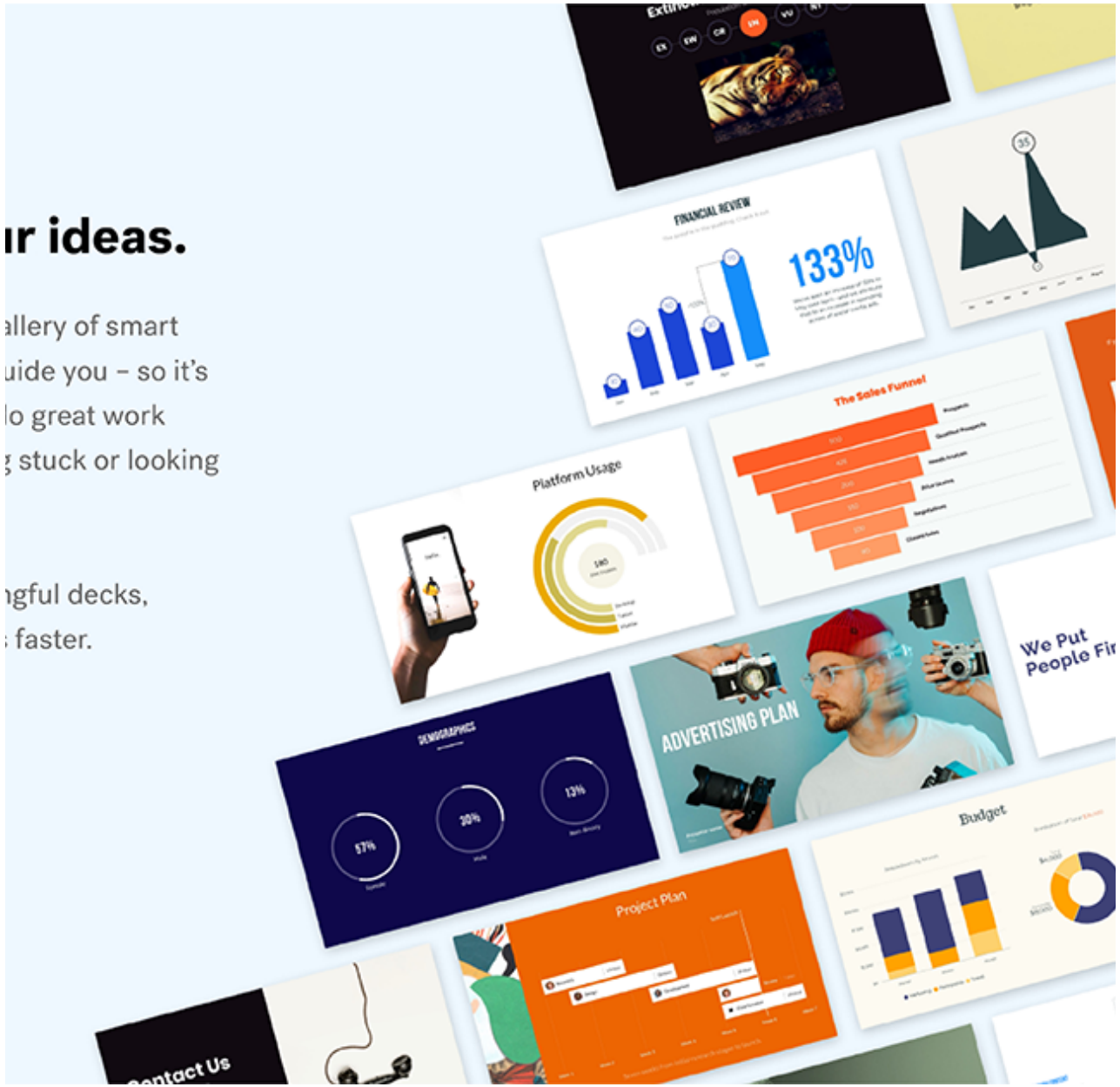
Generative AI

Mike Spertus

MPCS 27300

10/5/2023



Beautiful.ai wrote this slide

Mike and beautiful.ai worked together to create this presentation. Mike wrote most of the slides while beautiful.ai designed the visuals for all of them. This collaboration allowed them to efficiently produce a polished deck.

Beautiful.ai impressions

Compared to our using dropdeck last week, Beautiful.ai produced much better text, and it also has nice AI-driven text editing

However, I had a much harder time getting it to layout attractive slides

Indeed, this week's presentation took a lot longer to create than last week but looks less attractive

Last week was the "fun" lecture

This week is the "theory" lecture

In quotes because we won't do the math rigorously

Don't worry if some of it is unclear

Trying to help you develop an intuitive feel

Is it fun?

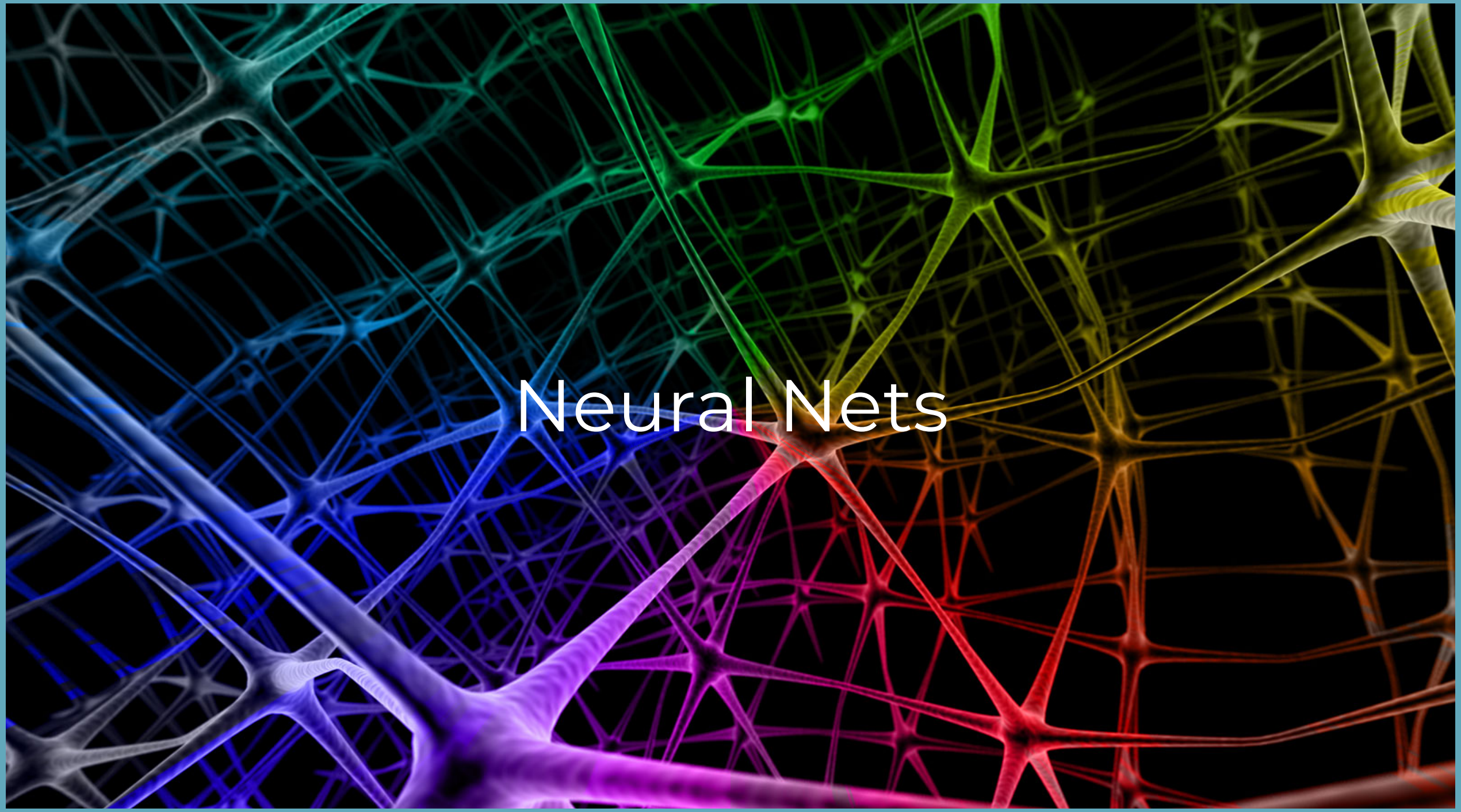
Depends whether you like theory

In one sense, you will never "need" the material from this lecture, so don't stress

In another sense, my goal is that you will be informed by it everyday

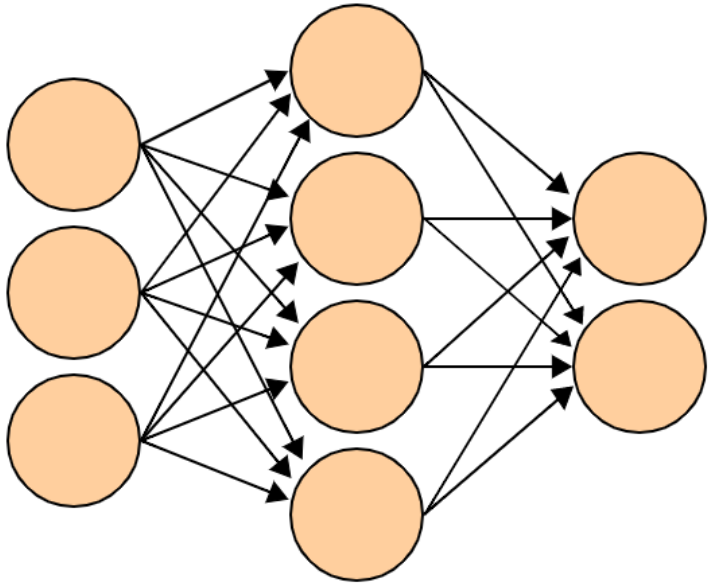
Agenda

1. Introduction to Neural Nets
2. The Anatomy of a Large-Language Model
3. Cement our understanding by building our own toy LLM


Neural Nets

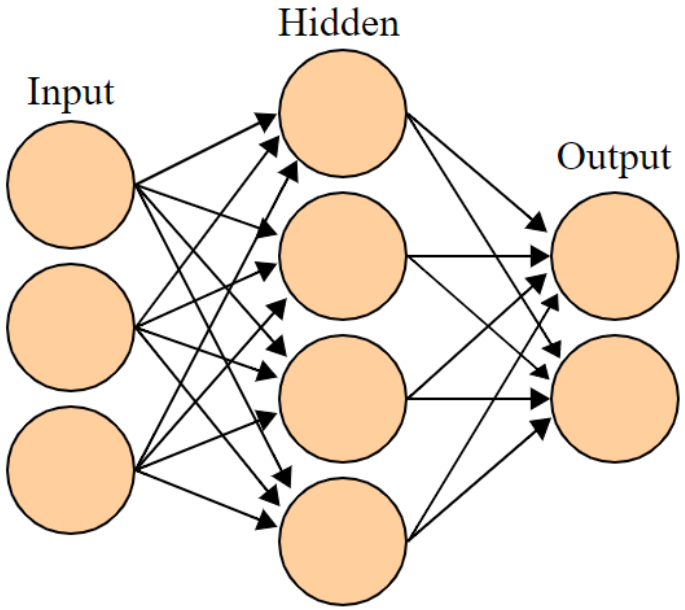
What is a neural net?

- Neural networks form the core of modern AI and deep learning
- A neural network is typically a directed graph
- The vertices are called *neurons* or *perceptrons*
  - Neither name is very accurate, so don't take them too seriously
- The edges are (conveniently) called *edges*
- The neural network shown here has 9 neurons and 20 edges



All neurons are not the same

- Neurons can be connected to the outside world
  - Neurons that receive inputs are called *input neurons*
  - Neurons that produce outputs are called *output neurons*
- Neurons that only connect to other neurons are called *hidden neurons*



All neurons are the same

- Every neuron has
  - a set of parameters w_i, one for each of its inputs
  - A fixed non-linear *activation function h*
- A neuron computes it output from inputs x_i by applying the activation function to the weighted average of the inputs
- Repeating this recursively for all the neurons is how the neural net computes its outputs from its inputs

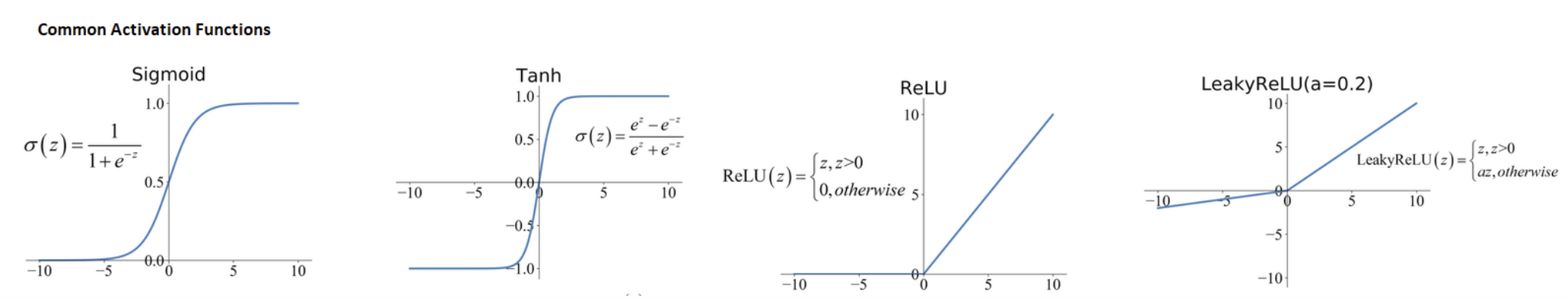
$$h(w_0 + \sum_{i=1}^{n} w_i x_i)$$

What should I use for an activation function?

There are a few nuances we will learn about in a bit, but mostly, you can just choose a non-linear, non-decreasing function and be done with it

The important point is that by adding a non-linear activation function, neural nets can solve nonlinear problems (Recall from the history, that neural nets didn't catch on until they could do so)

The figure shows some common activation functions



Training the neural net

- The shape of the directed graph of neurons and the activation function are fixed when you define the neural net, so they are not part of the training
- The one thing you can train to influence the output of the neural net is the weights

Defining a goal for your neural net

The loss function

- Before we can train our neural net, we need to decide what we are training it for
- Assume the training data consists of (x, y) pairs and Φ denotes the weights of the neural net
- We specify the goal of the neural net by defining a function called the *loss function* that measures how far the neural net's output differs from the desired output on the training data

$$\mathcal{L}(x, y, \Phi)$$

- Typical loss functions in machine learning are
  - *mean absolute error* (MAE), which is how much the output of the neural net differs from *y* when given *x*, averaged over the entire training set
  - *mean square error* (MSE), which is the average of the square of the error. Think of this as the "Pythagorean" error

The first fundamental equation of deep learning

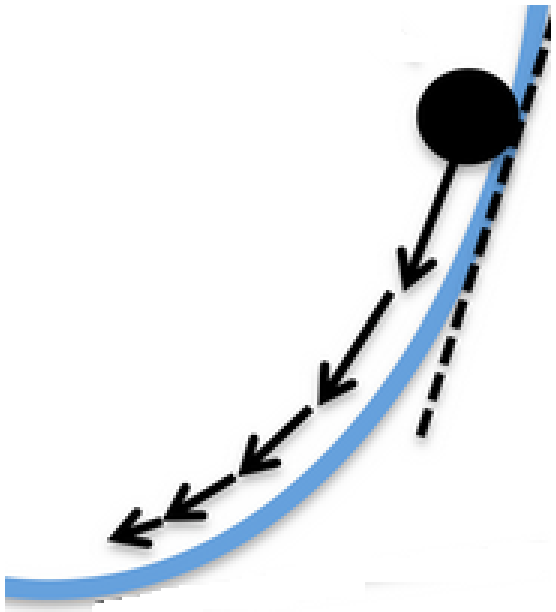
- Based on the above, our goal is to find weights Φ* that minimize the expected loss averaged over all the (x, y) training pairs, which can be written as the following formula

$$\Phi^* = \underset{\Phi}{\operatorname{argmin}} \, \mathrm{E}_{(x,y)}[\mathcal{L}(x, y, \Phi)]$$

- Dave McAllester calls this the first fundamental equation of deep learning
- Don't be intimidated by the notation
  - $\mathrm{E}_{(x, y)}$ means to take the expected value ("the average") over our (x, y) training set
  - argmin simply means choose the weight values that minimize this expected loss

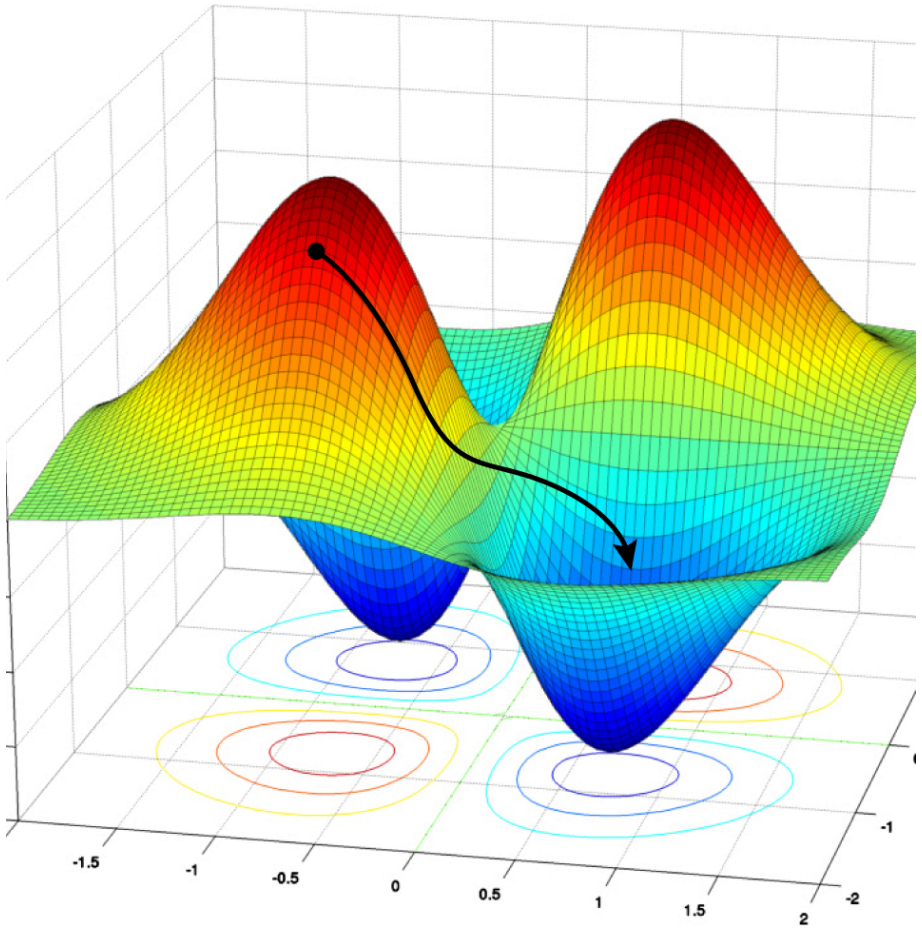
How do we minimize the loss function?

- As usual, the best way to minimize a function is through calculus
- Since machine-learning is general, we won't use any formulas from calculus but look at the slope (which we call the *derivative*) to minimize numerically
- Let's begin by looking at *Newton's Method* for minimizing a function with one input
- We start with a wild guess
- And then move in the direction that the slope is pointing downwards
- Repeat until done



Extending to multiple inputs

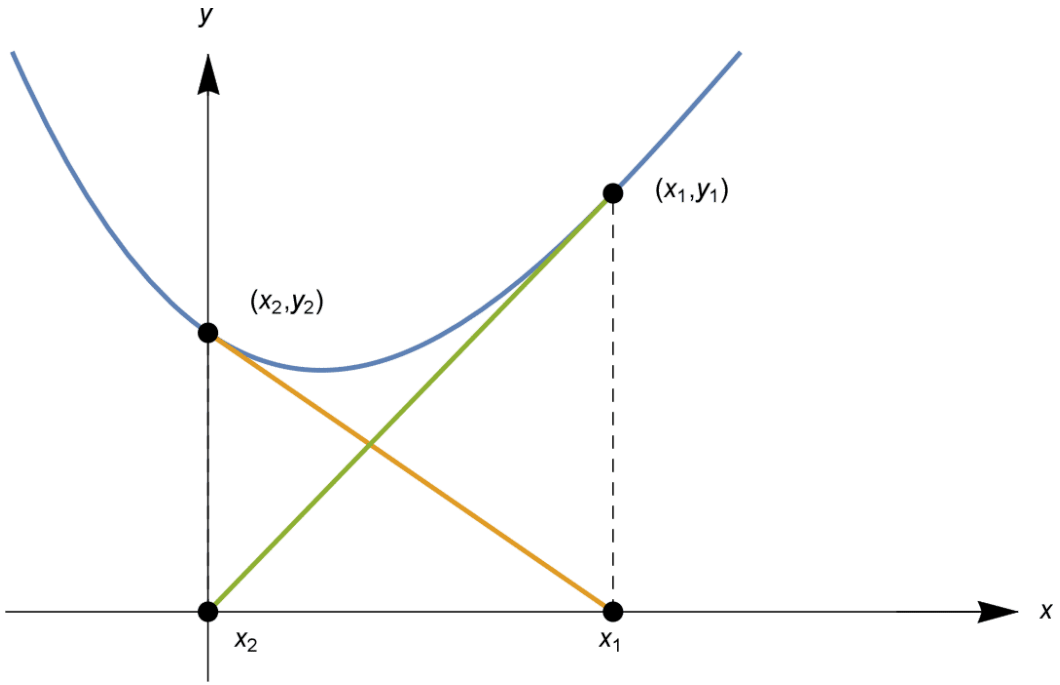
- Of course, our functions will have more than one input
  - One input for each weight
- *Gradient Descent* is a generalization of Newton's Method to multiple dimensions
- While we won't dig into the math because our deep learning frameworks do this under the hood
- Understanding the main ideas will help us use them better
- The *gradient* points in the direction that a differentiable function is increasing the fastest
  - If there is only one input, it is the same as the derivative
- So, just like with Newton's Method, move in the opposite direction
- Repeat until done



What breaks training and how to fix?

Bad learning rates

- We haven't said anything yet about how far to move in the opposite direction of the gradient
- This is called the *learning rate*
  - You will need to set it when you train a neural net
- Too low, and the neural net will take too long to converge
  - Since there can be billions of weights, the search for a minimum needs to be efficient
- Too high, and it could jump past the minimum and keep going back and forth...
- You may need to tweak until it is just right



https://www.quantamagazine.org/researchers-build-ai-that-builds-ai-20220125/

What breaks Gradient Descent and how to fix?

Large training sets

- We need lots of data to train the myriad weights of a neural net
- However, constantly recomputing the gradient of the expected loss over the entire data set will take far too long
- *Stochastic Gradient Descent* addresses this by choosing a random batch from the training data each step and moving in the direction that minimizes the average loss over that batch
- Choosing an appropriate batch size is another important tuning hook
- Too small, and batches won't reflect the full training set well enough to move towards a minimum
- Too large, and training will take too long

What breaks Gradient Descent and how to fix?

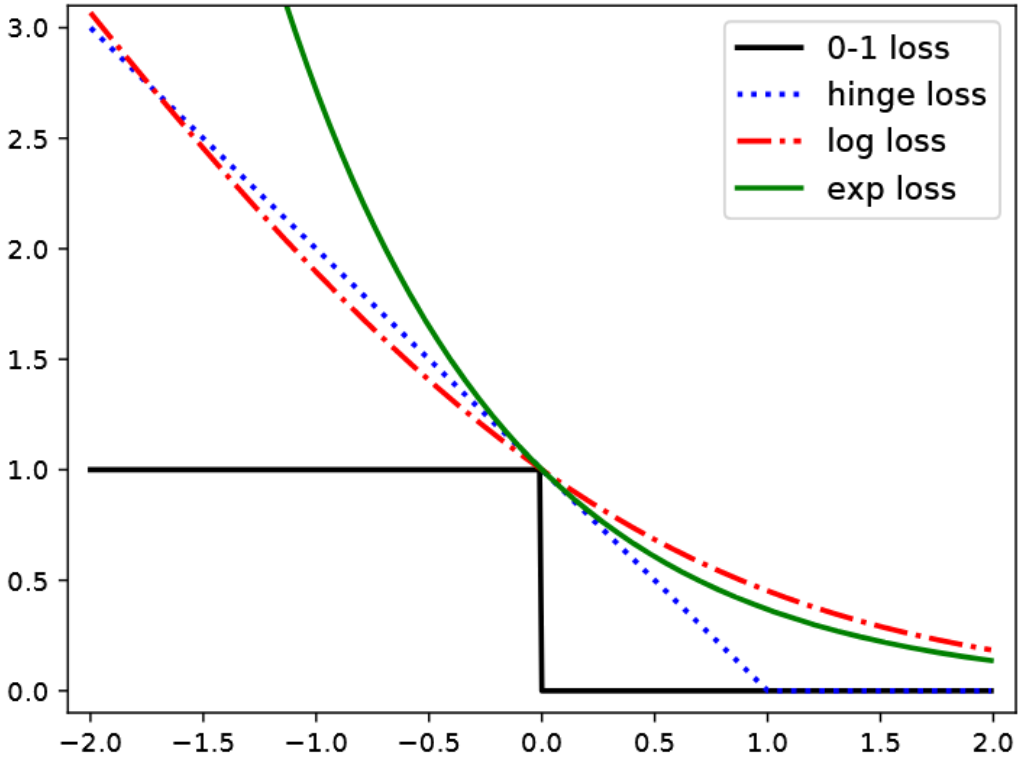
Too many layers and weights

- Even by reducing to batches
- You would guess that calculating the gradient, which involves billions of derivatives and chain rule applications for all the weights would be prohibitive
- Surprisingly, there is a simple "backpropagation" algorithm to calculate the gradients efficiently by flowing the derivatives backwards through the neural net
  - As long as it doesn't have cycles (i.e., is a "feed-forward" neural net).
- It also works beautifully on GPUs
- We won't give the formula here as it is messy and your framework and ML hardware will automatically apply it
- See Wikipedia for more information if you are interested

What breaks training and how to fix?

Bad loss functions

- Suppose we are trying to do *binary classification*
- Yes-No questions
  - Is this a picture of a dog?
  - Will this trade be profitable?
- The most natural loss function is the *0-1 loss*
  - 1 if the prediction is wrong, 0 loss if it is right
- But since the derivative is (almost always) zero, it can't be used for Newton's method/gradient descent
- The diagram to the right shows some alternate loss functions for classification problems
- We will discuss loss functions for LLMs shortly



https://probml.github.io/pml-book/book1.html, p 176

What breaks training and how to fix?

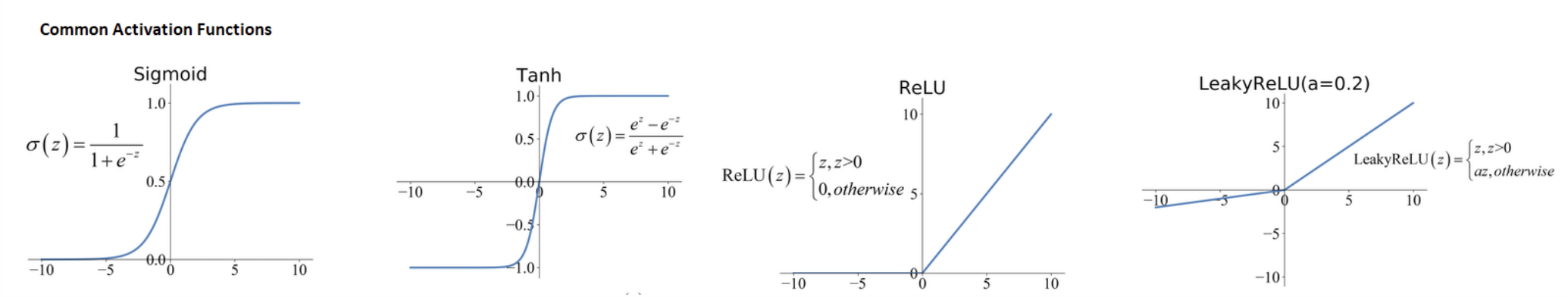
Bad activation functions

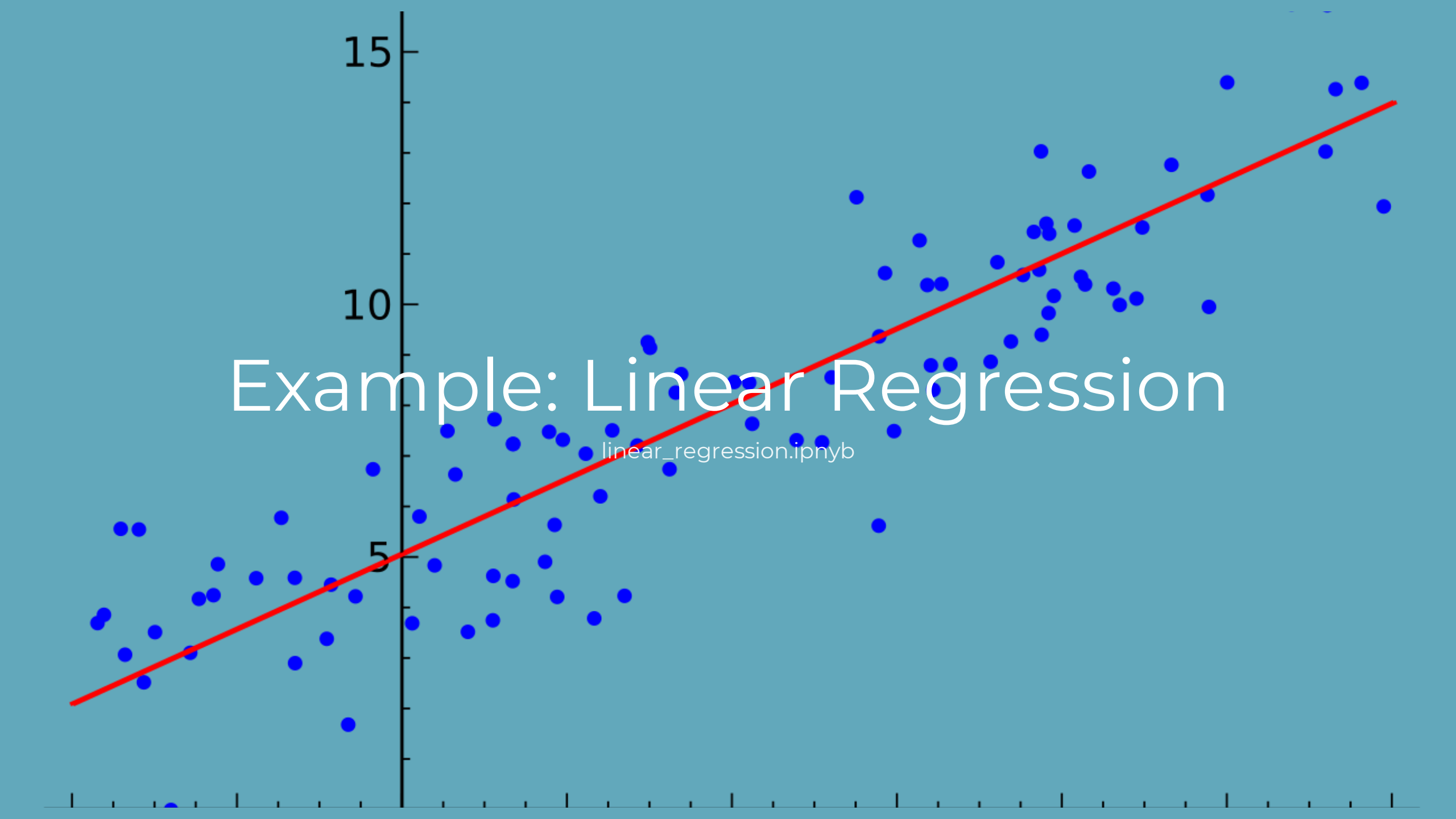
Just like loss functions need to have non-zero derivatives to power gradient descent

Activation functions do too

While sigmoids and tanh were traditionally used for neural nets, they are falling out of favor because unless their input is near zero, they are pretty much horizontal, which triggers the *vanishing gradient problem*

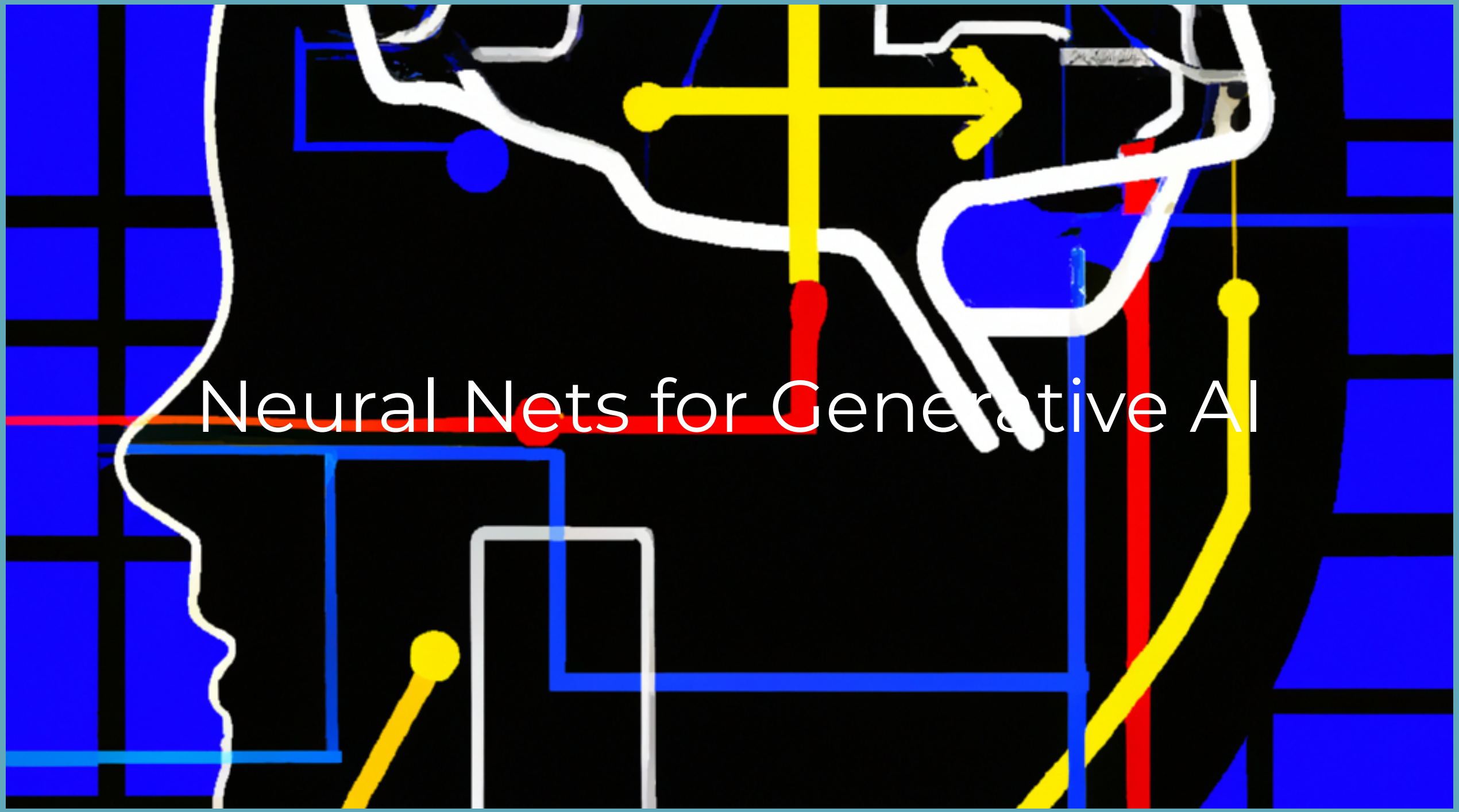
Lately *ReLU* (Rectified Linear Unit) activation functions have become the norm because they have a steep derivative no matter how big their input is, and if you need it, you can make them *leaky* to have a steep derivative everywhere

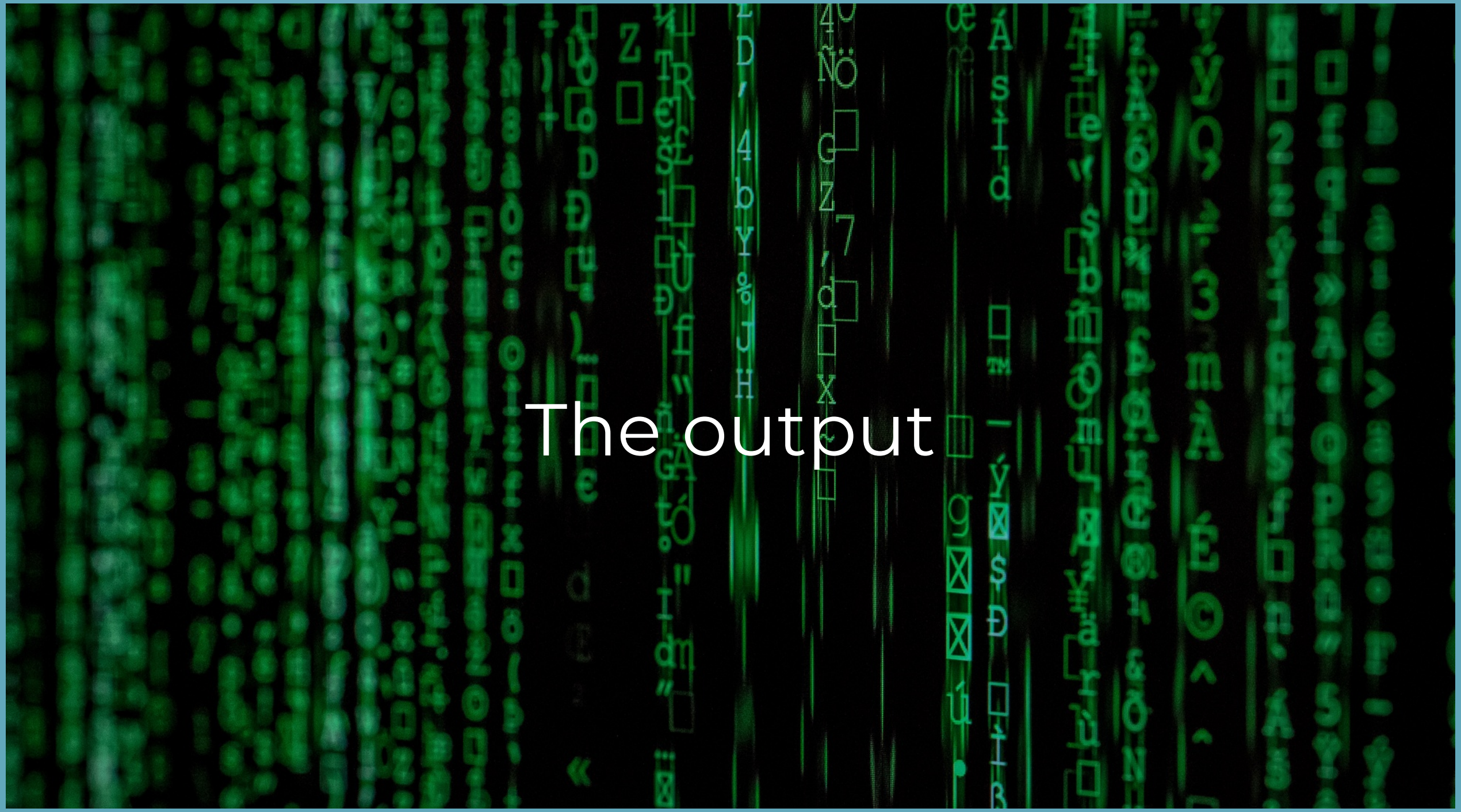




Example: Linear Regression

linear_regression.ipnyb


Neural Nets for Generative AI


The output

What should a generative neural net output?

What should a generative neural net output?

- Easy but wrong answer:
  - What it is generating

What should a generative neural net output?

- **Easy but wrong answer:**

  What it is generating

- **Better answer**

  A probability distribution

  E.g., how likely it is to generate each word

  Not only will this allow it to better model the training data

  But it will help us figure out the right loss function to optimize for generative AI

Example: Large Language Model

- Many LLMs are trained to predict the next word/token
  - We saw this with our GPT-2 notebook last week and will again when we build our own LLM this week
- This is sometimes referred to (not entirely accurately) as autoregressive
- Is this what we want? Not necessarily
  - But it is easiest to train for
  - We'll understand why when we learn about Yann Lecun's "layer cake"
- This *AI alignment problem* where the training data doesn't represent our desired behavior will raise its head regularly in this course

Example: Large Language Model (cont)

- Setting aside the question of AI alignment for the moment
- "Predict the next word" autoregressive LLMs will output a probability distribution on words

Getting a neural net to output a probability distribution

- Keep in mind the example of an LLM
- Since we need a (probability) number for each word
- We could use an output neuron for each word
- But there's a problem

Getting a neural net to output a probability distribution

The problem

- Gradient descent will tune the value of each neuron
- But nothing will force all of the output neurons' values to be nonnegative and add up to 1
- What we need is a way to interpret any set of numbers as a probability distribution
- Even if they don't add up to 1

Solution: Softmax

The *softmax* function turns any sequence of numbers into a set of positive numbers adding up to one

In other words, a probability distribution!

Consider a set of numbers that represents the output of the neuron corresponding to the word *y* when the input *x* is fed into a neural net with weights Φ

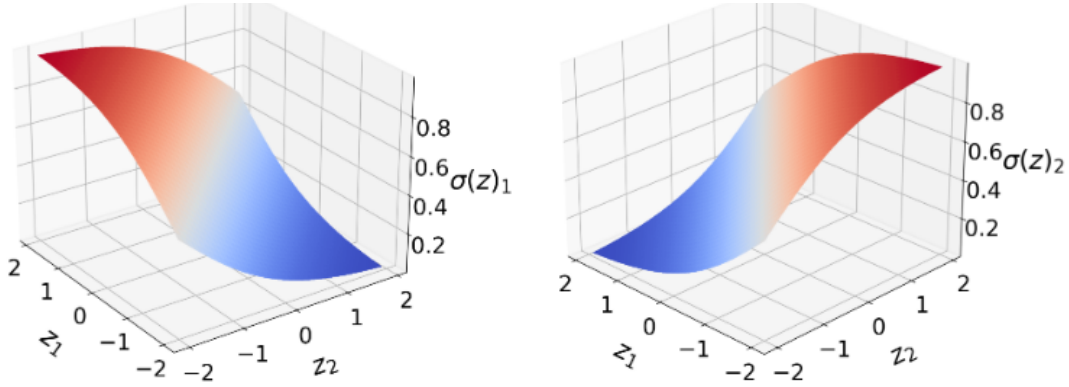
$$s_\Phi(y|x)$$

The softmax function is defined by

$$P_\Phi(y|x) = \frac{e^{s_\Phi(y|x)}}{\sum_y e^{s_\Phi(y|x)}} = \underset{y}{\text{softmax}}\, s_\Phi(y|x)$$

Why is it called softmax?

- Our main interest is that it can turn any sequence of numbers into a probability distribution
- But the name is intriguing
- If one number is substantially bigger than the other
  - It chooses that one, so the **max** choice gets almost all the probability
  - Exponentials grow so fast, the others are dwarfed
- However, if they are close
  - It balances the probability
  - **Soft**ening the move to the maximum
  - And giving us the good differentiability properties we need for training



https://arxiv.org/pdf/2208.08772.pdf

Wow, that was confusing!

Remind me what the point is

- Neural nets can output a sequence of numbers
- But Generative AI needs to produce a probability for each generated output
- Softmax can turn any sequence of numbers into probabilities
  - Non-negative numbers that add up to one
- So Generative AI systems routinely add a softmax layer as their final output activation
- This is so central that Dave McAllester calls the definition of softmax the Second Fundamental Equation of Deep Learning


The loss function

What is the loss function for Generative AI?

- **Easy but wrong answer:**

  zero loss if it generates the target

  one loss if it gets it wrong

  This fails for all the same reasons we described for the 0-1 classification loss

  In addition, it ignores that the neural net generates a probability distribution

- **Better answer**

  How similar the probability distribution it generates matches the empirical distribution of the training data

Cross-Entropy

- Information theory gives us a way to measure the similarity between two probability distribution called *cross entropy*
- While a rigorous explanation would take a lot of time
- There is a fairly simple intuition

Cross-Entropy: Intuition

Last week's homework suggested that one could

design a zip compressor for English that encodes common english words efficiently

This is a popular zipping implementation

and then identify non-English text because the compressor doesn't compress it well

The more dissimilar the probability distributions of the two languages, the worse the English compressor will perform on the non-English language

And that's the idea of cross-entropy!

The *cross-entropy* from probability distribution P to distribution Q measures the average number of bits needed to encode events generated from distribution P with a compressor designed to compress events generated by Q

Cross-Entropy: Formal Definition

The definition is simple enough that we give it here for reference

$$H(P, Q) = -\sum P(x) * \log Q(x)$$

where *x* ranges over all possible events

If you want to know more, David MacKay's book *Information Theory, Inference, and Learning Algorithms* is absolutely beautiful and available for free online here

Cross-Entropy and Machine Learning

- Since cross-entropy measures how similar the generated probability distribution is to the probability distribution of the training data
- We can use it as our loss function for how far we are from the ideal!
- It is fine-grained enough that it gives the gradually varying loss measurement needed by gradient descent
- And it is built into all machine learning frameworks


Generating Data

Once we've trained our LLM, How do we use it?

- When you put text into the LLM
- You get a probability distribution out
- But what word should you output?

The most likely word?

- One obvious solution is to simply output the word that the trained LLM says is most likely to come next
- And there's nothing wrong with that!

Adding some randomness

- Another option is to introduce some randomness into the output
- The simplest way to do this is to just choose a word according to the probability distribution that was output
- And there's nothing wrong with that!

Turning up the temperature

- You can also tune the amount of randomness through setting the *temperature* of the LLM
- More formally, divide the output neuron values by the temperature T before applying softmax
- In the below, I've omitted Φ because we're done training, and it's already baked in

$$P_T(y|x) = \frac{e^{\frac{s(y|x)}{T}}}{\sum_y e^{\frac{s(y|x)}{T}}} = \underset{y}{\text{softmax}} \frac{s(y|x)}{T}$$

Properties of the temperature

- We can reproduce the two above strategies
  - A temperature of 0 always chooses the most likely output (To avoid dividing by 0, choose a very small T to effectively take the limit)
  - A temperature of 1 chooses according to the original output distribution
- Setting the temperature higher makes the output more random
- Lower makes it less random

OK, but what does that really mean?

Let's try it out!


Overview

Let's review what we did

Neural nets

- Neural nets are a graph of neuron
- A neuron has an activation function and a weight for each incoming edge (and sometimes one more for a constant offset)
- The neuron computes its output by multiplying all the inputs by their corresponding weight, adding up, and applying the activation function

Let's review what we did

Training neural nets

- Neural nets can be trained with stochastic gradient descent that chooses weights that reduce our loss function
- There are a lot of details, but philosophically it is just like using a derivative to numerically minimize a function in first year calculus

Let's review what we did

Generative neural nets

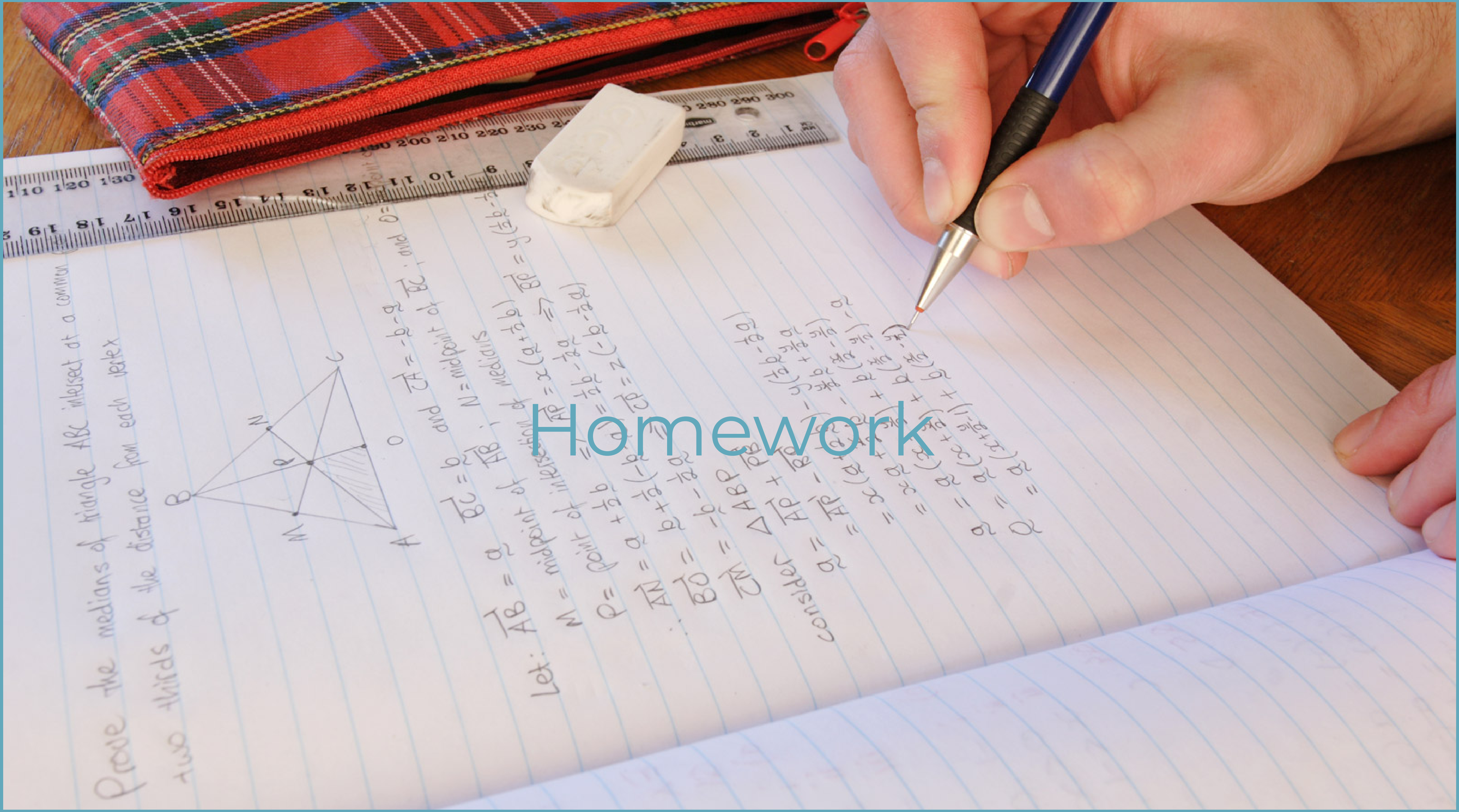
- Generative neural nets output probability distributions
  - A likelihood for each output
- We can make a neural net output a probability distribution by ending with a softmax layer
- And train the net by minimizing the cross-entropy

Let's review what we did

Running a generative neural net

- We can tune the randomness in a neural net by dividing by the temperature before softmax
- A temperature of 0 means always choose the most likely
- A temperature of 1 means use the learned distribution
- The higher the temperature the more the randomness

Homework



HW 2-1: Temperature

- Describe examples of when/why you would want
  - a temperature of 0
  - a temperature of 1
  - another temperature

HW 2-1 (extra credit part)

- Prove the following statements
  - A temperature near 0 chooses the most common
  - A temperature of 1 chooses the original distribution
  - A temperature near infinity generate outputs uniformly at random
- Modify our toy llm to use temperature
  - Hint: A keras lambda layer may come in handy here

HW 2-2

- Softmax seems complicated
- Scaling a set of numbers to add up to one seems by dividing by their sum seems like an easier way to force the total probability to be one (as required by a probability distribution)
- Could we have done that instead? Why or why not?