Intro

In Classical ML, the paradigm was

- construct, by hand, transformations of the input to alternatate representations
    - feature engineering: create representations corresponding to a "concept" useful for classification
    - we called this a pipeline
- after multiple transformations, the representation was good enough that a classifer could separate classes

In Deep Learning, the paradigm is very similar

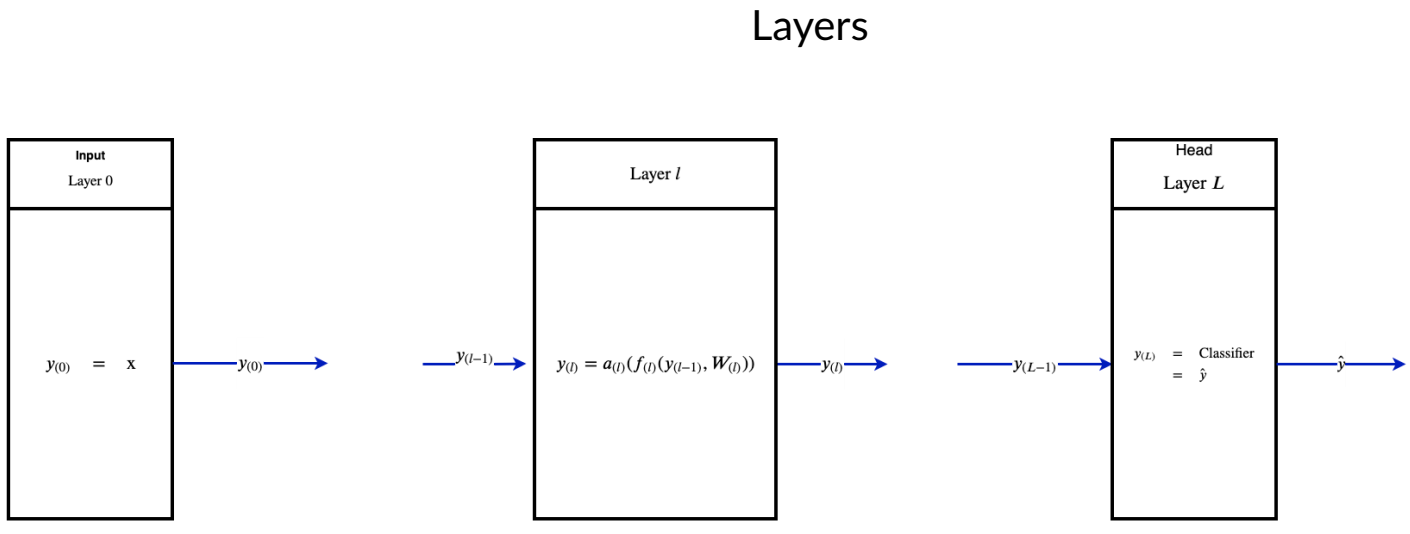
- a sequence of transformations
    - each transformation is called a "layer"
        - "Deep Learning" is many layers of transformation
    - each layer successively constructs a new representation

The key difference from Classical ML:

- the transformations are "discovered" rather than hand engineered

Deep Learning: the cartoon

Here is a "cartoon" diagram of Deep Learning (as applied to Classification)



- It is a sequence of *layers* (vertical boxes)
- It starts with the input layer (the large vertical box on the left)
- Layer l takes as input the output of layer $(l-1)$

- The output of the penultimate layer $(L-1)$
- Is used as input to a final layer L that implements
    - Classifier for Classification
    - Regression

The process of moving through the layers from Input to penultimate is

- Successive transformation of the input
- Each layer's output is an alternate *representation* of the input

- This is just a Transformation Pipeline (like we've seen in `sklearn`)
- Where the final version of the transformed data is fed into a Classifier/Regression model

As we will come to see:

- The **key difference** from Classical Machine Learning
- Is that the **transformations are non-linear**

Our old friend, the dot product, will play a starring role.

- it is a fundamental part of many layers that we will study

Let's start with our the inputs to the NN: an example with our original features

Recall that the dot product can be thought of implementing a kind of "pattern matching:

- looking for a subset of features in an example

So one way to think of the transformations implemented by the **first layer** is that it

- Recognizes patterns in the n original features
- Constructs $n_{(1)}$ new synthetic features

Layer l

- Recognizes patterns in the $n_{(l-1)}$ synthetic features created by layer $(l-1)$
- Constructs $n_{(l)}$ new synthetic features

Ultimately, the $n_{(L-1)}$ synthetic features created by penultimate layer $(L-1)$

- Is a representation of the input $\mathbf{x}$
- That has been sufficiently transfored
- So that a Classifier/Regression model at layer L can be successful

Neural networks: introduction and notation

The above was very informal.

We need to introduce more concepts and, unfortunately, the notation that will carry us through the Deep Learning part of the course.

Time to go under the covers of a layer (Intro_to_Neural_Networks.ipynb)

What is $W_{(l)}$? Where did Θ go ?

Our old friend the dot product is back in the forefront.

But now, the pattern matching is written

$$\mathbf{W}_{(l),j} \cdot \mathbf{y}_{(l-1)}$$

rather than the

$$\Theta \cdot \mathbf{x}$$

which was familiar in the Classical Machine Learning part of the course.

Unfortunately, this is an artifact of two different communities working independently

- The Classical Machine learning community uses the term *parameters* and the Greek letter Θ
- The Computer Science community uses the term *weights* and the letter W

They are *exactly the same thing*.

Moreover, Neural Networks operate in layers

- So only the dot product of the *first* layer involves $\mathbf{x}$, which we have equated with $\mathbf{y}_{(0)}$
- The dot product of layer l is finding patterns in $\mathbf{y}_{(l-1)}$ rather than $\mathbf{x}$.

Size of $\mathbf{W}$: Always count the number of parameters !

When constructing a layer of a Neural Network: **always count the number of weights/parameters. They grow quickly !!**

$\mathbf{W}_{(l)}$

- Consists of $n_{(l)} = ||\mathbf{y}_{(l)}||$ units, each unit producing a new feature
- Each unit performs the dot product
$$\mathbf{y}_{(l),j} = \mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),j}$$
- Each dot product thus involes $||\mathbf{y}_{(l-1)}|| + 1$ weights in $\mathbf{W}_{(l),j}$
    - the "+ 1" is because of the bias term in each unit
- Thus the number of weights in $\mathbf{W}_{(l)}$ is $||\mathbf{y}_{(l)}|| * (||\mathbf{y}_{(l-1)}|| + 1)$

What does a layer do ?

Surprisingly: this is not an easy question to answer !

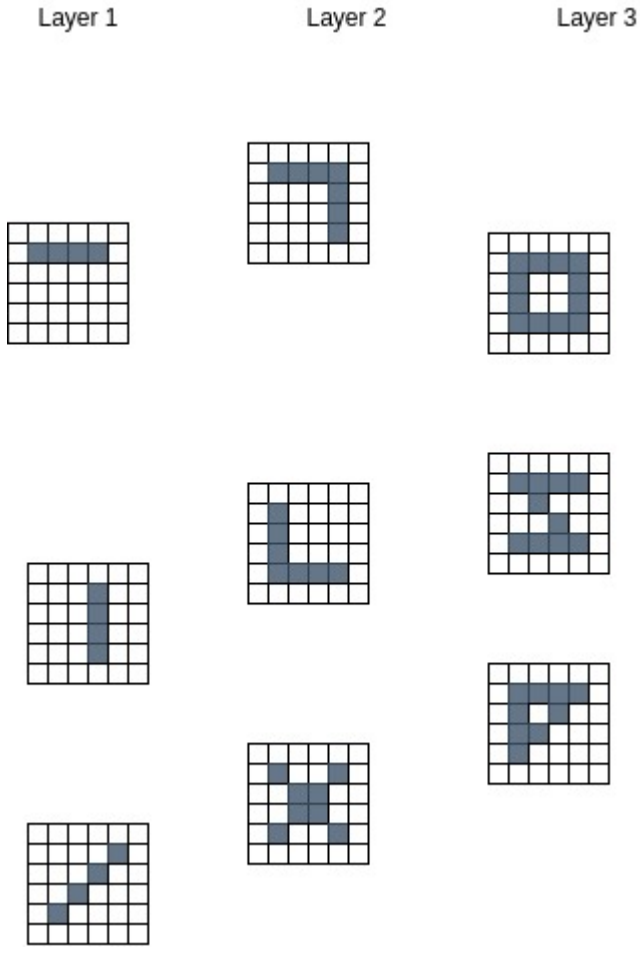
The behavior of Neural Networks is a bit mysterious.

We will spend part of a future lecture on trying to interpret what is happening inside the network.

For now, one can imagine that

- The first layer recognizes features (matches patterns) for *primitive* concepts
- The second layer recognizes features that are *combinations* of primitive concepts (layer 1 concepts)
- The l recognizes features that are *combinations* of layer $(l-1)$ concepts

Features by layer



Activation functions

At this point perhaps you have a mechanical understanding of a neural network

- A sequence of layers
- Each layer is creating new features
- Subsequent layers creating features of increasing complexity but transforming the prior layer's features
- A new feature is created by a liner dot product followed by an non-linear activaton

It turns out that the non-linear activation function is *one of the keys* to Neural Networks !

Let's explore Activation functions (Neural_Networks_Activations.ipynb) in more depth.

Final layer: Regression/Classification

Layer L in our cartoon implements Regression or Classification.

- Regression is nothing more than a dot product followed by an identity/linear activation
$$\mathbf{y}_L = \mathbf{y}_{(L-1)} \cdot \mathbf{W}_L$$
- Classification is nothing more than a dot product followed by a sigmoid activation
$$\mathbf{y}_L = \sigma(\mathbf{y}_{(L-1)} \cdot \mathbf{W}_L)$$
as discussed in our lecture on Classification.

Questions to consider

Some natural questions to ask at this point

- How many layers should we have (What is the right value for L) ?
- How many units $n_{(l)}$ should I have for each layer $1 \leq l \leq (L-1)$?
- What activation function should I use for each unit?

We will address each of these in the future.

Perhaps the biggest question

- $\mathbf{W}_{(l),j}$ is the pattern used to recognize the feature created by unit j of layer l
- How does $\mathbf{W}_{(l),j}$ get set ?

This will be the topic of the next section.

Training a Neural Network

We will start to answer the question of how $\mathbf{W}$ is determined.

We will briefly introduce training a Neural Network (Neural_Networks_Intro_to_Training.ipynb), a subject we will revisit in-depth in a later lecture.

Tensorflow: A toolkit for Neural Networks

Why do we need a dedicated toolkit (Tensorflow) to aid the programming of Neural Networks ?

It's mainly about the use of Gradient Descent in training the network.

Recall that a Neural Net (including one augmented by a Loss Layer) is doing nothing more than computing a function.

Gradient Descent needs to take the gradient of this function (evaluated on a minibatch of examples) in order to update the weights $\mathbf{W}$.

There are at least two ways to obtain the Gradient

- Numerically
- Analytically

Numerical differentiation applies the mathematical definition of the gradient

$$\frac{\partial f(x)}{\partial x} = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

- It evalues the function twice: at $f(x)$ and $f(x+\epsilon)$

- This can get expensive, especially since
    - $\mathbf{x}$ is a vector
    - Potentially very long (e.g., many features)
    - We need to evaluate the derivative of $f(\mathbf{x})$ with respect to each $\mathbf{x}_j$

Analytic derivates are how you learned differentiation in school

- As a collection of rules, e.g.,
$$\frac{\partial(a+b)}{\partial x} = \frac{\partial a}{\partial x} + \frac{\partial b}{\partial x}$$

This is very efficient.

The issue in ordinary code

- The expression `(a + b) * c`
- Is evaluated `tmp = a + b`
- And the result passed to the next step of the computation, e.g, `tmp * c`
- Losing the connection between `tmp` (a value) and the operation (plus) and addends (`a`, `b`)

There is no information recorded in ordinary code that would allow the application of analytic rules of differentiation.

Tensorflow is different in that `(a + b) * c`

- Is a symbolic expression (i.e., recorded as operation and arguments)
- That is saved
- Facilitating the application of analytic rules of differentiation

We still write `(a + b) * c` but it really results in something like:

- `tf.math.mult( tf.math.add(a, b), c)`
- The expression `tf.math.add(a, b)`
    - Can be differentiated analytically
    - Since we know the operation is addition
    - It records that the arguments are `a, b`

So Tensorflow facilitates analytic function differentiation while hiding the details from the user.

By the way: what is a Tensor ? It is an object with an arbitrary number of dimensions.

We use special cases all the time:

- A scalar is a tensor of dimension 0
- A vector is a tensor of dimension 1
- A matrix is a tensor of dimension 2

As you've seem, we are already dealing with higher dimensional objects.

Consider $\mathbf{y}$:

- $\mathbf{y}_{(l),j,j'}$
  - Output of layer l : $\mathbf{y}_{(l)}$
  - Unit j of layer l : $\mathbf{y}_{(l),j}$
  - Element j' of the output of unit j of layer l : $\mathbf{y}_{(l),j,j'}$

In the future we will talk about *sequences* of $\mathbf{y}$, thus adding another dimension: time.

The notation will become a little heavy but hopefully understandable as a way of indexing a high dimension object.

Sequential versus Functional construction of a Neural Network

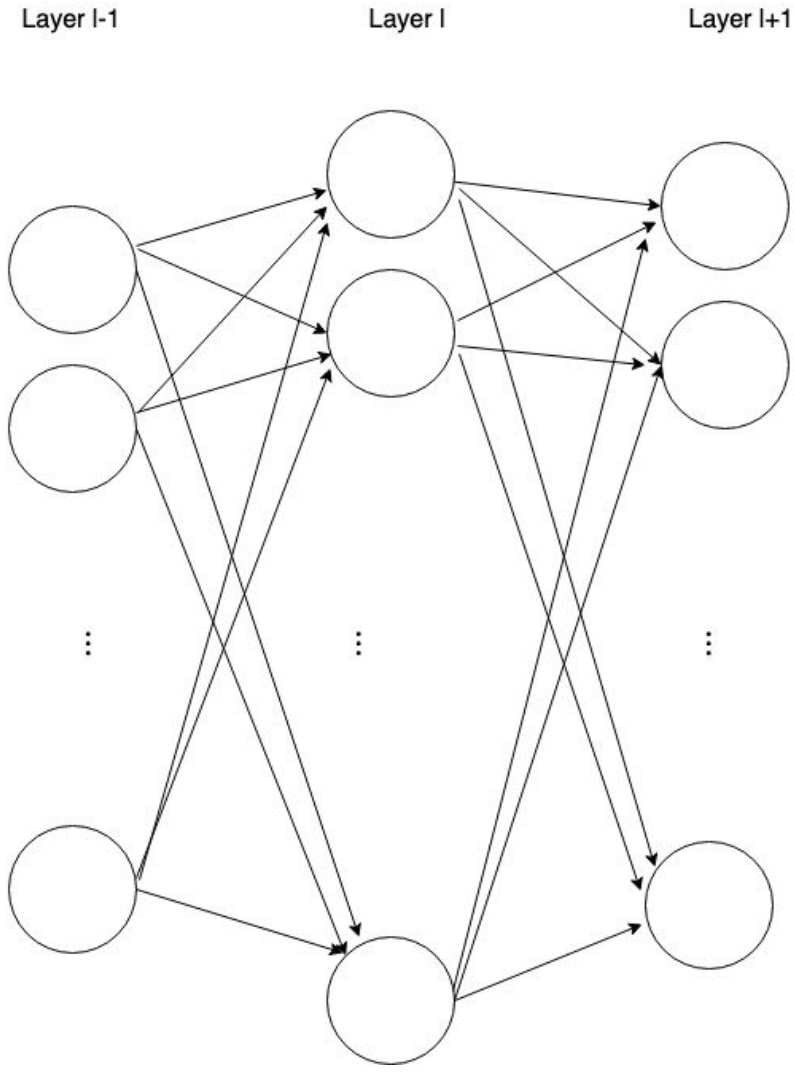
The above description of a Neural Network imposed several restrictions on units (neurons)

- Each layer is homogeneous
- Layers are organized sequentially

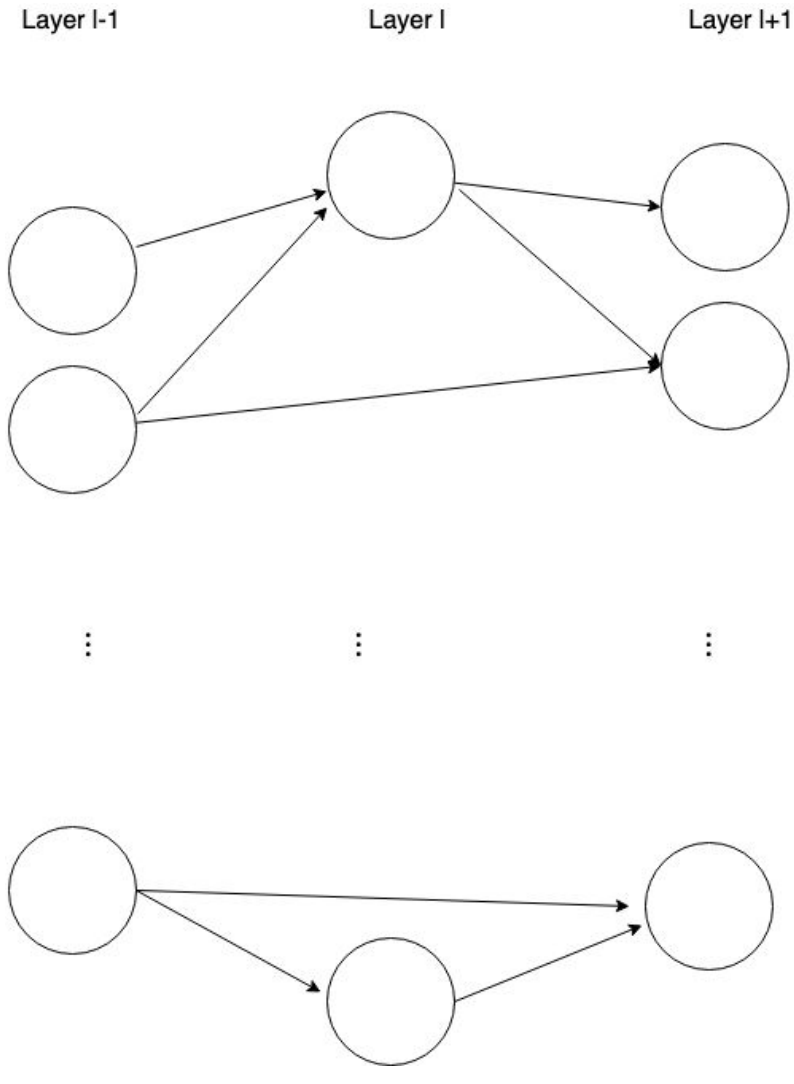
One can imagine a network graph without these restrictions

- Not organized as layers
- Pairs of units with arbitrary connection

Sequential architecture



Functional architecture



The layer-oriented network we described is called *Sequential* in Keras.

The unrestricted network form is called *Functional* in Keras.

For the most part, we will use Sequential networks

- Easier to build
- But less flexible

Some "Why's ?"

What took so long: Preview

An historical perspective:

- Perceptron invented 1957
- mid-1970's: First "AI Winter"
- Late 1980's: secibd "AI Winter"
- 2010: Re-emergence of AI

The promise of AI led to great expectations, that were ultimately unfulfilled. The difficulty was the inability to train networks.

We will defer a fuller answer to a later lecture.

For now: seemingly minor choices were more impactful than imagined

- Sigmoid as activation function turned out to be a problematic choice
- Initializing $\mathbf{W}$ properly was more important than imagined

- Vanishing/Exploding Gradients
    - problems arise when the gradient is effectively 0
    - problems also occurs when they are effectively infinite

- Computational limits
    - It turns out to be quite important to make your NN big; bigger/faster machines help
    - Actually: bigger than it needs to be
        - many weights wind up near 0, which renders the neurons useless
        - The Lottery Ticket Hypothesis (https://arxiv.org/abs/1803.03635)
            - within a large network is a smaller, easily trained network
            - increasing network side increases the chance of large network containing a trainable subset
            - summary (https://towardsdatascience.com/how-the-lottery-ticket-hypothesis-is-challenging-everything-we-knew-about-training-neural-networks-e56da4b0da27)

Why do GPU's matter ?

What makes TPU's fined tuned for Deep Learning (https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning)

GPU (Graphics Processing Unit): specially designed hardware to perform repeated vector multiplications (a typical calculation in graphics processing).

- It is not general purpose (like a CPU) but does what it does extremely quickly, and using many more cores than a CPU (typically several thousand).
- As matrix multiplication is a fundamental operation of Deep Learning, GPU's have the ability to greatly speed up training (and inference).

Google has a further enhancement called a TPU (Tensor Processing Unit) to speed both training and inference.

- highly specialized to eliminate bottlenecks (e.g., memory access) in fundamental Deep Learning matrix multiplication.

Both GPU's and TPU's

- Incur an overhead (a "set up" step is needed before calculation).
- So speedup only for sufficiently large matrices, or long "calculation pipelines" (multiplying different examples by the same weights).

DL involves

- Multiplying large matrices (each example)
- By large matrices (weights, which are same for each example in batch)

- Both GPU's and TPU's offer the possibility of large speed ups.

- GPU's are **not** necessary
    - but they are a **lot** faster
    - life changing experience
        - 30x faster means your 10 minute run (that ended in a bug) now only takes 20 seconds
            - increases your ambition by faster iteration of experimental cycle

In [4]:
```
print("Done")
```

```
Done
```