SEIS 764 Artificial Intelligence

Lecture 2: Neural Networks from Scratch

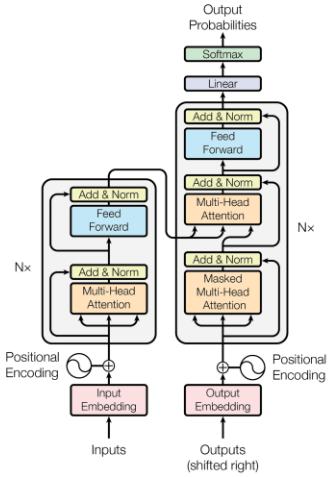
Andrew Van Benschoten, Ph.D.
Adjunct Professor, Department of Software Engineering & Data Science
University of St. Thomas
vanb2168@stthomas.edu

Course Outline (subject to change)

Lecture	**Date**	**Topic**
Lecture 1	February 3	Introduction to artificial Intelligence
Lecture 2	February 10	Neural networks from scratch
Lecture 3	February 17	Convolutional Neural Networks **(Quiz 1 due)**
Lecture 4	February 24	Overfitting & transfer learning **(Hw 1 due)**
Lecture 5	March 3	Recurrent Neural Networks **(Quiz 2 due)**
Lecture 6	March 10	Natural Language Processing: Part I (**Hw 2 due)**
Lecture 7	March 17	**Exam One**
No Class	March 24	
Lecture 8	March 31	Natural Language Processing: Part II
Lecture 9	April 7	Transformers **(Hw 3 due)**
Lecture 10	April 14	Large Language Models: Part I **(Quiz 3 due)**
Lecture 11	April 21	Large Language Models: Part II **(Hw 4 due)**
Lecture 12	April 28	AI Agents **(Quiz 4 due)**
Lecture 13	May 5	*Introduction to AI careers* **(Hw 5 due)**
Lecture 14	May 12	**Exam Two**

The (re)surgence of deep learning

- 2014: “basically all Computer Vision is NNs”
- 2016: AlphaGo
- 2017: Transformers (BERT)
- 2020: GPT-3
- 2021: DALL-E/GH Copilot
- 2022: ChatGPT



What's the "best" ML algorithm?

"From 2016 to 2020, the entire machine learning and data science industry has been dominated by two approaches: deep learning and gradient boosted trees. Specifically, gradient boosted trees is used for problems where structured data is available, whereas deep learning is used for perceptual problems such as image classification. … These are the two techniques you should be most familiar with in order to be successful in applied machine learning today."

– Francois Chollet

Deep learning is pushing boundaries

What are some of the most advanced applications of deep learning today?

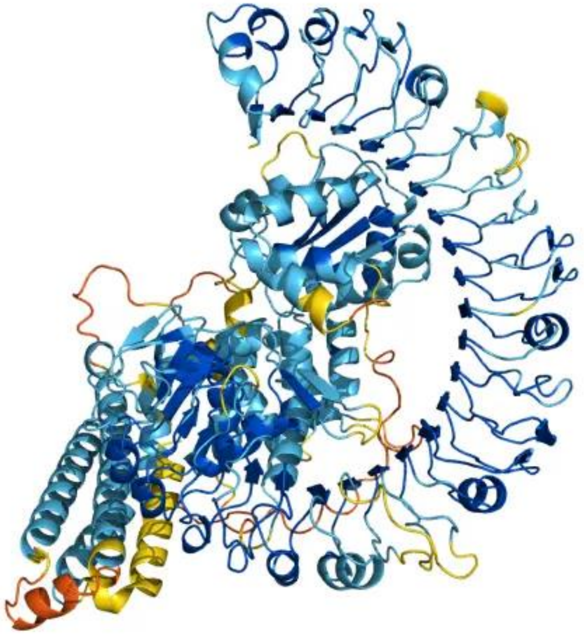
Some of the most advanced applications of deep learning today are transforming industries and pushing the boundaries of what machines can do. Here are key areas where deep learning is making cutting-edge contributions:

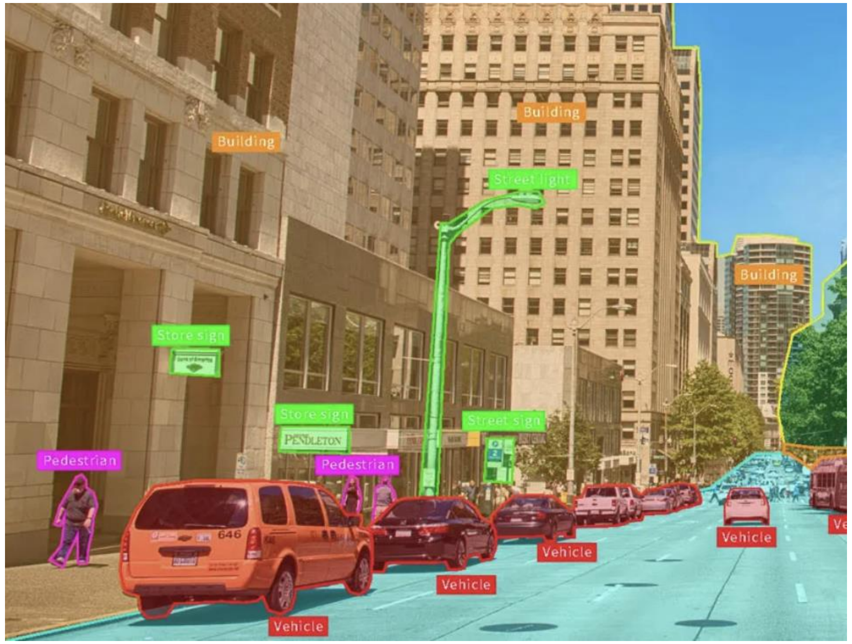
1. Multimodal Generative AI

- **Examples**: OpenAI's GPT-4o, Google's Gemini, Meta's LLaVA.
- **Description**: Models that can understand and generate text, images, audio, and video from a single architecture.
- **Impact**: Enabling more natural human-computer interaction, advanced content creation, and digital assistants that can process multiple input types at once.

2. Autonomous Systems

- **Examples**: Tesla Full Self-Driving (FSD), Waymo, Skydio drones.



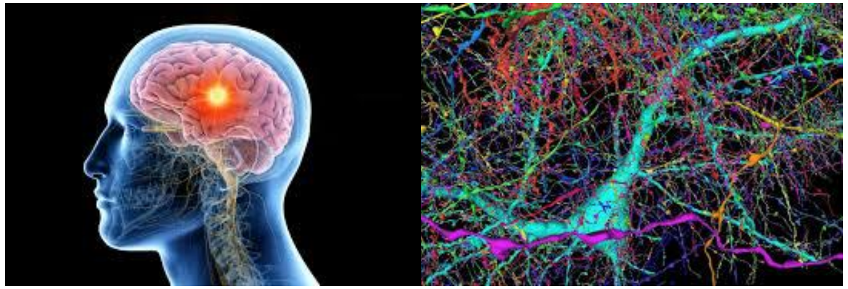


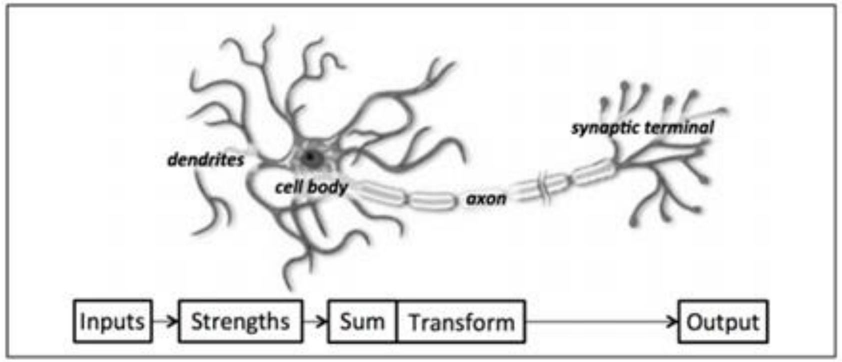
The neuron

- The building block of the human brain

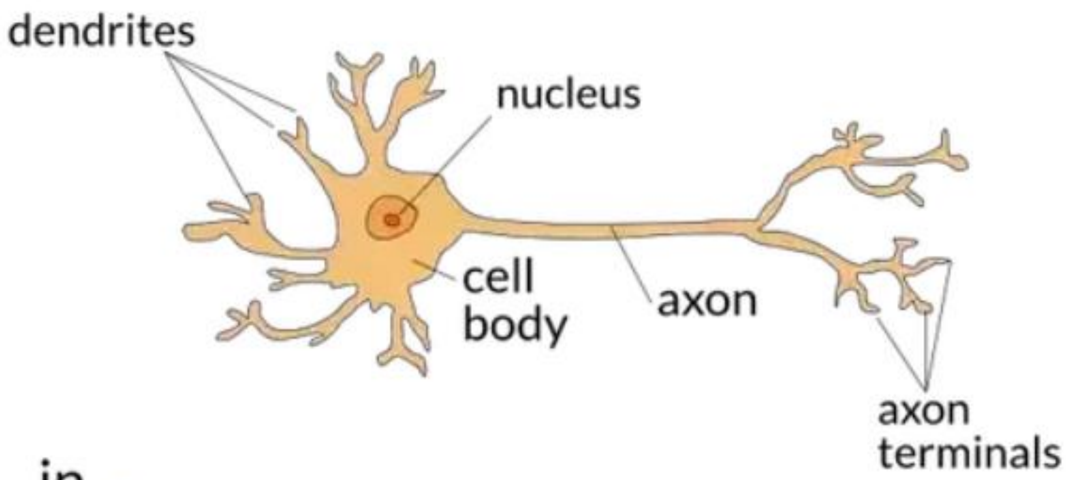
- A piece of brain the size of a grain of rice contains > 10,000 neurons, each of which forms ~ 6,000 connections.

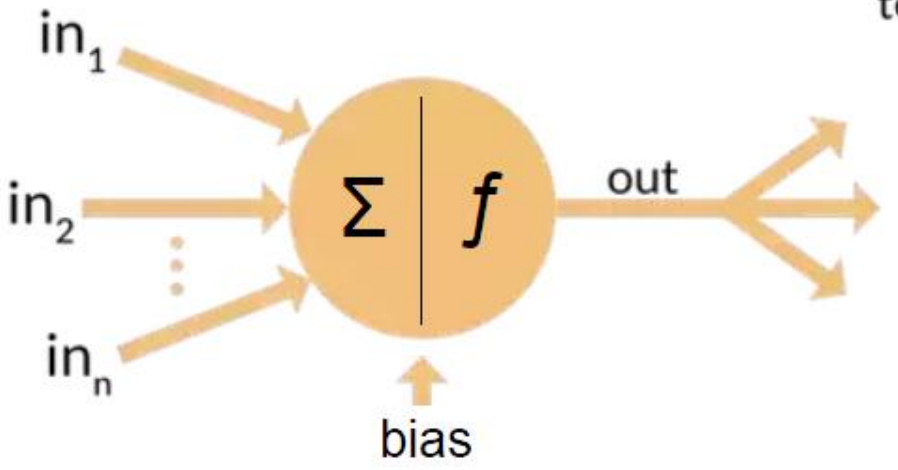
- *Neurons are optimized to receive information from other neurons, process this information in a unique way, and send the result to other cells.*





How do you build an artificial neuron?





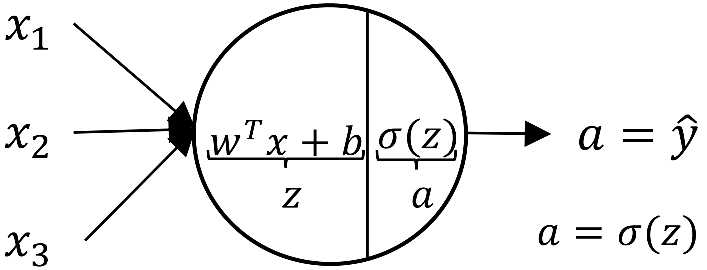
1) *Receive inputs*

2) *Modulate signals*

3) *Apply activation function*

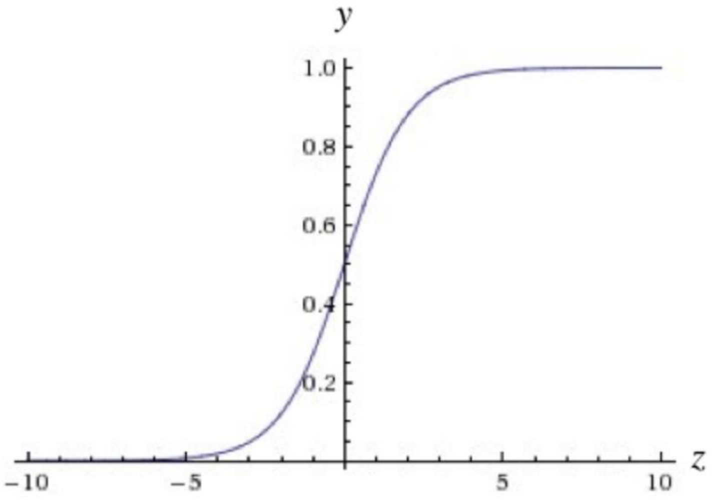
How do you build an artificial neuron?

Closer look



$$a = \sigma(z)$$

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



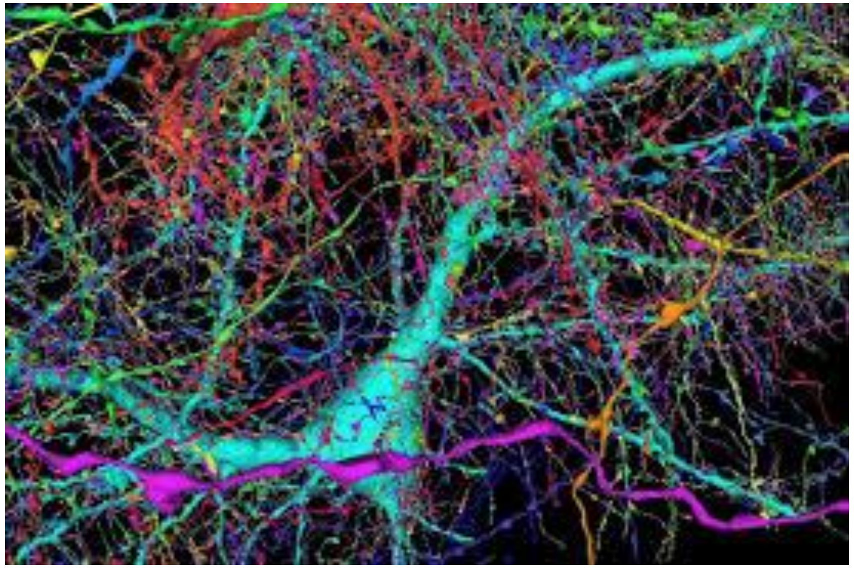
1) *Receive inputs*

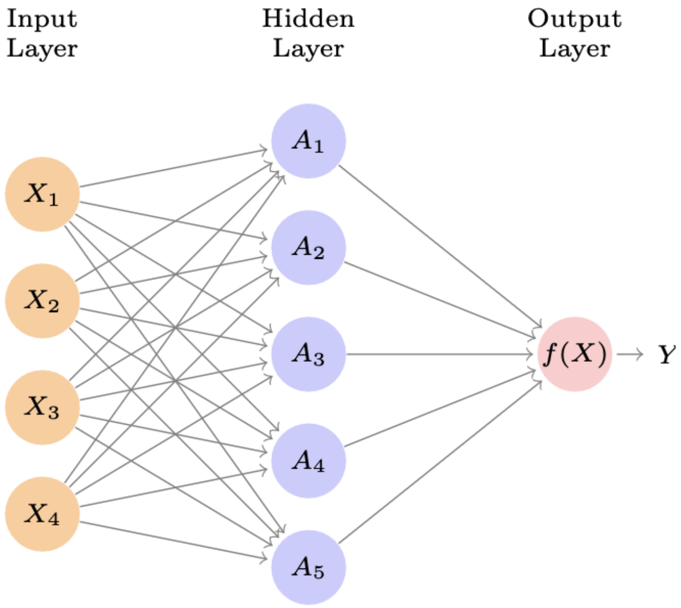
2) *Modulate signals*

3) *Apply activation function*

How do you build an artificial brain?

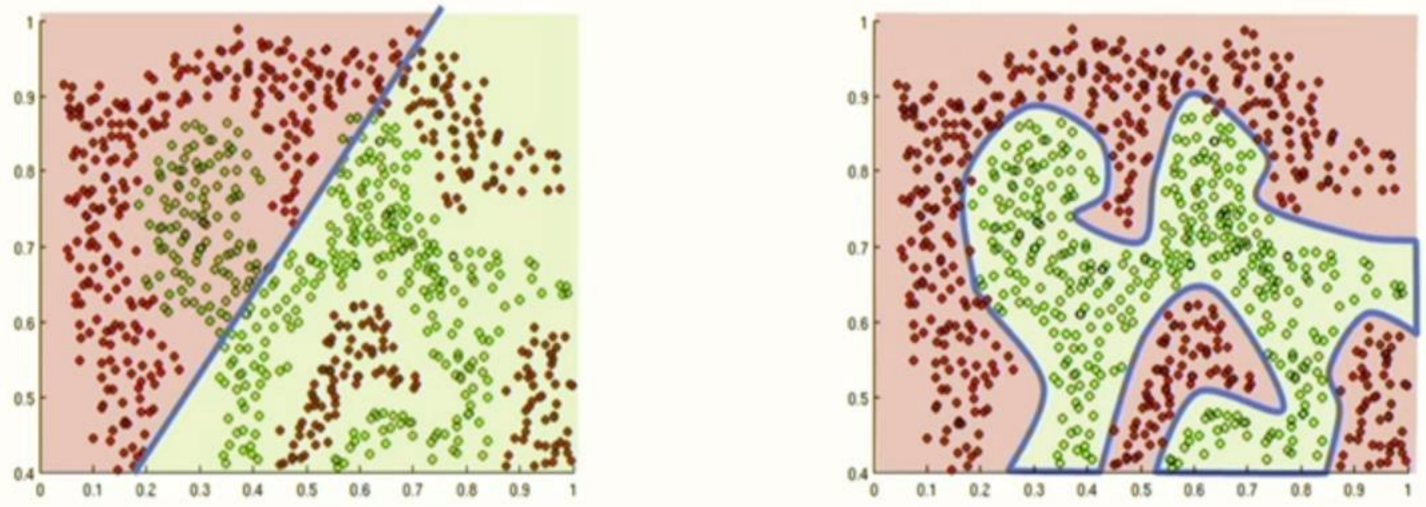
Layer neurons to capture the brain’s interconnectedness





Why is deep learning so powerful?

1) Non-linear boundaries



Linear activation functions produce linear decisions no matter the network size

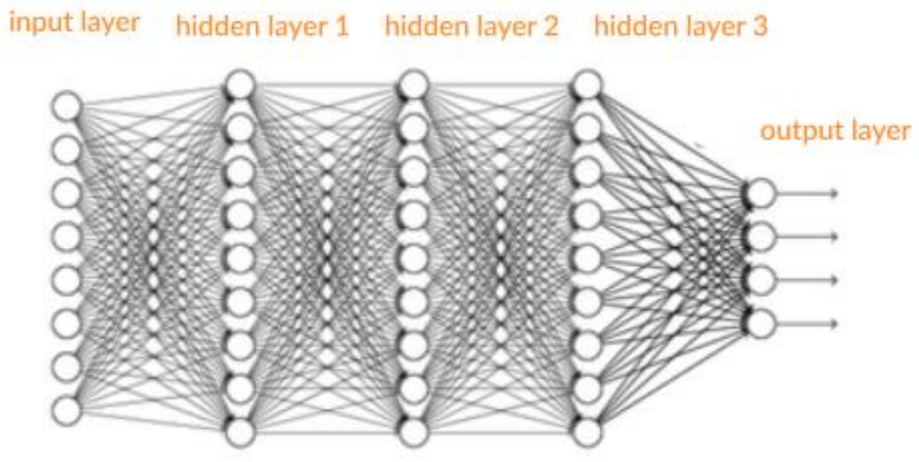
Non-linearities allow us to approximate arbitrarily complex functions

Why is deep learning so powerful?

2) Automatic feature engineering

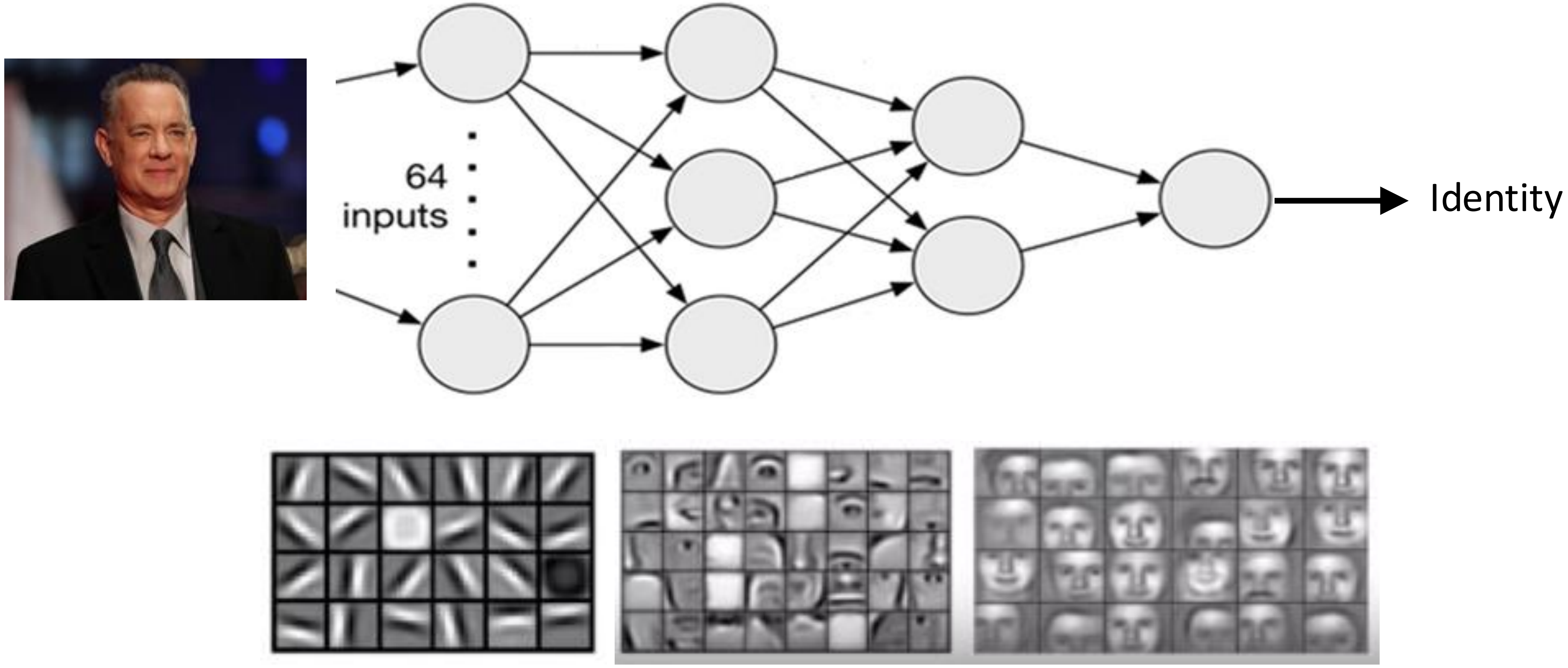
Non-deep learning

$$If\ 2\ inputs:\ x_1,\ x_2,\ x_1^{\,2},\ x_2^{\,2},\ x_1x_2$$
$$If\ 3\ inputs:\ x_1,\ x_2,\ x_3,\ x_1^{\,2},\ x_2^{\,2},\ x_3^{\,2},\ x_1x_2,\ x_1x_3,\ x_2x_3$$



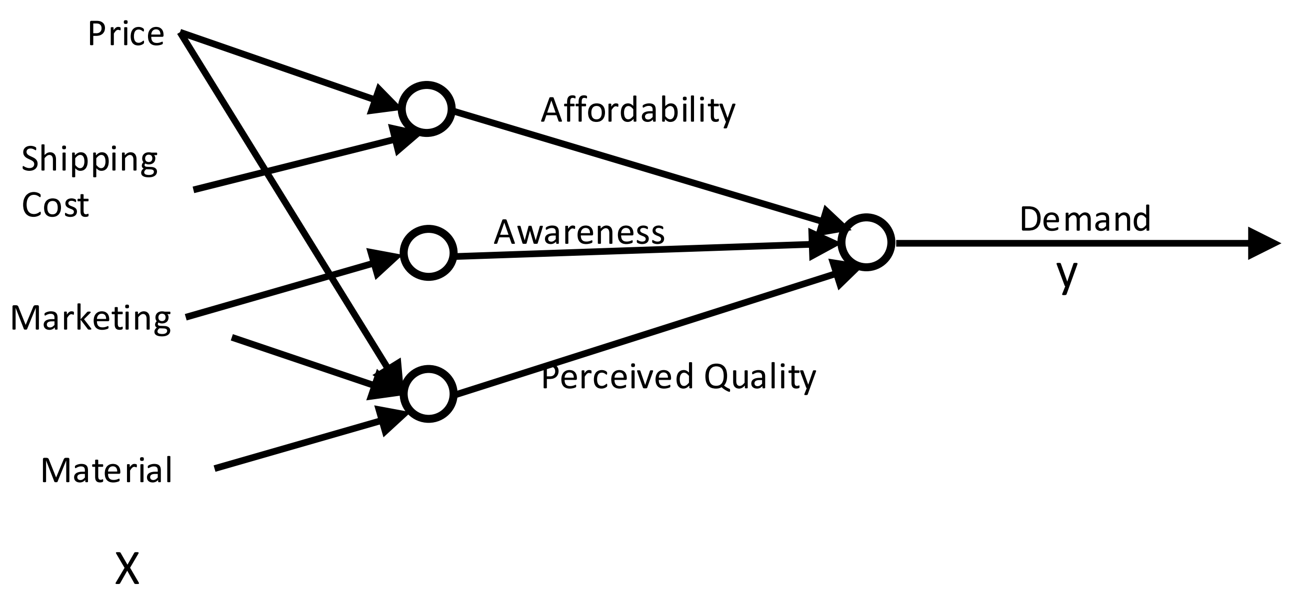
Why is deep learning so powerful?

3) Hierarchical feature learning

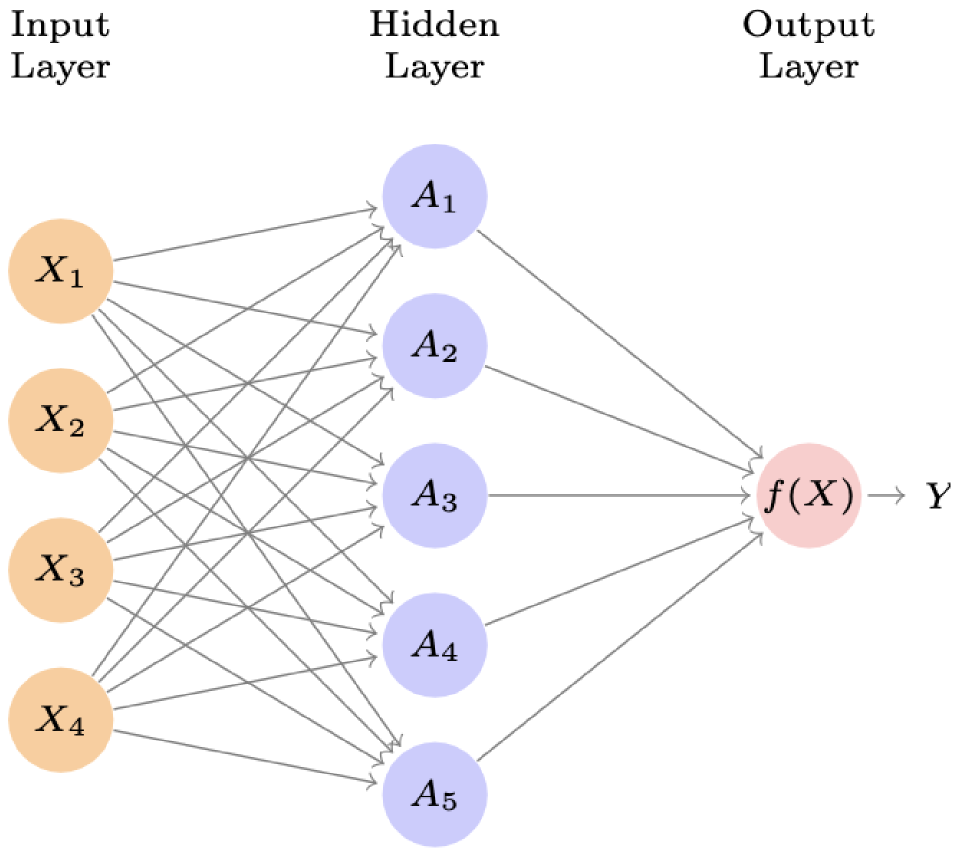


Hierarchical feature learning

3) Hierarchical feature learning



General NN architecture

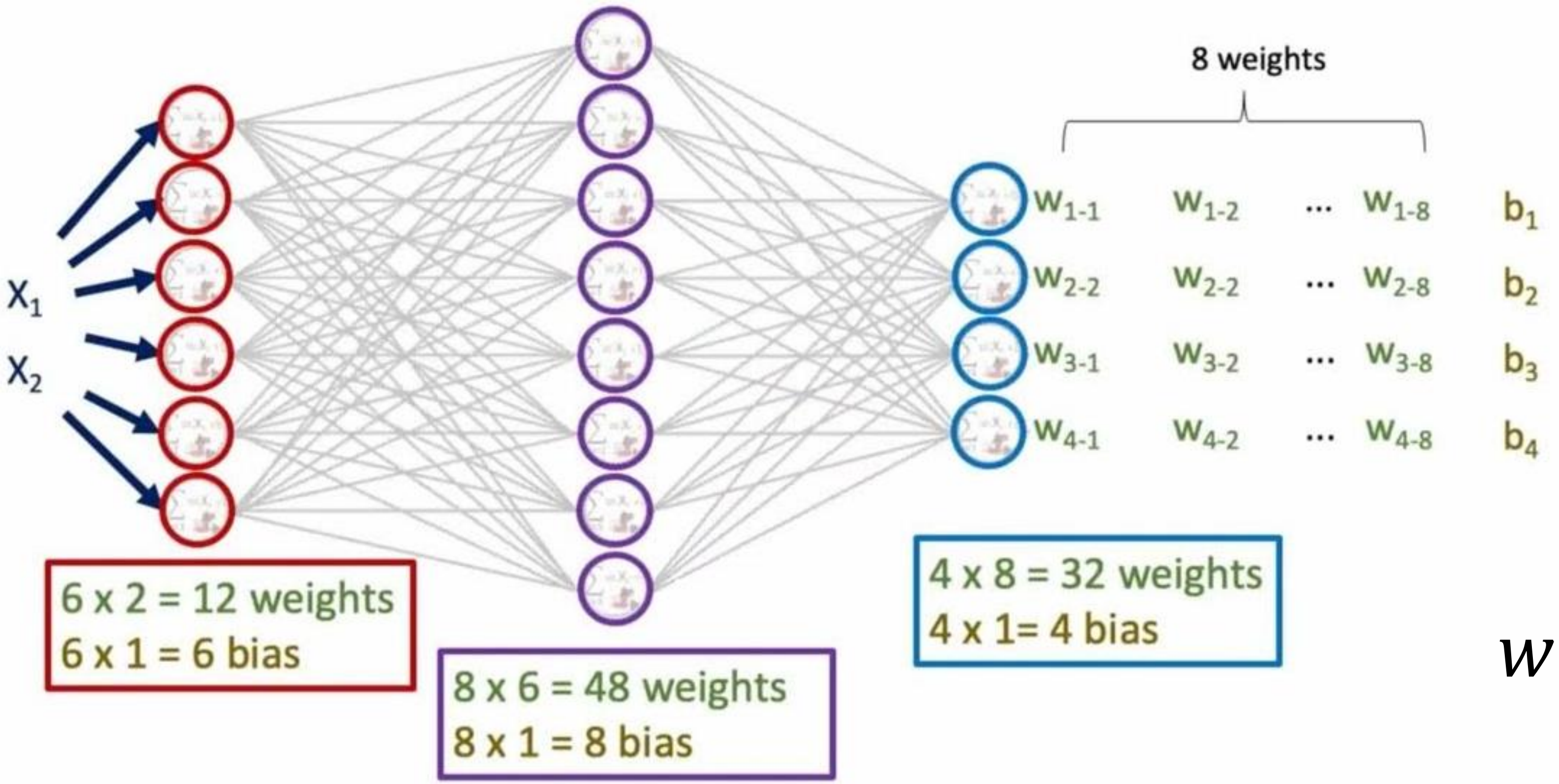


<u>Input</u>: *loads the data*

<u>Hidden</u>: *non-linear transformations, feature & hierarchy learning*

<u>Output</u>: *returns appropriate values (numerical, probabilities)*

Neural nets have *lots* of parameters



$$w^T x + b$$

Dr Anne Hsu

Parameter numbers get <u>very</u> big <u>very</u> fast…

How are neural networks built?

- Gradient descent (usually mini-batch)
- Loss functions: MSE or binary cross-entropy

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

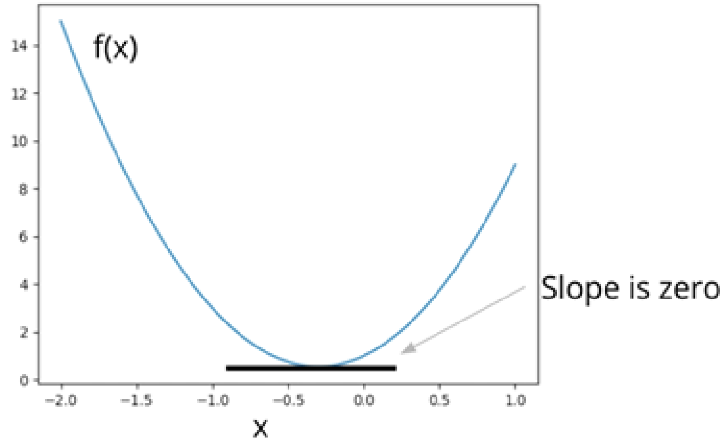
- “Epochs”: *one <u>complete</u> forward and backward pass over all the training data*
  - Forward pass: calculate y(i) and the error for each example
  - Backward pass: weights are updated to minimize error

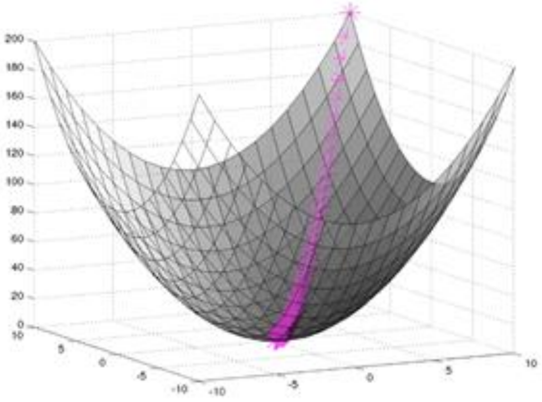
Gradient descent

$$L = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$$

$$L(\beta_0, \beta_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_i)^2$$





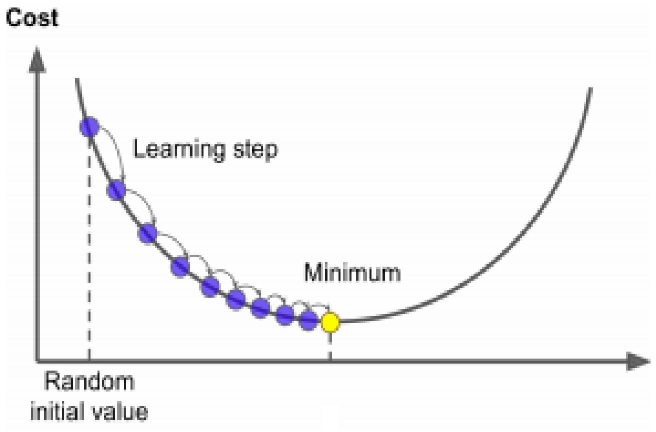
$$\nabla_w J = \left(\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_D}\right)$$

$$Set: \ \nabla_w J = 0, \ solve\ for\ w$$

Gradient Descent

We typically can't solve $\nabla_w J = 0$ *directly…*

… but we can iterate our way there using gradient descent



$$\beta_0 := \beta_0 - \alpha \cdot \frac{\partial J}{\partial \beta_0}$$

$$\beta_1 := \beta_1 - \alpha \cdot \frac{\partial J}{\partial \beta_1}$$

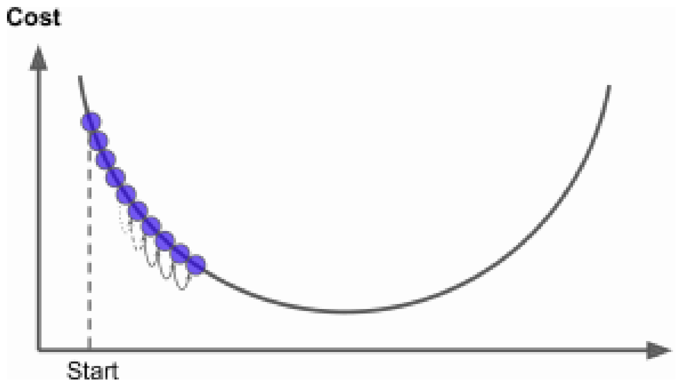
- $\frac{\partial J}{\partial \beta_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_i))$
- $\frac{\partial J}{\partial \beta_1} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_i))\, x_i$

The gradient at each step is the "steepest way down the hill"

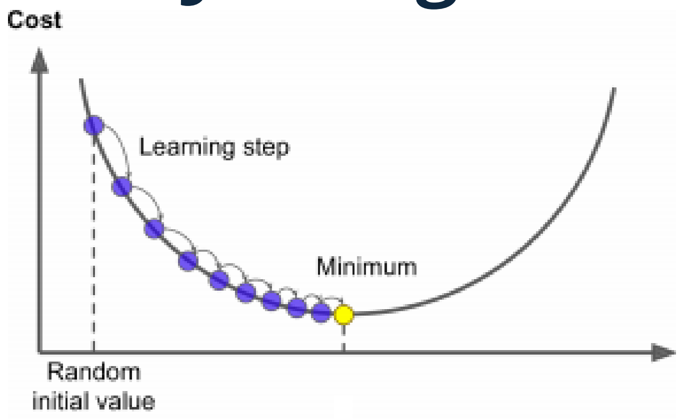
Learning rate

The learning rate is your "step size"

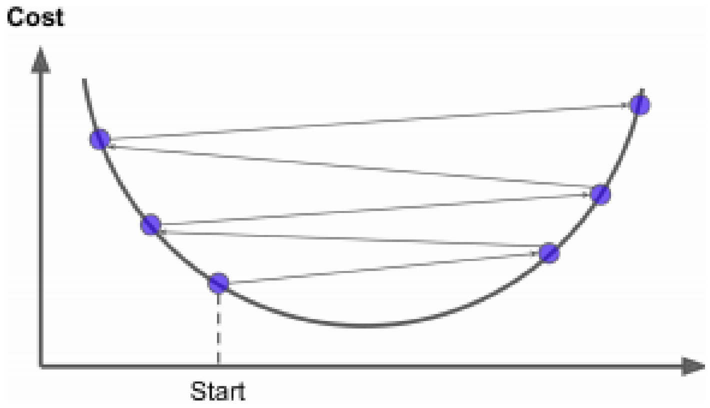
Too small



Just right

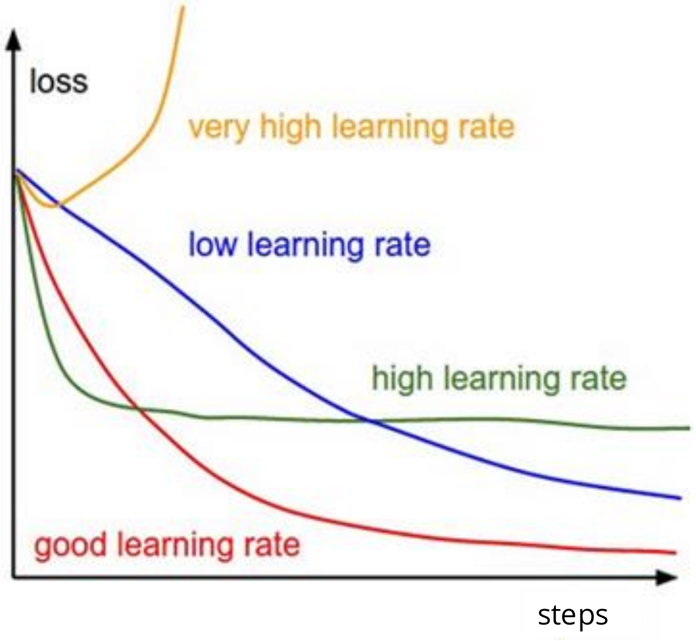


Too large



Learning rate

The learning rate is your “step size”



Loss function != performance metric

A loss function measures how well your predictors $\hat{y}^{(i)}$ match the ground truth $y^{(i)}$…and is differentiable!

<u>Regression Loss Function</u>

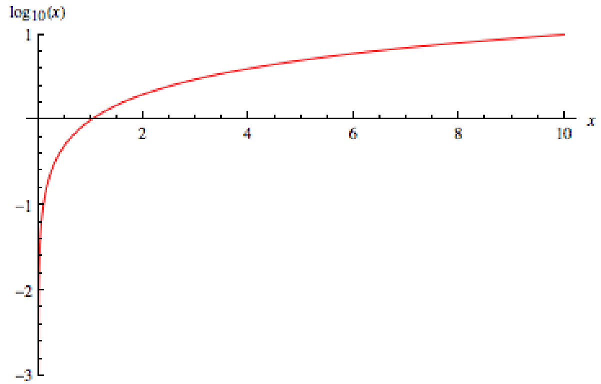
"Mean Squared Error"

$$L = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

<u>Classification Loss Function</u>

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}\log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})]$$

- If $y^{(i)} = 1$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(\hat{y}^{(i)})$ where $\log(\hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 1
- If $y^{(i)} = 0$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(1 - \hat{y}^{(i)})$ where $\log(1 - \hat{y}^{(i)})$ and $\hat{y}^{(i)}$ should be close to 0



Gradient descent flavors

Batch

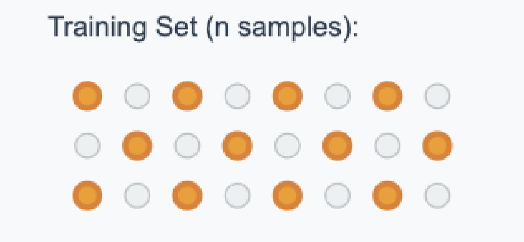
Smooth convergence

Best for small datasets

Can get stuck in local minima

More deterministic

Mini-batch



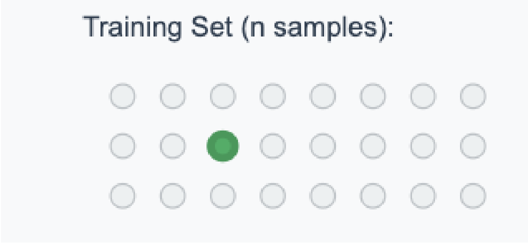
Balances extremes

Parallelization advantage

Extra hyperparameter

Default in deep learning

Stochastic



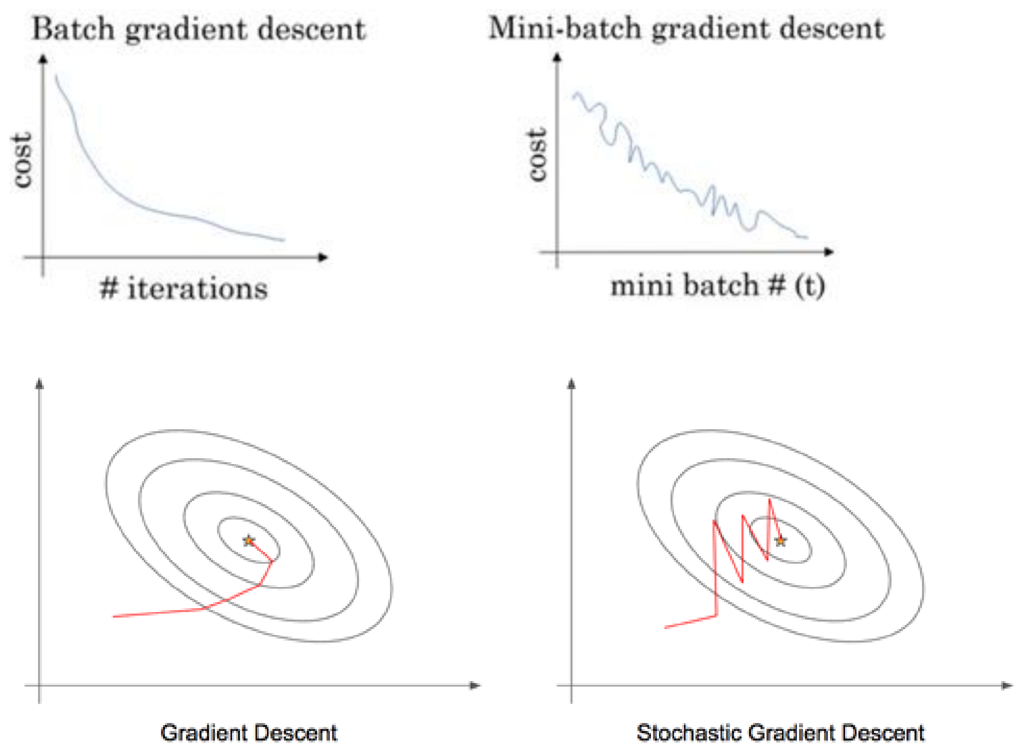
Fast performance

Avoids minima traps (random)

Noisy convergence

Large data & online learning

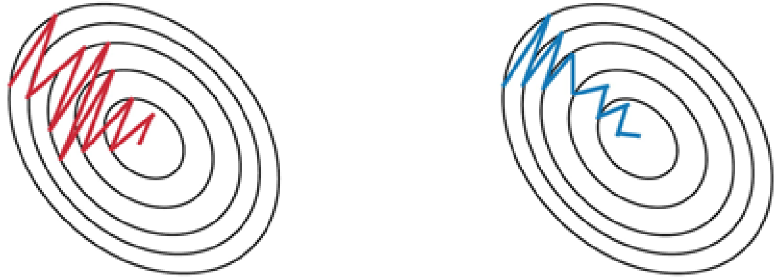
Batch vs mini-batch



https://www.andreaperlato.com/aipost/mini-batch-gradient-descent/

Momentum

Momentum smooths SGD/mini-batch noise



Stochastic Gradient Descent **withhout** Momentum

Stochastic Gradient Descent **with** Momentum

$$g_t = \nabla_\theta f(\theta_t)$$

1) Compute the gradient

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

2) Compute the velocity (moving average)

$$\theta_{t+1} = \theta_t - \alpha v_t$$

3) Update your parameters

RMSprop (Root Mean Square)

- Gradients can vary significantly across parameters (leads to oscillations)
- Ideal system dynamically adjusts large/small gradients have smaller/larger learning rates
- RMSprop accomplishes this!

$$g_t = \nabla_\theta f(\theta_t)$$

1) Compute the gradient

$$v_t = \beta v_{t-1} + (1-\beta) g_t^2$$

2) Compute the velocity (moving average)

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t} + \epsilon} g_t$$

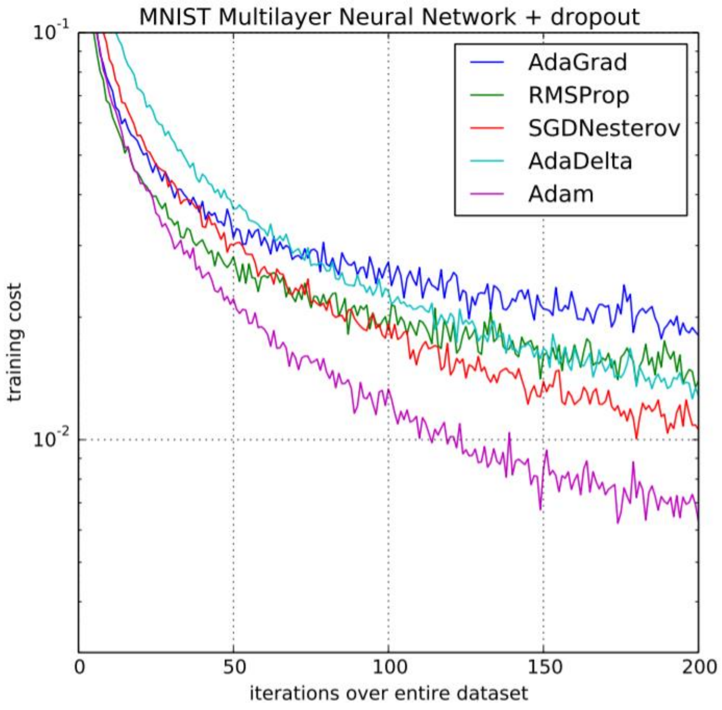
3) Update your parameters

Adam Optimizer = Momentum+RMSprop

- Momentum accelerates learning & preserves directional movement

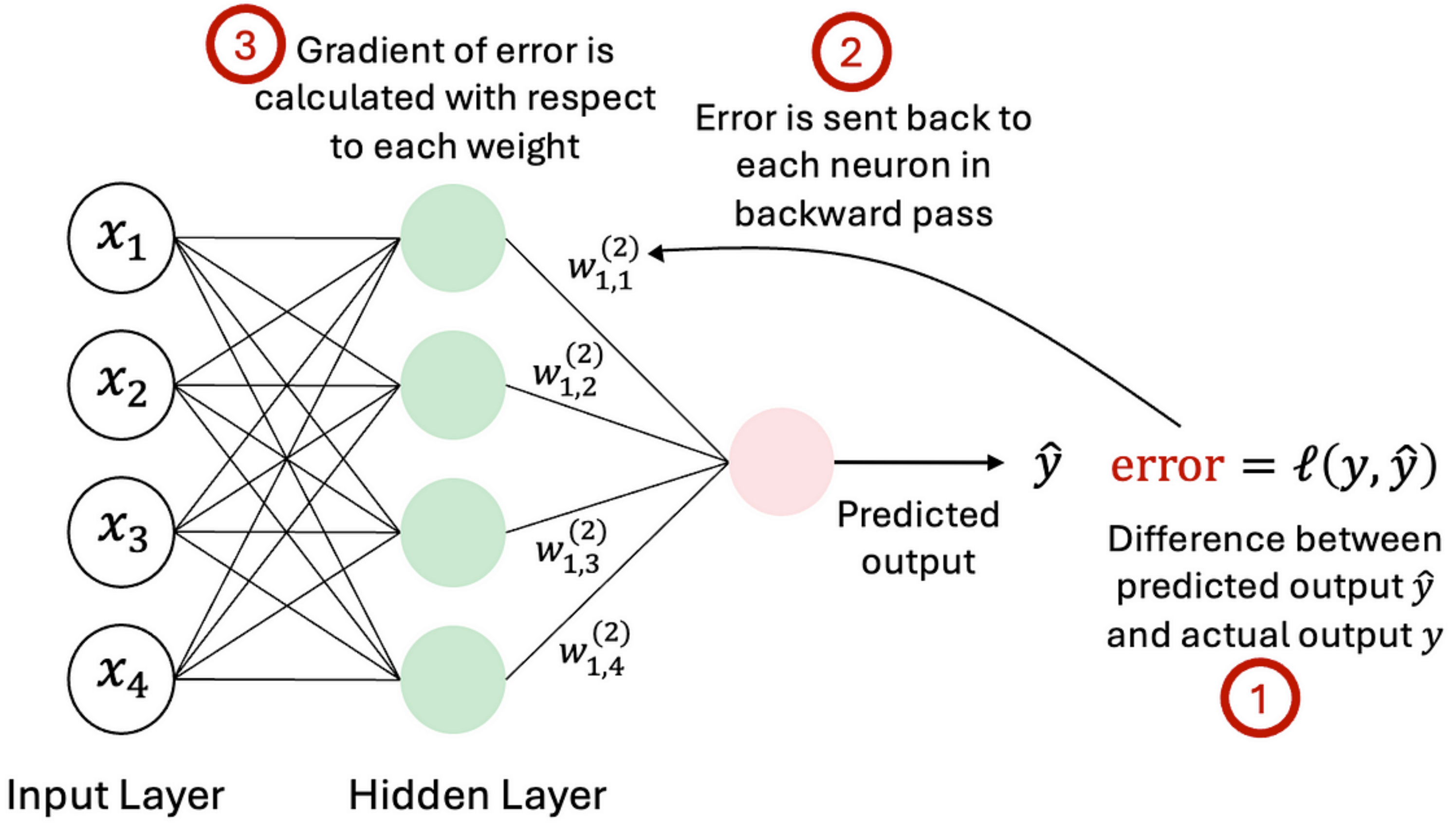
- RMSprop adapts learning rates, preventing large updates in high-gradient areas

- Adam leverages:
  - momentum's acceleration (particularly in flatter areas)
  - RMSprop's adaptive learning rates to prevent instability



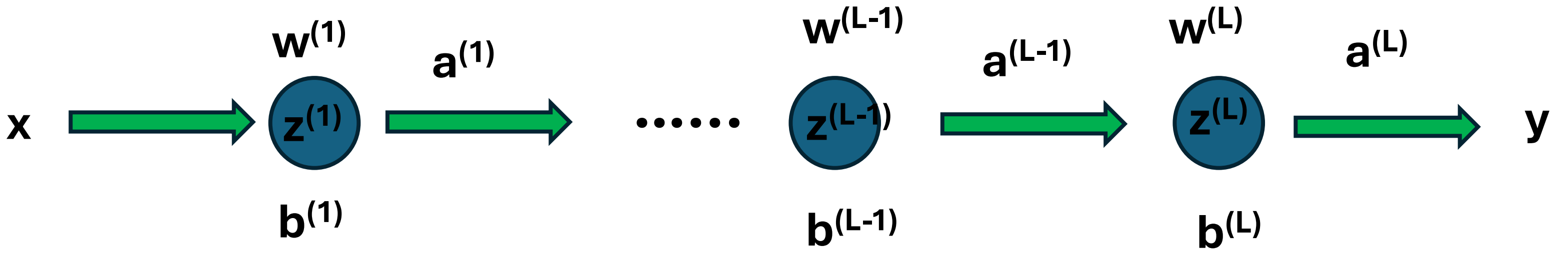
Backpropagation

*We need to find the gradient of the loss function … with respect to *all* of the network's weights and biases*



$$\frac{\partial C_0}{\partial w^{(L)}}$$

Backpropagation: Part I



$$z^{(L)} = w^{(L)} X + b^{(L)}$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma\left(z^{(L)}\right)$$

$$C_0 = \left(a^{(L)} - y\right)^2$$

Backpropagation: Part II

$$\begin{tikzcd} & y \arrow[dl] & & w^{(L)} \arrow[dl] \\ C_0 & a^{(L)} \arrow[l] & z^{(L)} \arrow[l] & a^{(L-1)} \arrow[l] \\ & & & b^{(L)} \arrow[ul] \end{tikzcd}$$

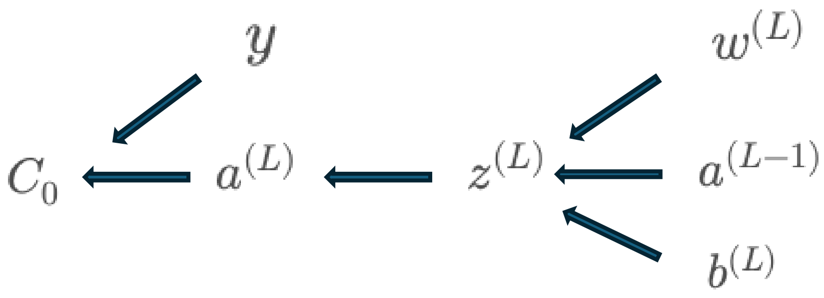
Chain Rule

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$$

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} \quad \Longrightarrow \quad \begin{aligned} C_0 &= \left(a^{(L)} - y\right)^2 \\ z^{(L)} &= w^{(L)} a^{(L-1)} + b^{(L)} \\ a^{(L)} &= \sigma\left(z^{(L)}\right) \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \frac{\partial C_0}{\partial a^{(L)}} &= 2\left(a^{(L)} - y\right) \\ \frac{\partial a^{(L)}}{\partial z^{(L)}} &= \sigma'\left(z^{(L)}\right) \\ \frac{\partial z^{(L)}}{\partial w^{(L)}} &= a^{(L-1)} \end{aligned}$$

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} = a^{(L-1)} \cdot \sigma'\left(z^{(L)}\right) \cdot 2\left(a^{(L)} - y\right)$$

Backpropagation: Part II



Chain Rule

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$$

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

$$C_0 = \left(a^{(L)} - y\right)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma\left(z^{(L)}\right)$$

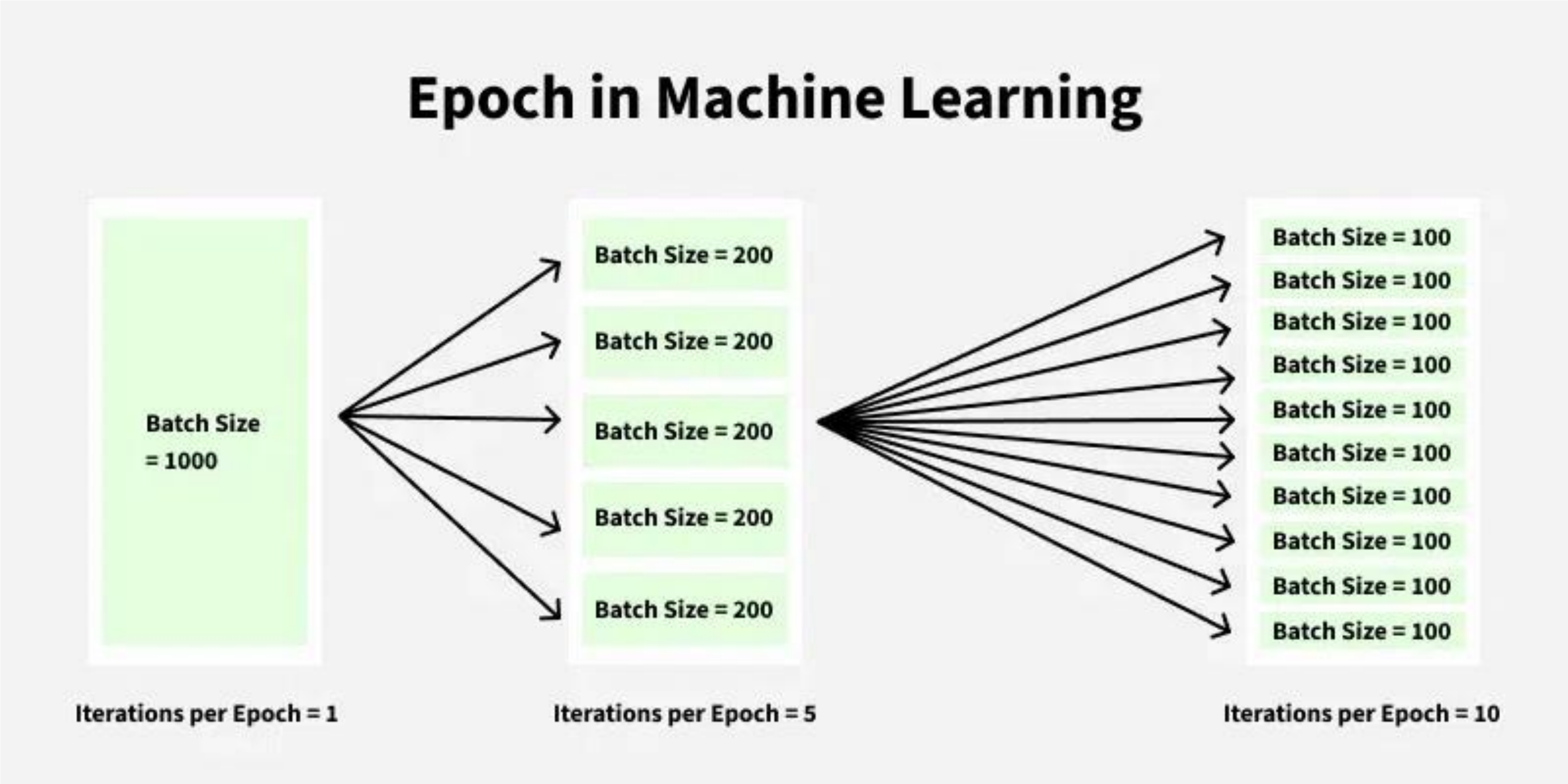
$$\frac{\partial C_0}{\partial a^{(L)}} = 2\left(a^{(L)} - y\right)$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'\left(z^{(L)}\right)$$

$$\frac{\partial C_0}{\partial w^{(L-1)}} = a^{(L-2)} \cdot \sigma'(z^{(L-1)}) \cdot w^{(L)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

GD vs SGD vs Batch GD



GeeksForGeeks

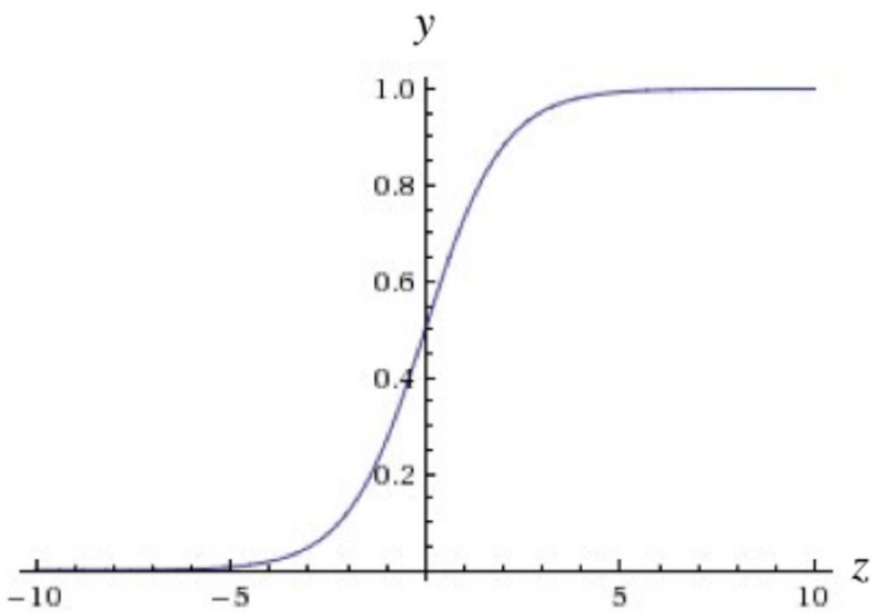
Deep learning = logistic regression?

$$\mathbf{x} = [x_1, x_2, , x_n]^\top$$

$$\mathbf{w} = [w_1, w_2, , w_n]^\top$$

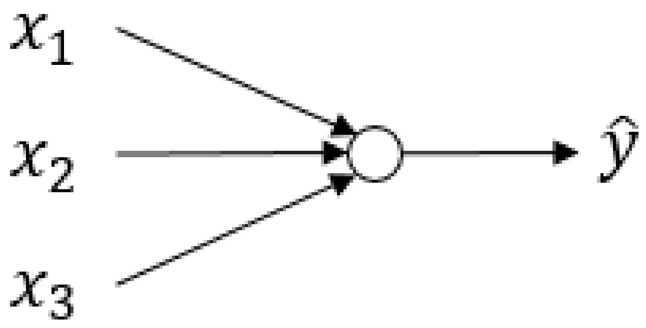
$$z = \mathbf{w}^\top \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b$$

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

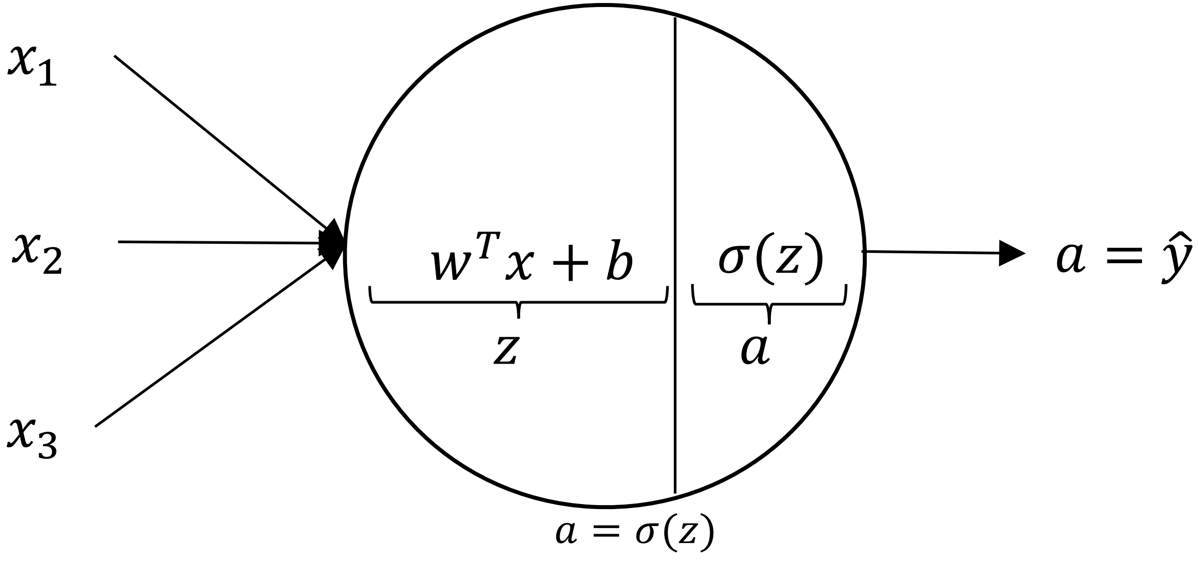


Our goal is to learn weights *W* and bias *b*

This should look familiar…

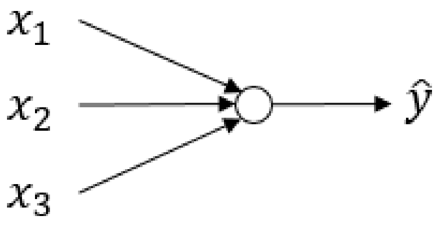


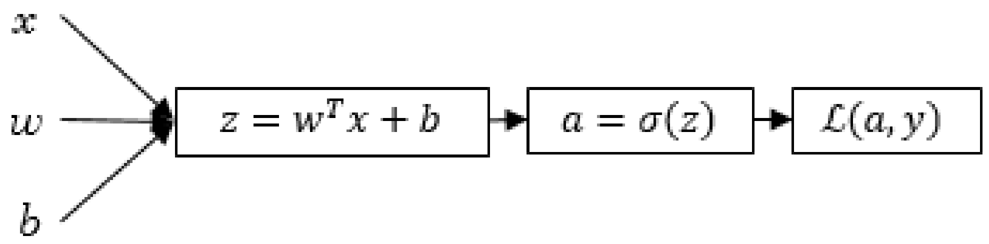
Closer look

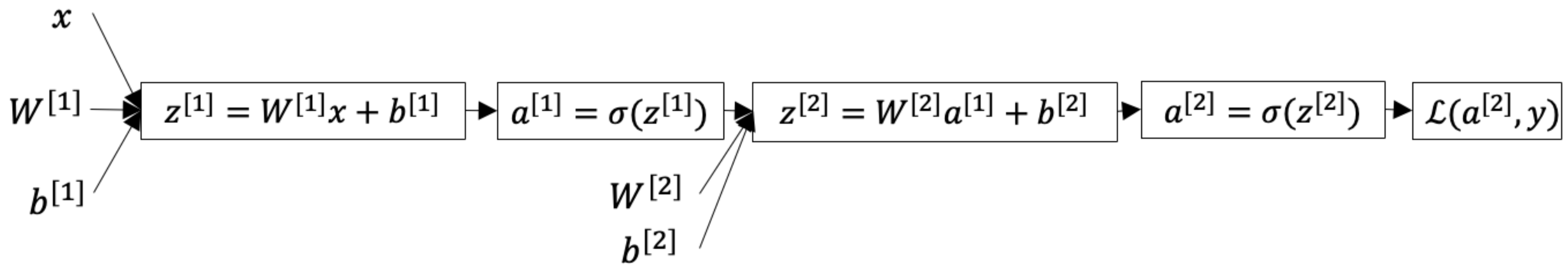


$$f(z) = \frac{1}{1 + e^{-z}}$$

Each neuron is (basically) logistic regression!

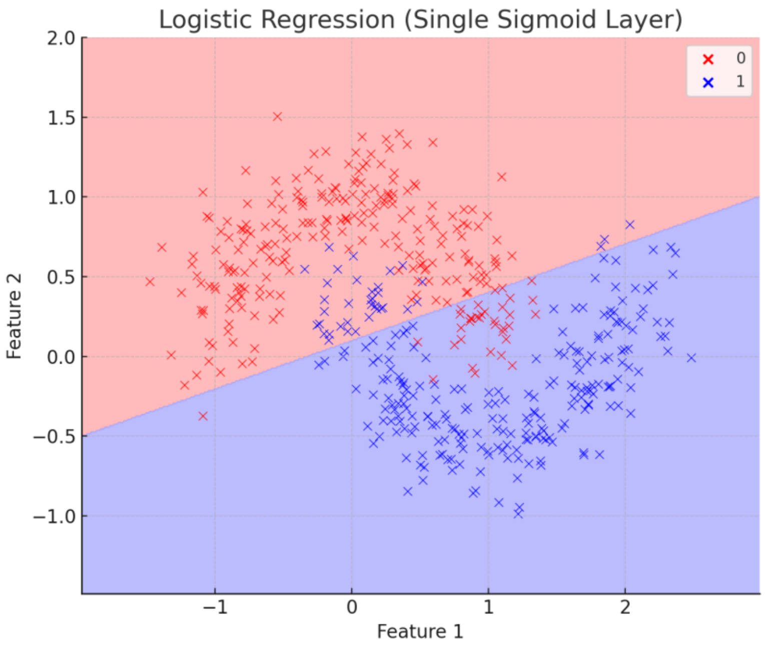


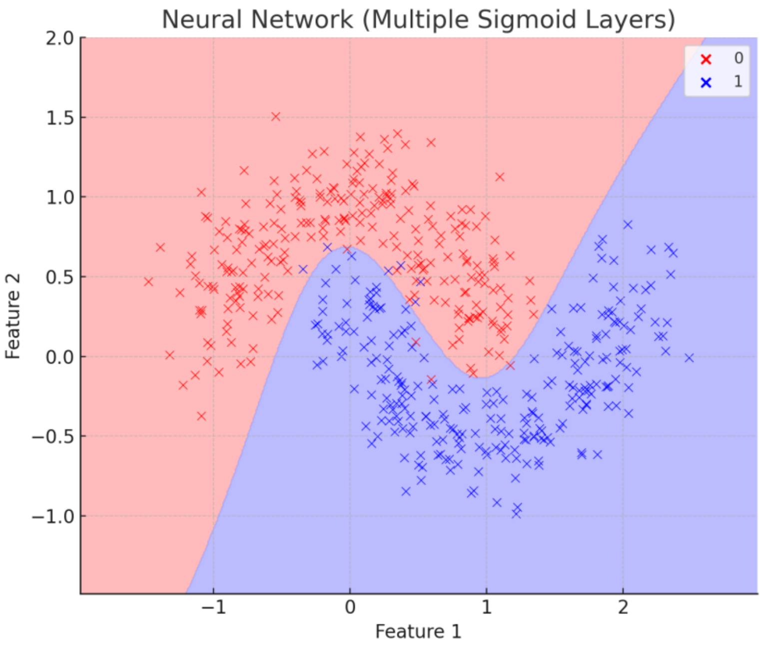


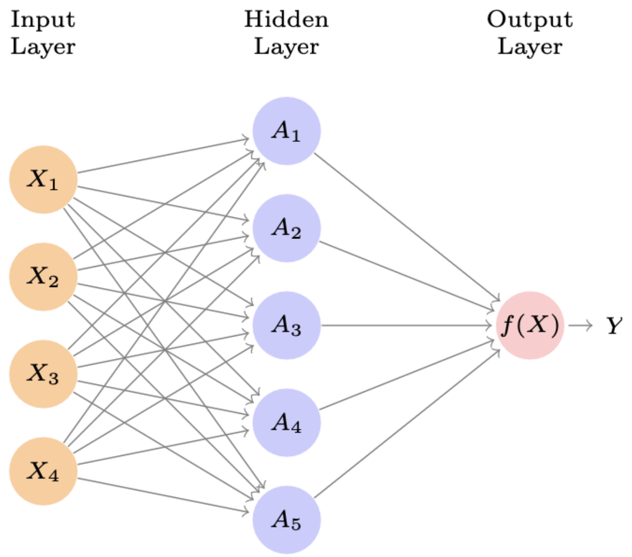


But where is the non-linearity?

Multiple neurons give non-linear boundaries!



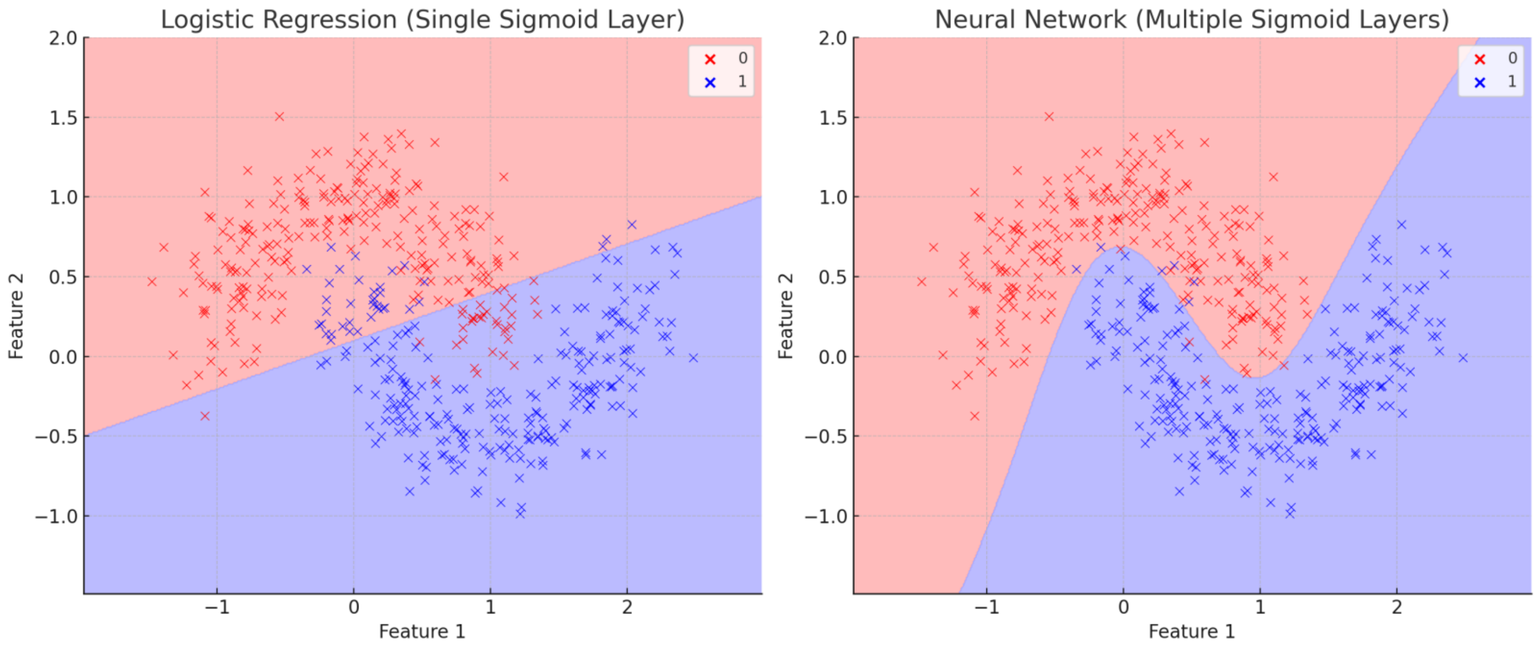




$$W^{(2)} \cdot \sigma(W^{(1)} \cdot x + b^{(1)}) + b^{(2)}$$

But where is the non-linearity?

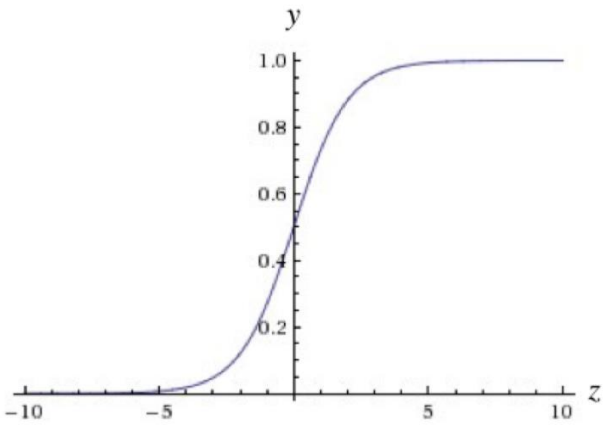
A single neuron is LR, but combined they give non-linear boundaries!



$$W^{(2)} \cdot (W^{(1)} \cdot x + b^{(1)}) + b^{(2)}$$

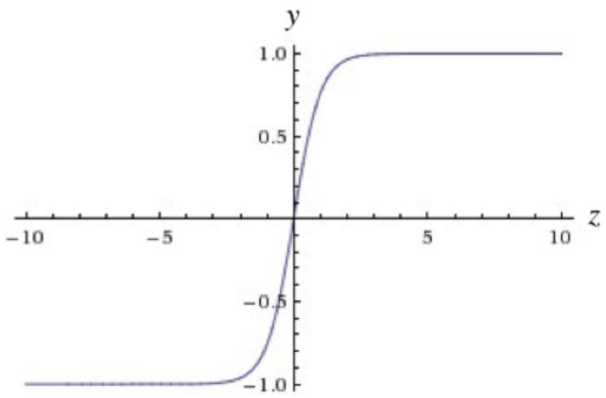
Other activation functions

Sigmoid



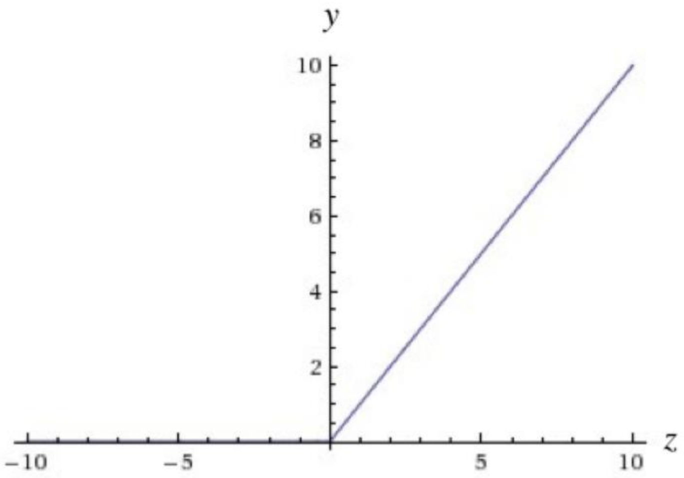
$$f(z) = \frac{1}{1 + e^{-z}}$$

Tanh



$$f(z) = \tanh(z)$$

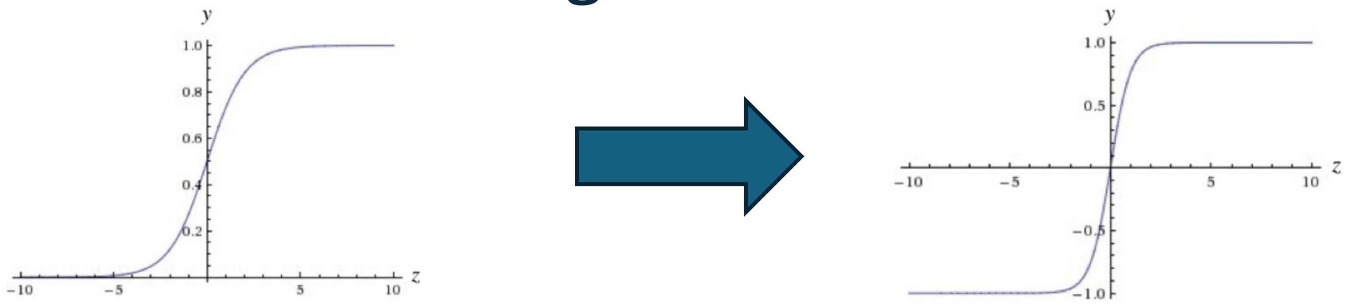
ReLU



$$f(z) = \max(0, z)$$

Issues with sigmoid activation

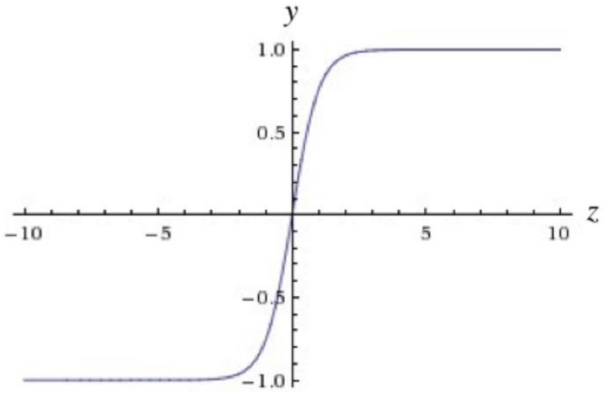
- We want the inputs to be standardized
  - $X_1 = 3{,}527{,}000$ $X_2 = 0.63$
- Sigmoid outputs values centered around 0.5
  - Inputs to the next layer are not centered around zero.
  - Makes model training a *lot* more unstable...

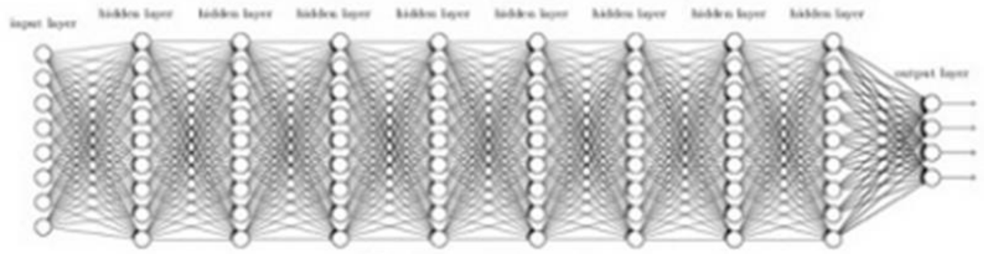


The vanishing gradient problem

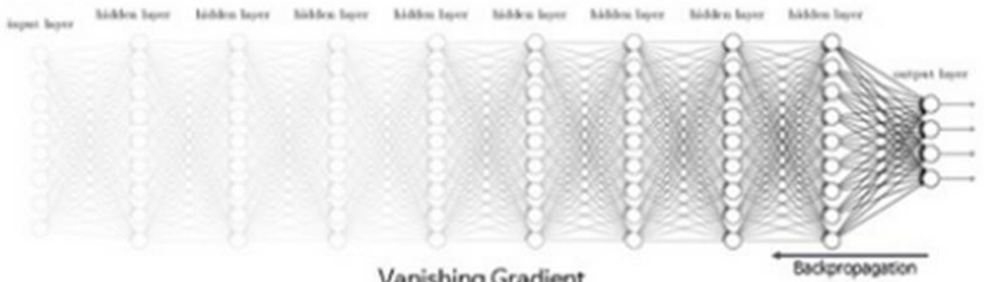
- Tanh is centered on zero, but…

- …both sigmoid & tanh suffer from the *vanishing gradient problem!*





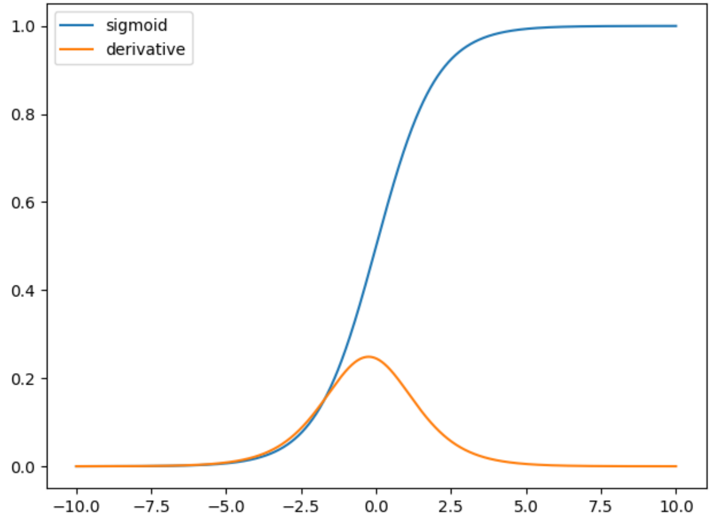
Deep Neural Network



Vanishing Gradient

The vanishing gradient problem

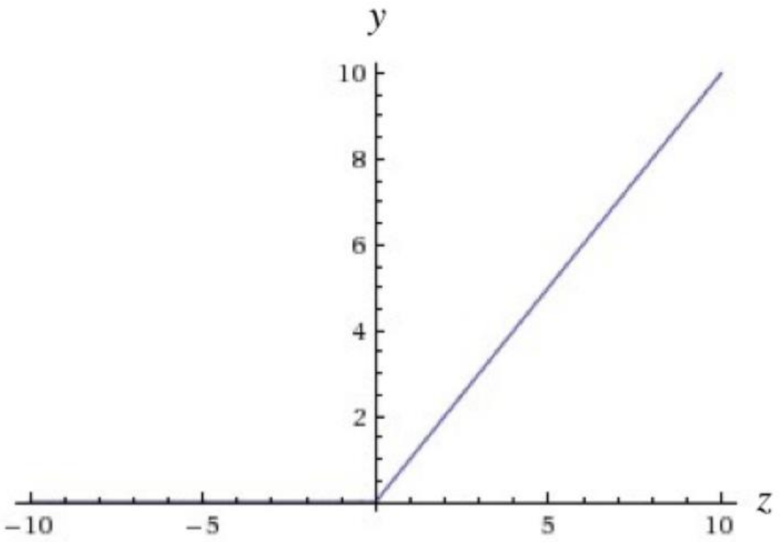
- The derivative of the sigmoid & tanh is ~ zero at most points.
- Max value of derivative is only 0.25!!
- Backpropagation collapses to zero



$$\frac{\partial C_0}{\partial w^{(L-1)}} = a^{(L-2)} \cdot \sigma'(z^{(L-1)}) \cdot w^{(L)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

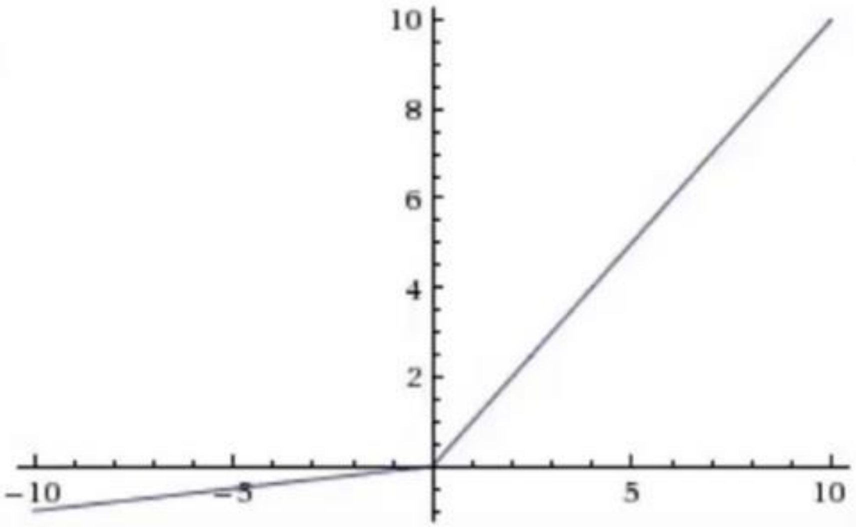
ReLU fixes the problem

Regular



f(z) = z if z >0, 0 elsewhere

“Leaky”



*f(z) = z if z >0, 0.01*z elsewhere*

ReLU is nearly always used for hidden layers

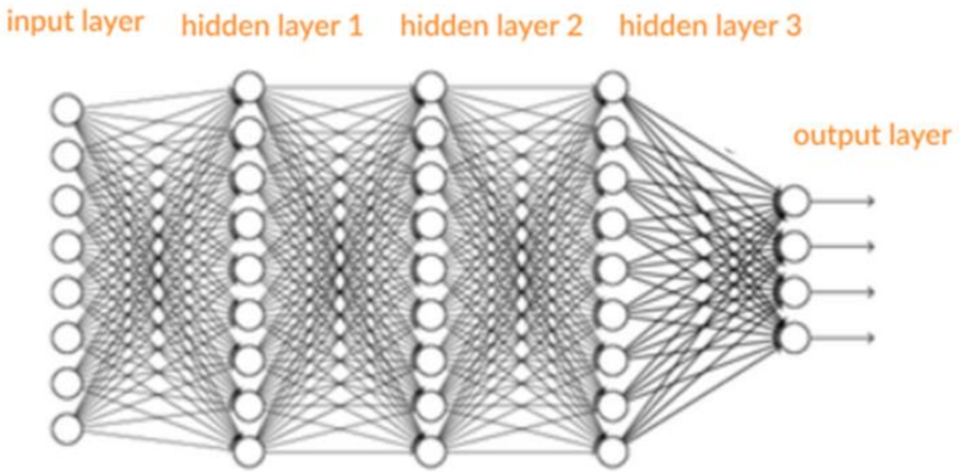
Softmax & multi-class classification

What do we do with the linear transformation at the output layer?

- Regression: *don't need anything*
- Classification: *need probabilities*

Single-class: *sigmoid function*

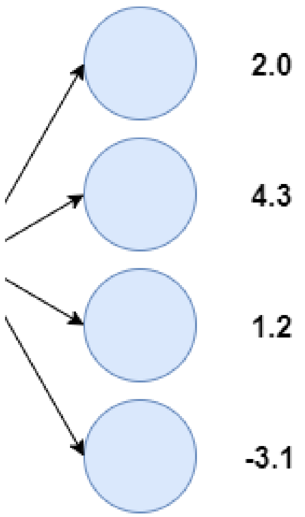
Multi-class: *???*



The "softmax" function converts an output vector of k values into a set of probabilities

$$\text{Softmax}(z_i) = \frac{\exp(z_i)}{\sum_i \exp(z_i)}$$

The softmax activation



$$\frac{exp(x_i)}{\sum_j exp(x_j))} = \frac{exp(2.0)}{exp(2.0)+exp(4.3)+exp(1.2)+exp(-3.1)} \quad 0.087492$$

$$\frac{exp(x_i)}{\sum_j exp(x_j))} = \frac{exp(4.3)}{exp(2.0)+exp(4.3)+exp(1.2)+exp(-3.1)} \quad 0.872661$$

$$\frac{exp(x_i)}{\sum_j exp(x_j))} = \frac{exp(1.2)}{exp(2.0)+exp(4.3)+exp(1.2)+exp(-3.1)} \quad 0.039312$$

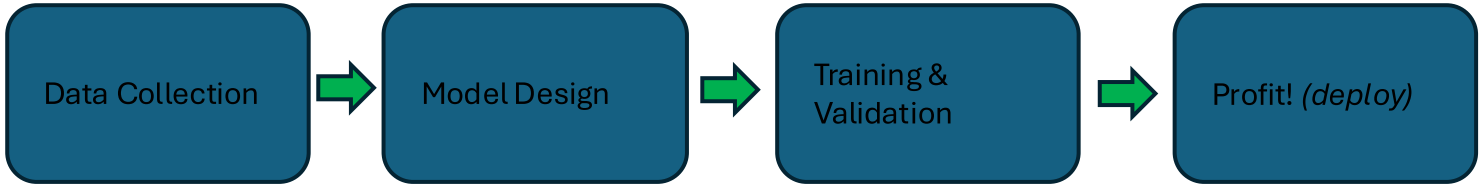
$$\frac{exp(x_i)}{\sum_j exp(x_j))} = \frac{exp(-3.1)}{exp(2.0)+exp(4.3)+exp(1.2)+exp(-3.1)} \quad 0.000292$$

Which activation function to use?

ML Task	No. of neurons in the last layer	Type of Activation
Regression	1	None
Regression for predicting only positive values	1	ReLU
Binary Classification	1	Sigmoid
Multiclass Classification (with k classes)	k	Softmax

Building a neural network

- Same steps as a normal ML problem



Choose your architecture

1) Problem Type

- Do you want classification, regression, or another type of task? Different tasks benefit from different architectures (e.g., CNNs for images, RNNs for sequences).

2) Complexity

- More complex problems may need deeper & wider networks (avoid overfitting!)

3) Training Data Size

- Larger networks generally require more data. If data is limited, simpler architectures may perform better.

4) Computational resources

- Consider the hardware & time available for training. More complex architectures = more computational power.

5) Experimentation

- Be prepared to experiment with different architectures and tweak them based on performance metrics.

Choose your hyperparameters

1) Learning rate

- Controls the size of the steps taken during optimization. Too high = divergence; too low = slow convergence

2) Batch Size

- Defines the number of samples processed before the model updates weights. Larger batches stabilize training, while smaller batches add noise that may improve generalization

3) Number of epochs

- Defines how many full passes are made over the dataset

4) Optimizer

- Algorithm used for weight updates (SGD, Adam, RMSprop)

5) Loss function

- Defined by the type of problem you want to solve

Choose your hyperparameters

1) Learning rate

- Controls the size of the steps taken during optimization. Too high = divergence; too low = slow convergence

2) Batch Size

- Defines the number of samples processed before the model updates weights. Larger batches stabilize training, while smaller batches add noise that may improve generalization

3) Number of epochs

- Defines how many full passes are made over the dataset

4) Optimizer

- Algorithm used for weight updates (SGD, Adam, RMSprop)

5) Loss function

- Defined by the type of problem you want to solve

Two last thoughts

Use simpler models when you can

Deep learning works when:

a) Training set size is massive

b) Interpretability doesn't matter

Google Colab

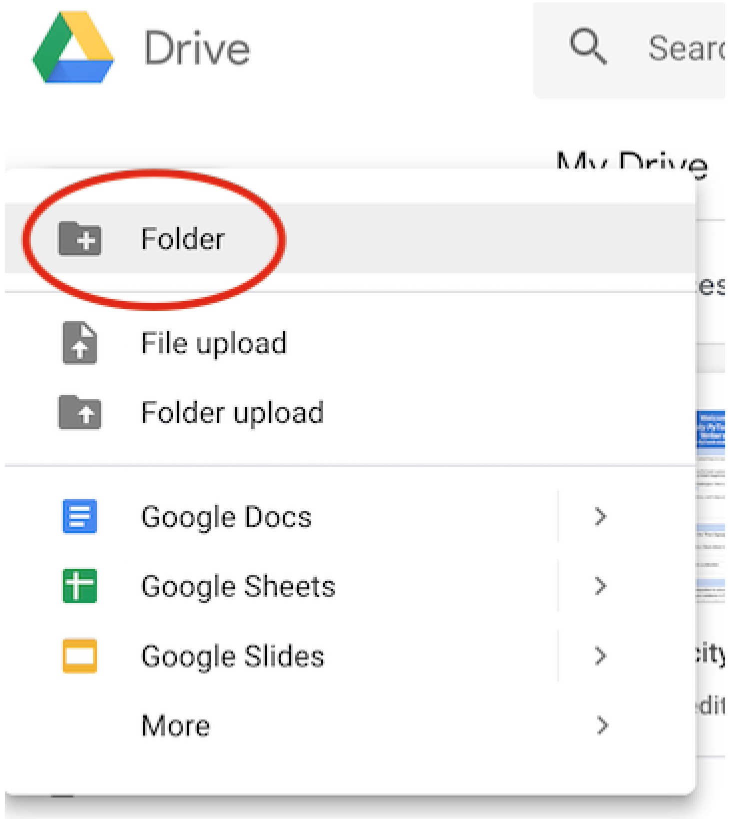
Jupyter notebooks rule AI…and Google has a free service

Solves those wonderful Python dependences…and GPUs!

Any Jupyter notebook solution will do, but could take time

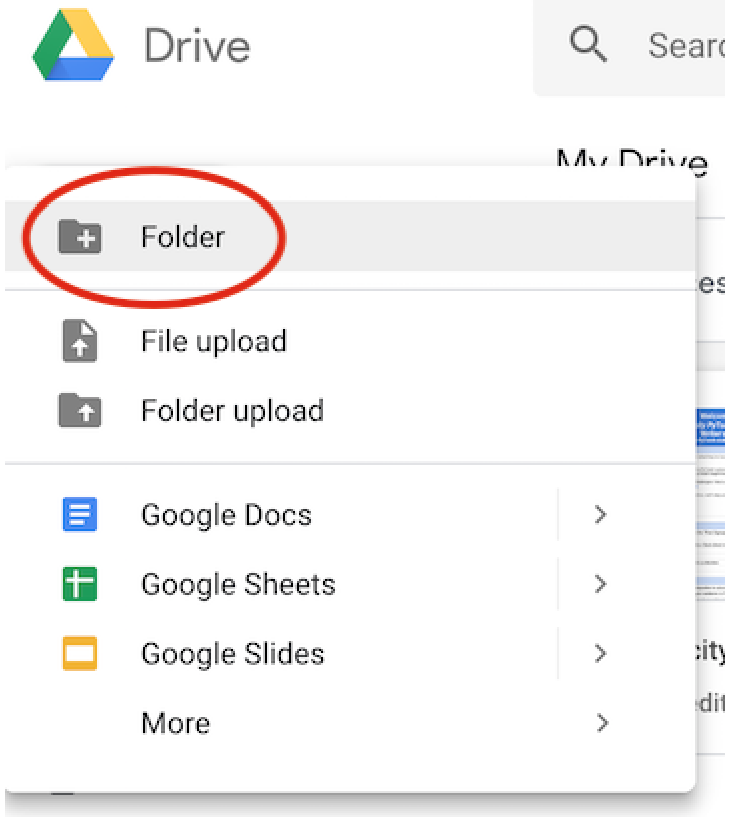
Google Colab

- **Create a folder for your notebooks**
  - You can do that by going to your Google Drive and clicking “New” and then creating a new folder.



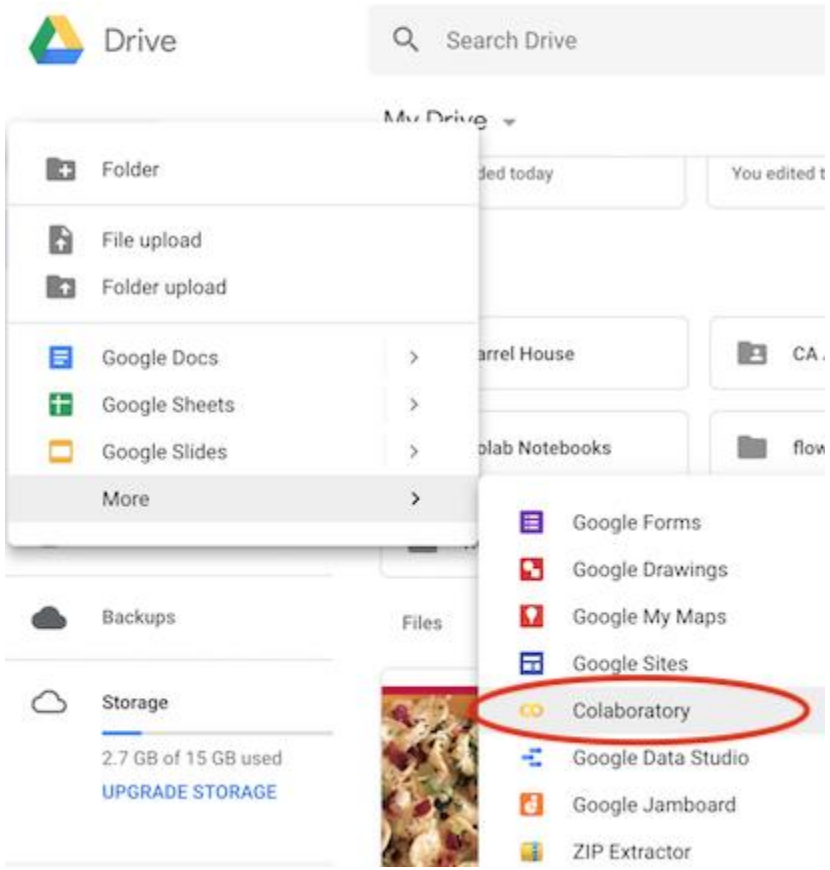
Google Colab

- **Create a folder for your notebooks**
  - You can do that by going to your Google Drive and clicking “New” and then creating a new folder.



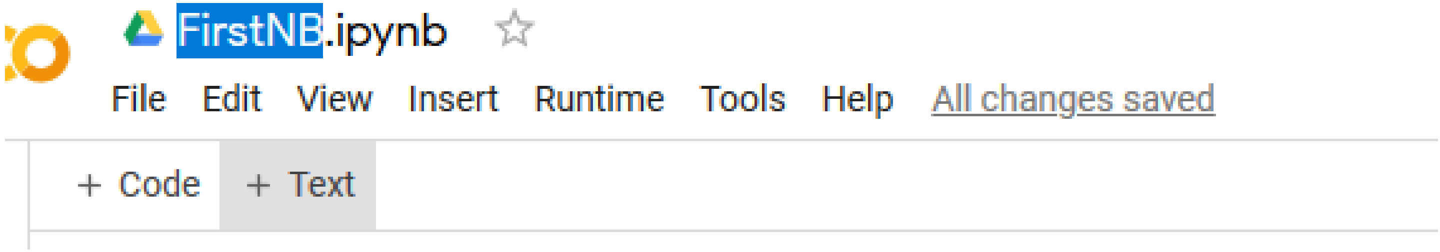
Google Colab

- **Creating a new notebook**
  - Click "New" and drop the menu down to "More" and then select "Colaboratory."



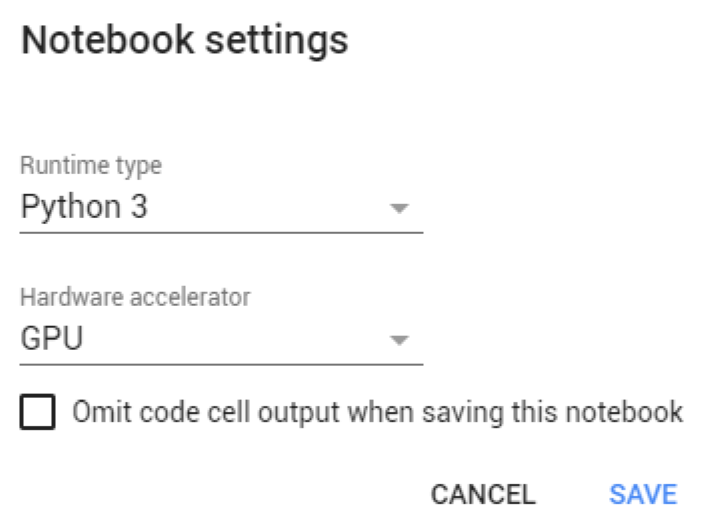
Google Colab

- **Renaming the notebook**
  - You can rename your notebook by clicking on the name of the notebook and changing it or by dropping the “File” menu down to “Rename.”



Google Colab

- **Setting up the free GPU**
  - Go to the “runtime” dropdown menu, select “change runtime type” and select GPU in the hardware accelerator drop-down menu



Google Colab: In Class Setup (load file successfully)

Google Colab

- **Mounting your Google Drive**

```
from google.colab import drive
drive.mount('/content/drive')
```

- Now you can see your drive right there on the left-hand side of the screen! (You may need to hit “refresh.”) Also, you can reach your drive any time with

```
!ls "/content/drive/My Drive/"
```

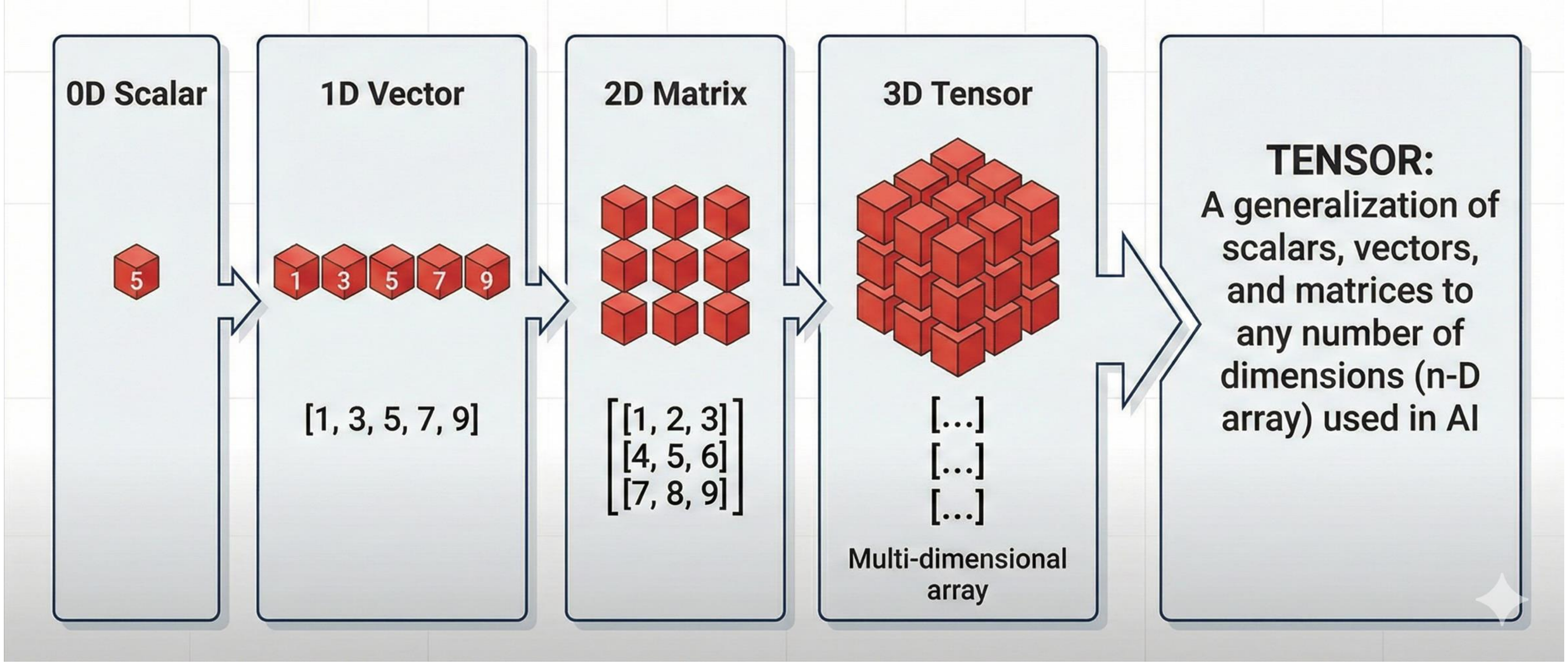
Which Python framework?

PyTorch

TensorFlow

K

PyTorch is built on tensors



Can't numpy do this instead?

PyTorch is built on tensors

Can't numpy do this instead?

PyTorch > numpy

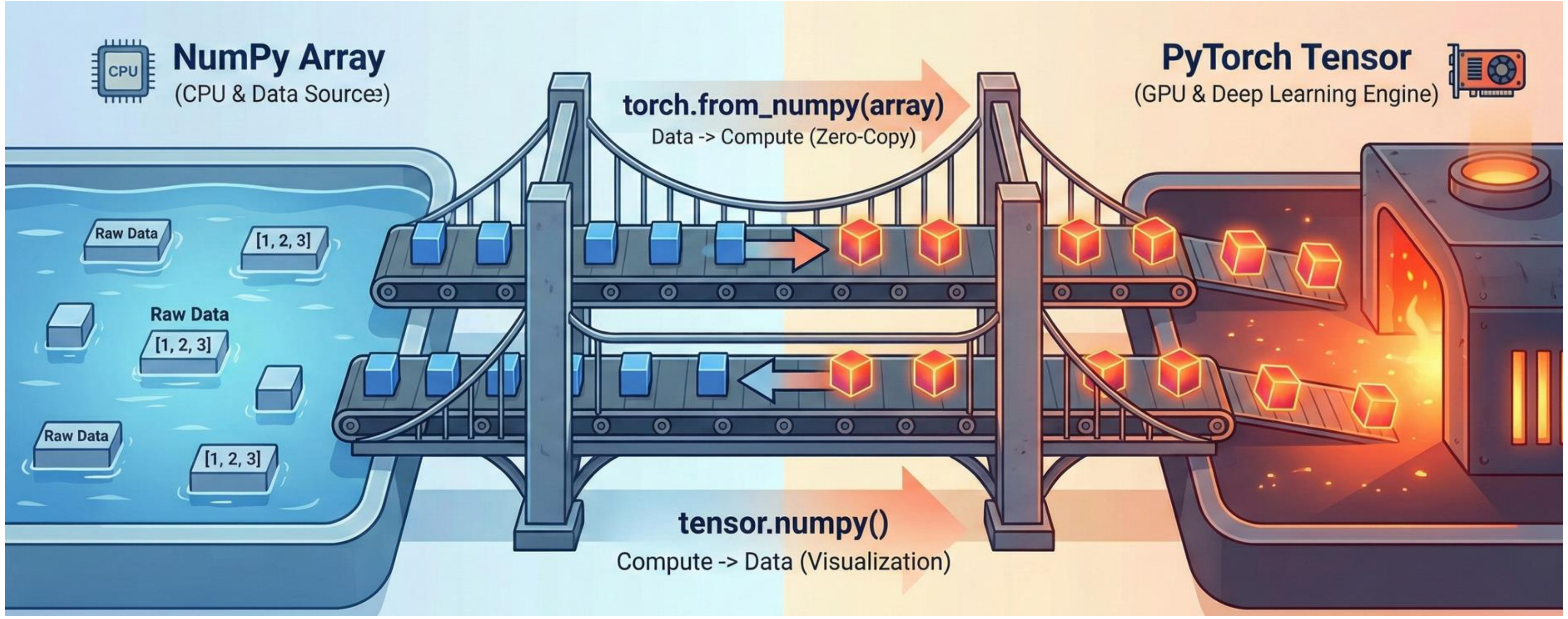
1) numpy is CPU-bound

2) PyTorch keeps track of matrix operations (A*B = C) to enable backprop

torch.autograd is a miracle …

PyTorch > numpy…kind of

numpy is still a key player in Python, so PyTorch is the “GPU bridge”



We'll use Huggingface later on

PyTorch basics

1) Everything is built from *nn.Module* (container for weights)

2) The model and the data are separate
   - *Dataset:* the object that holds your data
   - *Dataloader:* a wrapper that does the messy stuff: shuffling & organizing data, loading data into memory, etc

3) There's a standard 5-step process

PyTorch 5-step process

1) Forward Pass

    output = model(input) — Get the current prediction

2) Loss Calculation

    loss = criterion(output, target) — Measure how wrong the prediction is

3) Zero Gradients

    optimizer.zero_grad() — Clear the "memory" of the last attempt

4) Backward Pass

    loss.backward() — Calculate the "blame" for the error for every weight

5) Step

    optimizer.step() — Adjust the weights slightly to reduce the error next time

Let’s code!

”Iris dataset”: canonical multi-class classification dataset

- Ronald Fisher (1936)
- 3 species: *Iris setosa, Iris virginicolor, Iris virginica*
- 150 observations (50 per species)
- 4 numeric measurements

Sepal length, Sepal width, petal length, petal width

In-class coding lab

Try building your own neural architecture

- At least 3 layers
- Different layer sizes

Who can get the highest accuracy score?