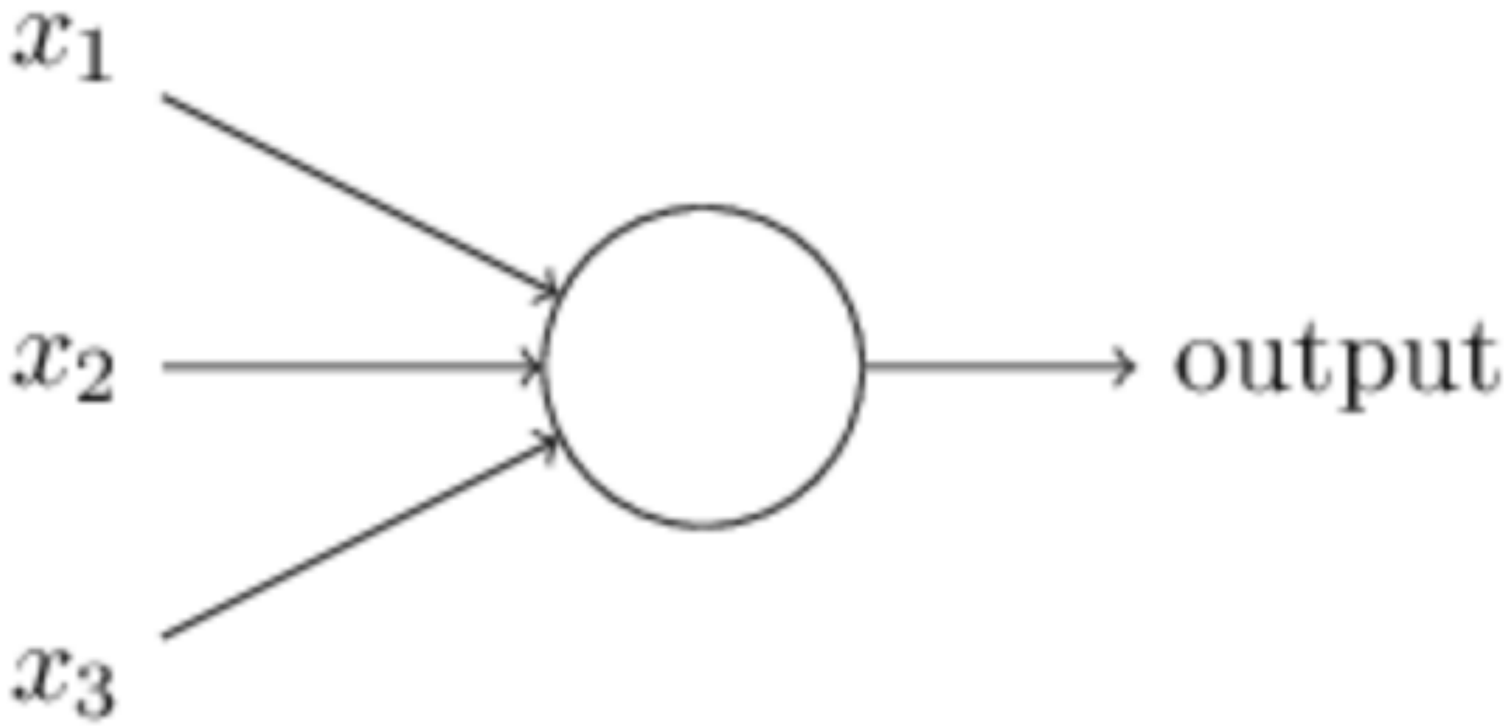
Why use neural networks?

- You've already seen the main limitation of linear approximation: need to hardcode the features.
- This may not even be possible if there are many of them.
- Also: in some sense, this is the crux of the problem.
  — you want an AI to learn to play a game
  — so first you've got figure out what is important to be successful in the game
  — that is, you've got to figure out how to play it first
- An AI will always be limited by the features a human being has hardcoded for them.
- Would be nice if somehow we could *learn* the features as we interact with the MDP.

- We are going to talk about a method that does this. The change we make to what we have done is pretty basic...we just have to move away a little bit from linear approximation.
- Next, we'll talk about neural networks.
  For a while, we'll just be talking about neural networks as if this was a generic ML class.
  Eventually, we'll bring everything back to RL.
- I'll follow Michael Nielsen's notes on the subject, which I think are pretty well written:
  http://neuralnetworksanddeeplearning.com
- I encourage you to read those notes yourselves in parallel with the lectures.



The basic building block of a neural network: the perceptron

Inputs are real numbers x_1, x_2, ... , x_n

The output is also a real number

output = sigma(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b)

w_1, ..., w_n and b are weights. The quantity b is sometimes called the bias.

sigma: R — > R is a function called an "activation function"

A typical choice is sigma(x) = max(x,0) called a ReLu

Other choices for sigma are also possible

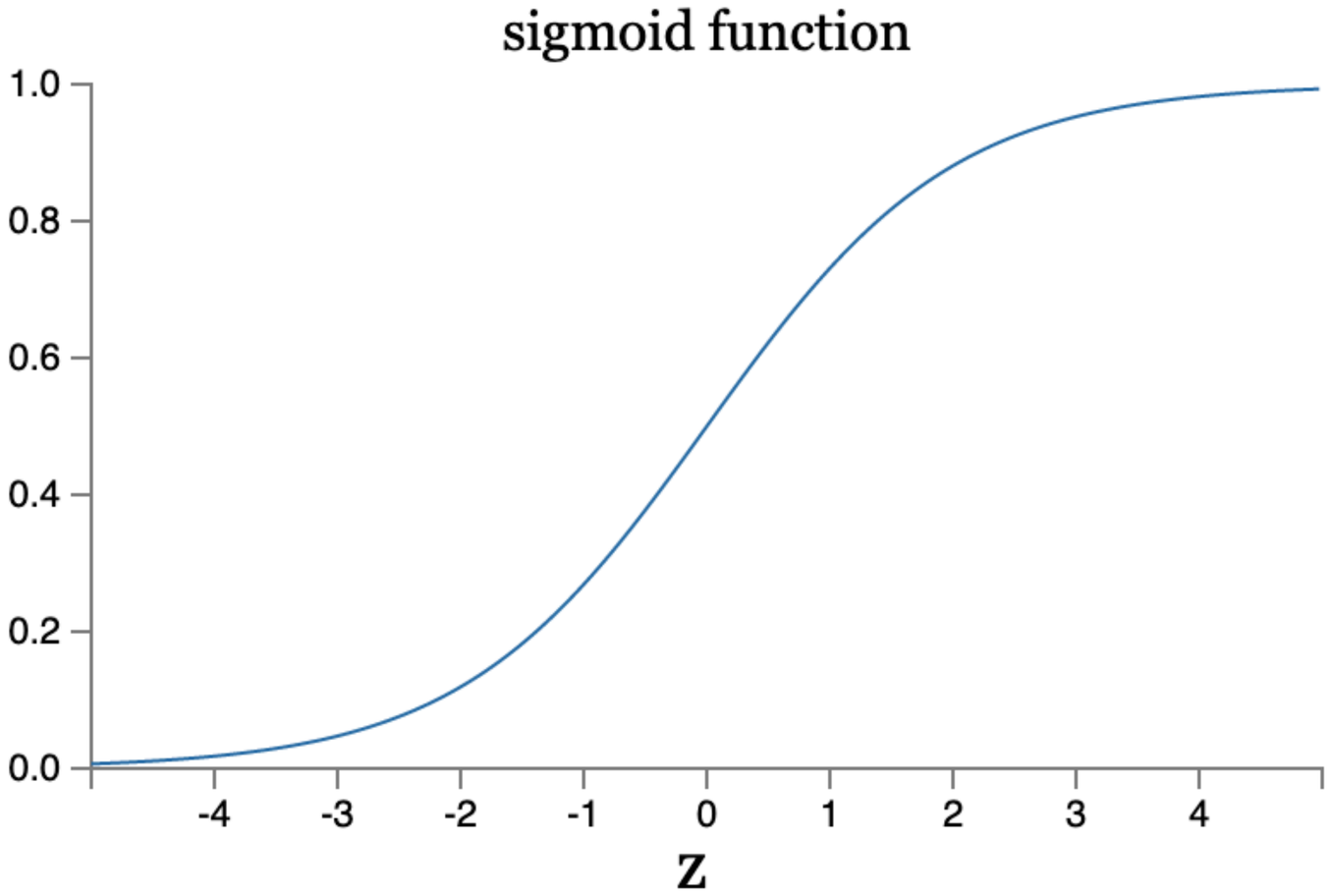
- Let's take a couple of examples of what a perceptron can do.
- We could have $y = \sigma_{\text{ReLu}}(x_1 - x_2 - 1)$, corresponding to weights $[1, -1]$ and bias -1.

  If $x_1 \leq x_2 + 1$, this equals zero.

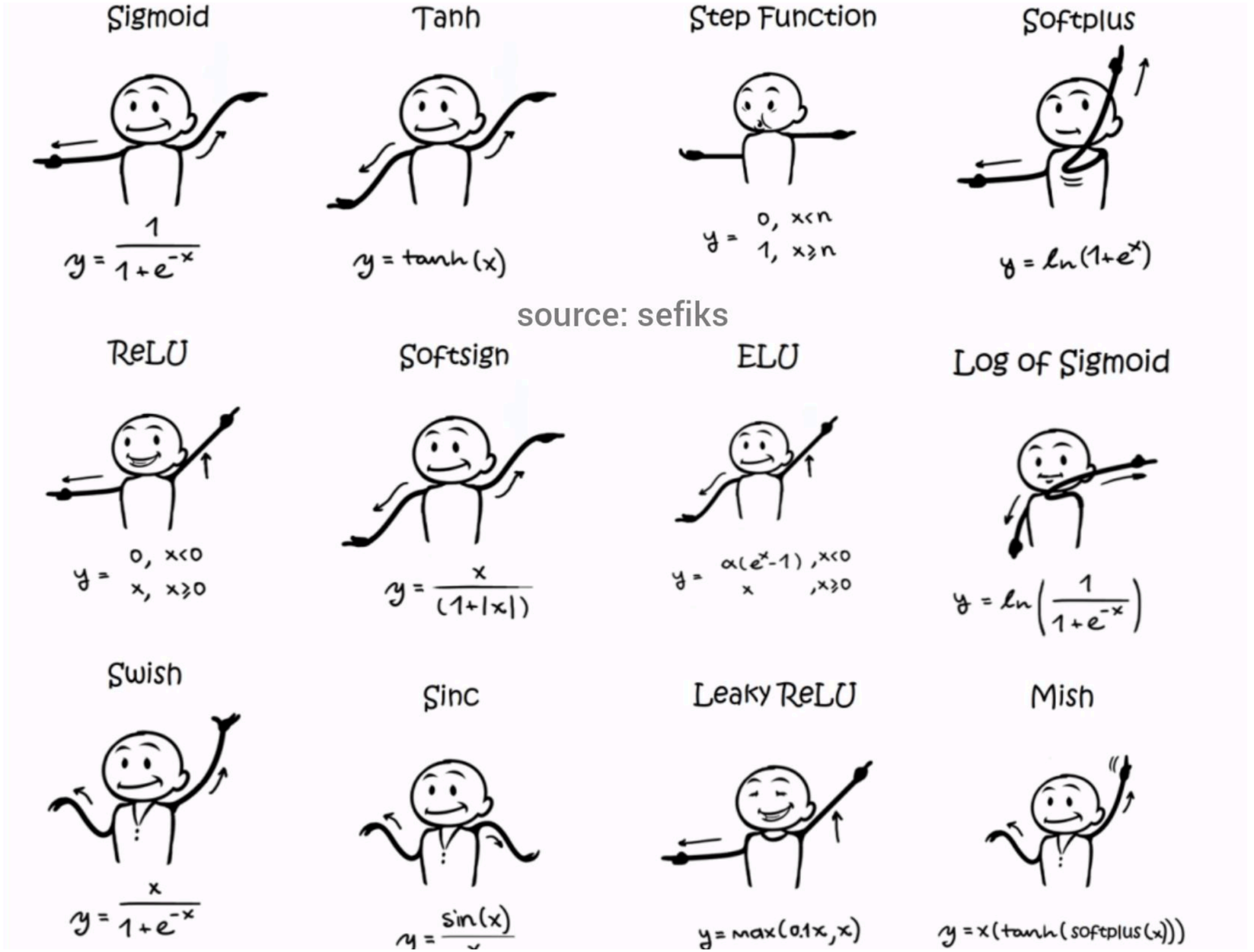
  If $x_1 > x_2 + 1$, this equals $x_1 - x_2 - 1$
- Let's define $\sigma_{\text{threshold}}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$
- Suppose $y = \sigma_{\text{threshold}}(x_1 + x_2)$.

  This equals 1 if $x_1 \geq -x_2$

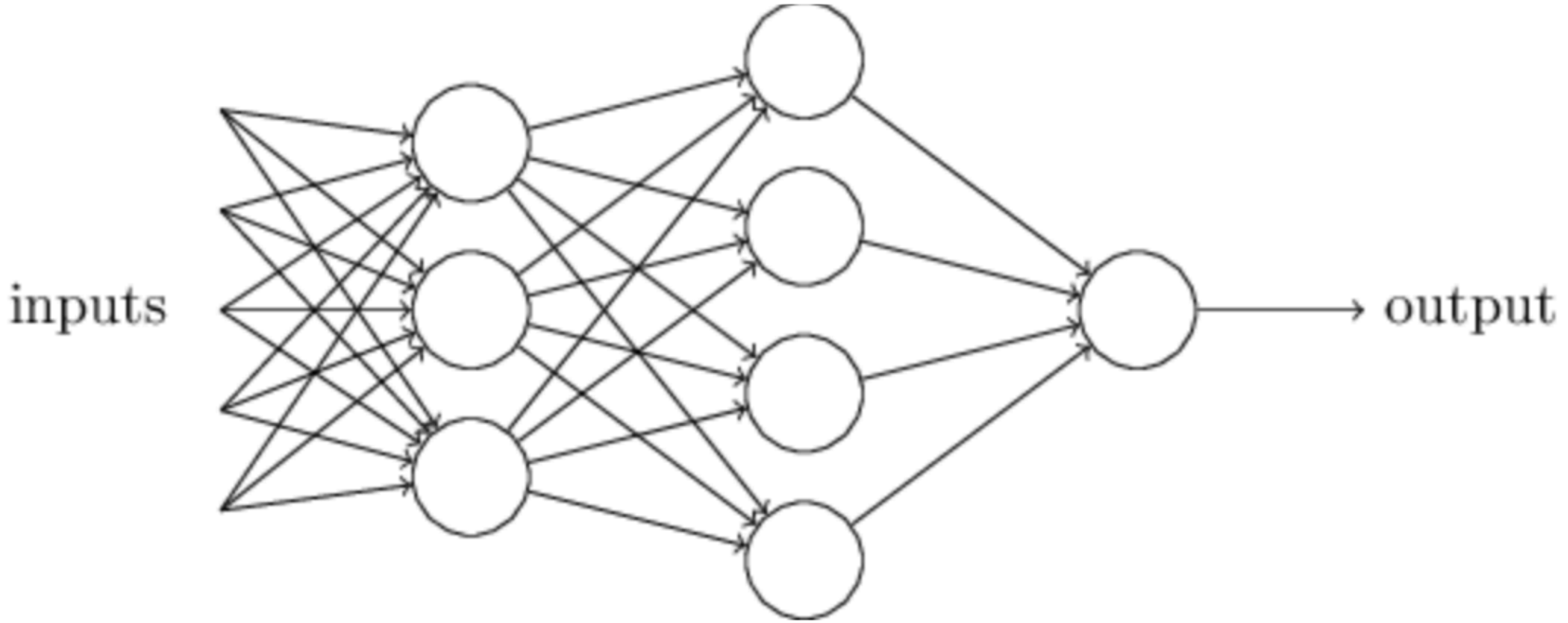
  It equals 0 if $x_1 < -x_2$
- The problem with this choice of activation function is that it is not continuous.
- We can mimic it with a continuous function as described on the next slide.



$$\sigma_{sigmoid}(x) = 1/(1+e^{-x})$$



Other activation functions



Each circle is a perceptron

This is what a neural network is: a collection of perceptrons such that the outputs of some perceptron fit into the others

This is called a feedforward architecture

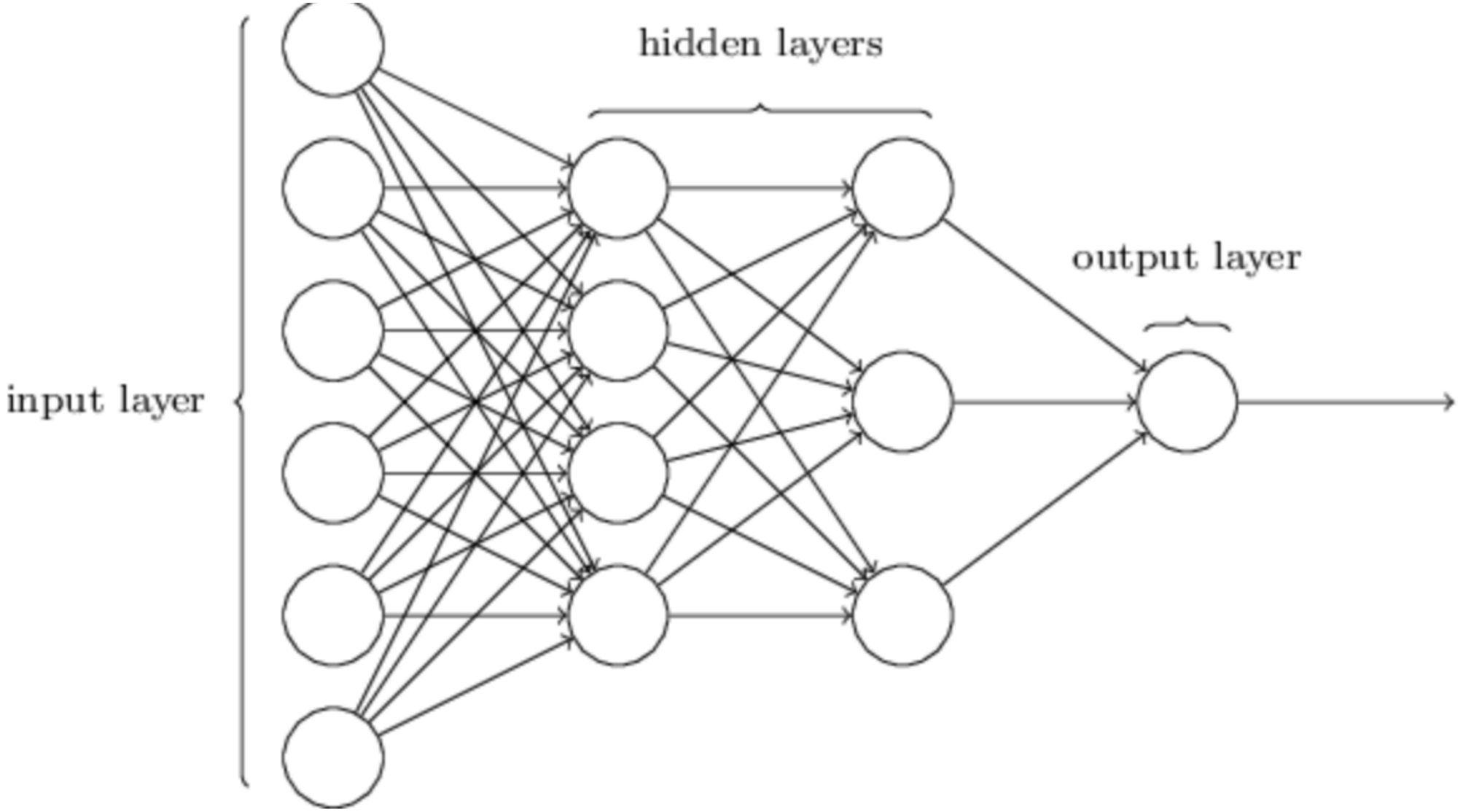
The number of layers is called the depth of the network

The number of perceptrons in each layer is called the width of that layer

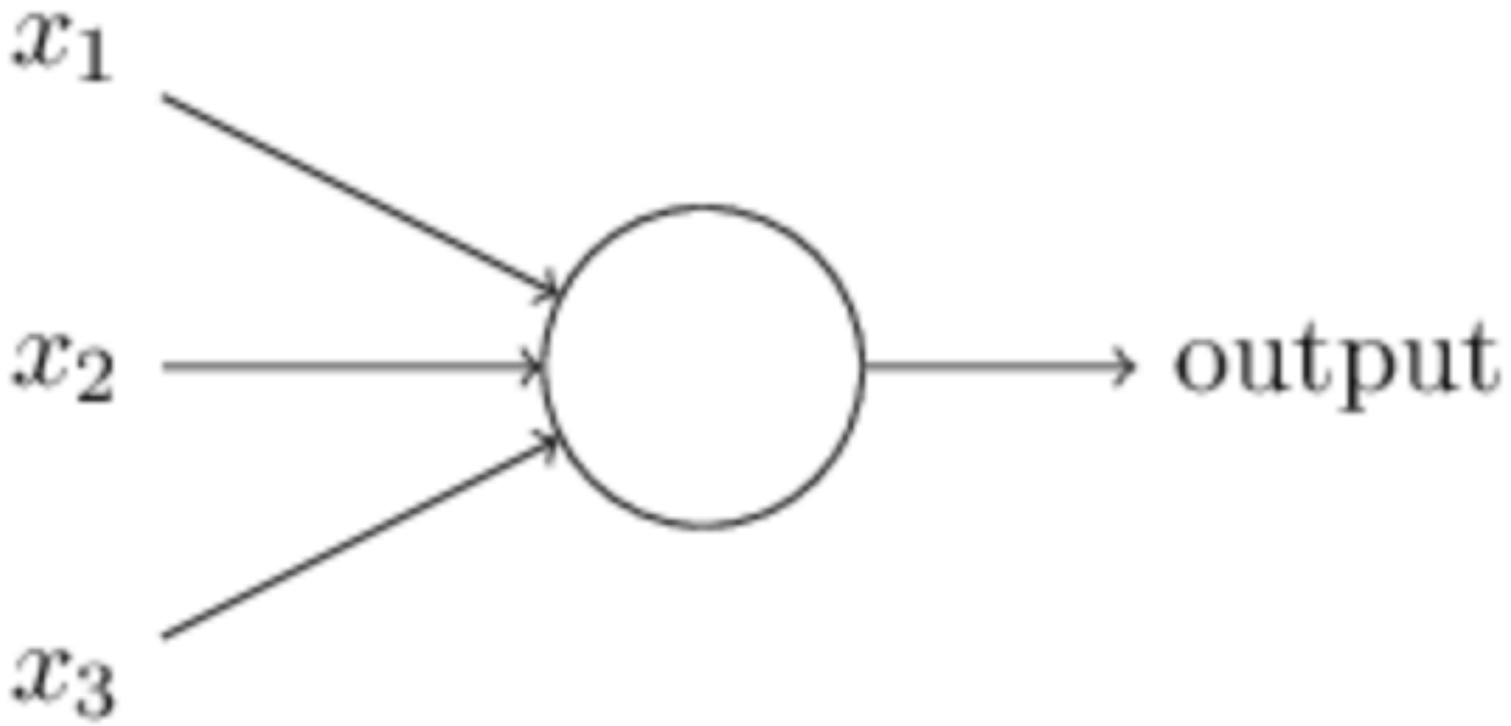
If all the width are the same, we can talk about the width of the network

The depth + widths + activation functions are usually referred to as the architecture of the network

(sometimes activation functions for different neurons are taken to be different)



You usually put the input as a layer as well (convention).
This network takes six numbers and maps them to one number
The perceptrons are also sometimes called neurons



If we fix the weights, we can think of a perceptron (or a neural network) as mapping inputs to outputs
Suppose the weights are [1,1,1,1] and the activation function is ReLU.
Then f_{[1,1,1,3]}(1,1,1) =6
On the other hand, f_{[1,1,1,0]}(-10,0,0) = 0

Similarly,
f_{[1,2,3,8]}(—3,-2,-1) = max(-2,0) = 0

- The problem we typically encounter is: given a neural network architecture and a collection of data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$

  find the weights so that the resulting NN maps each x_i to y_i.
- Often x_i is a vector while y_i is a scalar. For example, x_i could be the pixel-level representation of an image. The quantity y_i could be 1 or 0 depending on whether the image contains a cat or a dog.
- Regardless of whether y_i is a scalar, it is called the "label."
- Once we've fixed the architecture, let $f_w(x)$ be the output of the neural network with weights w on an input x

  [yes this is confusing because we should perhaps write $f_{w,b}(x)$....but we'll just write f_w]
- If its not possible to find weights w such that $f_w(x_i) = y_i$ for all i, then we want weights such that $f_w(x_i) \approx y_i$ as well as possible. For example, we want to solve $\min_w \sum_{i=1}^{m} \frac{1}{2m}(f_w(x_i) - y_i)^2$.
- How do we solve this?

- OK, so the "data set" is the collection $(x_1, y_1), \ldots, (x_n, y_n)$ and that is fixed.

  The architecture is fixed (we discuss later how to choose).

  But the weights need to fit to the data.

- So we want to solve $\min_w \sum_{i=1}^{m} \frac{1}{2m}(f_w(x_i) - y_i)^2$.

- OK, we know how to do this: we use gradient descent.

  $$w_{t+1} = w_t - \alpha_t \frac{1}{m} \sum_{i=1}^{m} (f_{w_t}(x_i) - y_i) \frac{\partial f_w(x_i)}{\partial w}(w_t)$$

- There are two issues to address here:

  — a gradient requires a single pass through the entire data set

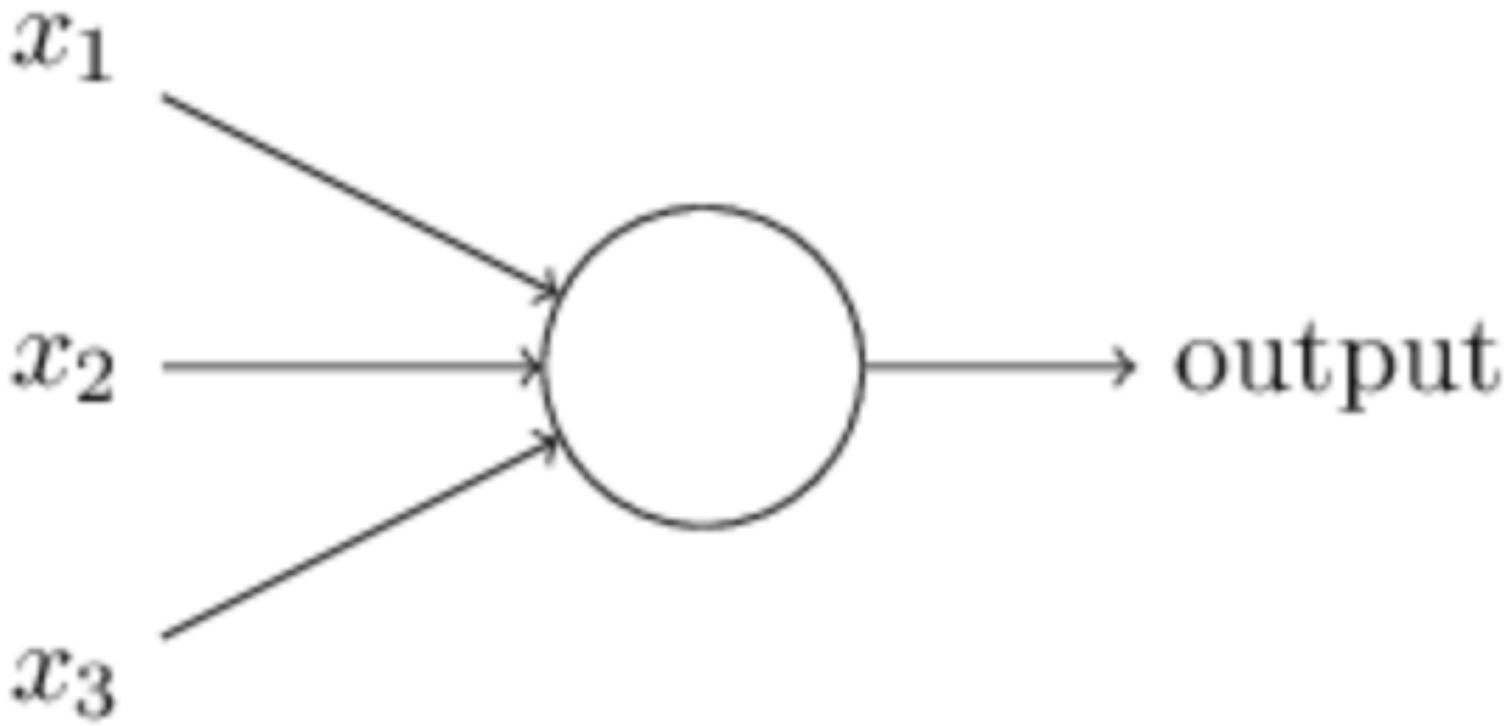
  — this can be really unrealistic if the data set is large (more on this later)

  — but at an even more basic level, what is $\frac{\partial f_w(x_i)}{\partial w}(w_t)$?

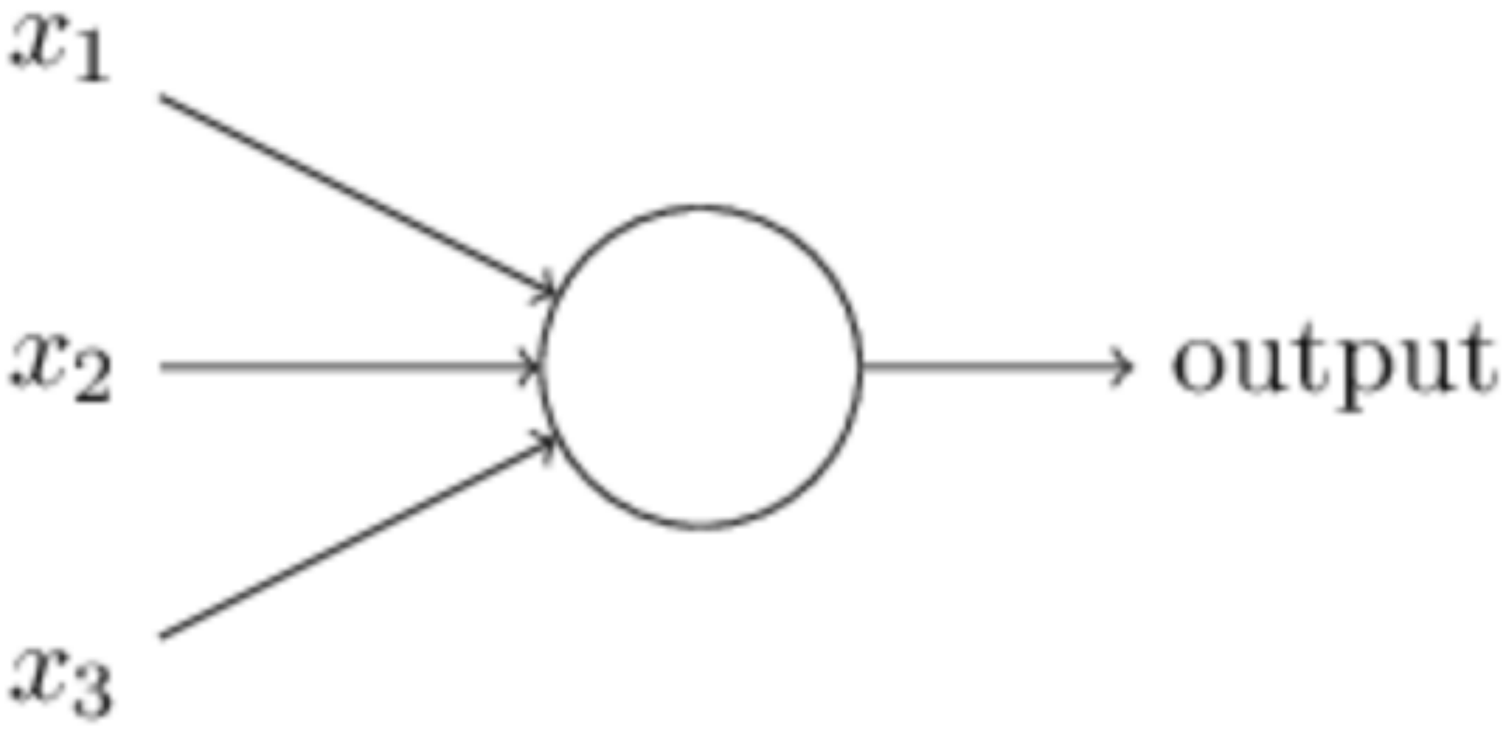
  We need to differentiate the NN with respect to the weights.

- Let's address the second problem first. At the risk of belaboring things, let's restate it.
- $f_w(x)$ is the output of a neural network with weights w when the input is x.
  [Presumably, the architecture is fixed here]
- We usually think of the weights as fixed and we think of this as mapping input to output.
- But now we need to compute $\frac{\partial f_w(x_i)}{\partial w}(w_t)$.
- So we need to think of the input as fixed at x_i, and then we think of the output as a function of the weights.

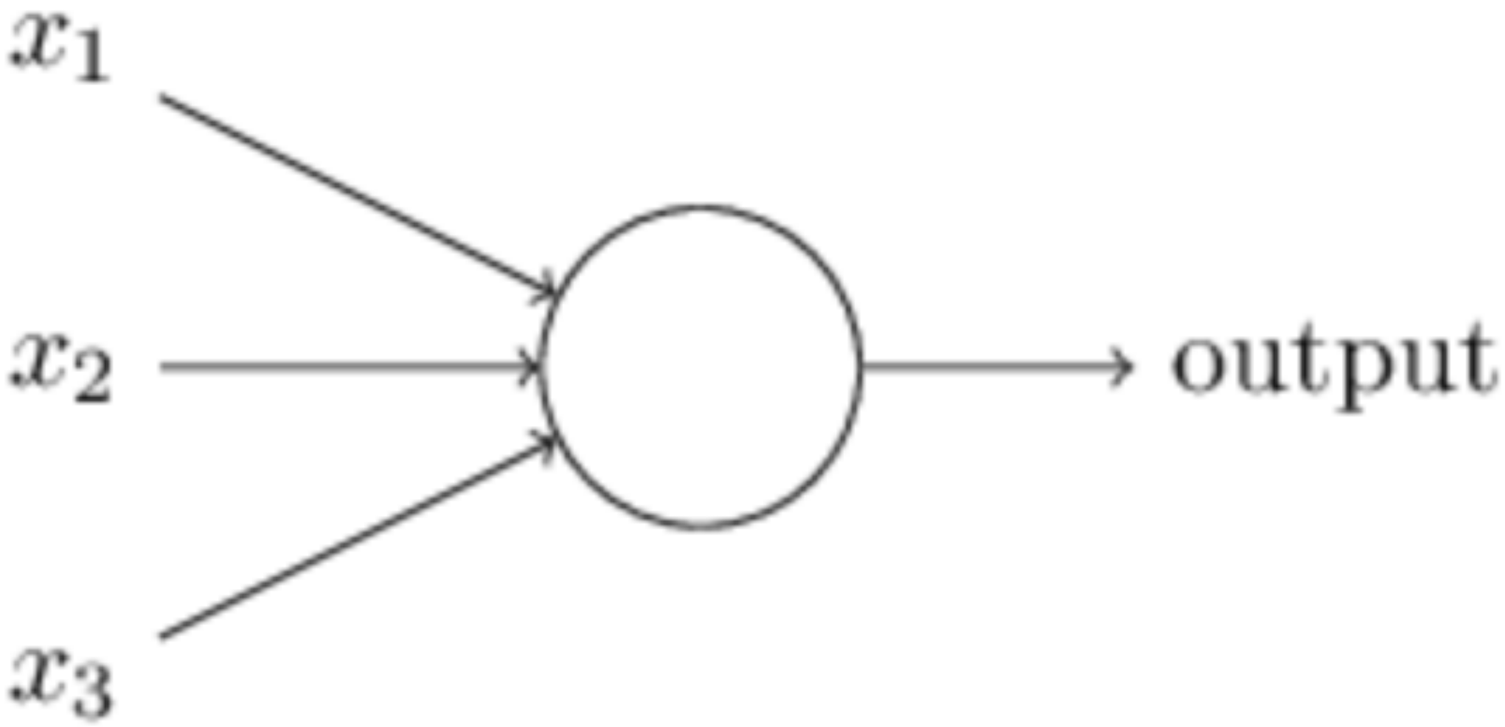
  We then need to find the gradient of this function, and evaluate that gradient at w_t.
- Make sure you understand this. Of course, it is natural — it goes without saying that $f_w(x)$ is, once we fix w, a function of x; but we can also think of it as a function of w if x is fixed.



Suppose x_1 = [5,6,7].
The activation function is sigma(u) = u
so if we fix the input to x_1 and view the output as a function of the weights then
Output = w_1*5 + w_2*6 + w3*7 + b
Its gradient is [5,6,7,1]^T.
Evaluated at any point, it is [5,6,7,1]^T



Suppose x_1 = [5,6,7].
The activation function is sigma(u) = (1/2)*u^2
[note: in practice we never use this activation function]
so if we fix the input to x_1 and view the output as a function of the weights then
Output = (1/2)*(w_1*5 + w_2*6 + w3*7 + b)^2
The gradient is (5w_1 + 6w_2 + 7w_3 + b)*[5,6,7,1]^T
Evaluated at w=[0,0,0,2] it equals [10,12,14,2]^T



Suppose x_1 = [5,6,7]. The activation function is ReLU: sigma(u) = max(u,0)
so if we fix the input to x_1 and view the output as a function of the weights then
Output = max(w_1*5 + w_2*6 + w3*7 + b, 0)
Here it helps to break the space of weights into two regions:
region 1: w_1*5 + w_2*6 + w3*7 + b => 0, region 2: everything else
In region 1, the gradient is [5,6,7,1]^T. In region 2, the gradient is 0
So the gradient evaluated at w=[1,1,1,1] is [5,6,7,1]^T
The gradient evaluated at [-1,-1,-1,-1] is zero
Punchline: I hope you are comfortable with the idea of fixing the input to x_i, viewing the output as a function of w, taking the gradient with respect to w, and evaluating that at w_t

- OK, so we need to compute $\frac{\partial f_w(x_i)}{\partial w}(w_t)$. This will require new notation.

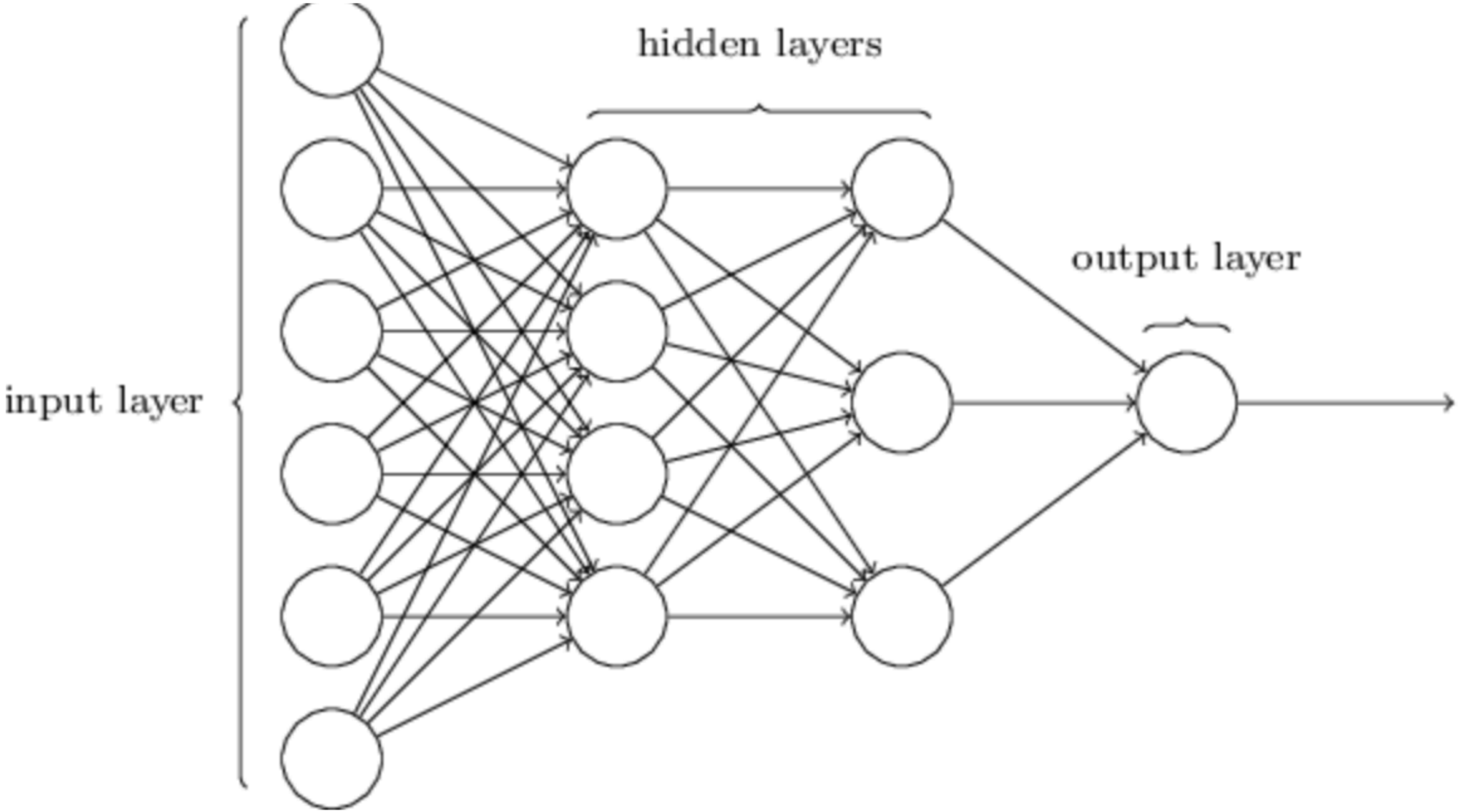
  A **lot** of notation.

  We need to think about how feedforward neural networks propagate information.
- Let us introduce the notation

  a_j^l

  for the output of the j'th neuron in the l'th level.

  So the neuron number is a subscript. The level of the neuron is a superscript.



The output of the network is a_1^4
The inputs are a_1^1, a_2^1, a_3^1, a_4^1, a_5^1, a_6^1
What is a_2^3 in this picture?

- Let b_j^l be the bias of the j'th neuron at the l'th level.
- Next, let w_{jk}^l be the weight the j'th neuron at the l'th level puts on the k'th neurons from the $l-1$ layer.
- So:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

- This is the key equation.
- As always, we like to stack these quantities up into vectors:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

Let's parse this:

— a^l is a vector with as many entries as the number of neurons on the l'th layer

— a^{l-1} is a vector with as many entries as the number of neurons in the $l-1$st layer.

— b^l is a vector with as many entries as the number of neurons on the l'th layer.

— w^l is a matrix.

— The activation function $\sigma(\cdot)$ when applied to a vector is applied elementwise.

- Useful to have the shorthand
  $z^l = w^l a^{l-1} + b^l$
- Then $a^l = \sigma(z^l)$
- If there are L layers, then $a^L(x)$ is the output of the network when x is fed into the input layer.
- Our objective is then
  $$C = \frac{1}{2m} \sum_{i=1}^{m} ||y_i - a^L(x_i)||_2^2$$
- Our update is
  $$w_{jk}^l(t+1) = w_{jk}^l(t) - \alpha_t \frac{\partial C}{\partial w_{jk}^l}(w(t))$$
- OK, but let's define $C^i = \frac{1}{2}(y^i - a^L(x_i))^2$. Then $\frac{\partial C}{\partial w_{jk}^l} = \sum_{i=1}^{m} \frac{\partial C^i}{w_{jk}^l}$.
- Thus we have
  $$w_{jk}^l(t+1) = w_{jk}^l(t) - \alpha_t \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C^i}{\partial w_{jk}^l}(w(t))$$
- So we need to talk about how to compute the quantity $\frac{\partial C^i}{\partial w_{jk}^l}(w(t))$.

  Interpretation: the derivative of $C^i = \frac{1}{2}(y^i - a^L(x_i)^2$, the squared error on the i'th datapoint, with respect to weight w_{jk}^l, evaluated at the collection of weights $w(t)$.

- Where we are:

  — given a data set and an architecture, we want to find a NN to give the right labels, as much as this is possible

  — we are going to do this by doing gradient descent

  — for this, we need to be able to compute the gradient

  — this reduces to being able to compute $\frac{\partial C^i}{\partial w^l_{jk}}(w(t))$

  where $C^i = \frac{1}{2}(y^i - a^L(x_i))^2$

- Interpretation: we are given a single data point x_i. The label should be y_i. The squared difference (divided by two) is C^i. We view C^i a a function of the weights.
- We will compute this using the chain rule.

- Let's start with something easy.

  We have $C^i = \frac{1}{2}(y^i - a^L)^2$

  [we stop writing $a^L(x_i)$ and write a^L. Recall that the data point x_i] is fixed.

  What is $\frac{\partial C^i}{\partial a^L}$?

- Of course, $\frac{\partial C^i}{\partial a^L} = a^L - y_i$.
- What about $\frac{\partial C^i}{\partial z^L}$
- Observe C^i is a function of a^L and $a^L = \sigma(z^L)$. So we can view C^i as a function of z^L.
- Easy: $\frac{\partial C^i}{\partial z^L} = (a^L - y_i)\sigma'(z^L)$
- These computations are easy to make, but there is an important trick here: we view C^i as a function of the last layer.

  Specifically, we have fixed x_i, we see that w determines z^L which determines a^L which determines C^i.
- Next step: view C^i as a function of the outputs of the intermediate levels.

- Let us now view C^i as a function of:

  — the outputs of the layer before last, a^{L-1}

  — the weights the last layer puts on the layer before it, namely w^L.

- Let's compute $\frac{\partial C^i}{\partial a_j^{L-1}}$.

- So a_j^{L-1} affects all the neurons on the last layer, and then those affect C^i.

- So $\frac{\partial C^i}{\partial a_j^{L-1}} = \sum_k \frac{\partial C^i}{\partial z_k^L} \frac{\partial z_k^L}{\partial a_j^{L-1}}$

- The first term on the RHS we already computed in the previous slides.

  As for the second term, we use

  $z_k^L = \sum_l w_{kl}^L a_l^{L-1} + b_k^L$

  so

  $\frac{\partial z_k^L}{\partial a_j^{L-1}} = w_{kj}^L$

- Finally, $\frac{\partial C^i}{\partial a_j^{L-1}} = \sum_k \frac{\partial C^i}{\partial z_k^L} w_{kj}^L$

- OK what if we view C^i as a function of z^{L-1} and w^L.
- Let's compute $\frac{\partial C^i}{\partial z_j^{L-1}}$
- Easy:

$$\frac{\partial C^i}{\partial z_j^{L-1}} = \frac{\partial C^i}{\partial a_j^{L-1}} \frac{\partial a_j^{L-1}}{\partial z_j^{L-1}}$$

$$= \frac{\partial C^i}{\partial a_j^{L-1}} \sigma'(z_j^{L-1})$$

$$= \sum_k \frac{\partial C^i}{\partial z_k^L} w_{kj}^L \sigma'(z_j^{L-1})$$

- Punchline: previously, we computed the derivatives with respect to the outputs of the last year we have been able to use this to compute the derivatives with respect to the outputs of the layer before that.

- Next: let's repeat this argument and compute the derivatives with respect to the outcomes of any layer!
  This technique is called *back propagation*.
- Natural question to have: why are we doing this? What we really want are derivatives with respect to the *weights*.
- Answer 1: patience...
- Answer 2: in general, often you should do what you *can* do, and see where it leads. Grothendiek's nutshell analogy.
- So let's define $\delta_j^l = \dfrac{\partial C^i}{\partial z_j^l}$ [evaluated at $w(t)$ but now, like all writing on the topic, we don't write that].
- We have computed δ_j^L. We used that to compute δ_j^{L-1}.
- Now let's write a general formula for δ_j^l in terms of all the δ_k^{l+1}.

- First step: chain rule.

  We can view C^i as a composition of two functions:

  — the first function maps the outputs of the l'th layer onto the outputs of the $l+1$'st layer

  — the second function maps the output of the $l+1'st$ layer to C^i
- This allows us to write:

$$\delta_j^l = \frac{\partial C^i}{\partial z_j^l} = \sum_k \frac{\partial C^i}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

$$= \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

- OK, but

$$z_k^{l+1} = \sum_u w_{ku}^{l+1} a_u^l + b_k^{l+1}$$

$$= \sum_u w_{ku}^{l+1} \sigma(z_u^l) + b_k^{l+1}$$

- So $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$.
- Putting it together: $\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l)$

- Summary: we have begun by computing the derivative of C^i with respect to z^L.
- The equation $\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l)$ allows us to "work backwards" and compute all the $\delta_j^l = \frac{\partial C^i}{\partial z_j^l}$.
- OK, but what we really want are the derivatives with respect to the weights. What should we do?
- Observation: the weight w_{jk}^l affects C^i only through z_j^l.
- So let's think of C^i as a composition of functions.

  The first function maps w^l to z^l, the second maps z^l to C^i

  So we have that $\frac{\partial C^i}{\partial w_{jk}^l} = \frac{\partial C^i}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$
- Likewise, b_j^l only affects C^i through z_j^l. So $\frac{\partial C^i}{\partial b_j^l} = \frac{\partial C^i}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_k^l$

- To summarize: $\delta_j^l = \dfrac{\partial C^i}{\partial z_j^l}$

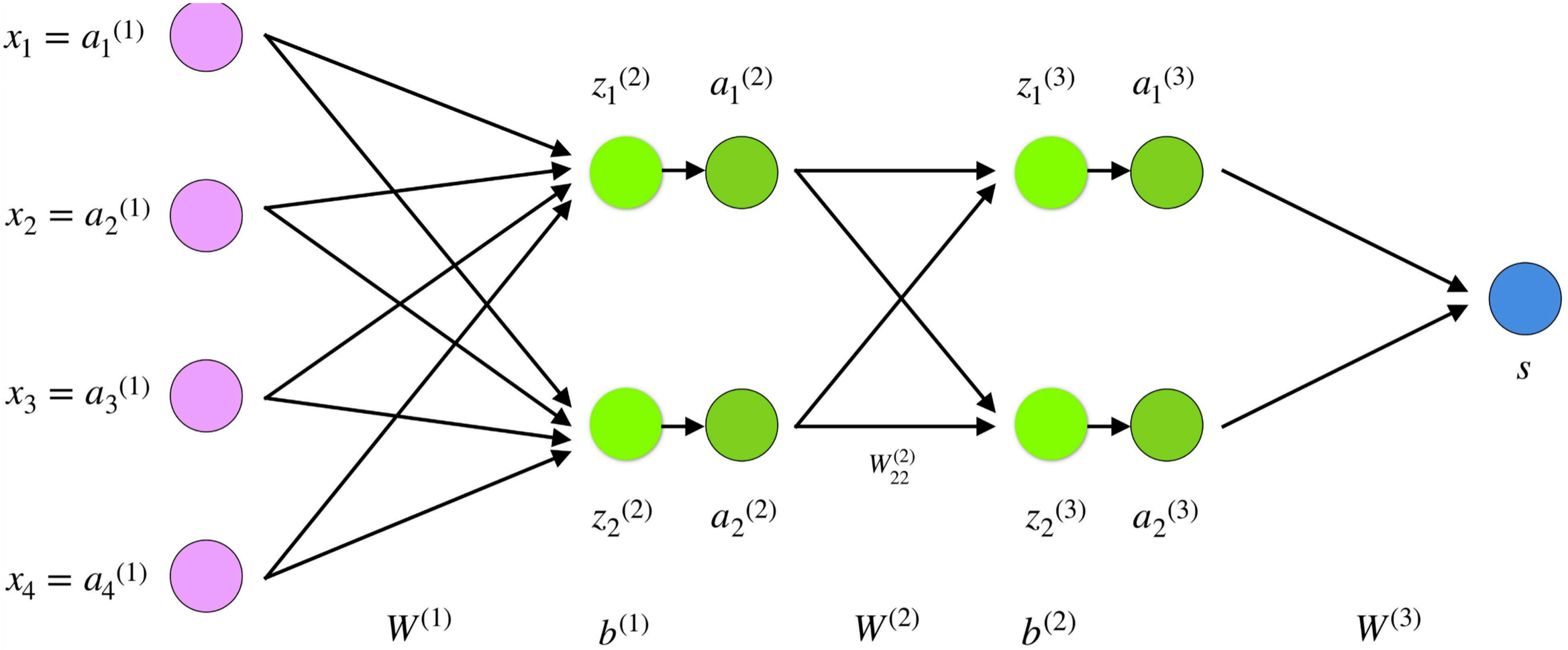
  (1) $\dfrac{\partial C^i}{\partial z^L} = (a^L - y_i)\sigma'(z^L)$ tells us what δ_1^L is.

  (2) $\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l)$ tells us how to compute the vector δ^l from

  (3) $\dfrac{\partial C^i}{\partial w_{jk}^l} = = \delta_j^l a_k^{l-1}$

  (4) $\dfrac{\partial C^i}{\partial b_j^l} = \delta_k^l$ (!!*)

- These are the equations for backpropagation.
- Note: we need to know what all the z_j^l and a^L is. So first you have to evaluate the NN. This is called the forward pass.
- Notation note: when we discuss all this, we sometimes stack up the w_{kj}^l into the matrix W^l



Suppose all b's are zero and

W^(1) = [10, 1, 1, 1

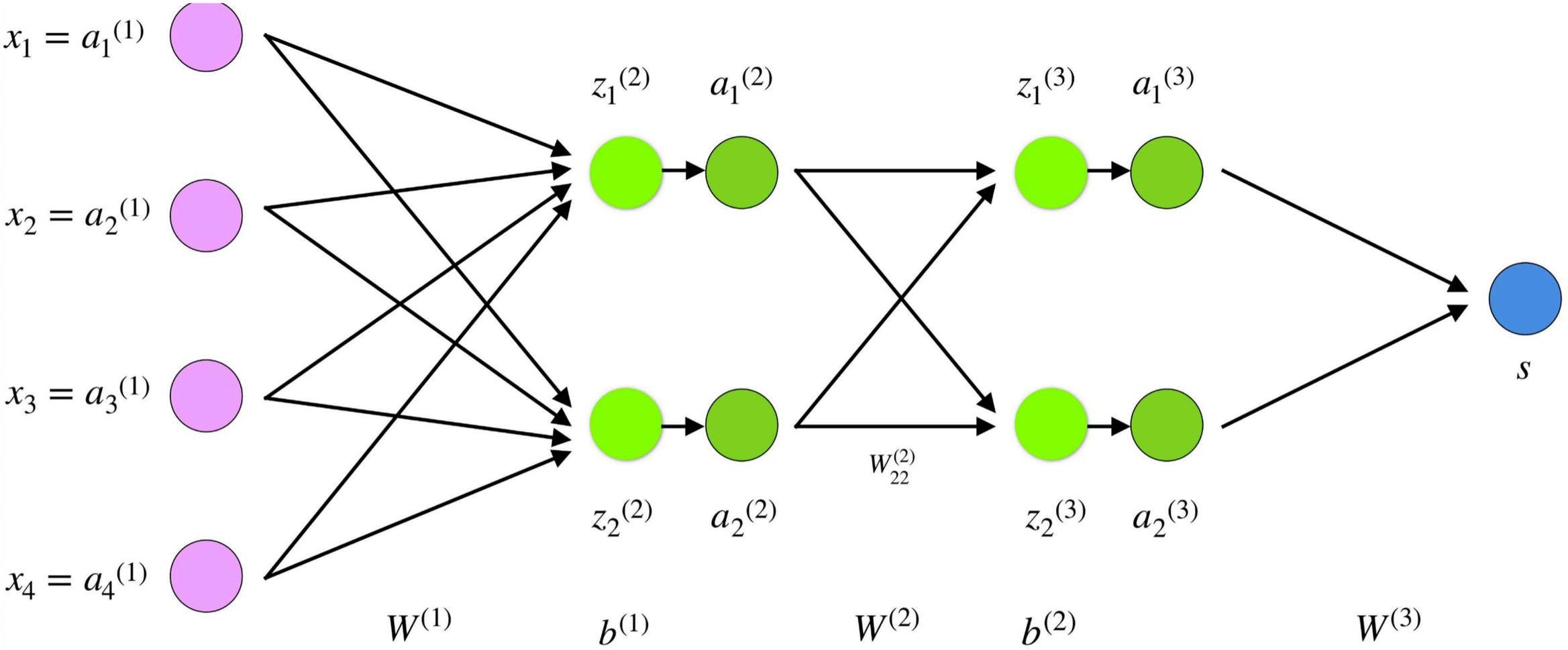
1, 1, 1, 1]

W^(2) = [1, 1

1, 1]

W^(3) = [1,1]

All activation functions are relu



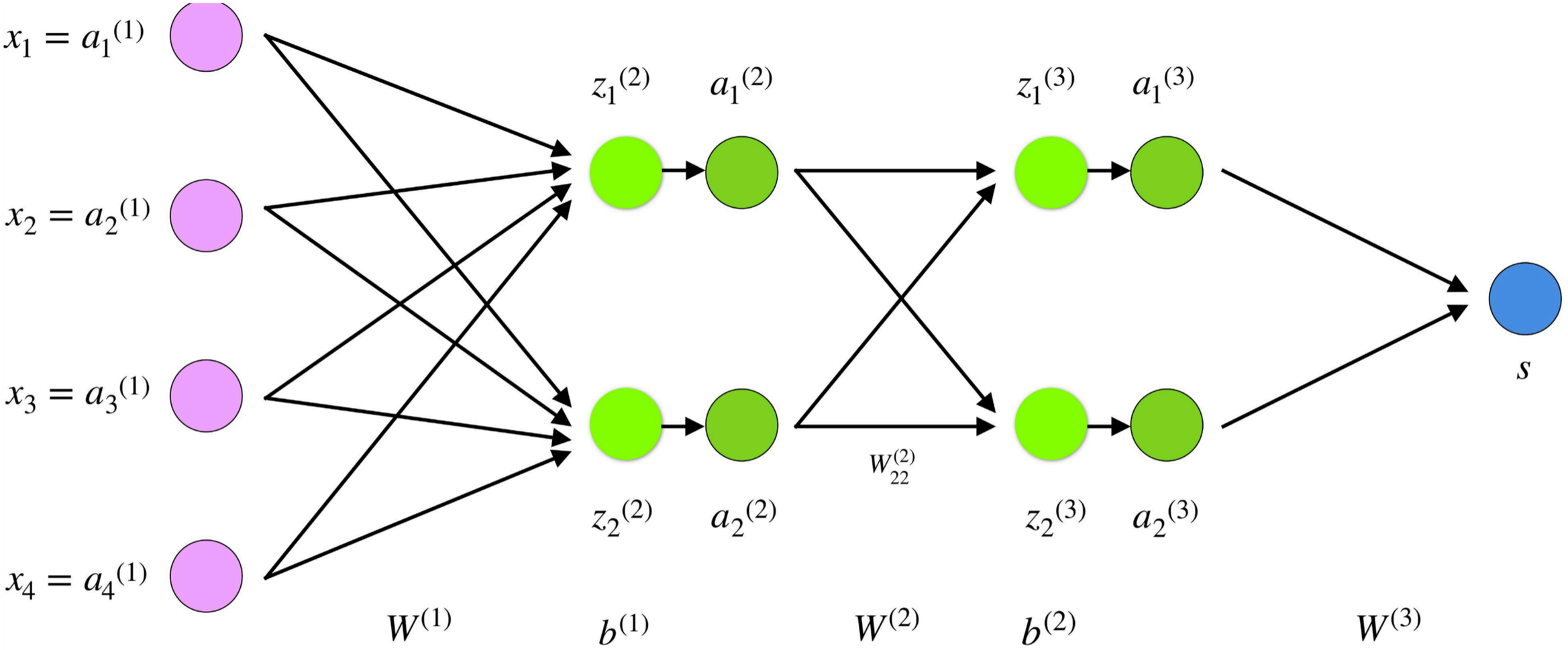
Suppose

x_1 = [-1,1,1,1]^T

y_1 = 1

Lets compute the derivative with respect to a couple of different weights at the first layer

First step: evaluate the NN



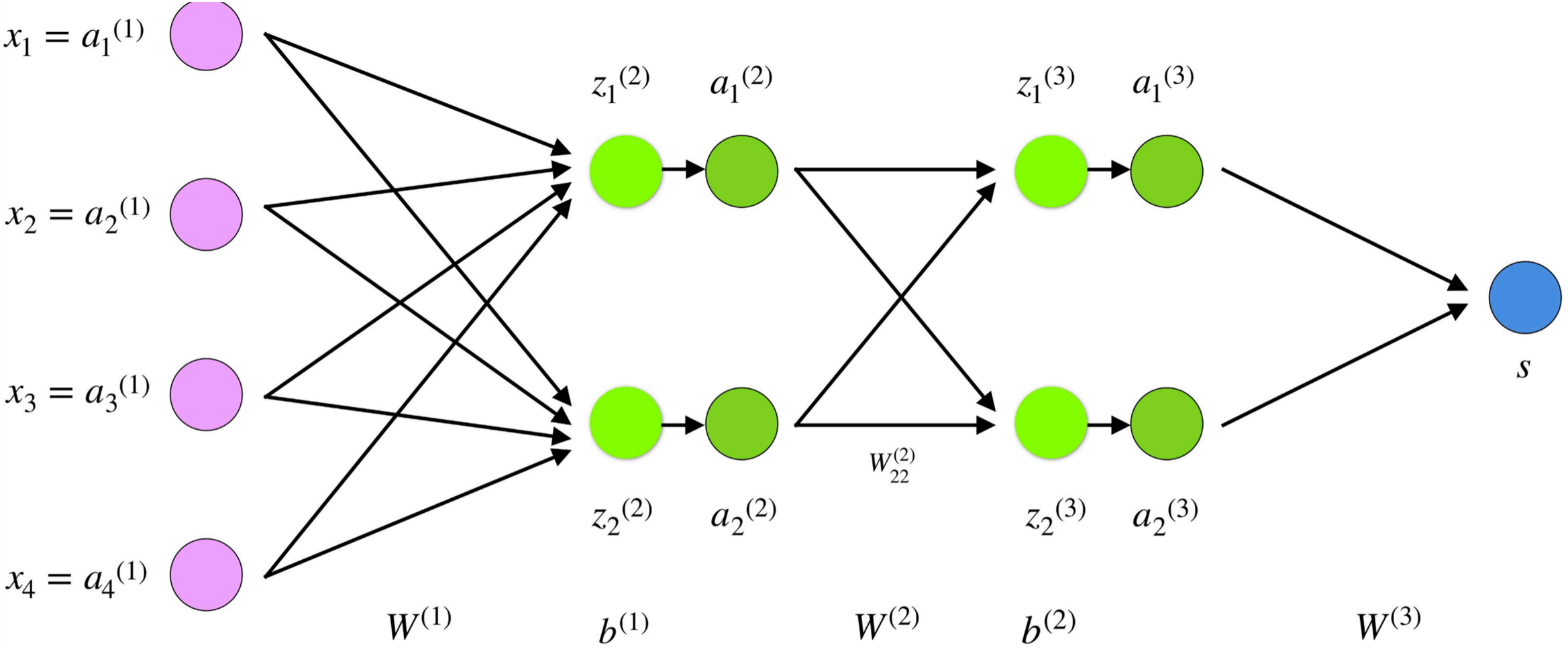
x_1 = [-1,1,1,1]^T

W^(1) = [10, 1, 1, 1

1, 1, 1, 1]

so z^(2) = [-7, 2]^T

a^(2) = [0,2]^T



a^(2) = [0,2]^T

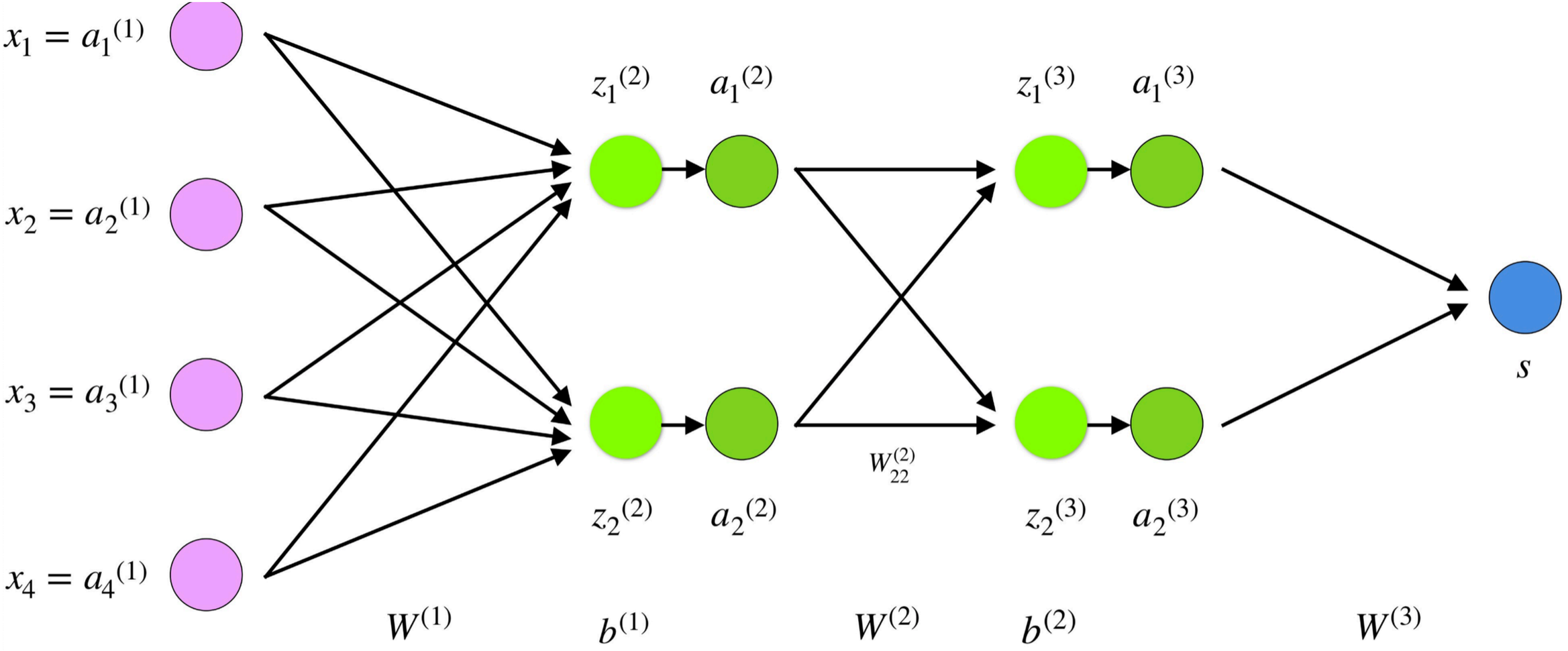
W^(2) = [1, 1

1, 1]

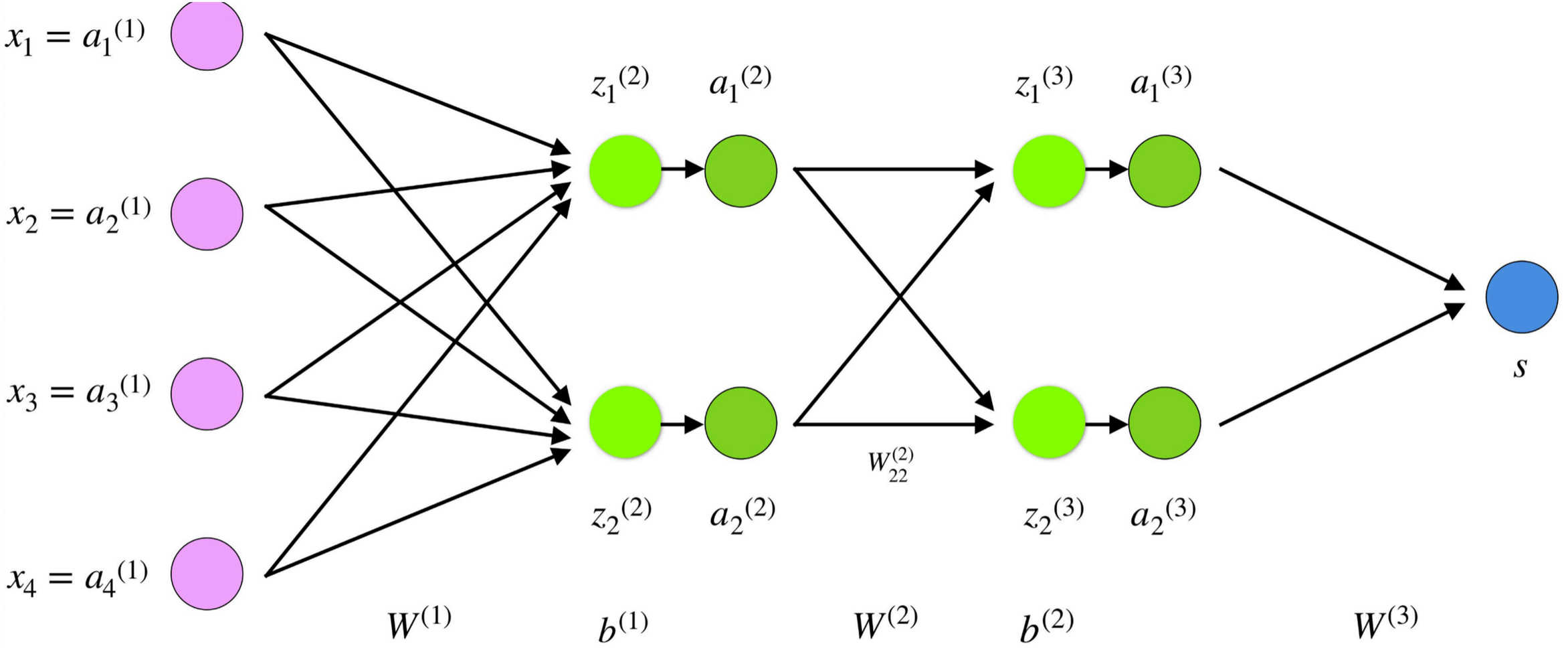
z^(3) = [2 2]

a^(3) = [2 2]

Since W^(3) = [1,1], we have the output = 4



Ok so the derivative with respect to a^L is a^L - y which is 4-1 = 3
The derivative with respect to z^L is the same here
Now let's look at the equation \delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l)
delta_1^3 = delta_1^4 * w_{41}^4 = 3
delta_2^3 = delta_1^4 * w_{42} = 3



Ok so the derivative with respect to a^L is a^L - y which is 4-1 = 3

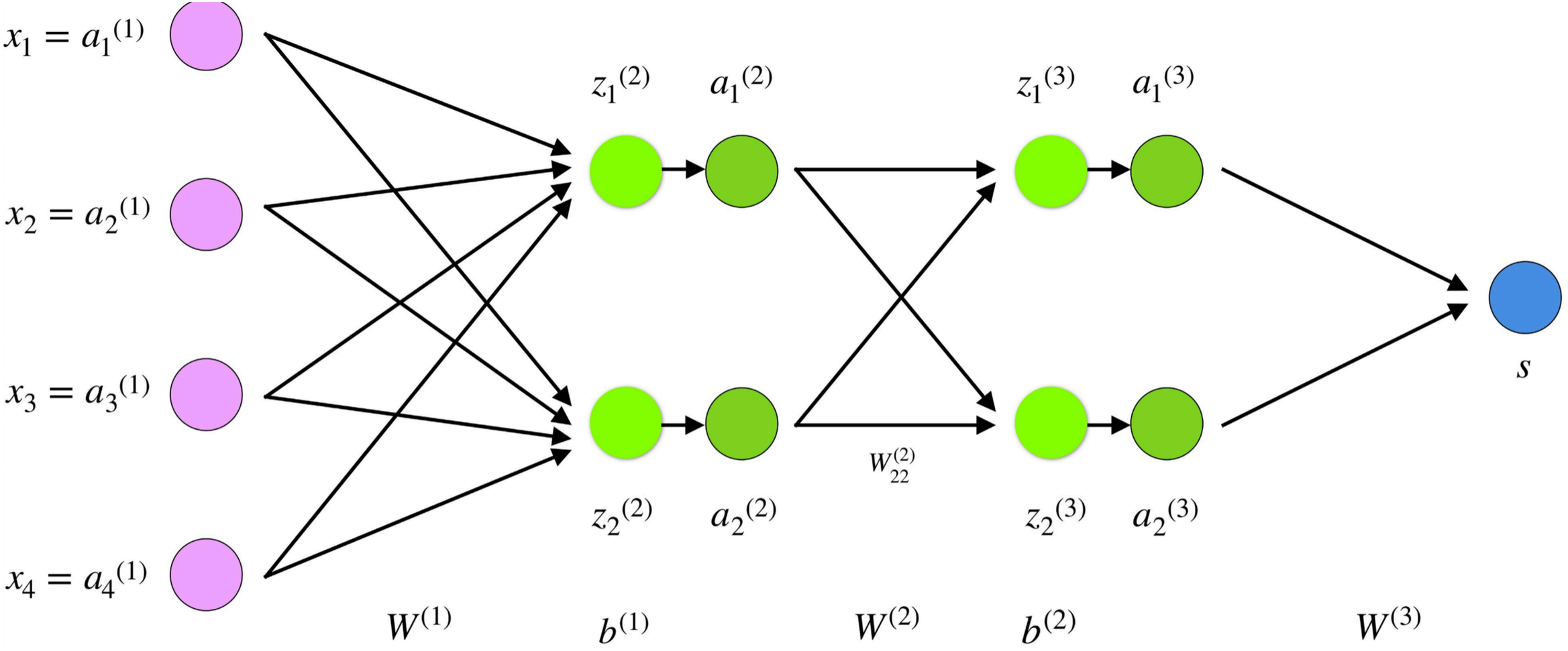
The derivative with respect to z^L is the same here

Now let's look at the equation \delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l)

Recall z^(3) = [2 2]

delta_1^2 = (w_{11}^3 delta_1^3 + w_{21}^3 delta_2^3)*sigma'(-7)= (something)*0 = 0

delta_1^2 = (w_{11}^3 delta_1^3 + w_{21}^3 delta_2^3)*sigma'(2)= (3*1 + 3*1)*1 = 6



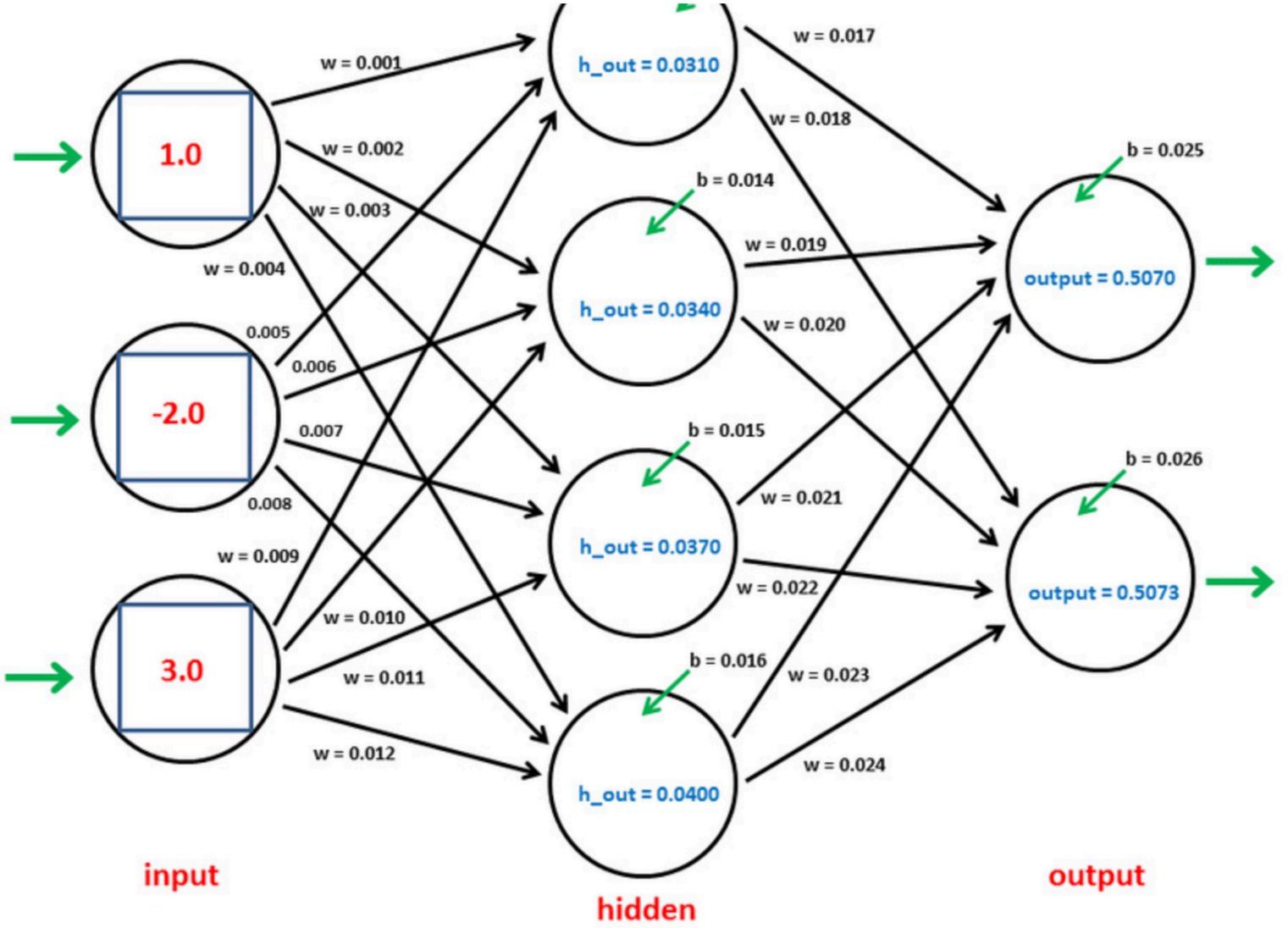
Now let's compute the derivatives with respect to w_{11}^2 and w_{21}^2

d C^i/(d w_{jk}^l)= \delta_j^l a_k^{l-1}

So the derivative with respect to w_{11}^2 =delta_1^2* a_1^1 = 0

On the other hand, the derivative with respect to w_{21}^2 is

delta_2^2* a_{1}^1 = 6*(-1) = -6



Next: what if a neural network has multiple outputs?

- Let us suppose the neural network has k outputs.
- That means that in the data set when we have (x_i, y_i) we should have $y \in \mathbb{R}^k$.
- Suppose that with weights w the neural network outputs $f_w(x_i)$. A natural objective to minimize is $(1/2)\,||f_w(x_i) - y_i||_2^2$. $= \min_w \frac{1}{2} \|f_w(x_i) - y_i\|_2^2$
- So we want to compute the gradient of this new objective.
- OK, but the new objective is $\frac{1}{2}\sum_j ([f_w(x_i)]_j - y_j)^2$
- It's the sum of the squared errors. Consequently, the derivative is exactly equal to the sum of the derivatives of the objectives $\frac{1}{2}([f_w(x_i)]_j - y_j)^2$
- ...each of these derivatives can be computed just as we did. So one possibility would be to do k backpropagations.
- ...think about it: do you really need to do k backpropagations? Can you save on computation?

- Let's come back to our motivating considerations.
- We have a data set $(x_1, y_1), \ldots, (x_n, y_n)$ and that is fixed.

  We fix an architecture.

  We now need to choose weights to fit the data.
- So we want to solve $\min_w \sum_{i=1}^{m} \frac{1}{2m}(f_w(x_i) - y_i)^2$.
- We do gradient descent $w_{t+1} = w_t - \alpha_t \frac{1}{m} \sum_{i=1}^{m} (f_w(x_i) - y_i) \frac{\partial f_w(x_i)}{\partial w}(w_t)$
- We now know how to compute the gradient! Just use backpropagation.
- There is still a second issue: we need to sum over the entire data set.

- Consider Imagenet, a data set released in 2010 which has been a catalyst for much of ML research.
- 14 Million images.
- 20,000 categories.
- About 150GB in total.
- If each iteration of gradient descent meant summing over the entire dataset, you wouldn't be able to do many iterations.
- Solution: for the last term, $w_{t+1} = w_t - \alpha_t \frac{1}{m} \sum_{i=1}^{m} (f_w(x_i) - y_i) \frac{\partial f_w(x_i)}{\partial w}(w_t)$

  pick b data points at random, and just average the terms corresponding to them:

  $$w_{t+1} = w_t - \alpha_t \frac{1}{b} \sum_{i \in \{i_1, i_2, \ldots, i_b\}} (f_w(x_i) - y_i) \frac{\partial f_w(x_i)}{\partial w}(w_t)$$

- Motivation: this has the right expectation. This method is called SGD.

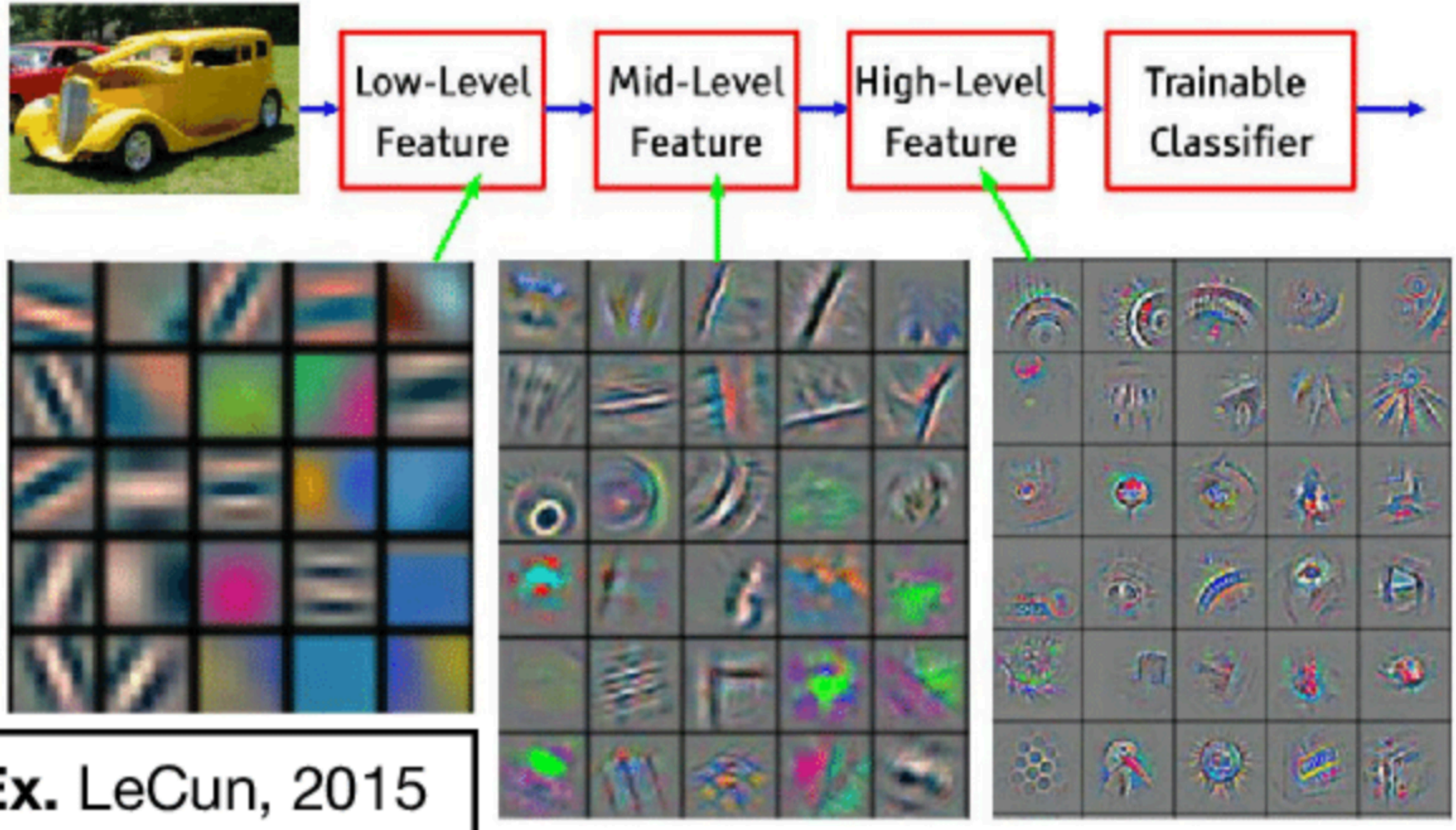
- Context: neural networks began to be the focus on ML research in the early 2010s after decades of neglect.
- Turns out that the main thing they needed was:
  — tons of data (e.g., millions of images)
  — large size (millions of parameters, large depth)
  — lots of SGD iterations (though now there are methods better than SGD).
- It's something of a controversy about who invented all this stuff first, see https://people.idsia.ch/~juergen/who-invented-backpropagation.html for your entertainment

- So let’s give this a try?
- ...not just yet. Need two more tricks which turn out to be really important to making this work.
- [details really matter here]
- First of all...let’s think of about why this might work.
- Key idea: image recognition is a composition.

  You have low-dimensional features (lines)

  These are ingredients of medium-dimensional features (brow, eye, nose)

  These in turn are ingredients of higher dimensional features (face, car)



Ex. LeCun, 2015

- This suggests: let us use something called a one-hot encoding.
- Suppose you are doing recognition of handwritten digits. The label is $\{0,\ldots,9\}$. We will encode 0 as $\begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ and 1 as $\begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ and so on, until 9 as $\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$.
- So the label, instead of being a number, is a vector in $\mathbb{R}^{10}$.
- Why is this better?
- The point is that if the class measures which combination of features, it is much easier to learn when the output is of this form.
- Consider the layer **before** the last one.
  Suppose the first entry = whether there is a brow in the picture
  Second entry = whether there is an eye in the picture
  Third entry = whether there is a wheel in the picture
  Four entry = whether there is a license plate number in the picture
  If we want to output either “face” or “car”, a natural way to do it is to check if
  “First entry + second entry” is bigger than “third entry + fourth entry”
- By contrast, if “car” mapped to output of 6 while face mapped to output of 9, the mapping would be more involved.

- So, for recognition of hand-written digits, we have the output as a vector in $\mathbb{R}^{10}$.
- We will interpret the number at location i as how likely the network thinks the output is $\mathbf{e}_i$ (this is the notation for the vector with a one in the i'th place and zero elsewhere).
  Higher = more likely.
- More formally, for a vector $z \in \mathbb{R}^k$, we define the function

$$f(z) = \frac{1}{e^{z_1} + \cdots e^{z_k}} \begin{pmatrix} e^{z_1} \\ e^{z_2} \\ \cdots \\ e^{z_k} \end{pmatrix}$$

- This is called the softmax function. Why does that name make sense?
- We use the soft-max as an activation function for the last layer.
- Note that, unlike the activation functions we used previously, this one depends on the entire vector.
- We thus interpret the output of the neural network, a vector in $\mathbb{R}^{10}$, as how likely it thinks each of the 10 possibilities is...and this actually makes sense because the output is a probability vector.

- Let’s take this one step further.

  Suppose the neural network outputs the distribution

  $$\begin{pmatrix} 3/4 \\ 1/4 \end{pmatrix}$$

  in the case of two possibilities.

  The correct answer is $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

  What cost should it pay?

- One possibility: $(3/4 - 1)^2 + (1/4 - 0)^2$.
- Actually, this is not a great choice of cost here. Let’s discuss why.

$1+1=?$

a) 2 80%

b) 3 15%

c) 4 5%

$\Big\} \; 0.8/1$

Q: ...—

A)

B)

Your think:

- $P(A \text{ correct}) = 3/4$
- $P(B \text{ correct}) = 1/4$

Option 1: Your answer

$3/4 \rightarrow A$

$1/4 \rightarrow B$

$E[\text{points}] =$

$= 3/4 \cdot 3/4 + \frac{1}{4} \cdot \frac{1}{4} = \frac{10}{16} = \frac{5}{8}$

Better ins:

$1 \rightarrow A$

$0 \rightarrow B$

$E[\text{points}] =$

$= \frac{3}{4} \cdot 1 + 0 \cdot 1/4 = \frac{6}{8}$

Grading scheme:

- If you put down x for the right ans, I give you $f(x)$ points

You think the true prob. are $(p, 1-p)$

You put down $(q, 1-q)$

by solving $\max_q \left(p f(q) + (1-p) f(1-q) \right)$

$$\frac{d}{dq} = p f'(q) - (1-p) f'(1-q) = 0$$

Conclusion

↳ I want $\Rightarrow$ $p f'(p) - (1-p) f'(1-p) = 0$

$$p f'(p) = (1-p) f'(1-p)$$

Idea:

Find a func. f s.t. $\rightarrow x f'(x) = \text{const}$ for all x

$$f(x) = c \log_2(x) + \Delta$$

$f(1) = 1$ $\quad$ $f(1/2) = 0$

$\hookrightarrow D = 1$ $\quad$ $\hookrightarrow C = I$

$$f(x) = 1 + \log_2(x)$$

$$f(x) = \log_2(x)$$

- Suppose your next exam is multiple choice.
- You are allowed to submit a probability distribution over the outcomes.
- For example, if you have a question with two answers A and B — and you think the probability that A is correct is 3/4 — then you are allowed to submit the distribution $(3/4, 1/4)$ as your answer.
- I will grade it as follows: if the answer is A, you will gain 3/4 of a point.
  If the answer is B, you will gain 1/4 of a point.
- Suppose you think the probability that the answer is A is 3/4. What should you do?
- One possibility: (3/4, 1/4). You expect to gain
  $(3/4) \cdot (3/4) + (1/4) \cdot (1/4) = 10/16.$
- Second possibility: $(1,0)$. You expect to gain $(3/4) \cdot 1 + (1/4) \cdot 0 = 3/4 = 12/16.$
- This scheme encourages you to lie!

- This gives rise a fun question:

  suppose you, as a student, submit the probability distribution $(q, 1-q)$.

  The true answer is 1.

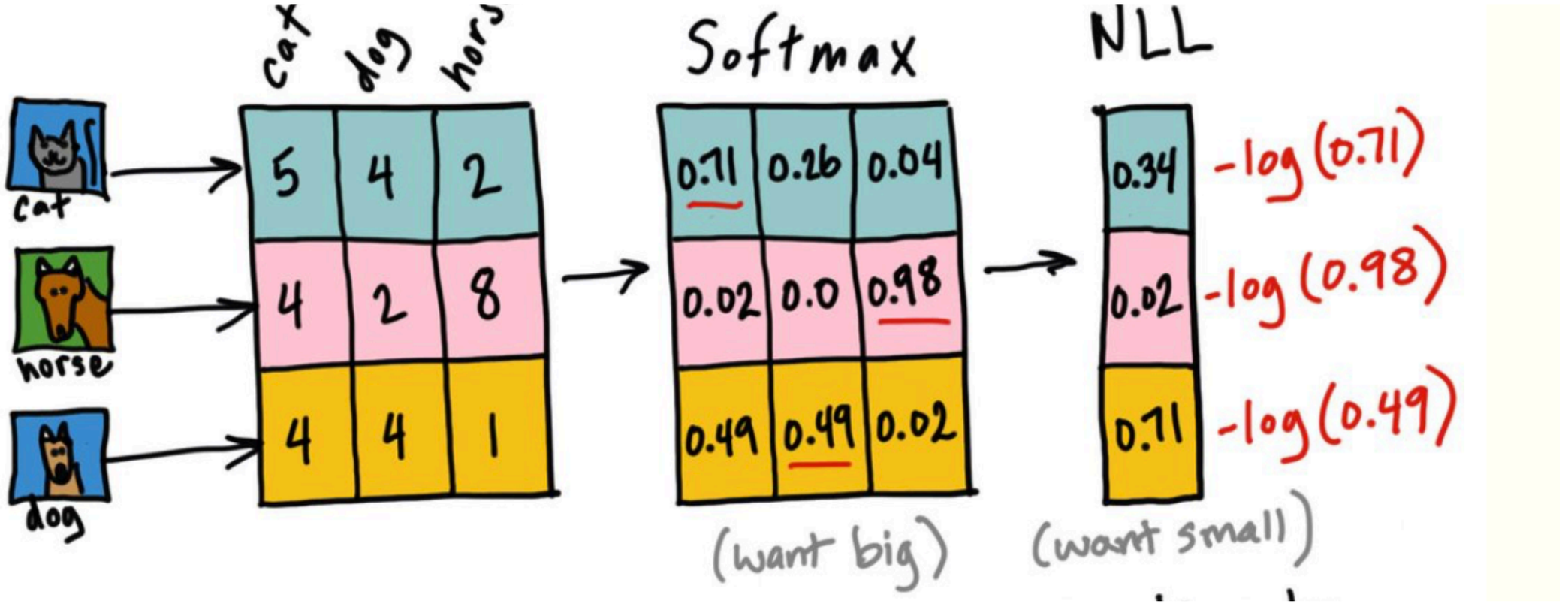
  What grade should I give you to encourage you to tell the truth?
- Suppose I assign you a grade of $f(q)$. Naturally, I announce the function $f(\cdot)$ ahead of time.
- If you think the true distribution is $(p, 1-p)$ you will submit $(q, 1-q)$ that maximizes

  $pf(q) + (1-p)f(1-q)$ over $q \in [0,1]$.
- You will solve $pf'(q) - (1-p)f'(1-q) = 0$.
- So I need to choose f with the property that $pf'(p) - (1-p)f'(1-p) = 0$ for all p.
- A natural choice: $pf'(p)$ should be constant.
- $f'(p) = C/p$
- $f(p) = C \log p + D$
- Let's further set $f(1) = 1, f(1/2) = 0$. This means $f(p) = \log_2(p) + 1$.
- Equivalently, can choose $f(p) = \log_2(p)$.
- Punchline: I've got to give you points equal to the log of the probability you gave to the correct answer.

- Back to neural networks. Your neural network outputs a vector of probabilities in $\mathbb{R}^{10}$.
- The “credit” you should give it is the log of the entry corresponding to the correct answer.
- But our algorithms minimize the loss the NN suffer.
- So you say the loss it suffers is the negative of the log of the entry corresponding to the correct answer:

$$-\log \frac{e^{z_i}}{e^{z_1} + e^{z_1} + \cdots e^{z_k}}$$

- This loss is nonnegative and equals zero exactly when you assign a 1 to the correct answer and zero to everything else.
- But also, this loss will encourage you never to give a zero probability to any posssibility. Why?



Picture —> Last layer of NN —> after softmax —> log loss suffered

- Now let's discuss how to do back propagation with the categorical cross-entropy loss.
- A slight difference is that the activation function on the last layer depends on the entire vector. This doesn't change much, as we'll see.
- As before, we draw a mini batch of data points (x_i, y_i) of size m and update all the weights as

  $$w_{jk}^l(t+1) = w_{jk}^l(t) - \alpha_t \frac{1}{m} \sum_{i=1}^{m} \frac{\partial C^i}{\partial w_{jk}^l}(w(t))$$

  where $C^i = l_{\text{cce}}(f_w(x_i), y_i)$
- Let $z_1, \ldots, z_k$ be the output of the last layer of the neural network and $p_j = \dfrac{e^{z_j}}{\sum_i e^{z_i}}$

  We have that $C^i = l_{\text{cce}}\left(\dfrac{1}{\sum_i e^{z_i}} \begin{pmatrix} e^{z_1} \\ \vdots \\ e^{z_n} \end{pmatrix}, \mathbf{e}_k \right)$

  where k is the right label on example i.
- So $C^i = -\log \dfrac{e^{z_k}}{\sum_i e^{z_i}}$.
- Key point: once we compute $\dfrac{\partial C^i}{\partial z_j}$ for all $j = 1, \ldots, k$, you just do back propagation just as before!

  More formally: go back to Eq. (!!*), and use Eq (2),(3),(4) once you have a replacement for Eq. (1). Only change is at the last layer!

- So we just need to compute the derivatives of $C^i = -\log \frac{e^{z_k}}{\sum_i e^{z_i}}$.
- Clearly, $\frac{\partial C^i}{\partial z_j} = -\frac{\sum_i e^{z_i}}{e^{z_k}} \frac{\partial}{\partial z_j} \frac{e^{z_k}}{\sum_i e^{z_i}}$.
- Two possibilities: $j = k$ and $j \neq k$.
- In the latter case, $\frac{\partial C^i}{\partial z_j} = \frac{\sum_i e^{z_i}}{e^{z_k}} \frac{e^{z_k}}{(\sum_i e^{z_i})^2} e^{z_j} = \frac{e^{z_j}}{\sum_i e^{z_i}} = p_j$
- In the former case, can use
  $$\frac{d}{dx} \frac{x}{x+a} = \frac{1 \cdot (x+a) - 1 \cdot x}{(x+a)^2} = \frac{a}{(a+x)^2}$$
  to obtain
  $$\frac{\partial C^i}{\partial z_j} = -\frac{\sum_i e^{z_i}}{e^{z_k}} \frac{\sum_i e^{z_i} - e^{z_k}}{(\sum_i e^{z_i})^2} e^{z_k} = -\frac{\sum_i e^{z_i} - e^{z_k}}{\sum_i e^{z_i}} = p_k - 1$$
- Makes sense: if you want to **decrease** the cost, you should move in the direction of the negative of the derivative, which will mean increasing z_k and decreasing all the other z_j.
- [**Note to self: double check this calculation, I didn't get it from any source]**

- OK, time to try this on a real world example.
- We'll use the MNIST data set https://en.wikipedia.org/wiki/MNIST_database
- Contains 60,000 images of hand-written digits along with labels. Each image is 28x28 pixels.
- Created in 1998. The classic paper http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf achieving 99% recognition rate on this.
- https://colab.research.google.com/drive/1ZpOWilfKB9ZpKYumKIEfxGm4wyeLS6b_?usp=sharing
- Let's try on CIFAR-10
- https://colab.research.google.com/drive/1ZpOWilfKB9ZpKYumKIEfxGm4wyeLS6b_?usp=sharing

- ADAM is an alternative to SGD based on two simple ideas.
- Let's take the SGD iteration

  $$w_{t+1} = w_t - \alpha_t \left[\nabla f(w_t) + \text{noise} \right]$$

- In neural networks, the noise comes from the selection of the mini-batch.
- How can we combat the noise?
- First idea: since the gradients don't change that quickly (because α_t is small), let's take linear combination of the gradients:

  Initialize $v_0 = 0$

  $$g_t = \nabla f(w_t) + \text{noise}$$

  $$v_t = \beta v_{t-1} + (1 - \beta) g_t$$

  $$w_{t+1} = w_t - \alpha_t v_t$$

- Let's consider this recursion:

  $v_t = \beta v_{t-1} + (1-\beta) g_t$
- We have $v_1 = \beta v_0 + (1-\beta) g_1$
- Next, $v_2 = \beta v_1 + (1-\beta) g_2 = \beta^2 v_0 + \beta(1-\beta) g_1 + (1-\beta) g_2$

  ...this is a convex combination!
- Next, $v_3 = \beta v_2 + (1-\beta) g_3 = \beta^3 v_0 + \beta^2(1-\beta) g_1 + \beta(1-\beta) g_2 + (1-\beta) g_3$

  ...this also is a convex combination.
- In general, you can show that convex combinations of convex combinations are convex combinations.
- The point: v_t is a convex combination of all the gradients, with the coefficients on the past decaying geometrically.
- In general, $v_t = \beta^t v_0 + \beta^{t-1}(1-\beta) g_1 + \cdots + (1-\beta) g_t$

- Let's look at this equation: $v_t = \beta^t v_0 + \beta^{t-1}(1-\beta)g_1 + \cdots + (1-\beta)g_t$
- A sort of problem is that because $v_0 = 0$, we have that there will be a bias towards zero.
- Not a huge problem: the bias will decrease over time. The coefficients on everything except v_0 add up to $1 - \beta^t$.
- One possibility is to start the method at time one initialized at g_1. That is what I would do.
- What the creators of ADAM did was to choose the approach

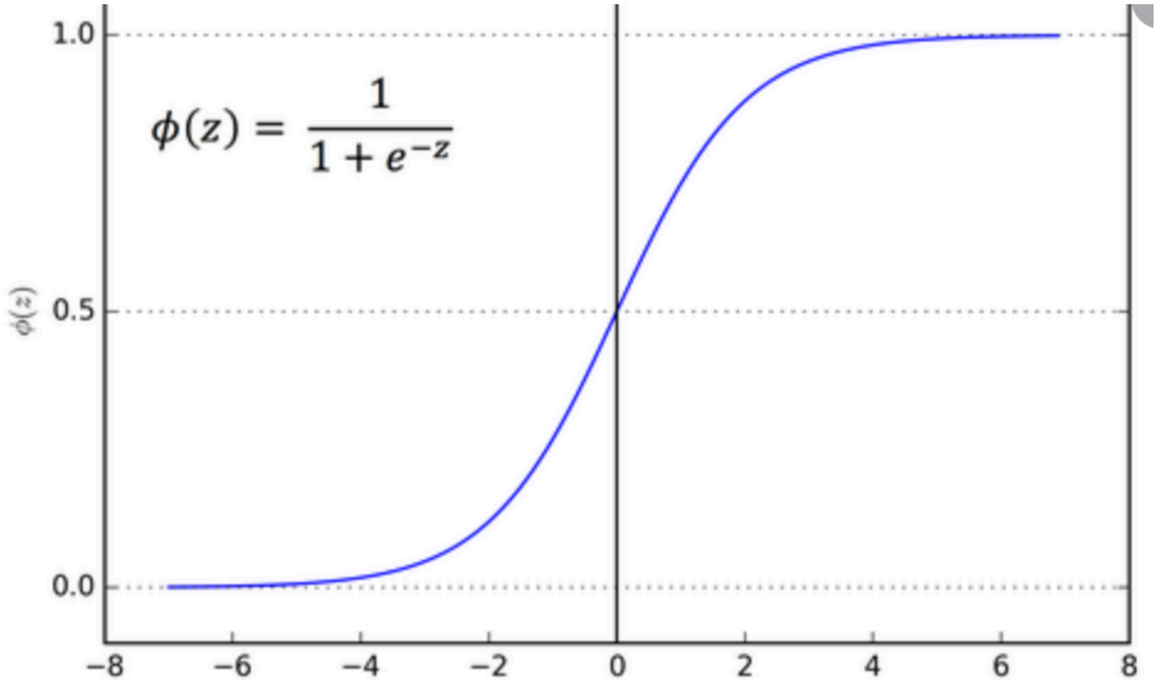
$$g_t = \nabla f(w_t) + \text{noise}$$

$$v_t = \beta v_{t-1} + (1-\beta)g_t$$

$$\hat{v}_t = \frac{v_t}{1-\beta^t}$$

$$w_{t+1} = w_t - \alpha_t \hat{v}_t$$

- Next insight: we can have the situations where the gradients are either huge or close to zero.
- If you get a huge gradient while your step-size α_t is not sufficiently small, your update will be all over the place.
- On the other hand, consider any of our activation functions that were flat at the extremes.



For very large or very small z, the gradient is close to zero

- Solution: try to normalize by the norm of the gradient, i.e., instead of updating as

  $$w_{t+1} = w_t - \alpha_t \left[\nabla f(w_t) + \text{noise} \right]$$

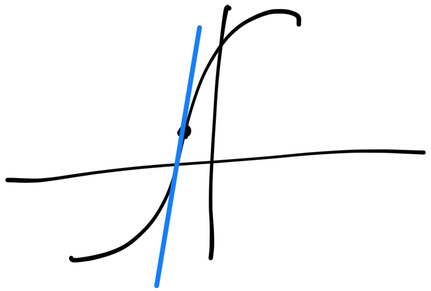
  instead update as

  $$w_{t+1} = w_t - \alpha_t \frac{\nabla f(w_t) + \text{noise}}{|| \nabla f(w_t) + \text{noise} ||_2}$$

- This is a good idea, but there is another complication.
  Suppose everything is perfect...on every minibatch, you classify things perfectly.
  All your gradients are zero, and you shouldn't move anywhere.
- Unfortunately, sometimes gradients which are zero are evaluated to something like 10^{-12} due to round-off error.
- Solution:

  $$w_{t+1} = w_t - \alpha_t \frac{\nabla f(w_t) + \text{noise}}{\sqrt{|| \nabla f(w_t) + \text{noise} ||_2^2 + \epsilon}}$$

  where ϵ is usually taken to be 10^{-8}.
- So a gradient of 10^{-6} will get normalized into a direction to follow. But a gradient of 10^{-10} will get ignored.
- OK, but this is SGD...we are using a linear combination of the gradients.
- No problem, just normalize that.



$$\frac{f'(x)}{|f(x)|} \in \{1, -1\}$$

- This finally brings us to ADAM:

  Initialize $v_t = 0, q_t = 0$

  $g_t = \nabla f(w_t) + \text{noise}$

  $v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$

  $\hat{v}_t = \frac{v_t}{1 - \beta_1^t}$

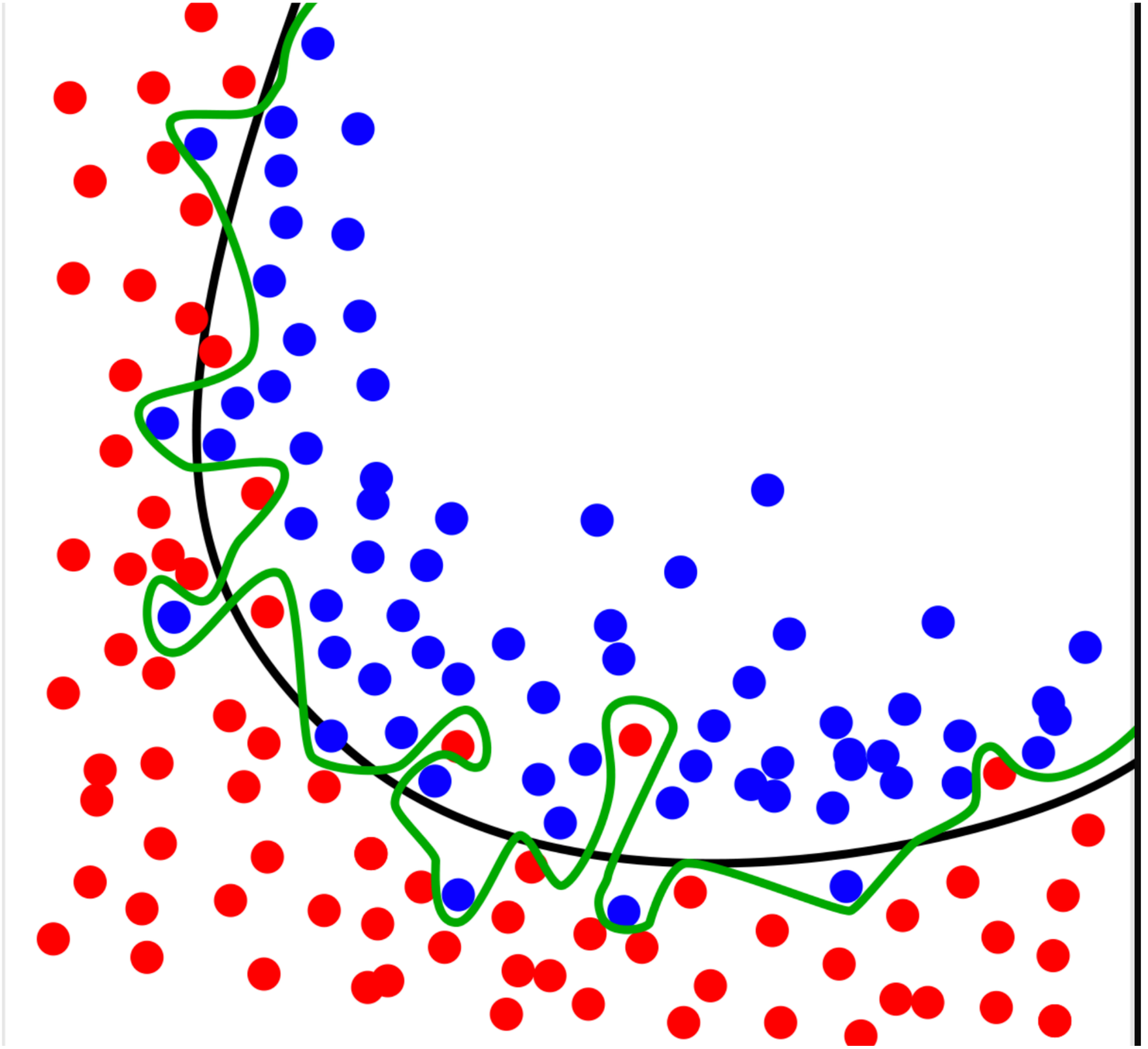
  $q_t = \beta_2 q_{t-1} + (1 - \beta_2) || g_t ||_2^2$

  $\hat{q}_t = \frac{q_t}{1 - \beta_2^t}$

  $w_{t+1} = w_t - \alpha_t \frac{1}{\sqrt{\hat{q}_t + \epsilon}} \hat{v}_t$

- Default values $\beta_1 = 0.9, \beta_2 = 0.999$. Why are they different? I have no idea.
- There is some skepticism that ADAM is a good method: https://parameterfree.com/2020/12/06/neural-network-maybe-evolved-to-make-adam-the-best-optimizer/

- Let’s discuss a few tips and tricks. As we’ve discussed, you can have the phenomenon where performance on the training set improves, while performance on the test set does not.
- Intuitively: the NN is learning to memorize the data.
- This is sometimes called *overfitting*.
- How do we prevent this?
- One possibility: don’t train too long. Sometimes called *early stopping*.
- Not a bad idea. But there’s another useful trick.



Overfitting

- Dropout: during training, set each input of each neuron to zero with probability p.
- Regular NN: let's write as

  $$x_i^{l+1} = \mathbf{w}_i^{l+1}\mathbf{y}^l + b_i^{l+1}$$

  $$y_i^{l+1} = f(x_i^{l+1})$$
- With dropout:

  $$r_j^l = \text{ Bernoulli } (p).$$

  $$\tilde{y} = \mathbf{r} \odot \mathbf{y}$$

  $$x_i^{l+1} = \mathbf{w}_i^{l+1}\tilde{\mathbf{y}}^l + b_i^{l+1}$$

  $$y_i^{l+1} = f(x_i^{l+1})$$
- You usually don't do dropout on the first (input) layer, but on intermediate layers.
- When the procedure is finished, you multiply the weights by $1/p$.
- Motivation: this is like training an infinite ensemble of neural network that share weights
- ...it's very hard for the system to memorize the data when you do this.

- Next trick: batch normalization.
- Suppose you sample a batch $\mathscr{B}$ and consider one particular neuron output x (before passing into the activation). Batch normalization layer does the operation

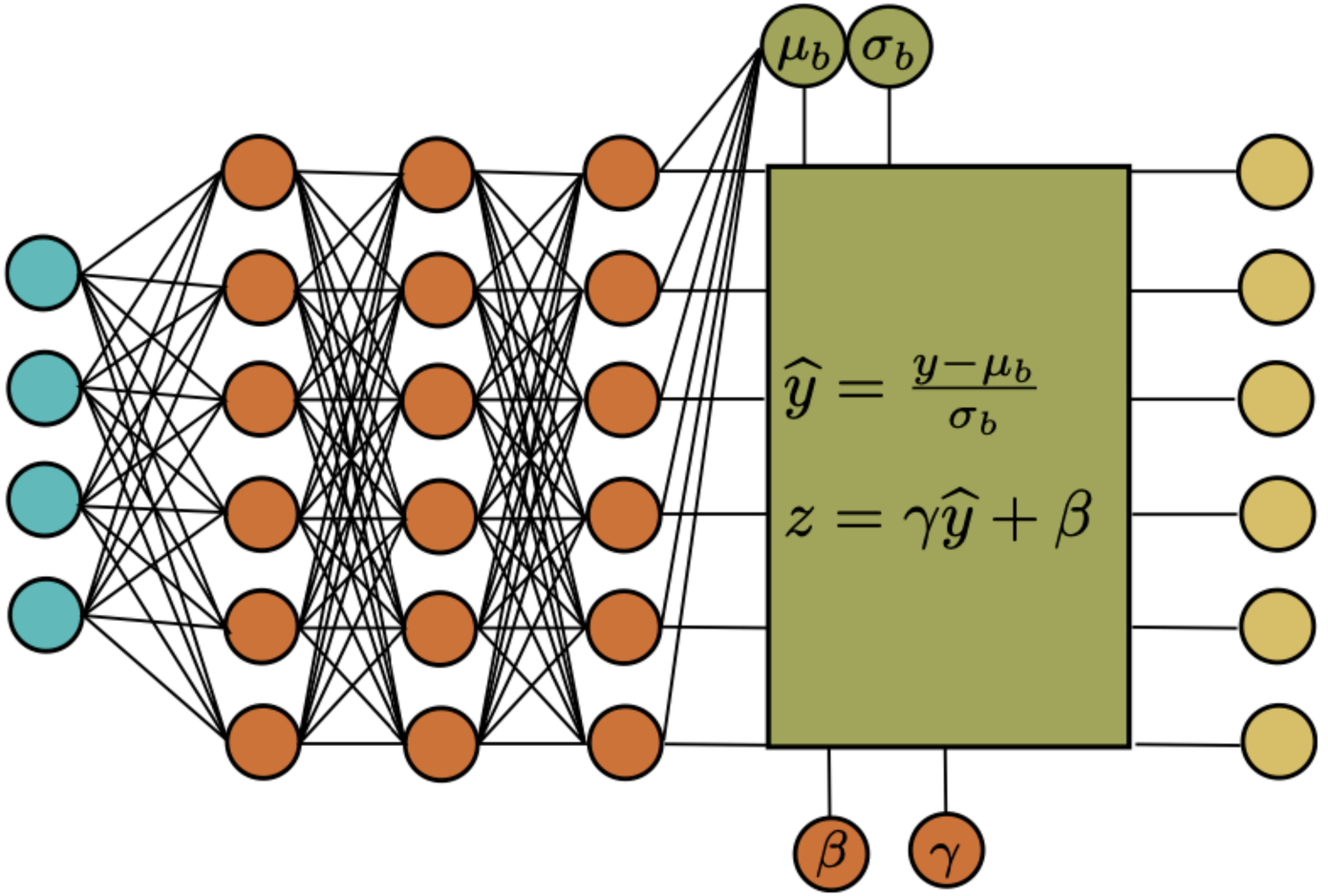
  $$\hat{x} \leftarrow \frac{x - \mu_{\mathscr{B}}}{\sigma_{\mathscr{B}}}$$

  where $\mu_{\mathscr{B}}, \sigma_{\mathscr{B}}$ is the variance of the output of that neuron **over the sampled batch** $\mathscr{B}$.

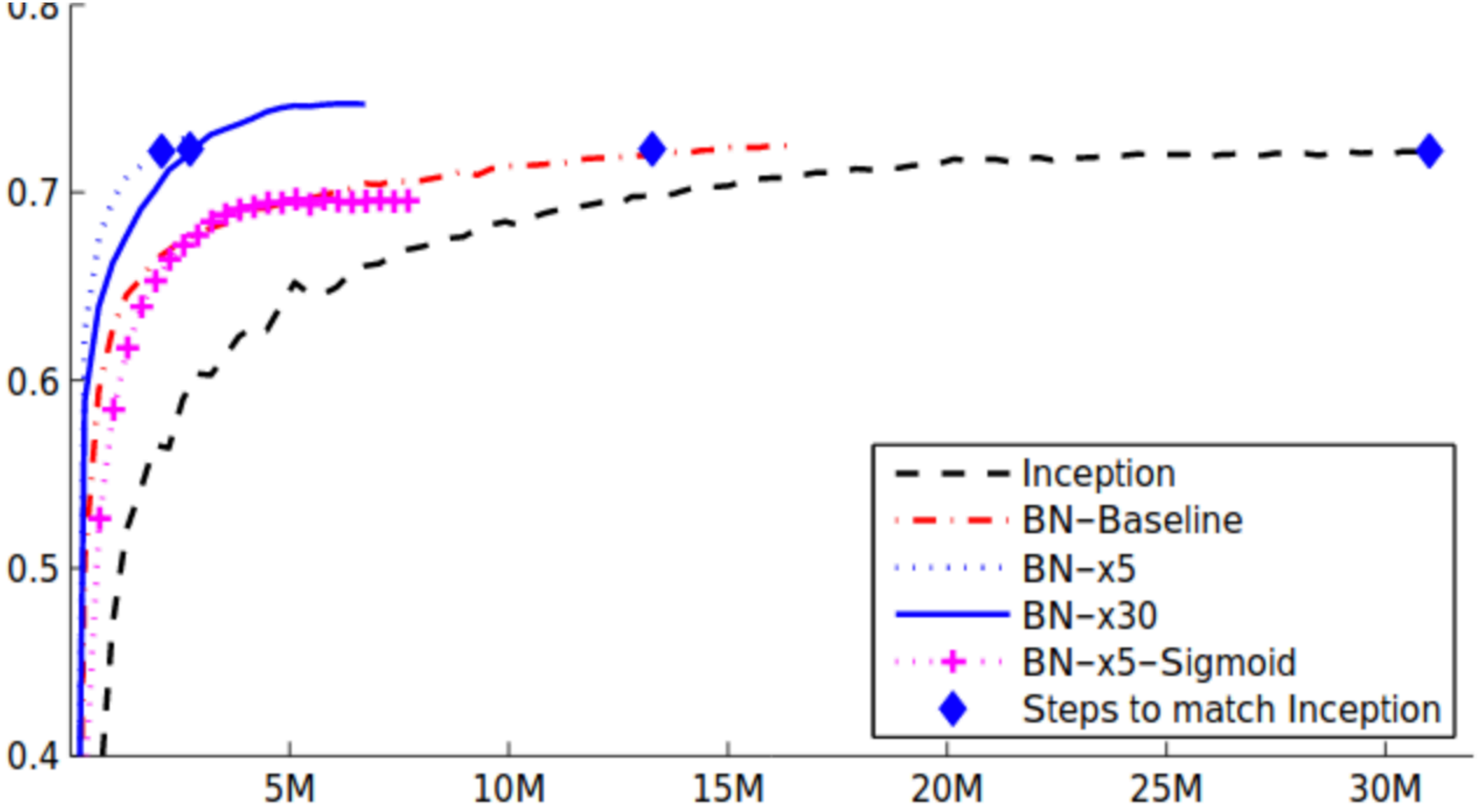
  The output of the neuron is then set to be $x_{\text{new}} = \gamma \hat{x} + \beta$.
- You try to learn the best γ, β (one for each neuron).
- After training is over, you implement

  $$x_{\text{new}} = \gamma \frac{x - \mu}{\sigma} + \beta$$

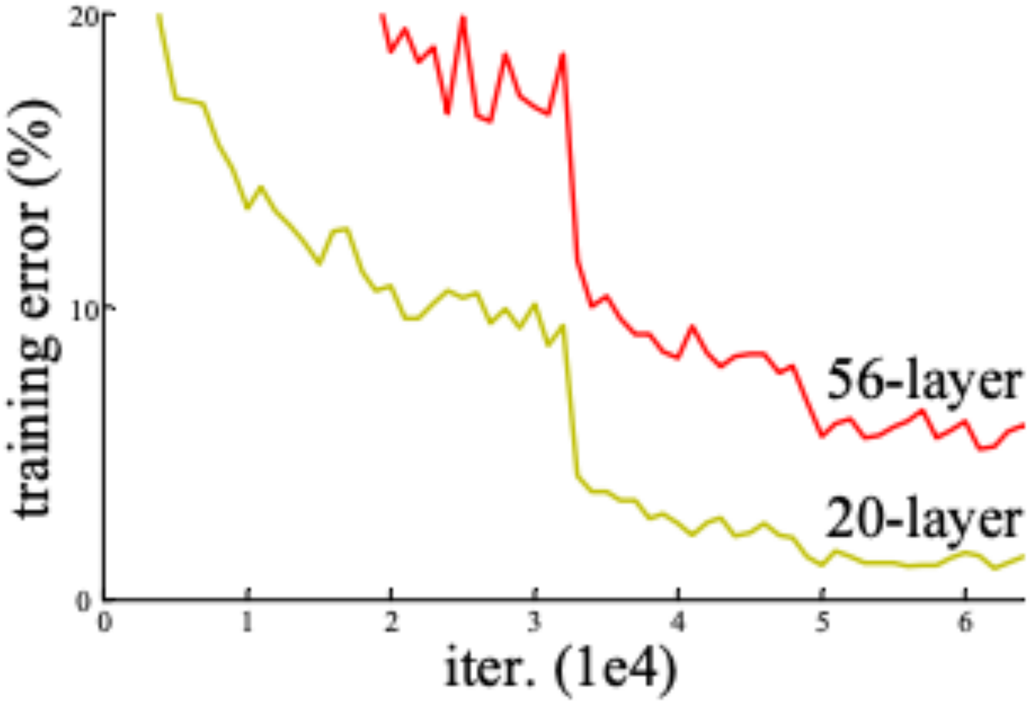
  where μ, σ are now the mean and variances over the entire training set.
- Doesn't seem to do very much!
- Original authors had a particular motivation for this, which turned out to be wrong.
- However: empirically, this helps.
- Right now, people often say batch normalization can help decrease the gradient sizes.

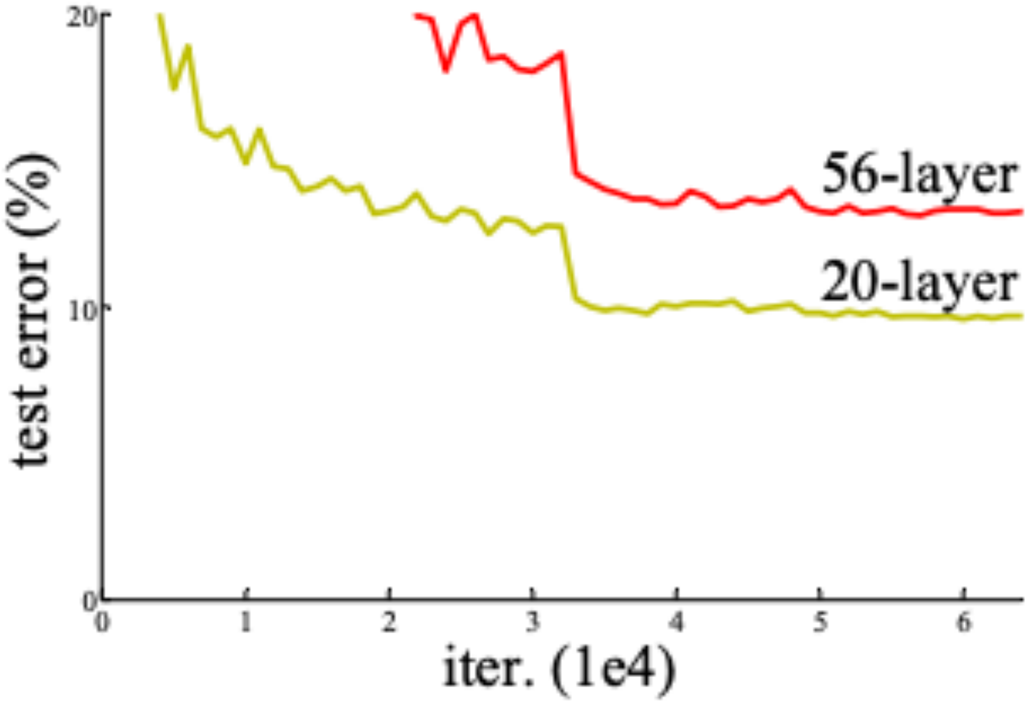


An illustration of a batch normalization layer.
Normally, it is put before the activation function.



From the original Batch Normalization paper
Different curves correspond to different learning rates
"Inception" refers to https://arxiv.org/pdf/1409.4842v1.pdf

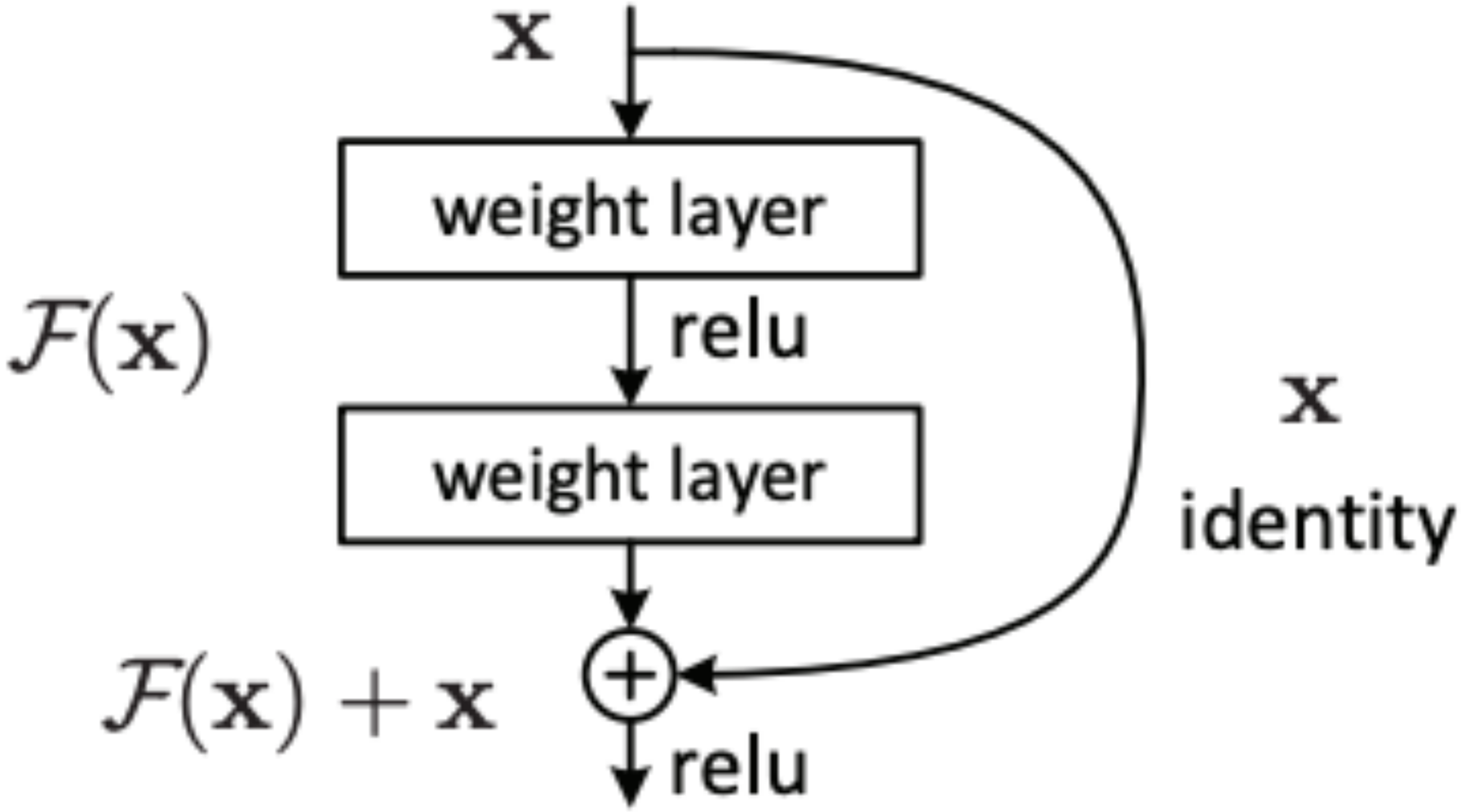




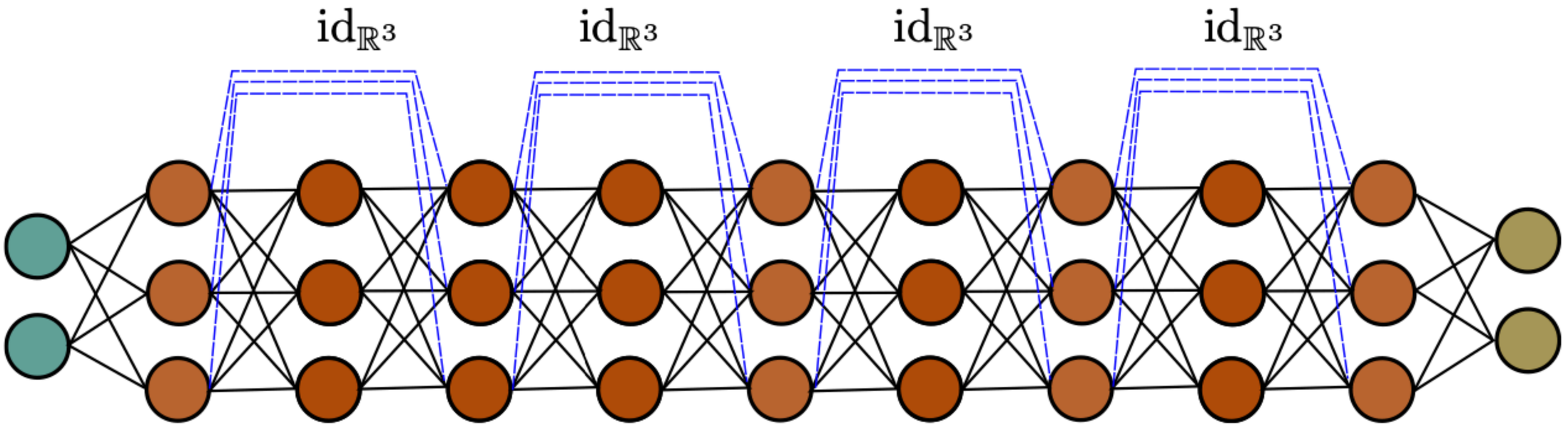
Final trick: residual networks

Motivated by this picture from https://arxiv.org/pdf/1512.03385.pdf

Poorer performance with deeper networks. Why is this puzzling?



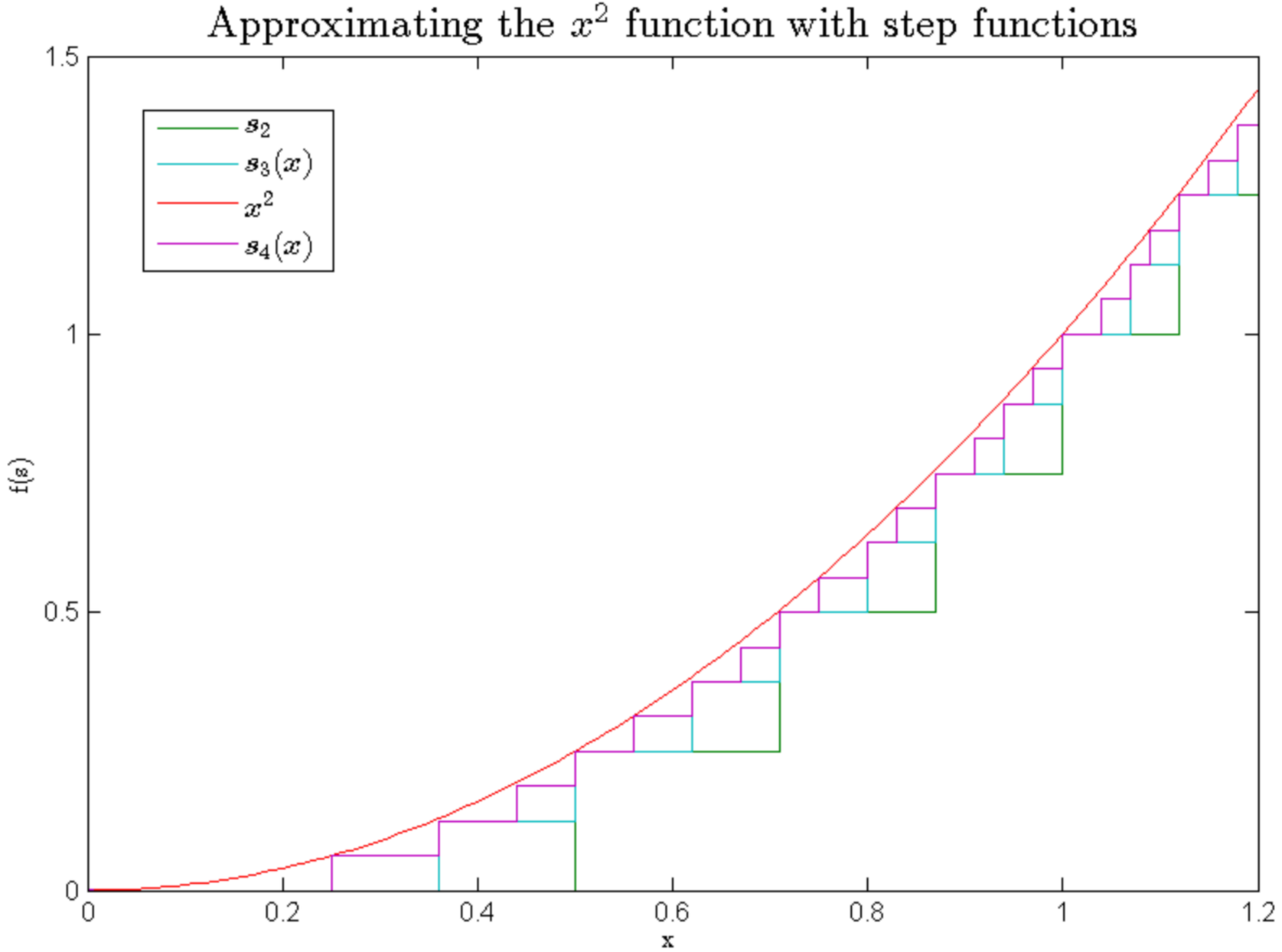
Solution: use blocks like this
Each block is sigma(W_2 sigma(W_1 x) + x)
This is called a ResNet
Somewhat surprisingly, this really helps!



Another illustration of a residual network

- Why am I telling you this at this level of detail?
  For example, you can train all these using backpropagation, but I have not given you allthe details.
- Because you can use these things in without fully understanding all the details.
- In Keras, just write model.add(Dropout(0.5)) to add a dropout layer that deletes neurons with probability 1/2.
- For batch normalization, just type model.add(BatchNormalization())
  at the place you want it.
- For residual blocks, see e.g., https://gist.github.com/mjdietzx/5319e42637ed7ef095d430cb5c5e8c64
- Similar solutions exist in pytorch.
- Keras even allows you to load the original resnet network with a single command: https://keras.io/api/applications/resnet/

- Let's talk a little bit about what makes neural networks so good.
  This is not really something we can answer fully at this time.
  But we can give some facets of the answer.
- One fact is the following statement: *neural networks can approximate any continuous function*.
- Let's be more precise. First, let us fix a bounded domain $\mathbb{D} \subset \mathbb{R}^n$. Let f by a continuous function from $\mathbb{D}$ to $\mathbb{R}$.
- For example, you can consider $f : [0,1]^n \to \mathbb{R}$ defined as $f(x_1, \ldots, x_n) = x_1 x_2^2 x_3^3 \cdots x_n^n$.
- We will say that the function g approximates f over $\mathbb{D}$ to accuracy ϵ if
  $$\int_{\mathbb{D}} ||f(x) - g(x)|| dx \leq \epsilon$$
- **Theorem:** consider an architecture where every neuron on every layer uses the ReLU activation function, except for the last layer, which has a single output and no activation function.
  For any accuracy $\epsilon > 0$, and any continuous function $f : \mathbb{D} \to \mathbb{R}$, there is a neural network with ReLU activation which approximates f over $\mathbb{D}$ to accuracy ϵ.
- Next, we will sketch a proof of this theorem.
- Warm up case: let's consider $\mathbb{D} \subset \mathbb{R}$.
  In that case, the input to the neural network is a number.



Idea 1: Any continuous function is approximated by a collection of step functions to arbitrarily small accuracy

- The function $\mathbf{1}_{[a,b]}(x) = \begin{cases} 1 & x \in [a,b] \\ 0 & \text{else} \end{cases}$ is called a *simple function*.
- What we saw on the previous page: any continuous function can be approximated by a function of the form
  $s(x) = c_1\mathbf{1}_{[a_1,b_1]}(x) + c_2\mathbf{1}_{[a_2,b_2]}(x) + \cdots + c_k\mathbf{1}_{[a_k,b_k]}(x)$
- The lower we want the approximation error to be, the more number of pieces k we will need.
- **Claim**: if we can approximate simple function to arbitrary accuracy with a NN, then we can approximate any continuous function on $\mathbb{D}$ to an arbitrary accuracy.
- Indeed, suppose $f_1(x)$ approximates $\mathbf{1}_{[a_1,b_1]}(x)$, $f_2(x)$ approximates $\mathbf{1}_{[a_2,b_2]}(x)$, and so forth. Then $\hat{f}(x) = \sum_{i=1}^{k} c_i f_i(x)$ will approximate $s(x)$.
- Moer formally,

$$\epsilon = \int_{\mathbb{D}} |\hat{f}(x) - s(x)| = \int_{\mathbb{D}} \left| \sum_{i=1}^{k} c_i f_i(x) - \sum_{i=1}^{k} c_i \mathbf{1}_{[a_i,b_i]}(x) \right|$$

$$\leq \sum_{i=1}^{k} c_i \int_D \left| \hat{f}_i(x) - \mathbf{1}_{[a_i,b_i]}(x) \right|$$

So if each $\hat{f}_i$ approximates $\mathbf{1}_{[a_i,b_i]}$ to accuracy δ, the total accuracy is $\delta \sum_{i=1}^{k} c_i$.

If we can make δ as small as we want, then we can make ϵ as small as we want.

- OK, so we just need to show that we can approximate $\mathbf{1}_{[a,b]}(x) = \begin{cases} 1 & x \in [a,b] \\ 0 & \text{else} \end{cases}$ to any accuracy.
- Let's play around with what a neural network can do.
- OK, let's start with a ReLu: $\max(x,0)$. What does this look like?
- Next, how about we shift it: $\max(x-f,0)$. What does that look like?
- What if we subtract two such things (which we can certainly do with a NN):

  $\max(x-f,0) - \max(x-f-\Delta,0)$. What does that look like?
- How about $\dfrac{\max(x-f,0) - \max(x-f-\Delta,0)}{\Delta}$
- If you've followed this discussion, you should see that by

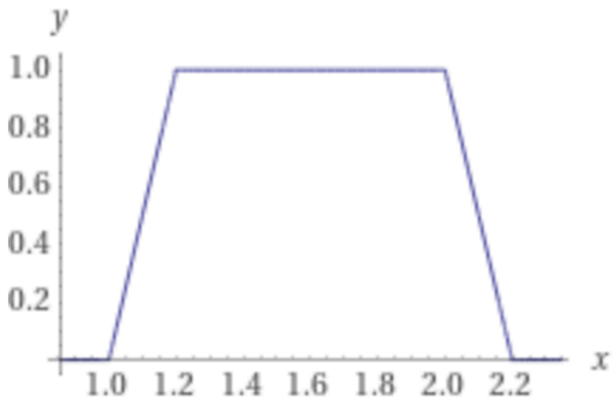
  $$\frac{\max(x-a_1,0) - \max(x-a_1-\Delta,0) - \Big(\max(x-b_1,0) - \max(x-b_1-\Delta)\Big)}{\Delta}$$

  approximates $\mathbf{1}_{[a_i,b_i]}(x)$ better and better as $\Delta \to 0$.

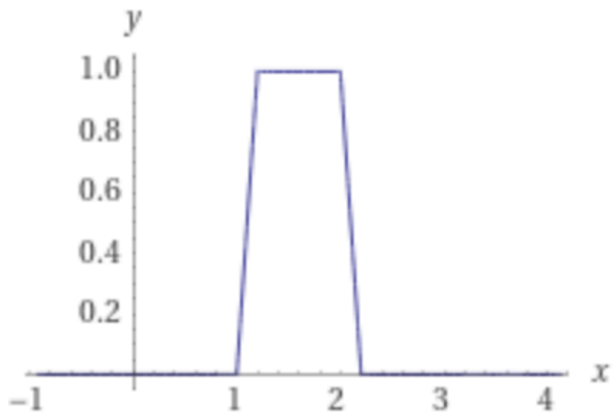
Input:

$$\frac{\max(x-1,0)-\max(x-1.2,0)}{0.2}-\frac{\max(x-2,0)-\max(x-2.2,0)}{0.2}$$

Plots:



(x from 0.9 to 2.3)



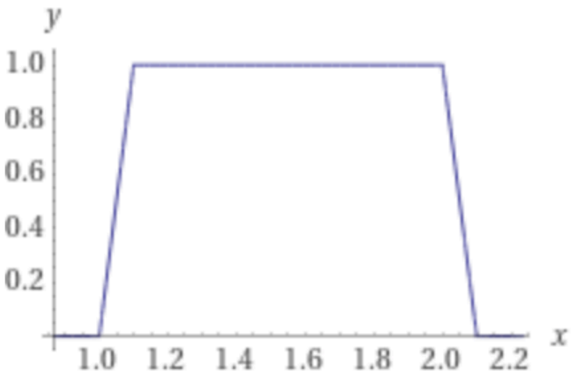
(x from −0.9 to 4.1)

Delta = 0.2

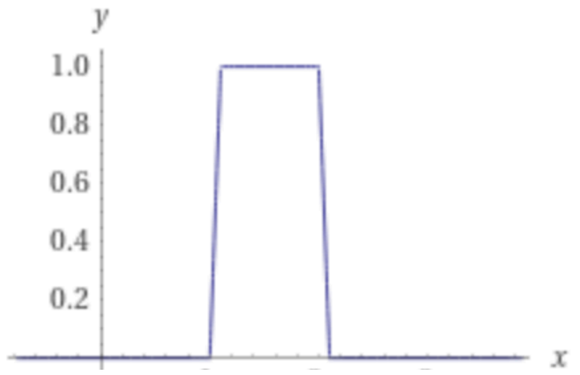
Input:

$$\frac{\max(x-1,0)-\max(x-1.1,0)}{0.1}-\frac{\max(x-2,0)-\max(x-2.1,0)}{0.1}$$

Plots:



(x from 0.9 to 2.2)

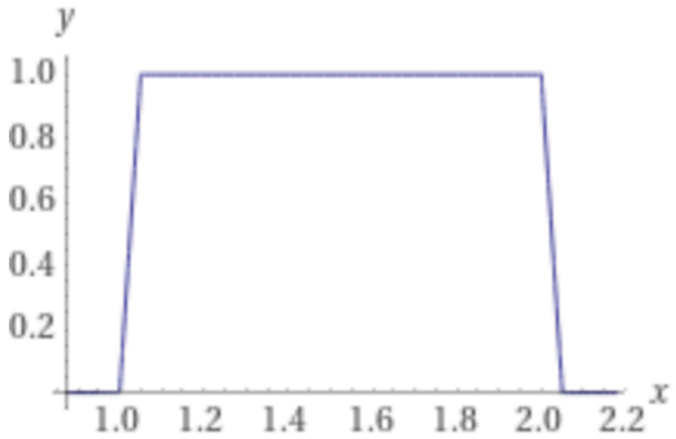


(x from −0.8 to 3.9)

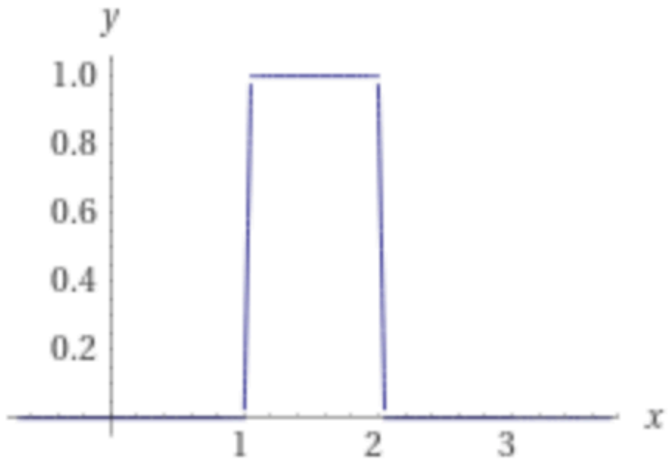
Delta = 0.1

$$\frac{\max(x-1,0)-\max(x-1.05,0)}{0.05}-\frac{\max(x-2,0)-\max(x-2.05,0)}{0.05}$$

Plots:



(x from 0.9 to 2.2)



(x from −0.7 to 3.7)

Delta = 0.05

- We we are done:
  — we showed any function can be approximated by a sum of simple functions
  — we showed that it suffices to approximate each simple function
  — we showed how to approximate each simple functions using linear combinations and maxes
- The approximation is never perfect.
- Let’s think about how the pieces we’ve discussed. How many layers are there in the approximation we’ve described?

- The answer is is 3.
- The first layer takes as the input the scalar x and computes

  $\max(x - a_1, 0)$

  $\max(x - a_1 - \Delta, 0)$

  $\max(x - b_1, 0)$

  $\max(x - b_1 - \Delta, 0)$

  $\max(x - a_2, 0)$

  $\max(x - a_2 - \Delta, 0)$

  $\max(x - b_2, 0)$

  $\max(x - b_2 - \Delta, 0)$

  etc
- On the second layer, the first neuron takes a linear combination of these to get an approximation of $\mathbf{1}_{[a_1, b_1]}$.

  The second neuron takes a linear combination to get an approximation of $\mathbf{1}_{[a_2, b_2]}$.

  etc
- The third and last layer takes a linear combination of the simple functions at the second layer to approximate the function.

- Great, so we shoed that a NN can approximate any function over a bounded domain $\mathbb{D} \subset \mathbb{R}$.
- Now we want to show the same but for $\mathbb{D} \subset \mathbb{R}^n$.
- The general form of the proof is the same. The first step is to define the simple function

$$\mathbf{1}_{\mathscr{A}}(x_1, \ldots, x_n) = \begin{cases} 1 & x_1 \in [a_1, b_1], x_2 \in [a_2, b_2], \ldots, x_n \in [a_n, b_n] \\ 0 & \text{else} \end{cases}$$

- Alternatively, $\mathbf{1}_{\mathscr{A}}(x_1, \ldots, x_n) = \mathbf{1}_{[a_1,b_1]}(x_1)\mathbf{1}_{[a_2,b_2]}(x_2)\cdots\mathbf{1}_{[a_n,b_n]}(x_n)$.
- Intuitively: this function is 1 one a "square," and zero outside of it
- Claim: any continuous function over a bounded domain $\mathbb{D}$ can be approximated by a linear combination of simple functions.

  That is, if f is continuous over $\mathbb{D}$, then we can write

$$f(x_1, \ldots, x_n) \approx \sum_j c_j \mathbf{1}_{\mathscr{A}_j}(x_1, \ldots, x_n)$$

- So just need to argue we can approximate any simple function $\mathbf{1}_{\mathscr{A}}(x_1, \ldots, x_n)$.

- We know how to approximate $\mathbf{1}_{[a_i,b_i]}(x_i)$. Indeed,

$$\mathbf{1}_{[a_i,b_i]}(x_i) \approx \frac{\max(x_i - a_i,0) - \max(x_i - a_i - \Delta,0) - \big(\max(x_i - b_i,0) - \max(x_i - b_i - \Delta)\big)}{\Delta}$$

- ... but what we want to approximate is the product

$$\mathbf{1}_{\mathscr{A}}(x_1, \ldots, x_n) = \mathbf{1}_{[a_1,b_1]}(x_1)\mathbf{1}_{[a_2,b_2]}(x_2)\cdots\mathbf{1}_{[a_n,b_n]}(x_n)$$

- ..we are only allowed to use operations a ReLU neural network can do. So we can take linear combinations, take the max of a number and zero, add and subtract constants.
  But a ReLU network cannot multiply.
- So what can we do?

- Let's look at $\mathbf{1}_{\mathscr{A}}(x_1, \ldots, x_n) = \mathbf{1}_{[a_1,b_1]}(x_1)\mathbf{1}_{[a_2,b_2]}(x_2)\cdots\mathbf{1}_{[a_n,b_n]}(x_n)$
- The trick is to write

$$\mathbf{1}_{\mathscr{A}}(x_1, \ldots, x_n) = \max\left(\left(\sum_{i=1}^{n} \mathbf{1}_{[a_i,b_i]}(x_i)\right) - (n-1), 0\right)$$

- Success!
- How many layers do we need?
- Can do with four layers:

  — first two layers as before build approximations of the scalar functions $\mathbf{1}_{[a_i,b_i]}(x_i)$ for each element on the input

  — third layer builds approximations of the simple functions $\mathbf{1}_{\mathscr{A}_j}(x_1, \ldots, x_n)$ for all the "squares" $\mathscr{A}_j$ we will use.

  — Finally layer output $\sum_j c_j \mathbf{1}_{\mathscr{A}_j}(x_1, \ldots, x_n)$

- Can you do with less layers?
- Can actually do with two layers using a different construction. Sometimes people state that these kinds of results held back neural networks for two decades.

- Let's discuss one more aspect of neural networks before coming back to RL.
- Suppose you take the product of n numbers $x_1 \cdots x_n$. If all the numbers are at least 3/2, the answer will be lower bounded by 1.5^n which is huge. For example, $1.5^{100} \approx 4 \cdot 10^{17}$.
- On the other hand, if all the numbers are at most 2/3, then the result will be upper bounded by $(2/3)^n$ which is tiny. For example, $(2/3)^{100} \approx 2 \cdot 10^{-18}$.
- Likewise, if you generate numbers randomly with a mean above one, their product will blow up exponentially fast.
  If you generate numbers randomly with a mean below one, their product will decay to zero quickly.
  A possible exception occurs when the numbers are distributed around one.
- What if you multiply matrices instead of numbers?
- In practice, you get the same thing. Except in a few special cases, the result will either blow up or decay to zero. Can easily come up with exceptions...

  ...but in practice, if you have a product of n matrices, unless they have some kind of special structure, you can expect things to either blow up or go to zero.

- With this in mind let's look at our back propagation equations:

  $\delta_j^l = \frac{\partial C^i}{\partial z_j^l}$

  (1) $\frac{\partial C^i}{\partial z^L} = (a^L - y_i)\sigma'(z^L)$ tells us what δ_1^L is.

  (2) $\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l)$ tells us how to compute the vector δ^l from

  (3) $\frac{\partial C^i}{\partial w_{jk}^l} = = \delta_j^l a_k^{l-1}$

  (4) $\frac{\partial C^i}{\partial b_j^l} = \delta_k^l$

  Does anything strike you as interesting about equation (2)?

- Let's stack up the δ_j^l into a vector δ^l and the w_{kj}^{l+1} into the matrix w^{l+1}. Then we can write Eq. (2) as

  $\delta_j^l = \sigma'(z_j^l) \sum_k [(w^{l+1})^T]_{jk} \delta_k^{l+1}$

- In other words,

  $\delta_j^l = \sigma'(z_j^l)((w^{l+1})^T \delta^{l+1})_j$

- Or:

  $\delta^l = \sigma'(z_j^l) \odot (w^{l+1})^T \delta^{l+1}$

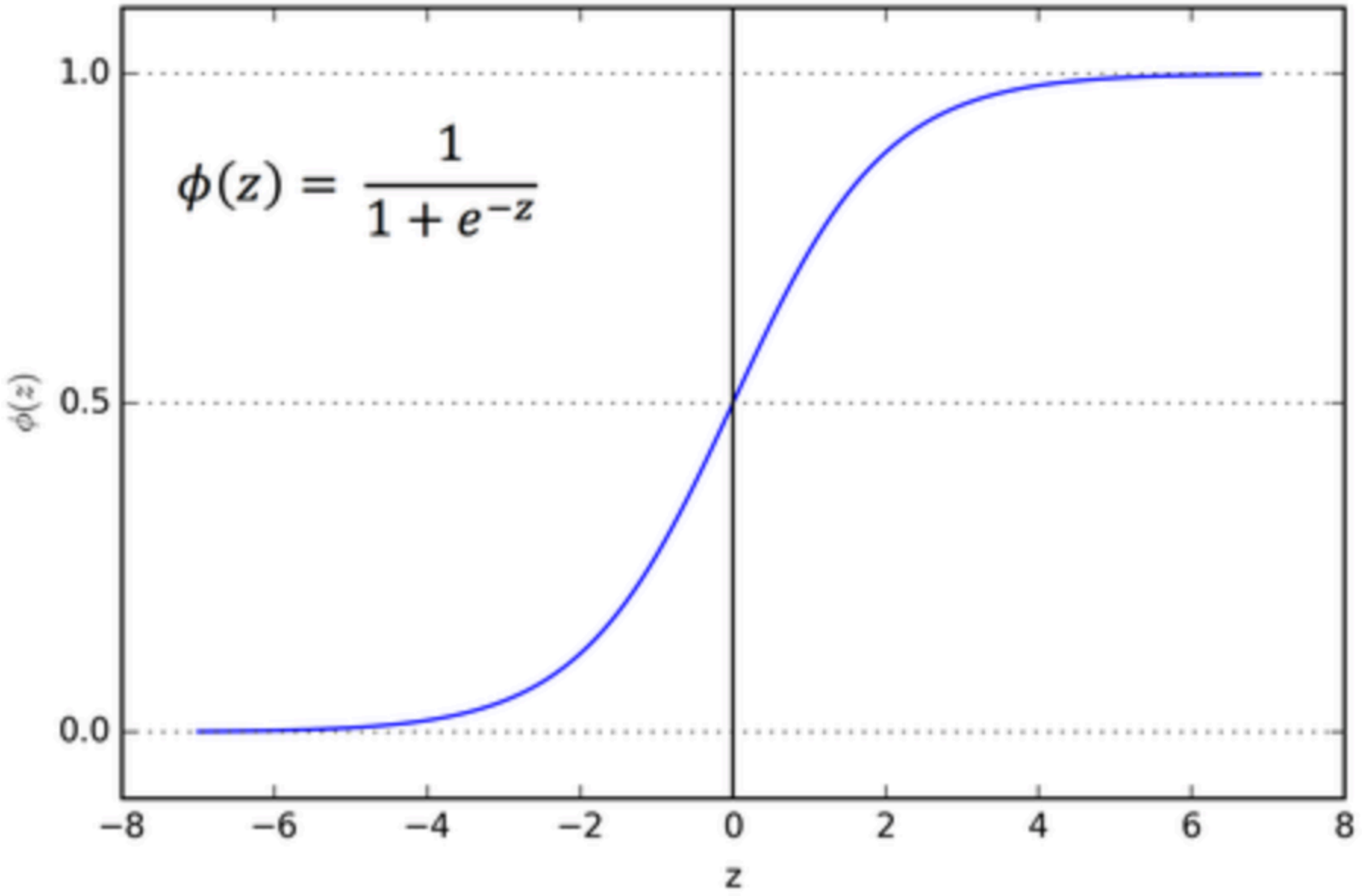
  where $\odot$ refers to the element-wise multiplication of vectors.

- Suppose we have no activation functions. In that case,

  $\delta^l = (w^{l+1})^T \delta^{l+1}$ and we have a product of matrices, i.e., $\delta^0 = (w^1)^T (w^2)^T \cdots (w^{L-1})^T \delta^l$.

- Exploding gradients are solved relatively easily: just scale down the gradient. ADAM does this automatically.
- Vanishing gradients are not so easily solved.
- Solution: try to initialize the weights so that at the first iteration, the product $(w^1)^T(w^2)^T\cdots(w^{L-1})^T$ does not blow up.
- ...then during training, the network has no incentive to either blow these up or shrink them: prediction becomes bad.
- ...helped by making the step-sizes small.
- But how to do the initialization?
- What works are variations on the following: initialize $w_{ij}^{l+1} = \sqrt{3}U\left[-\frac{1}{\sqrt{W^l}}, +\frac{1}{\sqrt{W^l}}\right]$

  where W^l is the width of the l'th layer (for a fully connected component.
- Key point: $W^l \mathrm{var}(w_{ij}^{l+1}) = 1.$
- **Claim:** Suppose x is a vector of independent random variables where every entry has zero mean and the same variance. Then any two entries of $w^{l+1}x$ have mean zero and are uncorrelated; moreover, if $y = w^{l+1}x$ then $E[||y||_2^2] = E[||x||_2^2]$.
- Proof: the first bit about having zero mean is obvious.
- As for as uncorrelatedness: $E[y_1y_2] = E\left[(\sum_i w_{1i}^{l+1}x_i)(\sum_i w_{2i}^{l+1}x_i)\right] = 0$ (why?)
- Now let's talk about variance. First, I claim that $\mathrm{var}(y_1) = W^l \mathrm{var}(w_{ij}^{l+1}x_j)$ (for any i, j). Why?
- Implication: $\mathrm{var}(y_1) = W^l E[(w_{ij}^{l+1})^2]E[x_j^2]$ for any i, j.
- So: $E[y_1^2] = E[x_j^2]$ for any j.
- Conclusion: $E[||y||_2^2] = E[||x||_2^2]$.
- This is called the Xavier Initialization: see http://proceedings.mlr.press/v9/glorot10a



What is the derivative of a sigmoid as you go to infinity or -infinity?

- Let's go back to:

  $\delta^l = \sigma'(z_j^l) \odot (w^{l+1})^T \delta^{l+1}$

- What we discussed earlier:

  — how to choose your initial w^{l+1} so that the product $w^1 \cdots w^L$ does not blow up.

  — but if you use sigmoid activation functions, then some of the components will get multiplied by zero.
- Solution: don't use sigmoid, except perhaps once at the end.
- Main activations should be ReLUs.
- Great, so Xavier Initalization + ReLU activations is what we do to solve vanishing gradients right?
- No! What's wrong?

- We derived Xavier initialization by analyzing $y = w^{l+1}x$.

  We argued that $E||y||_2^2 = E||x||_2^2$.
- Of course, this ignores the bias, but we will just initialize that to be zero.
- But actually, we have $y^{\text{correct}} = \sigma(w^{l+1}x)$ where σ is the activation function! So our derivation is invalid.
  - So suppose σ is ReLU.

    We will model what the lath layer of the neural network does as

    $$y^{l+1} = w^{l+1}x^l$$

    $$x^l = \max(y^l, 0).$$

    So here y^l is the output pre-activation function.

    We assume as before every entry is independent, identically distributed, with a distribution that is symmetric around zero.
  - Our goal is to have $E[||y^{l+1}||_2^2] = E[||y^l||_2^2]$.
  - Key point: now x^l no longer has zero mean.

- We consider $y^{l+1} = w^{l+1}x^l$

  $$x^l = \max(y^l, 0).$$

  Our goal is to have $E[||y^{l+1}||_2^2] = E[||y^l||_2^2]$.
- As before, we have that different entries of y^{l+1} have mean zero. Why?
- As before, we have that different entries of y^{l+1} are uncorrelated. Why?
- Due to the ReLu activation and the symmetry of y^l around zero, we have that $E[||x^l||_2^2] = \frac{1}{2}E||y_l||_2^2$. Why?
- Next, we have

  $$\begin{aligned}\text{var}(y_1^{l+1}) &= W^l \text{var}(w_{ij}^{l+1}x_j) \\ &= W^l E[(w_{ij}^{l+1})^2]E[(x_j)^l]^2 \\ &= W^l E[(w_{ij}^{l+1})^2]\frac{1}{2}E[(y_j^l)^2]\end{aligned}$$
- Conclusion: $\frac{1}{2}W^l E\left[(w_{ij}^{l+1})^2\right] = 1$. In other words, the variance should be doubled!!!!
- It is somewhat embarrassing that this factor of 1/2 was not discovered until 2015: https://arxiv.org/pdf/1502.01852v1.pdf
- This is called the He initalization.
- Makes a big difference in practice!

- OK, now let's come back to RL. The last thing we did was try to approximate the state by a linear function

  $V \approx \Phi\theta$

  or

  $V(s) = \sum_i \theta_i \phi_i(s)$

  which required knowing the features $\phi_i(s)$ in advance — but then we learned the weights θ over the course of a TD iteration.
- We will now try to approximate

  $V(s) \approx f_{NN,\theta}(s)$

  where $f_{NN,\theta}(\cdot)$ is a neural network with weights θ.

  s here will be fed into the input of the neural network.
  If the state is an image, then it will be flattened as before.
- Over the course of the algorithm, we will try to learn the best weights θ.
- If this works, it solves the problem of having to hard-code features.

- Let's take this idea and make it more concrete.
- Let's come back to the TD setting. We have an MDP and a policy π. Our goal is to compute V_π.
- Two constraints:
  — we don't know the MDP and can only interact with it through experience
  — the state is very high-dimensional and so we want to keep track of a low-dimensional approximation.
- We define the function V_θ by fixing some NN architecture that will take the state as input and output a real number, and we initialize all the weights θ randomly.
- Now we run the method. We start at some state s. We take action a by sampling from $\pi(s)$. We obtain reward r and move to state s'.
- We valuate $V_{\text{target}} = V_\theta(s')$. We define $l(\theta) = \frac{1}{2}(r + \gamma V_{\text{target}} - V_\theta(s))^2$.

  We update

  $$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta l(\theta)$$

  and, for the purpose of computing the gradient, we treat V_{target} as a constant.
- This is $\theta_{t+1} = \theta_t - \alpha_t(r + \gamma V_{\text{target}} - V_\theta(s)) \nabla_\theta V_\theta(s)$.

  We need to do a back propagation to evaluate $\nabla_\theta V_\theta(s)$ and two evaluations to evaluate V_{target} and $V_\theta(s)$.
- This is called **Neural TD.**

- Let's take this idea and make it more concrete.
- Let's come back to the TD setting. We have an MDP and a policy π. Our goal is to compute V_π.
- Two constraints:
  — we don't know the MDP and can only interact with it through experience
  — the state is very high-dimensional and so we want to keep track of a low-dimensional approximation.
- We define the function V_θ by fixing some NN architecture that will take the state as input and output a real number, and we initialize all the weights θ randomly.
- Now we run the method. We start at some state s. We take action a by sampling from $\pi(s)$. We obtain reward r and move to state s'.
- We valuate $V_{\text{target}} = V_\theta(s')$. We define $l_t(\theta) = \frac{1}{2}(r + \gamma V_{\text{target}} - V_\theta(s))^2$.

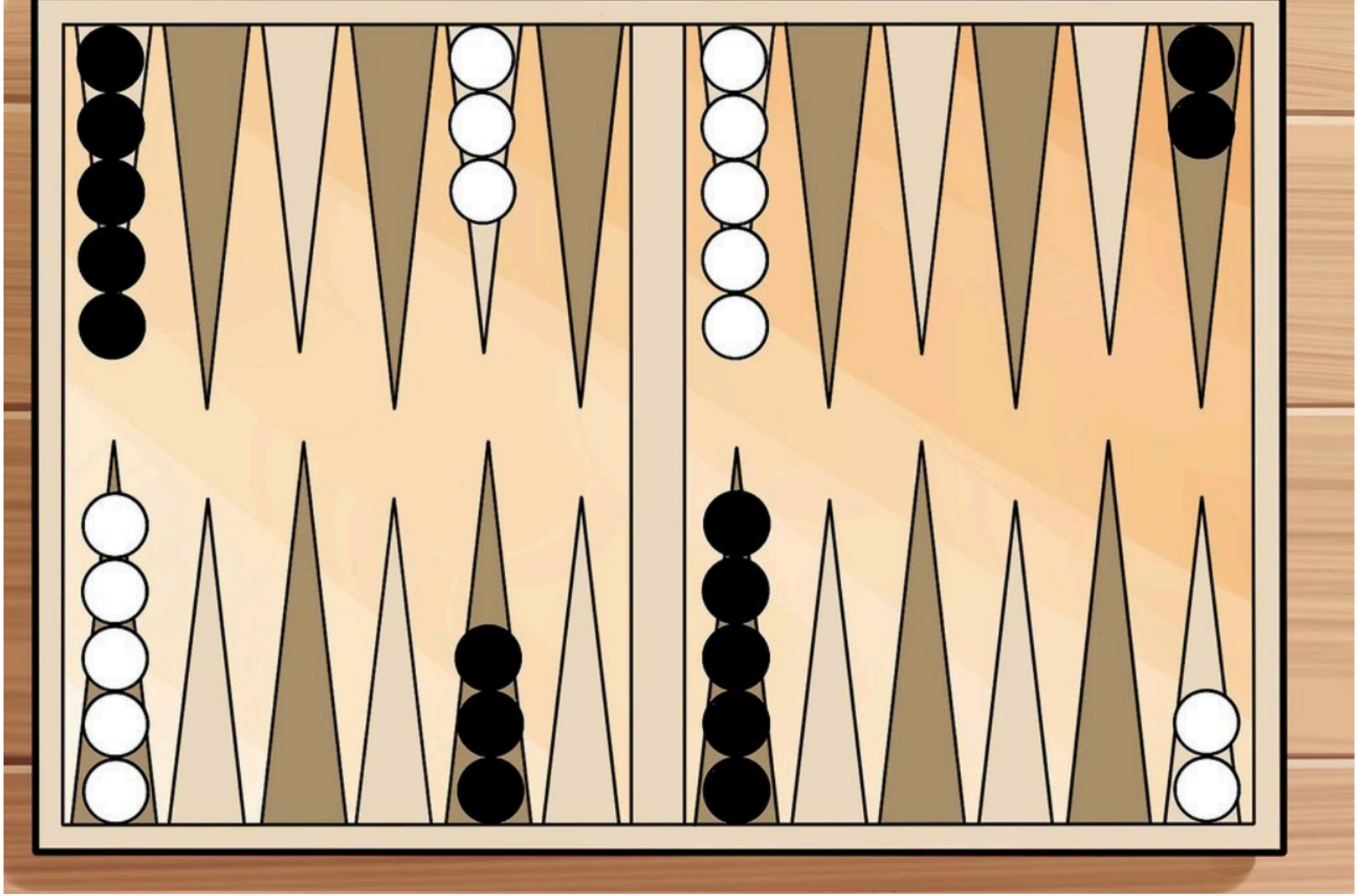
  We update

  $\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta l_t(\theta)$

  and, for the purpose of computing the gradient, we treat V_{target} as a constant.
- This is $\theta_{t+1} = \theta_t + \alpha_t(r + \gamma V_{\text{target}} - V_\theta(s))\nabla_\theta V_\theta(s)$.

  We need to do a back propagation to evaluate $\nabla_\theta V_\theta(s)$ and two evaluations to evaluate V_{target} and $V_\theta(s)$.
- This is called **Neural TD.**

- The first major success for this came in the late 1980s, when Gerald Tesauro used it to develop a computer program that played backgammon.
- In the 1980s, Tesauro’s program was rated as the top computer program. W
- By the mid 1990s, it was playing about as well as the best human players.
- By mid 2000s, it was so much better that the best human players that most players were trying to emulate it’s style.



Each player rolls a dice and can make a move that depends on the outcome of that dice
The winner is the first person to move pieces across the board

- Tesauro's method: set reward equals 1 if black wins and zero if white wins.
- The train a NN to predict the value of a position.
- Input: how many blocks at each location, and whether they are white and black (with some clever tricks, he got this to 16 numbers).
- The network: one hidden layer, about 80 hidden units.
- He then used TD learning to try to learn the value of each position with $\gamma = 1$.
- Interpretation: value is the probability that black wins.
- To turn that into a program that plays moves is easy: simply go through all the moves and select the ones with the highest value.

- This was before the era of multi-layer, deep, networks, so it was natural to train a NN with only one hidden layer.
  Also, think about how slow computers were back in the late 80s....
- What I outlined ended up doing OK, it played like an intermediate player. This was the late 80s.
- Tesauro improved it in several ways:
  — use six networks depending on the stage of the game (beginning, early, middle, etc).
  — but all six networks had a single layer
- Second, he began using a few hard-coded features which he fed into the input of the network (e.g., how close each side's pieces are to the end).

- This was before the era of multi-layer, deep, networks, so it was natural to train a NN with only one hidden layer.
  Also, think about how slow computers were back in the late 80s....
- What I outlined ended up doing OK, it played like an intermediate player. This was the late 80s.
- Tesauro improved it in several ways:
  — use six networks depending on the stage of the game (beginning, early, middle, etc).
  — but all six networks had a single layer
- Second, he began using a few hard-coded features which he fed into the input of the network (e.g., how close each side's pieces are to the end).

- Next step: use a little look-ahead.
- Go through all the moves, select, say, the top 5 or 6 the NN likes.
- For each of the moves, pretend you've made it, go through every dice roll your opponent makes, and look at all of their moves.
- Evaluate the position after the best moves of the opponent for every dice roll.
- To estimate the value of the position for your opponent, you want to average the result over all the dice rolls.
- This is called 2-ply search.
- The version that was superior to humans actually used 3-ply search.

Opinion: Have Computers Ruined Backgammon?

16th September 2015 By Julia Hayward

John Hurst writes:

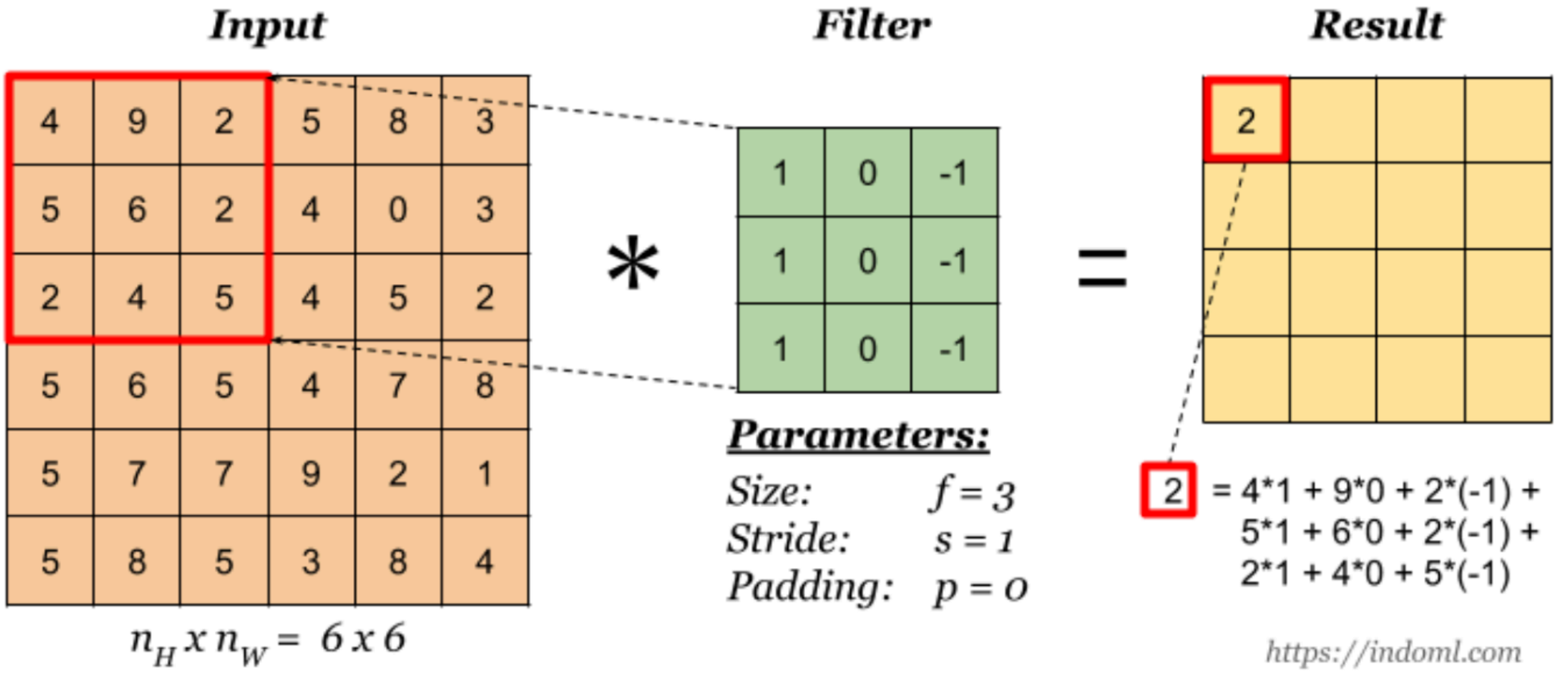
Ex World Champion Mike Sovodny once said that the emergence of computer programs such as Snowie had, for him, to some extent lessened the appeal of backgammon. Certainly playing props, at which he was an expert, is now pointless as all you need to do to find the correct move or doubling decision is to stick the position in the computer.

"It says that two on the bar is better by far, Professor"

In the past even experienced players would differ wildly in their opinions of the right move and nobody could be sure who was correct; that uncertainty made the game intriguing. Some enthusiasts would spend countless hours rolling out positions to find the right answer and presumably that work would be rewarded by their improved results. Now all a player has to do is ask Snowie or XG. Different players would develop different playing styles, now all players who take the game seriously try to ape the computer as closely as possible.There is no point in trying to develop your own strategies, if your move differs from the computer then it is wrong – it's as simple as that.

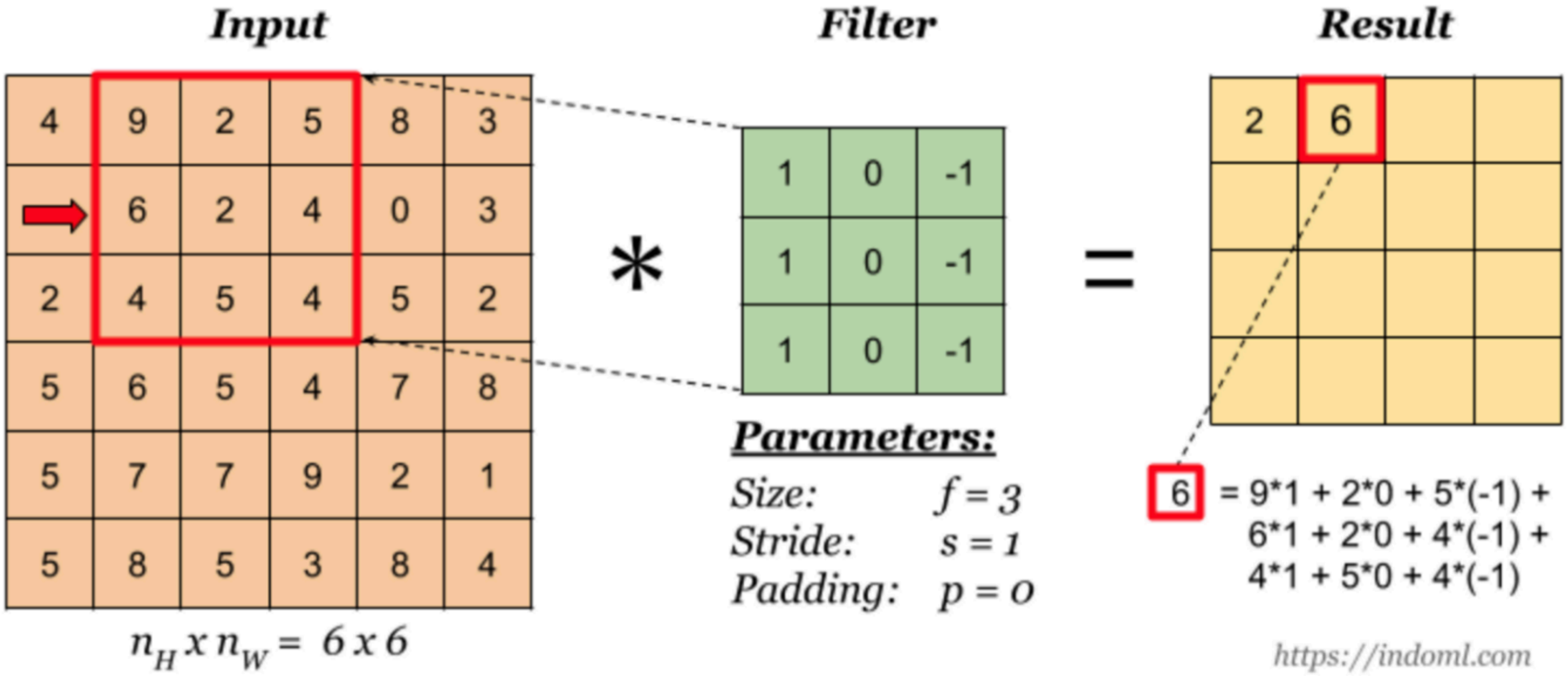
- The success of TD Gammon sparked a lot of enthusiasm, but it did not really lead to success in other games (chess, go, or computer games).
- One reason why: this needed a lot of computational power which nobody really had back then.
- But there are also other pieces of the puzzle — let's talk about them.
- What is the main problem with the way we attempted to do image recognition on MNIST?
- Images are not vectors. By flattening them into vectors, we seem to be losing key information.
- Sure, a neural network could "learn" over the training period that entry 23 and 458 of the vector code for adjacent pixels, and so they will be quite similar.
- But maybe we should give this sort of information to the NN.
- Solution: we need to a NN that works on 2D arrays.

- Further: image recognition is “shift invariant.”
  If you take a picture of a 3, shift it to the right, it’s still recognizable as a 3.
- ...but a neural network as we have described it may not be shift invariant.
- Counterpoint: if you give it enough 3s (and other digits) shifted all over the place, it will learn to be shift invariant.
- In practice: not quite. Part of the problem is that it is not clear what, in the end, the training process converges to — the problem of local vs global minima.
- Better approach: code shift invariance INTO the arhitecture.



Convolutional operation.

We take the "inner product" by multiplying the corresponding elements and adding to get the two

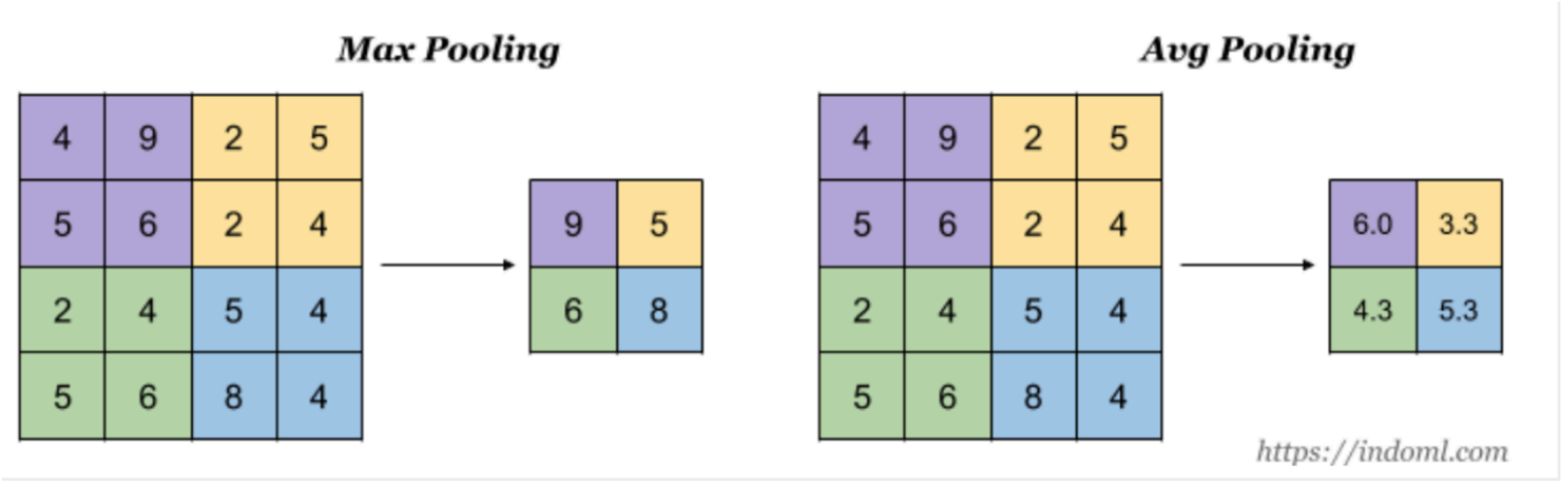


Stride Length = 1

We translate the input by one unit to the right and redo the operation
We'll then translate it one unit to the right and redo
And then one more unit to the right and redo
Then we'll reset to the starting position and move one unit DOWN
This is called a convolutional layer

- A convolutional layer is just taking linear combination of the pixel values...

  ....except it is constrained to take linear combinations of pixels that are close.
- As indicated earlier, if we flattened the image into vectors and added all to all connections, in principle the NN could reproduce a convolutional layer.
- But this way there is a “bias” towards the things it learns, and that bias makes sense.
- Convolutional layers are usually followed by pooling layers.



Pooling layers

- A convolutional layer is shift invariant.
- Indeed, suppose you shift the image i to right by one pixel to obtain i_s.

  If o is the output of i through a convolutional layer, then the output of i_s will be

  o shifted to the right by one pixel.

  [here we are ignoring border effects — imagine the imagine goes on to infinity in each direciton].
- The usual architecture for images

  — convolutional layer followed by pooling layer

  — after these are stacked up, they are followed by one or more fully connected layers.
- Let's try it out: With this in mind, let us try this out: https://colab.research.google.com/drive/1AdeopLi_9trH-amct5kpq9k0KJkEoGew?usp=sharing

- Let's come back to RL again. We've talked about using TD learning with a neural network. Why not Q-learning?
- Let's think about how this works. We fix an architecture and at each step of the algorithm we update the weight θ such that $Q_\theta(s, a)$ is our approximation to the optimal value function.
- Here s and a are the input to the neural network (we can stack up). The θ is the weight of the neural network.
- So we are at state s, we choose action a, obtain a reward r, and transition to s'.

  Define $V_{\text{target}} = \max_{a'} Q_\theta(s', a')$ and $l_t(\theta) = \frac{1}{2}(r + \gamma V_{\text{target}} - Q_\theta(s, a))^2$.
- We update $\theta_{t+1} = \theta_t - \alpha_t \frac{1}{2} \nabla l_t(\theta)$ where in the computation of the gradient we treat V_{target} as a scalar.
- That is $\theta_{t+1} = \theta_t + \alpha_t (r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)) \nabla_\theta Q_\theta(s, a)$
- This is called **Deep Q-learning**. Somehow, it's called that even when the architecture you choose is not all that deep...
- Deep Q-learning + convolutional neural networks are responsible for some of the more recent successes of RL.

- One way to improve this is to observe that there's no reason to generate new states all the time.
- Instead, every time you encounter (s, a, r, s') add it to an "experience replay buffer."
- Then, at each step, you could either update

$$\theta_{t+1} = \theta_t + \alpha_t(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)) \nabla_\theta Q_\theta(s, a)$$

or you could also sample a random batch of size b from the buffer and and update as

$$\theta_{t+1} = \theta_t + \alpha_t \frac{1}{b} \sum_{s,a,s',r \text{ in batch}} (r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)) \nabla_\theta Q_\theta(s, a)$$

- Experience replay significantly improves performance in practice.

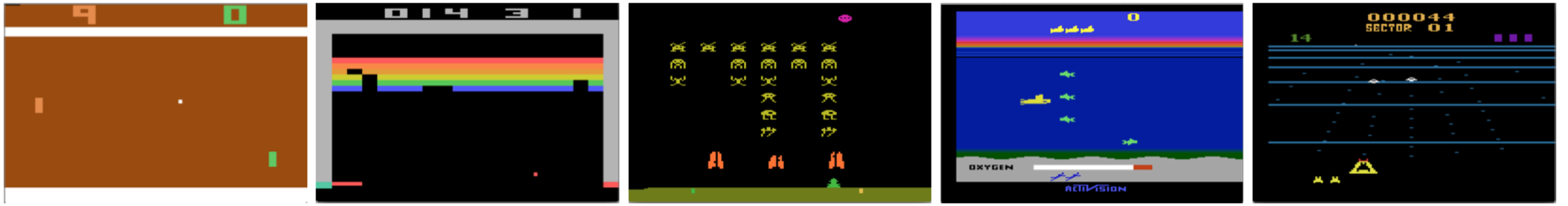


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

The 2013 paper of Mnih et al. showed really good performance for DQN

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

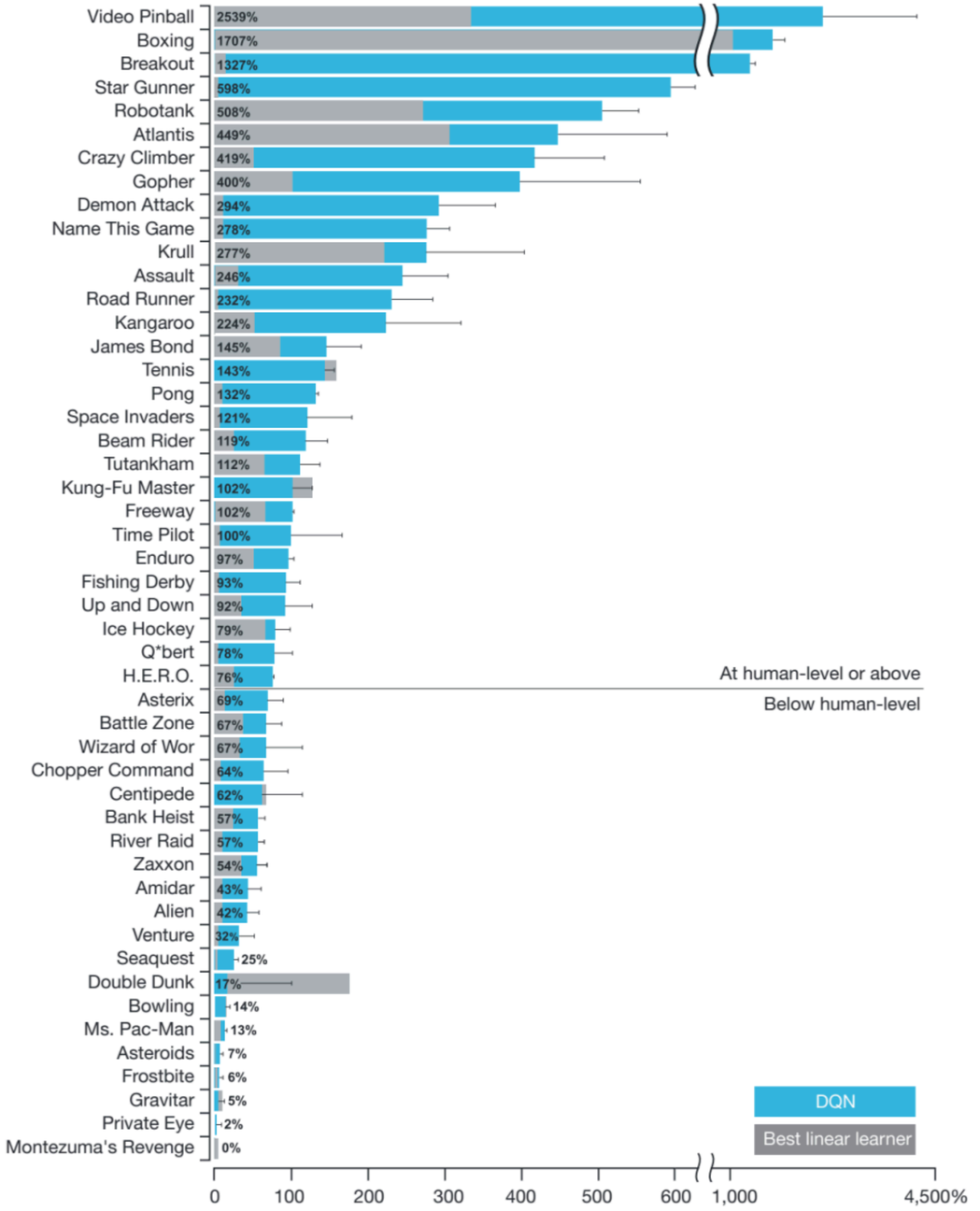
Their main result

- How they did it: first step is to pre-process the data (very important in practice).
  They take the images, make them grayscale, downscale them, and crop to show the “playing area” only.
  In the end, a 210x160 image is reduced to 84x84.
- They stack up the last four frames (after pre-processing) into the state. So in this case the state has dimension $4 \cdot 84^2$.
- Two convolutional layers (16 and 64 filters), followed by a fully connected layer (512 neurons), all with ReLu activations.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

The full paper may be looked at https://arxiv.org/abs/1312.5602



Two years later the same group was able to get more impressive results

- To explain how they did, let's talk about a core problem of this method: DQN consistently over-estimates true Q-values.
- Suppose you take a simple RL problem where the state space is sufficiently small that you can solve it exactly.

  You find the true Q-values $Q^*(s, a)$.
- You then solve it with DQN. You have your final Q-values $Q_{\theta*}(s, a)$ where $\theta*$ is the set of weights when you stop your method.
- Empirical finding: you will typically have $Q_{\theta*}(s, a) > Q^*(s, a)$.
- Think about why this is....

- Let's take a simple and wrong model.
  When we use DQN, there will be errors.
  Let's assume these errors are Gaussian and independent across nodes.
- That is, $Q_{\theta^*}(s,a) = Q^*(s,a) + w_{sa}$ where the random variables w_{sa} are all unit normal and independent.
- Again, this is clearly wrong....but let's go with it.
- Do you see why $r + \max_{a'} Q_{\theta^*}(s',a')$ will tend to over-estimate $r + \gamma \max_{a'} Q^*(s',a')$?
- Suppose there are five actions with roughly equal value, i.e.,
  $Q^*(s,a_1) = Q^*(s,a_2) = \cdots = Q^*(s,a_5).$
  If we add random noise to its, and THEN pick the largest, this will tend to over-estimate $\max_{a_1,\ldots,a_5} Q(s,a_i)$

- We can crystallize the issue by observing that our estimate of the value of the next state is

  $$V_{\text{target}} = r + \gamma Q_{\theta_t}(s', \arg\max_{a'} Q_{\theta_t}(s', a'))$$

- Solution: use two different networks!
- Ideally one might train to train $Q^1(\theta_t^1)$ and $Q^2(\theta_t^2)$ starting from independent random initial conditions and estimate the target as

  $$V_{\text{target}}^1 = r + \gamma Q_{\theta_t^1}(s', \arg\max_{a'} Q_{\theta_t^2}(s', a'))$$

- In practice: two time consuming. Usually, you fix some large K and you use $Q_{\theta_{t-K}}$ as your second network.
- Another possibility: Start with a random θ_0^2 for your second network and update

  $$\theta_{t+1}^2 = \tau\theta_{t+1}^1 + (1-\tau)\theta_t^2.$$

- People call this double Q-learning.
- Popular belief: target network strongly significantly improves performance.

- Worth reflecting on how RL is more difficult that supervised learning (e.g., image labeling).
- In supervised learning, you typically have labels.
  In RL, you use your method to generate your labels.
- In other words, your labels change as you run your method.
- This can lead to instability and divergence.
- Again, can think about interpreting the world through the lease of your beliefs.
- No good fix for this. Sometimes things work, sometimes they don’t.
- Example: https://www.youtube.com/watch?v=fv-oFPAqSZ4

- Let's talk about **Continuous Problems**. So your state space is a vector and your action space is a vector.
- No real problem in implementing the Q-value iteration

  $$\theta_{t+1} = \theta_t + \alpha_t(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)) \nabla_\theta Q_\theta(s, a)$$

  (of course, you have to fix an architecture first).
- What's the problem?
- Solution: use a neural network to compute the max.
- We will have two neural networks now. One neural networks will we referred to as $Q_{s_q}(s, a)$ (called the critic network) and will be update as above.

  The second neural network $\mu_{\theta_\mu}(s)$ (called the actor network) will decide which action to choose.
- How should we update θ_μ?
- Natural choice: $\theta_\mu(t+1) = \theta_\mu(t) + \beta(t) \nabla_{\theta_u} Q_{\theta_q}(s, \mu_{\theta_\mu}(s))$
- Can use chain rule: $\nabla_{\theta_\mu} Q_{\theta_q}(s, \mu_{\theta_\mu}(s)) = \nabla_a Q_{\theta_q}(s, a) \nabla_{\theta_\mu} \mu_{\theta_\mu}(s)$

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R

Receive initial observation state s_1
for t = 1, T **do**
Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
Execute action a_t and observe reward r_t and observe new state s_{t+1}
Store transition (s_t, a_t, r_t, s_{t+1}) in R
Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

end for
end for

The DDPG method
https://arxiv.org/pdf/1509.02971.pdf

- When it came out, DDPG was state of the art in a variety of robotics tasks: https://www.youtube.com/watch?v=tJBIqkC1wWM
- https://static-content.springer.com/esm/art%3A10.1038%2Fnature14236/MediaObjects/41586_2015_BFnature14236_MOESM124_ESM.mov
- Can still get OK results with it: https://www.youtube.com/watch?v=rd2pM43zzTo
- Many method since that (claim to) obtain better performance.
- Nice blog post: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html