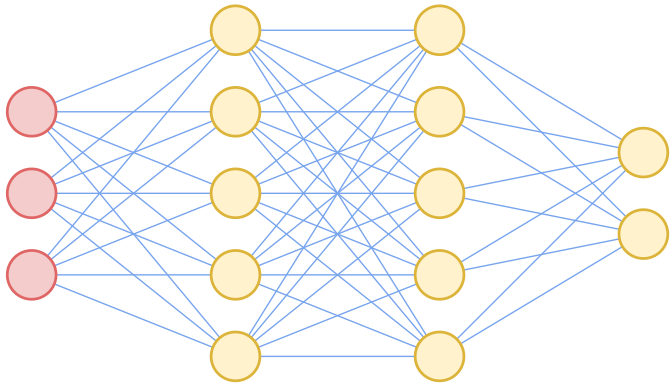


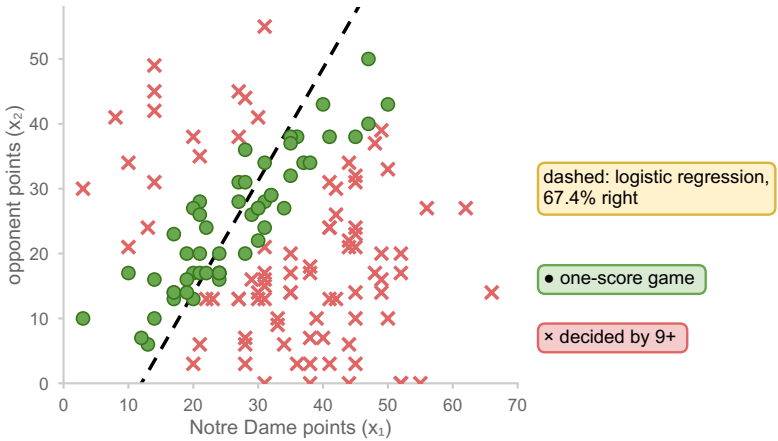
UNIT 02 · CHAPTER 1

Feed-Forward Networks

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

Linear Decision Boundaries

In Model Evaluation we asked "Did we win?" and a **logistic regression** answered with a straight line through the two teams' yards, getting `83.7%` of unseen games right. Let's ask a different question about the same games: "Was it close?" Call a game a **one-score game** if it was decided by `8` points or fewer (a touchdown and a two-point conversion), which `55` of our `141` games were.

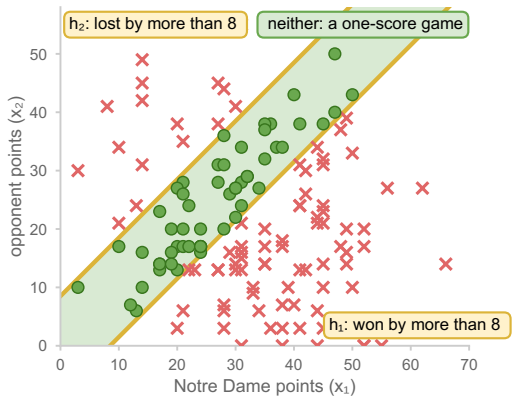


The close games sit in a strip along the diagonal, with blowouts on both sides. Logistic regression gets only `67.4%` of the games right, and it isn't the fit's fault. The best any straight line can do is `88.7%`, by cutting off one side of the strip and leaving the other side's blowouts in with the close games. **It's the XOR problem again, this time with our data.**

(Asking this of the final score is a little silly, since we could just subtract. It's here for the shape.)

Multi-Layer Perceptrons

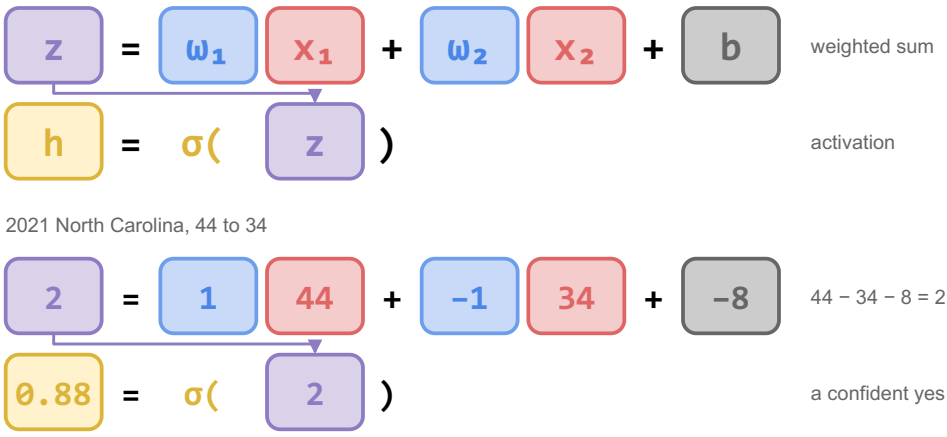
In the Perceptron chapter no single perceptron could solve XOR, but two side by side (an OR and a NAND) followed by a third (an AND) could. The same trick works here. One line asks "did Notre Dame win by more than 8?", another asks "did Notre Dame lose by more than 8?", and a game is close when **both answers are no**.



Stacking perceptrons like this, so some take the outputs of others as their inputs, gives a **multi-layer perceptron (MLP)**. Frank Rosenblatt described networks like this in the `1960s`, though in his only the last layer could learn and the earlier ones were fixed, a problem we'll come back to at the end of the chapter.

The Artificial Neuron

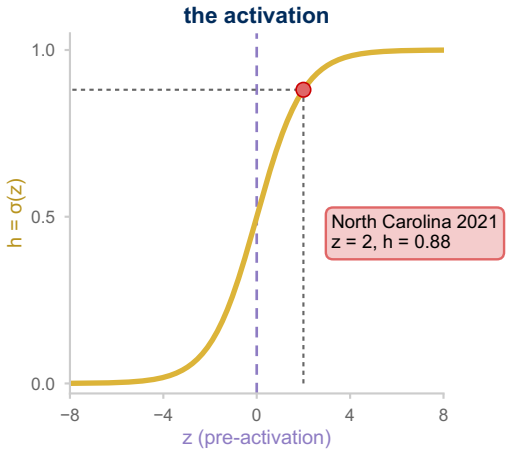
A perceptron's step function only ever says yes or no. Logistic Regression gave us a softer version, the sigmoid, which also says how sure it is. A perceptron with a smooth activation like this is usually called an **artificial neuron** (a loose nod to biology, where a neuron fires once its input passes a threshold).

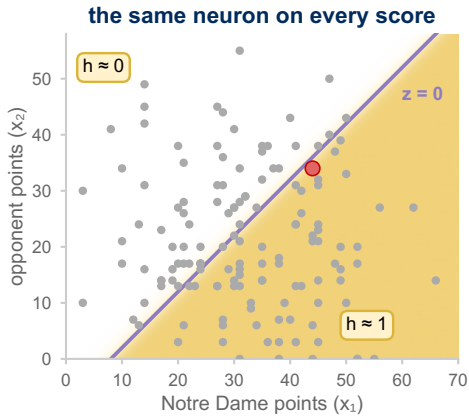


It takes a weighted sum of its inputs plus a bias, $z = w_1x_1 + w_2x_2 + b$, called the **pre-activation**, then passes z through an **activation function** (here the sigmoid, σ) to get its output. The weights and bias are its **learnable parameters**.

One Neuron, One Line

The numbers in that figure are a "won by more than `8`?" neuron on a `10`-point win. Drawn over every possible score, the neuron is a line with a soft edge, **h** near `0` on one side of **z** = `0` and near `1` on the other:

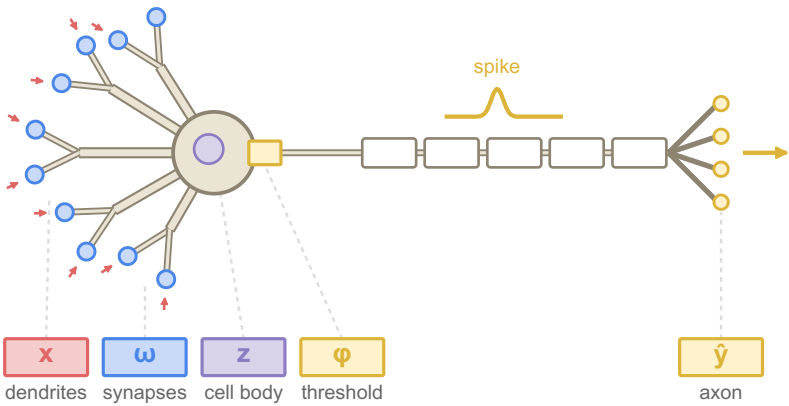




A `9`-point win would only get `0.73`, soft for a hard cutoff, so our network multiplies the weights by `10` and sets the bias to `-85`. The switch then sits at `8.5`, and **every real score lands within `0.007` of `0` or `1`**.

Real Neurons

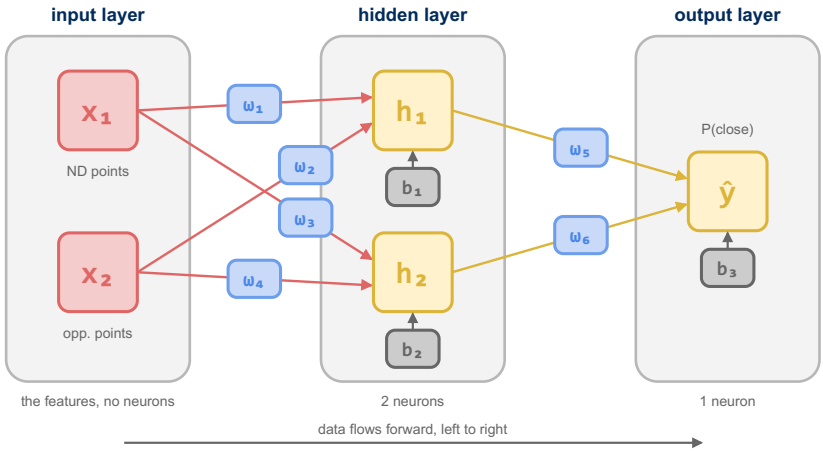
The artificial neuron is named after a real one, so it's worth seeing how loose the copy is. A real neuron collects signals from other neurons on its branching **dendrites** (a neuron in our cortex has about `7,000` connections). Each connection, a **synapse**, makes its signal count for more or less, the cell body adds them all up, and if the total passes a threshold the neuron **fires a spike down its axon** to the next neurons.



Pictured above is each part with the piece of our neuron it turned into: the inputs **x**, the weights **ω**, the pre-activation **z**, the activation function and the output. That's about where the resemblance ends. We have around `86` billion neurons, and our football network has `3`.

Feed-Forward Networks

Three of these neurons, arranged like our MLP, make a **feed-forward neural network (FFN)**. Each column of neurons is a **layer**.



The **input layer** is just the feature vector, the two teams' points, and has no neurons. The **hidden layer** holds h_1 and h_2, called hidden because their values never appear in the data, we only ever see the inputs and the label. The **output layer** holds $\hat{y}$, the network's answer. Data only flows forward, with no loops, hence the name.

A layer where every neuron takes every output of the layer before is a **fully connected layer** (also called a dense or linear layer). The number of layers is a network's **depth** and the number of neurons in a layer its **width**, and **deep learning** is the study of networks with many layers. Ours, with **9** learnable parameters, is about as shallow as they come.

The Forward Pass

Here are the weights for our network:

```
[0.1]  two neurons look at the score
       h1 = σ(10x1 − 10x2 − 85)      h2 = σ(10x2 − 10x1 − 85)
[0.2]  a third combines them
       ŷ = σ(2 − 4h1 − 4h2)
```

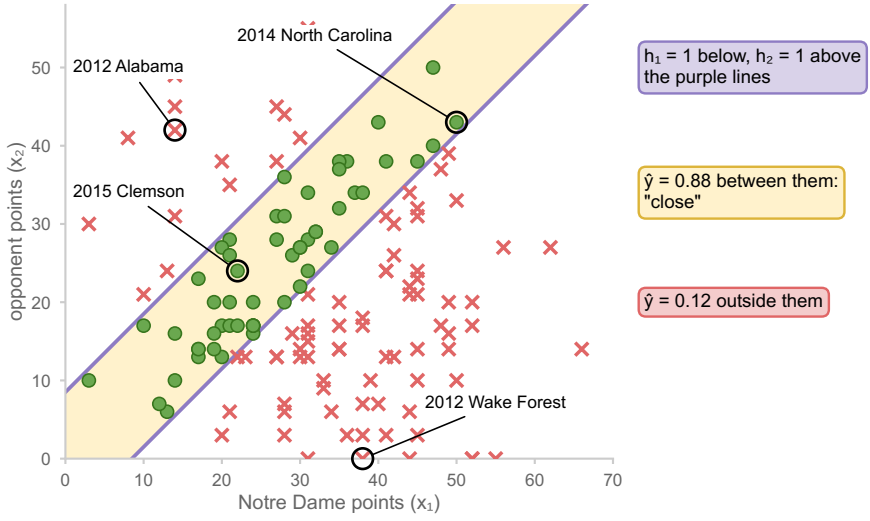
Read neuron by neuron, h_1 is `1` when Notre Dame won by more than `8` and `0` otherwise, and h_2 is the same for losses. The output starts at $\sigma(2) = 0.88$ and drops to $\sigma(-2) = 0.12$ when either hidden neuron fires, so **the output neuron says "neither"**, which is exactly a one-score game.

Computing the output from the inputs like this is called the **forward pass**. Here it is on four games, calling a game close when $\hat{y}$ is above `0.5`:

game	score	h_1	h_2	$\hat{y}$	close?
2014 North Carolina	50–43	0	0	0.88	yes
2015 Clemson	22–24	0	0	0.88	yes
2012 Wake Forest	38–0	1	0	0.12	no
2012 Alabama (title game)	14–42	0	1	0.12	no

Decision Regions

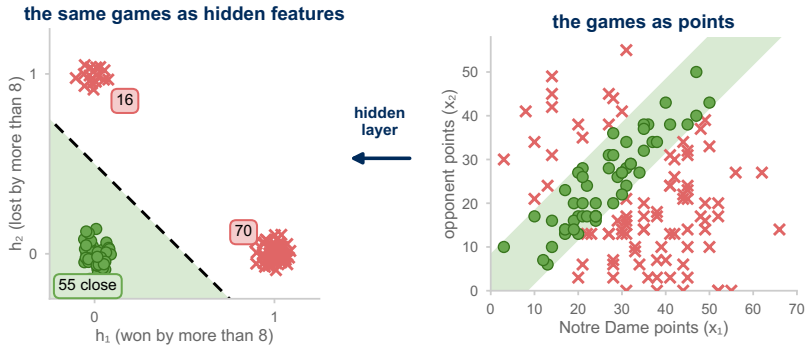
Each hidden neuron switches on past its own line, so over every possible score the output says "close" only between the two lines. The parts of the plane where a network gives each answer are its **decision regions**, and ours is a strip:



On all `141` games it gets every one right (`100%`), where no straight line can do better than `88.7%`. There's nothing here to overfit, the rule is exact, so it would do just as well on games it has never seen.

Hidden Features

So why can three neurons do what no line could? Look at what the hidden layer hands on. Every game comes out of it as a pair of numbers, (h_1, h_2), and since each is within `0.007` of `0` or `1`, every game lands on a corner:



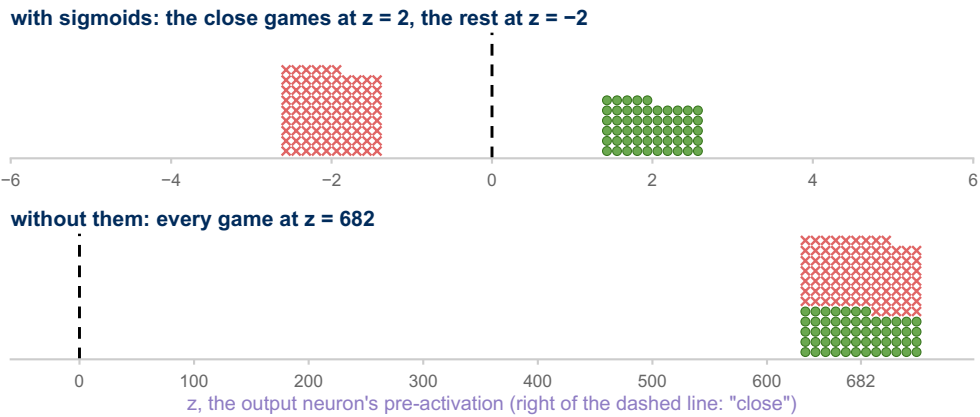
On the right the close games sit in a strip no line can cut out. After the hidden layer, on the left, the same `55` games all sit at `(0, 0)`, the `70` big wins at `(1, 0)` and the `16` big losses at `(0, 1)`, and the output neuron's line, $h_1 + h_2 = 0.5$, separates them perfectly. The output neuron is just a logistic regression, **the hidden layer moved the games somewhere a line works**.

This is the real power of a hidden layer. In Linear Regression we added features like x^2 by hand and had to guess which would help. A hidden layer computes its own features from the inputs (here "won big" and "lost big"), and the next layer only has to draw a line through them.

Non-Linearity

Does the hidden layer need its sigmoids to build those features? Let's take them out and see. Without them $h_1 + h_2 = -170$ for every game, since the $10x_1$ and $10x_2$ terms cancel and the two biases add up. The output neuron's pre-activation is then `682` whatever the score was, and the network calls all `141` games close (`39.0%` right). **The two neurons cancel out.**

Here is every game at the output neuron's pre-activation, which says "close" right of the dashed line. With the sigmoids the close games and the rest land on opposite sides, without them all `141` land in the same spot:



Linear Layers Collapse

That isn't bad luck with our weights. A layer with no activation function is a linear transformation of its inputs (a weighted sum plus a bias, as in Linear Regression), and a linear transformation of a linear transformation is still linear:

```
[0.3]
```

two layers with no activation are one layer

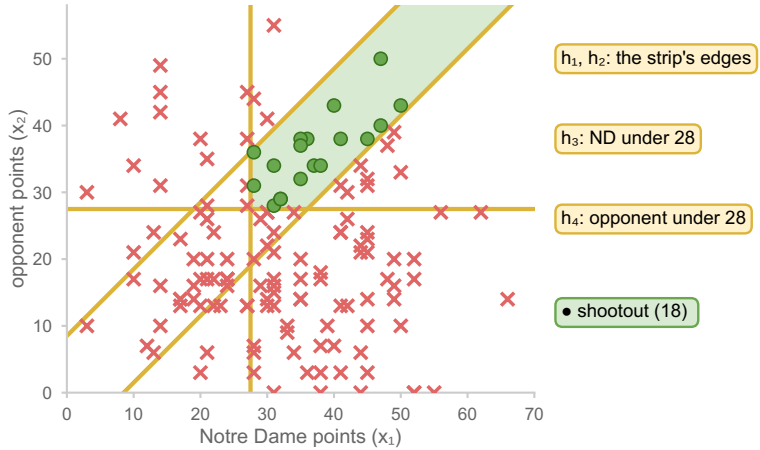
$$(\mathbf{X}\mathbf{w}_1^\top + \mathbf{b}_1)\mathbf{w}_2^\top + \mathbf{b}_2 = \mathbf{X}(\mathbf{w}_2\mathbf{w}_1)^\top + (\mathbf{b}_1\mathbf{w}_2^\top + \mathbf{b}_2)$$

However many layers we stack, without activation functions they collapse into a single layer with weights $\mathbf{w}_2\,\mathbf{w}_1$, which can only draw the one straight line a logistic regression could. (Ours collapses to $\mathbf{w}_2\,\mathbf{w}_1$ = `[0, 0]`, no line at all.)

So the activation function's real job is to **introduce non-linearity**. Without the bend, the "new features" are just the old ones stretched and rotated, and a line through them is still a line through the original points.

More Neurons, More Shapes

Each hidden neuron draws one line, so more neurons means more lines to build a shape from. Take a narrower question: was it a **shootout**, a one-score game where both teams scored **28** or more (four touchdowns)? **18** of our **141** games were.



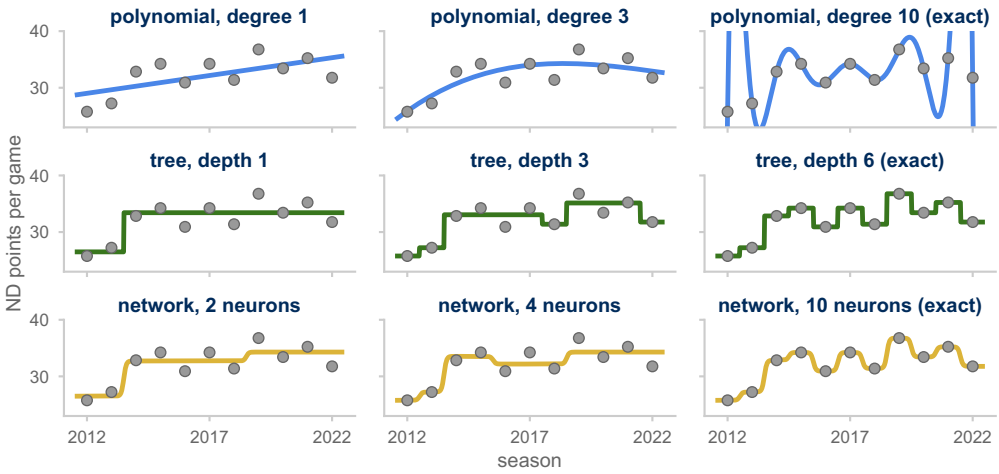
Two more hidden neurons, $h_3 = \sigma(275 - 10x_1)$ for "Notre Dame scored under **28**" and $h_4 = \sigma(275 - 10x_2)$ for the opponent, cap the strip. The output neuron is the same idea as before, $\sigma(2 - 4h_1 - 4h_2 - 4h_3 - 4h_4)$. It says yes only when none of the four fire, and it gets all **141** games right.

Every hidden neuron adds an edge, and an output neuron that wants all of them quiet (an AND) carves out the region they surround. One that fires when any of them does (an OR) would glue regions together instead.

Universal Approximation

How far does this go? The **universal approximation theorem** says a network with a single hidden layer, given enough hidden neurons, can approximate any continuous function as closely as we like (Cybenko proved it for sigmoids in **1989**, Hornik for almost any activation in **1991**).

A steep sigmoid is nearly a step, one step per neuron. Here are Notre Dame's points per game each season, fit by networks and by two models we've met before:

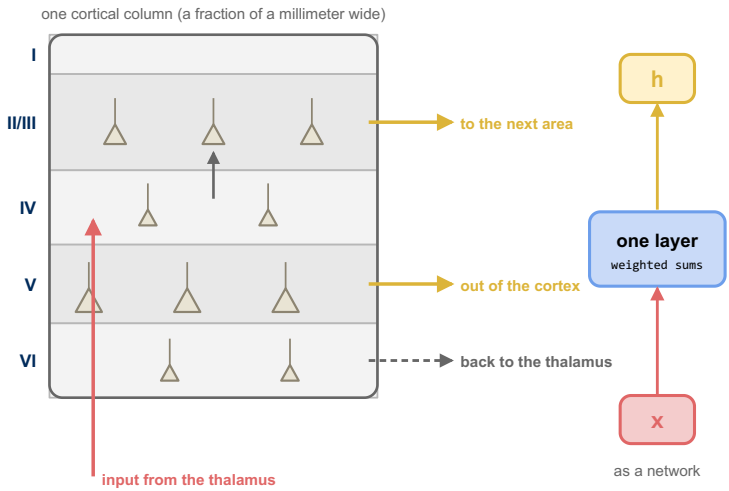


All three get there given enough capacity. **A fine enough staircase can follow any smooth curve** (or, with two inputs, any shape we could draw).

Unfortunately the theorem only says the right weights exist, not how many neurons it takes or how to find them. In practice several smaller layers do far more than one enormous one, the "deep" in deep learning.

Layers in the Brain

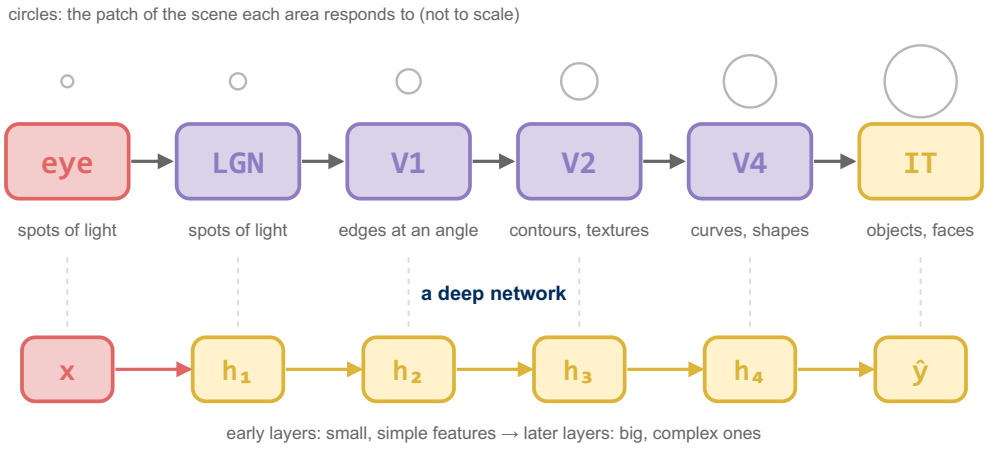
The brain seems to have reached the same conclusion about depth. Its outer layer, the **cortex**, is built from **cortical columns** a fraction of a millimeter wide (Vernon Mountcastle found them in **1957**, recording from cats), and each column has six layers of its own. Input arrives in the middle, in layer IV, gets passed up to layers II and III, and is sent on to the next area.



Those six layers are really **closer to one layer of our network** than six, since a signal goes in at the bottom, gets processed, and comes out once. Douglas and Martin (**1989**) went further and proposed that columns all over the cortex run the same basic circuit, a bit like a network built by stacking the same kind of layer over and over. So where does the brain's depth come from?

Depth in the Brain

From chaining areas together. What we see goes from the eye to the LGN, then through a series of cortical areas called V1, V2, V4 and IT, and **each area responds to a bigger patch of the scene, and to more complex patterns, than the one before**:



This is Hidden Features again, just with more layers. Each area computes its own features from the one before, so the next only has to do something fairly simple with them. Felleman and Van Essen (1991) counted 32 visual areas in the monkey brain, arranged in about 10 levels, which makes it a pretty deep network.

Networks as Models of Vision

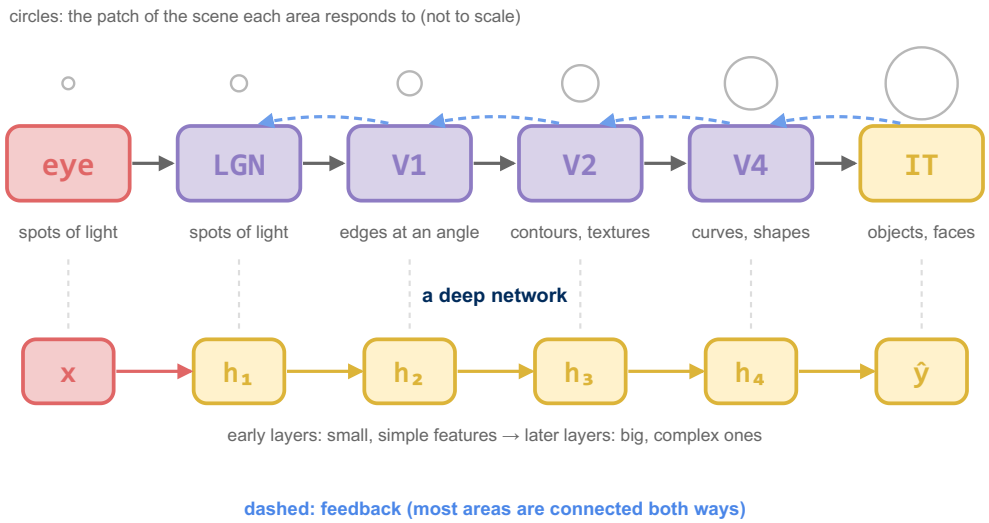
Neurons are slow (each one takes milliseconds to pass a signal on), which limits how many steps the brain can take. Thorpe, Fize and Marlot (1996) flashed photos for just 20 ms and asked people whether each one had an animal in it. Their brain signals for "animal" and "no animal" had already split apart about 150 ms after the photo appeared, only enough time for the signal to go up the hierarchy about once. **A first look at a scene seems to be mostly a forward pass**, with no time for loops or second guesses.

The similarity goes further than the wiring. Yamins et al. (2014) trained networks to recognize objects in photos, then compared them with recordings from neurons in monkeys looking at the same photos. The networks' middle layers predicted the responses of neurons in V4, their top layers predicted IT, and **the networks that were better at recognizing objects matched the brain better**.

The networks were never shown any brain data. They just learned to do the same job, and ended up with hidden features a lot like the brain's.

What Networks Leave Out

All that being said, an FFN is a very simplified brain. The biggest thing missing is **feedback**. Almost every connection between visual areas goes both ways, and only about 5–10% of the input to the LGN even comes from the eyes (most of the rest comes back down from the cortex). Our network only ever passes things forward, which is the "feed-forward" in its name.



Dendrites

Unfortunately a single real neuron isn't simple either. Its dendrites branch, and **each branch does its own non-linear sum** before the cell body adds them up, so the neuron is less like one of our neurons and more like a small network of them.

Poirazi, Brannon and Mel (2003) found that a model pyramidal neuron behaves like a **two-layer network**, every branch acting as a hidden neuron. Beniaguev, Segev and London (2021) trained networks to copy a detailed model of one cortical neuron spike for spike, and it took one 7 layers deep.

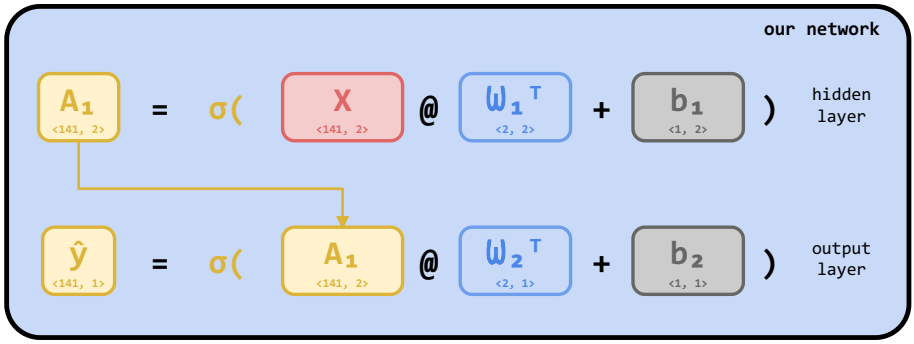
Real neurons also send all-or-nothing spikes whose timing matters, and they learn in ways nobody has fully worked out (more on that when we train networks). So **an FFN is a model of the brain's first, fast sweep**, and a rough one.

Layers as Matrices

Bigger networks need a compact way to be written down. Much like we stacked our games into a matrix X in Linear Regression, we can stack a layer's neurons into a weight matrix W, one row per neuron and one column per input, with their biases in a row b. A whole layer, for every game at once, is then one matrix multiplication:

[0.4] one fully connected layer, every game at once

$$Z = XW^T + b \qquad A = \sigma(Z)$$



Pictured above is our whole network this way, with all 141 games in X. W_1 is 2×2 (two neurons, two inputs), so $X\,W_1^T$ is 141×2, a row per game and a column per hidden neuron. The output layer has one neuron with two inputs, so W_2 is 1×2 and its transpose 2×1, which turns the hidden layer's 141×2 into 141×1, one $\hat{y}$ per game. **Getting the shapes to line up is most of the work** of writing a network down.

Activation Functions

Any non-linear function will do as the bend, the universal approximation theorem only needs one. Three are common, the **sigmoid**, the **hyperbolic tangent** (**tanh**) and the **Rectified Linear Unit** (**ReLU**):

[0.5] sigmoid

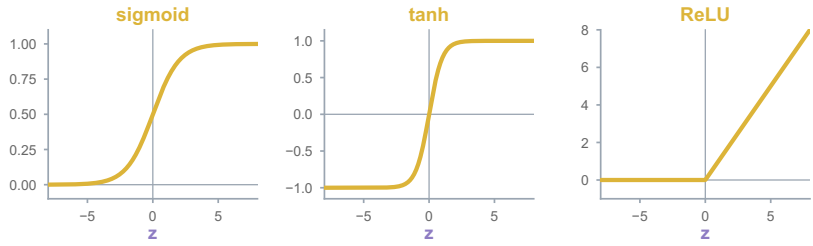
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

[0.6] hyperbolic tangent

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = 2\,\sigma(2z) - 1$$

[0.7] rectified linear unit

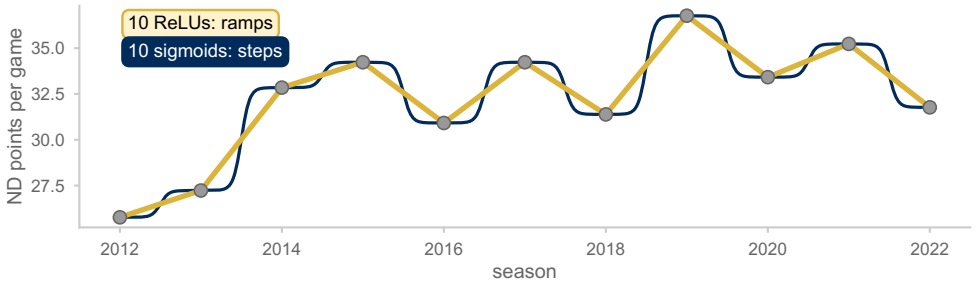
$$\mathrm{relu}(z) = \max(0, z)$$



The sigmoid is still the usual activation on the output of a binary classifier, since its output reads as a probability. Tanh is a sigmoid stretched to run from `-1` to `1`, centered on `0`, long the standard for hidden layers. Both are **soft steps**, which is why stacking them made a staircase on the last few pages. ReLU is a different shape altogether, `0` and then a straight ramp.

ReLU

ReLU builds with ramps instead of steps. Here are the same seasons with 10 ReLU neurons, one bend at each season, where each neuron's weight is how much the slope changes there:

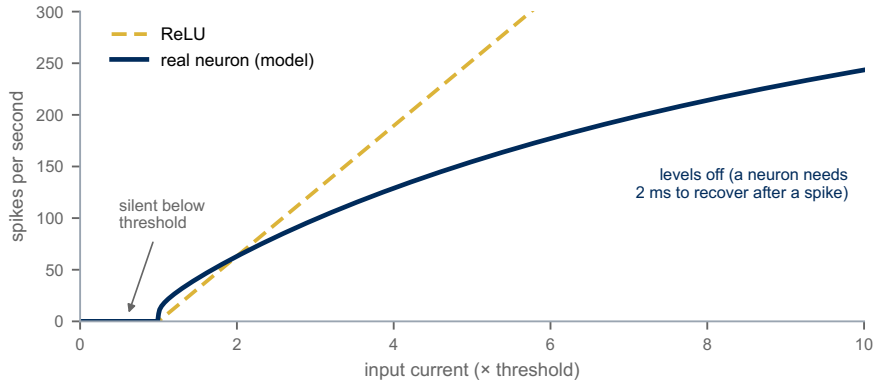


The result is the seasons joined by straight lines, **a curve made of straight pieces**, and with enough pieces that can be any curve. In Linear Regression a single straight line was all we had, here a network glues together as many as it needs. ReLU is also cheap (one comparison, no exponential) and, for reasons we'll see when we train networks, the most common activation in modern neural networks.

Our close-game network works with ReLUs too. $h_1 = \mathrm{relu}(x_1 - x_2 - 8)$ is how many points past 8 Notre Dame won by (and 0 otherwise), $h_2 = \mathrm{relu}(x_2 - x_1 - 8)$ is the same for losses, and with the same output neuron, $\sigma(2 - 4h_1 - 4h_2)$, it still gets all 141 games right. Notice that it needs no steepness factor, since any blowout makes h_1 or h_2 at least 1.

Firing Rates

ReLU also happens to be closer to a real neuron than the sigmoid is. A neuron stays silent until its input passes a threshold, then fires faster the harder it's pushed, and only levels off at a few hundred spikes a second (after every spike it needs a couple of milliseconds to recover).



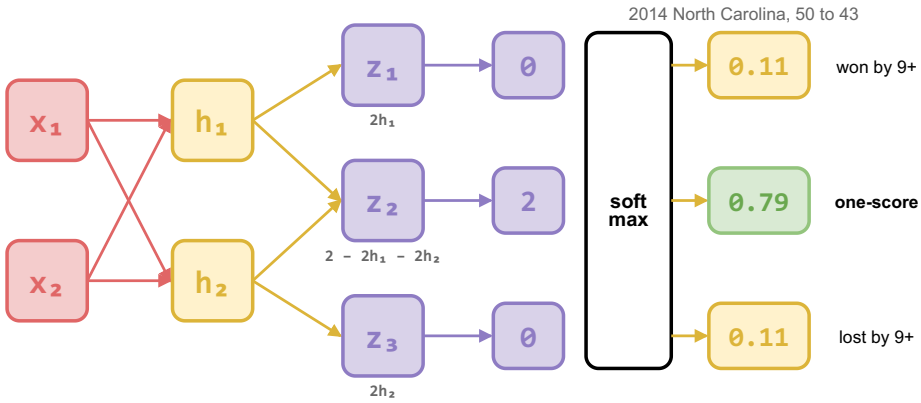
Pictured above is a standard model of this, the **leaky integrate-and-fire** neuron, next to a ReLU. Near the threshold the two are pretty close, and neither looks anything like a sigmoid, which sits at `0.5` when there's no input at all.

Glorot, Bordes and Bengio (`2011`) made this argument for ReLU, along with one more: only about `1`–`4`% of the brain's neurons are active at once. **A ReLU layer gets the same kind of sparseness for free**, since every neuron with a negative pre-activation outputs exactly `0`.

Multi-Class Classification

The output neuron only ever sees the hidden features, and nothing stops several output neurons from reading the same ones. That lets one network answer a question with more than two answers. What if we wanted to know whether Notre Dame won by more than 8, the game was within one score, or Notre Dame lost by more than 8? With our 8-point cut that's 70, 55 and 16 games.

A task with more than two possible labels is a **multi-class classification** task. Our output neuron can only answer one yes-or-no question, so we **give the output layer one neuron per outcome**:



The hidden layer is the same as before. Each output neuron scores one outcome, $z_1 = 2h_1$ for a win by more than 8, $z_2 = 2 - 2h_1 - 2h_2$ for a one-score game and $z_3 = 2h_2$ for a loss by more than 8. Whichever outcome actually happened gets a score of 2 and the other two get 0.

Softmax

Scores aren't probabilities. A sigmoid on each output neuron would give three numbers between `0` and `1`, but nothing would make them add up to `1`. Instead we use **softmax**, which exponentiates every score and divides by the total:

```
[0.8]
```

softmax

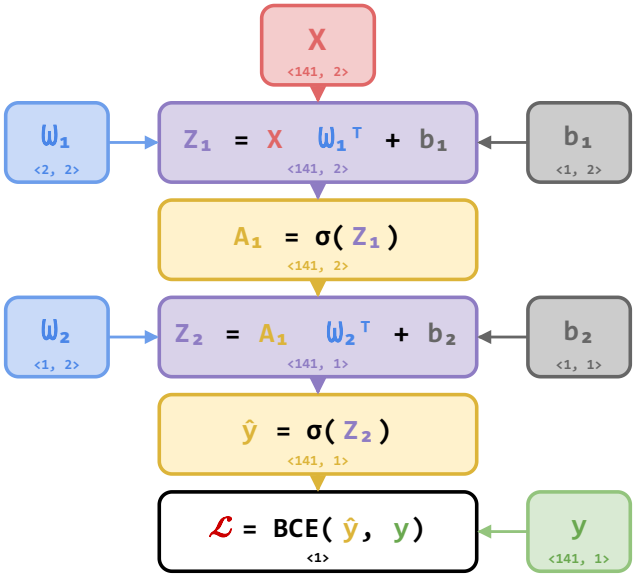
$$\mathrm{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

For `2014` North Carolina (`50` to `43`), `z` = `[0, 2, 0]` and softmax gives `0.11`, `0.79` and `0.11`, mostly a close game. For `2021` North Carolina (`44` to `34`), `z` = `[2, 0, 0]` gives `0.79` for a Notre Dame win by more than `8`. Taking the biggest probability as the answer, the network gets all `141` games right.

The biggest score always gets the biggest probability, so **softmax is the usual last-layer activation of a classifier**. With two outcomes it's exactly the sigmoid of the difference between the two scores, which is why we never needed it before. Training a network like this needs a loss for more than two outcomes (categorical cross-entropy), which comes in the next chapter.

Computation Graphs

To measure how well a network does we wrap a loss around its output, for our close-game question the binary cross-entropy from Logistic Regression (`0.13` on all `141` games). Each step of the forward pass, from X to the loss, uses the results of earlier steps. Drawing every step as a node and every dependency as an arrow gives the network's **computation graph**:



The forward pass computes each node from its parents, top to bottom. Since each node's output is the next node's input, the whole graph can also be written as **one statement of nested functions**:

`[0.9]` the loss as a single statement

$$\mathcal{L} = \mathrm{BCE}\big(\sigma\big(\sigma\big(X W_1^{\top} + b_1\big) W_2^{\top} + b_2\big),\ y\big)$$

Tensors

Three games by hand is fine, all `141` is tedious, and real networks have millions of weights. From here on we'll build networks with **PyTorch**, a Python library for neural networks first built at Facebook (now Meta). Its version of a numpy array is the **tensor**, and arithmetic works just as it does in numpy, `@` for matrix multiplication and `.T` for the transpose. Here's our hidden layer on three games (in code we write `W1` for W_1, since W is hard to type):

```
import torch
X = torch.tensor([[50., 43.], [38., 0.], [14., 42.]])
W1 = torch.tensor([[10., -10.], [-10., 10.]])
b1 = torch.tensor([-85., -85.])

A1 = torch.sigmoid(X @ W1.T + b1)  # b1 is added to every row
print(X.shape, W1.shape, b1.shape)
print(A1.round(decimals=3))
```

OUTPUT

```
torch.Size([3, 2]) torch.Size([2, 2]) torch.Size([2])
tensor([[0., 0.],
        [1., 0.],
        [0., 1.]])
```

`X @ W1.T` has shape `[3, 2]` and `b1` only `[2]`, so PyTorch adds `b1` to every row. Stretching the smaller tensor to fit like this is called **broadcasting**, and it means the same line works for any number of games. Unlike a numpy array, **a tensor can live on a GPU**, which is where real networks are trained.

Layers and Activations in PyTorch

We rarely write out `X @ W1.T + b1` ourselves. PyTorch has the fully connected layer pre-built as `nn.Linear`, and `nn.Linear(2, 2)` is a layer with `2` inputs and `2` neurons. Its `.weight` is our W_1, **stored one row per neuron** as we've been writing it, and its `.bias` is our b_1. The activation functions are built in too:

```python
import torch
import torch.nn as nn
torch.set_printoptions(precision=2)

fc1 = nn.Linear(2, 2)  # 2 inputs, 2 neurons
print(fc1.weight.shape, fc1.bias.shape)

print(torch.relu(torch.tensor([-3., 0., 5.])))
z = torch.tensor([[0., 2., 0.], [2., 0., 0.]])
print(torch.softmax(z, dim=1))  # one row per game
```

OUTPUT

```
torch.Size([2, 2]) torch.Size([2])
tensor([0., 0., 5.])
tensor([[0.11, 0.79, 0.11],
        [0.79, 0.11, 0.11]])
```

The two rows of `z` are our softmax examples, `2014` and `2021` North Carolina, and we get back the same `0.11`, `0.79` and `0.11`. `dim=1` tells softmax that each game's scores run along a row, so **each row adds up to `1`**, not each column.

Neural Networks in PyTorch

A whole network is a class with its layers in `__init__` and its forward pass in `forward`. Here's ours, run on all `141` games. Normally training sets the weights, so copying ours in under `torch.no_grad()` is a trick for this chapter only:

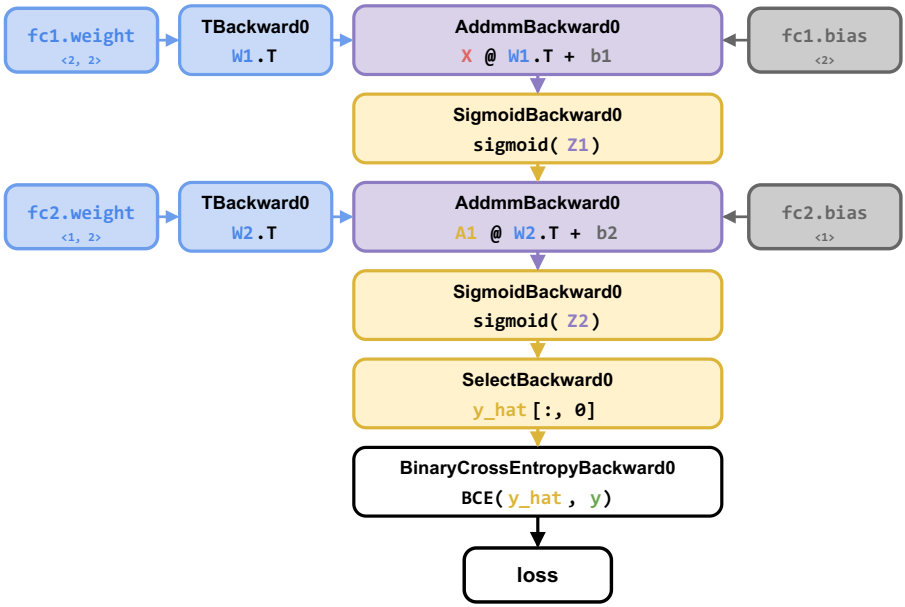
```
import torch
import torch.nn as nn
import pandas as pd
class CloseGame(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2, 2)
        self.fc2 = nn.Linear(2, 1)
    def forward(self, x):
        a_1 = torch.sigmoid(self.fc1(x))
        return torch.sigmoid(self.fc2(a_1))
model = CloseGame()
with torch.no_grad():  # our hand-picked weights
    model.fc1.weight[:] = torch.tensor([[10., -10.],
                                        [-10., 10.]])
    model.fc1.bias[:] = torch.tensor([-85., -85.])
    model.fc2.weight[:] = torch.tensor([[-4., -4.]])
    model.fc2.bias[:] = torch.tensor([2.])
games = pd.read_csv("nd_football.csv")
pts = games[["nd_points", "opponent_points"]].values
X = torch.tensor(pts, dtype=torch.float32)
close = (X[:, 0] - X[:, 1]).abs() <= 8
y_hat = model(X)[:, 0]
right = ((y_hat > 0.5) == close).sum().item()
print(right, "of", len(X))
```

OUTPUT

```
141 of 141
```

Training a Neural Network

Our network gets every game right only because we set its weights by hand, which gets us no further than Rosenblatt. What we need is the gradient of the loss with respect to every weight, so gradient descent can adjust them all at once. As a model runs, PyTorch **records every operation in a computation graph**:



The algorithm that walks this graph backwards, from the loss up to every weight, is **backpropagation**, made the standard way to train networks by David Rumelhart, Geoffrey Hinton and Ronald Williams in **1986** (the idea goes back to Linnainmaa in **1970**). It's the subject of the next chapter.