined into a single gate.
- Since there's a bit less work to be done, a GRU can be a bit faster than an LSTM.
- It also usually produces results that are similar to the LSTM.
- When working with RNNs, it's often worthwhile to try both the LSTM and GRU to see if either provides more accurate results for a particular network and data set.

What's difference between RNN and other networks?

- **In other networks:**
  - Once training is over, the weights are **frozen**. The weights are saved with the network and they don't change with the input. Once learning is done, the weights are fixed (until we go back to learning again).
  - This means that the network is **done learning** and **done changing**.
  - When we feed in new values, it simply applies the operations that make up the network, using the values it's learned.
- **In RNN networks:**
  - RNN unit is able to remember new information *after training has completed*.
  - That is, Unit keeps changing after training is over.
  - The key idea here is the **memory** that we referred to above.
  - The kind of memory we're talking about is different because it changes when the network is deployed and in actual use.
  - The RNN network itself will learn what it needs to remember and how.
  - We just have to set up the structure to enable it to do the job.

What are Gates used for by LSTM?

An LSTM uses gates for three purposes.

1. Forgetting
2. Remembering - (in input gate)
3. Selecting – (in output gate)

The remember and select gates are also called the **input** and **output** gates.

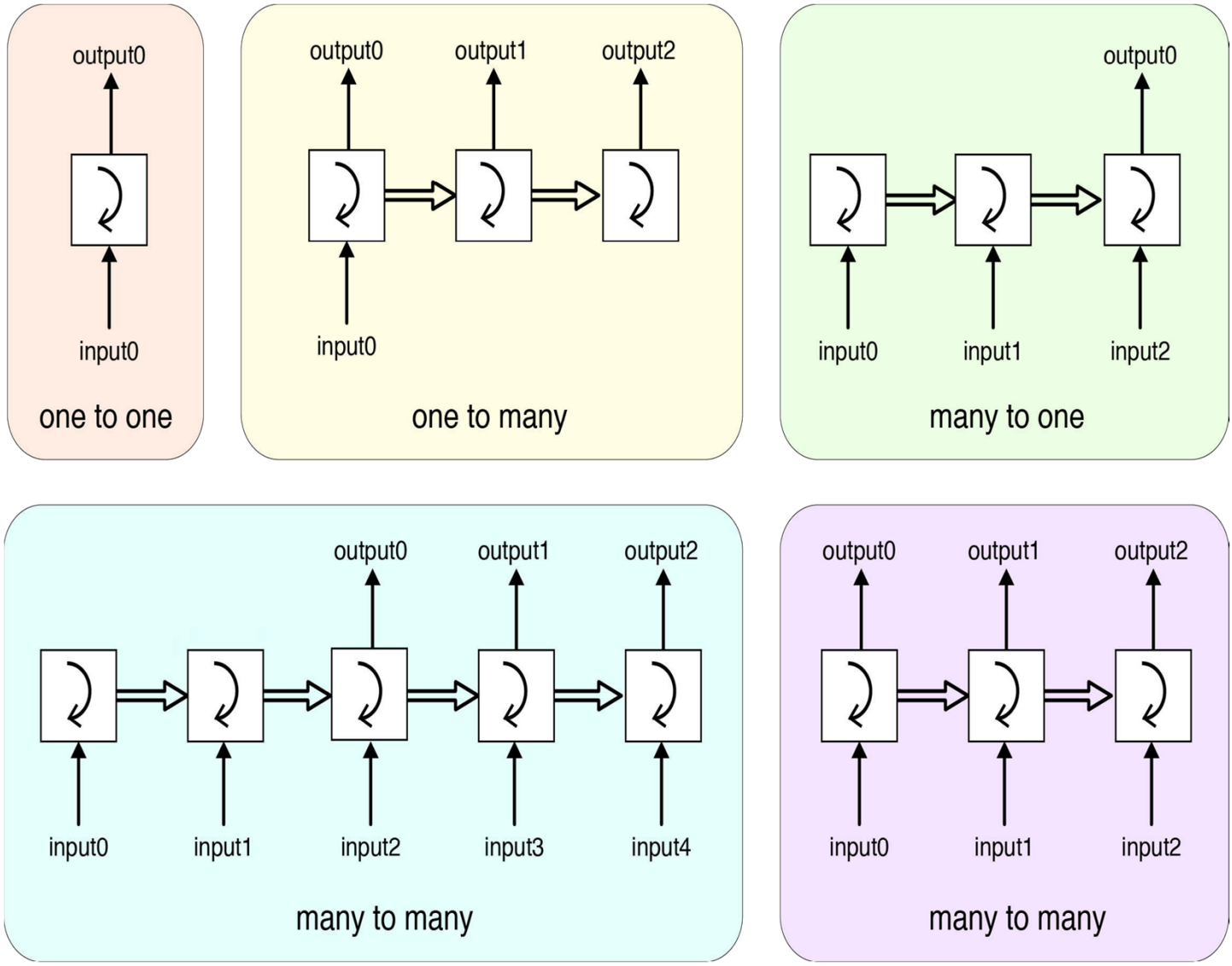


Figure 22.34: Five different types of RNN structures in unrolled form.

The names describe whether the input and output have one value, or are a sequence of many values [Karpathy].

Note: "many" means "sequence"

Refer to Fig. 22.34

- **One to one structure**: We give it a single input (that is, a feature with a single time step) and it produces a single output.
  - This is a waste of RNN. Why? because with a sequence length of 1, the RNN cell is making no use of its unique ability to remember things about its input sequence.
- **One to many structure:** Takes in a single piece of data and produces a sequence.
  - We do this by giving the RNN the information we have to get it started, and then we let it run for a while, producing multiple pieces of output.
  - **Example**: Give it the starting note for a song, and the network produces the rest of the melody.
- **Many to one structure:** Reads in a sequence and gives us back a single value.
  - This organization is used frequently in the field of **sentiment analysis**, where the network is given a piece of text and then reports on some quality inherent in the writing.
  - **Example:** A common example is to look at a movie review and determine if it was positive or negative.

- **Many to many structures**: Most interesting. Two cases
  - Bottom left of Fig. 22.34: We "prime" the RNN with several pieces of input before we ask it to start producing outputs. So, output is delayed.
  - **Example**: Machine translation. In some languages words don't come in the same order, so we can't start translating right away.
  - The English sentence "The black dog slept in the hot sun" can be expressed in French as "Le chien noir dormait dans le soleil chaud."
  - In French, the adjective "noir" (black) follows the noun "chien" (dog), so we need to have some kind of buffer so we can produce the words in their proper English order.
  - Bottom right of Fig. 22.34: We start producing outputs right away. So, each new input produces a corresponding new output.
  - **Example**: A description for every frame of a video.
  - Suppose we have a piece of video of a ball flying through the air.
  - We'd like to assign a label to every frame.

A layer that holds an LSTM unit is called a **recurrent layer**, or an **RNN layer**. The symbol we use for this layer is shown in Figure 20.12+ 22.35 below.

Figure shows

- Standard way to draw a recurrent cell,
- Icon for an entire layer of Recurrent cells.
- Icon for a recurrent cell that returns a sequence.

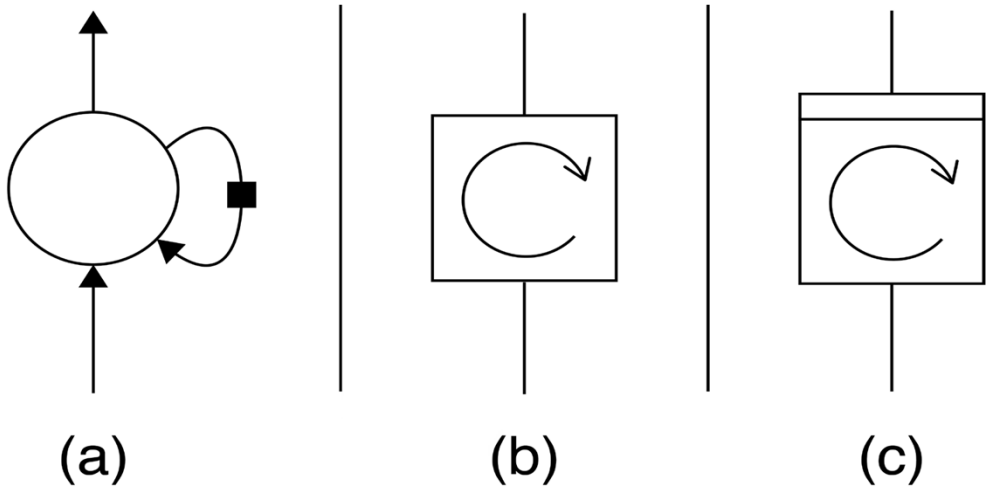


Figure 20.12 & 22.35: A recurrent cell.

(a) A typical way to draw a recurrent cell. The loop at the right is meant to remind us of the internal state being read and written. The black box represents a step of delay., i. e., one step of memory.

(b) Our icon for an RNN layer (of RNN cells).

(c) Our icon for a recurrent cell that returns a sequence.

Deep RNN

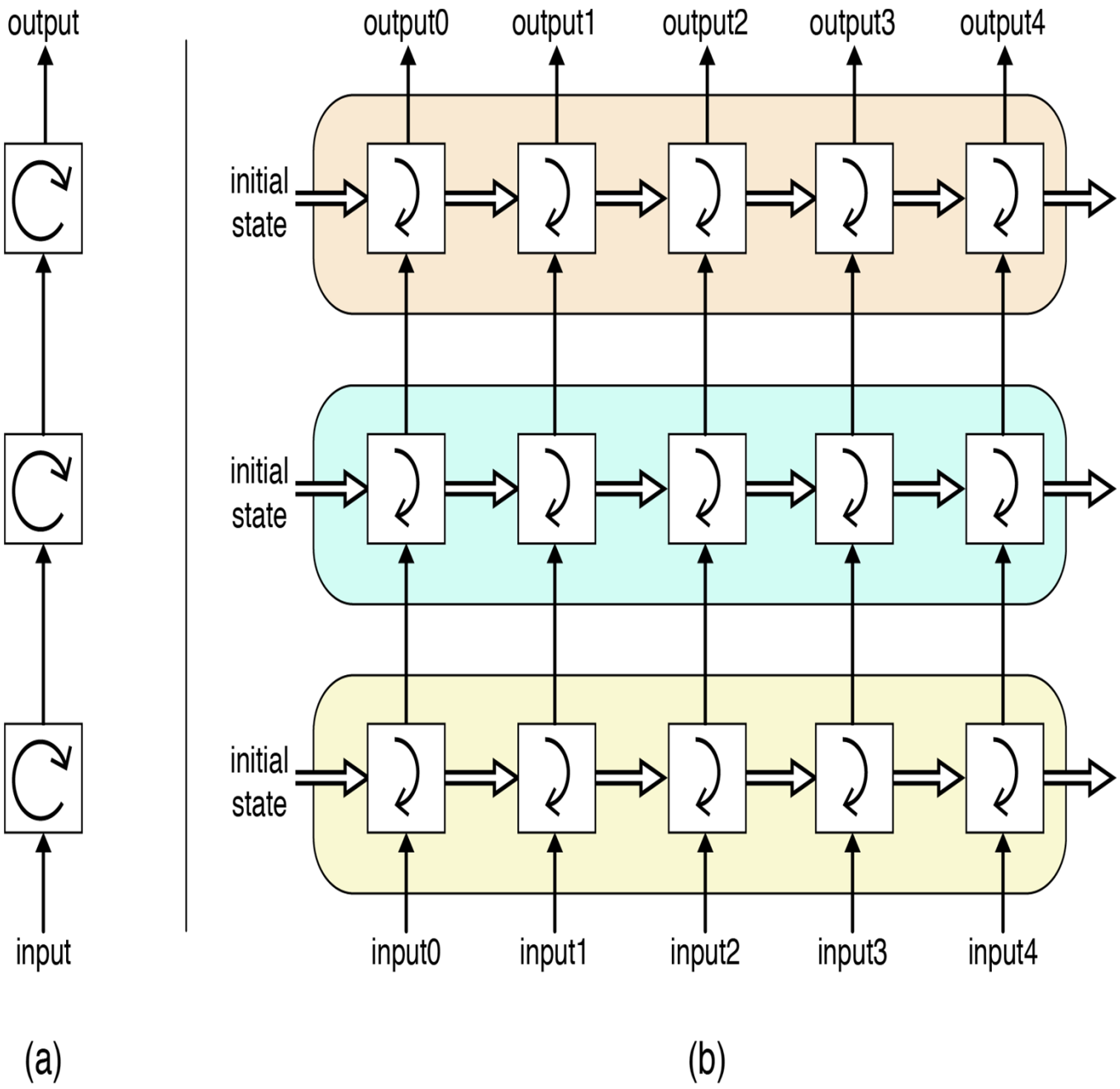


Figure 22.37: A deep RNN uses multiple RNN stages, with the output of one feeding the input of the next.

Each stage maintains its own state. This deep network uses three layers of RNNs.

(a) The usual rolled-up diagram.

(b) The unrolled version.

Deep RNN

- The basic idea is that the LSTM on each layer feeds the LSTM on the next layer.
- **First time step** of a feature is fed to the first LSTM.
  - First LSTM processes that data and produces an output (and a new state for itself).
  - That output is fed to the next LSTM, which does the same thing, and the next, and so on.
- **Second time step** arrives at the first LSTM, and the process repeats.
- All LSTMs but last one work with
  - intermediate representations that make sense to the next layer,
  - but not useful to us as they are not outputs of the network.
- So, early LSTMs can encode their data in dense and complex ways for maximum efficiency.
- In practice, we'd need the first layer to return a sequence, rather than a single value.

Bidirectional RNN

- LSTM is designed to process a sequence of values.
- Sometimes it makes sense to give inputs to an LSTM *backwards*, starting at the end and working backwards towards the beginning.
- **Example**: "Saying, 'I need a vacation', Charles sat down."
  - If we want to know who "I" refers to, then scanning the sentence from finish to start would let us know it was Charles.

- This idea led to the introduction of the **Bidirectional RNN** or **BRNN**.

- It's sometimes called a **Bidirectional LSTM** or **BLSTM** when we're specifically using LSTM units.

Bidirectional RNN (contd.)

- The BRNN network runs the input in both directions at once.

- The BRNN can only work in situations where we already have data all the way to the end of the chunk we're trying to analyze, so it's not applicable to every situation.
- **Example:** if we're trying to understand commands spoken in real time, to start at the end we'd have to wait for the person to finish speaking.

- Figure 22.38 shows the structure of a bidirectional RNN layer.
- We use two RNNs together in one layer, one getting the inputs from start to finish, the other getting the inputs from finish to start.

Bidirectional RNN (contd.)

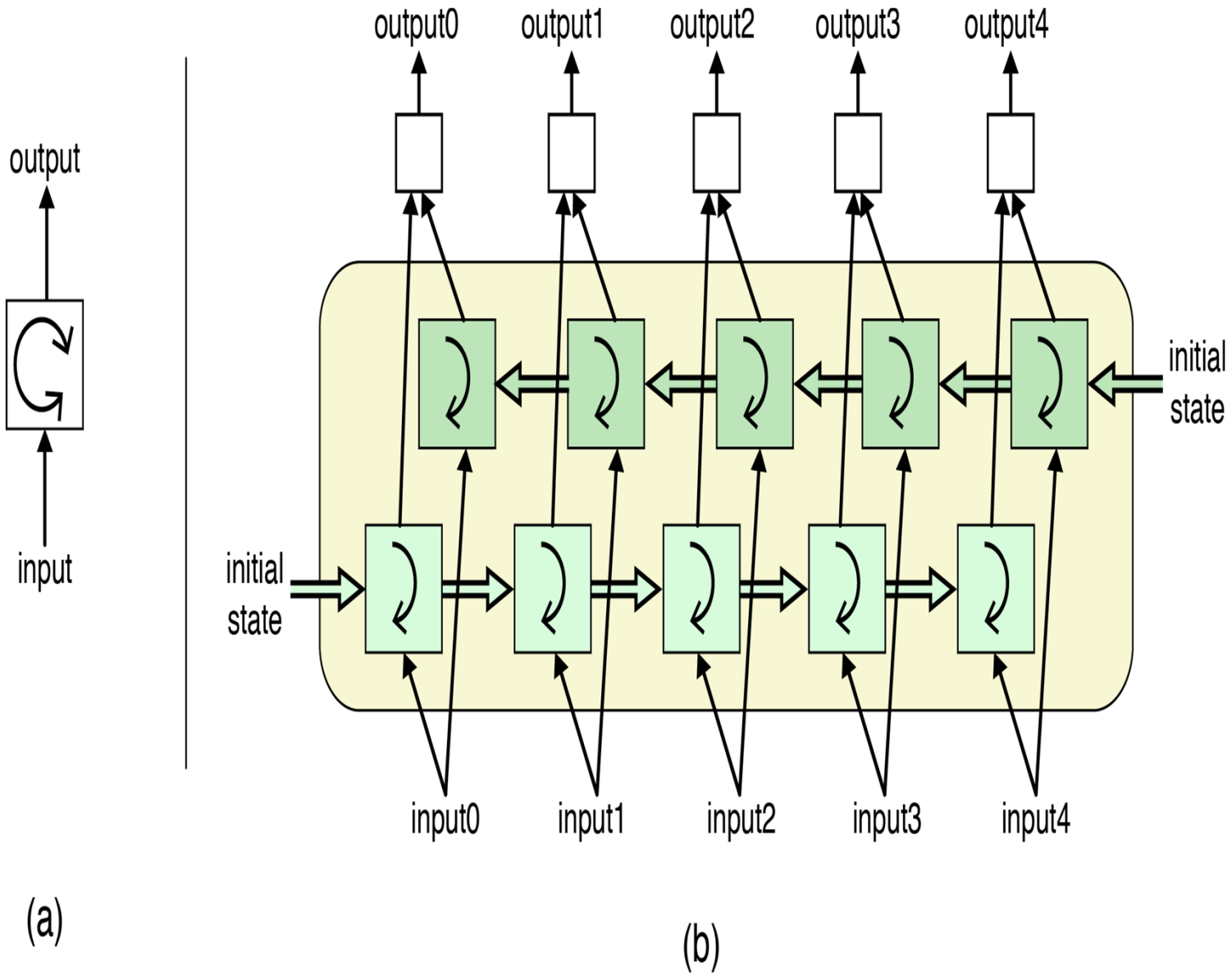


Figure 22.38: A bidirectional RNN is two RNNs running together.

One of them is given the time steps in the usual order, from first to last.

The other is given the time steps in the opposite order, from last to first.

In this diagram, there are only two LSTM units, each with its own state.

(a) Icon for a BRNN. (b) An unrolled BRNN.

Working of the Bidirectional RNN (BRNN) in Figure 22.38(b)

- We see two LSTM units, one in light green and the other in darker green.

- To get started, the value of time step 0 is handed to the "forward" LSTM unit (light green) while *simultaneously* time step 4 is handed to the "backward" LSTM unit (dark green).
- When both LSTMs have produced their outputs, they arrive at the white boxes at the top of the figure. The two values at each white box are produced at different times, so the first one just sits until the second one arrives.
- Now we proceed to the next time step.
- We give the value of time step 1 to the forward LSTM, and time step 3 to the backward LSTM. Their outputs are held at the white squares.
- We proceed to give time step 2 to the forward LSTM, and also to the backward LSTM. Both outputs go up to the white square (but before we deal with them let's finish processing the sequence.)
- We give input 3 to the forward LSTM and input 1 to the backward LSTM,
- Finally, we give input 4 to the forward LSTM, and input 0 to the backward LSTM.

Working of the Bidirectional RNN (BRNN) in Figure 22.38(b) ..... Contd.

- Now we have two values coming into each of the square boxes at the top.
- The square boxes combine their inputs in whatever way we choose.
- Typical options are to add them together, average them, multiply them, or create a 2-element tensor (that is, a list) by appending one value after the other.
- Those values can then go on to act as outputs of the network, or inputs to any other layer.

Deep Bidirectional RNN

- For even more compute power, combine deep RNNs with bidirectional RNNs.
- This creates a **deep bidirectional RNN**, shown in rolled and unrolled form in Figure 22.39.
- Deep bidirectional recurrent neural nets offer a lot of computational power.
- That power comes with a lot of weights to be trained, and a corresponding need for a lot of training data.
- Deep BRNNs have found use in applications like speech recognition, image captioning.

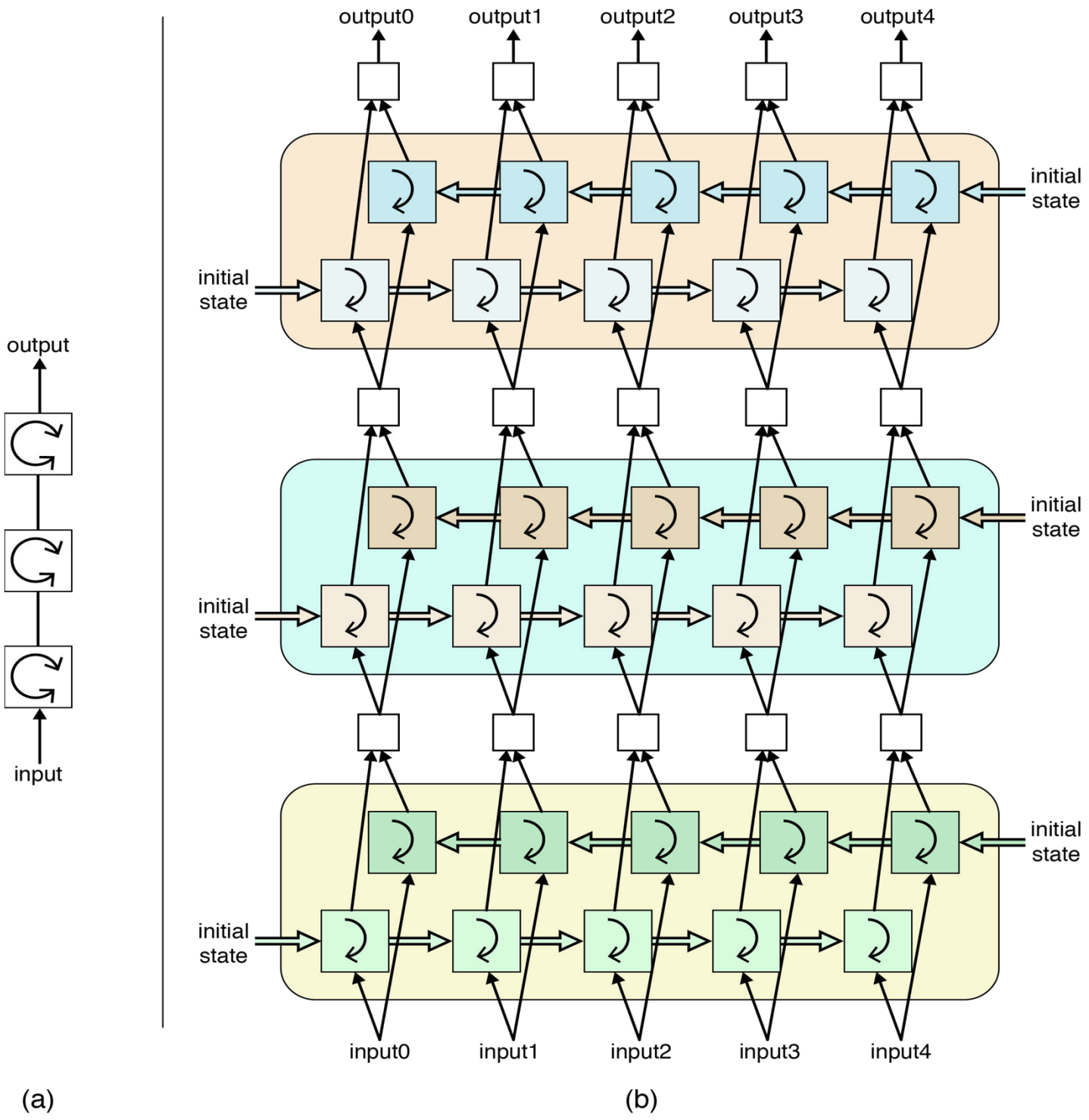


Figure 22.39: A deep bidirectional RNN. Each rounded box represents a single layer of the bidirectional RNN. We say this is "deep" because there are multiple such layers, each feeding the next. It's bidirectional because in each layer there is both a forward and backward LSTM unit. Keep in mind that there are only six LSTM units in this diagram, two on each layer. (a) The rolled deep bidirectional RNN. (b) The unrolled version.