Recap of "Classical" Machine Learning

Canonical Machine Learning tasks

- Classification
    - Given attributes, predict a target value from among a *discrete* set of values
- Regression
    - Given attributes, predict a target value from a *continuous* range of values

To solve the task

- We create a *parameterized* model: mapping from input features to label/target
- Fit the model parameters on a collection of examples (feature/target pairs)

$$\langle \mathbf{X}, \mathbf{y} \rangle = [\mathbf{x^{(i)}}, \mathbf{y^{(i)}} | 1 \leq i \leq m]$$

- Use the fitted model to predict a $\hat{\mathbf{y}}$ from an unseen feature vector $\mathbf{x}$

It is often the case that we can make our model more successful (e.g. more accurate) by

- rather than using raw inputs $\mathbf{x}$
- we transform the raw inputs
- fit our model and make predictions on the transformed feature vectors

To illustrate, consider some features describing a person

- Height
- Weight
- Income

Our goal is to predict the target: some measure of Happiness.

Extremely small differences in Height or Weight probably don't affect Happiness.

We may choose to transform each variable from a continuous value to a discrete value

- Height transformed to Height Bucket: { Short, Medium, Tall }
- Weight transformed to Weight Bucket: { Thin, Just Right, A Little Extra }

It is sometimes the case that each variable individually is a poor predictor, but the combination is powerful.

In such a case, a transformation that creates a new "combination" feature may be helpful

- Create a discrete binary variable Combo indicating "Perfect Height and Perfect Weight"

The absolute level of Income may not be as good a predictor of Happiness as the level relative to one's peers

- We may choose to transform Income into Relative Income: Income/(Median Income)

By fitting/training on

- (Height Bucket, Weight Bucket, Combo, Relative Income)
- rather than (Height, Weight,Income) our model might make better predictions.

This process of transforming/augmenting input features is called *feature engineering*

- In Classical Machine Learning, it is the responsibility of the person fitting the model

In "Classical" Machine Learning (ML) the paradigm is to

- *Manually* create a sequence of transformations from raw input to an alternate representation
    - Feature engineering: create representations corresponding to "concepts" expressed by the data
    - A sequence of transformations a *pipeline*
- The final representation created by the sequence may result in a better prediction than that which could be obtained from the raw representation

Deep Learning: Introduction

In Deep Learning, the paradigm is very similar to that of Classical Machine Learning

- a sequence of transformations
- resulting in a final representation better able to predict

One difference from Classical ML is that a transformation is implemented by a *Neural Network*

- The transformations are often organized into a sequential pipeline
- The Neural Network implementing a transformation is called a *layer*
- Deep Learning refers to a sequence of many layers

Similar to Classical ML

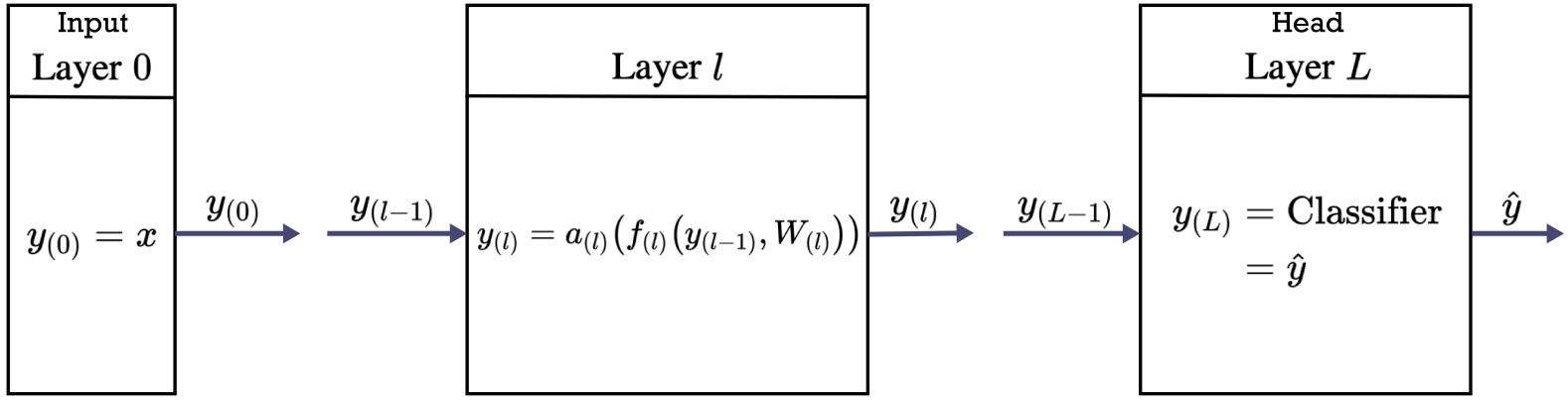
- the transformations of Deep Learning create alternate representations of the input that result in better prediction.

The key difference from Classical ML:

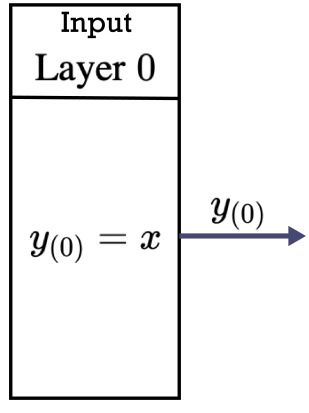
- the transformations of Deep Learning are "discovered" rather than hand engineered

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

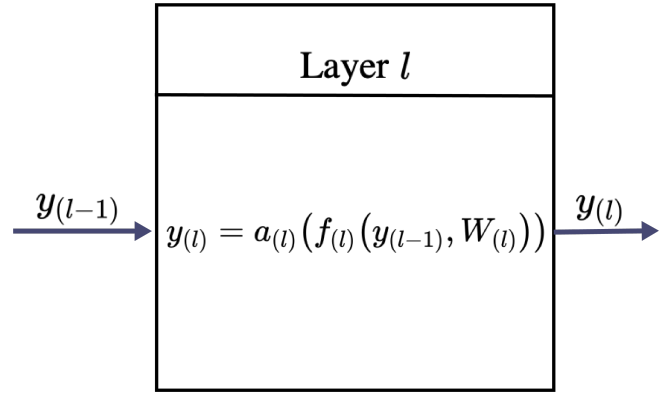
Layers


Input
Layer 0
$y_{(0)} = x$
$y_{(0)}$
$y_{(l-1)}$
Layer l
$y_{(l)} = a_{(l)}(f_{(l)}(y_{(l-1)}, W_{(l)}))$
$y_{(l)}$
$y_{(L-1)}$
Head
Layer L
$y_{(L)} = \text{Classifier}$
$= \hat{y}$
$\hat{y}$

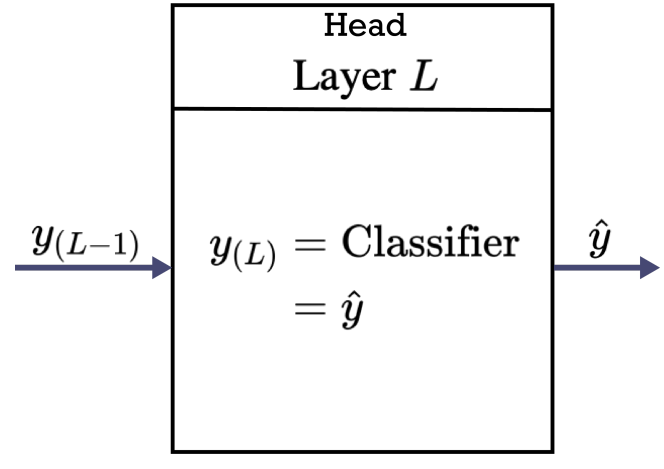
- A sequence of *layers* (vertical boxes)
- Starting with the input layer (the large vertical box on the left)



- Intermediate layer l takes as input the output of layer $(l-1)$
- Transforming it into an alternate representation

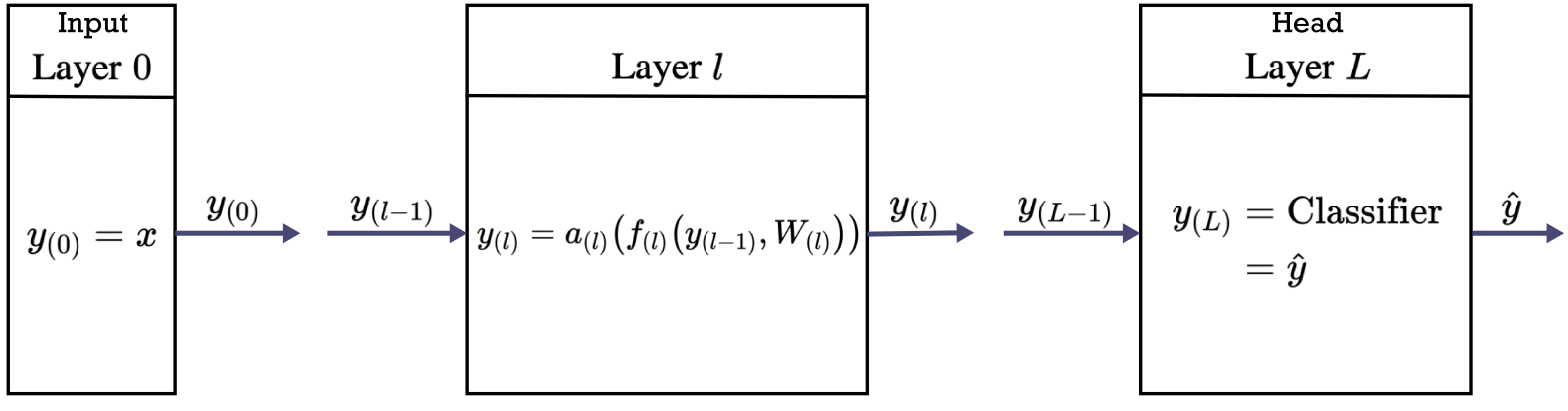


- The output of the penultimate layer $(L-1)$ is used as input to a final layer L that implements either
    - Classification
    - Regression



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 similar to the Feature Engineering pipeline of Classical ML**
    - **Implemented in many ML toolkits (e.g., `sklearn`)**
- **Where the final version of the transformed data is fed into a Classifier/Regression model**

As we will come to see:

- **Another key difference from Classical Machine Learning**
- **Is that the transformations we use in Deep Learning are non-linear**

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.

Let's review our Basic Notational standards (ML_Notation.ipynb)

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

What does a layer do ?

As we saw, Fully Connected layer l

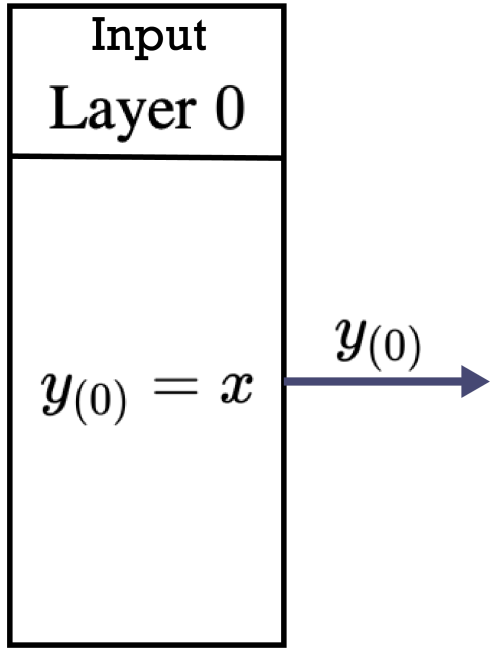
- **Computes $n_{(l)}$ features**
- **Each feature $\mathbf{y}_{(l),j}$ is the dot product of $\mathbf{y}_{(l-1)}$ and $\mathbf{W}_{(l),j}$**

We would argue that the dot product is nothing more than pattern matching

- **$\mathbf{W}_{(l),j}$ is a pattern**
- **That layer l is trying to match against layer $(l-1)$ output $\mathbf{y}_{(l-1)}$**

With this pattern/template matching intuition in mind, let's revisit our path through the layers.

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

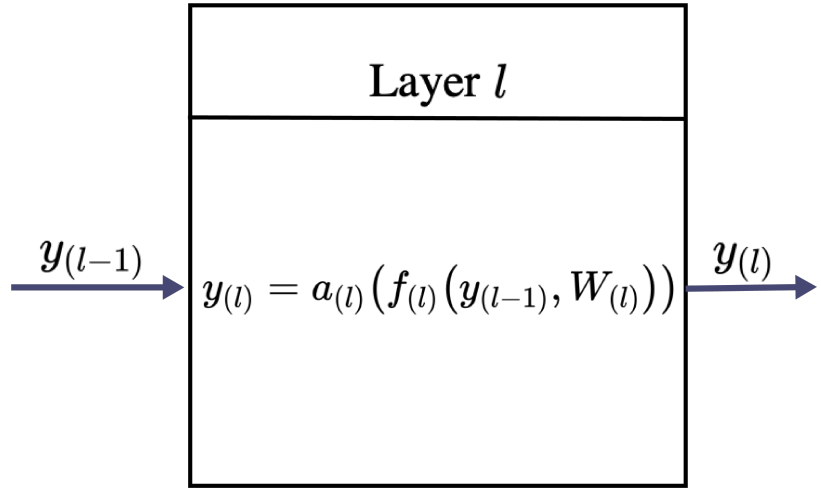


Layer 1 is attempting to match our input $\mathbf{x}$ against $n_{(1)}$ patterns.

So $\mathbf{y}_{(1)}$ is a vector of synthetic features (an alternate representation of $\mathbf{x}$)

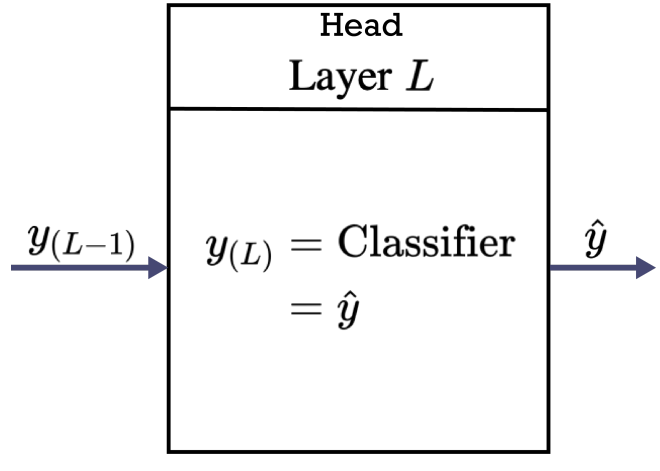
- **where each feature in $\mathbf{y}_{(1)}$ is an indicator as to whether a particular pattern appears in $\mathbf{x}$.**

Layer l works similarly except that the patterns it matches are against synthetic feature in $\mathbf{y}_{(l-1)}$rather than $\mathbf{x}$.



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

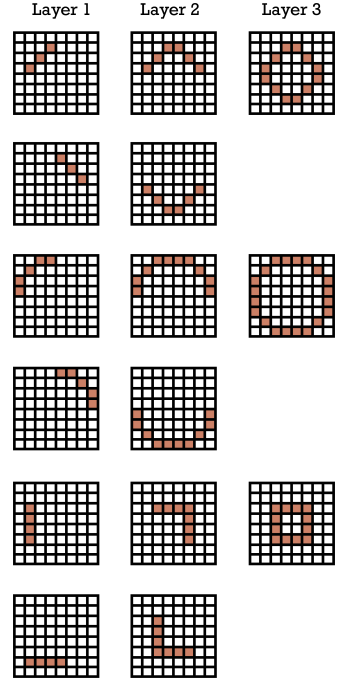
- Is a representation of the input $\mathbf{x}$
- That has been sufficiently transformed
- So that a Classifier/Regression model at layer L can be successful



To summarize, our intuition is 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)**
- **Layer l recognizes features that are *combinations* of layer $(l - 1)$ concepts**

Features by layer



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

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

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 involves $||\mathbf{y}_{(l-1)}|| + 1$ 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 $||\mathbf{y}_{(l)}|| * (||\mathbf{y}_{(l-1)}|| + 1)$**

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 linear dot product followed by an non-linear activation**

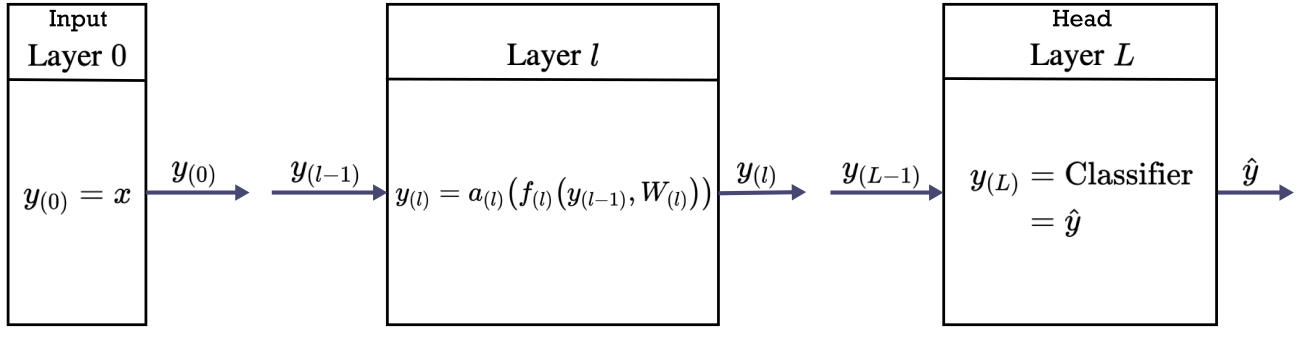
It turns out that the non-linear activation function is *one of the keys* to Neural Networks !

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

Final layer: Regression/Classification

The Head Layer (layer L)

- **takes the final representation of the data**
- **applies a layer determined by the task**
    - **Regression**
    - **Classification**



- **Regression is nothing more than a dot product**
    - $\mathbf{y}_L = \mathbf{y}_{(L-1)} \cdot \mathbf{W}_L$
    - **Implemented by a Fully Connected layer with no activation**
- **Classification is nothing more than a dot product followed by a sigmoid activation**
    - $\mathbf{y}_L = \sigma(\mathbf{y}_{(L-1)} \cdot \mathbf{W}_L)$
    - **Exactly the same way as discussed in our "Classical" Machine Learning lecture**
    - **Implemented by a Fully Connected layer with a sigmoid activation**

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 compute a function.

Gradient Descent needs to take the gradient of this function (evaluated on a mini batch 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 evaluates 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 derivatives 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 because it records

- **the arguments are `a, b`**
- **the operation is addition**
- **we know the derivative of an addition operation**

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

- **We will see some pseudo-code that shows how this is done**
- **Check out the Deeper Dive on Computation Graphs (Computation_Graphs.ipynb) if you want to know more**

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 seen, 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.

And, don't forget, the "batch index" dimension

- **Tensorflow processes *mini-batches* of examples, not singeltons**
- $\mathbf{y}^{(\mathbf{i})}_{(l),j,j'}$
    - **Element j' of the output (given input examples i) of unit j of layer l : $\mathbf{y}_{(l),j,j'}$**

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

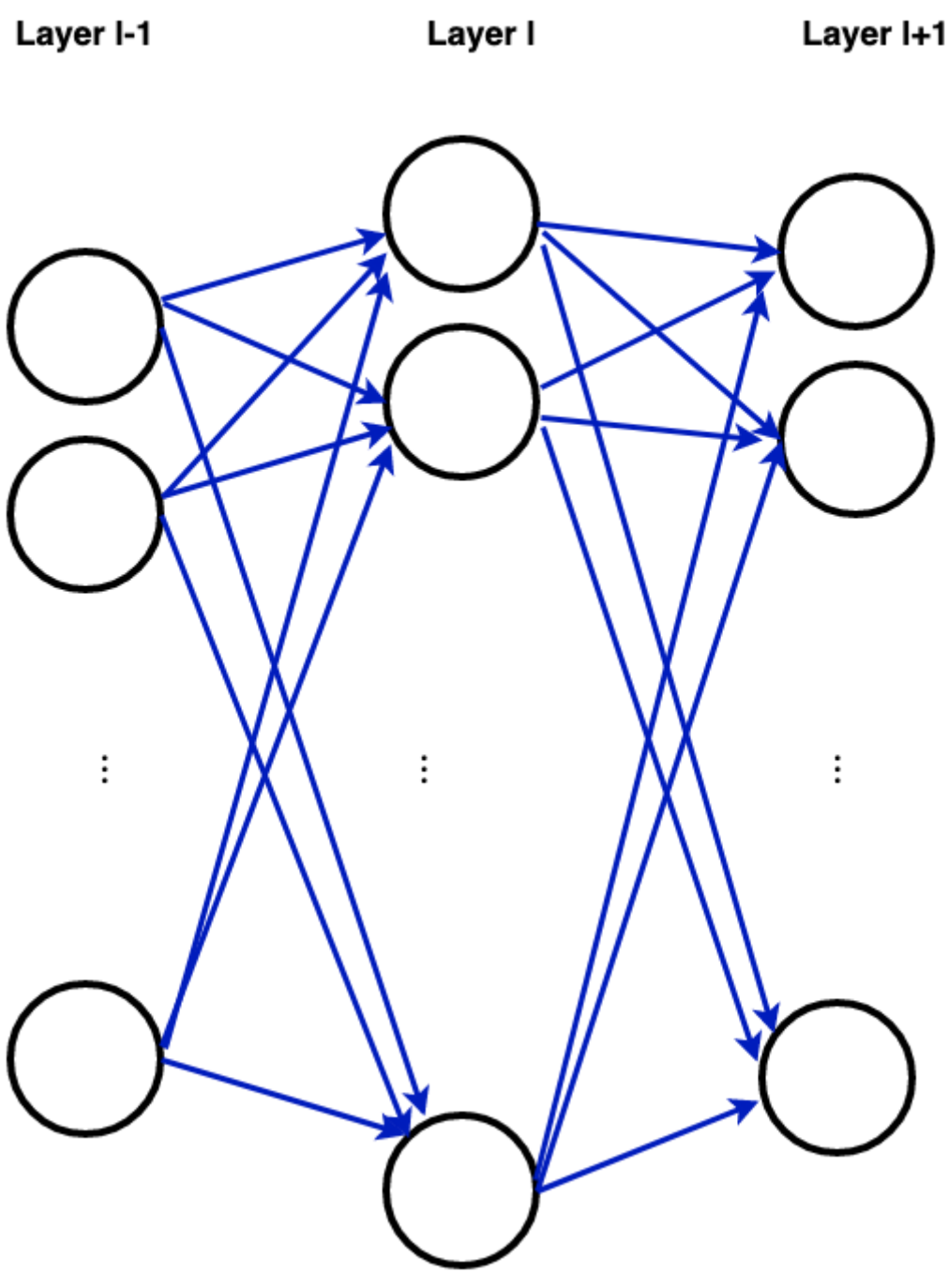
The above description of a Neural Network imposed several restrictions on units (neurons)

- **Each layer is homogeneous**
- **Layers are organized sequentially**

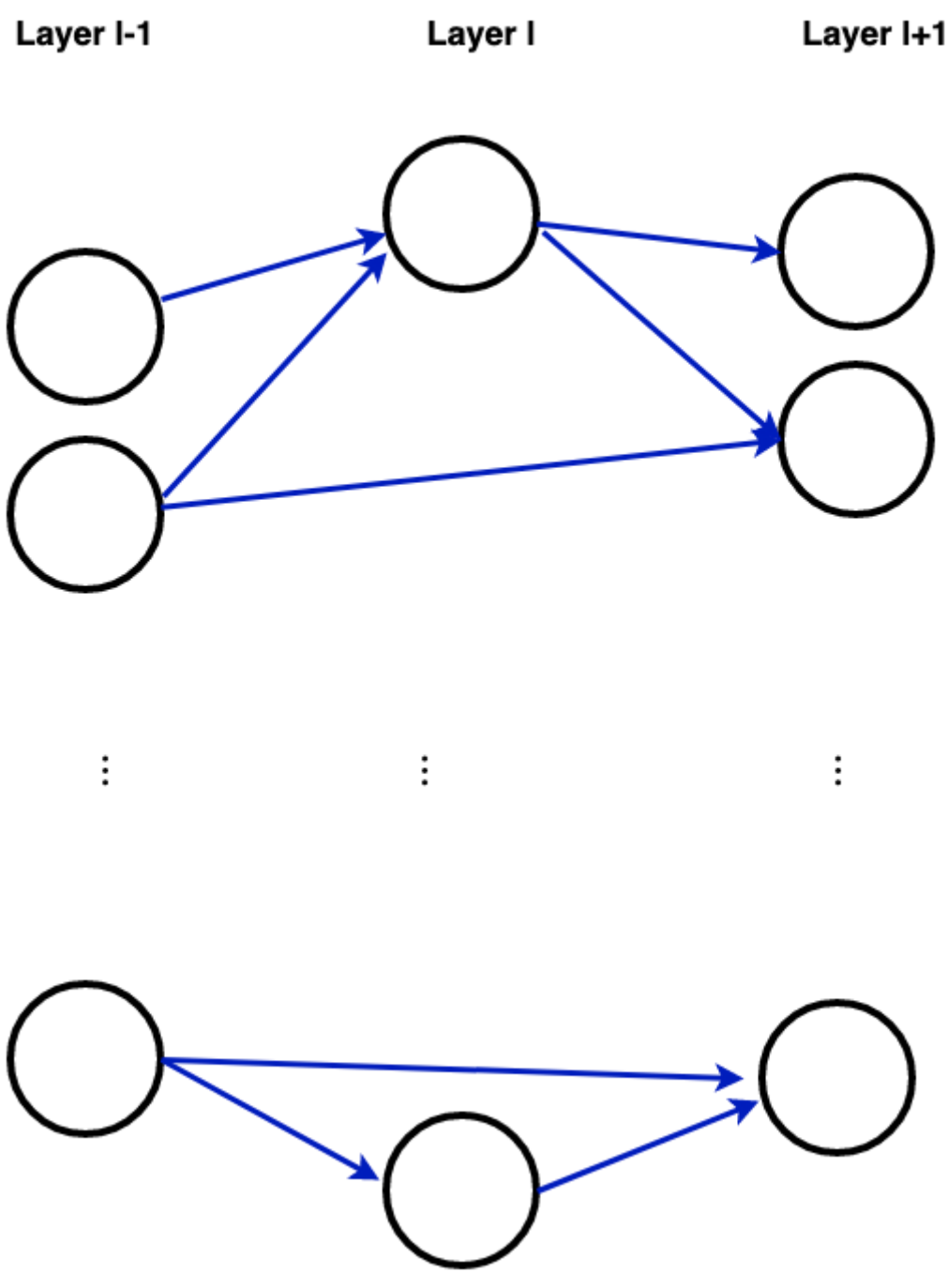
One can imagine a network graph without these restrictions

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

Sequential architecture


Layer l-1
Layer l
Layer l+1

Functional architecture


Layer l-1
Layer l
Layer l+1

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: second "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 ?

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 (https://cloud.google.com/tpu/docs/tpus) (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 [5]:
```
print("Done")
```

```
Done
```