Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2022

Contents

CHAPTER 7

Neural Networks 1 - Neurons, Layers, and Networks

The tools we've developed so far are interesting, and **varied**. We've discussed:

- **Regression**: the problem of creating **real-number** outputs based on data.
- **Classification**: the problem of **sorting** data points into **categories**.
- Gradient **descent**: A technique for gradually **improving** your model using **calculus**.

These concepts are fascinating in their own right, and can be used to handle some **simple** problems. But, when they are **combined** together, we get something much more **powerful**: **neural networks**.

7.0.1 Machine Learning Applications

Neural networks in the modern area are used to tackle complex and challenging problems:

- Image labelling and generation
  - **Example:** **Recognizing** a **picture** of a dog. Or, **creating** a picture of a dog when prompted.
- Physics simulation
  - **Example:** **Simulating** water flow realistically, or special-effects smoke for a **movie**.
- Financial prediction

  - **Example:** Predicting how the **market** moves over time, and what the best **financial** choices in the present are.

- Text processing and generation
  - **Example:** Creating machines that can understand human text **prompts**, and writing useful **explanations** for humans.

- Data analysis
  - **Example:** **Compressing** data, or processing it to isolate the **important** aspects, without the noise.

As you can see, **neural networks** are used in a wide array of very **difficult** problems. No wonder it's rapidly becoming so popular!

7.0.2 Neural Network Perspectives: The brain

So, what *is* a neural network? Well, there's a **couple** ways of looking at it:

First, the name comes from the fact that NNs are inspired by the **brain**: we call the basic units of a neural network a **neuron**.

This gives us some general idea of the **structure** of a neural network:

Just like in the **brain**, we take many individual units, called **neurons**, which we connect together to do more **complicated** tasks. That combined structure is a **neural network**.

Funny enough, as effective as neural networks are, we now think they don't work very much like the human brain! But we keep the terminology.

Concept 1

Neural networks are inspired by the brain and its **neurons**, in an effort to do better, **human-like** computation.

Based on this, neural networks are **built** out of simple **units** called **neurons**, connected to each other.

7.0.3 Neural Network Perspectives: Classification and Regression

In this class, we **won't** focus on the brain analogy, though it did inspire the model.

Instead, we will mostly think of **neural networks** in terms of what they're able to do, and how they work.

One problem we have struggled with is certain tasks that can't be handled by **linear** models. We have used **feature representations** to work on this problem.

Simply, some problems are outside our **hypothesis** space. But, there's another way: this is where **neural networks** come in.

By combining lots of simple **units** ("neurons"), we can get a very **complex** model for solving our problems.

With such a **rich** hypothesis class, combined with the power of **gradient descent**, we can create a model that can do **classification** or **regression** for very difficult problems!

Reminder: "richness" or "expressiveness" of a hypothesis reflect how wide our options are. Neural networks give us many possibilities for models. With more options, we can handle more problems!

Concept 2

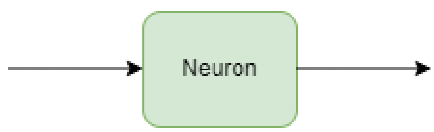
Neural Networks can create a very **rich hypothesis class** by combining many simple **units**.

With this **hypothesis class**, we can handle **regression** or **classification** for very challenging **problems**.

7.0.4 Building up a basic neural network

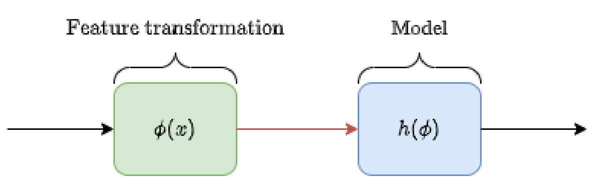
Let's make sense of what we said above, and **visualize** what a neural network might look like.

We start with one function: a **neuron**. This function could be, for example, one we've used before: our logistic **classifier**, or linear **regression**. We'll ignore the details for now.



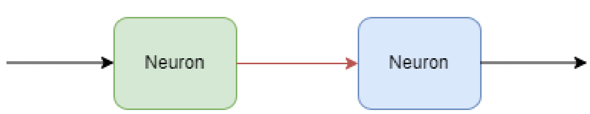
One neuron might not be very powerful, or **expressive**. It's useful, but limited. We've seen its weaknesses.

We could try to use **feature transformations** to help us. But, let's think in a more **general** way: a transformation is just another **function** we apply to our input!



This gives us an **idea**: rather than trying to think of a single, more **complex** model, we could **combine** multiple simple models!

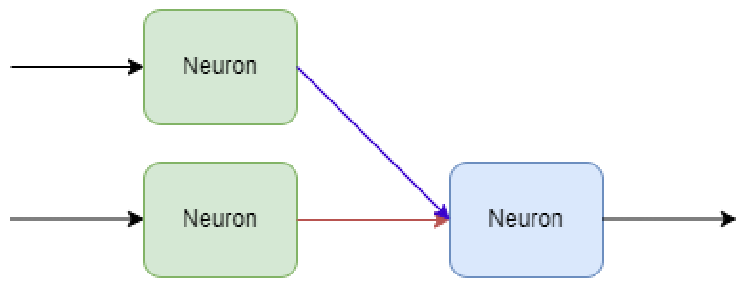
Note that feature transformations are a bit complex for what we'd usually put in a neuron. But, it gives us the right inspiration.



We could repeatedly add more neurons in **series**: each one being the input to another. And we'll do that later!

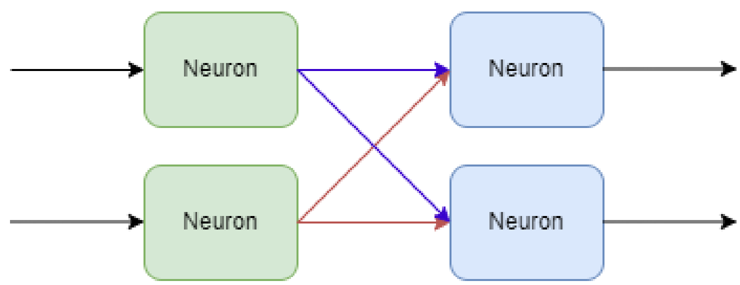
But, there's another type of **complexity** we haven't explored: we could have two neurons in **parallel**.

This parallel/series vocabulary is borrowed from circuits. We'll just use it for demonstration: you don't need to remember it.



Now, we have **two** neurons feeding into one output neuron! This already looks like a more **complicated** model.

We can go even further: what if we have two outputs as well?



Because we had two **inputs**, we had to add two new **links** when we added the output neuron. This is getting difficult to **view**!

We'll stop here for now, but you can imagine repeatedly **adding** more neurons in **parallel** (with the same inputs/outputs) or in **series** (as an input or output).

And we each addition, the function gets more and more **complex**: you can create a **richer** hypothesis class!

We'll explore how to do this **systematically** later in the chapter.

By "systematic", we just mean "in a way that's consistent and makes sense".

Definition 3

Neural Networks are a **class** of models that can be used to solve **classification**, **regression**, or other interesting problems.

They create very **rich** hypothesis classes by combining many **simple** models, called **neurons**, into a **complex** model.

We do this combination **systematically**, so that it is easy to **analyze** and work with our **model**.

This creates a very **flexible** hypothesis, which can be **broken down** into its **simple** parts and what **connects** them.

7.0.5 Neural Network Perspectives: Predictions with Big Data

Our last major **perspective** on neural networks is one that you see in lots of modern **applications**. We won't work much with this perspective in this **class**, but our techniques **enable** it.

Neural networks, because they can create such **sophisticated** models, can be used for problems in very **complex** domains: the kind of **applications** we discussed at the beginning of this chapter.

These applications require a lot of **data** to build a good **model**, however. So, machine learning models often take **huge** amounts of data, with lots of energy and time to train them.

But, once they are fully **trained**, they can give predictions very **quickly**, and often very **accurately**.

Concept 4

Neural networks can be seen as a way to make **predictions** based on huge amount of **data** for very **complex** problems.

7.1 Basic Element

Now, we have idea of what neural networks **are**. But, we have yet to handle the **details**:

- What **is** a neuron?
- How do we "systematically" **combine** our neurons?
- How do we **train** this, like we would a **simple** model?

We'll handle all of these steps and more - the above description was just to give a **high-level** view of what we want to **accomplish**.

Now, we go down to the **bottom** level, and think about just **one neuron**: what does it look like, and how does it work?

First, some terminology:

Notation 5

Neurons are also sometimes called **units** or **nodes**.

They are mostly **equivalent** names. They just reflect different **perspectives**.

7.1.1 What's in a neuron: The Linear Component

As we mentioned before, our goal is to combine **simple** units into a **bigger** one. So, we want a model that's **simple**.

Well, let's start with what we've done before: we've worked with the **linear** model

$$h(x) = \theta^T x + \theta_0 \tag{7.1}$$

This model has lots of nice properties:

- It limits itself to **addition** and **multiplication** (easy to compute)
- **Linearity** lets us prove some mathematical things, and use vector/**matrix** math
- The dot product between θ and x has a nice **geometric** interpretation.

This will make up the **first** part of our model.

> **Concept 6**
> Our **neuron** contains a **linear** function as its **first** component.

7.1.2 Weights and Biases

But, there's one minor **change**: before, we used θ because it represented our **hypothesis**.

But, every neuron is going to have its own **values** for its **linear** model:

$$\overbrace{f_1(x)}^{\text{Neuron 1}} = Ax + B \qquad\qquad \overbrace{f_1(x)}^{\text{Neuron 2}} = Cx + D \tag{7.2}$$

It wouldn't make much **sense** to call both A and C by the name θ.

We could use some clever **notation**, but why treat them as **hypotheses**? They are each only a **part** of our hypothesis Θ.

So, instead of thinking of each as a "hypothesis", let's switch perspectives.

Each value θ_k **scales** how much x_k affects the **output**: if we're doing

$$g(x) = 100x_1 + 2x_2 \tag{7.3}$$

Then, changing x_1 will have a much **bigger** effect on $g(x)$. Another way to say this is it **weighs** more heavily: it matters **more**.

Because of that, we call the number we scale x_1 by a **weight**.

Notation 7

A **weight** w_k tells you how heavily a **variable** x_k **weighs** into the output.

w_k is **equivalent** to θ_k: it's a **scalar** $w_k \in \mathbb{R}$.

$$\Big(\theta_1 x_1 + \theta_2 x_2\Big) \iff \Big(w_1 x_1 + w_2 x_2\Big)$$

We can combine it into a vector $w \in \mathbb{R}^m$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \qquad\qquad \theta^T x \iff w^T x$$

Remember that $a \iff b$ means a and b are equivalent!

What about our other term, θ_0? We call it an **offset**: it's the value we **shift** our linear model away from **origin**.

We'll use the same notation:

Notation 8

An **offset** w_0 tells you how far we **shift** $h(x)$ away from the origin.

w_0 is **equivalent** to θ_0: it's a **scalar** $w_0 \in \mathbb{R}$

$$\Big((\theta^T x) + \theta_0\Big) \iff \Big((w^T x) + w_0\Big)$$

We also sometimes call this the **threshold** or the **bias**.

Alternate notation: we might call this variable b, for bias.

This gives us our linear model using our new notation:

Definition 9

The **linear component** for a neuron is given by

$$z(x) = w^T x + w_0$$

where $w \in \mathbb{R}^m$ and $w_0 \in \mathbb{R}$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

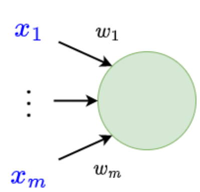
7.1.3 Linear Diagram

Now, we want to be able to depict our **linear** subunit. Let's do it piece-by-piece.

First, we have our vector $x = [x_1, x_2, ..., x_m]^T$:

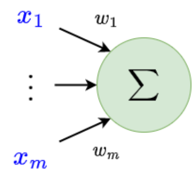
$$\begin{array}{c} x_1 \\ \\ \vdots \\ \\ x_m \end{array}$$

Now, we want to **multiply** each term x_i by its corresponding **weight** w_i. We'll combine them into a **function**:



The circle represents our function.

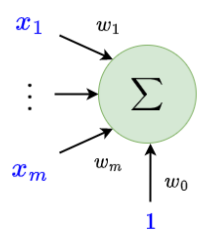
How are we combining them? Well, we're adding them together.



Note that we use the $\sum$ symbol, because we're **adding** after we **multiply**. In fact, we can write this as

$$w^T x = \sum_{i=1}^{m} w_i x_i \tag{7.4}$$

We'll include the bias term as well: remember that we can represent w_0 as $1 * w_0$ to match with the other terms.

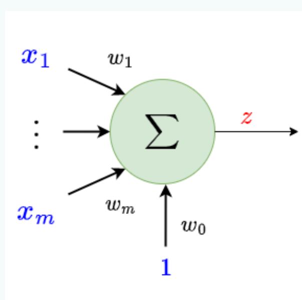


The blue "1" term is **multiplied** by w_0, just like how x_k gets multiplied by w_k.

We have our full function! All we need to do is include our output, z:

Notation 10

We can depict our linear function $z = w^T x + w_0$ as



Thus, z is a function of x:

$$z(x) = w^T x + w_0 \tag{7.5}$$

Which, in $\sum$ notation, we could write as

$$z(x) = \left(\sum_{i=1}^{m} w_i x_i\right) + w_0 \tag{7.6}$$

7.1.4 Adding nonlinearity

We'll continue building our neuron based on what we've done **before**. When doing linear regression, that linear unit was all we had.

But, once we do classification, we found that it was helpful to have a second, **non-linear** component: we used **sigmoid** $\sigma(u)$.

We might not necessarily want the **same** nonlinear function, so instead, we'll just generalize: we have *some* second component, which is allowed to be **nonlinear**.

We call this component our **activation** function. Why do we call it that? It comes from the historical **inspiration** of neurons in the brain.

Biological neurons only "fire" (give an output) above a certain threshold of **input**: that's when they **activate**.

Some activation functions reflect this, but they don't have to.

Definition 11

Our **neuron** contains a potentially **nonlinear** function f called an **activation function** as its **second** component.

We notate this as

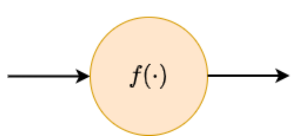
$$a = f(z) \tag{7.7}$$

Where z is the **output** of the **linear** component, and a is the **output** of the **activation** component.

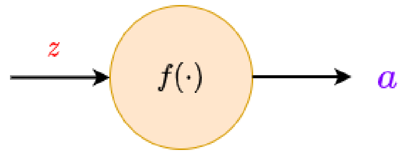
Note that z and a are **real numbers**: we have $f : \mathbb{R} \to \mathbb{R}$

7.1.5 Nonlinear Diagram

We'll depict a function f.



It takes in our **linear** output, z, and outputs our **neuron** output, a.



Note some vocabulary used for z:

Notation 12

z, the **output** of our **linear** function, is called the **pre-activation**.

This is because it is the result **before** we run the **activation** function.

And for a:

Notation 13

a, the **output** of our **activation** function, is called the **activation**.

7.1.6 Putting it together

So now, our neuron is complete.

Definition 14

Our **neuron** is made of

- A **linear** component that takes the neuron's input x, and applies a linear function

$$z = w^T x + w_0$$

  - The **pre-activation** z is the **output** of the **linear** function.
  - It is also the **input** of the **activation function** f.

- A (potentially nonlinear) **activation** component that takes the pre-activation z and applies an **activation function** f:

$$a = f(z)$$

  - The **activation** a is the **output** of this **activation function**.

When we **compose** them together, we get

$$a = f(z) = f\Big(w^T x + w_0\Big)$$

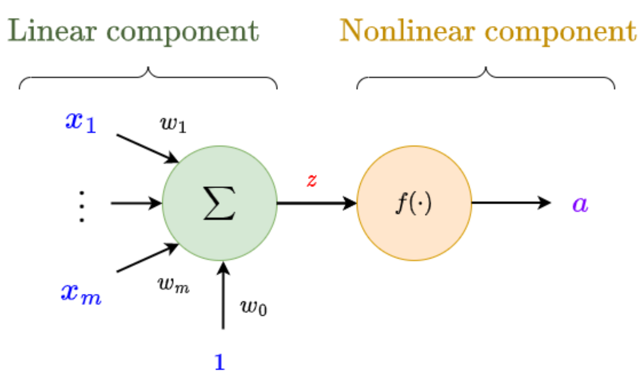
We can also use $\sum$ notation to get:

$$\color{purple}{a} = f(\color{red}{z}) = f\left(\left(\sum_{i=1}^{m} w_i \color{blue}{x_i}\right) + w_0\right)$$

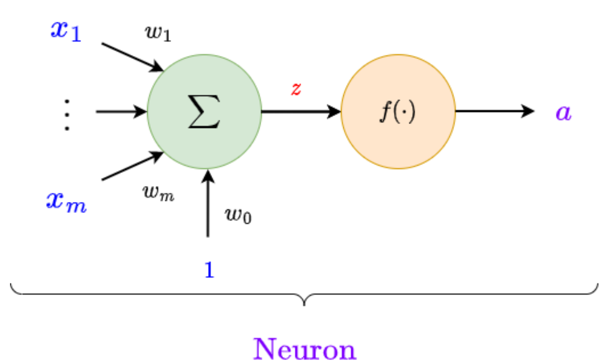
When we say "compose", we mean **function composition**: combining $f(x)$ and $g(x)$ into $f(g(x))$.

7.1.7 Neuron Diagram

Finally, we can **compose** our neuron into one big **diagram**:

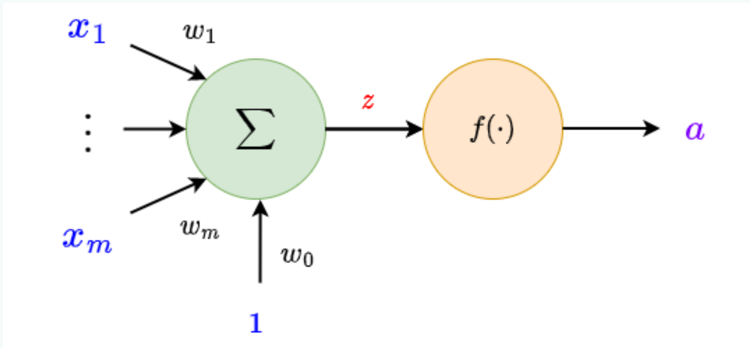


From here on out, we'll treat this as a **single** object:



Notation 15

We can depict our **neuron** $f(w^T x + w_0)$ as



- x is our **input** (neuron input, linear input)
- z is our **pre-activation** (linear output, activation input)
- a is our **activation** (neuron output, activation output)

This neuron will be the **basic unit** we work with for the rest of this **chapter** - it's one of the most **important** objects in all of machine learning.

7.1.8 Our Loss Function

One more detail: we will want to **train** these neurons. In order to be able to **measure** their performance, we'll need a **loss** function.

This **isn't** any different from usual: we just need a **function** of the form

$$\mathcal{L}(g, y) \tag{7.8}$$

In **regression**, we wrote our loss as

$$\mathcal{L}\Big(\; h(x;\Theta), \;\; y \; \Big)$$

The right term, $y^{(i)}$, is unchanged: we still need to compare against the **correct** answer.

The main change is we aren't using Θ notation: we'll **replace** it with (w, w_0)

$$\mathcal{L}\left(\; h\Big(x; (w, w_0)\Big), \;\; y \; \right)$$

And finally, we get the loss for multiple data points:

We skip doing $1/n$ averaging because we often use this for SGD: we plan to take small steps as we go, rather than adding up our steps all at once.

$$\sum_i \mathcal{L}\left(h\left(x^{(i)}; (w, w_0)\right), \; y^{(i)} \right)$$

And with this, not only is our neuron **complete**, but we have everything we need to **work** with it.

Concept 16

For a **complete neuron**, we need to specify

- Our **weights** and **offset**
- Our **activation** function
- Our **loss** function

From here, we could do **stochastic gradient descent** as we usually do, to **optimize** this neuron's **performance**.

7.1.9 Example: Linear Regression

Let's go through some **examples**. We mentioned in the **beginning** of this chapter that our neuron could be most of the simple **models** we've worked with.

So, let's give that a go: we'll start by doing **linear regression**.

$$h(x) = \theta^T x + \theta_0$$

This model is exclusively **linear**: we just have to replace θ with w.

$$z(x) = w^T x + w_0$$

So, our linear component is **done**: $(\theta, \theta_0) = (w, w_0)$.

What about our **activation** function?

Well, activation allows for **nonlinear** functions. But, we don't **want** to make it nonlinear.

In fact, we've already got what we **want**: we don't want the **activation** to do anything at **all**.

So, we'll use **this** function:

Concept 17

The **identity function** $f(z)$ is a function that has no **effect** on your **input**.

$$f(z) = z$$

By "having no effect", we mean that the input is **unchanged**: this is true even if your input is **another function**:

$$f(g(x)) = g(x) \tag{7.9}$$

We call it the "identity" because the input's identity is unchanged!

So, the **identity** function is our activation function: it keeps our **linearity**.

Concept 18

Linear Regression can be represented with a **single neuron** where

- We keep our **linear component**, but set $(\theta, \theta_0) = (w, w_0)$.

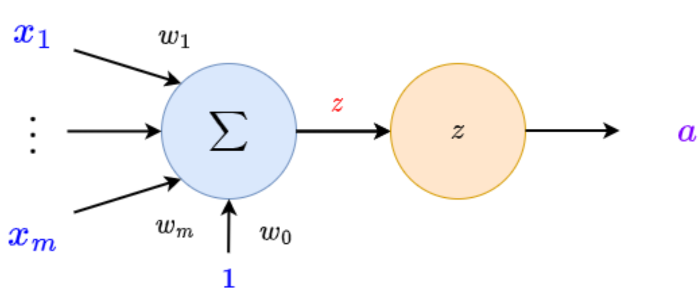
$$z(x) = w^T x + w_0$$

- Our **activation function** is the **identity** function,

$$f(z) = z$$

- Our **loss function** is **quadratic loss**.

$$\mathcal{L}(a, y) = (a - y)^2$$



7.1.10 Example: Linear Logistic Classifiers

Now, we do the same for LLCs: it's already broken up into **two** parts in our **classification** chapter.

First, the **linear** component. This is the same as linear regression:

$$z = \theta^T x + \theta_0 \tag{7.10}$$

And then, the **logistic** component:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{7.11}$$

This second part is **nonlinear**: its our **activation** function!

Concept 19

A **Linear Logistic Classifier** can be represented with a **single neuron** where

- We keep our **linear component**, but set $(\theta, \theta_0) = (w, w_0)$.

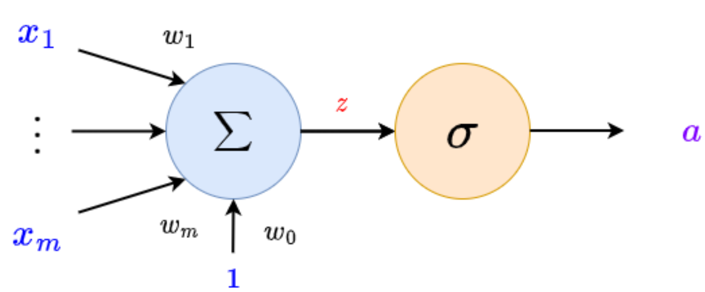
$$z(x) = w^T x + w_0$$

- Our **activation function** is the **sigmoid** function,

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

- Our **loss function** is **negative-log likelihood** (NLL)

$$\mathcal{L}_{nll}(a, \quad y^{(i)}) = -\left(y^{(i)} \log a + \left(1 - y^{(i)}\right) \log\left(1 - a\right)\right)$$



7.2 Networks

Now, we have fully developed the individual **neuron**.

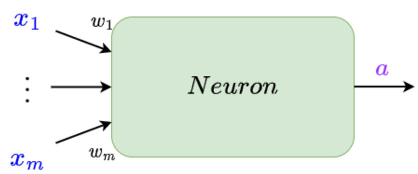
We can even do **gradient descent** on it: just like when we were doing LLCs, we can use the **chain rule**.

We'll get into this more, later in the chapter.

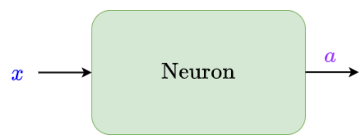
So, we return to the idea from the beginning of this chapter: combining multiple neurons into a **network**.

7.2.1 Abstraction

For this next section, we'll **simplify** the above diagram to this:



In fact, for more **simplicity**, we'll draw **one** arrow to represent the whole vector x. However, nothing about the **actual** math has changed.



This is also called **abstraction** - we need it a lot in this chapter.

Definition 20

Abstraction is a way to view your system more **broadly**: removing excess details, to make it **easier** to work with.

Abstraction takes a **complicated** system, and focuses on only the **important** details. Everything else is **excluded** from the model.

Often, this **simplified** view boils a system down to its the **inputs** and **outputs**: the "interface".

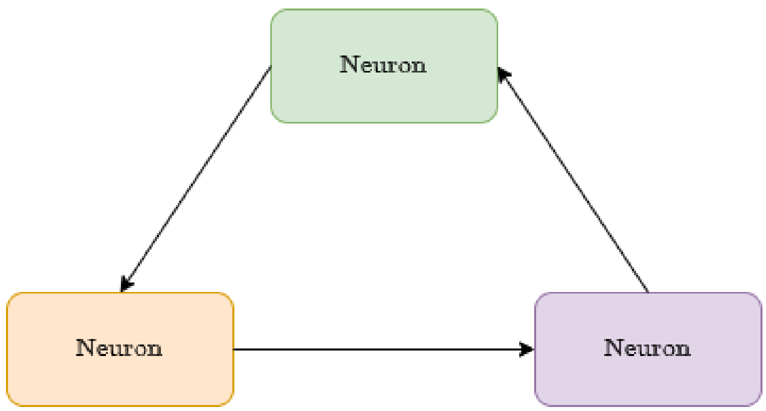
Example: Rather than thinking about all of the **mechanics** of how a car works, you might **abstract** it down to the pedals, the steering wheel, and how that causes the car to move.

7.2.2 Some limitations: acyclic networks

We won't allow for just **any** kind of network: we can create ones that might be unhelpful, or just very **difficult** to **analyze**.

For now, we can get interesting and **useful** behavior while keeping it **systematic**. We'll define this "system" later.

We'll assume our networks are **acyclic**: they do not create closed **loops**, where something can affects it own input.



This is a cyclic network: this is messy and we won't worry about this for now.

This means information only **flows** in one direction, "forward": it never flows "backwards".

Concept 21

For simple **neural networks**, we assume that they are **acyclic**: there are no **cycles**, or loops.

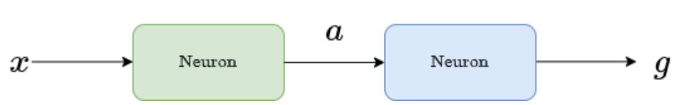
This means that **no neuron** has an output that affects its **input**, directly or indirectly.

We call these **feed-forward** networks.

We'll show how to build up the rest of what we need.

7.2.3 How to build networks

Suppose we have two neuron in **series**, our **simplest** arrangement:

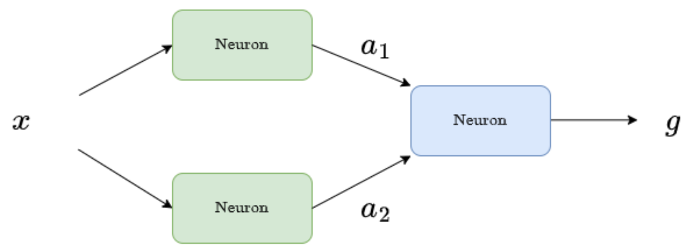


Our first neuron takes in a whole **vector** of values, $x = [x_1, x_2, ..., x_m]^T$. But, it only **outputs** a single value, a.

Remember that while we only see one arrow from x, each data point x_i is included.

That means the second neuron only receives **one** value, but it's capable of handling a full **vector**. We can add more values!

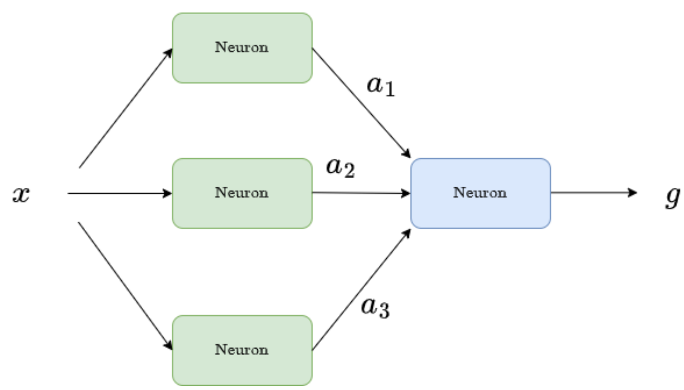
Let's add **another** neuron.



Our rightmost neuron now has **2 inputs**, which can be stored in a vector,

$$A = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \tag{7.12}$$

We could increase the **length** of this vector by adding more **neurons**.



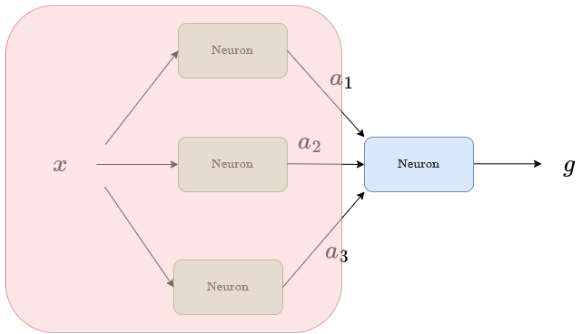
$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{7.13}$$

For our **rightmost** neuron, this is effectively the **same** as x: an **input vector**.

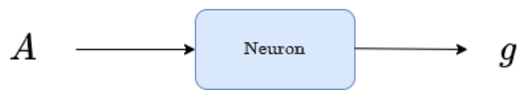
7.2.4 Layers

This gives us an idea for how to **build** our network: using multiple neurons in **parallel**, we can output a new vector A!

This is useful, because it means we can **simplify**: from the rightmost neuron's perspective, it just sees that **vector** as an input.



We can take this entire layer...



And just reduce it down to the vector A.

Because it's so useful, we'll give this set of neurons a name: a **layer**.

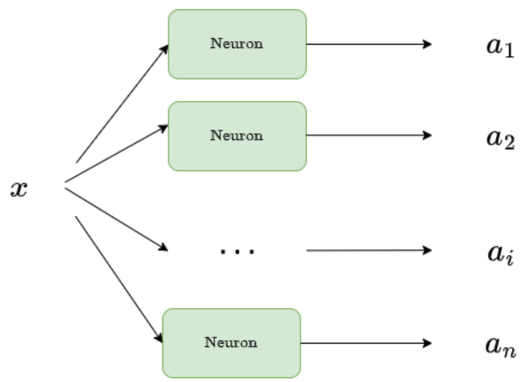
Definition 22

A **layer** is a set of **neurons** that are in "parallel":

- They all have **inputs** from the same **previous layer**
  - This **previous layer** could also be the **original input** x.
- They all have **outputs** to the same **next layer**
  - This **next layer** could also be the **final output** of the neural network.
- And none of these neurons are directly **connected** to each other.

This **layering** structure allows us to simplify our **analysis**: anything that comes after the layer only has to work with a **single vector**.

A layer in general might looks like this:



A general layer in a neural network.

7.2.5 The Basic Structure of a Neural Network

We could pick many structures for neural networks, but for simplicity, this will define our **template** for this chapter.

Definition 23

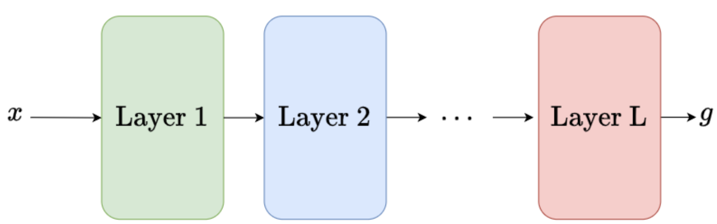
We structure our **neural networks** as a series of **layers**, where each layer is the **input** to the next layer.

This means that **layers** are a basic unit of a neural network, one level above a **neuron**.

In short, we have:

- A **neuron**, made of a linear and an activation component
- A **layer**, made of many neurons in parallel
- A **neural** network, made of many layers in series

Our goal is some kind of structure that looks something like this:

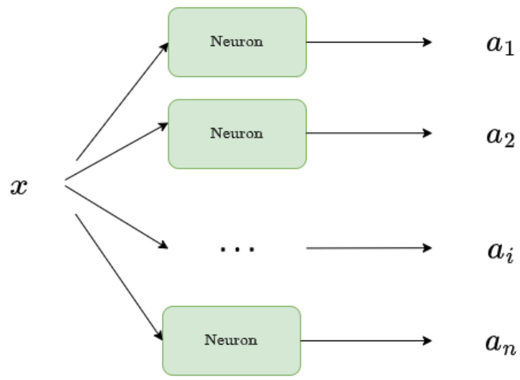


A neural network.

We now have a high-level view of our entire neural network, so now we dig into the details of a single layer.

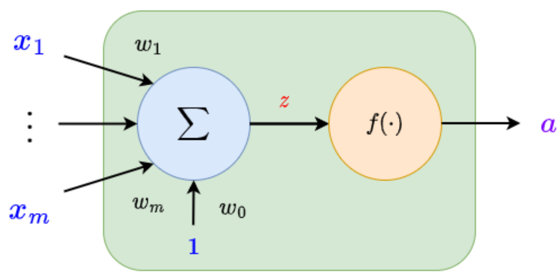
7.2.6 Single Layer: Visualizing our Components

Now, rather than analyzing a single neuron, we will analyze a single **layer**.



Our first layer.

In order to **analyze** this layer, we have to open back up the **abstraction**:

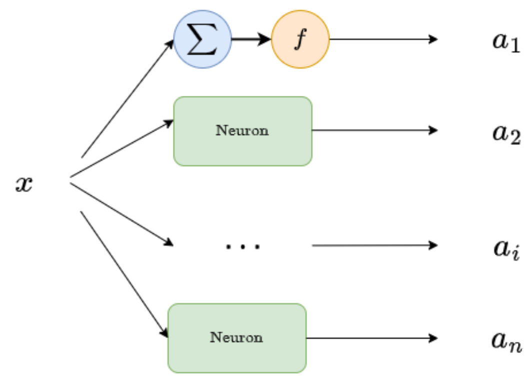


Each of those neurons looks like this.

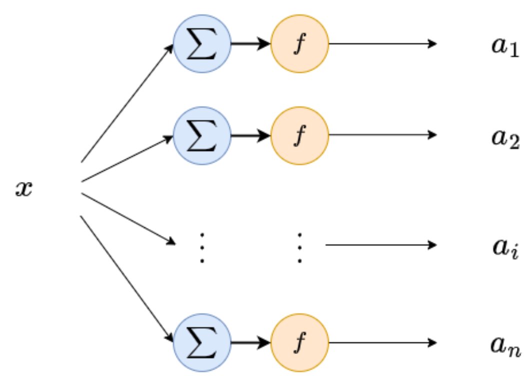
There are two important pieces of **information** we're hiding:

- We have two components inside of our neuron.
- We have many inputs x_i for one neuron.

The first piece of information is easier to visualize: we just replace each neuron with the two components.



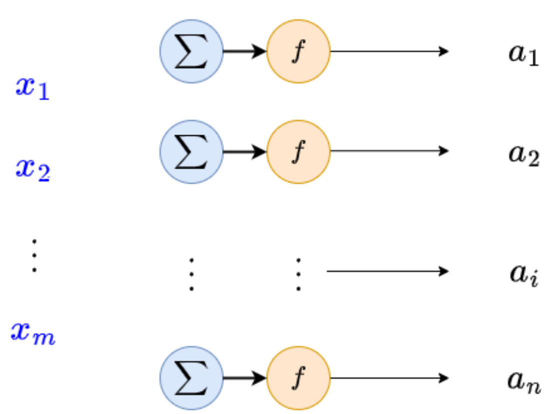
Replacing one neuron...



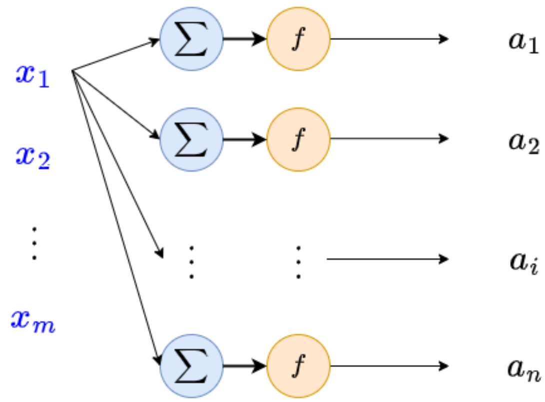
Replacing all neurons!

7.2.7 Single Layer: Visualizing our Inputs

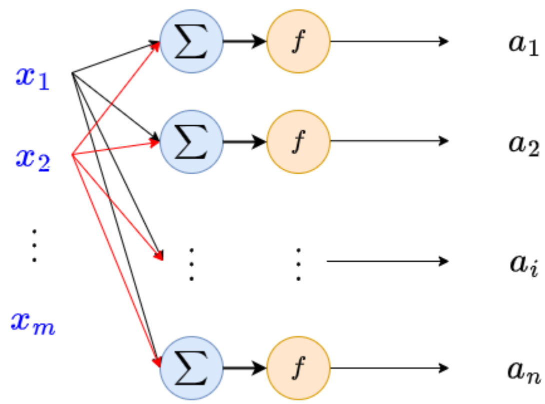
The second piece of information is much more difficult: we show all of the x_i outputs.



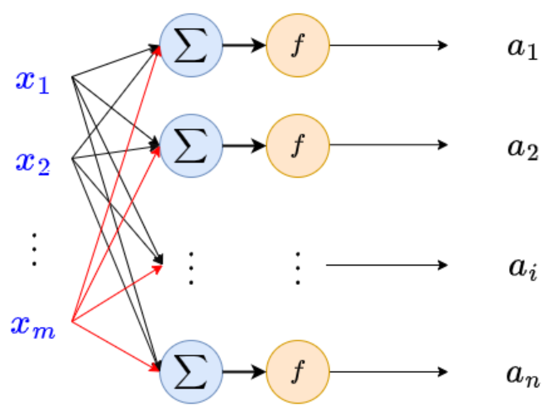
Now we have to draw the arrow for each input.



Every neuron receives the first input.



Every neuron receives the second input, too. This is getting messy...



The completed version: this is hard to look at.

Don't worry if this looks **confusing**! It's natural for it to be **hard** to read: the only thing you need to know is that we pair **every** input with **every** neuron.

This is our **final** view of this layer: because each of our m inputs has to go to every of n neurons, we end up with mn different **weights**.

This is a ton of **information**, and its only one layer! This shows how **complex** a neural network can be, just by **combining** simple neurons.

Note that this is a **fully connected** network: not all networks are FC.

> **Definition 24**
> A layer is **fully connected** if every neuron has the **same input vector**.

Example: If one of our neurons **ignored** x_1, but the others did **not**, the layer would not be **fully connected**.

7.2.8 Dimensions of a layer

Now that we've seen the **full** view, we can **analyze** it. Our goal is to create a more **useful** and **accurate** simplification.

Our first point: note that the input and output have a **different** dimensions!

> **Clarification 25**
> A **layer** can have a different **input** and **output** dimension. In fact, they are completely **separate** variables.

This is because **every** input variable is allowed to be applied to the **same** neuron:

Example: You can have one neuron of the form

$$z = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + w_0$$

In this case, our neuron has **one** output variable $f(z)$, but **three** inputs x_1, x_2, x_3.

Thus, our output dimension has been separated from our input dimension. Instead, it is the number of neurons.

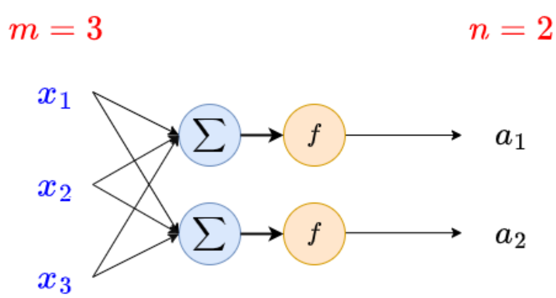
So, in general, we can say:

Notation 26

A **layer** has two associated **dimensions**: the **input** dimension m and the **output** dimension n.

- The **input** dimension m is based on the vector output from the **previous layer**: $x \in \mathbb{R}^m$
- The **output** dimension n is equal to the **number of neurons** in the **current** layer: $A \in \mathbb{R}^n$

Example: Suppose you have an **input** vector $x = [x_1, x_2, x_3]$ and two **neurons**. The dimensions are $m = 3$, and $n = 2$.



The input dimension and output dimensions are **separate**.

7.2.9 The known objects of our layer

So, we know we have two objects so far:

- Our **input** vector $x \in \mathbb{R}^m$
- Our **output** vector $A \in \mathbb{R}_n$

Where they each take the form

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \qquad A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \tag{7.14}$$

But, there are a couple other things we haven't **generalized** for our entire **layer**:

- Our weights
- Our offsets
- Our preactivation

7.2.10 The other variables of our layer: weights and offsets

First, our **weights**: each neuron has its own vector of weights $w \in \mathbb{R}^m$.

The dimension needs to match x so we can compute $w^T x$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \tag{7.15}$$

To distinguish them from each other, we'll represent the i^{th} neuron's weights as $\vec{w}_i$.

$$\vec{w}_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \tag{7.16}$$

Each needs to be used to **compute** a_i, but having so many objects is annoying.

Remember that, when we had **multiple** data points $x^{(i)}$, we worked with them at the **same time** by stacking them in a **matrix**. Let's do the same here:

$$W = \overbrace{\begin{bmatrix} \vec{w}_1 & \vec{w}_2 & \cdots & \vec{w}_n \end{bmatrix}}^{\text{Each neuron has a weight vector}} \tag{7.17}$$

If we expand it out, we get a full matrix...

$$W = \left.\overbrace{\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m1} & \cdots & w_{mn} \end{bmatrix}}^{n \text{ neurons}}\right\} m \text{ inputs} \tag{7.18}$$

This is our **weight matrix** W: it's an $(m \times n)$ matrix. It contains all of our mn weights, sorted by

- **Input variable** (row)
- **Neuron** (column)

We can do this for our **offsets** too: thankfully, there is only **one** offset per neuron, so we can write:

$$W_0 = \underbrace{\begin{bmatrix} w_{01} \\ w_{02} \\ \vdots \\ w_{0n} \end{bmatrix}}_{\text{Each neuron has an offset}} \tag{7.19}$$

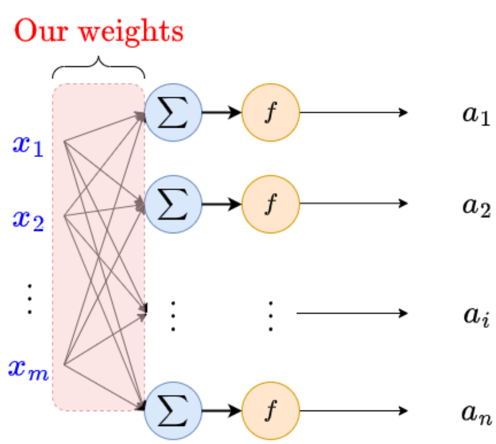
This is our offset vector, with the shape $(n \times 1)$.

Notation 27

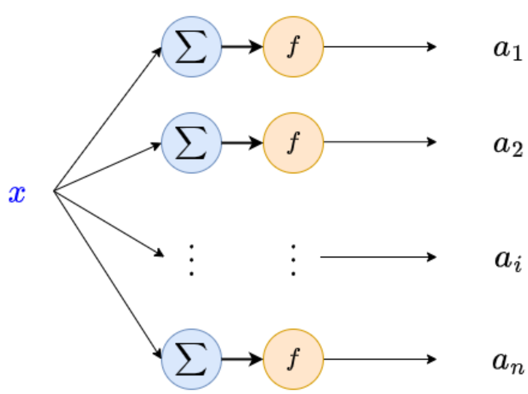
We can store our **weights** and **offsets** as **matrices**:

- **Weight** matrix W has the shape $(m \times n)$
- **Offset** matrix W_0 has the shape $(n \times 1)$

These matrices give us a tidy way to understand all of this mess:



Now that we understand it, we'll **hide** those weights again, for readability.



7.2.11 Pre-activation

Now, all that remains is the pre-activation z.

Before, we did

$$w^T x + w_0 = z \tag{7.20}$$

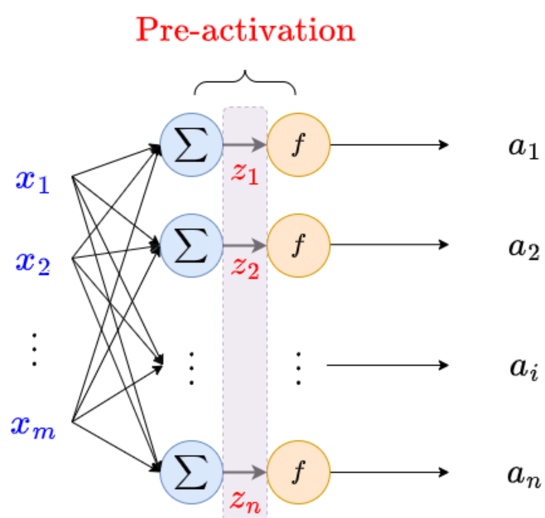
Because we so carefully kept our weights and offsets separate, we can still do this!

$$W^T x + W_0 = Z \tag{7.21}$$

This pre-activation vector Z contains all of the outputs of our linear components:

$$Z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \tag{7.22}$$

On our diagram, we can see it here:



This section is what Z details with.

And we can connect this to our activation: each a_i is the result of running our function f on z_i:

Because we run the function on each element in Z, we call this an **element-wise** use of our function.

$$A = f(Z) = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix} \tag{7.23}$$

7.2.12 Summary of layer

So, we can now break our our layer up into pieces:

Notation 28

Our **layer** is a **function** that takes in $x \in \mathbb{R}^m$, and returns $A \in \mathbb{R}^n$.

It is defined by:

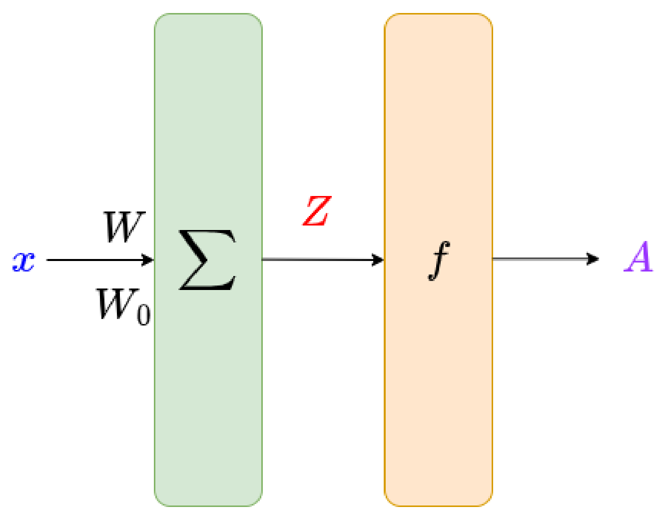
- **Dimensions**: m for **input**, n for **output** (number of neurons)

And our different **matrices**:

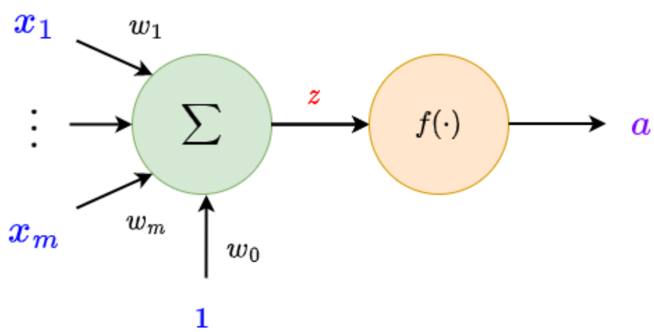
- **Input**: a **column vector** X in the shape $(m \times 1)$
- **Weights**: a **matrix** W in the shape $(m \times n)$
- **Offset**: a **column vector** W_0 in the shape $(n \times 1)$
- **Pre-activation**: a **column vector** Z in the shape $(n \times 1)$
- **Activation**: a **column vector** A in the shape $(n \times 1)$

We've now accomplished our goal: **simplify** the layer into its **base** components, without losing any crucial **information**.

We've can represent an entire layer like this:



Note how similar this looks to a **single** neuron: this works because the neurons in a **layer** are in **parallel**!



The math is very similar as well:

Definition 29

Our **layer** can be represented by

- A **linear** component that takes in x, and outputs **pre-activation** Z:

$$Z = W^T x + W_0$$

- A (potentially nonlinear) **activation** component that takes in Z, and outputs **activation** A:

$$A = f(Z)$$

When we **compose** them together, we get

$$A = f(Z) = f\Big(W^T x + W_0\Big)$$

7.2.13 The weakness of a single layer

What can we do with a single layer? Well, our **LLC** model gives us an example: it has the **nonlinear** sigmoid activation, but acts as a **linear** separator.

Why is that? Why is the separator still linear, if the **activation** isn't?

Well, let's take the **linear** separator created by the pre-activation:

$$z = w^T x + w_0 = 0 \tag{7.24}$$

This is our **boundary** for just a linear function. But adding the nonlinear activation should make it more **complex**, right?

Well, it turns out, we can represent our **activation** boundary with a **linear** boundary.

Example: Continue our LLC example. If $z = 0$, then $\sigma(z) = \sigma(0)$. Our boundary is

$$\sigma(z) = \sigma(0) = \frac{1}{2} \tag{7.25}$$

Wait. But that means that $\sigma(z) = .5$ is the same as $z = 0$: the same inputs x cause both of them, so they have the same boundary!

$$\text{Linear boundary } z = 0 \iff f(z) = \frac{1}{2} \tag{7.26}$$

Summary:

- $\sigma(z) = .5$ is the **same** as $z = 0$.
- $z = 0$ is **linear**.
- Thus, our sigmoid boundary is **linear**.

We can apply this to other activation functions. In general, any constant boundary for most $f(z)$ is equivalent to some linear boundary $z = C$:

Assuming that f is invertible, which it often is.

$$z = C \qquad \iff \qquad f(z) = f(C) \tag{7.27}$$

Since $z = C$ is linear, we know that our activation separator $f(x) = f(C)$ is linear too.

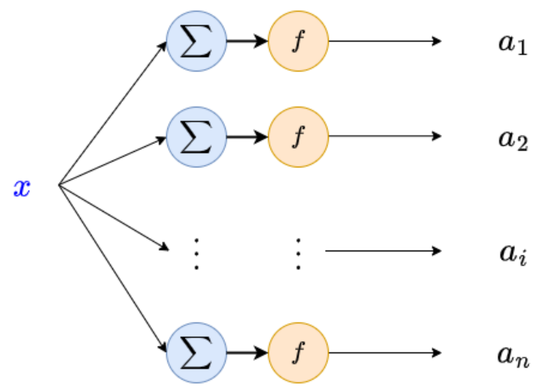
Concept 30

A single neuron creates a **linear separator**, even if it has a **nonlinear** activation.

This is because any **boundary** for $f(z)$ we can create, can be represented by some **linear** boundary in z.

There are exceptions, but this is true for most useful activation functions.

It turns out, adding more neurons **within** the layer doesn't change much: because they act in **parallel**, each neuron acts separately, and the things we said above are still **true** for each output a_i.



Each of these neurons has the same input, x.

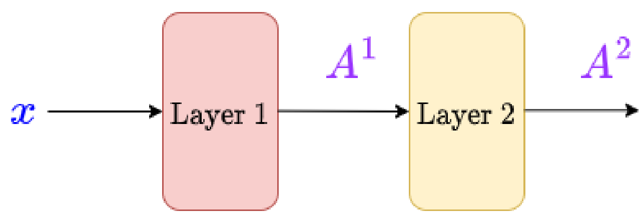
So, in order to create nonlinear behavior, we need at least two layers of neurons in **series**.

So, we'll start **stacking** layers on each other: each layer **feeds** into the next one.

> **Concept 31**
> A **single layer** of neurons has **linear** behavior.
>
> We need **multiple** layers to get a nonlinear **neural network**.

7.2.14 Adding a second layer

So, let's add one more **layer**. We'll label layers by using a **superscript**: W^1 is the set of **weights** for the **first** layer, for example.



We have two separate outputs: A^1 and A^2.

> **Clarification 32**
> **Superscripts** in our notation indicate the **layer** that our value is associated with.
>
> They do **not** represent exponentiation!

Example: Z^3 would be the **pre-activation** for layer 3: it is **not** Z "cubed".

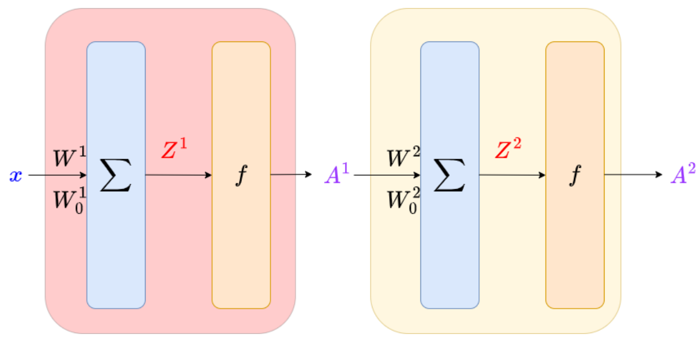
What can we learn from this?

- The **output** of layer 1, A^1, is the **input** to layer 2.

- Thus, the output dimension n^1 of layer 1 must **match** the input m^2 of layer 2:

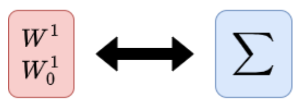
$$n^1 = m^2 \tag{7.28}$$

Let's break these into their components again.



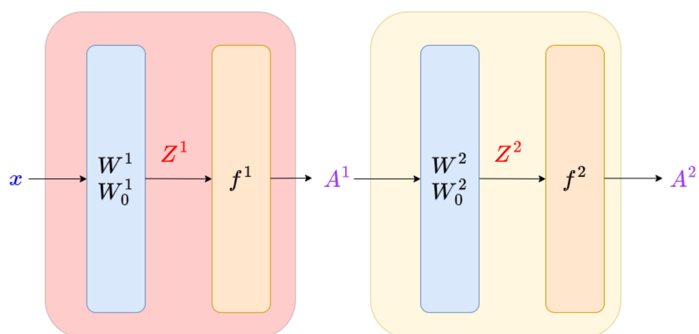
We have two separate outputs: A^1 and A^2.

To distinguish between the linear functions in each layer, we'll just notate them using the weights and offsets.



These two are equivalent (if in the same layer)! We'll use the notation on the left, so that you know which layer our unit is in.

And this gives us:



Now, we can make our functions. For layer one:

$$A^1 = f(Z^1) = f\Big((W^1)^T x + W_0^1\Big) \tag{7.29}$$

And layer two:

$$A^2 = f(Z^2) = f\Big((W^2)^T A^1 + W_0^2\Big) \tag{7.30}$$

We can use this to build our **general** pattern.

7.2.15 Many Layers

We are finally ready to build our **complete** neural network. We'll just retrace the steps of the 2-layer case.

Notation 33

The total **number** of **layers** in our **neural network** is notated as L.

Typically we notate an **arbitrary** layer as ℓ (or l).

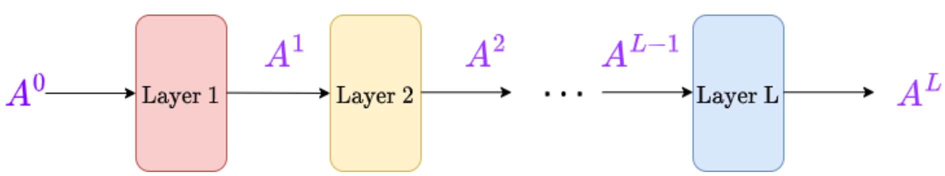
Since x is, for all purposes, **equivalent** to a vector A, we will call it A^0.

Notation 34

Our **neural network**'s input x is used in the **same** way as every term A^ℓ.

So, we will **represent** it as

$$x = A^0$$



Again, we see that the **output** of layer ℓ is the **input** of layer $\ell + 1$.

Concept 35

Each layer **feeds** into the next layer.

A^ℓ is the **output** of layer ℓ, and the **input** of layer $\ell + 1$.

This means that the **output** dimension must **match** the next **input** dimension.

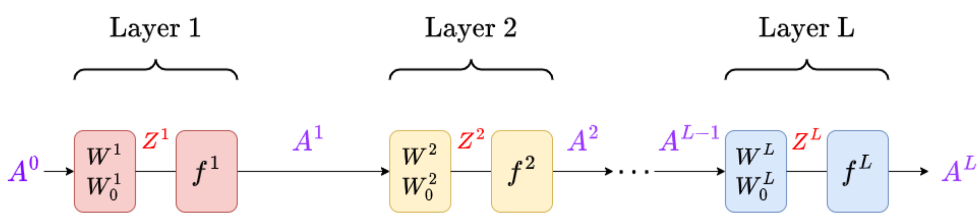
$$\overbrace{n^\ell}^{\text{Output}} = \overbrace{m^{\ell+1}}^{\text{Output}}$$

And the **dimension** of A^ℓ is $(n^\ell \times 1) = (m^{\ell+1} \times 1)$.

7.2.16 Our Complete Neural Network

We can break our layers into components, so we can see the functions involved.

With this, we build our final neural network:



With this, we can see how each layer is **related** to each other: as we **mentioned**, the **output** of one layer is the **input** of the next layer.

Here is the computation we do for layer ℓ:

Key Equation 36

The calculations done by layer ℓ are given by

$$Z^\ell = (W^\ell)^T A^{\ell-1} + W_0^\ell$$

and

$$A^\ell = f(Z^\ell)$$

Which combine into:

$$A^\ell = f(Z^\ell) = f\left((W^\ell)^T A^{\ell-1} + W_0^\ell \right)$$

One more comment: a useful definition.

Definition 37

A **hidden layer** is any layer except for the **last** one.

It is called a "**hidden**" layer because, if you're viewing the whole neural network based on

- **Input** x (first input)
- **Output** A^L (final output)

Then you can't see the **output** of any of the layers except for the **last** one.

Sometimes you'll hear someone say that a hidden layer is any except the "**first** or **last**": by that, they mean you can view the **input** for the **first** layer, as well as the **output** for the **last** layer.

But, when we're talking about **activation** functions (which we often are when we mention hidden layers, see below), we only care about whether the **output** is hidden!

7.3 Choices of activation function

Our linear model is entirely **defined** by its input: the number of **weights** in a neuron is just the number of **inputs** m.

But our **activation** function is up to us to decide: what works best?

7.3.1 Trying out linear activation

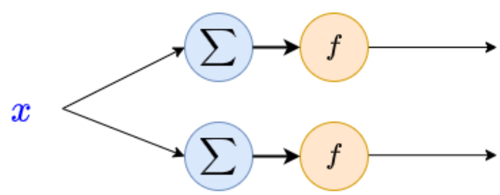
The simplest assumption would be to just use the **identity** function

$$f(z) = z \tag{7.31}$$

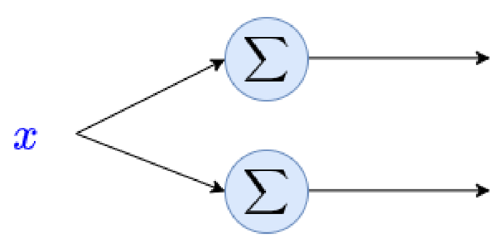
We might hope that we can combine a bunch of simple, **linear** models, and get a more sophisticated model. Why bother having a **nonlinear** activation at all?

Well, it turns out, combining **multiple** linear layers doesn't make our model any stronger. Let's try an example: we'll take a network with 2 layers, two neurons each.

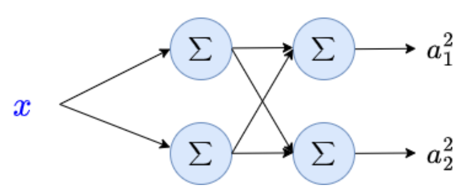
Let's look at layer 1:



Since the activation function has **no effect** on our result, we can **omit** it:

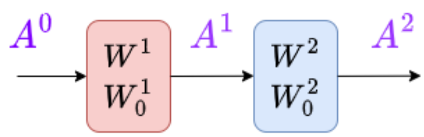


And now, we can show our **full** network:



7.3.2 Linear Layers: An example

We'll assume **two** inputs $A_0 = [x_1, x_2]^T$. For our sanity, we'll lump all of the weights in each **layer**:



We'll leave out W_0 terms to make it more readable, but the same will apply.

Layer 1:

$$A^1 = (W^1)^T A_0 \tag{7.32}$$

Layer 2:

$$A^2 = \overbrace{(W^2)^T (W^1)^T}^{\text{Weight matrices}} A_0 \tag{7.33}$$

The full function for this equation is two matrices, **multiplied** by our input vector.

Let's take an arbitrary example:

$$W^1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad W^2 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \tag{7.34}$$

Our equation becomes:

$$A^2 = \overbrace{\begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}}^{\text{Transposed matrices}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{7.35}$$

We created this function by applying two matrices separately. But, can't we **combine** them?

$$A^2 = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{7.36}$$

Wait, but this looks like a **one-layer** network with those weights! The second layer is **pointless**, we could have represented it with a single layer...

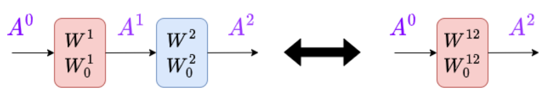
$$(W^{12})^T = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix} \tag{7.37}$$

7.3.3 The problem with linear networks

In fact, this is true in general: we can always take our **two** linear layers and combine them into **one**.

$$(W^2)^T(W^1)^T = W^{12} \tag{7.38}$$

Our network is **equivalent** to the supposedly "simpler" one-layer network.



What if we have more layers? Well, we can just combine them one-by-one. At the end, we're just left with one layer:

$$(W^L)^T(W^{L-1})^T \cdots (W^2)^T(W^1)^T = W^X \tag{7.39}$$

And so, we can't just use linear layers: we **need** a **nonlinear** activation function.

> **Concept 38**
> Having multiple consecutive **linear layers** (i.e. layers with linear **activation** functions) is **equivalent** to having one linear layer in its place.
>
> This means that we do not expand our **hypothesis** class by using more linear layers: we have to use **nonlinear** activation functions.

If we use something **nonlinear**...

$$A^2 = f\Big((W^2)^T A^1\Big) \tag{7.40}$$

We get something that doesn't **simplify**:

This is ugly, but we don't have to worry about the details.

$$A^2 = f\Bigg((W^2)^T \overbrace{\boxed{f\Big((W^1)^T x\Big)}}^{A^1}\Bigg) \tag{7.41}$$

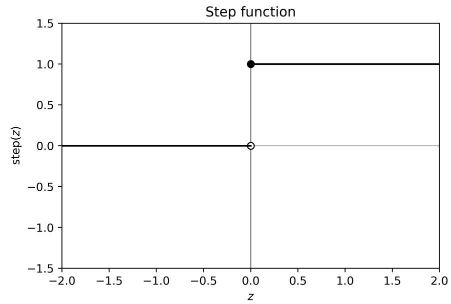
Which is what we want!

7.3.4 Example of Activation Functions

So, let's look at some possible **activation** functions:

- **Step** function $\text{step}(z)$:

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{7.42}$$

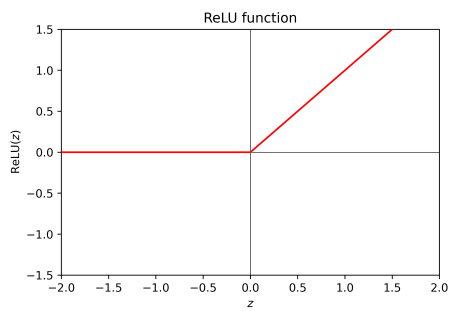


- This function is basically a **sign** function, but uses $\{0, 1\}$ instead of $\{-1, +1\}$.
- Step functions were a common early choice, but because they have a **zero** gradient, we can't use **gradient descent**, and so we basically **never** use them.

Same reason we replaced the sign function with sigmoid.

- **Rectified Linear Unit** $\text{ReLU}(z)$:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{7.43}$$

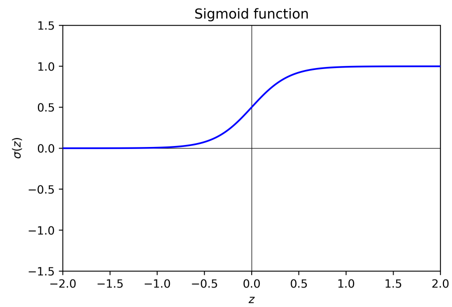


- This is a very **common** choice for activation function, even though the derivative is undefined at 0.
- We specifically use it for internal ("**hidden**") layers: layers that are neither the **first** nor **last** layer.

They're "hidden" because they aren't visible to the input or output.

- **Sigmoid** function $\sigma(z)$:

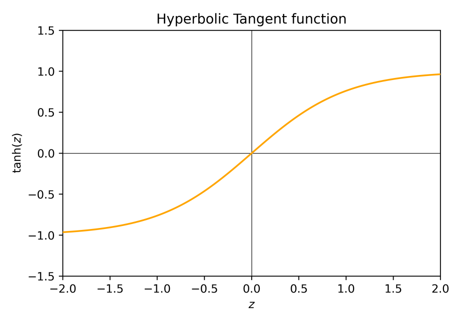
$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{7.44}$$



- This is the **activation** function for our **LLC** neuron from before.
- Just like it was then, it's useful for the **output neuron** in **binary classification**.
- Can be interpreted as the **probability** of a positive $(+1)$ binary classification.

• **Hyperbolic Tangent** $\tanh(z)$:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{7.45}$$



- This is function looks similar to sigmoid over a different **range**.
- Unfortunately, it will not get much use in this class.

• **Softmax** function $\text{softmax}(z)$:

$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_n)/\sum_i \exp(z_i) \end{bmatrix} \tag{7.46}$$

- Behaves a like a **multi-class** version of **sigmoid**.
- Appropriately, we use it as the **output neuron** for **multi-class** classification.
- Can be interpreted as the **probability** of our k possible classifications.

Concept 39

For the different **activation functions**:

- $\text{sign}(z)$ is **rarely** used.
- $\text{ReLU}(z)$ is often used for "**hidden**" layers.
- $\sigma(z)$ is often used as the **output** for **binary classification**.
- $\text{softmax}(z)$ is often used as the **output** for **multi-class classification**.

$\tanh(z)$ is useful, but not a focus of this class.

7.4 Loss functions and activation functions

As we can see above, your **activation** function depends on what kind of **problem** you're dealing with.

The same is true for our **loss** function: we used **different** loss functions for classification and regression.

Classification can be further broken up into **binary** versus **multiclass** classification.

To summarize our findings, we'll **sort** this information:

Concept 40

Each of our **tasks** requires a different **loss** and output **activation** function.

We emphasize that we specifically mean the output activation function: the activation function used in **hidden layers** doesn't have to match the loss function.

task	f^L		Loss	
Regression	Linear	z	Squared	$(g-y)^2$
Binary Class	Sigmoid	$\sigma(z)$	NLL	$y \log g + (1-y) \log(1-g)$
Multi-Class	Softmax	$\text{softmax}(z)$	NLLM	$\sum_j y_j \log(g_j)$

Terms

- Neuron (Unit, Node)
- Neural Network
- Series and Parallel
- Linear Component
- Weight w
- Offset (Bias, Threshold) w_0
- Activation Function f
- Pre-activation z
- Activation a
- Identity Function
- Acyclic Networks
- Feed-forward Networks
- Layer
- Fully Connected
- Input dimension m
- Output dimension n
- Weight Matrix
- Offset Matrix
- Layer Notation A^ℓ
- Step function
- ReLU function
- Sigmoid function
- Hyperbolic tangent function
- Softmax function