Lecture 5: Softmax, gradient descent, and neural networks

Announcements:

- HW #2 is due, Monday Jan 24, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

- Moving forward, the assignments have a good amount of Python coding. Get started early! This assignment will cover k-nearest neighbors and the softmax classifier.

- I received feedback last week that we are taking too many questions in class which is impeding the pace of lecture. I've spoken with the TAs so we can improve question answering in the chat. If your question is answerable over chat, we ask you first please ask there. I will still take questions in class, but may limit if we've had several questions on a topic that isn't of high importance.

Classifiers based on linear classification

Example 2: Consider a matrix, $\mathbf{W}$, defined as:

$$\begin{bmatrix} - \mathbf{w}_1^T - \\ \vdots \\ - \mathbf{w}_c^T - \end{bmatrix}$$

Then, $\mathbf{W} \in \mathbb{R}^{c \times n}$. Let $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{b}$ is a vector of bias terms. Then $\mathbf{y} \in \mathbb{R}^c$ is a vector of scores, with its ith element corresponding to the score of $\mathbf{x}$ being in class i. The chosen class corresponds to the index of the highest score in $\mathbf{y}$.

$x \in \mathbb{R}^n$

$y \in \{1, 2, \ldots, 10\}$

score of class 1

$$\mathbf{y} = \begin{bmatrix} w_1^T x + b_1 \\ w_2^T x + b_2 \\ \vdots \\ w_c^T x + b_c \end{bmatrix} = \begin{bmatrix} — \; w_1^T \; — \\ — \; w_2^T \; — \\ \vdots \\ — \; w_c^T \; — \end{bmatrix} \begin{bmatrix} x \end{bmatrix} + \begin{bmatrix} b \end{bmatrix}$$

10×1 $\quad$ 10×3072 $\quad$ 3072×1 $\quad$ 10×1

score of class c

CIFAR-10

$x \in \mathbb{R}^{3072}$

$y \in \mathbb{R}^{10}$ – correspond to scores for 10 classes

$W \in \mathbb{R}^{10 \times 3072}$

$b \in \mathbb{R}^{10}$

$x^{(i)} \longrightarrow \hat{y}^{(i)} = \begin{bmatrix} 300 \\ 500 \\ 150 \\ \vdots \\ 700 \end{bmatrix}$ $\quad$ $y^{(i)} = 2$

$(2-1)^2 = 1$

$(2-10)^2 = 64$

Classifiers based on linear classification

Each row of **W** can be thought of as a template.

Turn the scores into a probability

A first thought is to turn the scores into probabilities.

Softmax function

There are several instances when the scores should be normalized. This occurs, for example, in instances where the scores should be interpreted as probabilities. In this scenario, it is appropriate to apply the *softmax* function to the scores.

The softmax function transforms the class score, $\text{softmax}_i(\mathbf{x})$, so that:

$$\text{softmax}_i(\mathbf{x}) = \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}}$$

$a_i(x) = y_i = w_i^T x + b_i$

$0 \leq \text{softmax}_i(x) \leq 1$

for $a_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + b_i$ and c being the number of classes.

$a_1(x) = w_1^T x$ $\quad$ $a_2(x) = w_2^T x$

$$\text{softmax}_1(x) = \frac{e^{w_1^T x}}{e^{w_1^T x} + e^{w_2^T x}} \qquad \text{softmax}_2(x) = \frac{e^{w_2^T x}}{e^{w_1^T x} + e^{w_2^T x}}$$

Softmax classifier

$$\text{softmax}_i(\mathbf{x}) = \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}}$$

for $a_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + w_i$ and c being the number of classes.

If we let $\theta = \{\mathbf{w}_j, w_j\}_{j=1,\ldots,c}$, then $\text{softmax}_i(\mathbf{x})$ can be interpreted as the probability that $\mathbf{x}$ belongs to class i. That is,

$$\Pr(y^{(j)} = i | \mathbf{x}^{(j)}, \theta) = \text{softmax}_i(\mathbf{x}^{(j)})$$

↓

Prob. that data point $x^{(j)}$ comes from class i.

Softmax classifier

Although we know the softmax function, how do we specify the *objective* to be optimized with respect to θ?

One intuitive heuristic is that we should choose the parameters, θ, so as to maximize the likelihood of having seen the data. Assuming the samples, $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$ are iid, this corresponds to maximizing:

image → dog, image → image, cat

$$p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}, y^{(1)}, \ldots, y^{(m)}|\theta) = \prod_{i=1}^{m} p(\mathbf{x}^{(i)}, y^{(i)}|\theta)$$

$$= \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta) p(y^{(i)}|\mathbf{x}^{(i)}, \theta)$$

image

→ softmax

Softmax classifier

$\theta = \{W, b\}$

$$\arg\max_{\theta} \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta) p(y^{(i)}|\mathbf{x}^{(i)}, \theta) \quad = \quad \arg\max_{\theta} \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}, \theta)$$

↳ Distribution of input images are independent of the model parameters.

$p(x_i|\theta) = p(x_i)$

$$= \arg\max_{\theta} \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}, \theta)$$

$$= \arg\max_{\theta} \sum_{i=1}^{m} \log \text{softmax}_{y^{(i)}}(x^{(i)})$$

$$= \arg\max_{\theta} \sum_{i=1}^{m} \log \left[\frac{e^{a_{y^{(i)}}(x^{(i)})}}{\sum_{j=1}^{c} e^{a_j(x^{(i)})}} \right]$$

$$= \arg\max_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left[a_{y^{(i)}}(x^{(i)}) - \log \sum_{j=1}^{c} e^{a_j(x^{(i)})} \right]$$

↑ Normalizing

$$= \arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left[\log \sum_{j=1}^{c} e^{a_j(x^{(i)})} - a_{y^{(i)}}(x^{(i)}) \right]$$

note: $a_j(x^{(i)}) = w_j^T x^{(i)} + b_j$

Example 1 : dog, $y^{(i)} = 1$

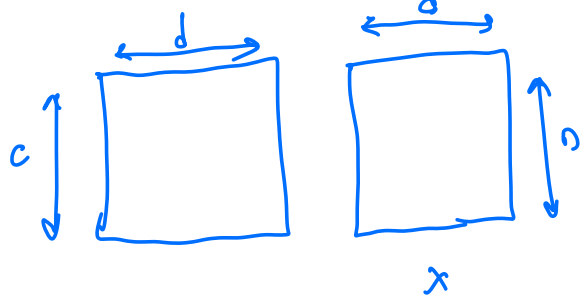
Example 2 : airplane, $y^{(i)} = 5$

Example 3 : cat, $y^{(i)} = 3$

Example 4 : dog, $y^{(i)} = 1$

$$\frac{e^{w_1^T x^{(i)} + b_1}}{\sum_{j=1}^{m} e^{w_j^T x^{(i)} + b_j}} //$$

$w \rightarrow c \times d$



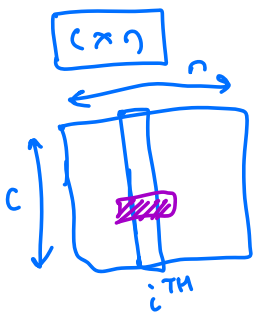
$y = wx$

$y:$

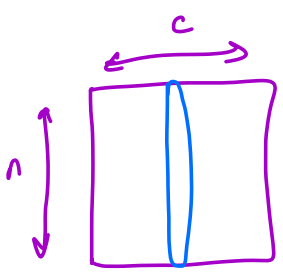
Homework:

$w\ x^T$

$c \times d \quad d \times n$



(x_i)



0 - col

1 → row

Softmax classifier

Now we have our softmax loss function.

$$\arg\min_{\theta} \sum_{i=1}^{m} \left(\log \sum_{j=1}^{c} e^{a_j(\mathbf{x})} - a_{y^{(i)}}(\mathbf{x}^{(i)}) \right)$$

Note, we haven't figured out yet how to get the optimal parameters.

(That'll be later.)

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2

Score check: $-a_{y^{(i)}}(\mathbf{x}^{(i)}) + \log \sum_{j=1}^{c} \exp(a_j(\mathbf{x}^{(i)}))$

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2

Cat: -2.1 + log(exp(2.1) + exp(3.4) + exp(-2.0)) = 1.54

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	1.54		

Car: -5.1 + log(exp(0.2) + exp(5.1) + exp(1.7)) = 0.04

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	1.54	0.04	

Bird: 1.2 + log(exp(2.3) + exp(3.1) + exp(-1.2)) = 4.68

Softmax classifier

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	1.54	0.04	4.68

$$-a_{y^{(i)}}(\mathbf{x}^{(i)}) + \log \sum_{j=1}^{c} \exp(a_j(\mathbf{x}^{(i)}))$$

Softmax classifier

A few additional notes on the softmax classifier:

Softmax classifier: intuition

When optimizing likelihoods, we typically work with the "log likelihood." When applying the softmax, we interpret its output as the probability of a class.

$$\begin{aligned} \log \Pr(y = i|\mathbf{x}) &= \log \text{softmax}_i(\mathbf{x}) \\ &= a_i(\mathbf{x}) - \log \sum_{j=1}^{c} \exp(a_j(\mathbf{x})) \end{aligned}$$

When maximizing this, the term $a_i(\mathbf{x})$ is made larger, and the term $\log \sum_j \exp(a_j(\mathbf{x}))$ is made smaller. The latter term can be approximated by $\max_j a_j(\mathbf{x})$. (Why?)

We consider two scenarios:

- If $a_i(\mathbf{x})$ produces the largest score, then the log likelihood is approximately 0.
- If $a_j(\mathbf{x})$ produces the largest score for $j \neq i$, then $a_i(\mathbf{x}) - a_j(\mathbf{x})$ is negative, and thus the log likelihood is negative.

Softmax classifier

A few additional notes on the softmax classifier:

Overflow of softmax

A potential problem when implementing a softmax classifier is overflow.

- If $a_i(\mathbf{x}) \gg 0$, then $e^{a_i(\mathbf{x})}$ may be very large, and numerically overflow and / or result to numerical impression.
- Thus, it is standard practice to normalize the softmax function as follows:

$$\begin{aligned}
\text{softmax}_i(\mathbf{x}) &= \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}} \\
&= \frac{ke^{a_i(\mathbf{x})}}{k\sum_{j=1}^{c} e^{a_j(\mathbf{x})}} \\
&= \frac{e^{a_i(\mathbf{x})+\log k}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})+\log k}}
\end{aligned}$$

- A sensible choice of k is so that $\log k = -\max_i a_i(\mathbf{x})$, making the maximal argument of the exponent 0.

Support Vector Machine

In prior years, we also taught on the support vector machine (SVM) and the hinge loss. Since most modern neural networks today only use a softmax classifier, we have decided to remove this material, and you will not be tested on SVMs. We have kept these slides in as extra resources.

Support vector machine

NOT TESTED

Before getting to parameter fitting, we'll want to introduce one more classifier that is commonly used: the support vector machine.

Support vector machine: introduction

The SVM is a commonly used and has much theory behind it. A typical machine learning class will formulate the SVM as a convex optimization problem. However, this is beyond the scope of this class.

Instead, we'll talk about the SVM at a very high-level using a soft-margin "hinge loss" and provide appropriate intuitions. We'll focus on linear SVMs and will *not* touch on kernels. Please look into a machine learning class for more information about the SVM.

Support vector machine: introduction

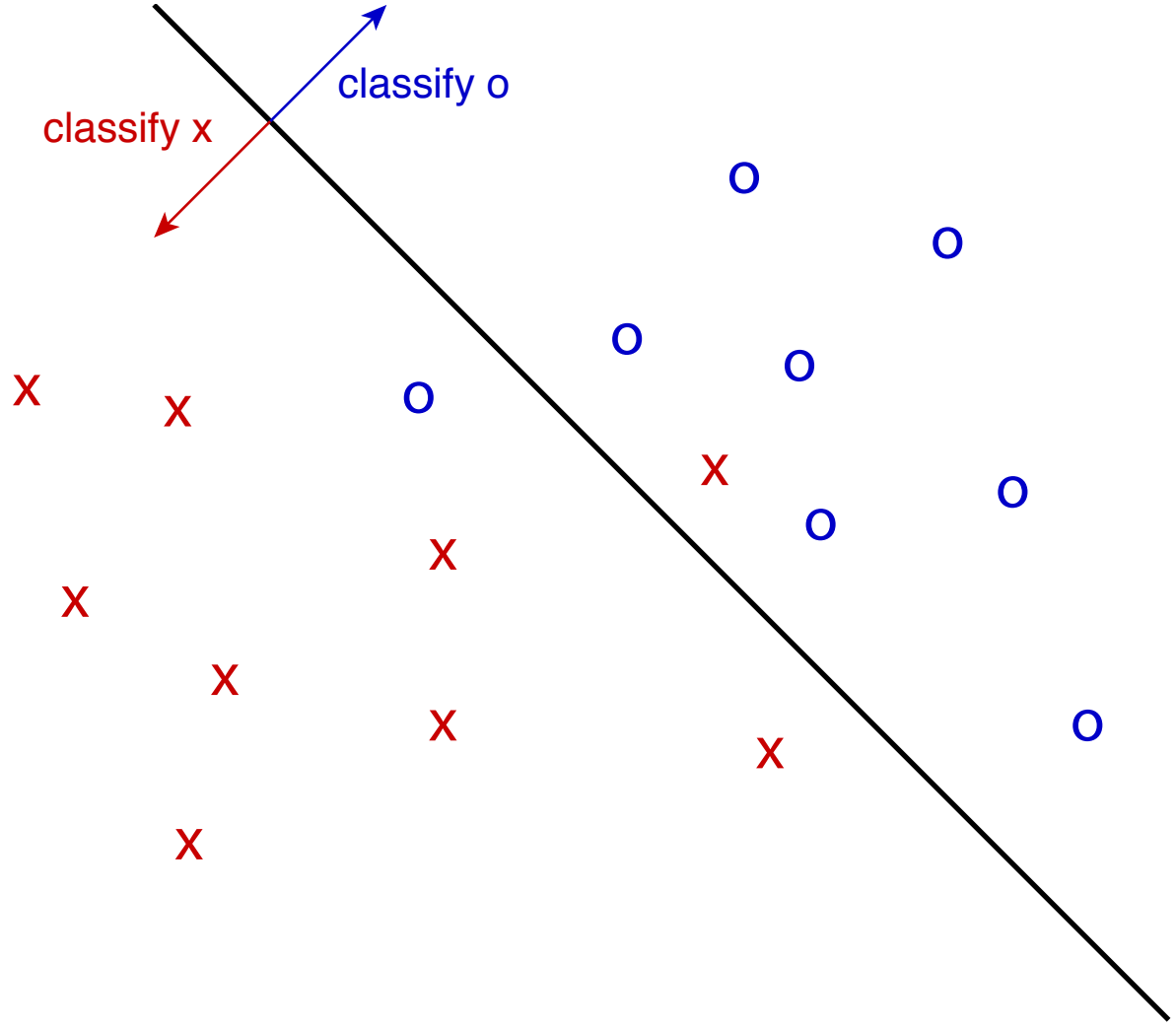
Another common decision boundary classifier is the support vector machine (SVM).

Informally, the SVM finds a boundary that maximizes the *margin*, or intuitively the "gap" between the boundary and the data points. The fundamental idea here is that if a point is further away from the decision boundary, there ought to be greater *confidence* in classifying that point.

Support vector machine

NOT TESTED

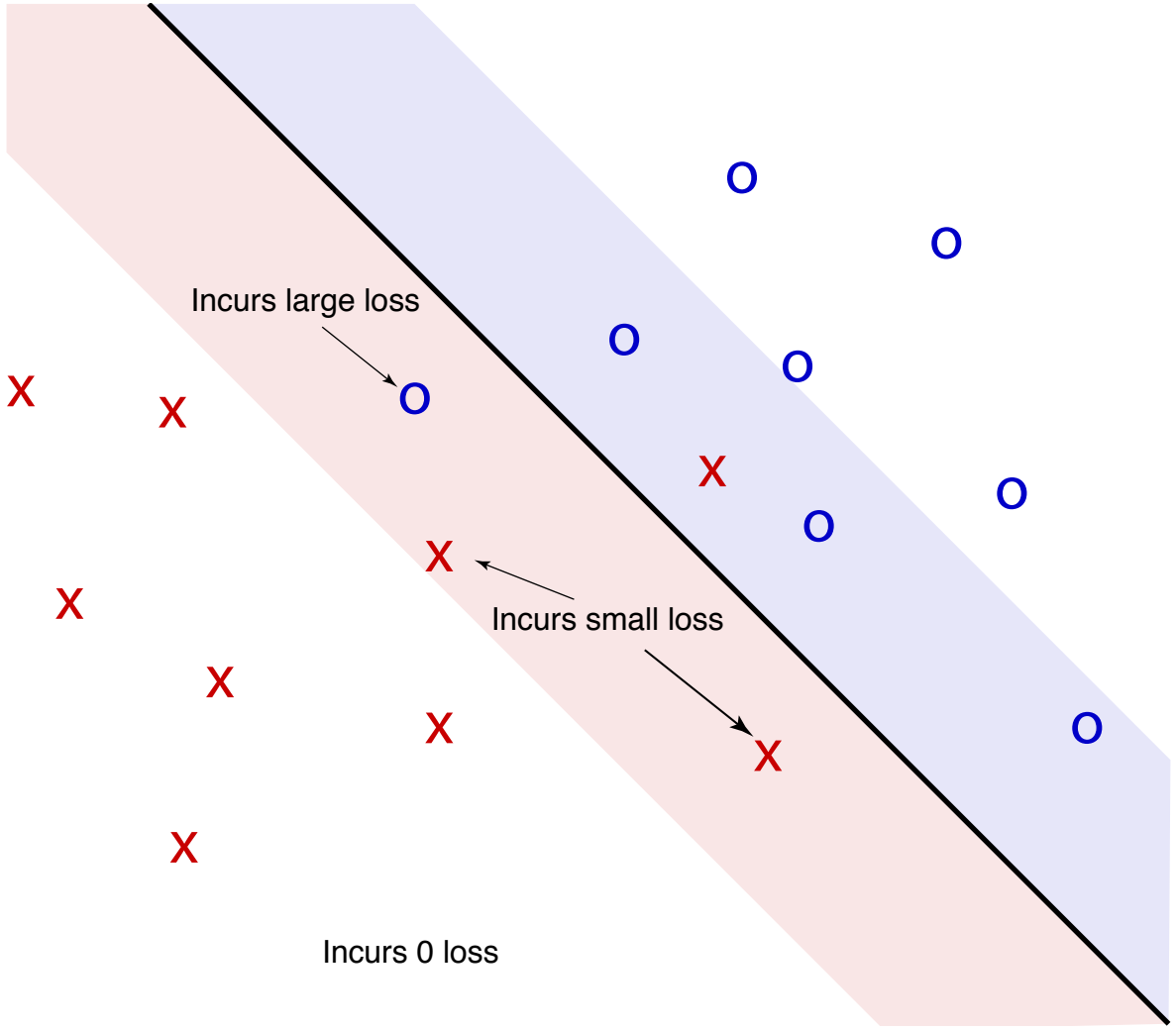
This is the picture we should have in mind:



Support vector machine

NOT TESTED

This is the picture we should have in mind:



The hinge loss function

The hinge loss is standardly defined for a binary output $y^{(i)} \in \{-1, 1\}$. If $y^{(i)} = 1$, then we would like $\mathbf{w}^T\mathbf{x}^{(i)} + b$ to be large and positive. If $y^{(i)} = -1$, then we would like $\mathbf{w}^T\mathbf{x}^{(i)} + b$ to be large and negative. The larger $a_i(\mathbf{x}^{(j)})$ is in the right direction, the larger the margin.

In this scenario, the hinge loss is for $x^{(i)}$ being in class $y^{(i)}$ is:

$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \max(0, 1 - y^{(j)}(\mathbf{w}^T\mathbf{x}^{(i)} + b))$$

Support vector machine

NOT TESTED

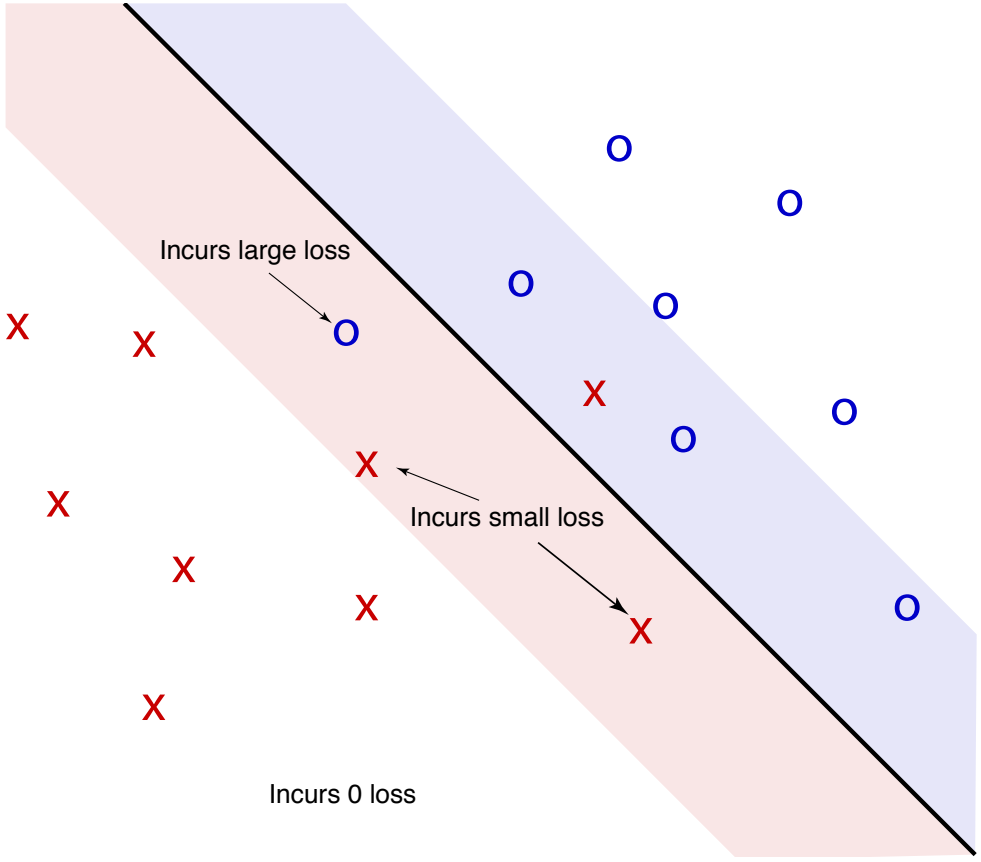
$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \max(0, 1 - y^{(j)}(\mathbf{w}^T\mathbf{x}^{(i)} + b))$$

This is a loss, and hence something we wish to minimize. There are a few things to notice about the form of this function.

- If $\mathbf{w}^T\mathbf{x}^{(i)} + b$ and $y^{(i)}$ have the same sign, indicating a correct classification, then $0 \leq \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) \leq 1$.
  - The error will be zero if $\mathbf{w}^T\mathbf{x}^{(i)} + b$ is large, corresponding to a large margin.
  - The error will be nonzero if $\mathbf{w}^T\mathbf{x}^{(i)} + b$ is small, corresponding to a small margin.
- If $\mathbf{w}^T\mathbf{x}^{(i)} + b$ and $y^{(i)}$ have opposite signs, then the hinge loss is non-negative, i.e., $\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = 1 + |\mathbf{w}^T\mathbf{x}^{(i)} + b|$.

Hinge loss intuition

The intuition of the prior slide is that the hinge loss is greatest for misclassifications, and the greater the error in misclassification, the worse the loss. For correct classifications, the loss will be zero only if there is a large enough margin.



Hinge loss extension

An extension of the hinge loss to multiple potential outputs is the following loss:

$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

for $a_j(\mathbf{x}^{(i)}) = \mathbf{w}_j^T \mathbf{x}^{(i)}$. Some intuitions, for the scenario that there are c classes:

- When the correct class achieves the highest score, $a_{y^{(i)}}(\mathbf{x}^{(i)}) \geq a_j(\mathbf{x}^{(i)})$ for all $j \neq y^{(j)}$, then $a_j(\mathbf{x}^{(i)}) - a_{y_{(i)}}(\mathbf{x}^{(i)}) \leq 0$ and

$$0 \leq \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) \leq c - 1$$

- When an incorrect class, class i, achieves the highest score, then $a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}) \geq 0$ and has the potential to be large.
- In both scenarios, it is still desirable to make the correct margins larger and the incorrect margins smaller.

The SVM cost function

If we let $\theta = \{\mathbf{w}_j\}_{j=1,\ldots,c}$, where there are c classes, we can now formulate the SVM optimization function. In particular, we want to minimize the hinge loss across all training examples. Then, to optimize θ for a linear kernel and hinge loss, we solve the following minimization problem:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)})$$

which, for the sake of completeness, can be writen as:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

Support vector machine

NOT TESTED

Is there a closed-form solution?

Support vector machine

NOT TESTED

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2

Score check: $\sum_{i \neq y^{(j)}} \max(0, 1 + a_i(\mathbf{x}^{(j)}) - a_{y^{(j)}}(\mathbf{x}^{(j)}))$

Support vector machine

NOT TESTED

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2

Cat: max(0, 1 - 2.1 + 3.4) + max(0, 1 - 2.1 - 2.0) = max(0, 2.3) + max(0, -3.1) = 2.3

Support vector machine

NOT TESTED

Simple sanity check:

Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	2.3		

Car: max(0, 1 - 5.1 + 0.2) + max(0, 1 - 5.1 +1.7) = max(0, -3.9) + max(0, -2.4) = 0

Support vector machine

NOT TESTED

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	0.3	0	

Bird: max(0, 1 + 1.2 + 2.3) + max(0, 1 + 1.2 + 3.1) = max(0, 4.5) + max(0, 5.3) = 9.8

Support vector machine

NOT TESTED

Simple sanity check:



Cat:	2.1	0.2	2.3
Car:	3.4	5.1	3.1
Bird:	-2.0	1.7	-1.2
Loss:	0.3	0	9.8

$$\sum_{i \neq y^{(j)}} \max(0, 1 + a_i(\mathbf{x}^{(j)}) - a_{y^{(j)}}(\mathbf{x}^{(j)}))$$

Softmax loss function

Softmax:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left(\log \sum_{j=1}^{c} e^{a_j(\mathbf{x})} - a_{y^{(i)}}(\mathbf{x}^{(i)}) \right)$$

Parameters?

Softmax loss function

Softmax: $$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \left(\log \sum_{j=1}^{c} e^{a_j(\mathbf{x})} - a_{y^{(i)}}(\mathbf{x}^{(i)}) \right)$$

Parameters: $\mathbf{W} \in \mathbb{R}^{c \times n}, \mathbf{b} \in \mathbb{R}^{c}$

Big question: how do we find these parameters?

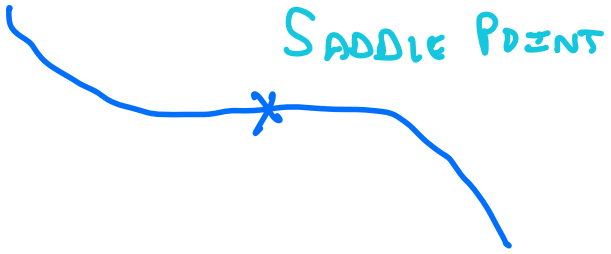
Finding the optimal weights through gradient descent

- Our goal in machine learning is to optimize an objective function, $f(x)$. (Without loss of generality, we'll consider minimizing $f(x)$. This is equivalent to maximizing $-f(x)$.)
- From basic calculus, we recall that the derivative of a function, $\frac{df(x)}{dx}$ tells us the slope of $f(x)$ at point x.
  - For small enough ϵ, $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$.
  - This tells us how to reduce (or increase) $f(\cdot)$ for small enough steps.
  - Recall that when $f'(x) = 0$, we are at a stationary point or critical point. This may be a local or global minimum, a local or global maximum, or a saddle point of the function.
- In this class we will consider cases where we would like to maximize f w.r.t. vectors and matrices, e.g., $f(\mathbf{x})$ and $f(\mathbf{X})$.
- Further, often $f(\cdot)$ contains a nonlinearity or non-differentiable function. In these cases, we can't simply set $f'(\cdot) = 0$, because this does not admit a closed-form solution.
- However, we can iteratively approach an critical point via gradient descent.

Finding the optimal weights through gradient descent

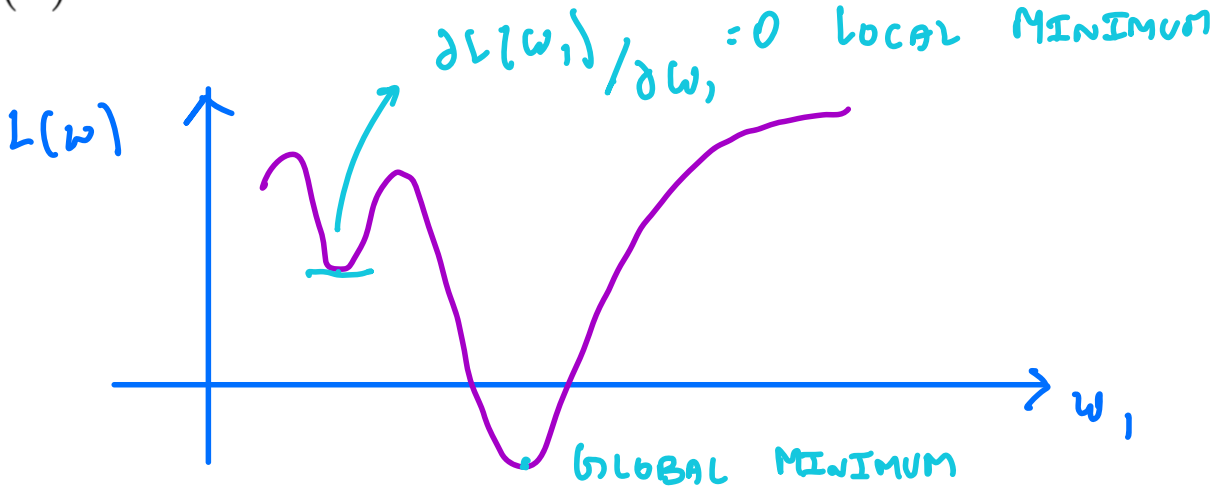
To do so, we use the technique of gradient descent.

Finding the optimal weights through gradient descent



Terminology

- A **global minimum** is the point, $\mathbf{x}_g$, that achieves the absolute lowest value of $f(\mathbf{x})$. i.e., $f(\mathbf{x}) \geq f(\mathbf{x}_g)$ for all $\mathbf{x}$.
- A **local minimum** is a point, $\mathbf{x}_\ell$, that is a critical point of $f(\mathbf{x})$ and is lower than its neighboring points. However, $f(\mathbf{x}_\ell) > f(\mathbf{x}_g)$.
- Analogous definitions hold for the **global maximum** and **local maximum**.
- A **saddle point** are critical point of $f(\mathbf{x})$ that are not local maxima or minima. Concretely, neighboring points are both greater than and less than $f(\mathbf{x})$.



The problem of 'stuck in local minima' is not an issue for Neural Network.

↳ has millions of weights.

So local minima of one weight are not local minima for other weights. So we can minimize along those directions.

Finding the optimal weights through gradient descent

Gradient

Recall the gradient, $\nabla_{\mathbf{x}} f(\mathbf{x})$, is a vector whose ith element is the partial derivative of $f(\mathbf{x})$ w.r.t. x_i, the ith element of $\mathbf{x}$. Concretely, for $\mathbf{x} \in \mathbb{R}^n$,

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial (x_n)} \end{bmatrix}$$

$\Delta x: \begin{bmatrix} \delta x_1 \\ \delta x_2 \\ \vdots \\ \delta x_n \end{bmatrix}$

$x \to f(x)$

$x + \Delta x \to f(x) + \Delta x^T \nabla_x f(x)$

- The gradient tells us how a small change in $\Delta \mathbf{x}$ affects $f(\mathbf{x})$ through

$$f(\mathbf{x} + \Delta \mathbf{x}) \approx f(\mathbf{x}) + \Delta \mathbf{x}^T \nabla_{\mathbf{x}} f(\mathbf{x})$$

- The directional derivative of $f(\mathbf{x})$ in the direction of the unit vector $\mathbf{u}$ is given by:

$$\mathbf{u}^T \nabla_{\mathbf{x}} f(\mathbf{x}) \qquad \|u\| = 1$$

- The directional derivative tells us the slope of f in the direction $\mathbf{u}$.

Finding the optimal weights through gradient descent

Arriving at gradient descent

- To minimize $f(\mathbf{x})$, we want to find the direction in which $f(\mathbf{x})$ decreases the fastest. To do so, we find the direction $\mathbf{u}$ which minimizes the directional derivative.

$$\begin{aligned} \min_{\mathbf{u}, \|\mathbf{u}\|=1} \mathbf{u}^T \nabla_{\mathbf{x}} f(\mathbf{x}) &= \min_{\mathbf{u}, \|\mathbf{u}\|=1} \|\mathbf{u}\| \, \|\nabla_{\mathbf{x}} f(\mathbf{x})\| \cos\theta \\ &= \min_{\mathbf{u}} \nabla_{\mathbf{x}} f(\mathbf{x}) \cos(\theta) \end{aligned}$$

  where θ is the angle between the vectors $\mathbf{u}$ and $\nabla_{\mathbf{x}} f(\mathbf{x})$.
- This quantity is minimized for $\mathbf{u}$ pointing in the opposite direction of the gradient, so that $\cos(\theta) = -1$.
- Hence, we arrive at gradient descent. To update $\mathbf{x}$ so as to minimize $f(\mathbf{x})$, we repeatedly calculate:

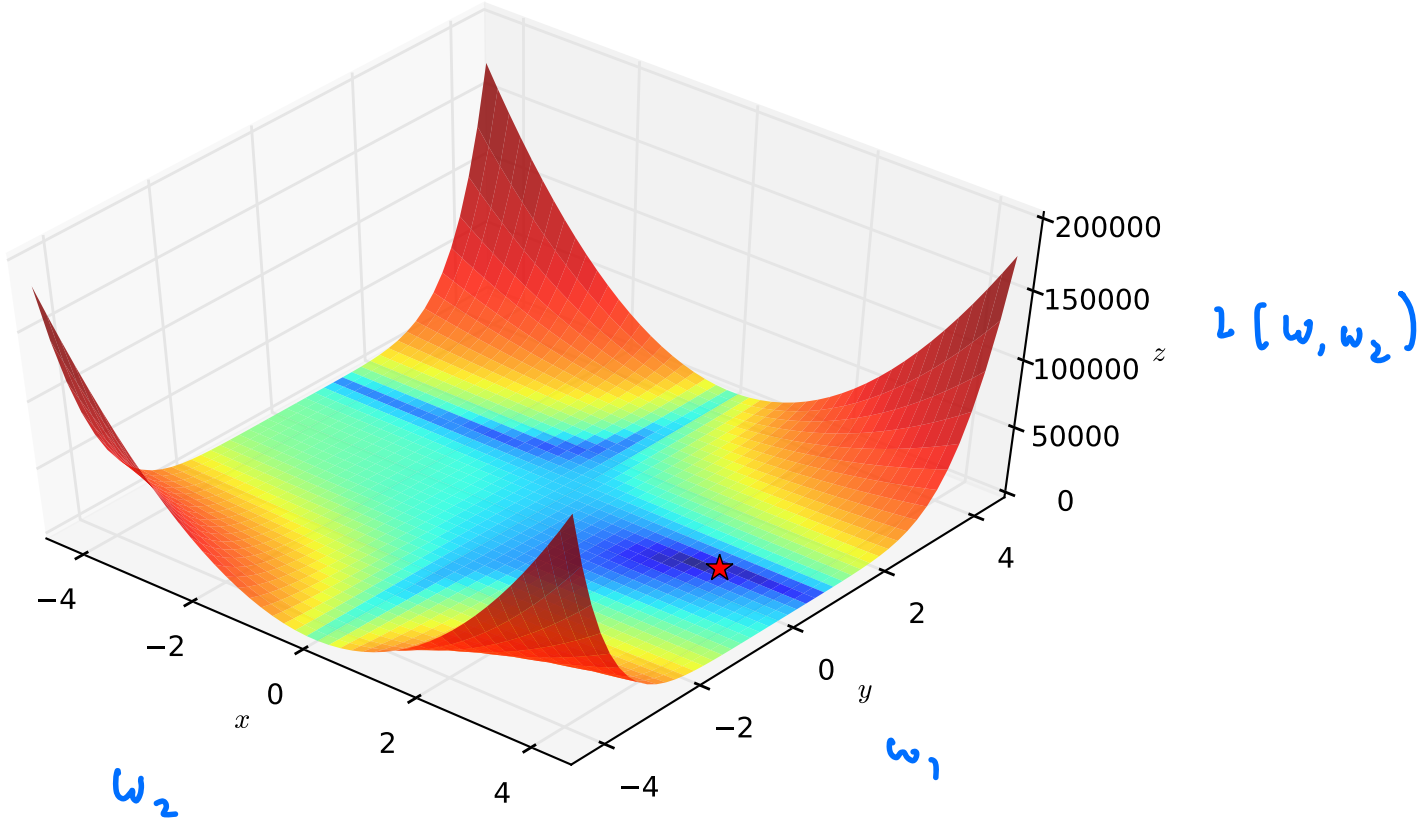
$$\mathbf{x} := \mathbf{x} - \epsilon \nabla_x f(\mathbf{x})$$

- ϵ is typically called the *learning rate*. It can change over iterations. Setting the value of ϵ appropriately is an important part of deep learning.

Finding the optimal weights through gradient descent

Example:

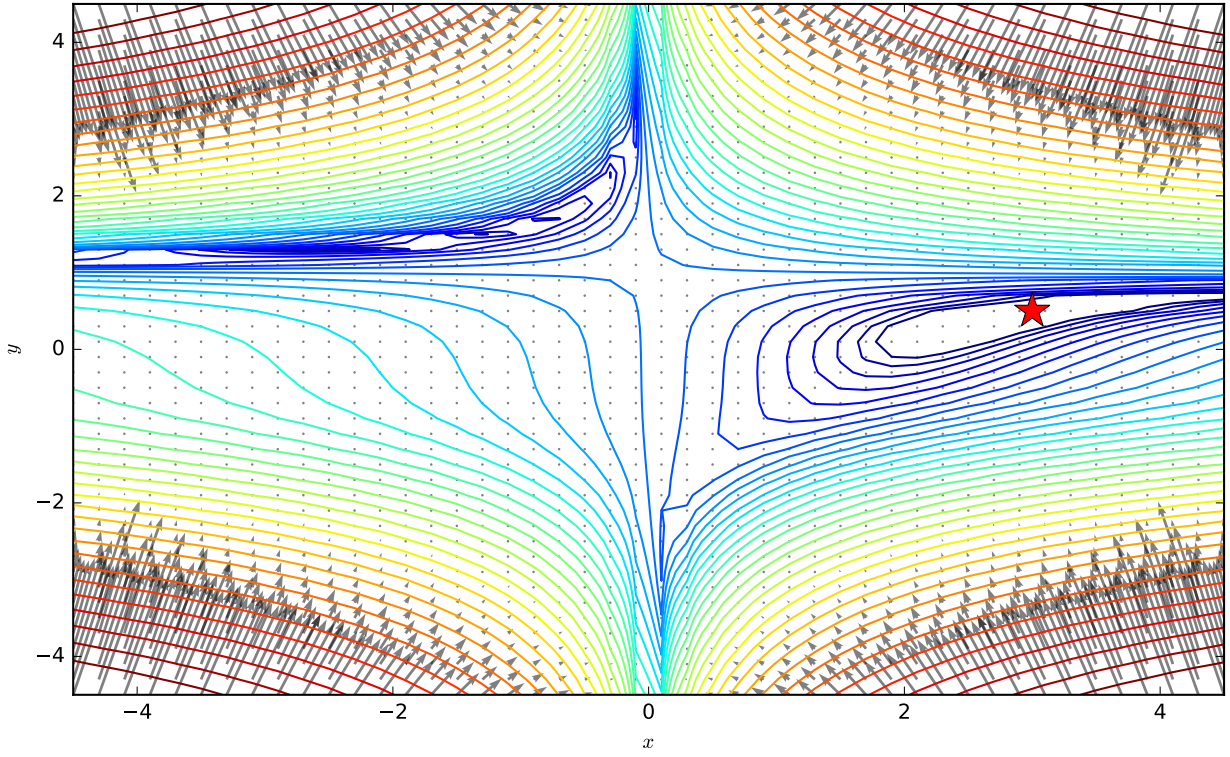
Animations thanks to: http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/



Finding the optimal weights through gradient descent

Example:

Animations thanks to: http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/



Finding the optimal weights through gradient descent

```
def gd(func, x0, eps=1e-4, tol=1e-3):
    last_diff = np.Inf
    x = x0
    path = [np.copy(x0)]
    costs = [func(x0)[0]]
    grads = []
    i = 1
    hit_max = False

    while last_diff > tol:
        cost, g = func(x) # returns the cost and the gradient
        x -= eps*g # gradient step
        last_diff = np.linalg.norm(x - path[-1]) # stopping criterion

        i += 1
        if i > max_iters:
            hit_max = True
            break
        path.append(np.copy(x))
        costs.append(cost)
        grads.append(g)

    return path, costs, grads, hit_max
```

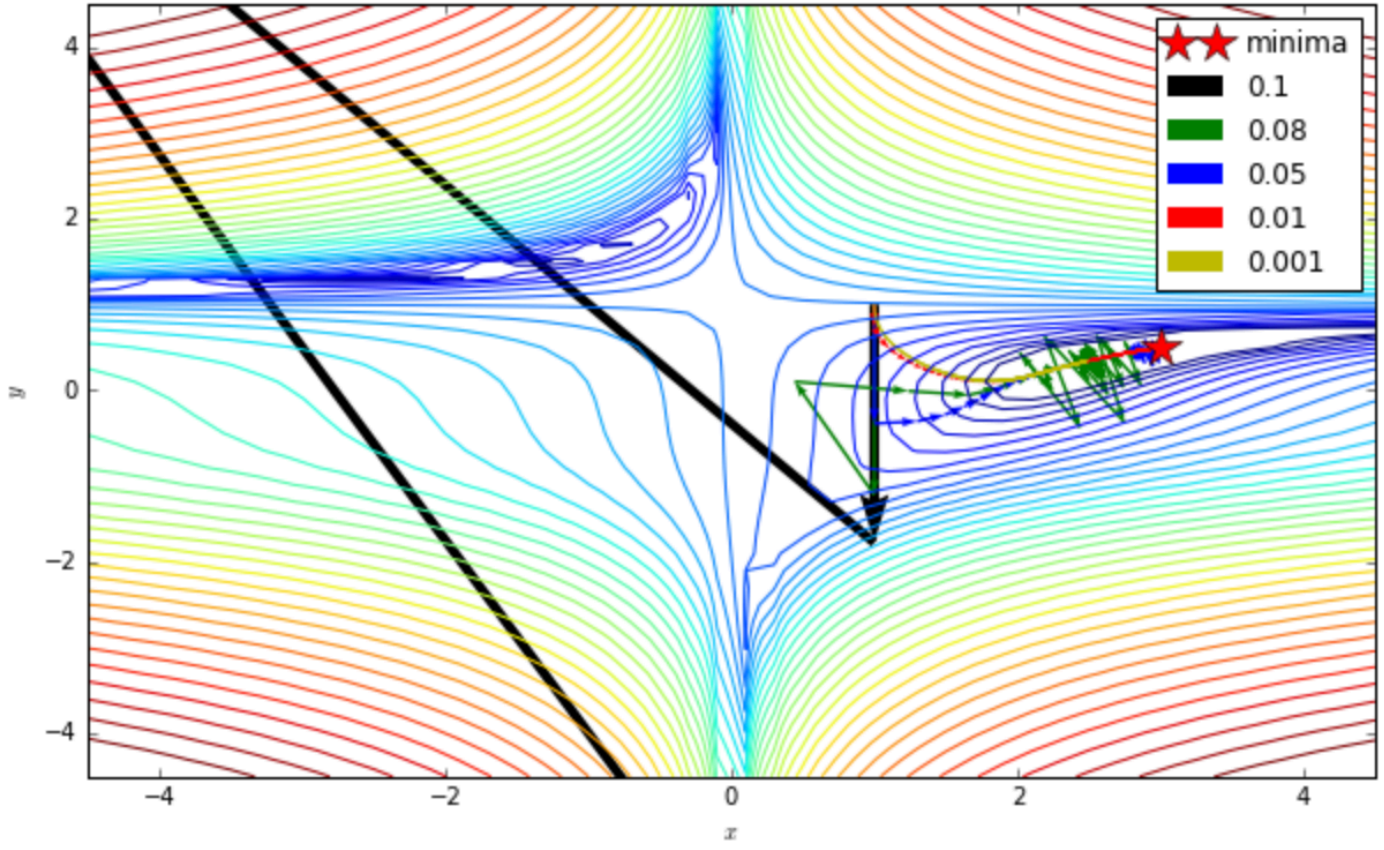
softmax_loss_and_grad

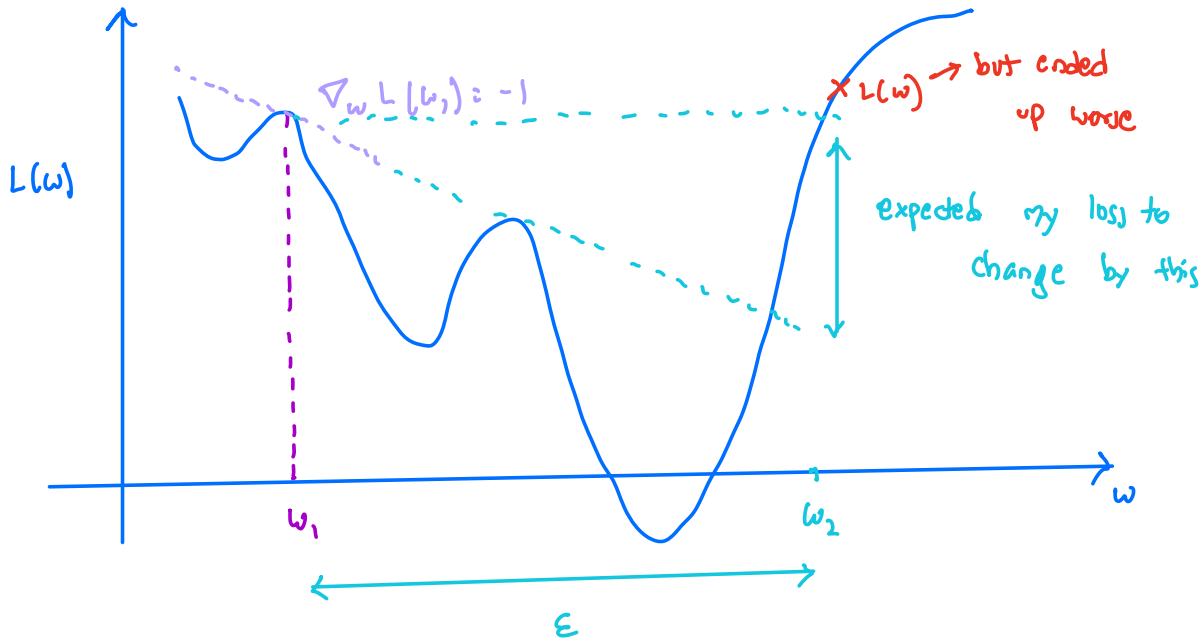
Finding the optimal weights through gradient descent

http://seas.ucla.edu/~kao/opt_anim/1gd.mp4
http://seas.ucla.edu/~kao/opt_anim/2gd.mp4

Finding the optimal weights through gradient descent

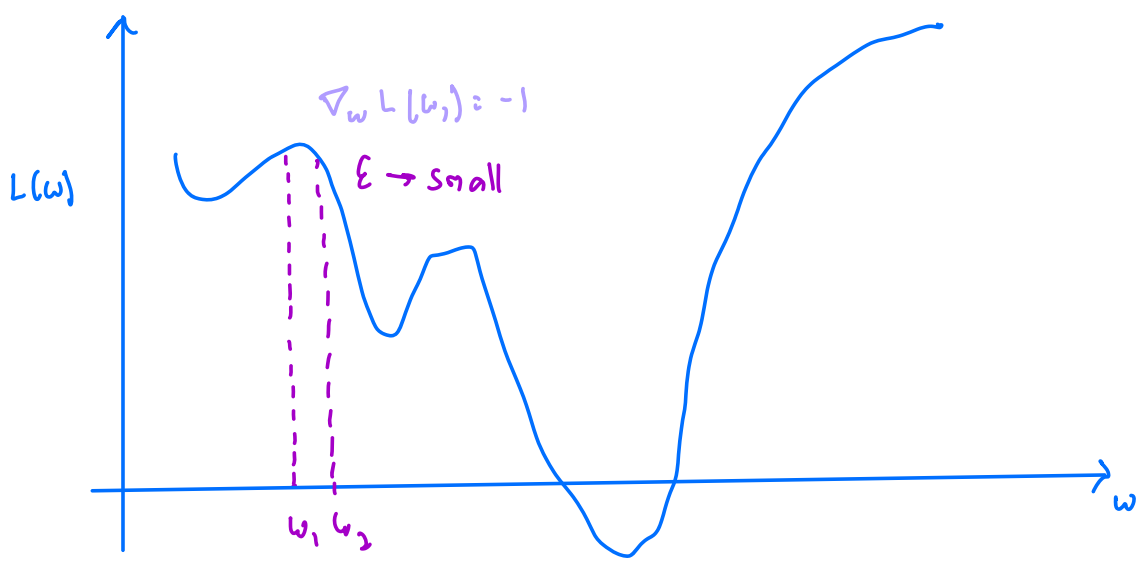
How do I pick the right step size?





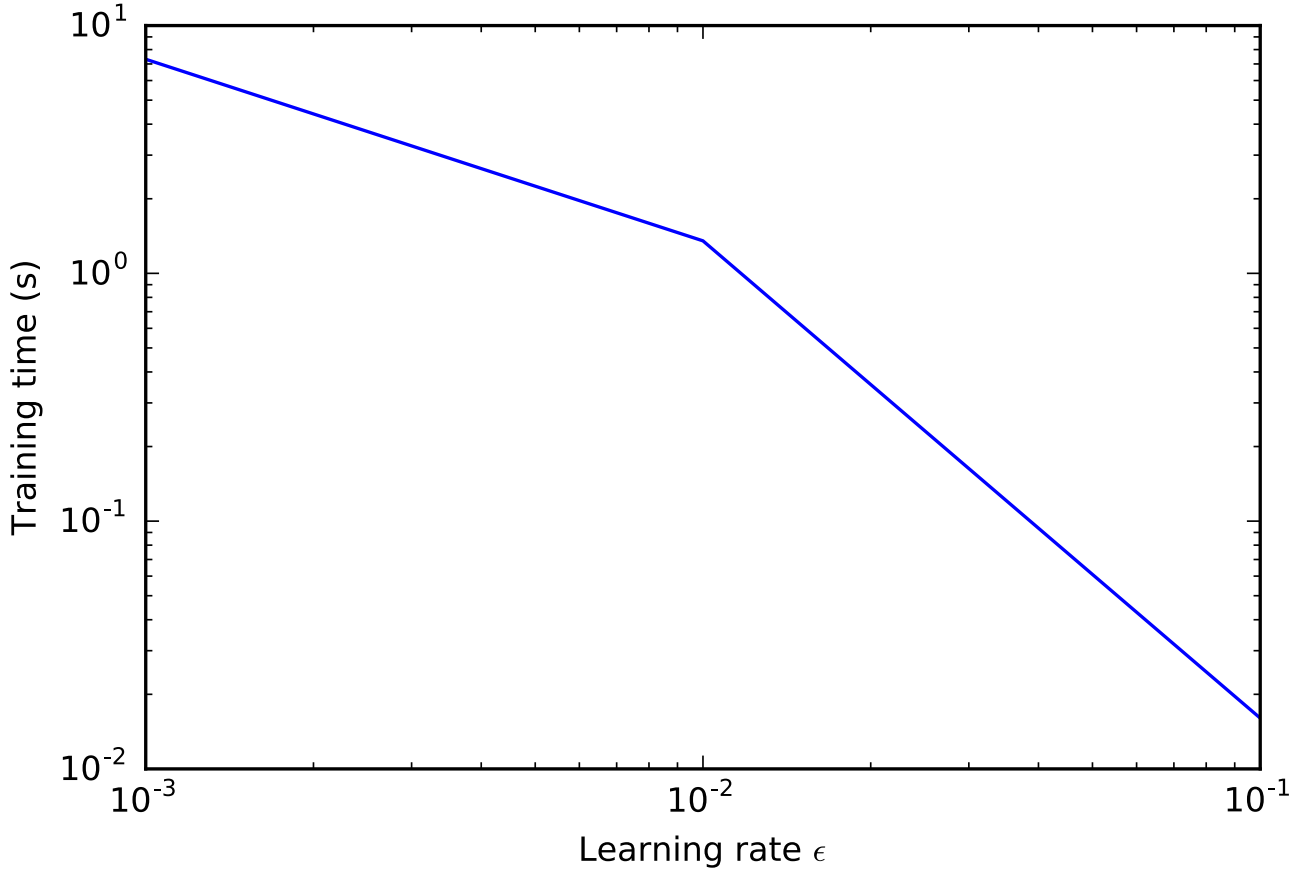
$$w_2 = w_1 - \varepsilon \nabla_w L(w_1)$$

$$= w_1 + \varepsilon$$



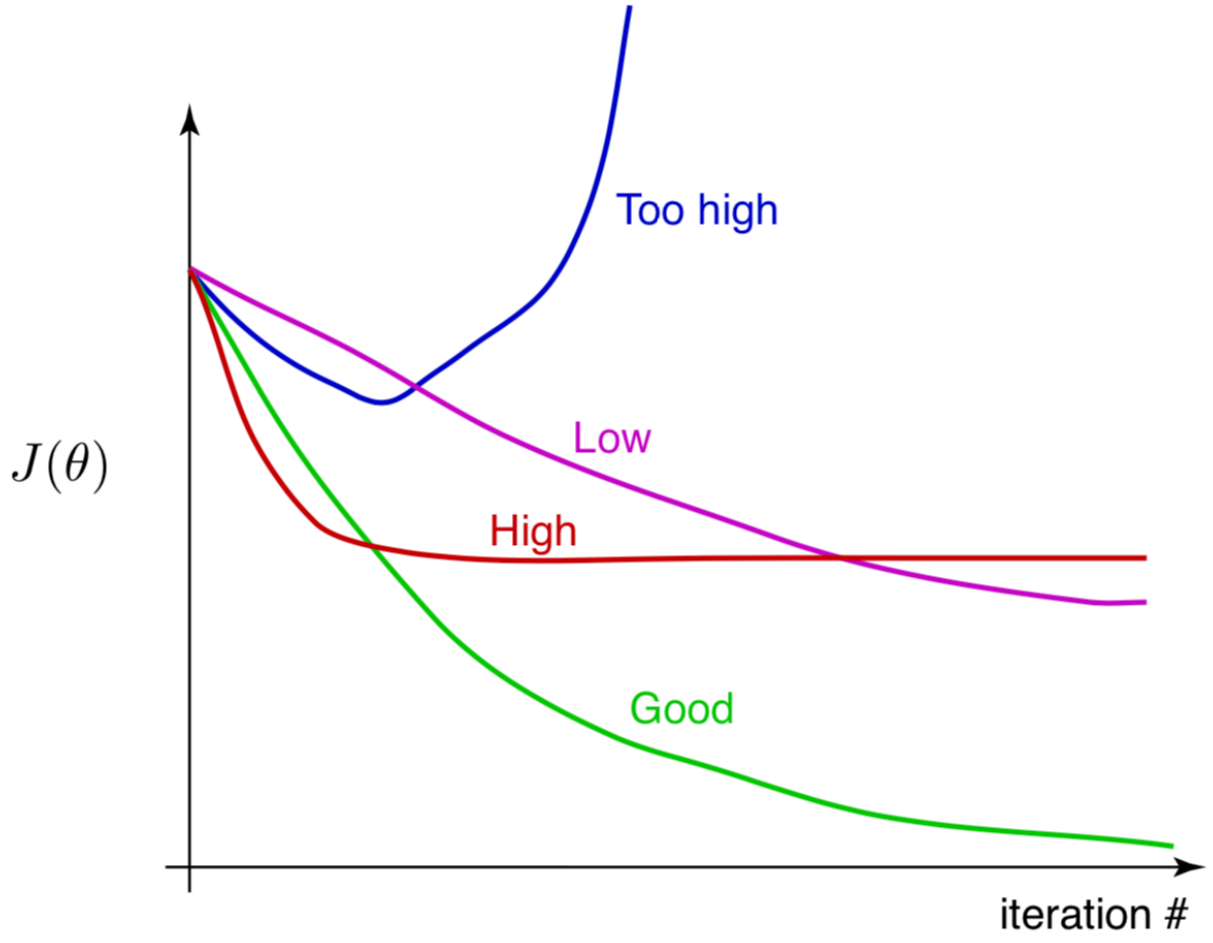
Finding the optimal weights through gradient descent

Why not always use smaller learning rates?



Interpreting the cost

The cost function can be very informative as to how to adjust your step sizes for gradient descent.



Why not use a numerical gradient?

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

(1) # of parameters

(2) f may be expensive

ANALYTICAL: $\nabla_{w_i} L(\theta) = f(x^{(i)}, y^{(i)})$

eg: $f(x^{(i)}, y^{(i)})$ could be $2x^{(i)}$

NUMERICAL: Given θ

$$\nabla_\theta L(\theta) = \lim_{\Delta\theta \to 0} \frac{L(\theta + \Delta\theta) - L(\theta)}{\Delta\theta}$$

Finding the optimal weights through gradient descent

How does this example differ from what we will really encounter?

- In this example, we know the function f() exactly, and thus at every point in space, we can calculate the gradient at that point exactly.
- In optimization, we differentiate the cost function f() with respect to the parameters.
    - The gradient of f() w.r.t. parameters is a function of the training data!
    - Hence, we can think of each data point as providing a noisy estimate of the gradient at that point.

Finding the optimal weights through gradient descent

However, it's expensive to have to calculate the gradient by using *every example* in the training set.

To this end, we may want to get a noisier estimate of the gradient with fewer examples.

Batch vs minibatch (cont)

Calculating the gradient exactly is expensive, because it requires evaluating the model on all m examples in the dataset. This leads to an important distinction.

- Batch algorithm: uses all m examples in the training set to calculate the gradient.
- Minibatch algorithm: approximates the gradient by calculating it using k training examples, where $m > k > 1$.
- Stochastic algorithm: approximates the gradient by calculating it over one example.

It is typical in deep learning to use minibatch gradient descent. Note that some may also use minibatch and stochastic gradient descent interchangeably.

Finding the optimal weights through gradient descent

To get a more robust estimate of the gradient, we would use as many data samples as possible.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta)$$

and its gradient is:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta) \\
&= \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta) \\
&\approx \mathbb{E}\left[\nabla_\theta \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta)\right]
\end{aligned}$$

Finding the optimal weights through gradient descent

You'll do this in the HW. More on this later in the optimization lecture…

- And a lot more to be said about optimization.
- First order vs second order methods
- Momentum
- Adaptive gradients.
- … all of these will become quite important when we get to neural networks. We'll cover these in an optimization lecture.

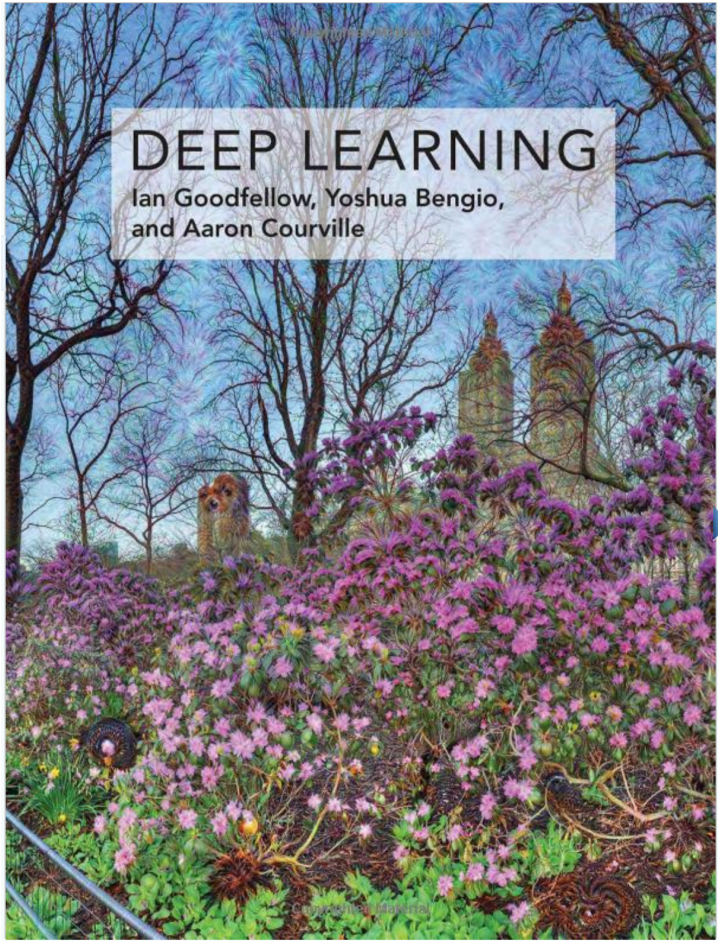
Lecture 5: Neural networks

In this lecture, we'll introduce the neural network architecture, parameters, and its inspiration from biological neurons.

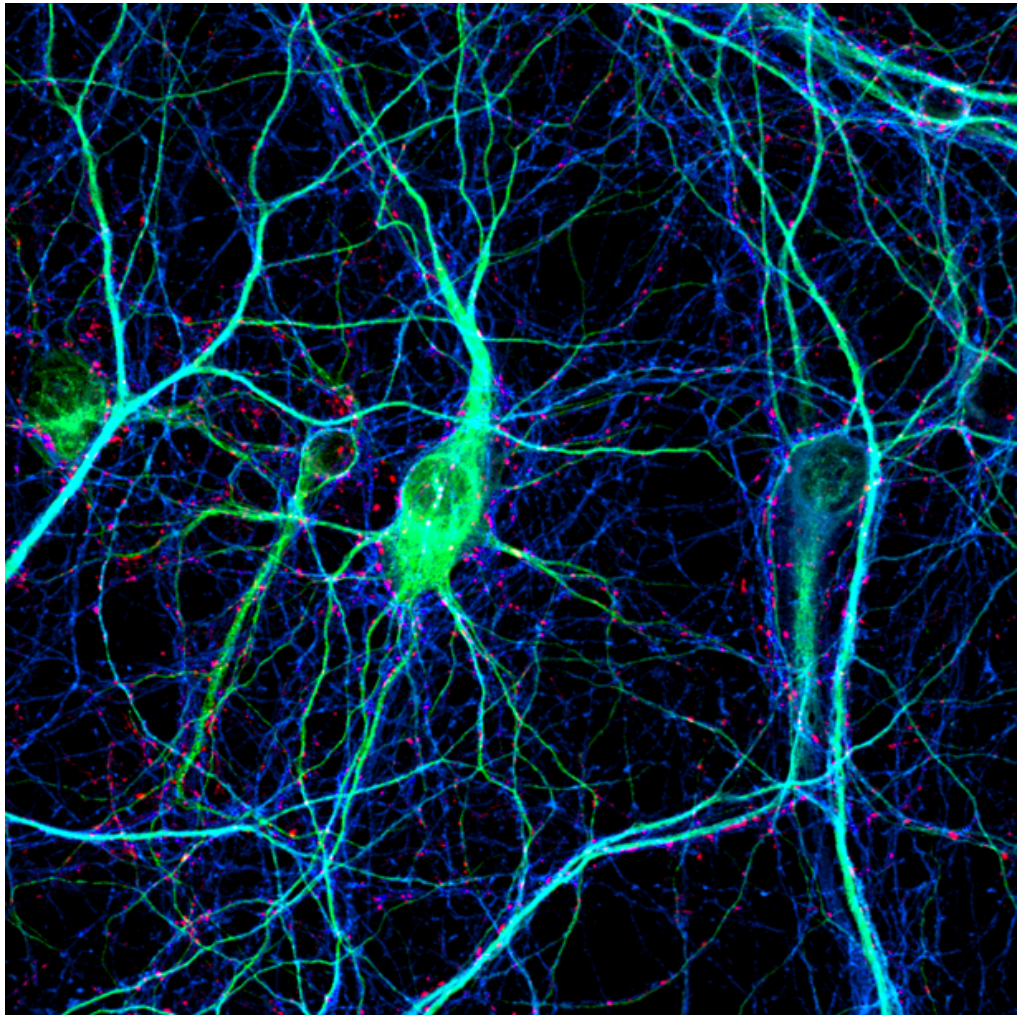
Announcements, 2018-01-22

Reading:

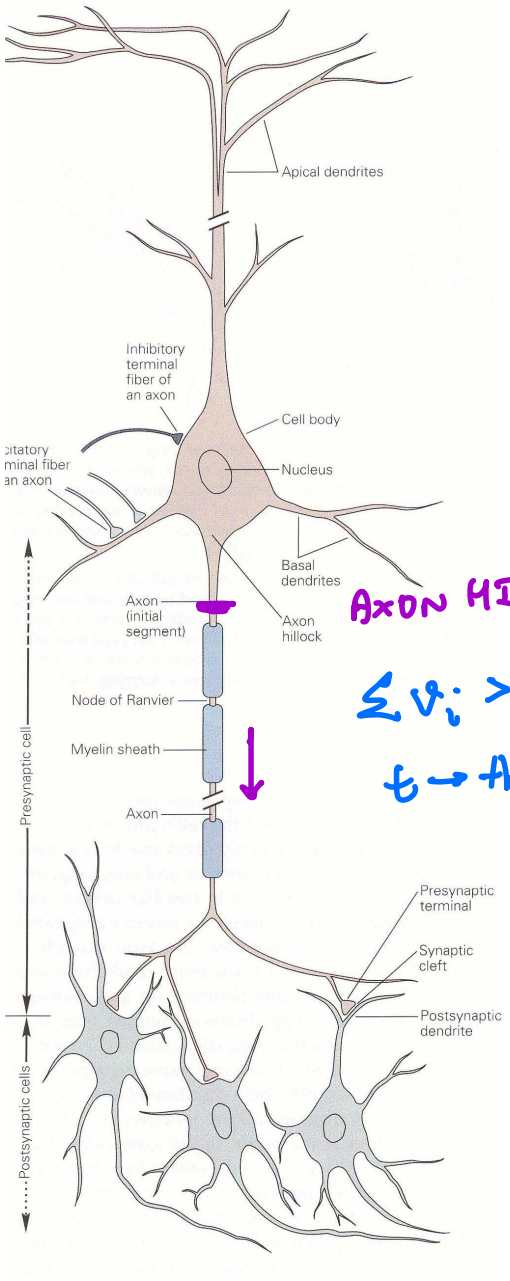
Deep Learning, 6 (intro), 6.1, 6.2, 6.3, 6.4



Inspiration from neuroscience



Inspiration from neuroscience



Axon Hillock

$\sum v_i > t$

$t \rightarrow$ threshold

sends signal called spike

Neurons are the main signaling units of the nervous system.

- Neurons have four regions:
  1) Cell body (soma) – metabolic center, with nucleus, etc.
  2) Dendrites – tree like structure for receiving **input** signals.
  3) Axon – single, long, tubular structure for sending **output** signals.
  4) Presynaptic terminals – sites of communication to next neurons.

- Axons (the **output)** convey signals to other neurons:
  - Conveys electrical signals long distances (0.1mm – 3 m).
  - Conveys action potentials (~100 mV, ~1 ms pulses).
  - Action potentials initiate at the axon hillock.
  - Propagate w/o distortion or failure at 1-100 m/s.

Inspiration from neuroscience

Neurons are diverse (unlike in neural networks)

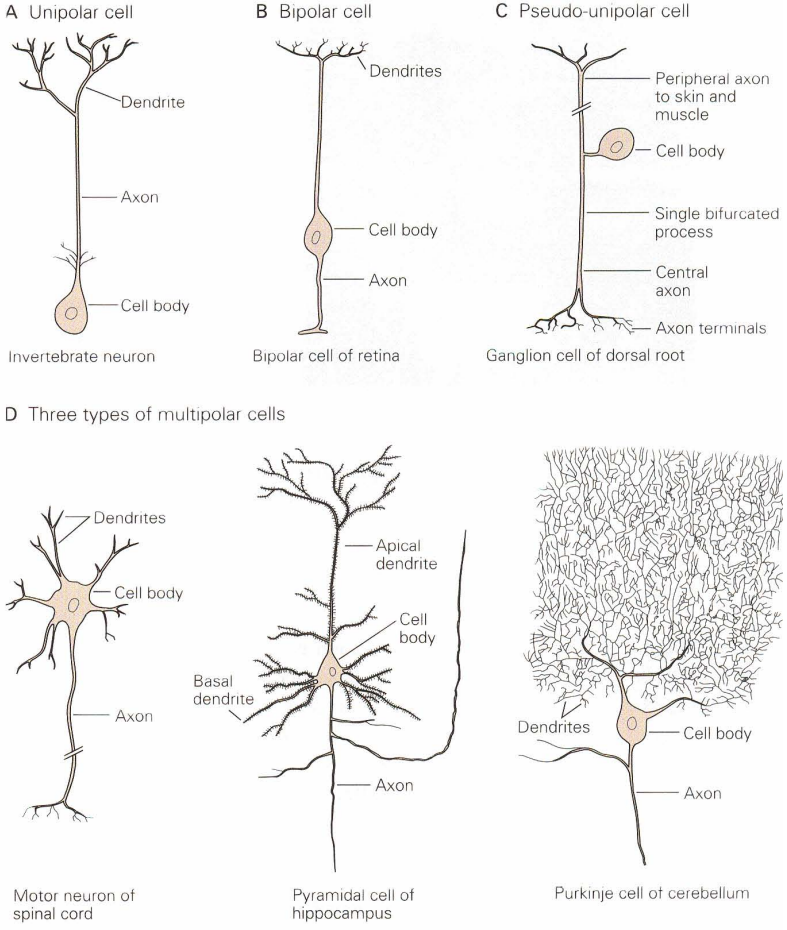


Figure 2-4 Neurons can be classified as unipolar, bipolar, or multipolar according to the number of processes that originate from the cell body.

A. Unipolar cells have a single process, with different segments serving as receptive surfaces or releasing terminals. Unipolar cells are characteristic of the invertebrate nervous system.

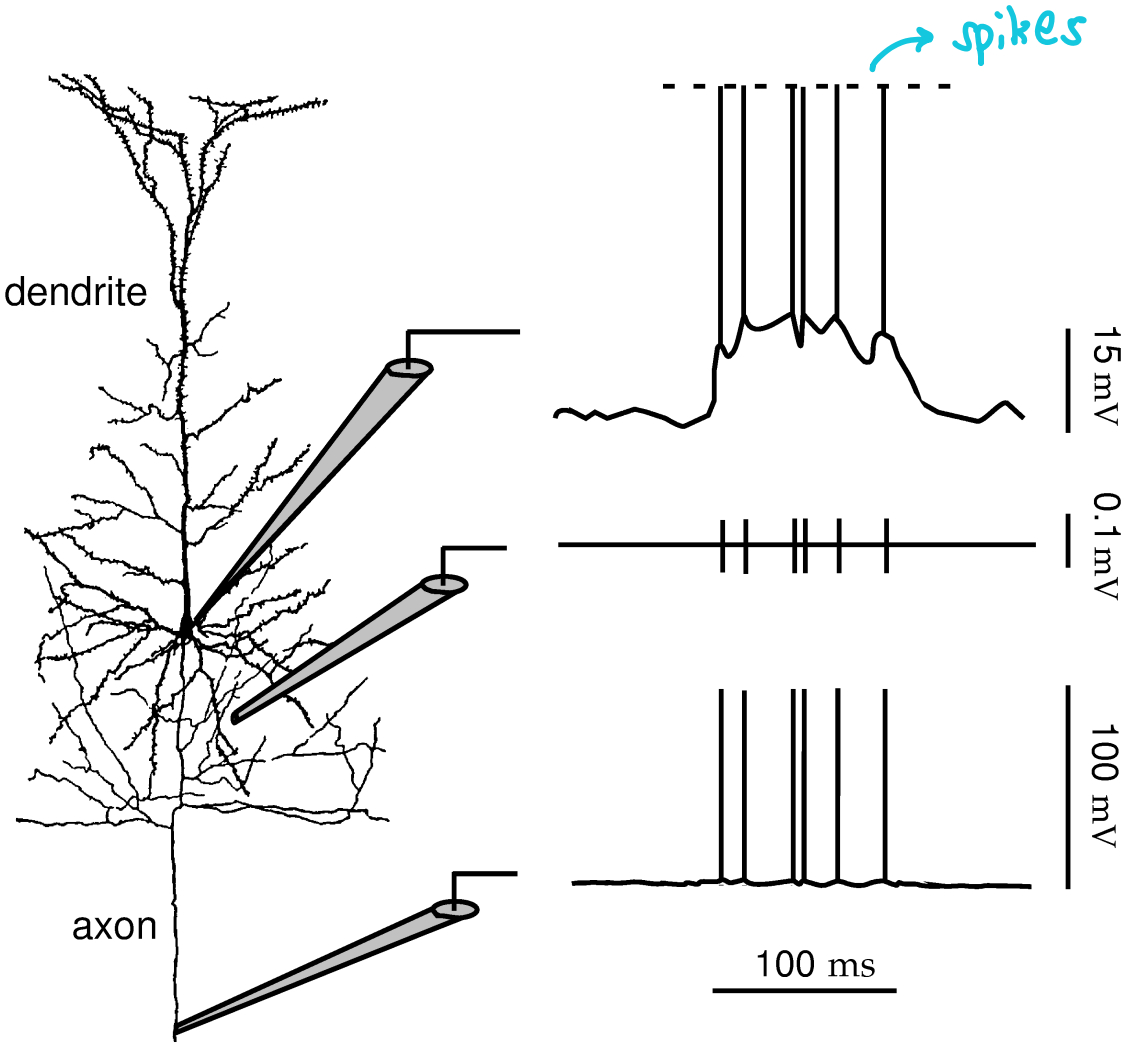
B. Bipolar cells have two processes that are functionally specialized: the dendrite carries information to the cell, and the axon transmits information to other cells.

C. Certain neurons that carry sensory information, such as information about touch or stretch, to the spinal cord belong to a subclass of bipolar cells designated as pseudo-unipolar. As such cells develop, the two processes of the embryonic bipolar cell become fused and emerge from the cell body as a single process. This outgrowth then splits into two processes, *both* of which function as axons, one going to peripheral skin or muscle, the other going to the central spinal cord.

D. Multipolar cells have an axon and many dendrites. They are the most common type of neuron in the mammalian nervous system. Three examples illustrate the large diversity of these cells. Spinal motor neurons (left) innervate skeletal muscle fibers. Pyramidal cells (middle) have a roughly triangular cell body; dendrites emerge from both the apex (the apical dendrite) and the base (the basal dendrites). Pyramidal cells are found in the hippocampus and throughout the cerebral cortex. Purkinje cells of the cerebellum (right) are characterized by the rich and extensive dendritic tree in one plane. Such a structure permits enormous synaptic input. (Adapted from Ramón y Cajal 1933.)

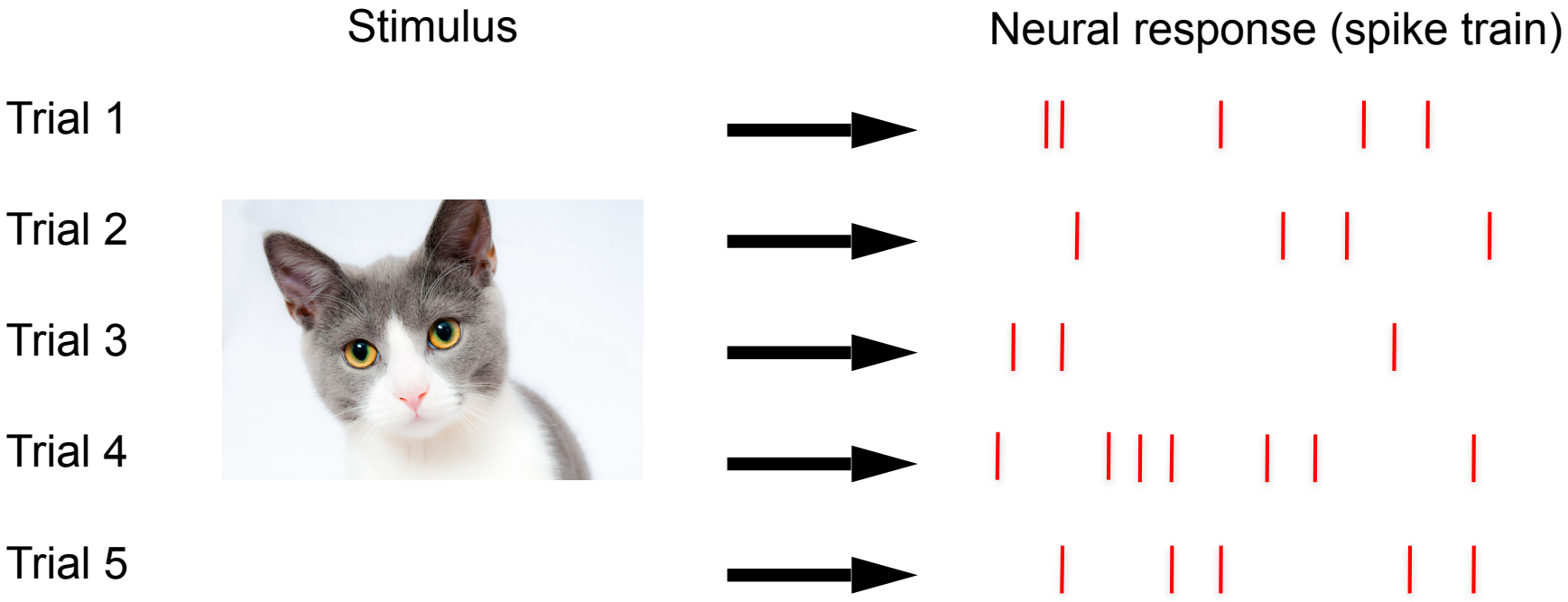
Inspiration from neuroscience

Neurons fundamentally communicate through all-or-nothing spikes (not analog values!):



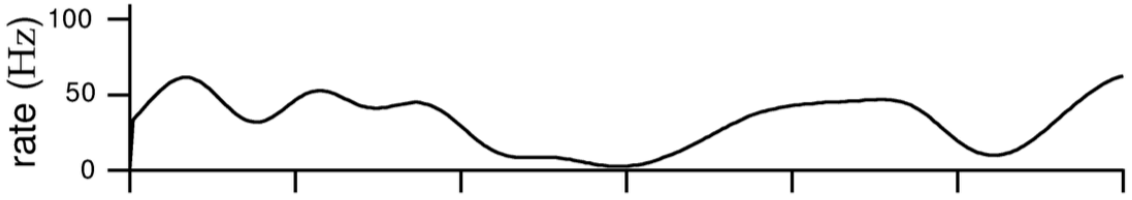
Inspiration from neuroscience

… And the spikes are probabilistic.



Inspiration from neuroscience

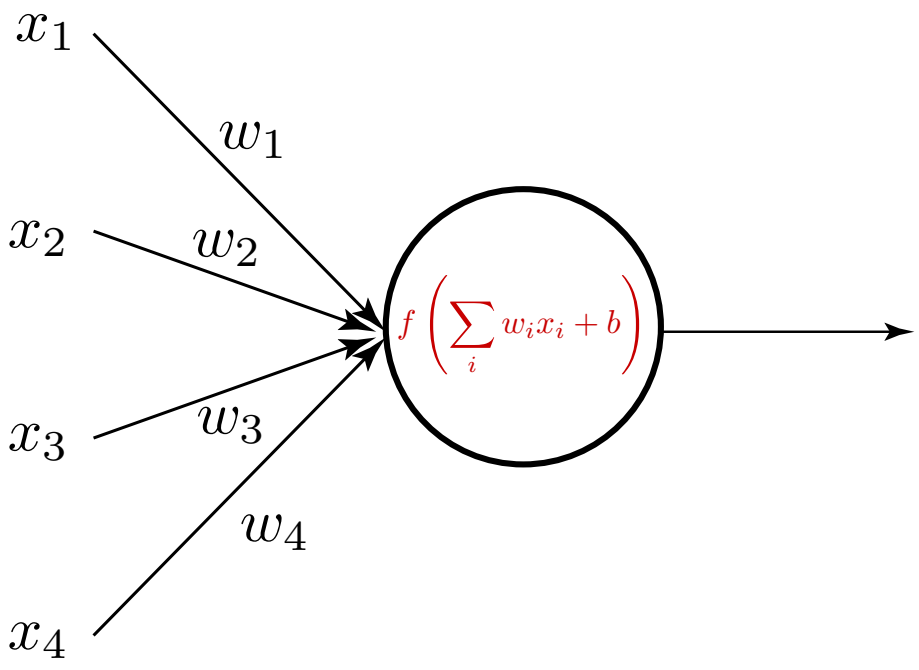
The spikes reflect an underlying rate.



This rate is what the neural networks are “encoding.”

Inspiration from neuroscience

How does the artificial neuron compare to the real neuron?



Inspiration from neuroscience

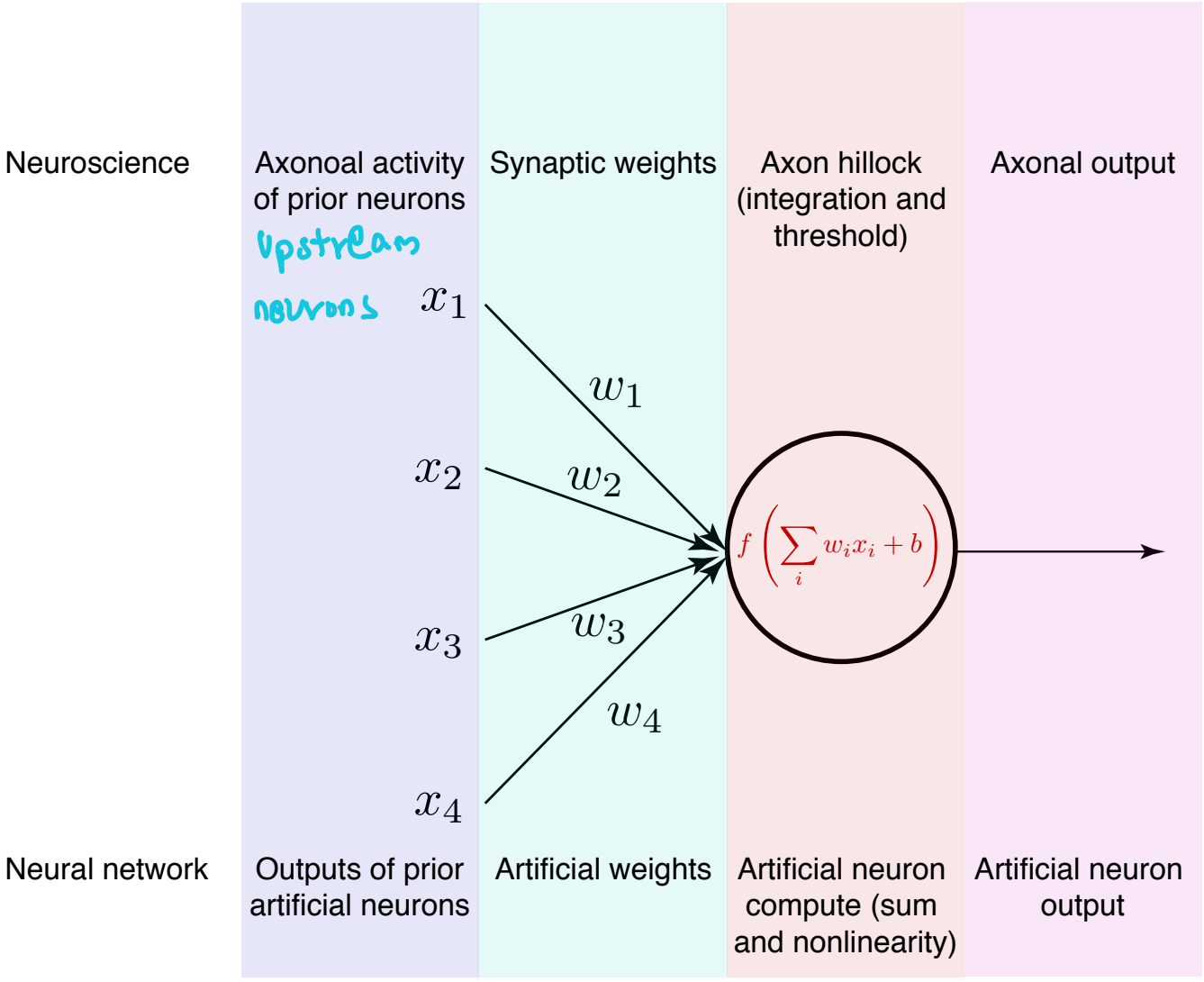
How does the artificial neuron compare to the real neuron?

The artificial neuron (cont.)

- The incoming signals, a vector $\mathbf{x} \in \mathbb{R}^N$, reflects the output of N neurons that are connected to the current artificial neuron.
- The incoming signals, $\mathbf{x}$, are pointwise multiplied by a vector, $\mathbf{w} \in \mathbb{R}^N$. That is, we calculate $w_i x_i$ for $i = 1, \ldots, N$. This computation reflects dendritic processing.
- The "dendritic-processed" signals are then summed, i.e., we calculate $\sum_i w_i x_i + b$. This computation reflects integration at the axon hillock (the first "Node of Ranvier") where action potentials are generated if the integrated signal is large enough.
- The output of the artificial neuron is then a nonlinearly transformation of the integrated signal, i.e., $f(\sum_i w_i x_i + b)$. Rather than reflecting whether an action potential was generated or not (which is a noisy process), this nonlinear output is typically treated as the *rate* of the neuron. The higher the rate, the more likely the neuron is to fire action potentials.

Inspiration from neuroscience

How does the artificial neuron compare to the real neuron?



Inspiration from neuroscience

Caution when comparing to biology

These computing analogies are not precise, with large approximations. Limitations in the analogy include:

- Synaptic transmission is probabilistic, nonlinear, and dynamic.
- Dendritic integration is probabilistic and may be nonlinear.
- Dendritic computation has associated spatiotemporal decay.
- Integration is subject to biological constraints; for example, ion channels (which change the voltage of the cell) undergo refractory periods when they do not open until hyperpolarization.
- Different neurons may have different action potential thresholds depending on the density of sodium-gated ion channels.
- Feedforward and convolutional neural networks have no recurrent connections.
- Many different cell types.
- Neurons have specific dynamics that can be modulated by e.g., calcium concentration.
- And so many more...

Inspiration from neuroscience

Caution when comparing to biology

On the prior list, several of these bullet points constitute entire research areas. E.g., several labs work specifically on studying the details of synaptic transmission.

Big picture: though neural networks are inspired by biology, they approximate biological computation at a fairly crude level. These networks ought not be thought of as models of the brain, although recent work (including my research group's work) has used them as a means to propose mechanistic insight into neurophysiological computation.

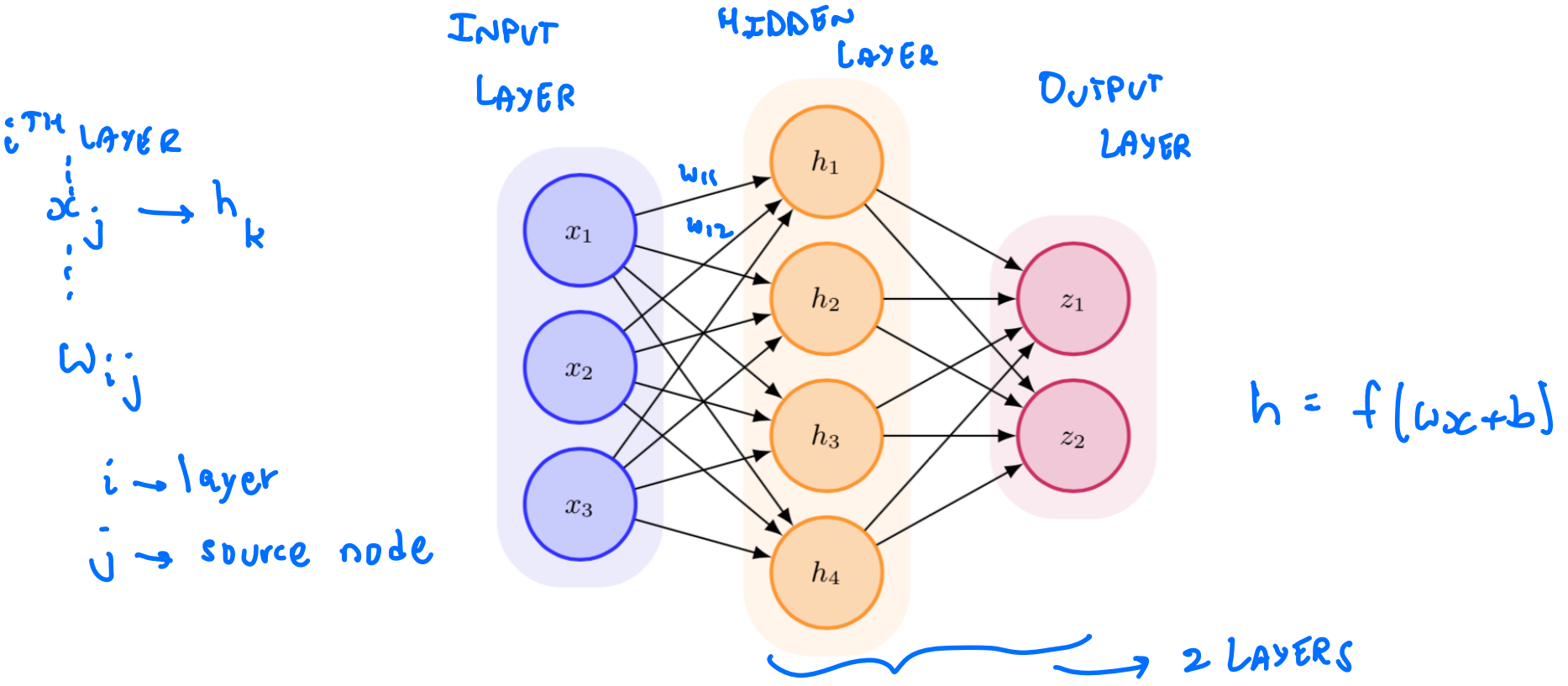
Nomenclature

Some naming conventions.

- We call the first layer of a neural network the "input layer." We typically represent this with the variable $\mathbf{x}$.
- We call the last layer the "output layer." We typically represent this with the variable $\mathbf{z}$. (Note: why not $\mathbf{y}$ to match our prior nomenclature for the supervised outputs? Because the output of the network may be a processed version of $\mathbf{z}$, e.g., $\mathrm{softmax}(\mathbf{z})$.)
- We call the intermediate layers the "hidden layers." We typically represent this with the variable $\mathbf{h}$.
- When we specify that a network has N layers, this does not include the input layer.

Neural network architecture

An example 2-layer network is shown below.



i^{TH} LAYER

$x_j \rightarrow h_k$

W_{ij}

$i \rightarrow$ layer

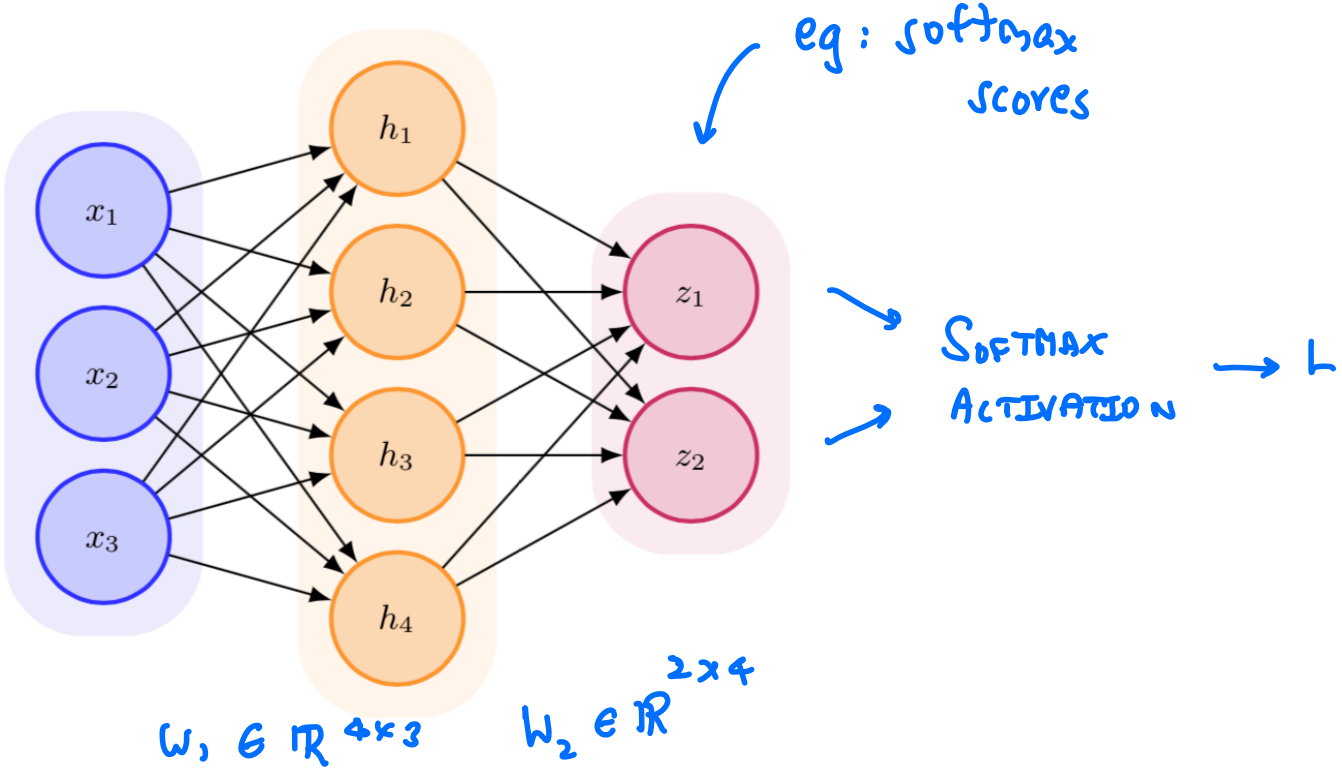
$j \rightarrow$ source node

$h = f(Wx+b)$

Here, the three dimensional inputs ($\mathbf{x} \in \mathbb{R}^3$) are processed into a four dimensional intermediate representation ($\mathbf{h} \in \mathbb{R}^4$), which are then transormed into the two dimensional outputs ($\mathbf{z} \in \mathbb{R}^2$).

$$h_1 = f(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1)$$

Neural networks



First layer: $\mathbf{h} = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

Second (output layer): $\mathbf{z} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$ No f

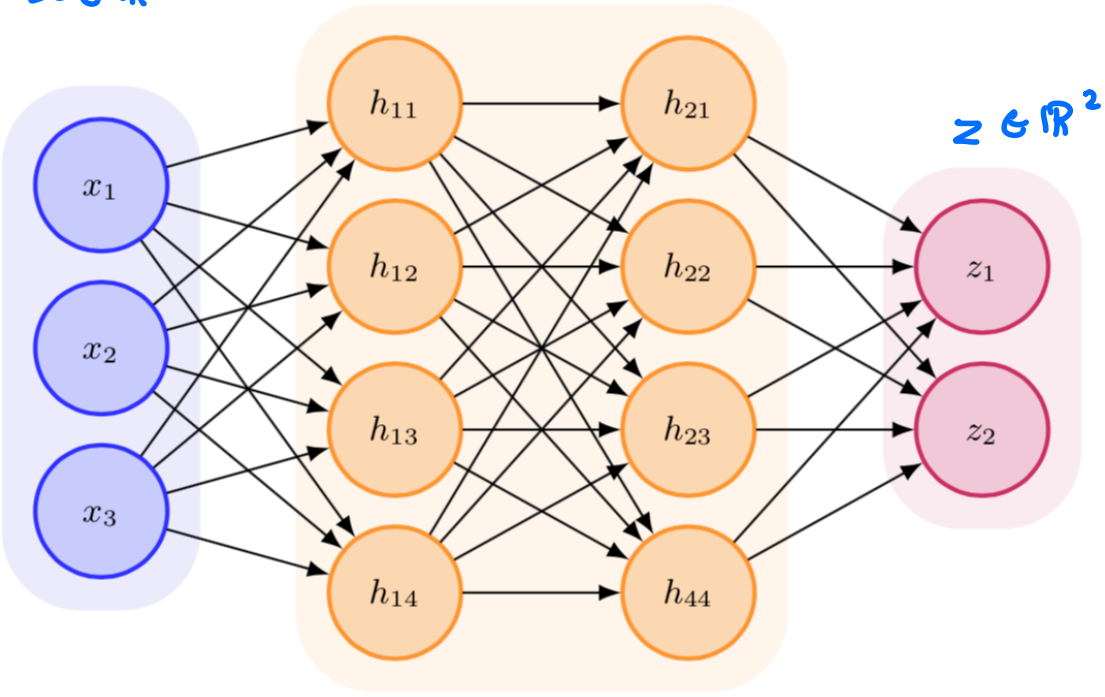
This network has 6 neurons (not counting the input). It has (3x4) + (4x2) = 20 weights, and 4+2 = 6 biases for a total of 26 learnable parameters.

Neural network architecture 2

FULLY CONNECTED (FC) NEURAL NETWORKS / MULTILAYER PERCEPTRON

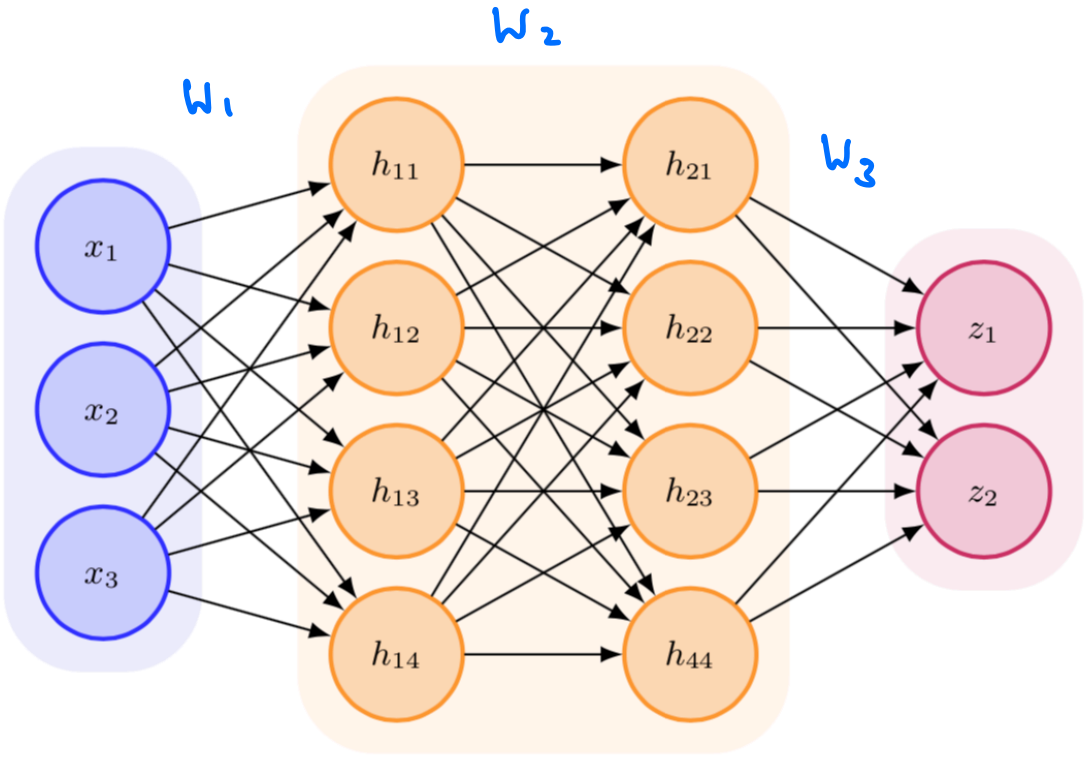
An example 3-layer network is shown below.

$x \in \mathbb{R}^3$ $h_1 \in \mathbb{R}^4$ $h_2 \in \mathbb{R}^4$ $z \in \mathbb{R}^2$



Here, h_{ij} denotes the jth element of $\mathbf{h}_i$. There are many considerations in architecture design, which we will later discuss.

Neural networks

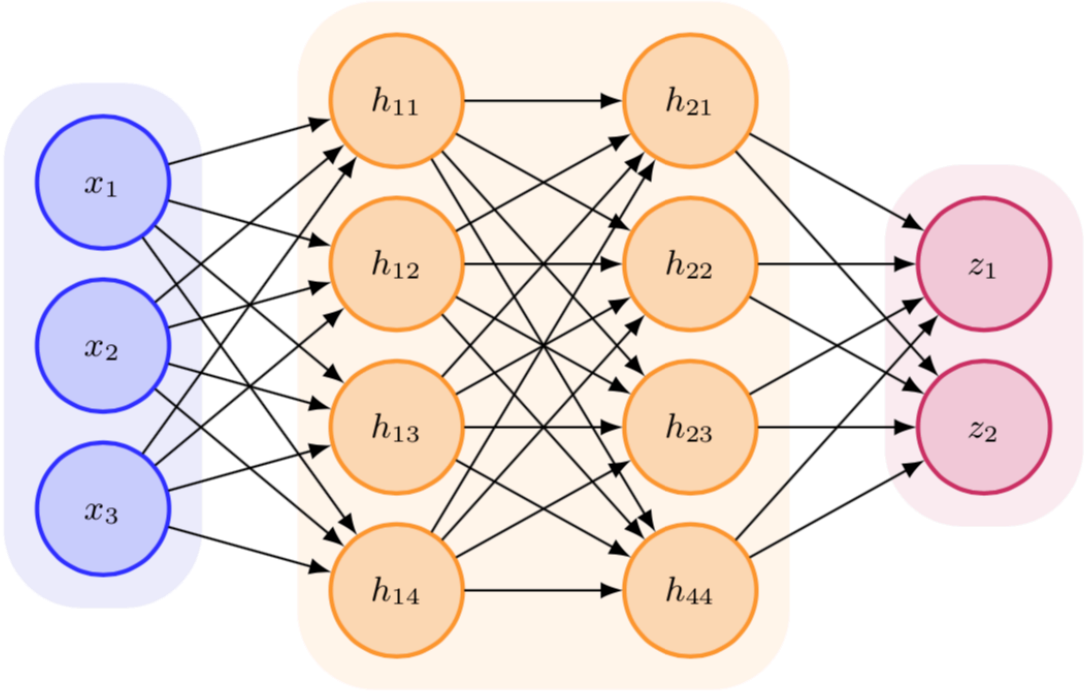


First layer: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

Second layer: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$

Third (output layer): $\mathbf{z} = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3$

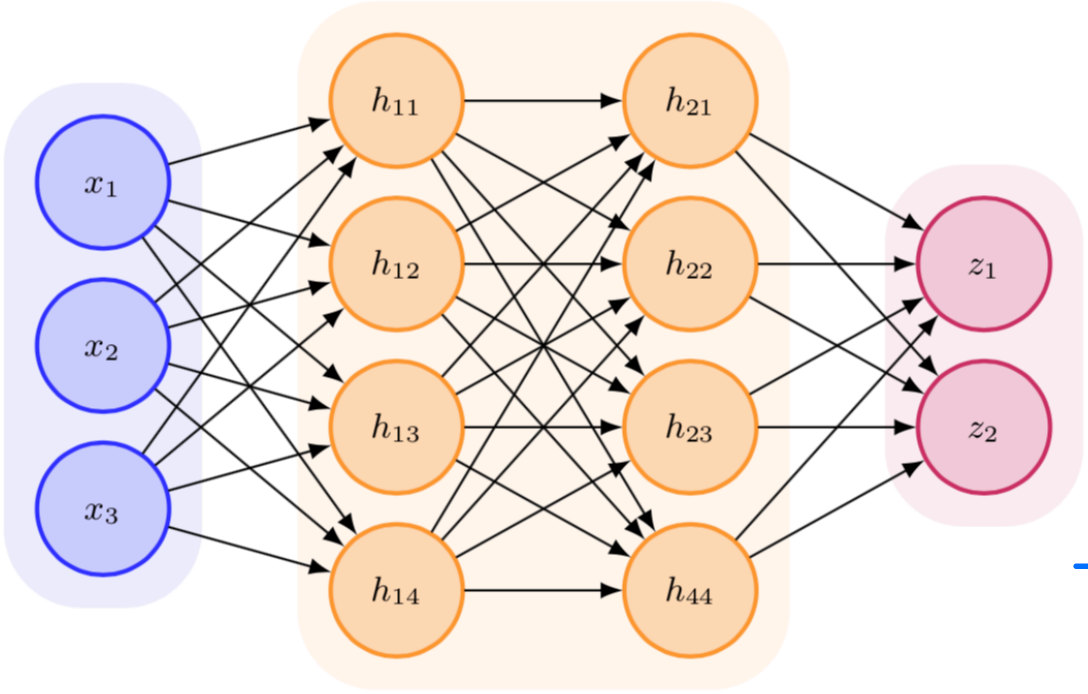
Neural networks

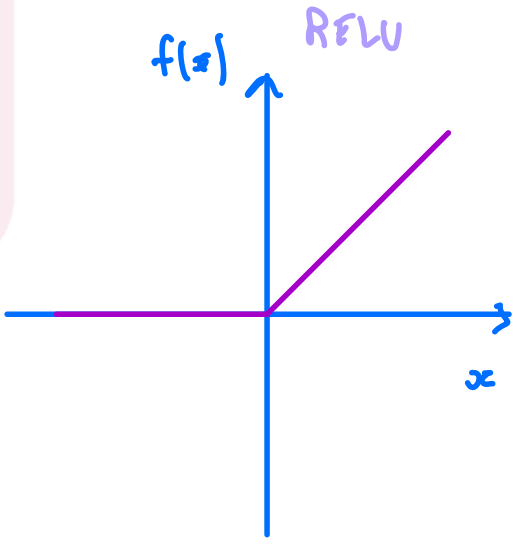


This network has 10 neurons (not counting the input). It has (3x4) + (4x4) + (4x2) = 36 weights, and 4+4+2= 10 biases for a total of 46 learnable parameters.

Perspective: Convolutional neural networks typically have on the order of hundreds of millions of parameters.

Neural networks





$x > 0$ return x

else return 0

```
# Define the activation function
f = lambda x: x * (x > 0)
# Forward pass of a 3-layer network
h1 = f(np.dot(W1, x) + b1)
h2 = f(np.dot(W2, h1) + b2)
z =   np.dot(W3, h2) + b3
```

What if f() is linear?

Neural networks

if $f(x) = x$

The above figure suggests the following equation for a neural network.

- Layer 1: $\mathbf{h}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$
- Layer 2: $\mathbf{h}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

$h_2 = W_2 h_1 + b_2$

$= W_2 (W_1 x + b_1) + b_2$

$= W_2 W_1 x + \underbrace{W_2 b_1 + b_2}_{\tilde{b}} = \tilde{W} x + \tilde{b}$

Any composition of linear functions can be reduced to a single linear function. Here, $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where

$$\mathbf{W} = \mathbf{W}_N \cdots \mathbf{W}_2 \mathbf{W}_1$$

and

$$\mathbf{b} = \mathbf{b}_N + \mathbf{W}_N \mathbf{b}_{N-1} + \cdots + \mathbf{W}_N \cdots \mathbf{W}_3 \mathbf{b}_2 + \mathbf{W}_N \cdots \mathbf{W}_2 \mathbf{b}_1$$

SO, IT WILL BE A LINEAR MODEL //

What if f() is linear?

- This may be useful in some contexts. For example, when $\dim(\mathbf{h}) \ll \dim(\mathbf{x})$, this corresponds to finding a low-rank representation of the inputs.
- However, a system with greater complexity may require a higher capacity model.

USE:

AUTOENCODER:

FIND A 2D REPRESENTATION OF x, SUCH THAT WE CAN GENERATE z FROM h, MINIMIZING $L: \|z-x\|^2$

$x \in \mathbb{R}^4$

$h \in \mathbb{R}^2$

FC

$z \in \mathbb{R}^4$

FOR LINEAR NN,

> 1 HIDDEN LAYERS ARE NOT VERY USEFUL //

Introducing nonlinearity

To increase the network capacity, we can make it nonlinear. We do this by introducing a nonlinearity, $f(\cdot)$, at the output of each artificial neuron.

- Layer 1: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
- Layer 2: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

A few notes:

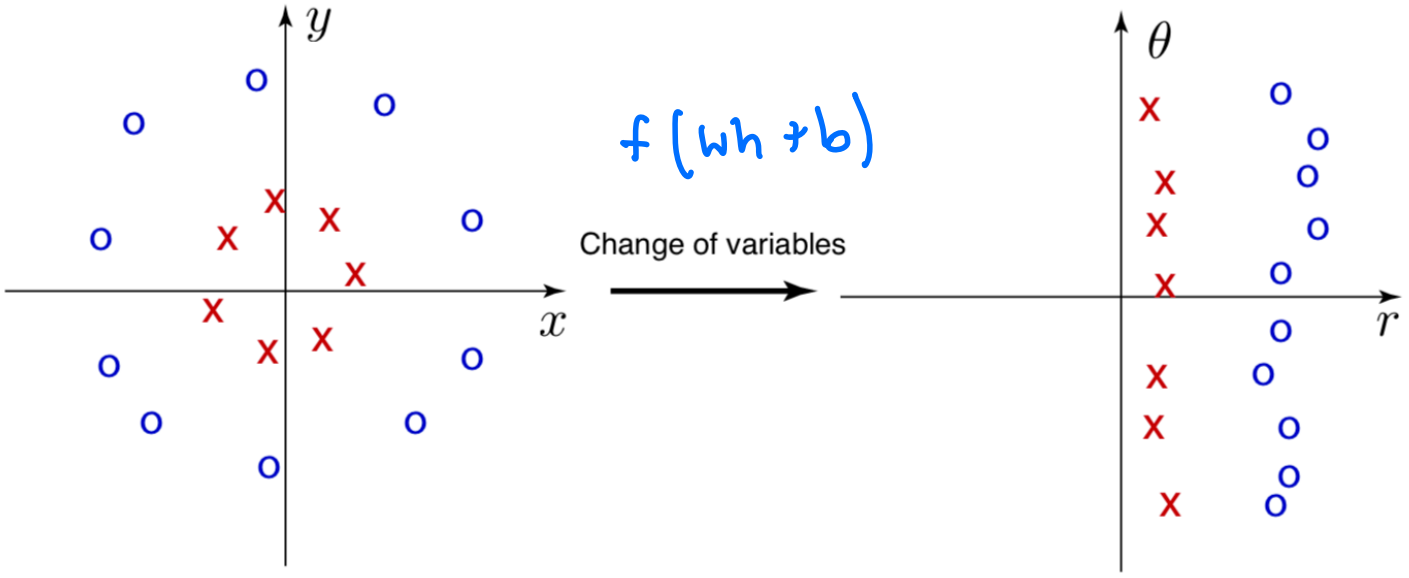
- These equations describe a *feedforward neural network*, also called a *multilayer perceptron*.
- $f(\cdot)$ is typically called an *activation function* and is applied elementwise on its input.
- The activation function does not typically act on the output layer, $\mathbf{z}$, as these are meant to be interpreted as scores. Instead, separate "output activations" are used to process $\mathbf{z}$. While these output activations may be the same as the activation function, they are typically different. For example, it may comprise a softmax or SVM classifier.

↳ can be another g

The hidden layers as learned features

A perspective on feature learning

One area of machine learning is very interested in finding *features* of the data that are then good for use as the input data to a classifier (like a SVM). Why might this be important?



The intermediate layers of the neural network (i.e., $\mathbf{h}_1, \mathbf{h}_2$, etc.) are features that the later layers then use for decoding. If the performance of the neural network is well, these features are good features.

Importantly, these features don't have to be handcrafted.

Example: XOR

Consider a system that produces training data that follows the $\mathrm{xor}(\cdot)$ function. The xor function accepts a 2-dimensional vector $\mathbf{x}$ with components x_1 and x_2 and returns 1 if $x_1 \neq x_2$. Concretely,

x_1	x_2	$\mathrm{xor}(\mathbf{x})$
0	0	0
0	1	1
1	0	1
1	1	0

$$J(\theta) = \frac{1}{2}\sum_{\mathbf{x}}(g(\mathbf{x}) - y(\mathbf{x}))^2$$

(Note, we wouldn't know $\mathrm{xor}(\mathbf{x})$, but we would have samples of corresponding inputs and outputs from training data. Hence, it may be better to simply replace $\mathrm{xor}(\mathbf{x})$ with $y(\mathbf{x})$ representing training examples.)

Example: XOR

Consider first a linear approximation of xor, via $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$. Then,

$$\begin{aligned}\frac{\partial J(\mathbf{w}, b)}{\partial \mathbf{w}} &= \sum_{\mathbf{x}} (\mathbf{w}^T\mathbf{x} + b - y(\mathbf{x}))\mathbf{x} \\ \frac{\partial J(\mathbf{w}, b)}{\partial b} &= \sum_{\mathbf{x}} (\mathbf{w}^T\mathbf{x} + b - y(\mathbf{x}))\end{aligned}$$

Equating these to 0, we arrive at:

$$(w_1 + b - 1)\begin{bmatrix} 1 \\ 0 \end{bmatrix} + (w_2 + b - 1)\begin{bmatrix} 0 \\ 1 \end{bmatrix} + (w_1 + w_2 + b)\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

These two equations can be simplified as:

$$\begin{aligned}(w_1 + b - 1) + (w_1 + w_2 + b) &= 0 \\ (w_2 + b - 1) + (w_1 + w_2 + b) &= 0\end{aligned}$$

These equations are symmetric, implying $w_1 = w_2 = w$. This means:

$$3w + 2b - 1 = 0 \implies b = \frac{1 - 3w}{2}$$

Example: XOR

Now let's consider using a two-layer neural network, with the following equation:

$$g(\mathbf{x}) = \mathbf{w}^T \max(0, \mathbf{W}^T \mathbf{x} + \mathbf{c}) + b$$

We haven't yet discussed how to optimize these parameters, but the point here is to show that by introducing a simple nonlinearity like $f(x) = \max(0, x)$, we can now solve the $\text{xor}(\cdot)$ problem. Consider the solution:

$$\begin{aligned} \mathbf{W} &= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \\ \mathbf{c} &= [0, -1]^T \\ \mathbf{w} &= [1, -2]^T \end{aligned}$$

What nonlinearity to use?

There are a variety of activation functions. We'll discuss some more commonly encountered ones.

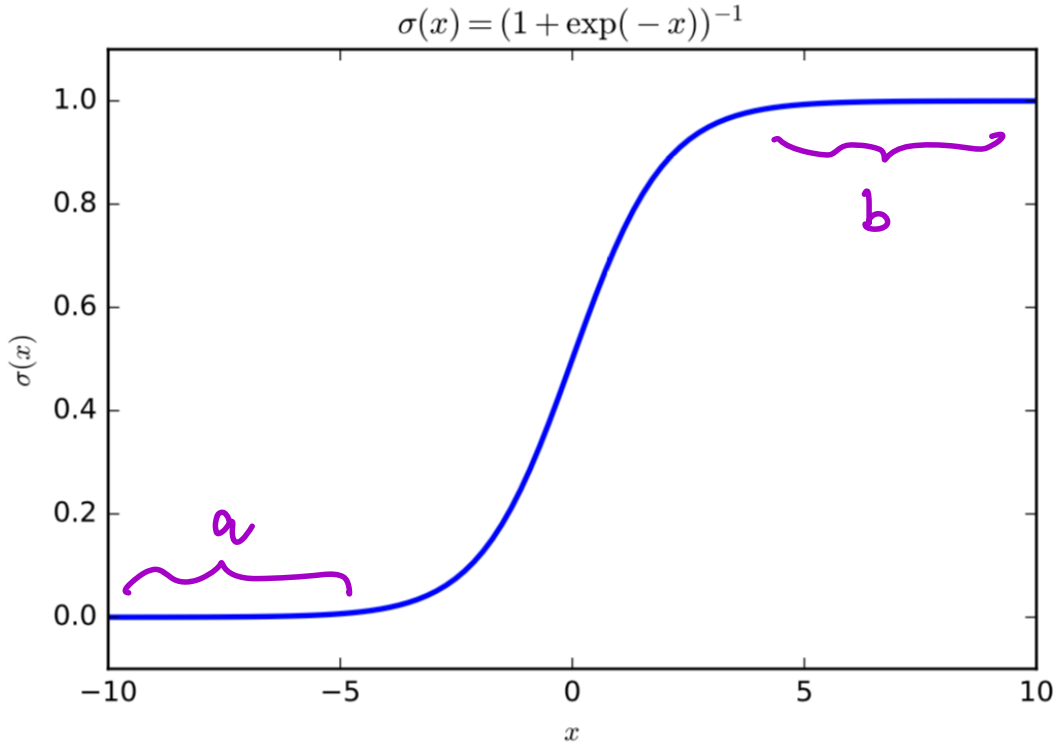
How to choose $f(\cdot)$?

"One recurring theme throughout NN design is that the gradient of the cost function(L) must be large and predictable enough to serve as a good guide for the learning algorithm"

Sigmoid unit

Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$

```
f = lambda x: 1.0 / (1.0 + np.exp(-x))
```



Its derivative is:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Loss Function, $L(w)$

$$\sigma(w) = \sigma(w^T x + b)$$

$$\frac{\partial L(w)}{\partial w} = \underbrace{\frac{\partial \sigma(w)}{\partial w}}_{\text{a small \# in region } a,b} \cdot \frac{\partial L(w)}{\partial \sigma(w)}$$

$$w \leftarrow w - \varepsilon \frac{\partial L(w)}{\partial w}$$

↳ small and hence w does not learn.

"One recurring theme throughout NN design is that the gradient of the cost function(L) must be large and predictable enough to serve as a good guide for the learning algorithm"

Sigmoid unit

Sigmoid activation, $\sigma(x) = \frac{1}{1+\exp(-x)}$

Pros:

- Around $x = 0$, the unit behaves linearly. $\sigma(x) \approx x$ $\frac{\partial \sigma}{\partial x} \approx 1$ around $x=0$
- It is differentiable everywhere.

Cons:

activation function

- At extremes, the unit *saturates* and thus has zero gradient. This results in no learning with gradient descent.

activation fn ≥ 0

- The sigmoid unit is not zero-centered; rather its outputs are centered at 0.5.

  Consider $f(\mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x} + b)$. The derivative with respect to $\mathbf{w}$, the parameters, is:

  Defining $z = \mathbf{w}^T h_1 + b$ (h_1 = $\mathbf{x}$)

$$\frac{\partial f(\mathbf{w})}{\partial \mathbf{w}} = \sigma(z)(1-\sigma(z))\mathbf{x}$$

  If $\mathbf{x} \geq 0$ (e.g., if the prior unit had a sigmoidal output), then the gradient has all positive entries or all negative entries. Lets say we had some gradient $\frac{\partial L}{\partial f}$, which can be +ve/-ve. Then $\partial L/\partial w$ will have all +ve/-ve entries. This can result in zig-zagging during gradient descent.

- As a result, the sigmoid activation is rarely used.

$x \xrightarrow{w,\sigma} h_1 \xrightarrow{w} z$

$$f(w) = \sigma(\underbrace{w^T h_1 + b}_{z})$$

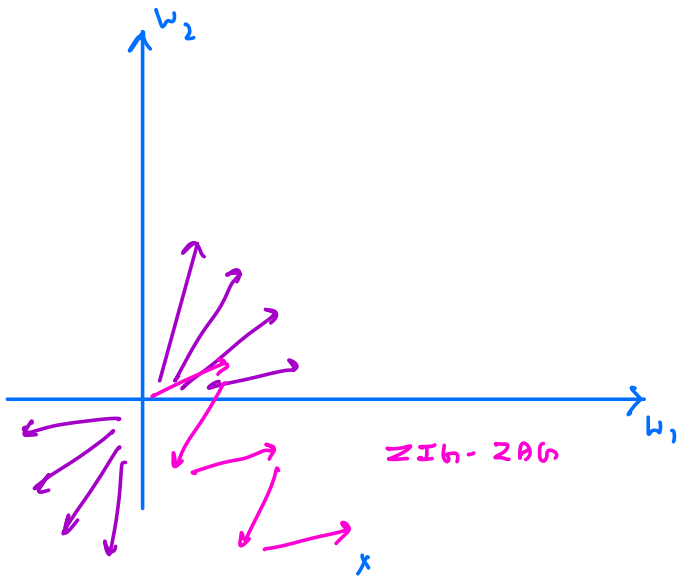
$$z = w^T h_1 + b$$

$$\frac{\partial f(w)}{\partial w} = \underbrace{\sigma(z)}_{\geq 0}\underbrace{(1-\sigma(z))}_{\geq 0}\underbrace{h_1}_{\geq 0} = \frac{\partial z}{\partial w} \cdot \frac{\partial f}{\partial z}$$

$$\frac{\partial L}{\partial w} = \underbrace{\frac{\partial f(w)}{\partial w}}_{\geq 0} \cdot \frac{\partial L}{\partial f(w)}$$

↳ is scalar, ≥ 0 or ≤ 0

so all $\frac{\partial L}{\partial w} \geq 0$ or ≤ 0.



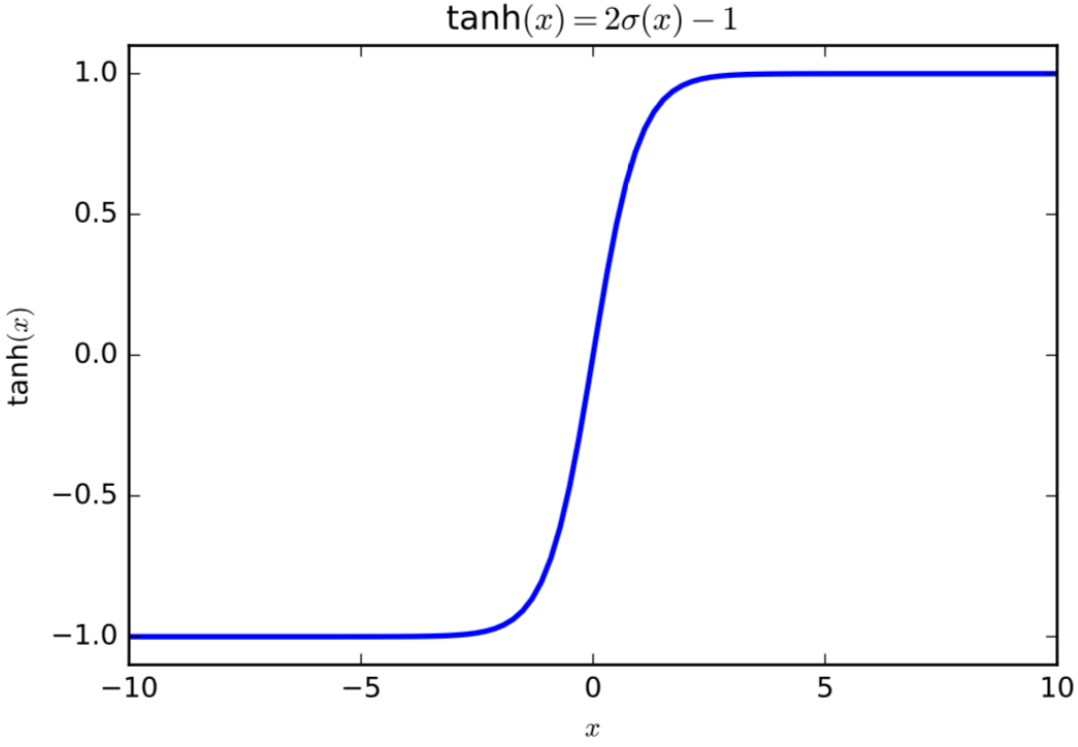
↳ assume optimal w

Hyperbolic tangent

Hyperbolic tangent, $\tanh(x) = 2\sigma(x) - 1$

The hyperbolic tangent is a zero-centered sigmoid-looking activation.

```
f = lambda x: np.tanh(x)
```



Its derivative is:

$$\frac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$$

Hyperbolic tangent

Hyperbolic tangent, $\tanh(x) = 2\sigma(x) - 1$

Pros:

- Around $x = 0$, the unit behaves linearly.
- It is differentiable everywhere.
- It is zero-centered, fixing a major flaw of the sigmoid activation.

Cons:

- Like the sigmoid unit, when a unit saturates, i.e., its values grow larger or smaller, the unit saturates and no additional learning occurs.
- The gradient is subject to second order effects; as $|x|$ grows, the gradient decreases to zero. This makes the gradient smaller at larger $|x|$, which may distort the gradient.

ReLU unit

Hyperbolic tangent, $\tanh(x) = 2\sigma(x) - 1$

Pros:

- Around $x = 0$, the unit behaves linearly.
- It is differentiable everywhere.
- It is zero-centered, fixing a major flaw of the sigmoid activation.

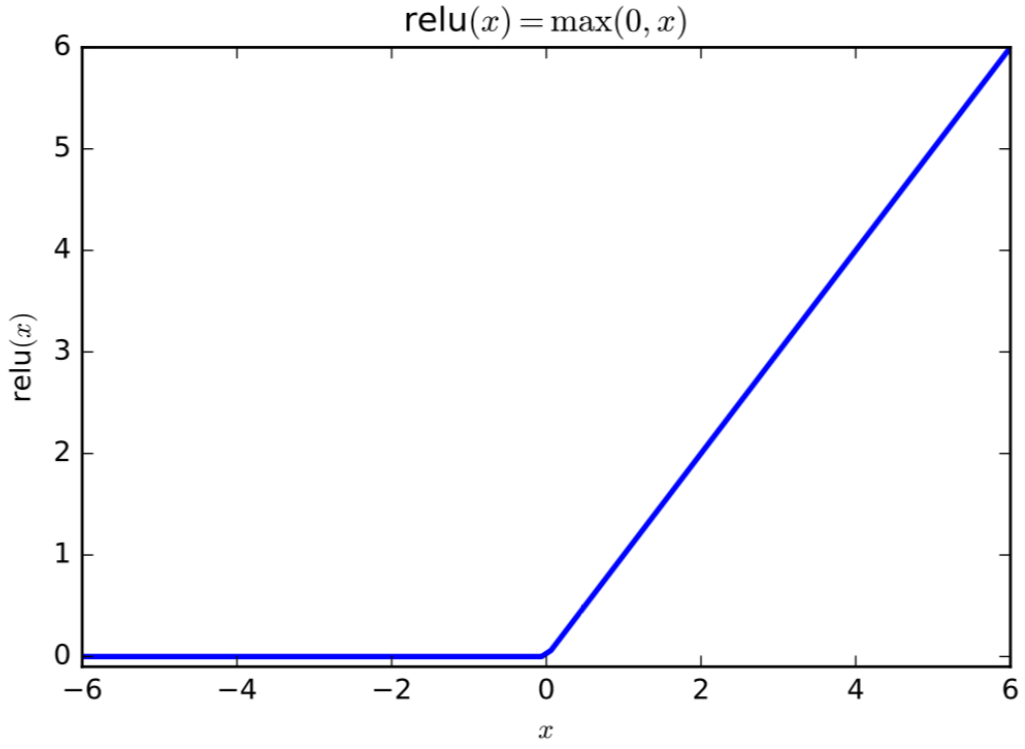
Cons:

- Like the sigmoid unit, when a unit saturates, i.e., its values grow larger or smaller, the unit saturates and no additional learning occurs.
- The gradient is subject to second order effects; as $|x|$ grows, the gradient decreases to zero. This makes the gradient smaller at larger $|x|$, which may distort the gradient.

ReLU unit

Rectified linear unit, $\mathbf{ReLU}(x) = \max(0, x)$

```
f = lambda x: x * (x > 0)
```



Its derivative is:

$$\frac{d\mathrm{ReLU}(x)}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

This function is not differentiable at $x = 0$. However, we can define its subgradient by setting the derivative to be between $[0, 1]$ at $x = 0$.

ReLU unit

Rectified linear unit, $\mathbf{ReLU}(x) = \max(0, x)$

Pros:

- In practice, learning with the ReLU unit converges faster than sigmoid and tanh. AlexNet ReLU was 6x faster than tanh.
- When a unit is active, it behaves as a linear unit. → gradient = 1 (not small)
- The derivative at all points, except $x = 0$, is 0 or 1. When $x > 0$, the gradients are large, and not scaled by second order effects. → input ≥ 0
- There is no saturation if $x > 0$.

Cons:

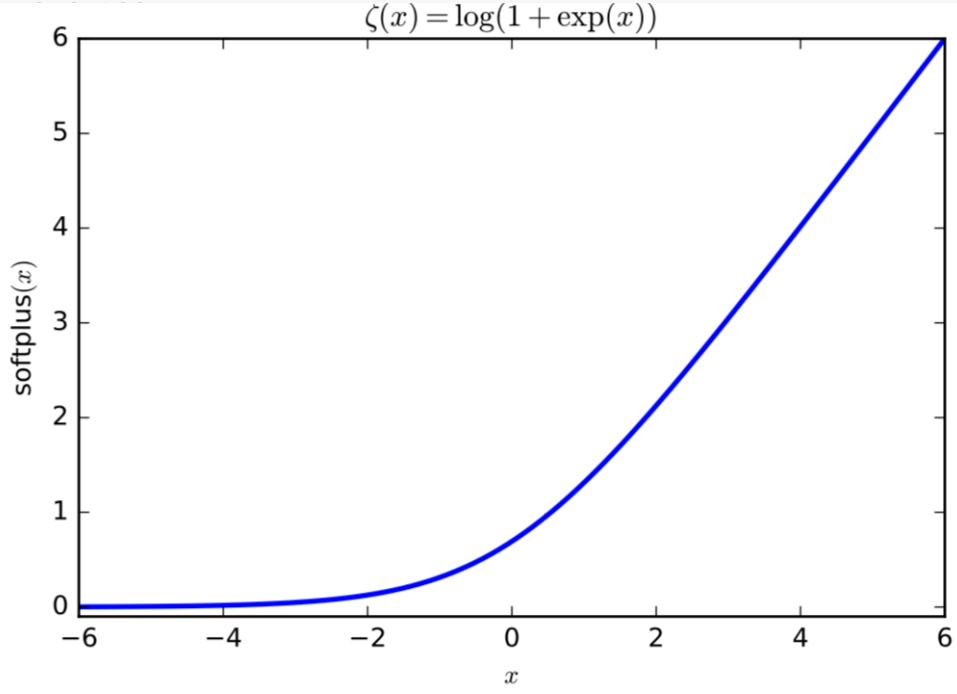
- $\mathrm{ReLU}(x)$, like sigmoid, is not zero-centered. non-negative (it zig-zags)
- $\mathrm{ReLU}(x)$ is not differentiable at $x = 0$. However, in practice, this is not a large issue. A heuristic when evaluating $\frac{d\mathrm{ReLU}(x)}{dx}\Big|_{x=0}$ is to return the left derivative (0) or the right derivative (1); this is reasonable given digital computation is subject to numerical error.
- Learning does not happen for examples that have zero activation. This can be fixed by e.g., using a leaky ReLU or maxout unit.

Softplus unit

Softplus unit, $\zeta(x) = \log(1 + \exp(x))$

One may consider using the softplus function, $\zeta(x) = \log(1 + e^x)$, in place of $\mathrm{ReLU}(x)$. Intuitively, this ought to work well as it resembles $\mathrm{ReLU}(x)$ and is differentiable everywhere. However, empirically, it performs worse than $\mathrm{ReLU}(x)$.

```
f = lambda x: np.log(1+np.exp(x))
```



Its derivative is:

$$\frac{d\zeta(x)}{dx} = \sigma(x)$$

Softplus unit

"One might expect it to have an advantage over the [ReLU] due to being differentiable everywhere or due to saturating less completely, but empirically it does not." (Goodfellow et al., p. 191)

$x < 0, \ \partial \ell / \partial x \neq 0$

Neuron	MNIST	CIFAR10	NISTP	NORB
***With* unsupervised pre-training**				
Rectifier	**1.20%**	**49.96%**	**32.86%**	**16.46%**
Tanh	**1.16%**	**50.79%**	35.89%	17.66%
Softplus	**1.17%**	**49.52%**	**33.27%**	19.19%
***Without* unsupervised pre-training**				
Rectifier	**1.43%**	**50.86%**	**32.64%**	**16.40%**
Tanh	1.57%	52.62%	36.46%	19.29%
Softplus	1.77%	53.20%	35.48%	17.68%

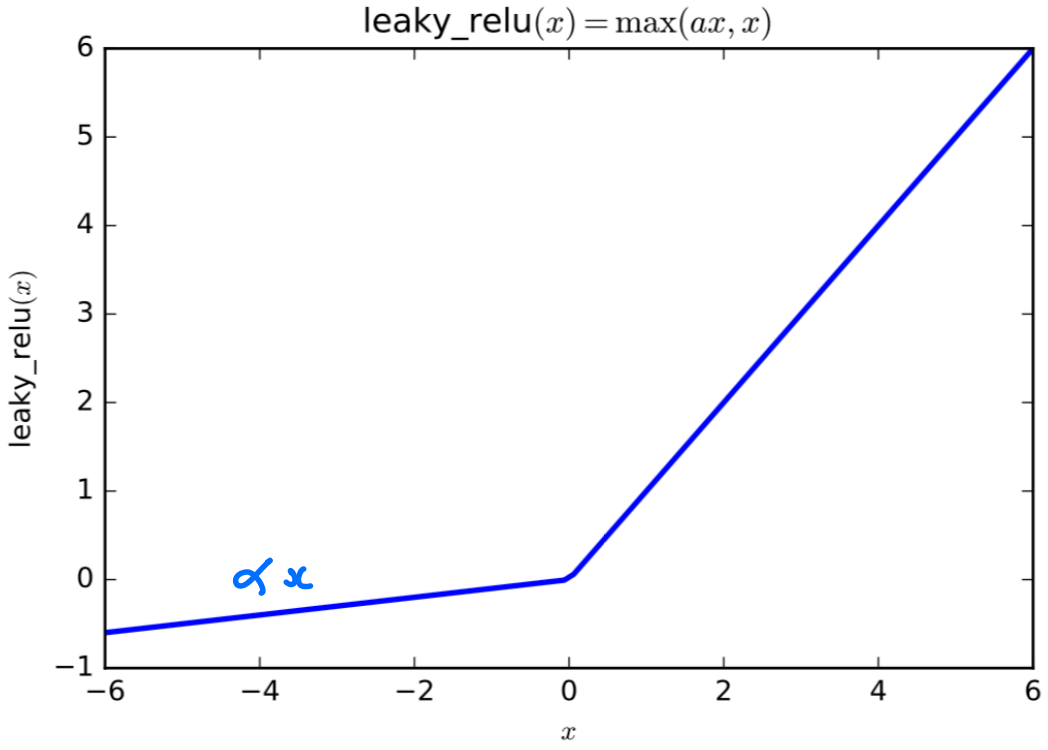
ReLU

Glorot et al., 2011a

Leaky ReLU / PReLU unit

Leaky rectified linear unit, $f(x) = \max(\alpha x, x)$

```
f = lambda x: x * (x>0) + 0.1*x * (x<0)
```



$\alpha \approx 0.01 / 0.02$ usually

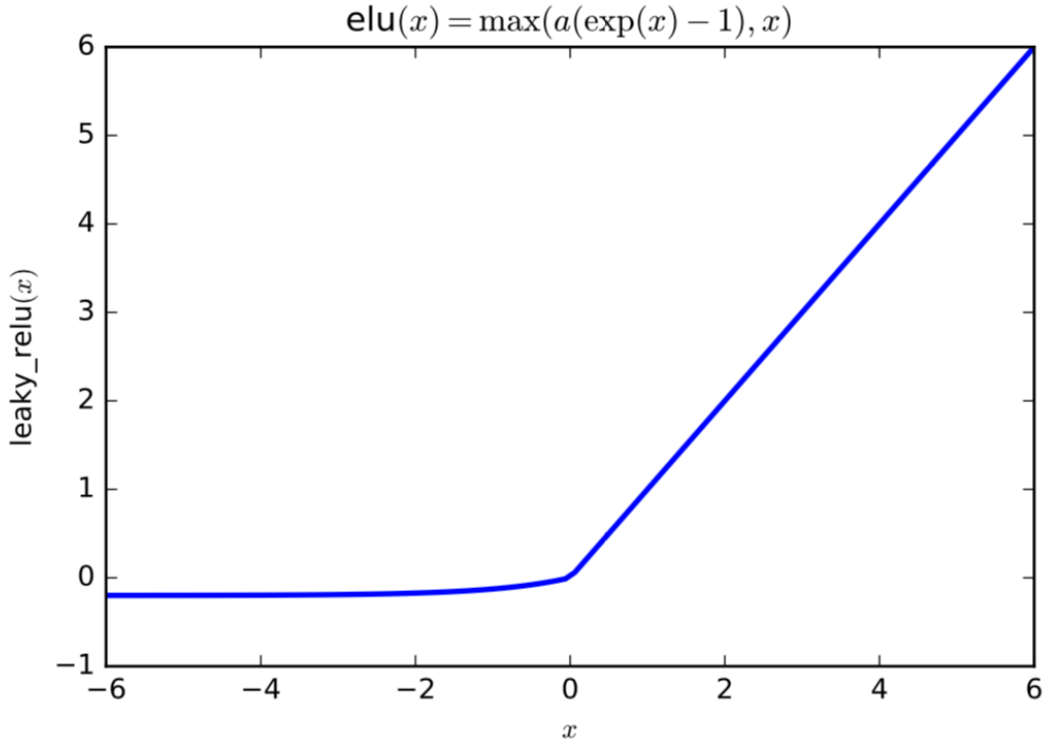
The leaky ReLU avoids the stopping of learning when $x < 0$. α may be treated as a selected hyperparameter, or it may be a parameter to be optimized in learning, in which case it is typically called the "PReLU" for parametrized rectified linear unit.

↓ parameterized

ELU unit

Exponential linear unit, $f(x) = \max(\alpha(\exp(x) - 1), x)$

```
f = lambda x: x * (x>0) + 0.2*(np.exp(x) - 1) * (x<0)
```



The exponential linear unit avoids the stopping of learning when $x < 0$. A con of this activation function is that it requires computation of the exponential, which is more expensive.

Which LU do I use?

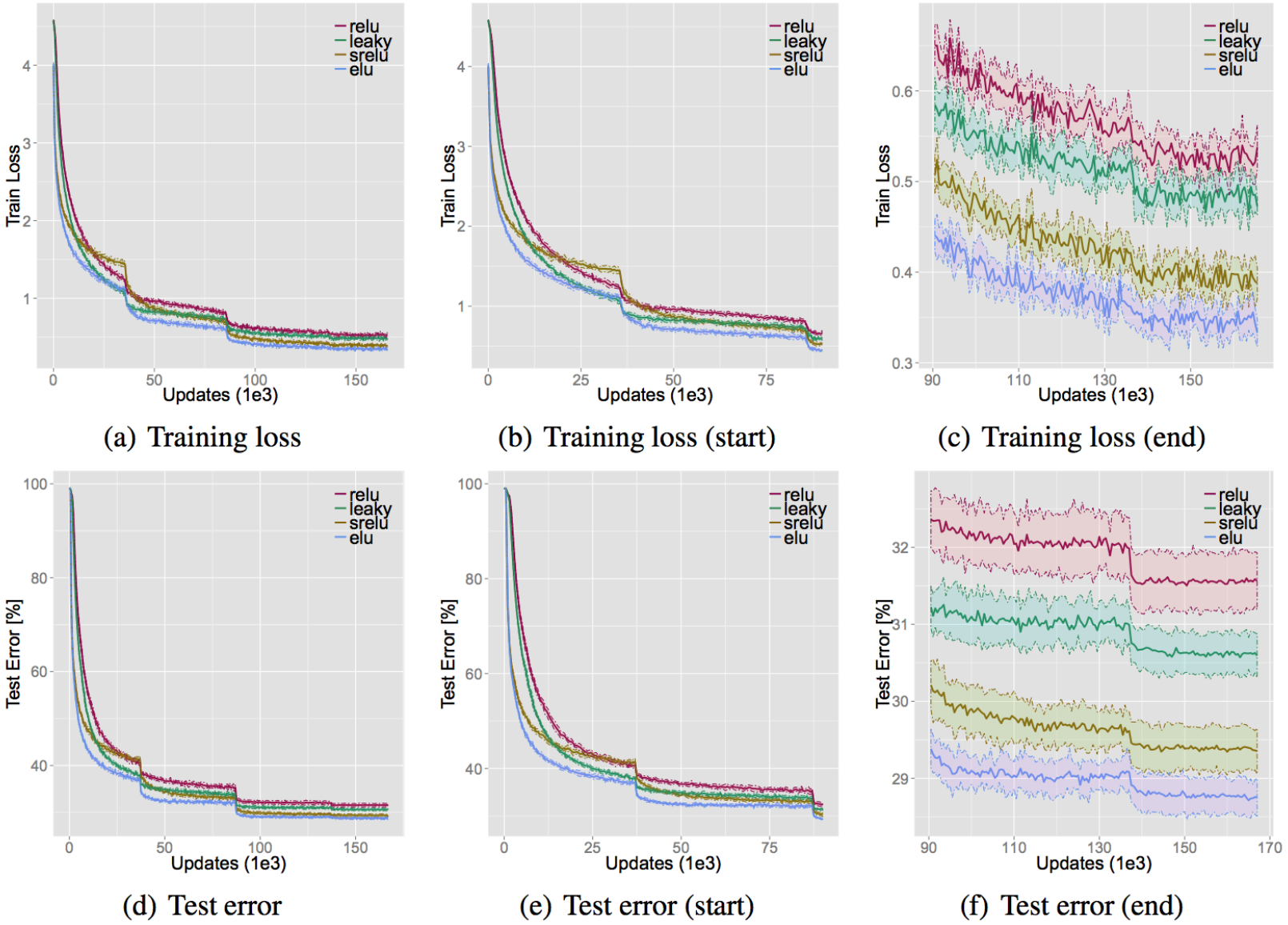


Figure 4: Comparison of ReLUs, LReLUs, and SReLUs on CIFAR-100. Panels (a-c) show the

Clevert et al., 2015

Maxout unit

A generalization of the ReLU and PReLU units is the maxout unit, where:

$$\text{maxout}(\mathbf{x}) = \max(\mathbf{w}_1^T \mathbf{x} + b_1, \mathbf{w}_2^T \mathbf{x} + b_2)$$

This can be generalized to more than two components. If $\mathbf{w}_1 = \mathbf{0}$ and $b_1 = 0$, this is the rectified linear unit.

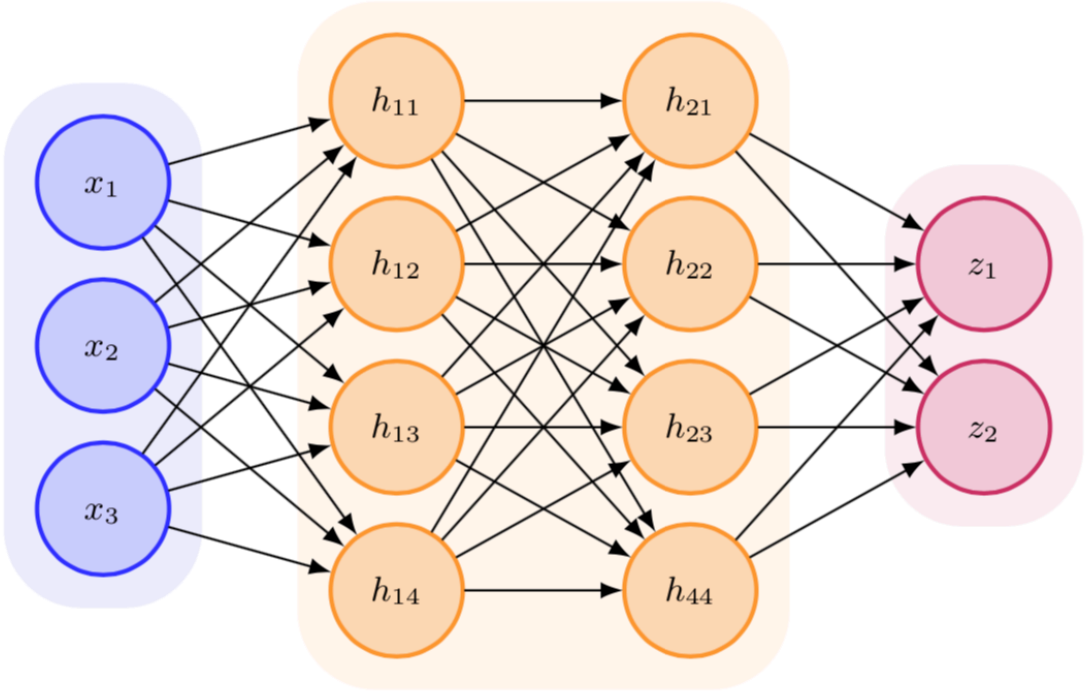
What activation function do I use?

In practice...

In practice...

- The ReLU unit is very popular.
- The sigmoid unit is ~~almost never~~ rarely used; tanh is preferred.
- It may be worth trying out leaky ReLU / PReLU / ELU / maxout for your application.
- This is an active area of research.

Back to NN architecture



$y = g(z)$

SOFTMAX CLASSIFIER → Classify

- Layer 1: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
- Layer 2: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$
- $\vdots$
- Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$

What output activation do we use?

↳ g

The output activation interacts with the cost function

What outputs and cost functions?

There are several options to process the output scores, $\mathbf{z}$, to ultimately arrive at a cost function. The choice of output units interacts with what cost function to use.

BINARY CLASSIFICATION

Example: Consider a neural network that produces a single score, z, for binary classification. As the output unit, we choose the sigmoid nonlinearity, so that $\hat{y} = \sigma(z)$. On a given example, $y^{(i)}$ is either 0 or 1, and $\hat{y}^{(i)} = \sigma(z^{(i)})$ can be interpreted as the algorithm's probability the output is in class 1. Is it better to use mean-square error or cross-entropy (i.e., corresponding to maximum-likelihood estimation) as the cost function? For n examples:

$$\text{MSE} = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)^2$$

CROSS-ENTROPY

$$\text{CE} = -\sum_{i=1}^{n}\left[y^{(i)}\log\sigma(z^{(i)}) + (1-y^{(i)})\log(1-\sigma(z^{(i)}))\right]$$

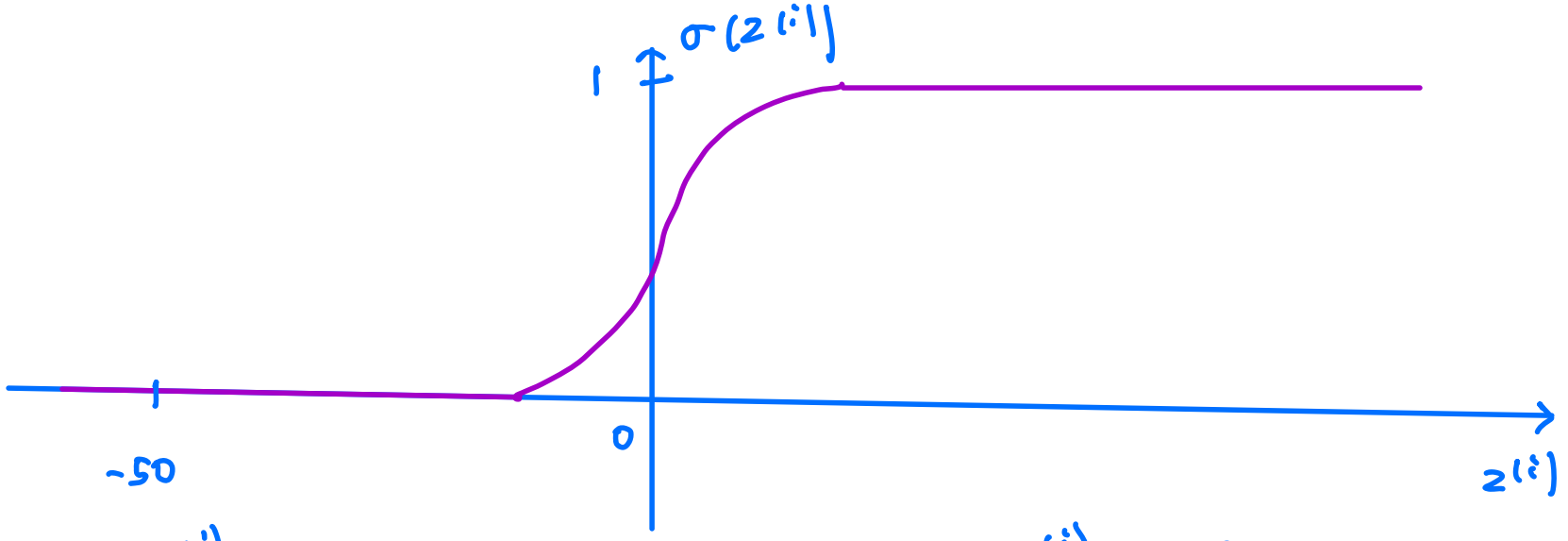
large z → class 1, $y^{(i)} = 1$

negative z → class 0, $y^{(i)} = 0$

$\sigma(z) = \Pr\{x^{(i)} \text{ is in class } 1\}$

A picture for intuition

Cross Entropy Error is same as the max-likelihood loss that we derived for softmax.



if $y^{(i)} = 1$ and we start at $z^{(i)} = -50$

MSE for example $i = (y^{(i)} - \sigma(z^{(i)}))^2$

$$\frac{\partial MSE_i}{\partial z^{(i)}} = \frac{\partial \sigma(z^{(i)})}{\partial (z^{(i)})} \frac{\partial MSE_i}{\partial \sigma(z^{(i)})}$$

$\partial(z^{(i)}) = 0$ when $z^{(i)} = -50$ No learning.

Why do we not learn?

MSE at -50 is $\|y^{(i)} - \sigma(-50)\|^2 \approx 1$

MSE at -40 is also $\|y^{(i)} - \sigma(-40)\|^2 \approx 1$

So there is no much difference in MSE, so it does not take a step from -50 to -40.

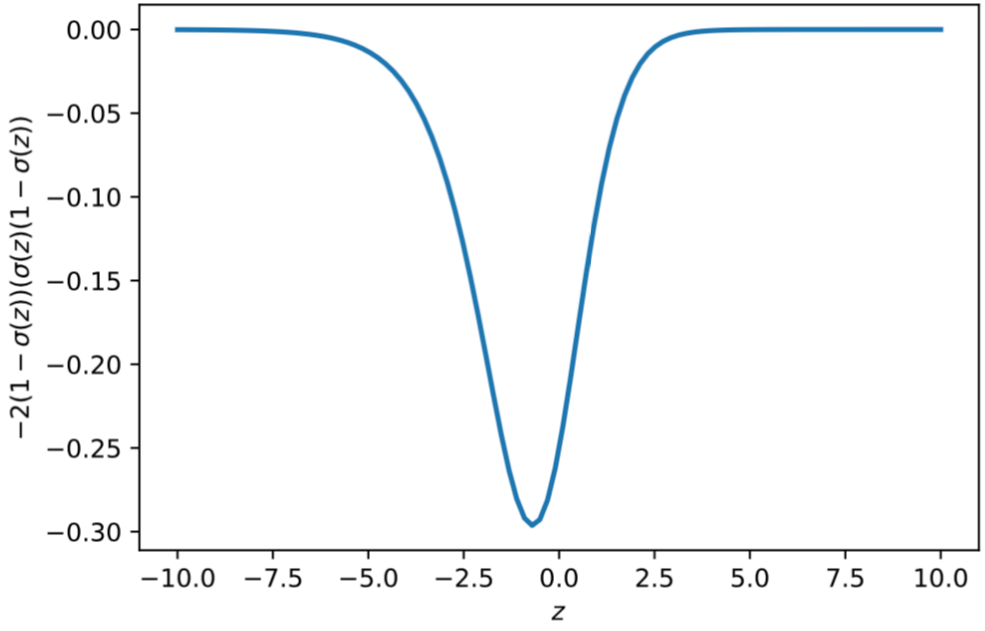
The output activation interacts with the cost function

What outputs and cost functions? (cont.)

Example (cont): Consider just one example, where $y^{(i)} = 1$. For this example,

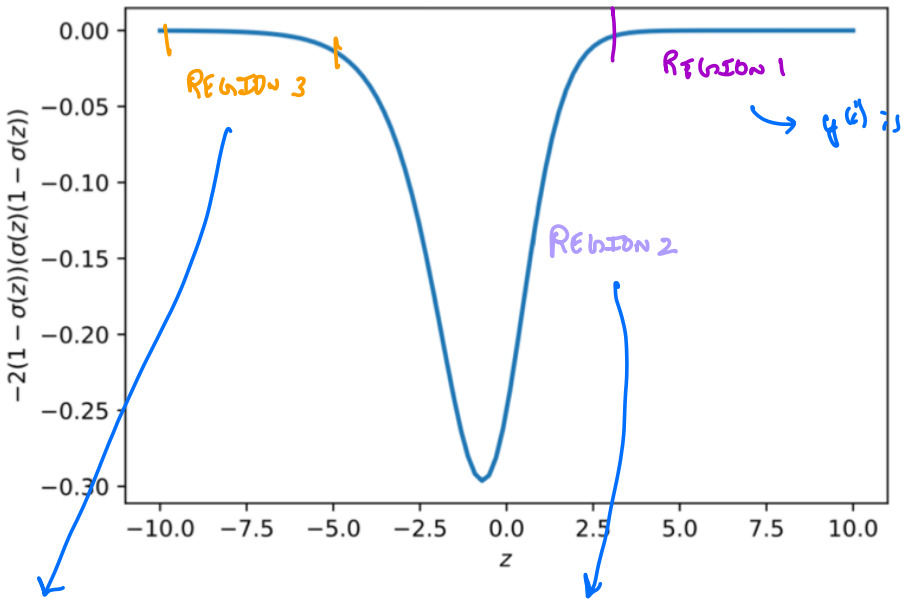
$$\frac{\partial \text{MSE}}{\partial z^{(i)}} = -2(y^{(i)} - \sigma(z^{(i)}))(\sigma(z^{(i)})(1 - \sigma(z^{(i)})))$$

This derivative looks like the following:



When z is very negative, indicating a large MSE, the gradient saturates to zero, and no learning occurs.

$y^{(i)} = 1$



→ $y^{(i)}$ is 1 and $z^{(i)}$ is indeed correct.

so we don't have to learn, slope is less and so its correct

No learning as grad ≈ 0.

Very bad!

we need to reach the $y^{(i)} = 1$

we want to learn and increase the $z^{(i)}$ as $y^{(i)}$ is 1. So the slope > 0 and big.

So it is good //

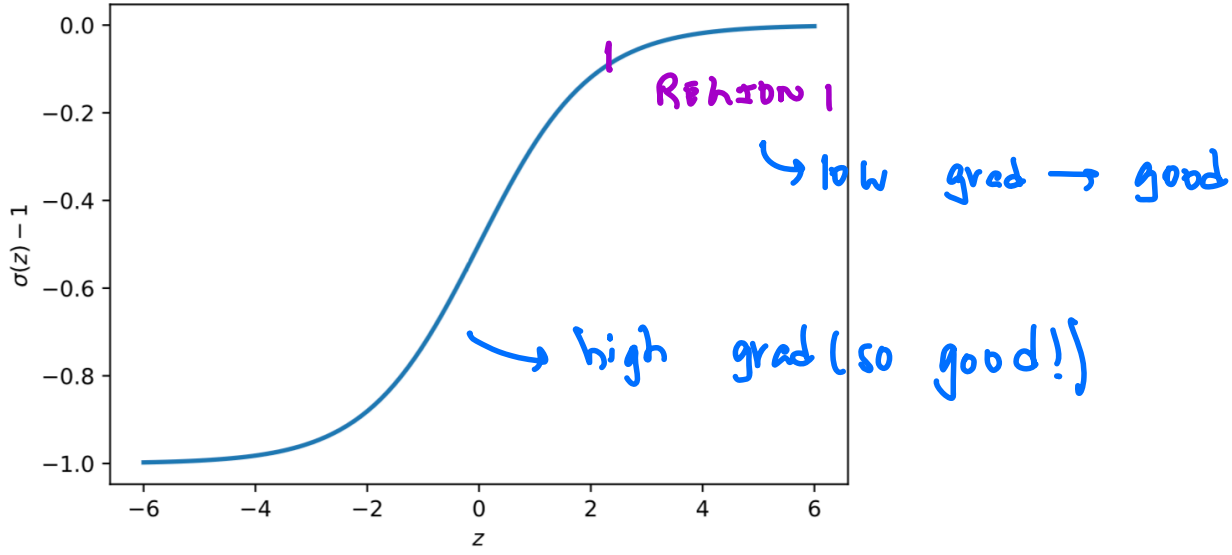
The output activation interacts with the cost function

What outputs and cost functions? (cont.)

Example (cont): Now let's consider the cross-entropy. For one example, where $y^{(i)} = 1$,

$$\frac{\partial \text{CE}}{\partial z^{(i)}} = \sigma(z^{(i)}) - 1$$

This derivative looks like the following:



Notice that when z is very negative, learning will occur, and it will only "stall" when z gets close to the right answer.

Output activations

- Linear output units: $\hat{\mathbf{y}} = \mathbf{z}$.

  These output units typically specify the conditional mean of a Gaussian distribution, i.e.,

  $$p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{z}, \mathbf{I})$$

  and in this case, MLE estimation is equivalent to minimizing squared error.

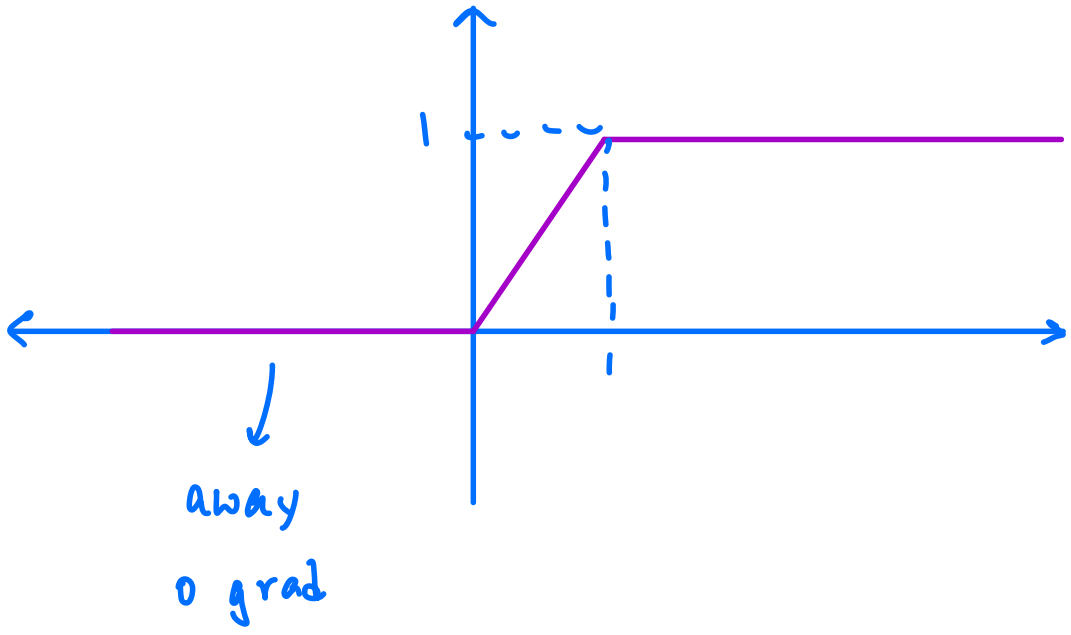
Output activations

- Sigmoid outputs: $\hat{\mathbf{y}} = \sigma(\mathbf{z})$.

  These outputs are typically used in binary classification to approximate a Bernoulli distribution.

  Question: Why not use the following output?

$$\Pr(y = 1|\mathbf{x}) = \max(0, \min(1, \mathbf{z}))$$



Output activations

- Softmax output: $\hat{\mathbf{y}}_i = \text{softmax}_i(\mathbf{z})$.

  The softmax is the generalization of the sigmoid output to multiple classes.

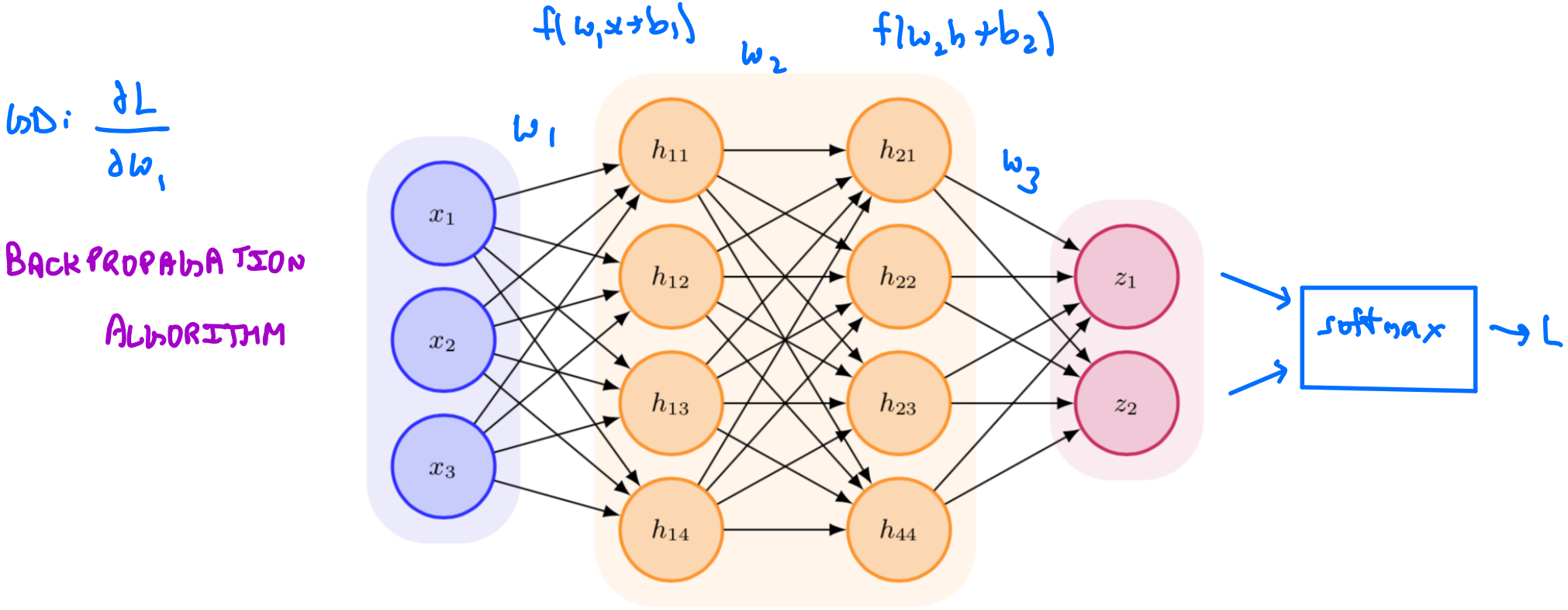
A softmax output activation is fairly common.

What next?

Now that we've defined all the elements of the neural network, the question now becomes: how do we learn its parameters?

The short answer is that we use versions of gradient descent.

However, neural networks architectures have units that are several layers from the output. In these scenarios, how do we arrive at the gradient?

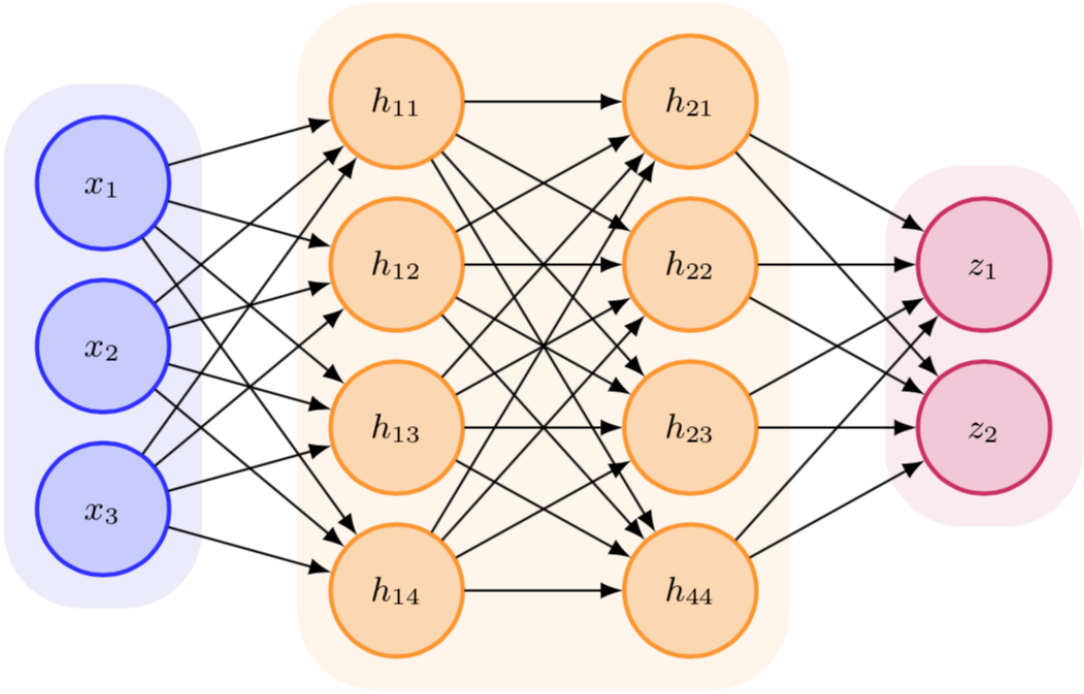


GD: $\frac{\partial L}{\partial w_1}$

BACKPROPAGATION ALGORITHM

Backpropagation

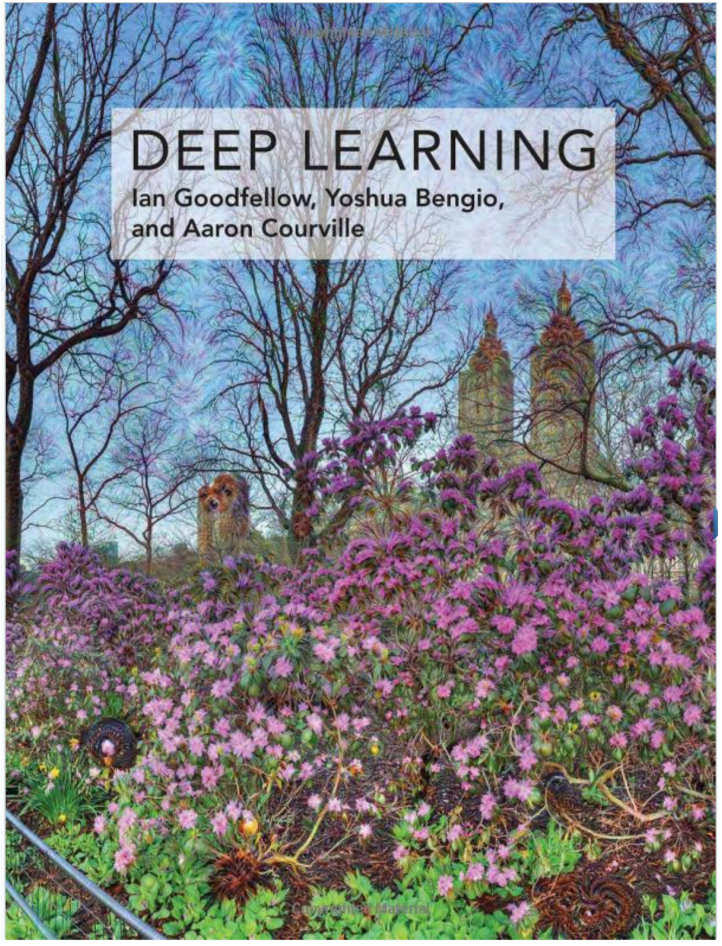
In this lecture, we'll introduce backpropagation as a technique to calculate the gradient of the loss function with respect to parameters in a neural network.



Reading

Reading:

Deep Learning, 6.5-6.6



Backpropagation

In the most intuitive sense, backpropagation is an application of the chain rule for derivatives.

Motivation for backpropagation

To do gradient descent, we need to calculate the gradient of the objective with respect to the parameters, θ. However, in a neural network, some parameters are not directly connected to the output. How do we calculate then the gradient of the objective with respect to these parameters? Backpropagation answers this question, and is a general application of the chain rule for derivatives.

Backpropagation

Nomenclature

Forward propagation: $h = f(w_1 x + b_1)$ $z = w_2 h + b_2$

- Forward propagation is the act of calculating the values of the hidden and output units of the neural network given an input.
- It involves taking input $\mathbf{x}$, propagating it to through each hidden unit sequentially, until you arrive at the output $\mathbf{y}$. From forward propagation, we can also calculate the cost function $J(\theta)$.
- In this manner, the forward propagated signals are the activations.
- With the input as the "start" and the output as the "end," information propagates in a forward manner.

Backpropagation (colloquially called backprop): $\partial L/\partial z \rightarrow \partial L/\partial w_2 \rightarrow \partial L/\partial w_1$

- As its name suggests, now information is passed backward through the network, from the cost function and outputs to the inputs.
- The signal that is backpropagated are the gradients.
- It enables the calculation of gradients at every stage going back to the input layer.

Backpropagation

Why do we need backpropagation?

It is possible in many cases to calculate an analytical gradient. So why use backpropagation?

- Evaluating analytical gradients may be computationally expensive.
- Backpropagation is generalizable and is often inexpensive.

A few further notes on backpropagation.

- Backpropagation is not the learning algorithm. It's the method of computing gradients.
- Backpropagation is not specific to multilayer neural networks, but a general way to compute derivatives of functions.

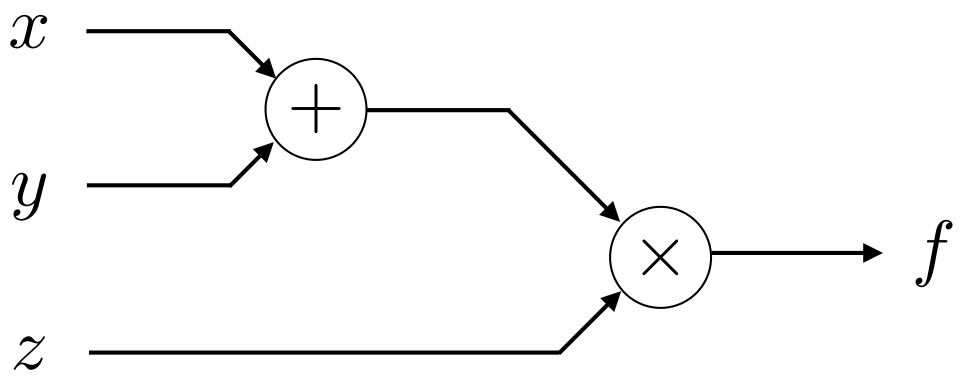
A simple example

$$f(x, y, z) = (x + y)z$$

$\partial f / \partial z = x + y$

$\partial f / \partial x = \partial f / \partial y = z$

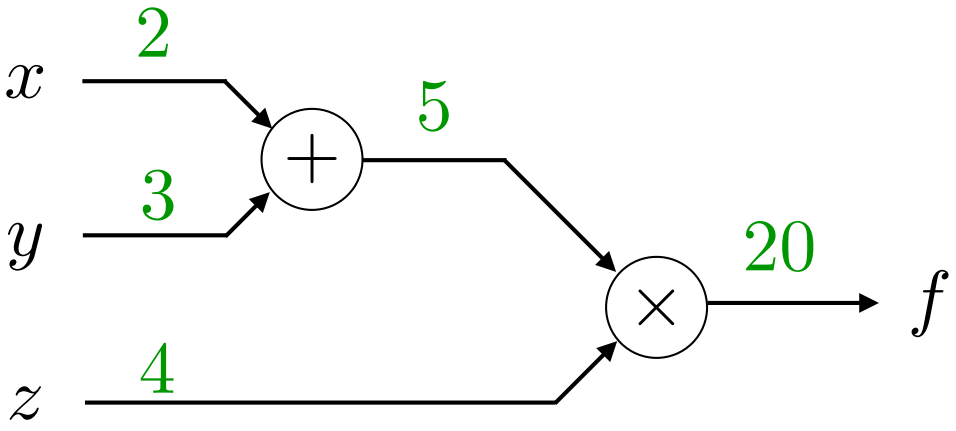
COMPUTATIONAL GRAPH:



A simple example

$$f(x, y, z) = (x + y)z$$

Forward propagation:

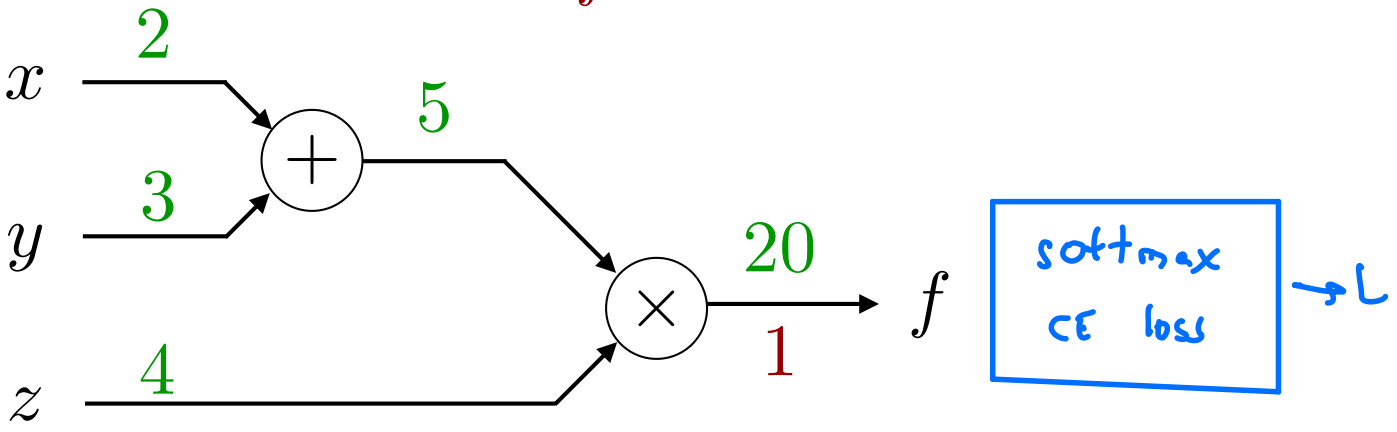


A simple example

$$f(x, y, z) = (x + y)z$$

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial f} = 1$$

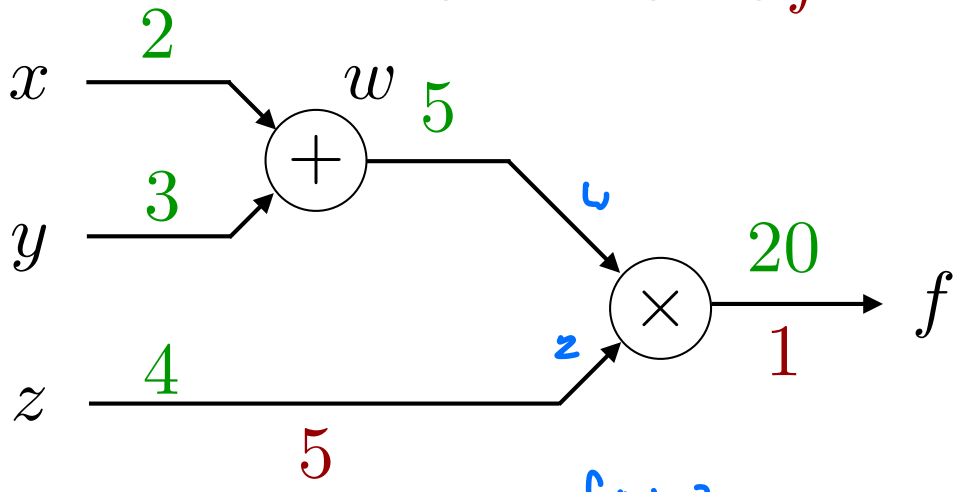


A simple example

$$f(x, y, z) = (x + y)z$$

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial f}{\partial z}\frac{\partial \mathcal{L}}{\partial f}$$



$f = wz$

$\frac{\partial f}{\partial z} = w$

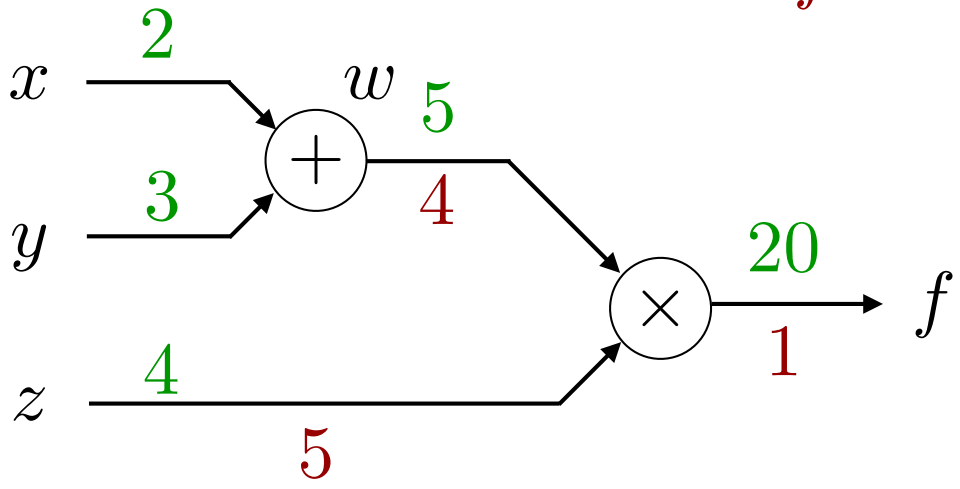
$\frac{\partial L}{\partial z} = w \cdot 1 = 5$

A simple example

$$f(x, y, z) = (x + y)z$$

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial f}{\partial w}\frac{\partial \mathcal{L}}{\partial f}$$



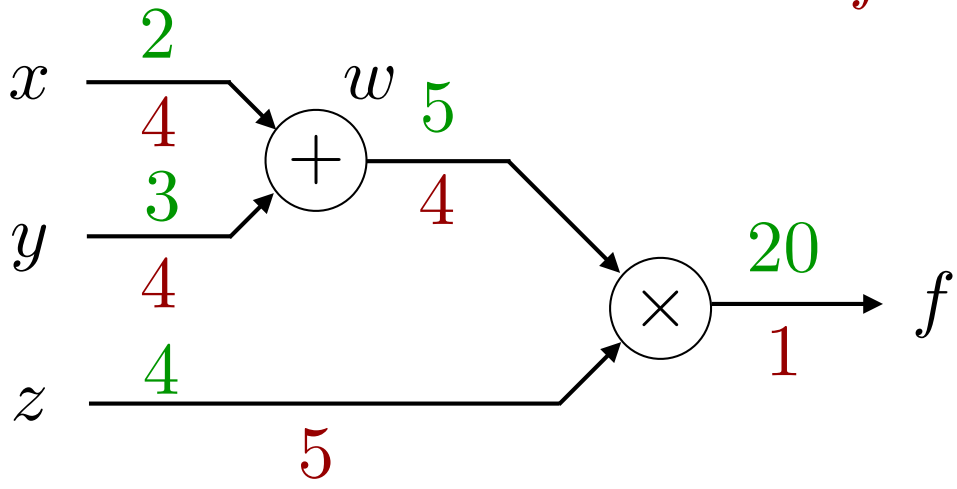
$$\frac{\partial L}{\partial w} = \frac{\partial f}{\partial w} \cdot \frac{\partial L}{\partial f} = 2 \cdot 4 = 8$$

A simple example

$$f(x, y, z) = (x + y)z$$

Backpropagation:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial w}{\partial x}\frac{\partial f}{\partial w}\frac{\partial \mathcal{L}}{\partial f}$$



$w = x + y$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial w}{\partial x} \cdot \frac{\partial \mathcal{L}}{\partial w}$$

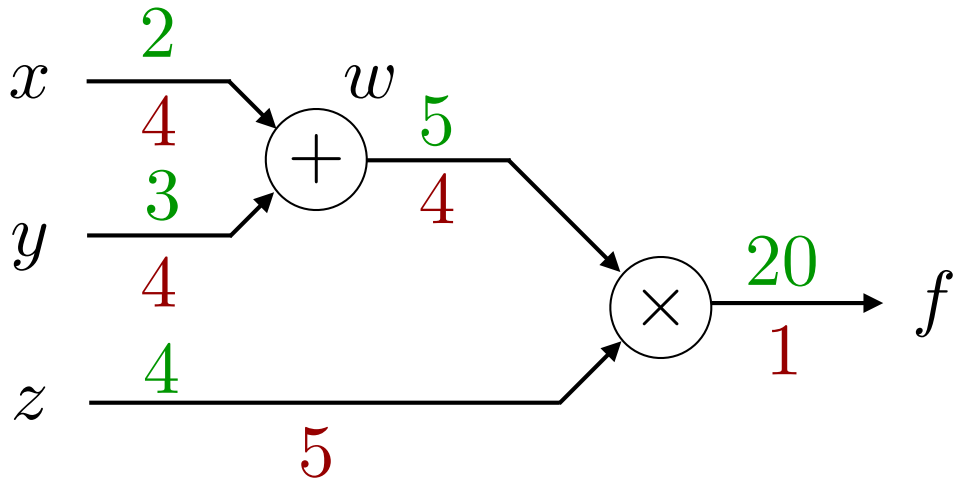
$$\frac{\partial w}{\partial x} = 1$$

$$\frac{\partial \mathcal{L}}{\partial w} = 4$$

$$\frac{\partial \mathcal{L}}{\partial y} = 4$$

Idea: computational graphs apply to gradients

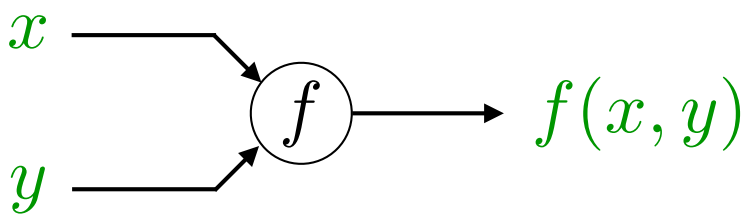
In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.



Idea: computational graphs apply to gradients

In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.

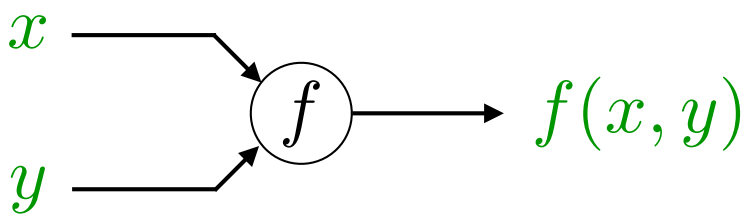
Forward pass:



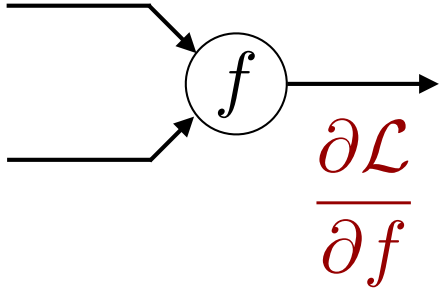
Idea: computational graphs apply to gradients

In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.

Forward pass:



Backward pass:



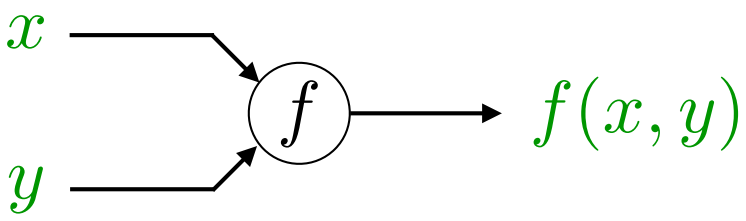
$$\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial L}{\partial f}$$

"LOCAL GRADIENT"

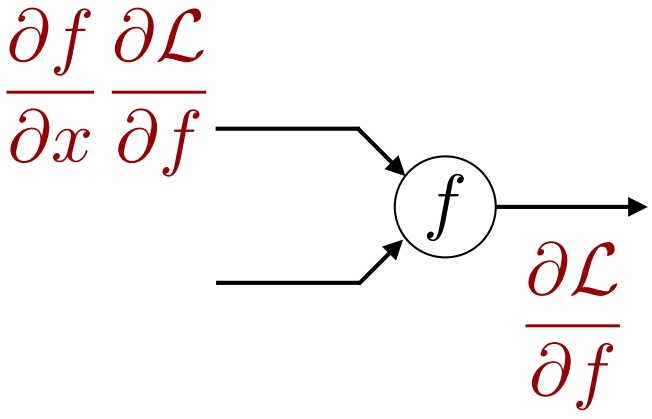
Idea: computational graphs apply to gradients

In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.

Forward pass:



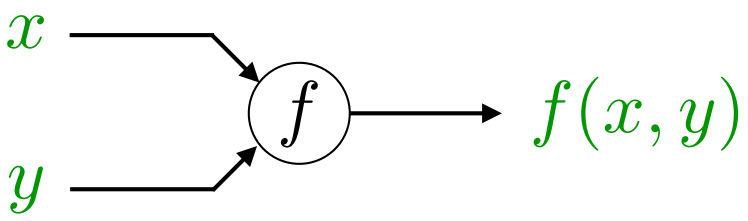
Backward pass:



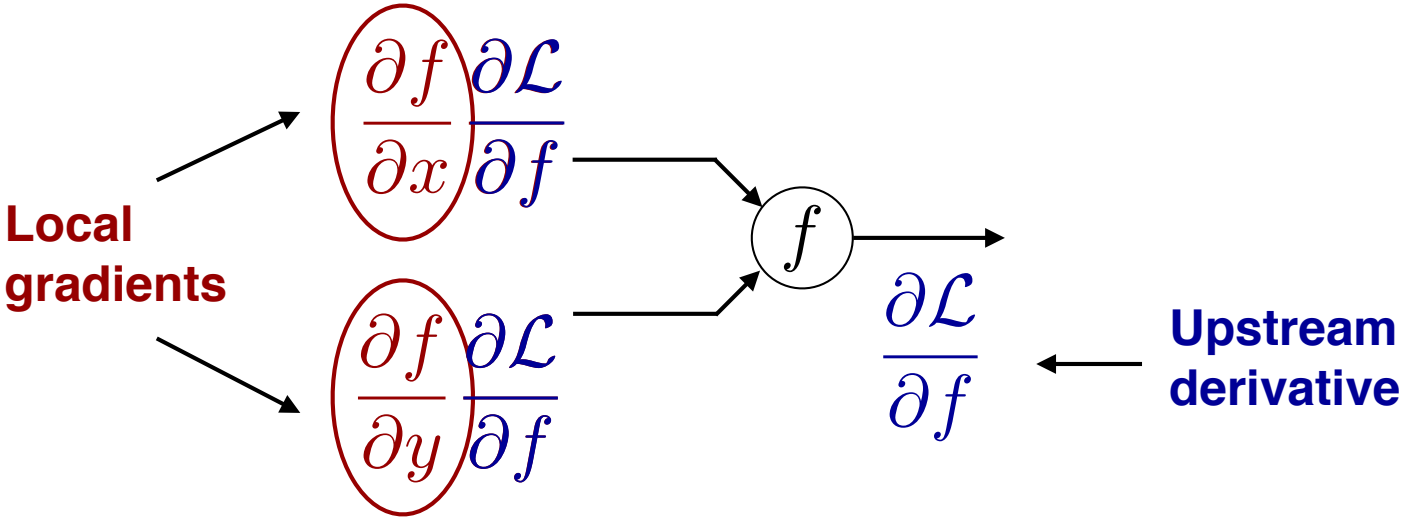
Idea: computational graphs apply to gradients

In the **forward pass**, we apply a function to the node inputs to calculate an output.
In the **backward pass**, we take the **upstream derivative** and apply a **local gradient** to calculate the backpropagated derivative.

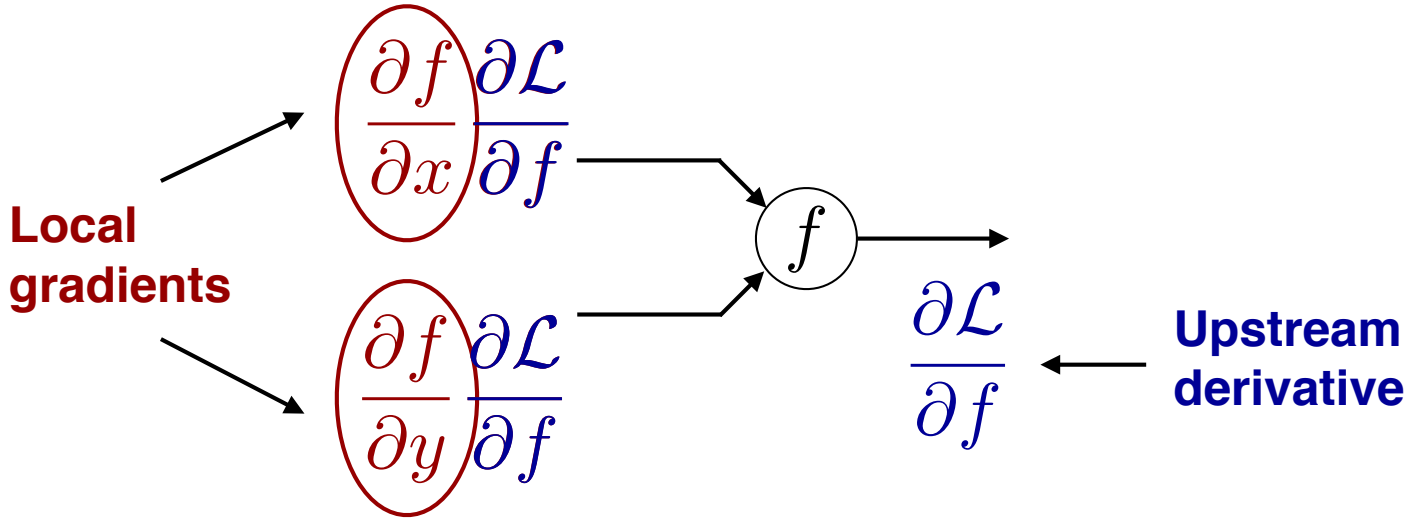
Forward pass:



Backward pass:



Idea: computational graphs apply to gradients

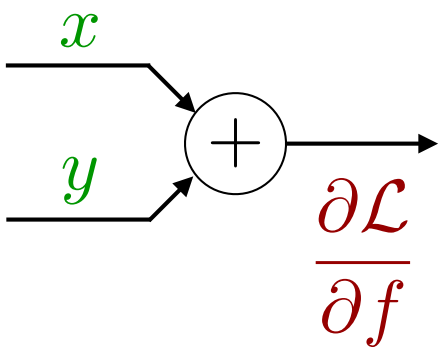


The basic intuition of backpropagation is that we break up the calculation of the gradient into **small and simple steps**. Each of these nodes in the graph is a straightforward gradient calculation, where we multiply an **input** (the "upstream derivative") with a **local gradient** (an application of the chain rule).

Composing all of these gradients together returns the overall gradient.

A gate view of gradients

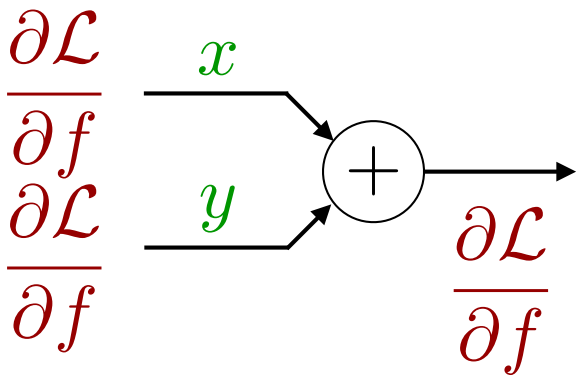
Interpreting backpropagation as gradient “gates”:



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

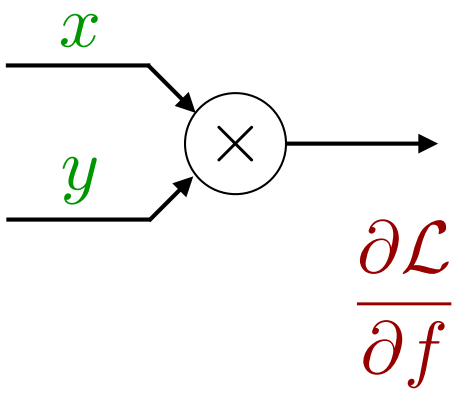
Add gate: distributes the gradient because $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = 1$



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

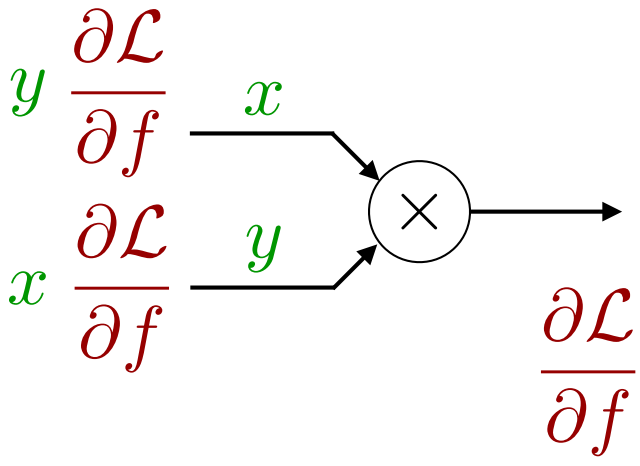
Add gate: distributes the gradient



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

Add gate: distributes the gradient
Mult gate: switches the gradient



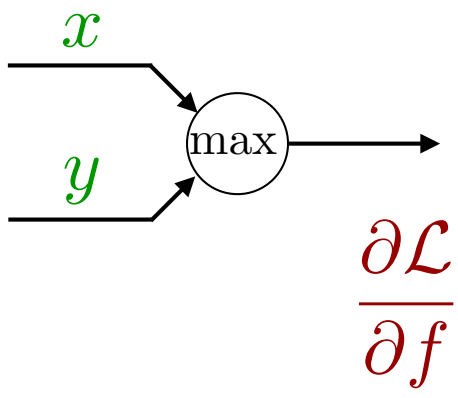
A gate view of gradients

Interpreting backpropagation as gradient “gates”:

Add gate: distributes the gradient
Mult gate: switches the gradient

$$\frac{\partial f}{\partial x} = \begin{cases} 1, & x > y \\ 0, & \text{else} \end{cases}$$

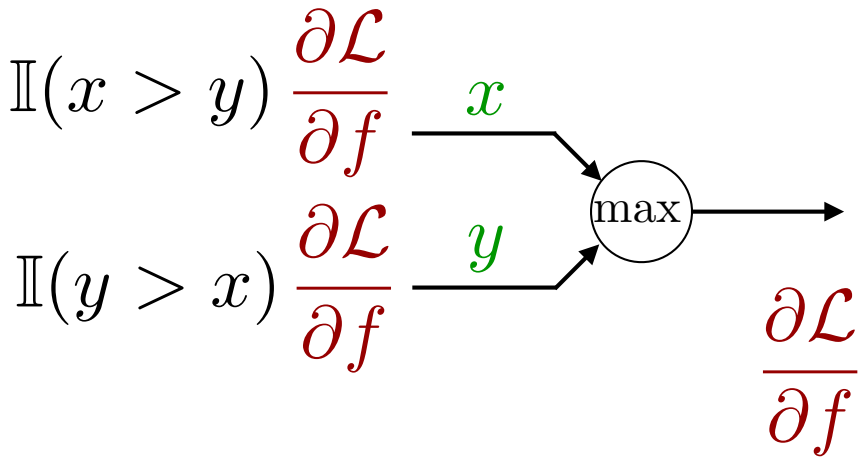
$$= \mathbb{1}\{x > y\}$$



A gate view of gradients

Interpreting backpropagation as gradient “gates”:

Add gate: distributes the gradient
Mult gate: switches the gradient
Max gate: routes the gradient

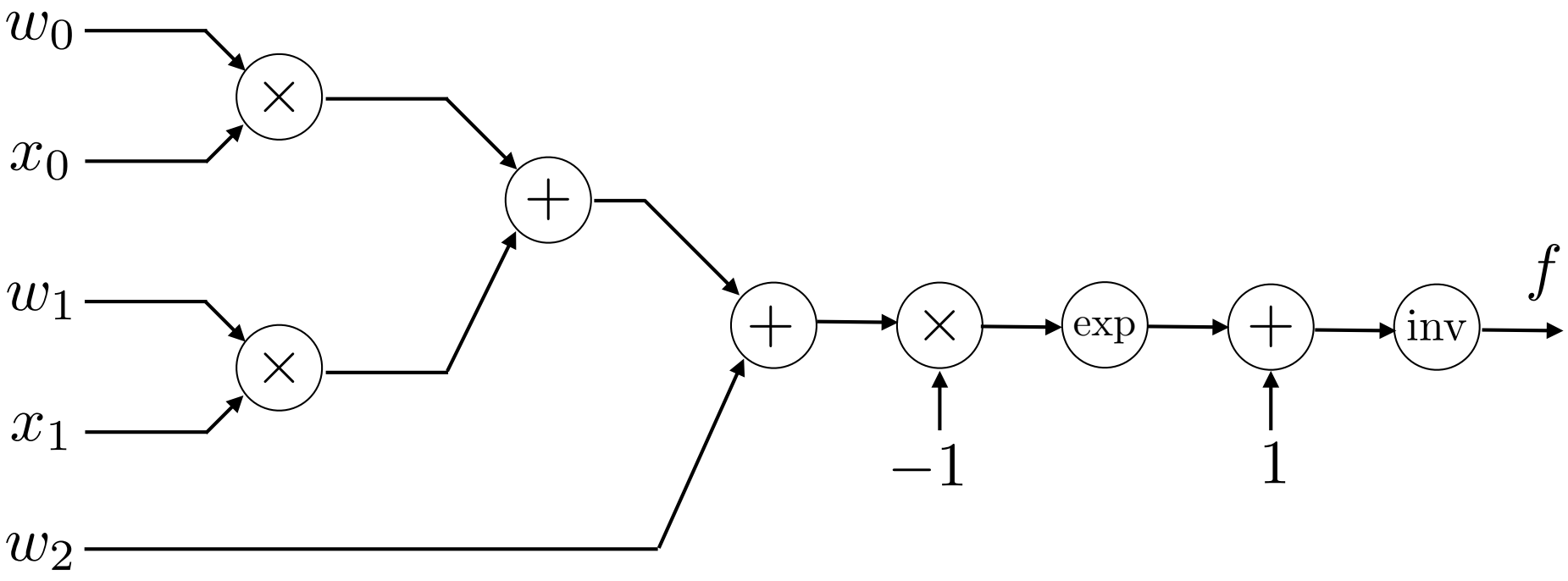


A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

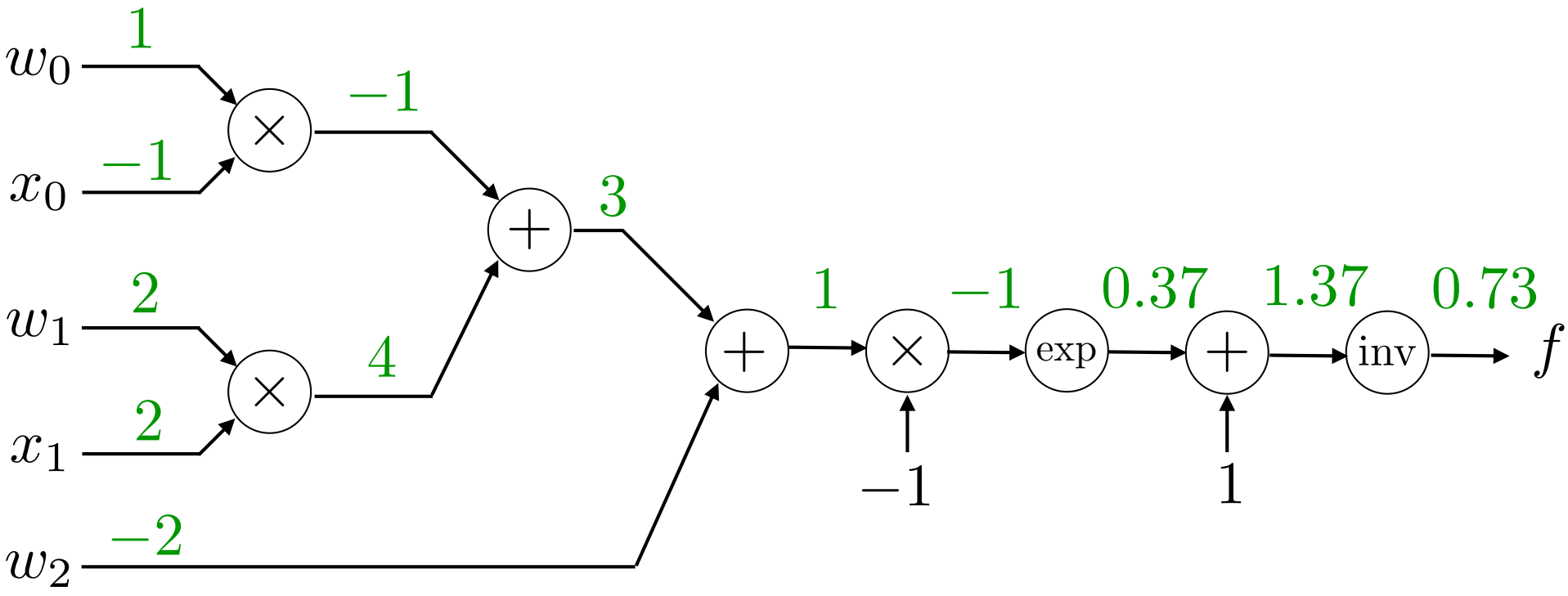
A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



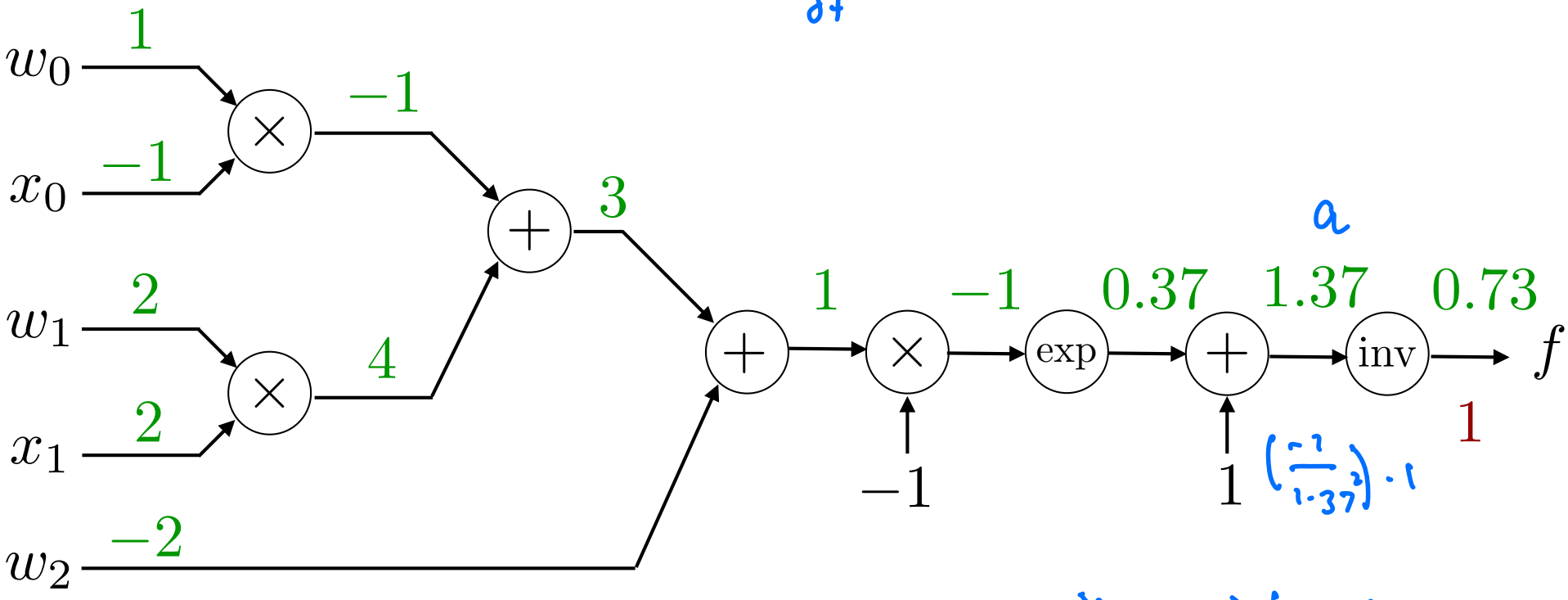
A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

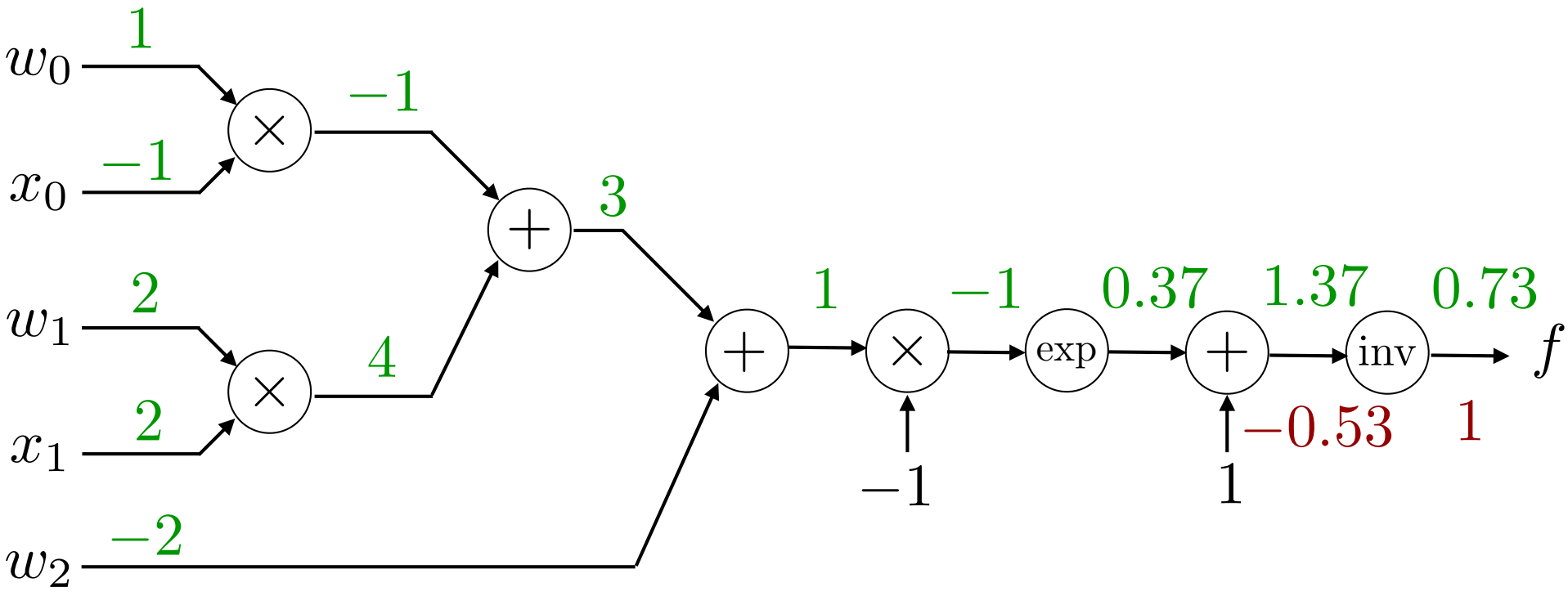


$$\frac{\partial L}{\partial a} = \frac{\partial f}{\partial a} \cdot \frac{\partial L}{\partial f}$$

$f = 1/a$ $\quad \partial f/\partial a = -1/a^2$

A more involved scalar example

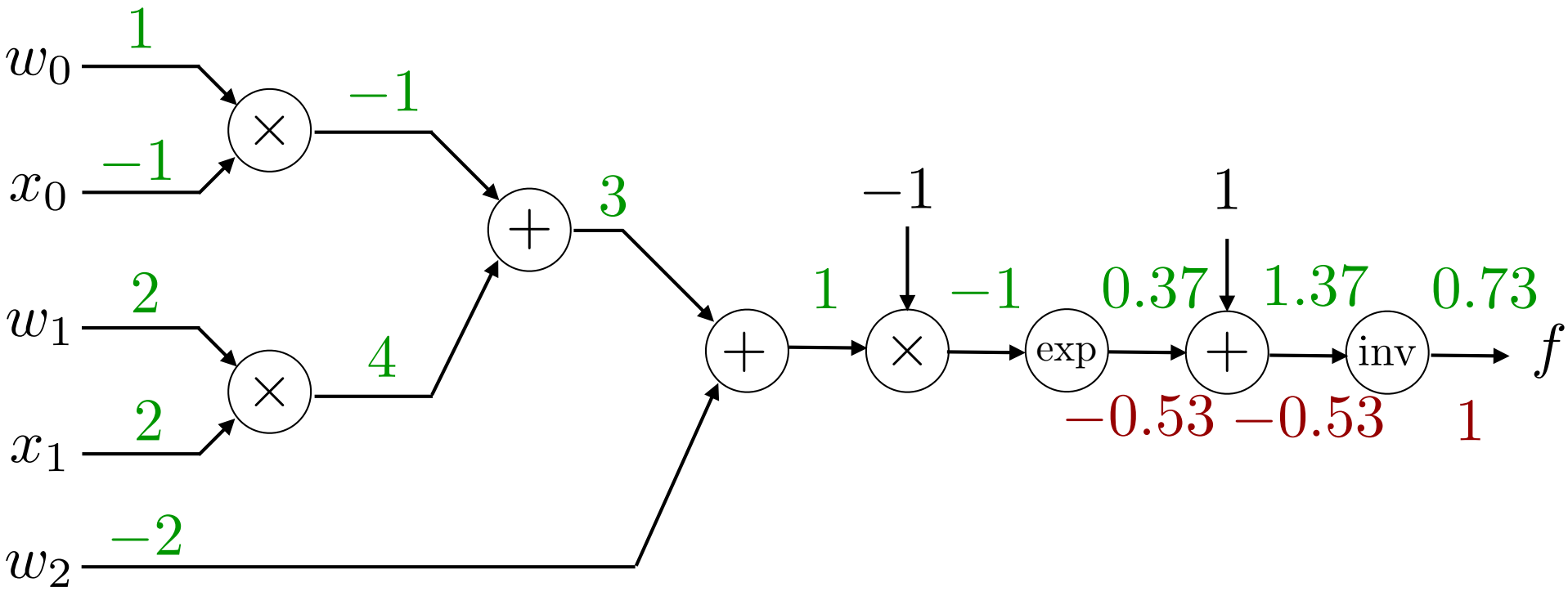
$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial f}{\partial z}\frac{\partial \mathcal{L}}{\partial f} \qquad \frac{\partial f}{\partial z} = -\frac{1}{z^2}$$

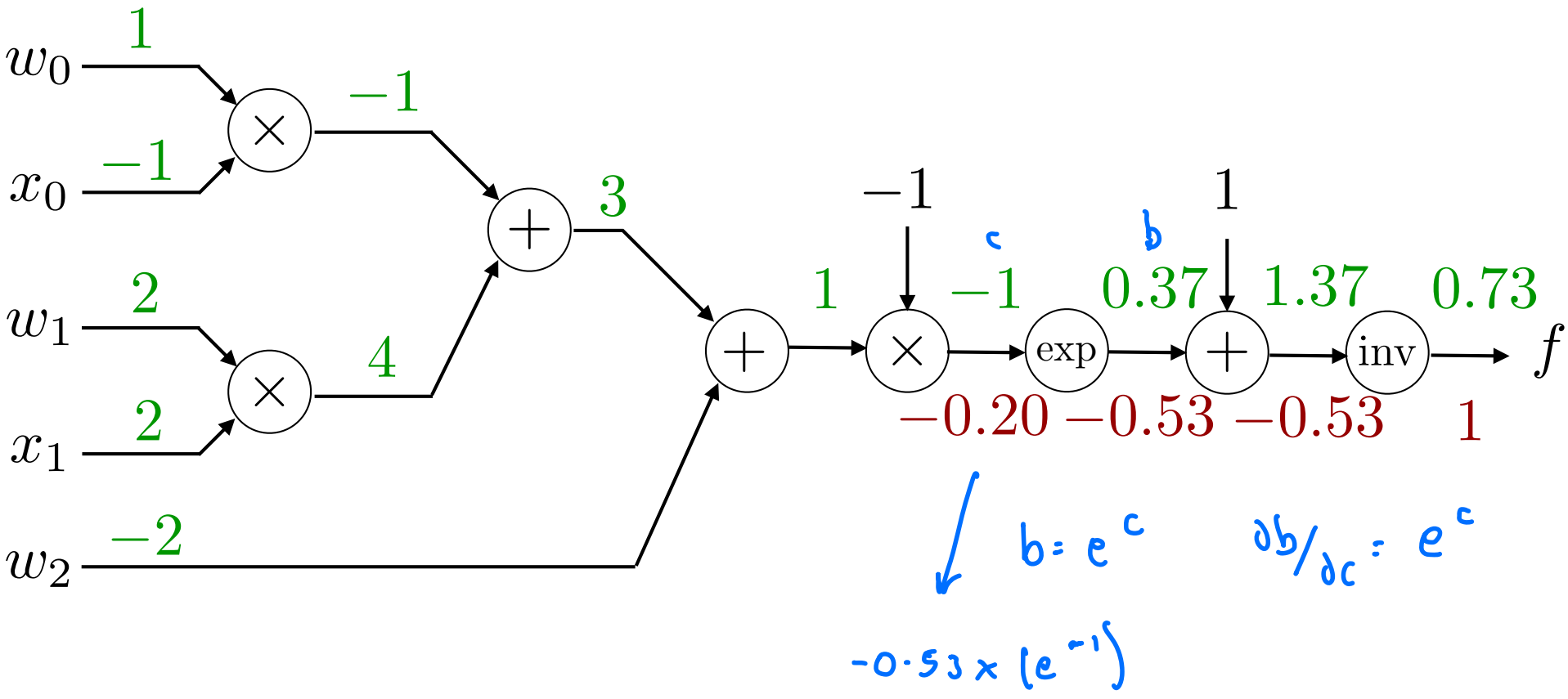
A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



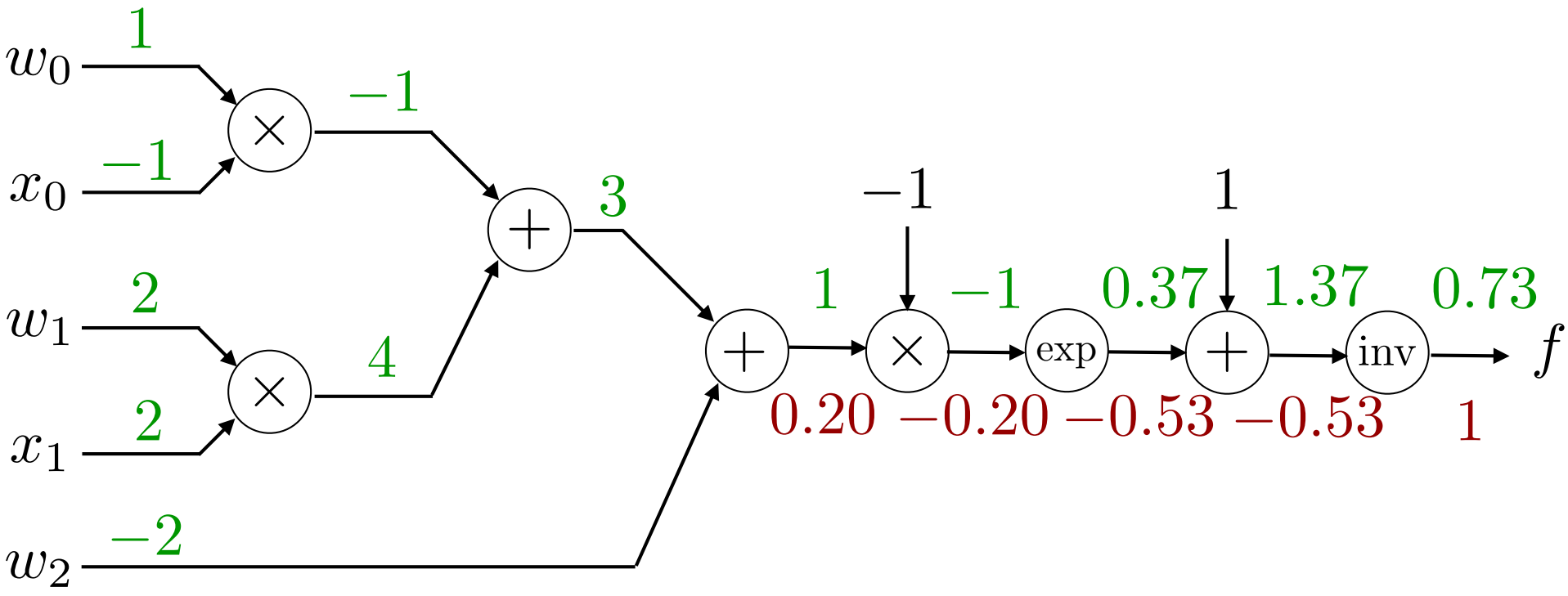
A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



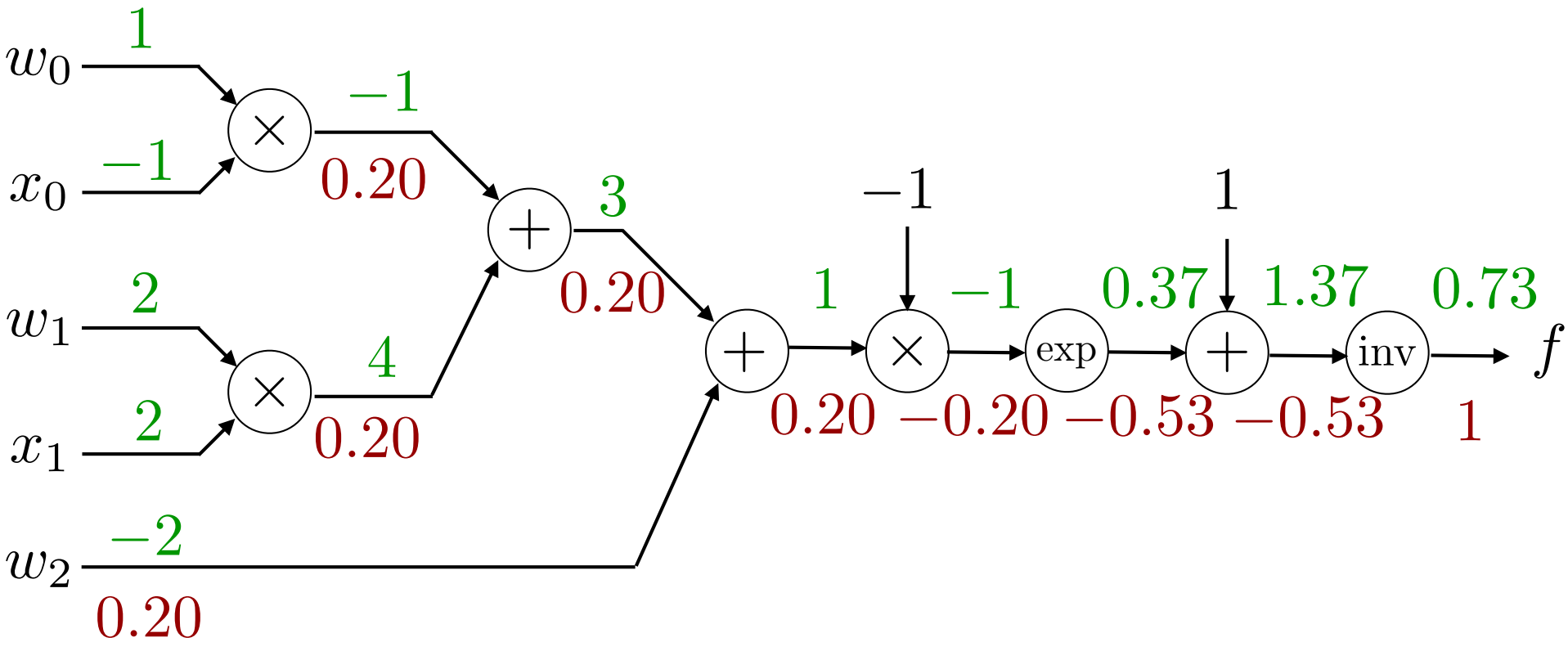
A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



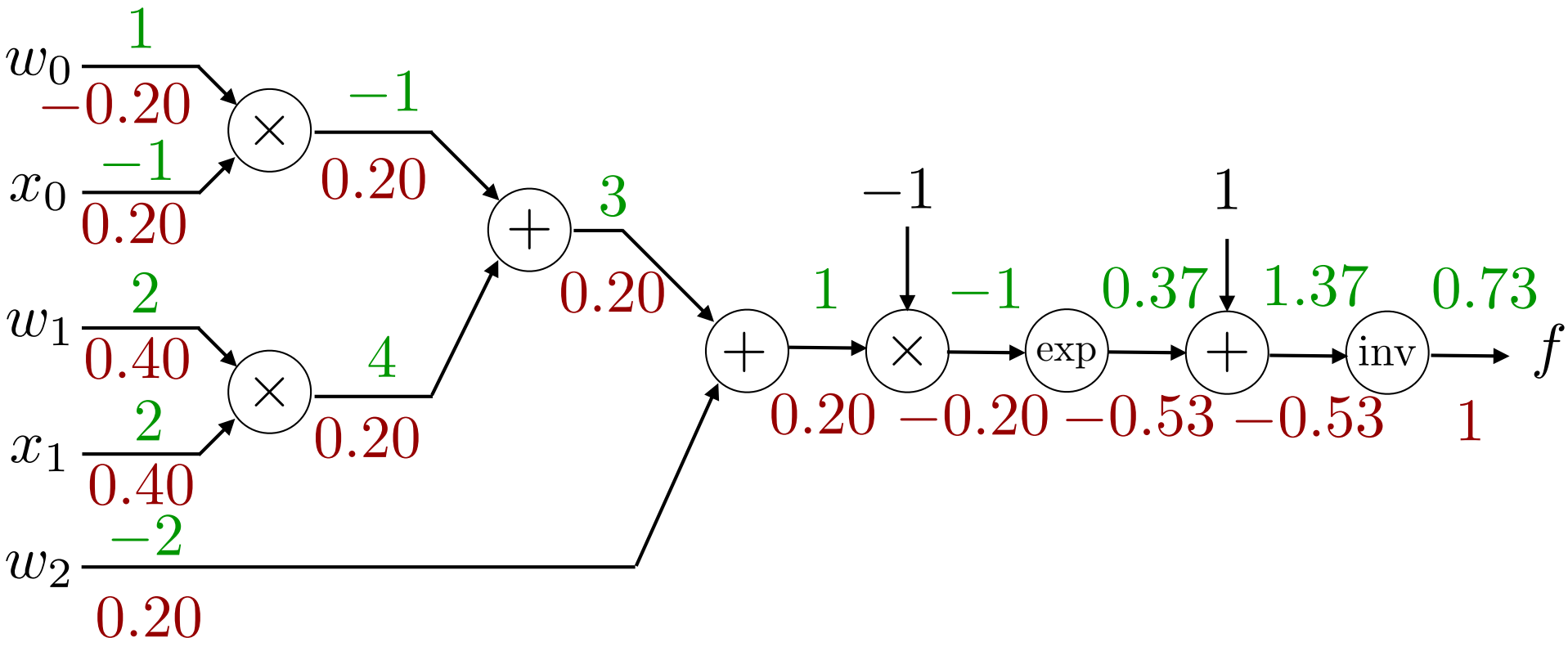
A more involved scalar example

$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$



A more involved scalar example

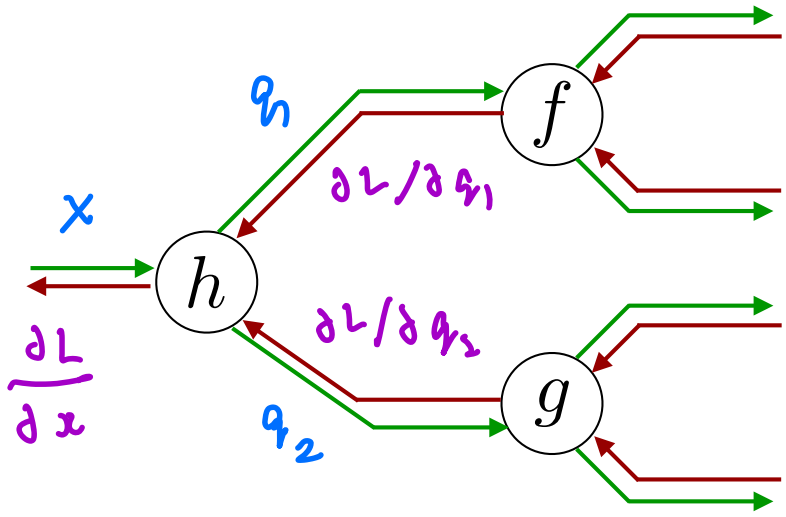
$$f(\mathbf{w}, \mathbf{x}) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}$$

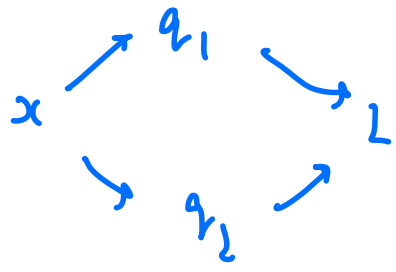


You can take any gradient this way

With backpropagation, as long as you can **break the computation into components where you know the local gradients, you can take the gradient of anything.**

What happens when two gradient paths converge?



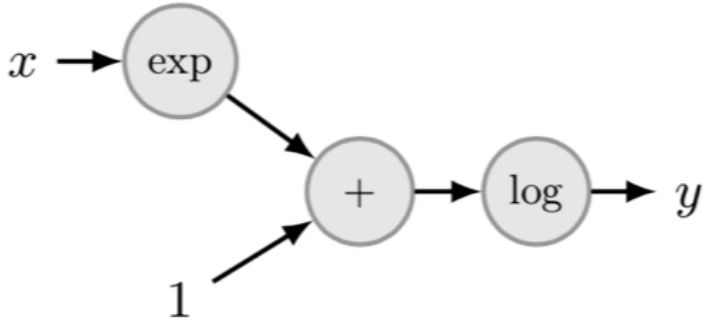


LAW OF TOTAL DERIVATIVES:

$$\frac{\partial L}{\partial x} = \sum_{i=1}^{n} \frac{\partial L}{\partial q_i} \cdot \frac{\partial q_i}{\partial x} \quad //$$

One last example

- $y = \text{softplus}(x)$.

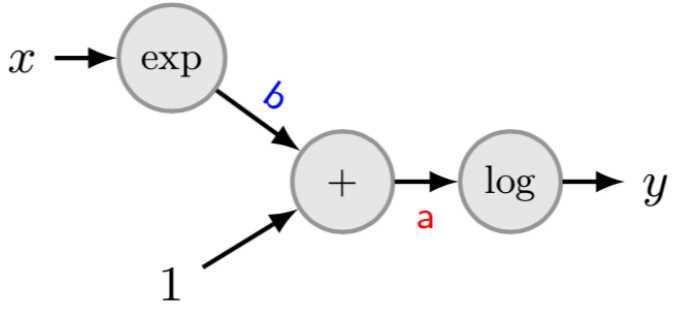


Backpropagation example: softplus

- $y = \text{softplus}(x) = \log(1 + \exp(x))$. We can analytically calculate the derivative.

$$\frac{dy}{dx} = \frac{\exp(x)}{1 + \exp(x)}$$

Let's now calculate it via backpropagation.



Here, we have that $b = \exp(x)$ and that $a = 1 + \exp(x)$. Applying the chain rule, we have that:

$$\begin{aligned}
\frac{dy}{da} &= \frac{d}{da}\log a = \frac{1}{a} \\
\frac{dy}{db} &= \frac{da}{db}\frac{dy}{da} = \frac{dy}{da} \\
\frac{dy}{dx} &= \frac{db}{dx}\frac{dy}{db} = \exp(x)\frac{1}{a}
\end{aligned}$$

Multivariate backpropagation

To do multivariate backpropagation, we need a multivariate chain rule.

Derivative of a scalar w.r.t. a vector

$$\nabla_{\mathbf{x}} y = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}$$

In other words, the gradient is:

- A *vector* that is the same size as $\mathbf{x}$, i.e., if $\mathbf{x} \in \mathbb{R}^n$ then $\nabla_{\mathbf{x}} y \in \mathbb{R}^n$.
- Each dimension of $\nabla_{\mathbf{x}} y$ tells us how small changes in $\mathbf{x}$ in that dimension affect y. i.e., changing the ith dimension of $\mathbf{x}$ by a small amount, Δx_i, will change y by

  $$\frac{\partial y}{\partial x_i} \Delta x_i$$

  We may also denote this as:

  $$\left(\nabla_{\mathbf{x}} y\right)_i \Delta x_i$$

Derivative of a scalar w.r.t. a matrix

The derivative of a scalar, y, with respect to a matrix, $\mathbf{A} \in \mathbb{R}^{m \times n}$, is given by:

$$\nabla_{\mathbf{A}} y = \begin{bmatrix} \frac{\partial y}{\partial a_{11}} & \frac{\partial y}{\partial a_{12}} & \cdots & \frac{\partial y}{\partial a_{1n}} \\ \frac{\partial y}{\partial a_{21}} & \frac{\partial y}{\partial a_{22}} & \cdots & \frac{\partial y}{\partial a_{2n}} \\ \cdots & \cdots & \ddots & \vdots \\ \frac{\partial y}{\partial a_{m1}} & \frac{\partial y}{\partial a_{m2}} & \cdots & \frac{\partial y}{\partial a_{mn}} \end{bmatrix}$$

Like the gradient, the i, jth element of $\nabla_{\mathbf{A}} y$ tells us how small changes in a_{ij} affect y.

Note:

- If you search for the derivative of a scalar with respect to a matrix, you may find people give a transposed definition to the one above.
- Both are valid, but you must be consistent with your notation and use the correct rules. Our notation is called "denominator layout" notation; the other layout is called "numerator layout" notation.
- In the denominator layout, the dimensions of $\nabla_{\mathbf{A}} y$ and $\mathbf{A}$ are the same. The same holds for the gradient, i.e., the dimensions of $\nabla_{\mathbf{x}} y$ and $\mathbf{x}$ are the same. In the numerator layout notation, the dimensions are transposed.
- More on this later, but in "denominator layout," the chain rule goes right to left as opposed to left to right.

Derivative of a vector w.r.t. a vector

$\vec{y} = W\vec{x}$
$(n) \quad (n \times m)\ (m)$

Let $\mathbf{y} \in \mathbb{R}^n$ be a function of $\mathbf{x} \in \mathbb{R}^m$. What dimensionality should the derivative of $\mathbf{y}$ with respect to $\mathbf{x}$ be?

- e.g., to see how $\Delta\mathbf{x}$ modifies y_i, we would calculate:

$$\Delta y_i = \nabla_{\mathbf{x}} y_i^T \cdot \Delta\mathbf{x}$$

- This suggests that the derivative ought to be an $n \times m$ matrix, denoted $\mathbf{J}$, of the form:

$\Delta y_i = \frac{\partial y_i}{\partial x_1} \cdot \Delta x_1 + \frac{\partial y_i}{\partial x_2} \cdot \Delta x_2 + \ldots + \frac{\partial y_i}{\partial x_n} \cdot \Delta x_n$

"JACOBIAN"

$$\mathbf{J} = \begin{bmatrix} (\nabla_{\mathbf{x}} y_1)^T \\ (\nabla_{\mathbf{x}} y_2)^T \\ \vdots \\ (\nabla_{\mathbf{x}} y_n)^T \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

$n \times m$

The matrix would tell us how a small change in $\Delta\mathbf{x}$ results in a small change in $\Delta\mathbf{y}$ according to the formula:

$$\Delta\mathbf{y} \approx \mathbf{J}\Delta\mathbf{x}$$

$$\Delta y_i = \frac{\partial y_i}{\partial x_1} \cdot \Delta x_1 + \frac{\partial y_1}{\partial x_2} \cdot \Delta x_2 + \dots + \frac{\partial y_i}{\partial x_n} \cdot \Delta x_n$$

$$\Delta y = \begin{bmatrix} \nabla_x y_1^T \\ \nabla_x y_2^T \\ \vdots \\ \nabla_x y_n^T \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_n \end{bmatrix}$$

$$\downarrow \qquad \downarrow$$

$$J \qquad \Delta x$$

Derivative of a vector w.r.t. a vector (cont.)

The matrix $\mathbf{J}$ is called the Jacobian matrix.

A word on notation:

- In the denominator layout definition, the denominator vector changes along columns.

$$\nabla_{\mathbf{x}}\mathbf{y} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \triangleq \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

- Hence the notation we use for the Jacobian would be:

$$\mathbf{J} = (\nabla_{\mathbf{x}}\mathbf{y})^T = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T$$

Derivative of a vector w.r.t. a vector

Example: derivative of a vector with respect to a vector

The following derivative will appear later on in the class. Let $\mathbf{W} \in \mathbb{R}^{h \times n}$ and $\mathbf{x} \in \mathbb{R}^n$. We would like to calculate the derivative of $f(\mathbf{x}) = \mathbf{W}\mathbf{x}$ with respect to $\mathbf{x}$.

$$\nabla_{\mathbf{x}} \mathbf{W}\mathbf{x} = \nabla_{\mathbf{x}} \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2n}x_n \\ \vdots \\ w_{h1}x_1 + w_{h2}x_2 + \cdots + w_{hn}x_n \end{bmatrix} \quad y_1, y_2$$

$$= \begin{bmatrix} w_{11} & w_{21} & \cdots & w_{h1} \\ w_{12} & w_{22} & \cdots & w_{h2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1n} & w_{2n} & \cdots & w_{hn} \end{bmatrix} \quad \partial y_1/\partial x, \ \partial y_2/\partial x, \ \partial y_n/\partial x$$

$$= \mathbf{W}^T$$

$y = Wx$

$y \in \mathbb{R}^h$

$x \in \mathbb{R}^n$

Hessian: the generalization of the second derivative

Hessian

scalar

The Hessian matrix of a function $f(\mathbf{x})$ is a square matrix of second-order partial derivatives of $f(\mathbf{x})$. It is composed of elements:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial f}{\partial x_1^2} & \frac{\partial f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f}{\partial x_1 \partial x_n} \\ \frac{\partial f}{\partial x_2 x_1} & \frac{\partial f}{\partial x_2^2} & \cdots & \frac{\partial f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_m x_1} & \frac{\partial f}{\partial x_m \partial x_2} & \cdots & \frac{\partial f}{\partial x_m^2} \end{bmatrix}$$

We can denote this matrix as $\nabla_{\mathbf{x}} (\nabla_{\mathbf{x}} f(\mathbf{x}))$. We often denote this simply as $\nabla_{\mathbf{x}}^2 f(\mathbf{x})$.

$\nabla_x f(x) \in \mathbb{R}^m$

$\nabla_x (\nabla_x f(x)) \in \mathbb{R}^{m \times m}$

The scalar chain rule

The scalar chain rule states that if $y = f(x)$ and $z = g(y)$, then

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Intuitively: the chain rule tells us that a small change in x will cause a small change in y that will in turn cause a small change in z, i.e., for apropriately small Δx,

$$\begin{aligned} \Delta y &\approx \frac{dy}{dx}\Delta x \\ \Delta z &\approx \frac{dz}{dy}\Delta y \\ &= \frac{dz}{dy}\frac{dy}{dx}\Delta x \end{aligned}$$

Vector chain rule

Chain rule for vector valued functions

In the "denominator" layout, the chain rule runs from right to left. We won't derive this, but we will check the dimensionality and intuition.

Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, and $\mathbf{z} \in \mathbb{R}^p$. Further, let $\mathbf{y} = f(\mathbf{x})$ for $f : \mathbb{R}^m \to \mathbb{R}^n$ and $\mathbf{z} = g(\mathbf{y})$ for $g : \mathbb{R}^n \to \mathbb{R}^p$. Then,

$$\nabla_{\mathbf{x}} \mathbf{z} \quad = \quad \nabla_{\mathbf{x}} \mathbf{y} \, \nabla_{\mathbf{y}} \mathbf{z}$$

Equivalently:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \quad = \quad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}}$$

- Note that $\nabla_{\mathbf{x}} \mathbf{z}$ should have dimensionality $\mathbb{R}^{m \times p}$.
- As $\nabla_{\mathbf{x}} \mathbf{y} \in \mathbb{R}^{m \times n}$ and $\nabla_{\mathbf{y}} \mathbf{z} \in \mathbb{R}^{n \times p}$, the operations are dimension consistent.

$x \in \mathbb{R}^m$

$y \in \mathbb{R}^n$

$z \in \mathbb{R}^p$

$y = f(x)$

$z = g(y)$

(1) $\Delta x \rightarrow \Delta z$

$$\Delta z = \left(\frac{\partial z}{\partial x}\right)^T \Delta x$$

(2) $\Delta x \rightarrow \Delta y \rightarrow \Delta z$

$$\Delta y \simeq \left(\partial y / \partial x\right)^T \Delta x$$

$$\Delta z \simeq \left(\partial z / \partial y\right)^T \Delta y = \left(\partial z / \partial y\right)^T \left(\partial y / \partial x\right)^T \Delta x$$

$$\left(\partial z / \partial x\right)^T = \left(\partial y / \partial x\right)^T \left(\partial z / \partial y\right)^T$$

$$\boxed{\partial z / \partial x = \partial y / \partial x \cdot \partial z / \partial y}$$

$\partial z / \partial x \in \mathbb{R}^{(m \times p)}$ $\quad$ $\partial y / \partial x \in \mathbb{R}^{(m \times n)}$ $\quad$ $\partial z / \partial y \in \mathbb{R}^{(n \times p)}$

HINT: DENOMINATOR LAYOUT: $\frac{\partial x}{\partial y}$ $\begin{matrix} \leftarrow m \\ \leftarrow n \end{matrix}$

$(n \times m)$

↳ denominator first

Chain rule for vector valued functions (cont.)

- Intuitively, a small change $\Delta\mathbf{x}$ affects $\Delta\mathbf{z}$ through the Jacobian $(\nabla_{\mathbf{x}}\mathbf{z})^T$

$$\Delta\mathbf{z} \approx (\nabla_{\mathbf{x}}\mathbf{z})^T \Delta\mathbf{x} \tag{1}$$

- The chain rule is intuitive, since:

$$\begin{aligned} \Delta\mathbf{y} &\approx (\nabla_{\mathbf{x}}\mathbf{y})^T \Delta\mathbf{x} \\ \Delta\mathbf{z} &\approx (\nabla_{\mathbf{y}}\mathbf{z})^T \Delta\mathbf{y} \end{aligned}$$

  Composing these, we have that:

$$\Delta\mathbf{z} \approx (\nabla_{\mathbf{y}}\mathbf{z})^T (\nabla_{\mathbf{x}}\mathbf{y})^T \Delta\mathbf{x} \tag{2}$$

- Combining equations (1) and (2), we arrive at:

$$(\nabla_{\mathbf{x}}\mathbf{z})^T = (\nabla_{\mathbf{y}}\mathbf{z})^T (\nabla_{\mathbf{x}}\mathbf{y})^T$$

  which, after transposing both sides, reduces to the (right to left) chain rule:

$$\nabla_{\mathbf{x}}\mathbf{z} = \nabla_{\mathbf{x}}\mathbf{y}\, \nabla_{\mathbf{y}}\mathbf{z}$$

Vector chain rule example

Multivariate chain rule example

Consider the squared loss function: $\varepsilon = \|\mathbf{x} - \mathbf{h}\|^2$, where $\mathbf{h}$ is some target we are trying to approximate via $\mathbf{x}$. We wish to calculate $\nabla_{\mathbf{x}}\varepsilon$.

We will set $\mathbf{y} = \mathbf{x} - \mathbf{h}$ and use the chain rule. Note that $\varepsilon = \mathbf{y}^T\mathbf{y}$.

- First, we find $\nabla_{\mathbf{y}}\varepsilon$.

$$\begin{aligned} \nabla_{\mathbf{y}}\varepsilon &= \nabla_{\mathbf{y}}\left(\sum_i y_i^2\right) \\ &= 2\mathbf{y} \end{aligned}$$

- Next, we find $\nabla_{\mathbf{x}}\mathbf{y}$.

$$\begin{aligned} \nabla_{\mathbf{x}}\mathbf{y} &= \begin{bmatrix} \frac{\partial(x_1-h_1)}{\partial x_1} & \frac{\partial(x_2-h_2)}{\partial x_1} & \cdots & \frac{\partial(x_n-h_n)}{\partial x_1} \\ \frac{\partial(x_1-h_1)}{\partial x_2} & \frac{\partial(x_2-h_2)}{\partial x_2} & \cdots & \frac{\partial(x_n-h_n)}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial(x_1-h_1)}{\partial x_n} & \frac{\partial(x_2-h_2)}{\partial x_n} & \cdots & \frac{\partial(x_n-h_n)}{\partial x_n} \end{bmatrix} \\ &= \mathbf{I} \end{aligned}$$

Vector chain rule example

Multivariate chain rule example (cont.)

- Using the chain rule,

$$\begin{aligned} \nabla_{\mathbf{x}} \varepsilon &= (\nabla_{\mathbf{x}} \mathbf{y}) (\nabla_{\mathbf{y}} \varepsilon) \\ &= 2(\mathbf{x} - \mathbf{h}) \end{aligned}$$

A few notes:

- Formally, $\nabla_{\mathbf{x}} \mathbf{y}$ is an $n \times n$ matrix. However, knowing that $\mathbf{y} = \mathbf{x} - \mathbf{h}$, we can intuit that its affect should be multiplication by 1 (or formally, the identity matrix). This intuition is important for deep learning and dealing with derivatives of tensors.
- Often times, we don't want to represent and store the entire tensor derivative, which could be enormous. Many entries in this tensor derivative will be 0, much like many of the entries in $\nabla_{\mathbf{x}} \mathbf{y}$ are zero.

Tensor Derivatives

$y = Wx$

$y \in \mathbb{R}^m$

$x \in \mathbb{R}^n$

$W \in \mathbb{R}^{m \times n}$

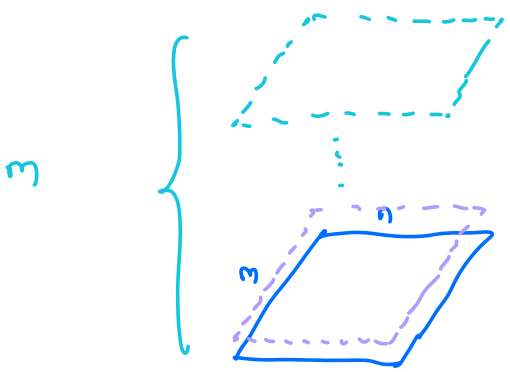
$$\frac{\partial y}{\partial W} \in \mathbb{R}^{m \times n \times m}$$

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$\frac{\partial y_1}{\partial W} \in \mathbb{R}^{n \times m}$$

$$\frac{\partial y_2}{\partial W} \in \mathbb{R}^{n \times m}$$

$$\frac{\partial y_m}{\partial W} \in \mathbb{R}^{n \times m}$$



$$\rightarrow \frac{\partial y}{\partial W} \quad //$$

Derivatives of tensors

Occasionally in this class, we may need to take a derivative that is more than 2-dimensional. For example, we may want to take the derivative of a vector with respect to a matrix. This would be a 3-dimensional tensor. The definition for this would be as you expect. In particular, if $\mathbf{z} \in \mathbb{R}^p$ and $\mathbf{W} \in \mathbb{R}^{m \times n}$, then $\nabla_{\mathbf{W}} \mathbf{z}$ is a three-dimensional tensor with shape $\mathbb{R}^{m \times n \times p}$. Each $m \times n$ slice (of which there are p) is the matrix derivative $\nabla_{\mathbf{W}} z_i$.

Note, these are sometimes a headache to work with. We typically can find a shortcut to perform operations without having to compute and store these high-dimensional tensor derivatives. The next slides show an example.

Tensor derivatives

Multivariate chain rule and tensor derivative example

Consider the squared loss function:

$$\mathcal{L} = \frac{1}{2} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - \mathbf{W}\mathbf{x}^{(i)} \right\|^2$$

Here, $\mathbf{y}^{(i)} \in \mathbb{R}^m$ and $\mathbf{x}^{(i)} \in \mathbb{R}^n$ so that $\mathbf{W} \in \mathbb{R}^{m \times n}$. We wish to find $\mathbf{W}$ that minimizes the mean-square error in linearly predicting $\mathbf{y}^{(i)}$ from $\mathbf{x}^{(i)}$.

$$L = \frac{1}{2} \sum_{i=1}^{N} (y^{(i)} - Wx^{(i)})^T (y^{(i)} - Wx^{(i)})$$

$$z^{(i)} = y^{(i)} - Wx^{(i)}$$

$$L = \frac{1}{2} \sum_{i=1}^{N} z^{(i)T} z^{(i)}$$

omit superscript i

$z \in \mathbb{R}^m$

$$\frac{\partial L}{\partial w} = \frac{\partial z}{\partial w} \cdot \frac{\partial L}{\partial z}$$

$(m \times n) \quad (m \times n \times m) \; (m \times 1)$

Multivariate chain rule example

We consider one example, $\varepsilon^{(i)} = \frac{1}{2}\left\|\mathbf{y}^{(i)} - \mathbf{W}\mathbf{x}^{(i)}\right\|^2$.

We wish to calculate $\nabla_{\mathbf{W}}\varepsilon^{(i)}$.

To do so, we define $\mathbf{z}^{(i)} = \mathbf{y}^{(i)} - \mathbf{W}\mathbf{x}^{(i)}$. Then, $\varepsilon^{(i)} = \frac{1}{2}(\mathbf{z}^{(i)})^T\mathbf{z}^{(i)}$. For the rest of the example, we're going to drop the superscripts (i) and assume we're working with the ith example (to help notation). Then, we need to calculate is:

$$\nabla_{\mathbf{W}}\varepsilon = \nabla_{\mathbf{W}}\mathbf{z}\nabla_{\mathbf{z}}\varepsilon$$

Multivariate chain rule example

Now, we need to calculate $\nabla_{\mathbf{W}}\mathbf{z}$. This is a three dimensional tensor with dimensionality $m \times n \times m$.

This makes sense dimensionally, because when we multiply a $(m \times n \times m)$ tensor by a $(m \times 1)$ vector, we get out a $(m \times n \times 1)$ tensor, which is equivalently an $(m \times n)$ matrix. Because $\nabla_{\mathbf{W}}\varepsilon$ is an $(m \times n)$ matrix, this all works out.

Multivariate chain rule example

$$\frac{\partial z}{\partial w} \in \mathbb{R}^{m \times n \times m}$$

$$\frac{\partial z_k}{\partial w} \in \mathbb{R}^{m \times n} \qquad \text{(I have } m \text{ of these)}$$

$$z = y - Wx$$

$$z_k = y_k - \sum_j W_{kj} x_j$$

$z_k \rightarrow$ scalar

$$\frac{\partial z_k}{\partial w} = \frac{\partial}{\partial w}\left[y_k - \sum_j W_{kj} x_j \right]$$

$$\frac{\partial z_k}{\partial w_{ip}} = \frac{-\partial}{\partial w_{ip}} \left(\sum_j W_{kj} x_j \right) = \frac{-\partial}{\partial w_{ip}} \left(W_{k1} x_1 + W_{k2} x_2 + \cdots + W_{kn} x_n \right)$$

Case 1: when $i \neq k$

$$\frac{\partial z_k}{\partial w_{ip}} = 0$$

Case 2: when $i = k$

$$\frac{\partial z_k}{\partial w_{ip}} = -x_p$$

$\frac{\partial z_k}{\partial w_{ip}}$ is $m \times n$ matrix

k^{TH} row $\rightarrow$

$$\begin{bmatrix} & 0 & \\ & 0 & \\ & 0 & \\ -x_1 & -x_2 \ \ldots & -x_n \\ & 0 & \\ & 0 & \end{bmatrix}$$

Multivariate chain rule and tensor derivative example (cont.)

$\nabla_{\mathbf{W}}\mathbf{z}$ is an $(m \times n \times m)$ tensor.

- Letting z_k denote the kth element of $\mathbf{z}$, we see that each $\frac{\partial z_k}{\partial \mathbf{W}}$ is an $m \times n$ matrix, and that there are m of these for each element z_k from $k = 1, \ldots, m$.
- The *matrix*, $\frac{\partial z_k}{\partial \mathbf{W}}$, can be calculated as follows:

$$\frac{\partial z_k}{\partial \mathbf{W}} = \frac{\partial}{\partial \mathbf{W}} \sum_{j=1}^{n} -w_{kj} x_j$$

and thus, the (k, j)th element of this matrix is:

$$\left(\frac{\partial z_k}{\partial \mathbf{W}}\right)_{k,j} = \frac{\partial z_k}{\partial w_{kj}} = -x_j$$

It is worth noting that

$$\left(\frac{\partial z_k}{\partial \mathbf{W}}\right)_{i,j} = \frac{\partial z_k}{\partial w_{ij}} = 0 \qquad \text{for } k \neq i$$

Multivariate chain rule and tensor derivative example (cont.)

Hence, $\frac{\partial z_k}{\partial \mathbf{W}}$ is a matrix where the kth row is $\mathbf{x}^T$ and all other rows are the zeros (we denote the zero vector by $\mathbf{0}$). i.e.,

$$\frac{\partial z_1}{\partial \mathbf{W}} = \begin{bmatrix} -\mathbf{x}^T \\ \mathbf{0}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} \qquad \frac{\partial z_2}{\partial \mathbf{W}} = \begin{bmatrix} \mathbf{0}^T \\ -\mathbf{x}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} \qquad \text{etc...}$$

Now applying the chain rule,

$$\frac{\partial \varepsilon}{\partial \mathbf{W}} = \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \frac{\partial \varepsilon}{\partial \mathbf{z}}$$

is a tensor product between an $(m \times n \times m)$ tensor and an $(m \times 1)$ vector, whose resulting dimensionality is $(m \times n \times 1)$ or equivalently, an $(m \times n)$ matrix.

Multivariate chain rule example

$$\frac{\partial \ell}{\partial z} = \begin{bmatrix} \partial \ell / \partial z_1 \\ \partial \ell / \partial z_2 \\ \vdots \\ \partial \ell / \partial z_m \end{bmatrix}$$

$$\frac{\partial \ell}{\partial W} = \frac{\partial z}{\partial W} \cdot \frac{\partial \ell}{\partial z} = \sum_{k=1}^{m} \frac{\partial z_k}{\partial W} \cdot \frac{\partial \ell}{\partial z_k}$$

$(m \times n \times m)$ $(m \times 1)$

$$= \frac{\partial \ell}{\partial z_1} \cdot \begin{bmatrix} -x^T \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \frac{\partial \ell}{\partial z_2} \begin{bmatrix} 0 \\ -x^T \\ \vdots \\ 0 \end{bmatrix} + \cdots$$

$\frac{\partial \ell}{\partial z_1}$ / scalar

$$= \begin{bmatrix} -\partial z_1/\partial p \; x^T \\ -\partial z_2/\partial p \; x^T \\ \vdots \\ -\partial z_m/\partial p \; x^T \end{bmatrix} = -\partial z/\partial p \; x^T$$

$(m \times 1)$ $(1 \times n)$

$\Downarrow$

$(m \times n)$

Multivariate chain rule and tensor derivative example (cont.)

We carry out this tensor-vector multiply in the standard way.

$$
\begin{aligned}
\frac{\partial \varepsilon}{\partial \mathbf{W}} &= \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \frac{\partial \varepsilon}{\partial \mathbf{z}} \\
&= \sum_{i=1}^{m} \frac{\partial z_i}{\partial \mathbf{W}} \left(\frac{\partial \varepsilon}{\partial \mathbf{z}} \right)_i \\
&= \frac{\partial \varepsilon}{\partial z_1} \begin{bmatrix} -\mathbf{x}^T \\ \mathbf{0}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} + \frac{\partial \varepsilon}{\partial z_2} \begin{bmatrix} \mathbf{0}^T \\ -\mathbf{x}^T \\ \vdots \\ \mathbf{0}^T \end{bmatrix} + \cdots \\
&\quad + \frac{\partial \varepsilon}{\partial z_m} \begin{bmatrix} \mathbf{0}^T \\ \mathbf{0}^T \\ \vdots \\ -\mathbf{x}^T \end{bmatrix}
\end{aligned}
$$

Multivariate chain rule and tensor derivative example (cont.)

Continuing the work from the previous page...

$$\begin{aligned}\frac{\partial \varepsilon}{\partial \mathbf{W}} &= -\begin{bmatrix} \frac{\partial \varepsilon}{\partial z_1}\mathbf{x}^T \\ \frac{\partial \varepsilon}{\partial z_2}\mathbf{x}^T \\ \vdots \\ \frac{\partial \varepsilon}{\partial z_m}\mathbf{x}^T \end{bmatrix} \\ &= -\frac{\partial \varepsilon}{\partial \mathbf{z}}\mathbf{x}^T\end{aligned}$$

Multivariate chain rule example

Hence, with a final application of the chain rule, we get that

$$\begin{aligned} \nabla_{\mathbf{W}} \varepsilon &= -\mathbf{z}\mathbf{x}^T \\ &= -(\mathbf{y} - \mathbf{W}\mathbf{x})\mathbf{x}^T \end{aligned}$$

Setting this equal to zero, we find that for one example,

$$\mathbf{W} = \mathbf{y}\mathbf{x}^T(\mathbf{x}\mathbf{x}^T)^{-1}$$

Summing across all examples, this produces least-squares.

Multivariate chain rule example

A few notes on tensor derivatives

- In general, the simpler rule can be inferred via pattern intuition / looking at the dimensionality of the matrices, and these tensor derivatives need not be explicitly derived.
- Indeed, actually calculating these tensor derivatives, storing them, and then doing e.g., a tensor-vector multiply, is usually not a good idea for both memory and computation. In this example, storing all these zeros and performing the multiplications is unnecessary.
- If we know the end result is simply an outer product of two vectors, we need not even calculate an additional derivative in this step of backpropagation, or store an extra value (assuming the inputs were previously cached).

$\nabla_x x^T y = y$

$\nabla_x wx = w^T$

$\nabla_w wx$ "ought look like x^T"

MATCH DIMENSIONS.

$\frac{\partial \mathcal{E}}{\partial z} \in \mathbb{R}^m$

$\frac{\partial \mathcal{E}}{\partial W} \in \mathbb{R}^{m \times n}$ ought to look like x^T

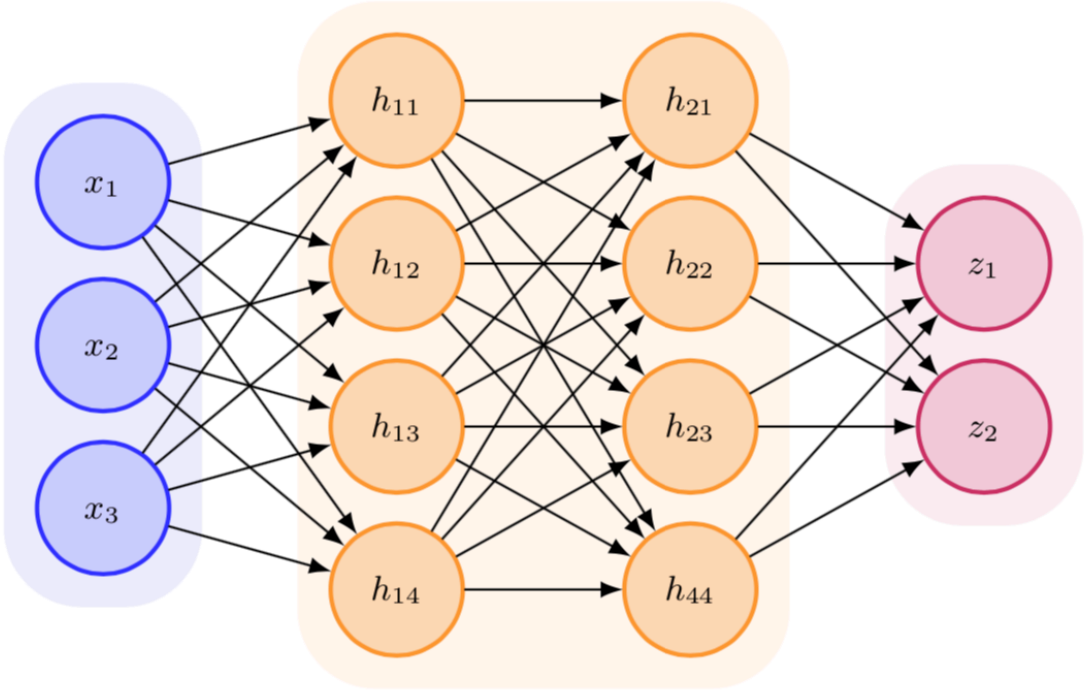
$$\frac{\partial \mathcal{E}}{\partial W} = \frac{\partial z}{\partial W} \cdot \frac{\partial \mathcal{E}}{\partial z}$$

$m \times n$ $\qquad$ $m \times 1$

$$\frac{\partial \mathcal{E}}{\partial W} = \frac{\partial \mathcal{E}}{\partial z} \; x^T$$

$(m \times n) \quad (m \times 1) \quad (1 \times n)$

Back to backpropagation



$y = Wx$

$$\partial L/\partial x = W^T \partial L/\partial y$$

$$\partial L/\partial W = \partial L/\partial y \, x^T$$

$Y = X W$

$(n \times p) \quad (n \times m) \quad (m \times p)$

know $\partial L/\partial Y$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^T$$

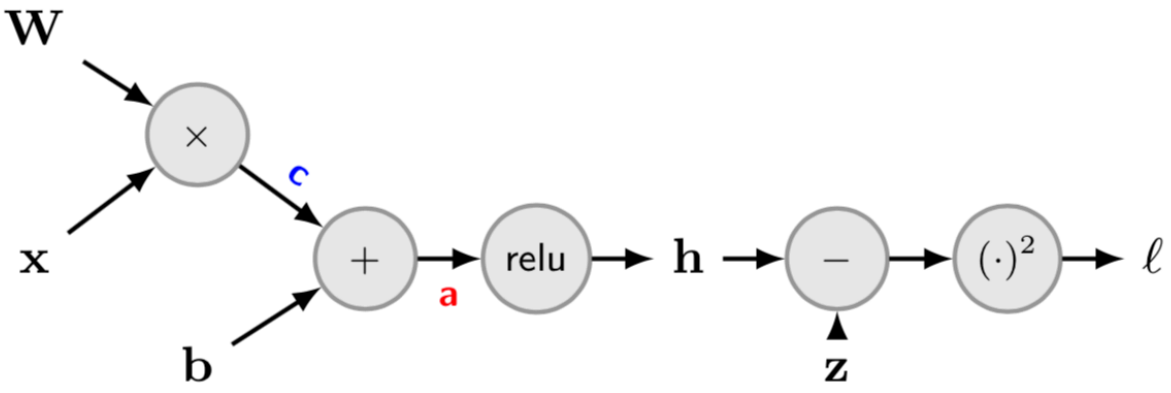
$(m \times p) \quad (n \times m) \quad (n \times p)$

$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}$$

$(m \times n) \quad (m \times p) \quad (p \times n)$

Backpropagation: neural network layer

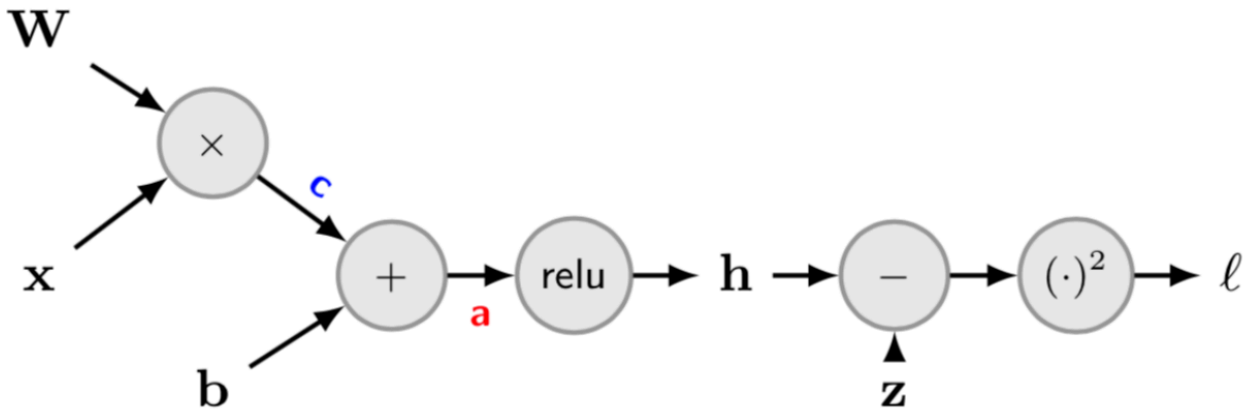
Here, $\mathbf{h} = \mathrm{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{h} \in \mathbb{R}^h$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{h \times m}$, and $\mathbf{b} \in \mathbb{R}^h$. While many output cost functions might be used, let's consider a simple squared-loss output $\ell = (\mathbf{h} - \mathbf{z})^2$ where $\mathbf{z}$ is some target value.



A few things to note:

- Note that $\nabla_{\mathbf{h}} \ell = 2(\mathbf{h} - \mathbf{z})$. We will start backpropagation at $\mathbf{h}$ rather than at ℓ.
- In the following backpropagation, we'll have need for elementwise multiplication. This is formally called a Hadamard product, and we will denote it via $\odot$. Concretely, the ith entry of $\mathbf{x} \odot \mathbf{y}$ is given by $x_i y_i$.

Backpropagation for a neural network layer



Backpropagation: neural network layer (cont.)

Applying the chain rule, we have:

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{a}} &= \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}} \\
\frac{\partial \ell}{\partial \mathbf{c}} &= \frac{\partial \ell}{\partial \mathbf{a}} \\
\frac{\partial \ell}{\partial \mathbf{x}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}}
\end{aligned}$$

A few notes:

- For $\frac{\partial \mathbf{c}}{\partial \mathbf{x}}$, see example in the Tools notes.
- Why was the chain rule written right to left instead of left to right? This turns out to be a result of our convention of how we defined derivatives (using the "denominator layout notation") in the linear algebra notes.
- Though maybe not the most satisfying answer, you can always check the order of operations is correct by considering non-square matrices, where the dimensionality must be correct.

Backpropagation: neural network layer (cont.)

What about $\frac{\partial \ell}{\partial \mathbf{W}}$? In doing backpropagation, we have to calculate $\frac{\partial \mathbf{c}}{\partial \mathbf{W}}$ which is a 3-dimensional tensor.

However, we also intuit the following (informally):

- $\frac{\partial \ell}{\partial \mathbf{W}}$ is a matrix that is $h \times m$.
- The derivative of $\mathbf{W}\mathbf{x}$ with respect to $\mathbf{W}$ "ought to look like" $\mathbf{x}$.
- Since $\frac{\partial \ell}{\partial \mathbf{c}} \in \mathbb{R}^h$, and $\mathbf{x} \in \mathbb{R}^m$ then, intuitively,

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T$$

This intuition turns out to be correct, and it is common to use this intuition. However, the first time around we should do this rigorously and then see the general pattern of why this intuition works.

Now we have the gradients, we can do gradient descent

With the gradients of the parameters, we can go ahead and now apply our learning algorithm, gradient descent.

However, we'll soon find that if we do this naively, performance won't be great **for neural networks** (you'll see this on HW #3).

There are many other important considerations we now need to consider before we can train these networks adequately.