Machine Learning from the ground up

aka: **Demystifying ML**

Writing a mock DNN from scratch using only numpy

Demystifying ML

- Intro, what do ML experts and Scientists have in common
- Intro to ML model design
- Intro to Backpropagation
- How to write this into code
- Examples from writing and training a simple DNN from scratch
- Considerations around DNN models and some of their limitations

Training a model on data

- "**Training a model on data**" is the same as "**fitting a PDF to a dataset**".

- We want to tune free-parameters in the model(*PDF*) to find the set of parameters corresponding to the **maximum likelihood**.

- This is the same as saying we want to find the **minima** of

  -log(*Likelihood*)

- This is the same as we want to **minimize the loss of a model for a given dataset**.

- To paraphrase a conversation, I once had:

  "Newton-Raphson the truth out of it until you're done"

“Training” according to a Scientist

- In science fitting is ‘easy’ I have ~10s of free parameters ***p*** and dataset ***x*** which I need to minimize such that:

- Best solution for that? Minuit(2)

$$-\frac{d\ln\left(\mathcal{L}(\boldsymbol{p};\boldsymbol{x})\right)}{d\boldsymbol{p}} = \frac{d\ln\left(PDF(\boldsymbol{p};\boldsymbol{x})\right)}{d\boldsymbol{p}} = 0$$

1. Take (2*parameter+1) “steps” for each ***epoch***
   - Each step is evaluating $\mathcal{L}(\boldsymbol{p} + \boldsymbol{\Delta p}; \boldsymbol{x})$ <u>for all ***x***</u> at 1 new point.

2. Calculate gradients using these steps and estimate distance to the minima
3. Adjust ***p*** and loop back to 1 until a ‘true’ minima has been reached (*no guarantee this is global*)
4. Make sure *Jacobian*&*Hessian* are well defined

- For each parameter report final values and errors(uncertainty) at the minima.

All very scientific, all very ‘*simple*’…

Training according to a Scientist

- Evaluating a complex PDF for the **whole dataset** is ***expensive***.

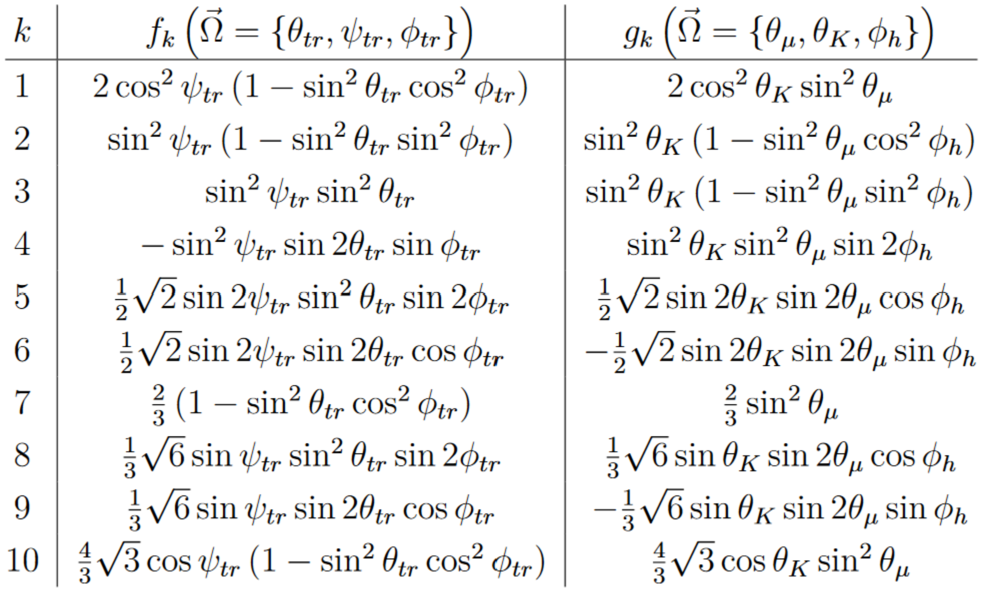
- PDF is often effectively a custom written, complex piece of software considering many physical properties/relationships/theories.
  (For 99.999…% of the time also normalised to 1.0!)

- Automated tools like Minuit cope will with this by adjusting from our starting parameters and step-sizes to find a minima.

- Traditionally **every time** we evaluate our PDF in science, we typically must do this for the **whole dataset**.
- This is because our parameters have some physical interpretation.

Training according to a Scientist

- **PDFs** are complex and fun:



k	$f_k\left(\vec{\Omega}=\{\theta_{tr},\psi_{tr},\phi_{tr}\}\right)$	$g_k\left(\vec{\Omega}=\{\theta_{\mu},\theta_K,\phi_h\}\right)$
1	$2\cos^2\psi_{tr}\left(1-\sin^2\theta_{tr}\cos^2\phi_{tr}\right)$	$2\cos^2\theta_K\sin^2\theta_\mu$
2	$\sin^2\psi_{tr}\left(1-\sin^2\theta_{tr}\sin^2\phi_{tr}\right)$	$\sin^2\theta_K\left(1-\sin^2\theta_\mu\cos^2\phi_h\right)$
3	$\sin^2\psi_{tr}\sin^2\theta_{tr}$	$\sin^2\theta_K\left(1-\sin^2\theta_\mu\sin^2\phi_h\right)$
4	$-\sin^2\psi_{tr}\sin 2\theta_{tr}\sin\phi_{tr}$	$\sin^2\theta_K\sin^2\theta_\mu\sin 2\phi_h$
5	$\frac{1}{2}\sqrt{2}\sin 2\psi_{tr}\sin^2\theta_{tr}\sin 2\phi_{tr}$	$\frac{1}{2}\sqrt{2}\sin 2\theta_K\sin 2\theta_\mu\cos\phi_h$
6	$\frac{1}{2}\sqrt{2}\sin 2\psi_{tr}\sin 2\theta_{tr}\cos\phi_{tr}$	$-\frac{1}{2}\sqrt{2}\sin 2\theta_K\sin 2\theta_\mu\sin\phi_h$
7	$\frac{2}{3}\left(1-\sin^2\theta_{tr}\cos^2\phi_{tr}\right)$	$\frac{2}{3}\sin^2\theta_\mu$
8	$\frac{1}{3}\sqrt{6}\sin\psi_{tr}\sin^2\theta_{tr}\sin 2\phi_{tr}$	$\frac{1}{3}\sqrt{6}\sin\theta_K\sin 2\theta_\mu\cos\phi_h$
9	$\frac{1}{3}\sqrt{6}\sin\psi_{tr}\sin 2\theta_{tr}\cos\phi_{tr}$	$-\frac{1}{3}\sqrt{6}\sin\theta_K\sin 2\theta_\mu\sin\phi_h$
10	$\frac{4}{3}\sqrt{3}\cos\psi_{tr}\left(1-\sin^2\theta_{tr}\cos^2\phi_{tr}\right)$	$\frac{4}{3}\sqrt{3}\cos\theta_K\sin^2\theta_\mu$

- In **PPE** particles are distributed across all possible phase-space allowed by nature.

- Fitting/Training process isn’t just about extracting signal features over background/noise. ***PDF*** must describe both, i.e. signal+background.

Training according to a Machine Learning expert

- I have **millions of free parameters** (*or more*). I don’t care about their physical interpretations.

- I want to minimize these parameters to minimize a loss-function($\mathcal{L}$) to most closely matches what I see in my training data.

$$-\frac{d\ln\left(\mathcal{L}(\boldsymbol{p};\boldsymbol{x})\right)}{d\boldsymbol{p}}=\frac{d\ln\left(loss(\boldsymbol{p};\boldsymbol{x})\right)}{d\boldsymbol{p}}=0$$

1. Batch up my data
2. Evaluate my loss function for a batch
3. Calculate the gradients of my parameters in n-dimensional fit-space
4. Update my parameters and move onto the next batch. After using all batches start again.

5. Repeat 2-4 until I’ve reached the number of epochs I’m interested in
6. Repeat steps 1-5 until I’m “happy”…

Happy here is subjective, it often changes depending on what the user is trying to do.

Yes, it’s wishy-washy and full of soft-squishy stuff. **IMHO**: “*it’s not very clean or scientific*”

Training according to a ML expert

- This process needs to evaluate the gradient of all parameters ***p*** in n-dimensional fit-space, this is done via back-propagation.

(NB: more on this later!)

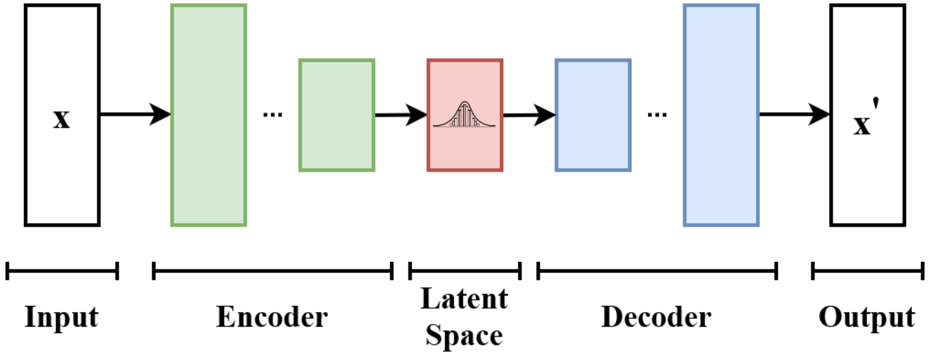
- Actual **function being evaluated** is often described in terms of models with data flowing between layers. Often, **much simpler** than complex models used by a scientists.

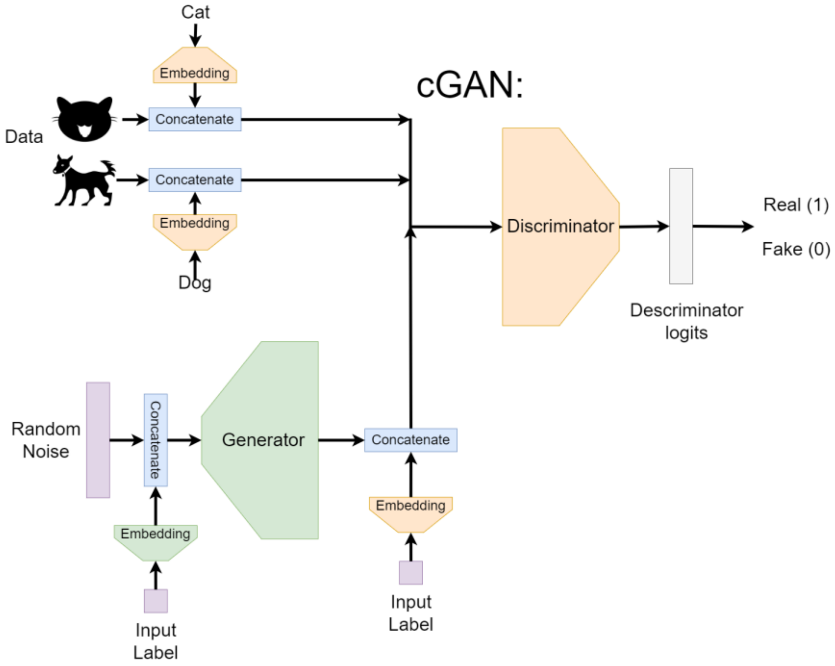
- Compared to Minuit, less steps are involved, and the minimization can be much quicker.
- Unlike Minuit where we take an approach using all information we have; **here we rely on a statistically guided tools** to take us to the minima much quicker.

- We only need to evaluate small batches of data at a time.
  That is good because that’s how computers tend to like to work ☺

Training according to a ML expert

- Sample the input data
- Build a model as a graph
- Pass data forward and backward through model to evaluate
- Use backpropagation to differentiate
- Trained model <u>aims</u> to be *realistic*…





Training, according to different communities…

	Physics Analysis	ML model training
Minima	*Well defined minima* with a well-defined error	Minima is when training has given the ‘***best’*** outcome
Training/Free Parameters	***Physical interpretation of each parameter*** makes fine-tuned, clear understanding of limits and constraints possible	***Parameters are effectively random*** with only limits being statistical in nature
Fit-Space	Fit-space normally <<**100 dimensions**	Fit-space >> **1,000,000,000 dimensions (!)**
PDF/Model	Well defined but **complex**. Normalised to 1. Whole dataset evaluated for every epoch. Intended to describe the real world. Almost never fully analytically differentiable. Every single event is evaluated against whole complex PDF to determine probabilities.	Loss function is **simple**. Models control complex flow of data during training. Normalisation often handled through training. Used to extract structure from real-world during training. Always analytically differentiable. Model designed to extract signal and remove/ignore background but often has no knowledge of how to best separate the 2.

So why now (now being 2020+)?

- OK, great ideas, Minuit was published in 1973, why did the latest round of ML/AI companies only launch 50yr later?

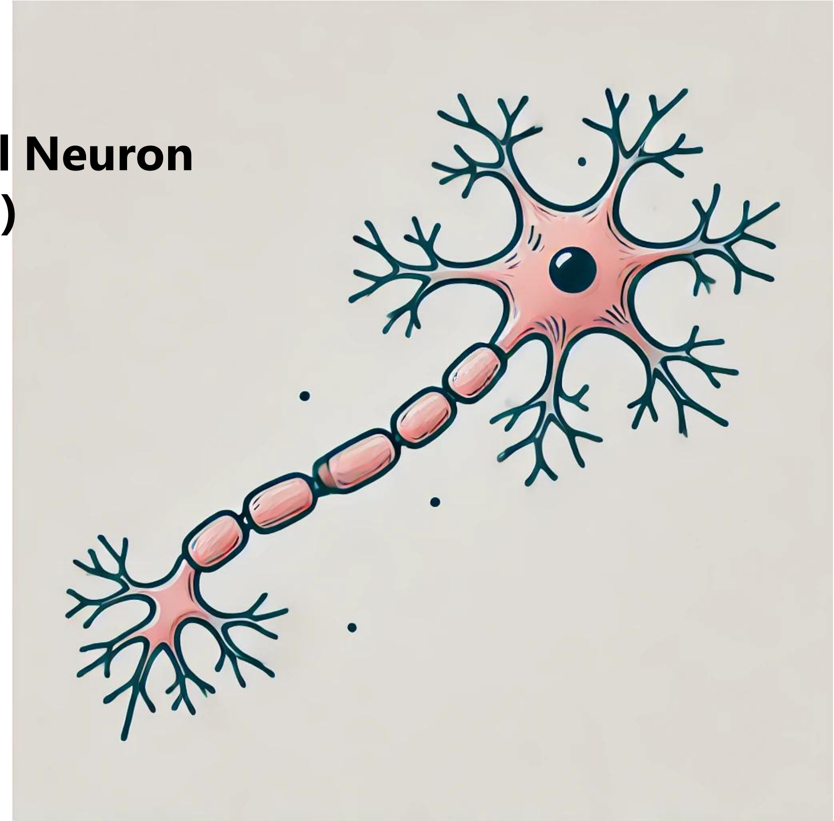
Bandwidth…

- ML/AI training shuffles data around at **huge** rates that have only been possible with *consumer-facing* tech for the past 5-10yr.

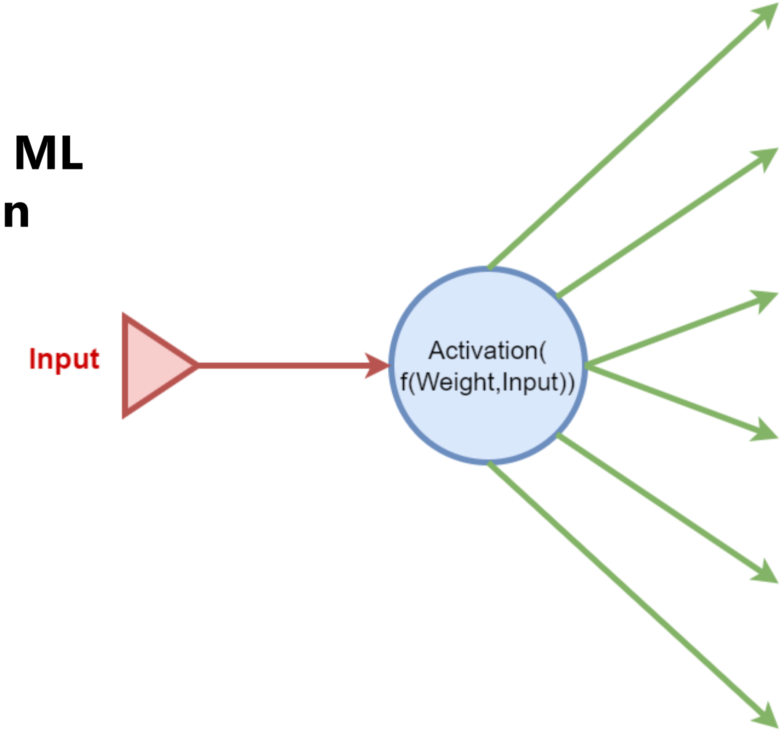
Neural Networks – The Neuron

Artificial **N**eural **N**etworks are inspired by the way neurons work in nerve clusters in biological systems.

Biological Neuron (ChatGPT)

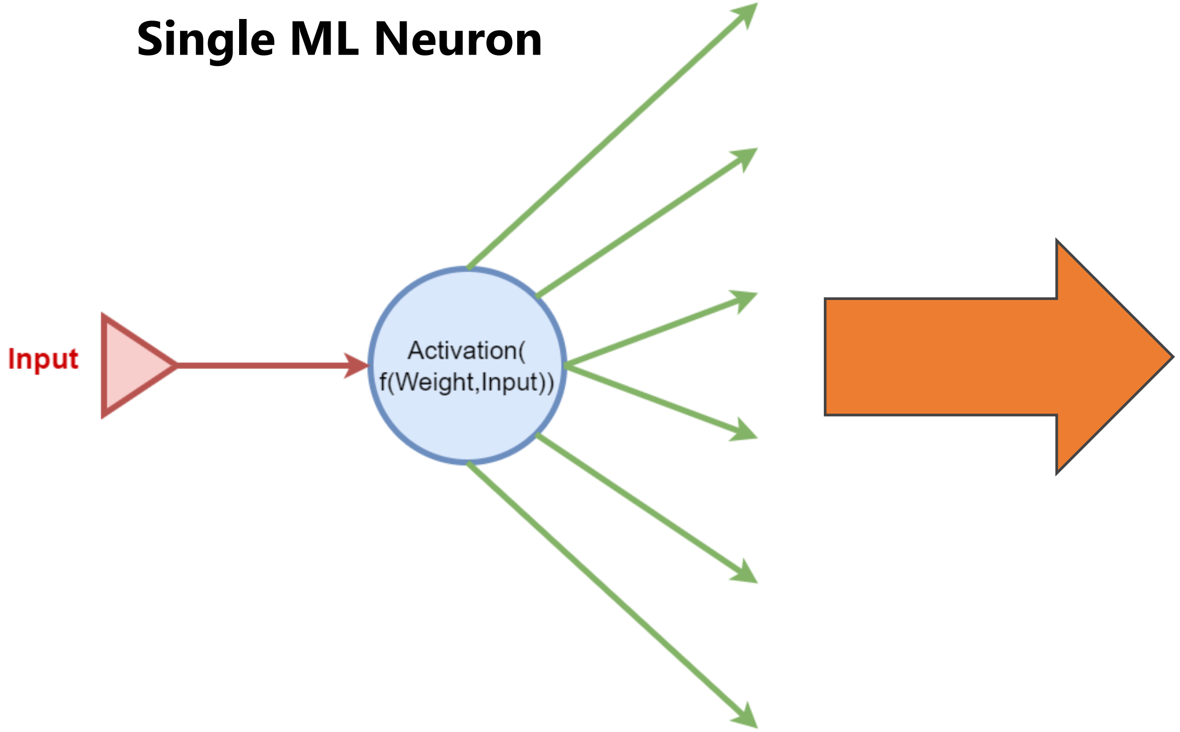


Single ML Neuron

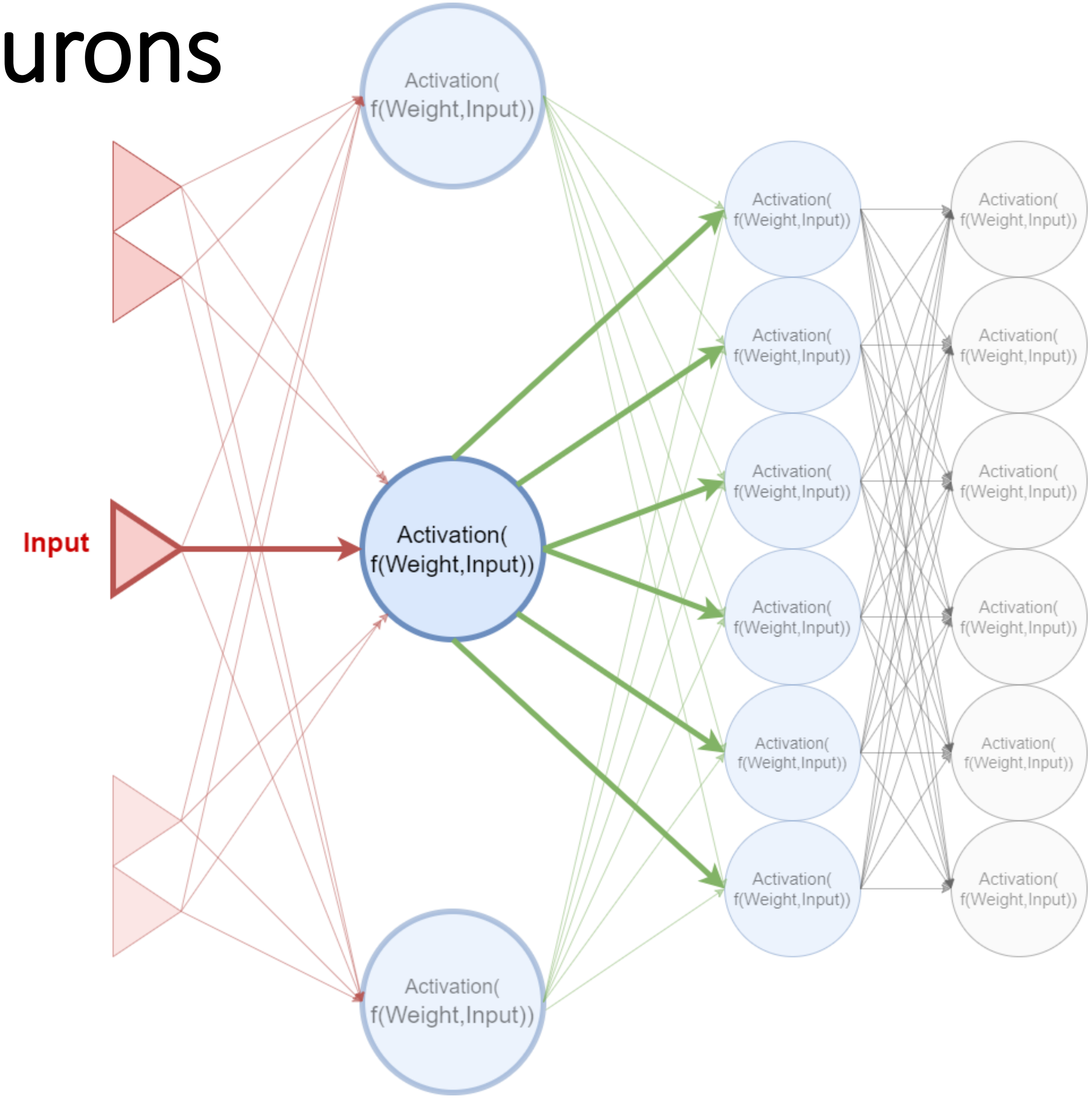


Neural Networks – Neurons

Single ML Neuron



ML Neuron Network



$$A(x \odot W + b) = \hat{y}$$

Neural Networks – Neurons

- The behaviour of an **ANN** neuron is inspired by the biological equivalent:

$$\boldsymbol{A}(\boldsymbol{x} \odot \boldsymbol{W} + \boldsymbol{b}) = \widehat{\boldsymbol{y}}$$

- The **input** to the neuron is acted on by the **weights** in the neuron.

  (Simplest case is that the **input** is ***multiplied*** by the **weight(s)** and **summed**).

- *After* the neuron has acted on all the **inputs** the **output** of the neuron comes from applying an **Activation** function and optionally adding a **Bias**.

- **Activation** functions are required to allow the model to describe non-linear data.

- *So far basic multi-dimensional (linear-) algebra(!)*

ANN – Universal Approximation Theorem

“… *with a* ***single hidden layer*** *can be shown to be universal approximators*.” ...
“*However, the* ***required number of hidden units may be very large***.”

– Ian Goodfellow et al.

(Emphasis mine)

$$[I_1 \quad I_2] \times \begin{bmatrix} W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 \end{bmatrix} = [A_1 \quad A_2 \quad A_3]$$

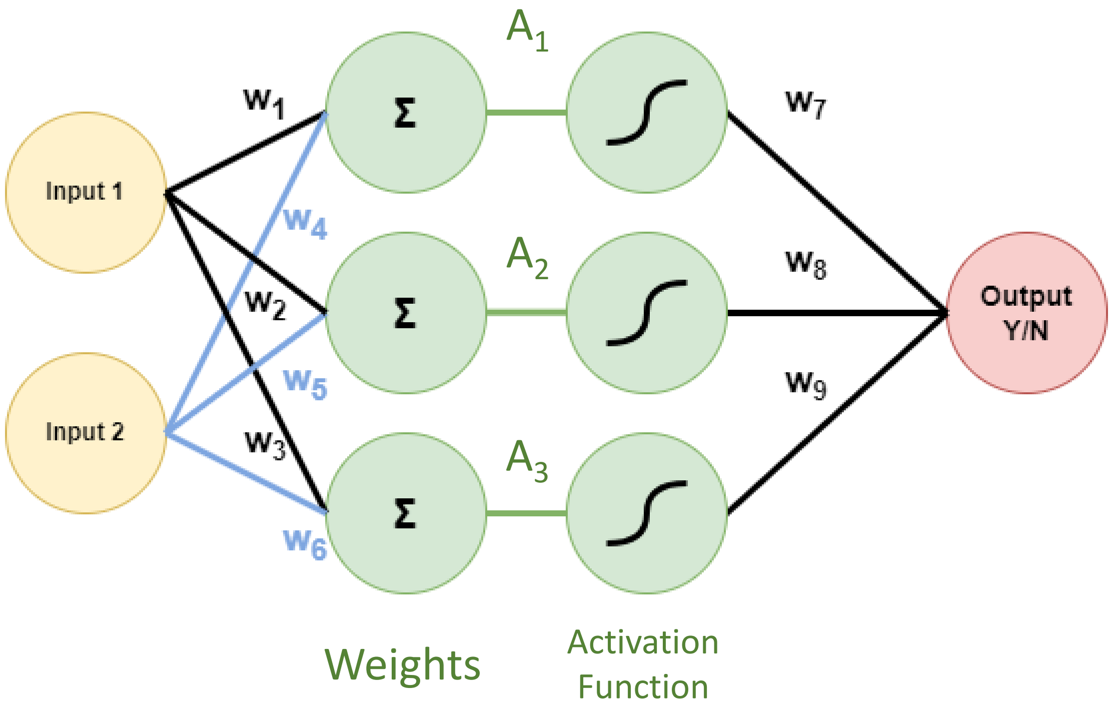
ML model design

$$[A_1 \quad A_2 \quad A_3] \times \begin{bmatrix} W_7 \\ W_8 \\ W_9 \end{bmatrix} = [Output]$$

- Looking at the simplest model, input passing through 1 Dense layer to give some output. (Most basic, slightly crap, classifier)

- Input $\boldsymbol{x}$ is operated on by Weights $\boldsymbol{W}$

- We can also apply an activation function, but for the simple case let's ignore this. *(only for now!)*



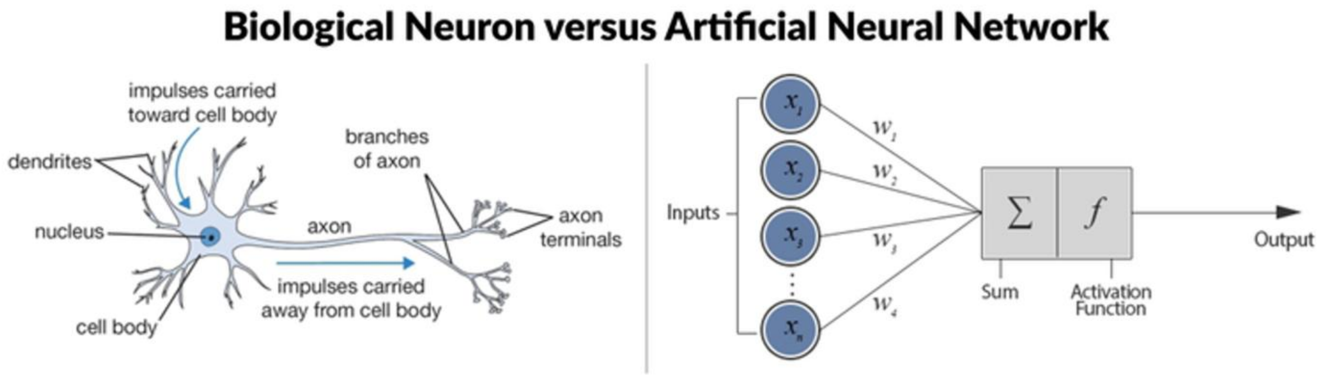
ML model design – Neurons

- Every single neuron in a normal Dense layer is made of a **function**, an **activation** and a **bias**.

(output)

$$\boxed{\hat{y} = A(x \odot W + b)}$$

- The activation function keeps the output of the operation on the input within a given range and the bias adjusts the output in a normal linear manner.



Biological Neuron versus Artificial Neural Network

https://www.datacamp.com/tutorial/deep-learning-python 2023

ML model design

- Since we have a classifier, we are going to minimize the function which calculates the difference between the model output $\hat{\mathbf{y}}$ and the truth $\boldsymbol{y}$.

$$\boldsymbol{L}(\boldsymbol{y};\hat{\mathbf{y}}) = \boldsymbol{loss}(\boldsymbol{y};\hat{\mathbf{y}}) = \frac{(\hat{\mathbf{y}} - \boldsymbol{y})^2}{2} \quad , \qquad \hat{\mathbf{y}} = \boldsymbol{A}(\boldsymbol{x} \odot \boldsymbol{W} + \boldsymbol{b})$$

- For simplicity we're going to choose $\boldsymbol{b} = 0$ and that $\boldsymbol{A}$ is a simple pass-through function.

*
Different implementations add/remove factors of 2 or when we're –ve/+ve, this doesn't impact the result.

ML model design – Backpropagation

- The free parameters we are left with for this example are: $\boldsymbol{W}$

- Now we want to calculate: $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = 0$

- Using the chain rule: $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{L}}{\partial \widehat{\boldsymbol{y}}} \times \frac{\partial \widehat{\boldsymbol{y}}}{\partial \boldsymbol{W}}$

$$\frac{\partial \boldsymbol{L}}{\partial \widehat{\boldsymbol{y}}} = \frac{-\boldsymbol{2}(\boldsymbol{y}-\widehat{\boldsymbol{y}})}{2} = (\widehat{\boldsymbol{y}} - \boldsymbol{y}) \qquad , \qquad \frac{\partial \widehat{\boldsymbol{y}}}{\partial \boldsymbol{W}} = \boldsymbol{x}$$

- Therefore, we want to get:

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \boldsymbol{x}(\widehat{\boldsymbol{y}} - \boldsymbol{y}) = 0$$

*= layer input *(output gradient)*

ML model design – Backpropagation

- For a given set of weights $\boldsymbol{W}$, we now know the gradient, based on a set of observables to be: $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \boldsymbol{x}(\widehat{\boldsymbol{y}} - \boldsymbol{y})$

- If we want to change $\boldsymbol{W}$ such that we end up at zero gradient, we can step down the gradient (in n-dimensions!) by some small step-size using the “Learning Rate” $\boldsymbol{LR}$.

- Hence, we can adjust our parameters to $\boldsymbol{W}$ be:

$$\boldsymbol{W}' \Rightarrow \boldsymbol{W} - \boldsymbol{LR} \times \boldsymbol{x}(\widehat{\boldsymbol{y}} - \boldsymbol{y})$$

ML model design – Backpropagation

OK, all sounds easy, why not just take a bigger steps and call it finished?

- The magnitude of the gradient of the weights can vary over n-dimensions.
- Taking a step in one dimension may be correlated with the gradient in another dimension.
- The gradient function is also expected to be highly non-linear.
- We are almost always starting from a random position in fit-space and don’t know how far from the best minima we are.

Scientists shouldn’t be surprised by this; this isn’t yet anything *“new”*.

ML model design – Backpropagation

- OK, let's go back to our ‘simple’ example and make it a ***bit more*** realistic.

- Assuming we add a bias to our dense layer:$\boldsymbol{b}$

- To first order we will assume the Weights($\boldsymbol{W}$) and biases($\boldsymbol{b}$) can be handled separately.
- Therefore, we can minimize the loss function separately for each:

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{\hat{y}}} = \frac{-2(\boldsymbol{y}-\boldsymbol{\hat{y}})}{2} = (\boldsymbol{\hat{y}} - \boldsymbol{y}) \quad , \frac{\partial \boldsymbol{\hat{y}}}{\partial \boldsymbol{b}} = 1 \qquad => \quad \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{b}} = (\boldsymbol{\hat{y}} - \boldsymbol{y})$$

ML model design – Backpropagation

- OK, let's go back to our ‘simple’ example and make it a bit more realistic.

- Assuming we use an Activation function: $\mathbf{A} = \tanh(\boldsymbol{x})$

(**NB:** not a random function but chosen because of its mathematical properties!)

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{A}} \times \frac{\partial \boldsymbol{A}}{\partial \boldsymbol{W}} \Rightarrow \qquad \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{A}} = 1 - \boldsymbol{A}(\boldsymbol{W})^{\mathbf{2}}\,, \quad \frac{\partial \boldsymbol{A}}{\partial \boldsymbol{W}} = \boldsymbol{x}(\widehat{\boldsymbol{y}} - \boldsymbol{y})$$

- Hence: $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \left(1 - \boldsymbol{A}(\boldsymbol{W})^{\mathbf{2}}\right) \boldsymbol{x}(\widehat{\boldsymbol{y}} - \boldsymbol{y})$

*= layer input *(output gradient)*

ML model design – Backpropagation

- So, for a more complete 1-Dense layer model what do we need:

- Gradient from previous layer: $\boldsymbol{\partial}_{n-1} = (\hat{\boldsymbol{y}} - \boldsymbol{y})$

- Differential of Weights in layer: $\left(1 - \boldsymbol{A}(\boldsymbol{W})^{\mathbf{2}}\right)\boldsymbol{x}\partial_{l-1}$, this just depends on the output from this layer L_{n-1}.

So: $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \left(1 - \boldsymbol{L}_{n-1}^{2}\right)\boldsymbol{x}\boldsymbol{\partial}_{n-1}$

***NB:** The (1-Output^2) only applies when using Tanh(!)*

- Differential of Biases in a layer: $(\hat{\boldsymbol{y}} - \boldsymbol{y})$ = Gradient from previous layer…

ML model design – Backpropagation

- OK, so what if I apply more than 1 layer?

- Chain rule saves us again:

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{Layer_n}} \times \frac{\partial \boldsymbol{Layer_n}}{\partial \boldsymbol{W}} \quad , \quad \frac{\partial \boldsymbol{Layer_n}}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{Layer_n}}{\partial \boldsymbol{Layer_{n-1}}} \times \frac{\partial \boldsymbol{Layer_{n-1}}}{\partial \boldsymbol{W}}$$

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{Layer_n}} \times \frac{\partial \boldsymbol{Layer_n}}{\partial \boldsymbol{Layer_{n-1}}} \times \frac{\partial \boldsymbol{Layer_{n-1}}}{\partial \boldsymbol{W}}$$

ML model design – Backpropagation

- When building a modern ML/AI model the requirement to be able to back-propagate means we're limited to functions we can analytically differentiate.
  (*Or analytically approximate the differentiation of if it's "good enough"*)

- This is a key difference to a lot of "scientific" numerical fitting where the gradient is almost never known analytically but is calculated purely numerically.

DNN – Training (Back Propagation)

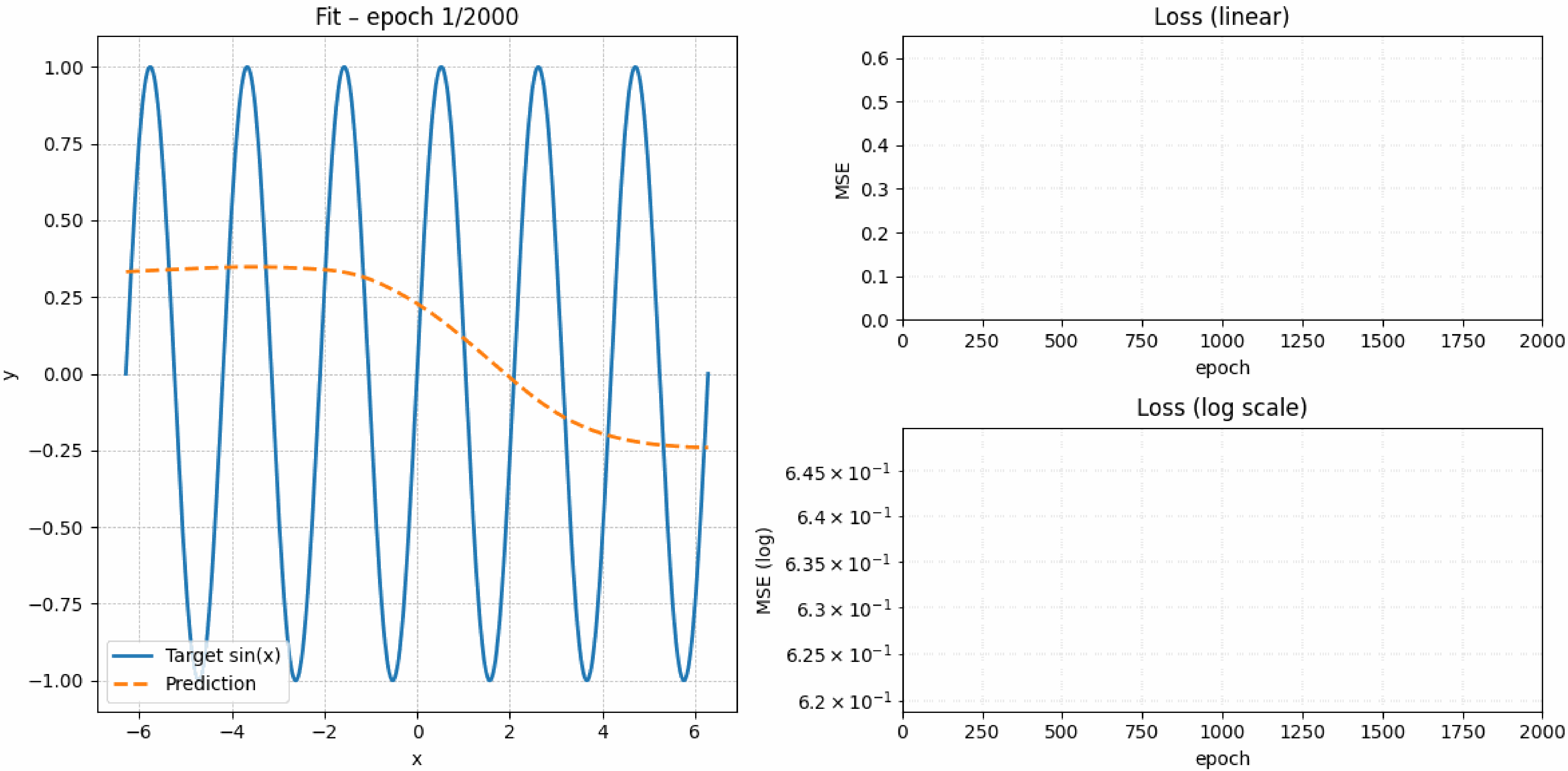
- OK, so we have a method for calculating the gradient of each parameter in the model.

- We know we want to reach the true global minima of the model.
- This would give the version of the model which has the smallest error. This should be the version of the model which best describes the input data.

- To get there we **take a step** down the gradient.
- The size of the step is determined by the **Learning Rate**.

DNN – Training (Back Propagation)

- When we take this “step” we update our free parameters in the model and re-evaluate.
- After this the gradient has now changed.
- We go back into the back-propagation method, update and repeat until the gradient reaches zero.
- *We have now trained our model by minimizing our gradient.*


Fit – epoch 1/2000
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−6
−4
−2
0
2
4
6
x
Target sin(x)
Prediction
Loss (linear)
0.6
0.5
0.4
0.3
0.2
0.1
0.0
MSE
0
250
500
750
1000
1250
1500
1750
2000
epoch
Loss (log scale)
6.45×10^{-1}
6.4×10^{-1}
6.35×10^{-1}
6.3×10^{-1}
6.25×10^{-1}
6.2×10^{-1}
MSE (log)
0
250
500
750
1000
1250
1500
1750
2000
epoch

ML model design – Writing Code

- Input data is passed as a ***batch*** to a model to be evaluated/train/make-predictions.

- For a Dense layer, a ***batch*** of **n** input elements is mapped to **m** neurons giving us a matrix of $(\boldsymbol{n} \times \boldsymbol{m})$ dimensions.
  With a vector of length $\boldsymbol{m}$ biases.

- Connecting a 2nd Dense layer with an output dimension $\boldsymbol{y_n}$ means we need a 2nd matrix of dimension $(\boldsymbol{m} \times \boldsymbol{y_n})$.

- *We know to evaluate the differential at each layer we need to record the output from each layer and the input from each previous layer.*

ML model desi

The model might look like this.

Layer1

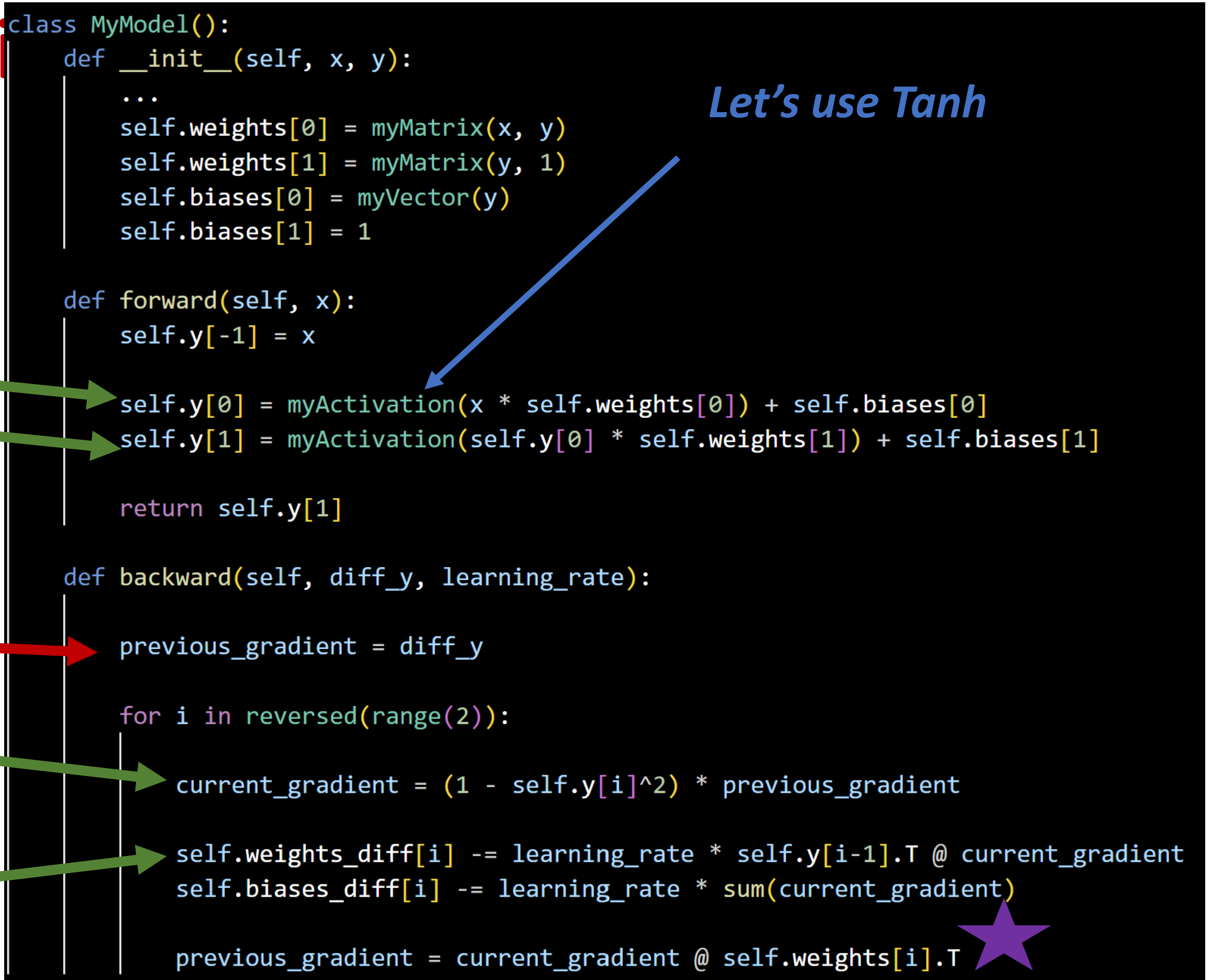
Layer2

Need to start with the gradient at the end.

$$\left(1 - L_n^2\right) x \partial_n$$

"Undo" the Tanh Activator

$$W \Rightarrow W - LR \times \frac{\partial L}{\partial W}$$

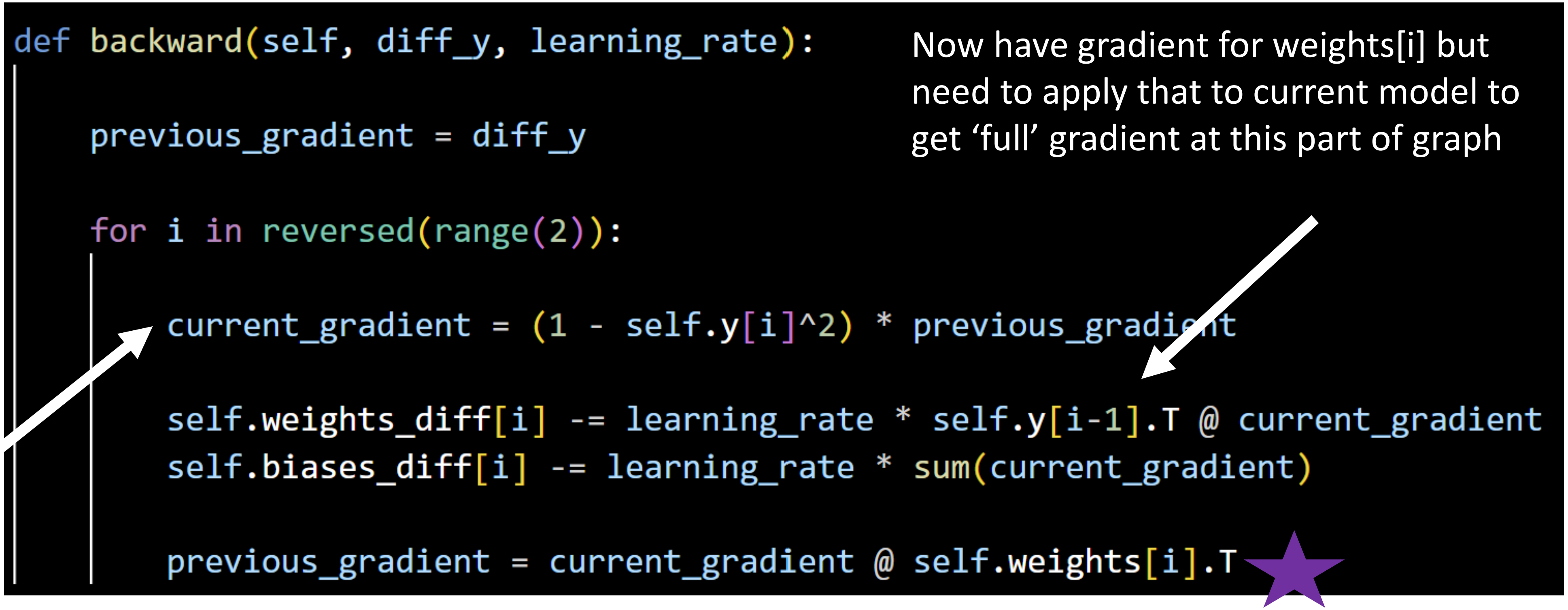


Let's use Tanh

```
class MyModel():
    def __init__(self, x, y):
        ...
        self.weights[0] = myMatrix(x, y)
        self.weights[1] = myMatrix(y, 1)
        self.biases[0] = myVector(y)
        self.biases[1] = 1

    def forward(self, x):
        self.y[-1] = x

        self.y[0] = myActivation(x * self.weights[0]) + self.biases[0]
        self.y[1] = myActivation(self.y[0] * self.weights[1]) + self.biases[1]

        return self.y[1]

    def backward(self, diff_y, learning_rate):

        previous_gradient = diff_y

        for i in reversed(range(2)):

            current_gradient = (1 - self.y[i]^2) * previous_gradient

            self.weights_diff[i] -= learning_rate * self.y[i-1].T @ current_gradient
            self.biases_diff[i] -= learning_rate * sum(current_gradient)

            previous_gradient = current_gradient @ self.weights[i].T
```

ML model design – Writing Code

Start with initial gradient and loop backwards through graph



```
def backward(self, diff_y, learning_rate):

    previous_gradient = diff_y

    for i in reversed(range(2)):

        current_gradient = (1 - self.y[i]^2) * previous_gradient

        self.weights_diff[i] -= learning_rate * self.y[i-1].T @ current_gradient
        self.biases_diff[i] -= learning_rate * sum(current_gradient)

        previous_gradient = current_gradient @ self.weights[i].T
```

Now have gradient for weights[i] but need to apply that to current model to get ‘full’ gradient at this part of graph

‘Undo’ the gradient modification from our Activation function

Input to step before this needs us to ‘Undo’ the gradient effect from our current set of weights

Saying ‘undo’ here is to help conceptualize what is going on.
What is really going on is the errors in the graph are being decomposed to individual components within the graph using the chain rule so the graph can be minimized to give the smallest total error.

ML model design – Writing Code

- Following through the code there is an important subtlety which is lost when looking through the simplified algebra. *Dimensionality*. ★

- When we move back through the model we need to go backwards up and down in different dimensionality of the different layers to get the **correct** gradients.

- This is achieved by using the Transposed version of the "matrix" previously used when walking forward through the model.

  (Hence why we use ***Weights.T*** *rather than applying the weights again)*

ML model design – Writing Code

- In our example we have:

1. Input of dimension 1

2. connected to a dense layer of dimension$(1, \boldsymbol{x})$
3. connected to a dense layer of dimension $(\boldsymbol{x}, \boldsymbol{y})$
4. connected to a 2nd dense layer of dimension $(\boldsymbol{y}, 1)$

5. giving an output of dimension 1.

- Moving backwards through our network we need to go from 1 to $(\boldsymbol{y}, 1)$ to $(\boldsymbol{x}, \boldsymbol{y})$ back to 1 we achieve this by transposing the original matrices to match the dimensions.

ML model design – Writing Code

- There's a lot of matrices multiplied with potentially many calculations.

- When performing a matrix multiplication in Python with numpy:

  np.matmul(x, W) is the same as using the matrix operator: x **@** W

- There are some functional differences between np.dot and **np.matmul** which shouldn't impact us too much, but beware(!)

- Last week I demonstrated that ***matmul*** and ***@*** can be much faster than pure Python.

ML model design – ~~Writing~~ Reading Code

- Here are some conventions when looking at ML related code (and to some extent algebra):

- $\hat{y}$ this is often the **predicted** or **logits** value from a model
- x is often the **batch** of data being processed, y is labels/truth
- a is often the **output** of an **activated** layer
- z is often the **output** of **non-activated** layer

ML model design – Training

- OK, to train a model this is ‘easy’:

1. Move forwards through the model (save outputs as you go)

2. Move backwards through the model (to calculate gradients)

3. Step in fit-space –ve to the gradient

4. Repeat steps 1-3 until we are “**happy**”

ML model design – Training

- Minuit and other scientific based statistical minimizers often return the absolute likelihood for a whole fit function.
- This is due to the minimizer attempting to move along the surface of the likelihood function to reach a minima.

- Common tools such as **TensorFlow** or **PyTorch** tend to report the ‘average’ of the loss function for the last batch.

  This is normally the mean of the logits but can technically differ.

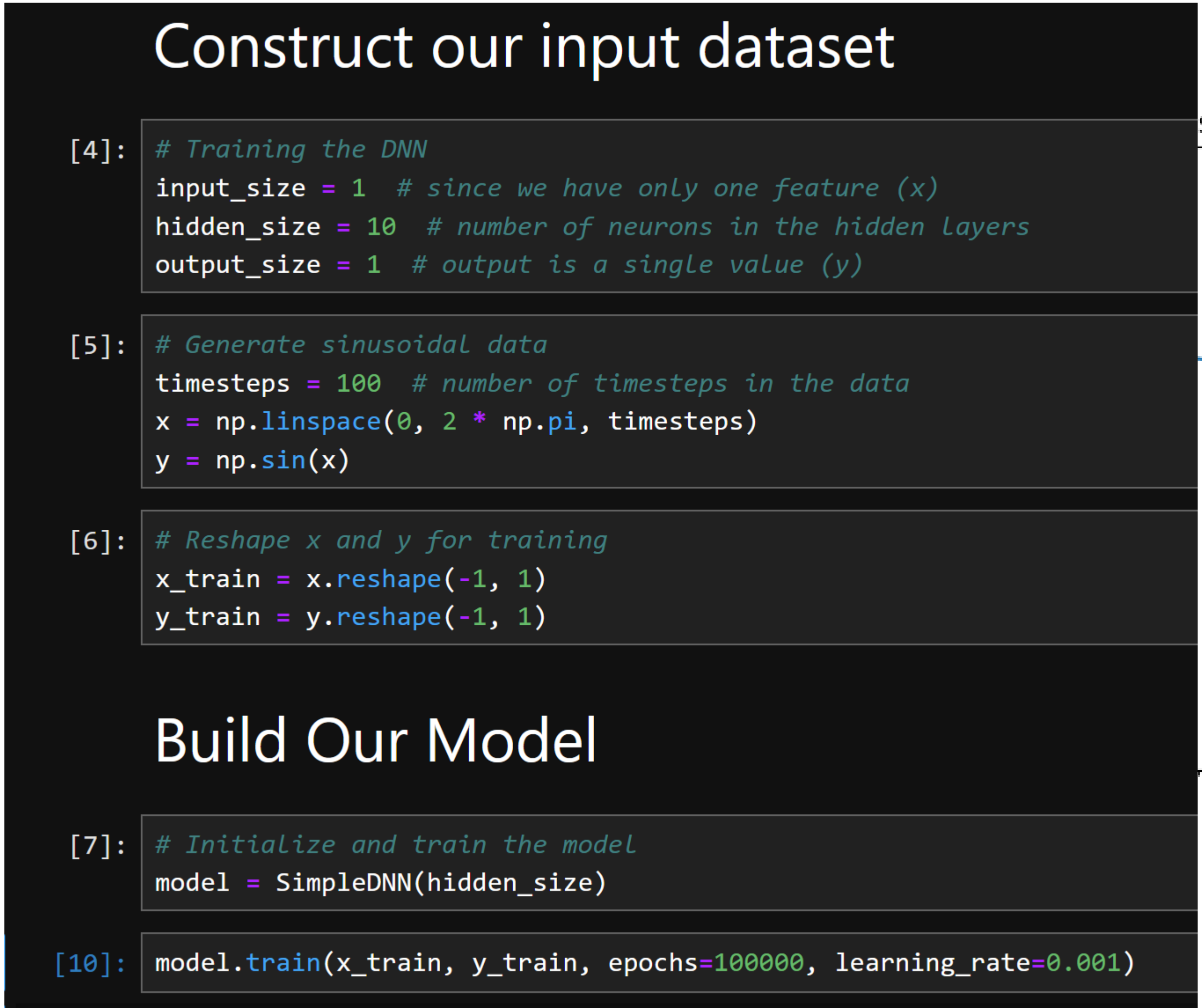
ML model design – Training (by example)

- Training a DNN to a sinusoid dataset allows it to learn the features of a dataset and give back the correct value when correctly trained.

- In our simple case we won't bother to "batch-up" our dataset, we can just evaluate it element by element on a CPU and O(1k) points isn't too intensive.

- The "***secret sauce***" comes from our use of **Activation** Functions(!)

ML model design – Training (by example)



Construct our input dataset

```
[4]: # Training the DNN
     input_size = 1  # since we have only one feature (x)
     hidden_size = 10  # number of neurons in the hidden layers
     output_size = 1  # output is a single value (y)
```

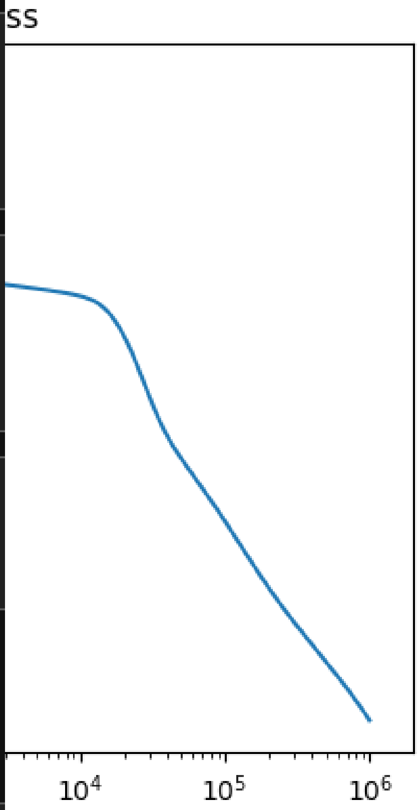
```
[5]: # Generate sinusoidal data
     timesteps = 100  # number of timesteps in the data
     x = np.linspace(0, 2 * np.pi, timesteps)
     y = np.sin(x)
```

```
[6]: # Reshape x and y for training
     x_train = x.reshape(-1, 1)
     y_train = y.reshape(-1, 1)
```

Build Our Model

```
[7]: # Initialize and train the model
     model = SimpleDNN(hidden_size)
```

```
[10]: model.train(x_train, y_train, epochs=100000, learning_rate=0.001)
```



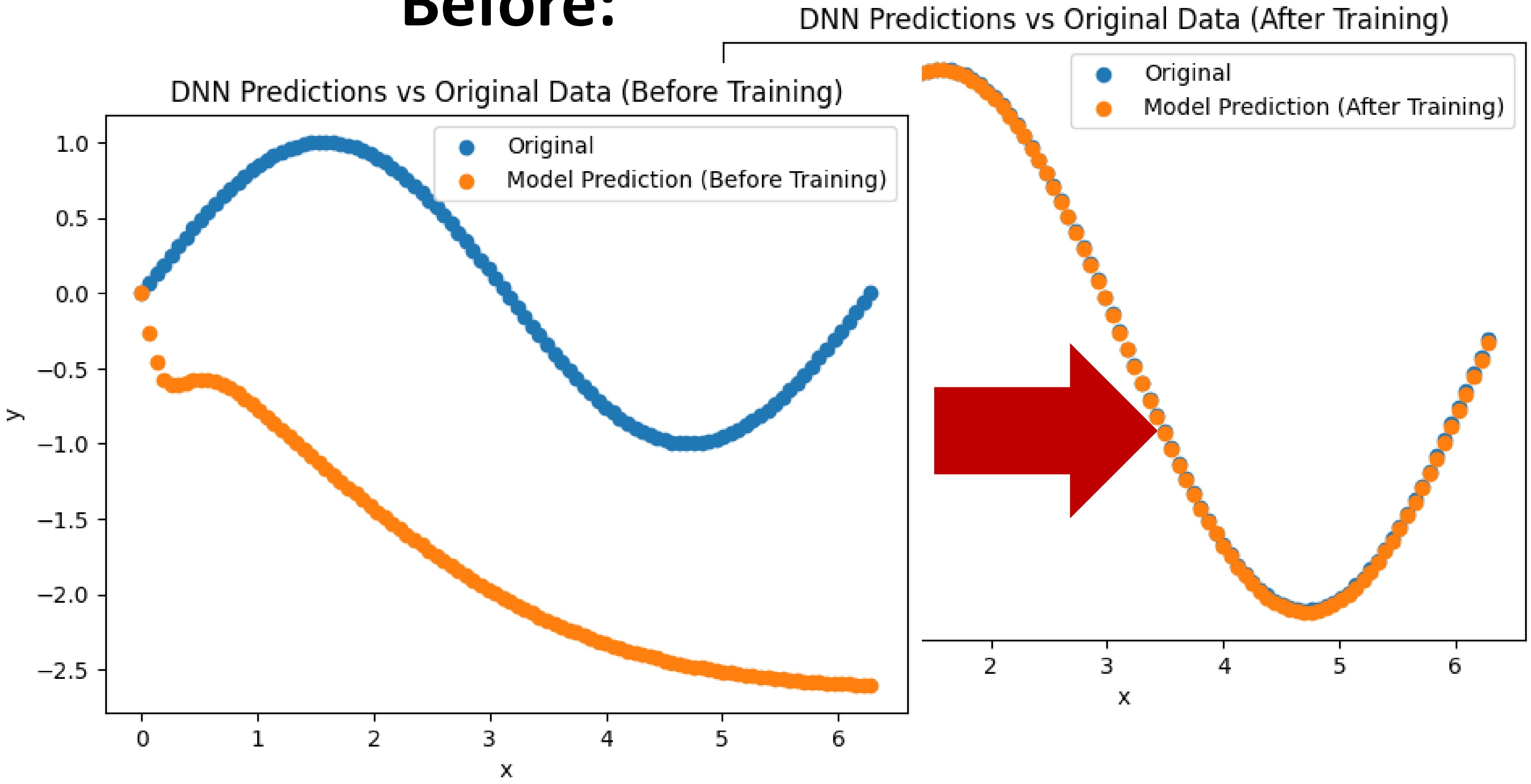
ML model design – Training (by example)

- Our (*average*) loss function is going down (huzzah!)
- This probably means it’s “*learning*” something.
- If it’s correctly trained our model can predict the value of y for a given input of x, i.e. it can mimic a sinusoid function

ML model design – Training (by example)

Before:

After:

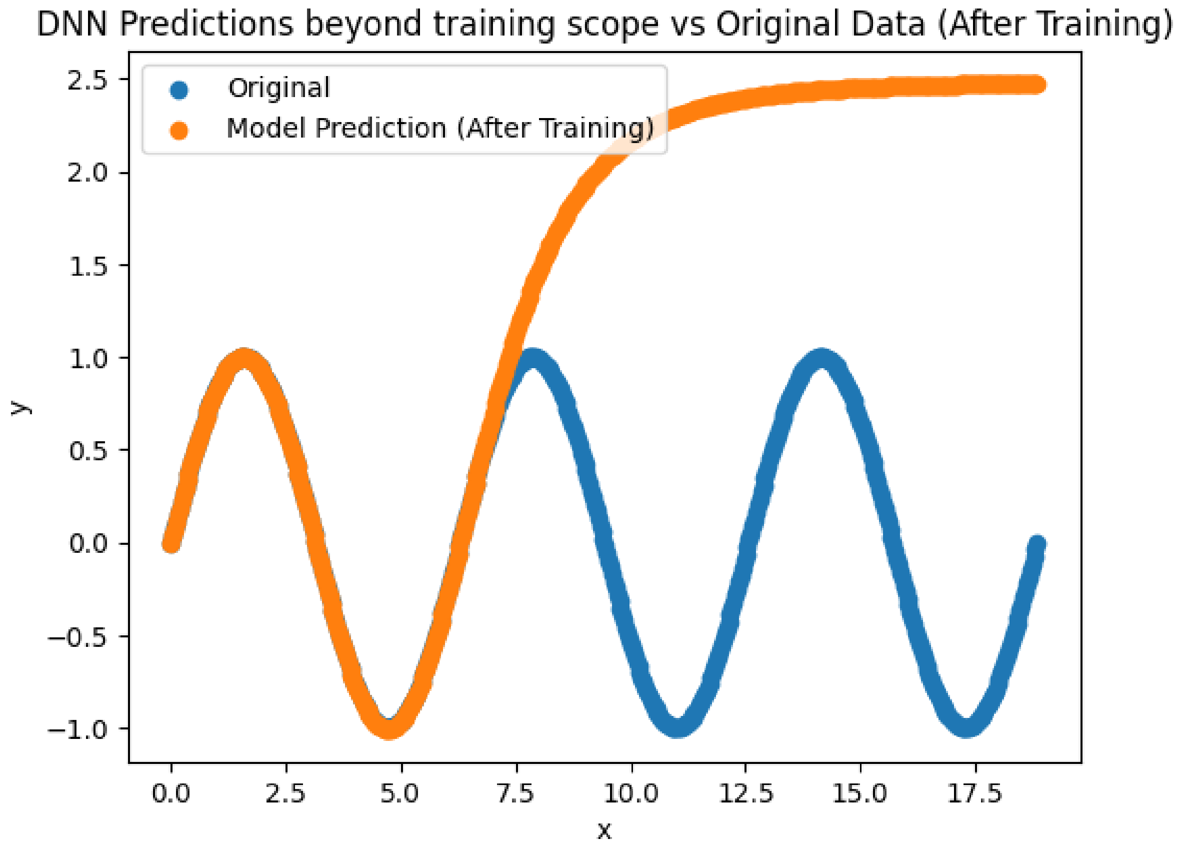


DNN Predictions vs Original Data (Before Training)

DNN Predictions vs Original Data (After Training)

ML model design – Training (by example)

- OK, fantastic we've trained our model, now what happens if we predict beyond what it's previously seen:



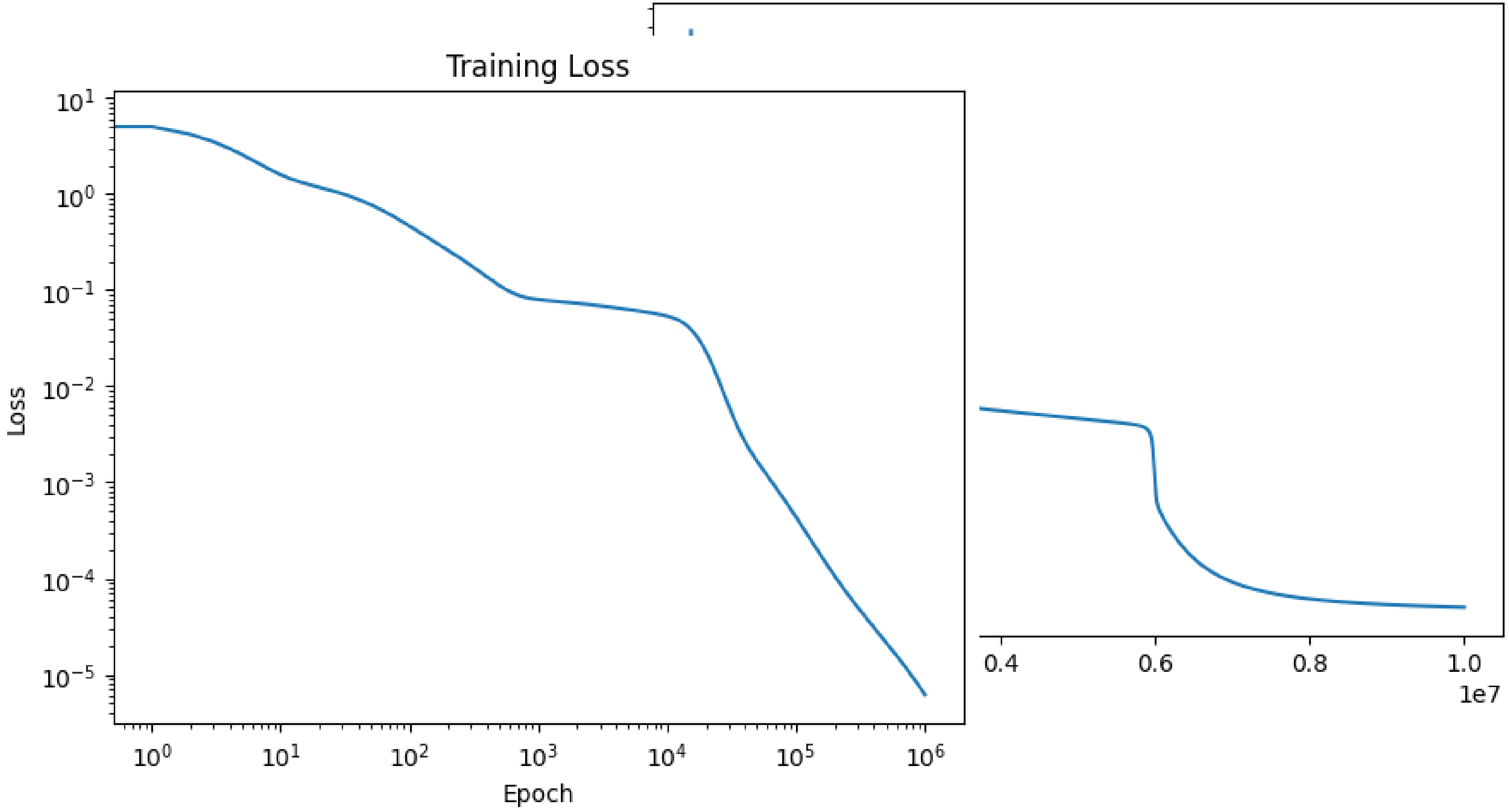
ML DNN model design

So why does it fail?

- DNN have no concept of time/positional-ordering, they memorize, and they spit the input back out.
- This is fine if we want to up/down-sample data between different input/output networks/models.
- Is powerful if we just want a quick/easy model to do something such as **anomaly detection** or **classification**.
- This is basically useless by itself for analysing a piece of music or extracting emotion from a sentence.

ML model design

When do we stop training?



ANN So Far – A Recap

This Photo by Unknown Author is licensed under CC BY-SA-NC

- We’ve covered that an **ANN** composed of neurons and it is trained to describe a given dataset and perform a task.
- Neurons contain **Weights**, **Biases** and **Activation** functions
- To make this useful we want to know:

- What is inside a model?

(How are neurons connected?)

- How do we evaluate this?

(How is data read into the model?)

- How do train/fit/tune this?

(How is a model trained on data?)

Shallow Neural Network – A Special Case

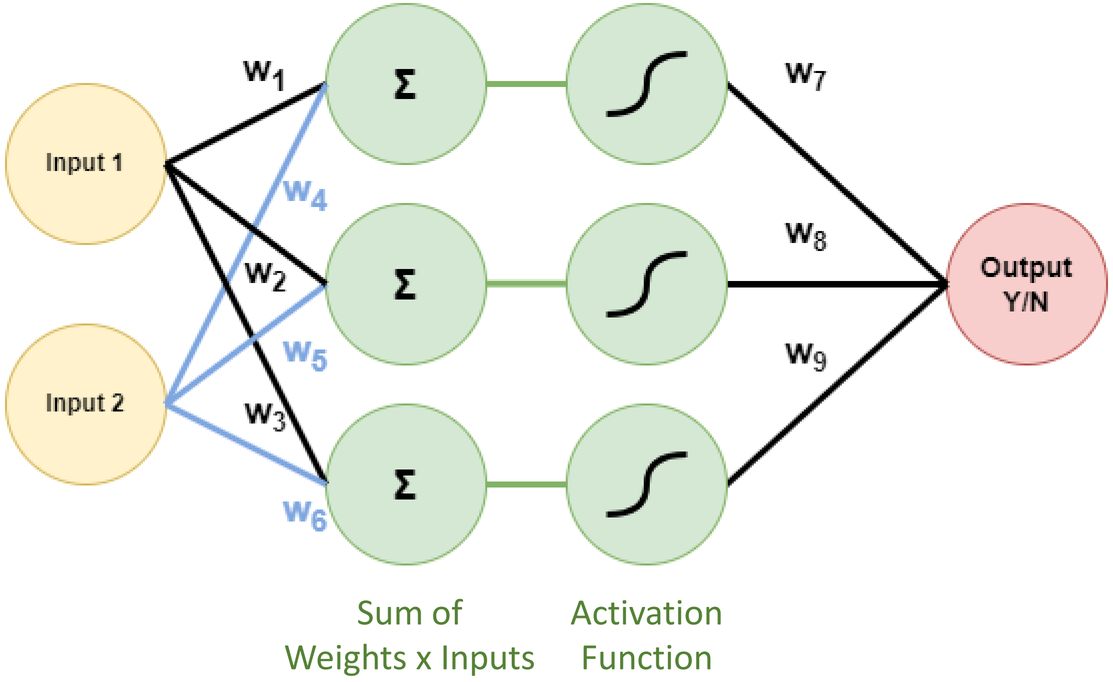
A Shallow Neural Network is an ANN which has a ***single*** hidden layer.

This is typically taken to be a “special case” which isn’t always practical to use.

Example of a SNN with 1 hidden layer.

In this case we have ***9 free weights***.
(***9 model parameters***)

This means to train this model on data we would need to find the combination of **9 weights** which best match the model to the training data.



Shallow Neural Network – A Special Case

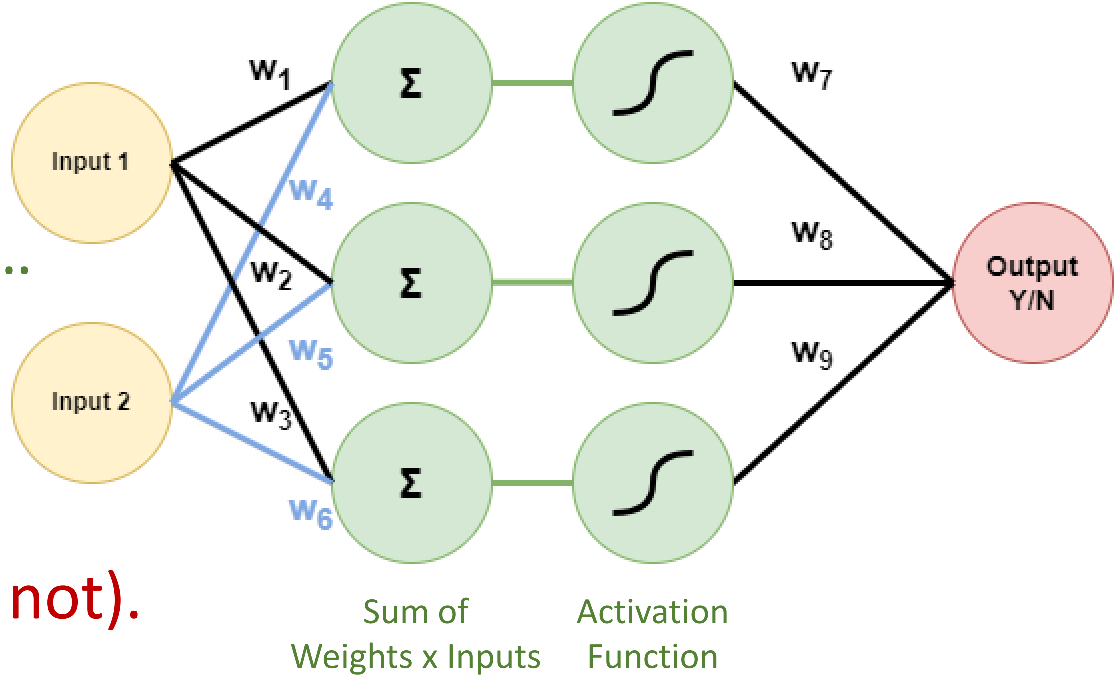
To *evaluate* a **SNN/DNN** model, we need our data to be represented as *numerical input*.

The numerical input is connected to (read by) the first layer in the model.

The output from this layer is passed to the next.
And the next. And the next. And the next …

Finally, the ***logits*** are passed to the output layer which gives some output.

Output is then evaluated to be correct (or not).



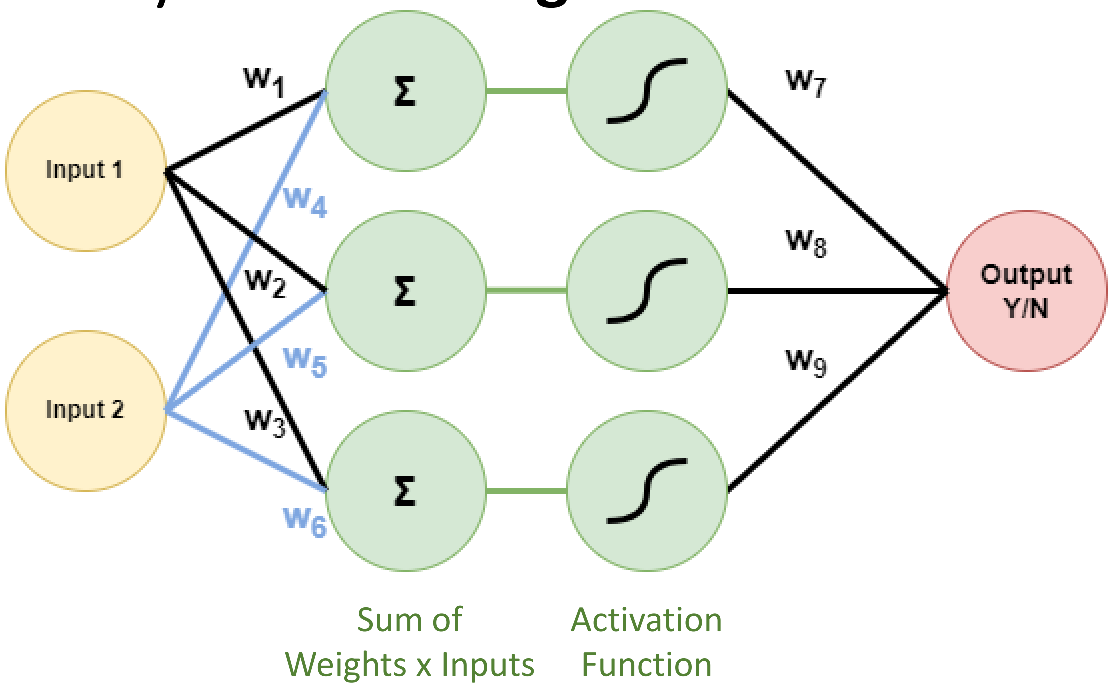
Shallow Neural Network – A Special Case

To ***train*** this model we need to calculate the full matrix of 9x9 partial derivatives to know what step size to take.

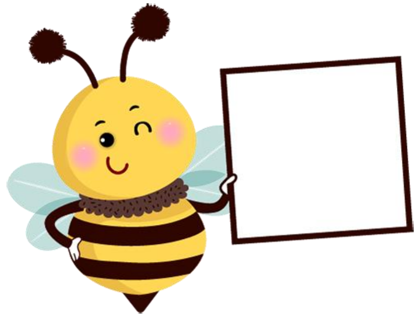
The size of the step we then take is controlled by the "**learning rate**".

Calculating the various parameters in this ***Jacobian*** is expensive, but there are mathematical tricks to reduce the steps required.

These tricks mean this is referred to as an $\mathbf{O}\left(\boldsymbol{nlog}(\boldsymbol{n})\right)$ vs an $\boldsymbol{O}(\boldsymbol{n}^2)$ problem.



ANN – Activation Functions



- ***Activation Functions*** in ANN give the network the ability to handle *non-linear relationships*.

- The choice of an activator can have an impact on how quickly a function can be trained on a dataset.

- The choice of activator can also impact model stability during training.

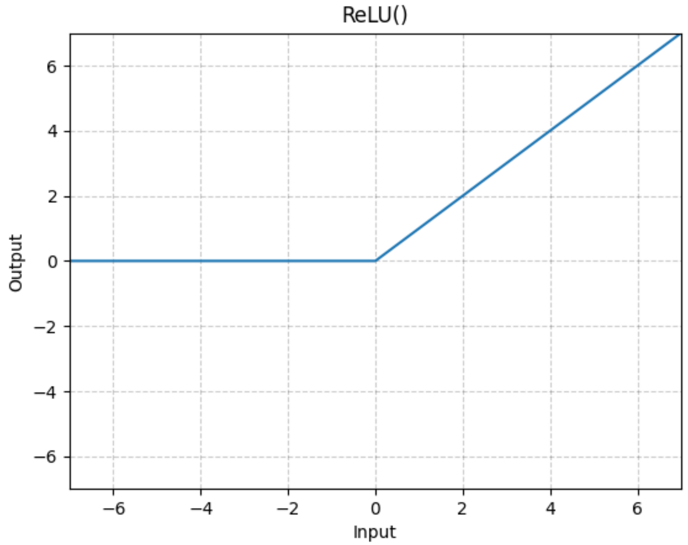
- Activators are also used to **normalize** or ***cap*** the distribution of model weights passed forward from each hidden layer.

ANN – Activators (Some Examples) ReLU

ReLU or **Re**ctified **L**inear **U**nit activator is probably the most common function used in building and training ANN.

It is fast, simple, has many advantages but also has issues.

$$\boldsymbol{ReLU}(x) = \max(0, x)$$



ReLU from PyTorch documentation

This activation function effectively turns on/off nodes within an ANN during training.

One of the biggest problems with **ReLU** is that the function is discontinuous. This can cause instabilities in training which can be difficult to understand/control/rectify.

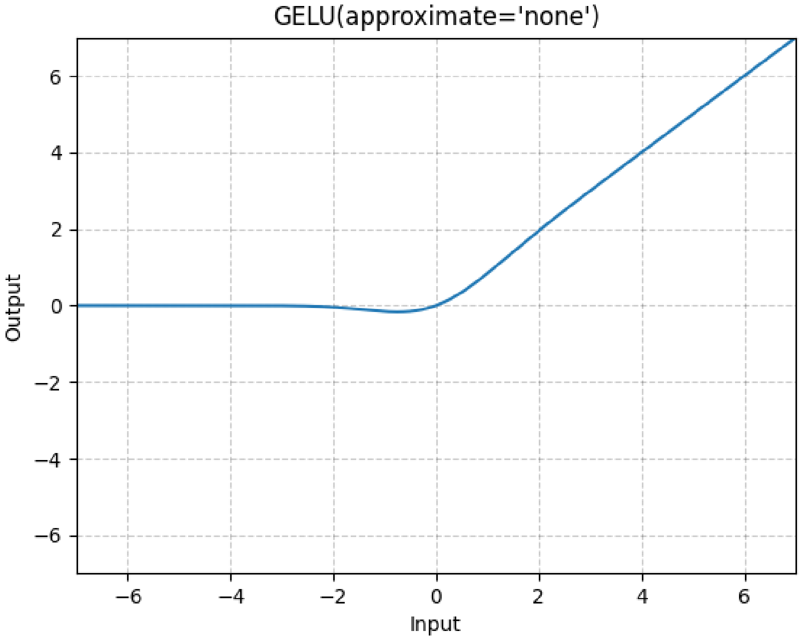
As used in ChatGPT2(!

ANN – Activators (Some Examples) GELU

GELU or **G**aussian **E**rror **L**inear **U**nit is a a choice of activation function which is gaining significant traction.
Can be viewed as a *'smoother* ***ReLU****'* function. This has the advantages of the **ReLU**, it is always continuous, but at the cost of being computationally more expensive.

$$\begin{aligned} GELU(x) &= x \times P(X \leq x) \\ &= x \times \Phi(x) \\ &\approx \frac{x}{2}\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \end{aligned}$$

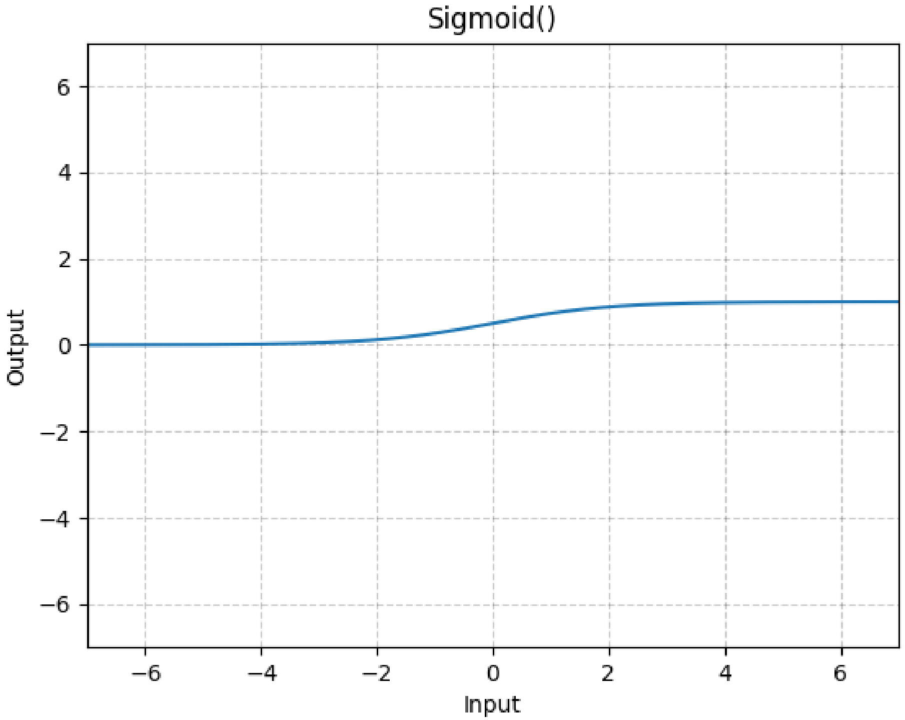
Where $\Phi(x)$ is the CDF for the Gaussian Distribution

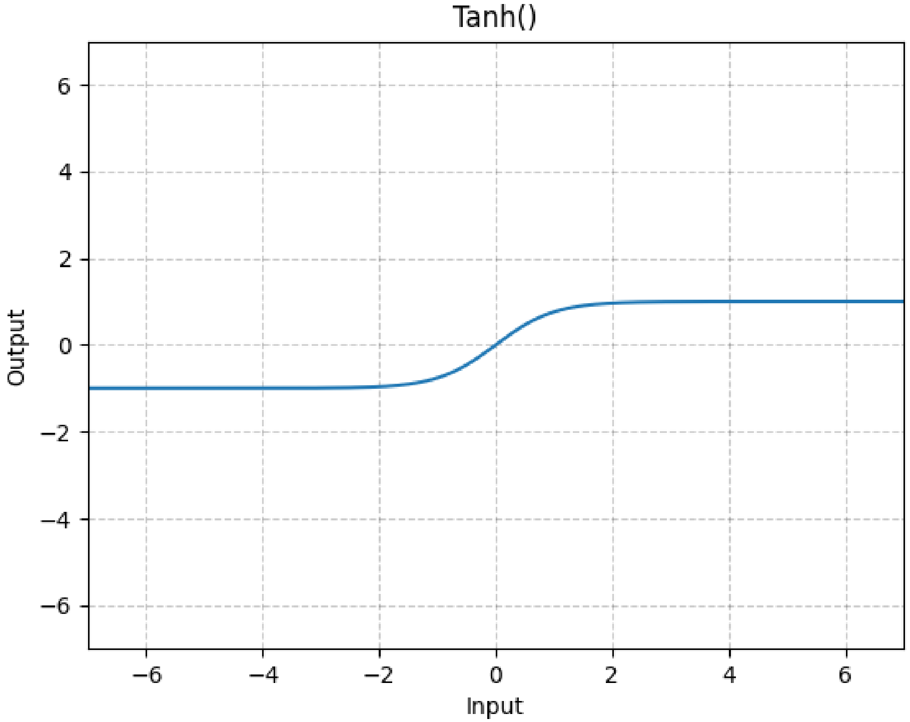


GELU from PyTorch documentation

ANN – Activators (Some Examples) sigmoid/tanh

Other commonly used activation functions.





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

Plots from PyTorch docs

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

DNN Classifier – Output Layer

- The final “step” in a **DNN** network is the **Output** layer.
  (Also sometimes known as the ***projection*** *layer)*

- This layer maps the ***raw-***hidden results from the network (***logits***) to the model **output**.

- If your model is a **classifier** the output may be a **category-id**.
- If your model is a **generator** the output may be an **image**.
- If your model is an **LLM** the output is **encoded text**.

DNN Classifier – Output Layer

- With **DNN** classifiers we want to be able to predict the class of some input data.

- Typically, this means that we need to identify the '**category**' of our input using our model.

- We now need to consider:

  a) What form should the output layer take?
  b) What should the loss function attempt to calculate?

DNN Classifier – Output Layer

- Output for a classifier needs to make a prediction of a category.
- To be consistent all predictions *need to sum to 100%*

- It’s possible to build an activation function which give a **one-hot result** when making predictions.

i.e. truth = $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, prediction = $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

DNN Classifier – Output Layer

Typically **output** layer looks similar to a hidden dense node layer.

The layer first includes a dense layer of ***n*** nodes. Here ***n*** is defined as the number of categories in the labels the model is trained on.

After this, the output from each node is normalized using an activator such as a Softmax function.

$$\text{Softmax}(z) = \sigma(\tilde{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \dots, K$$

DNN Classifier

- So, a simple DNN Classifier takes some inputs (of different types)

- It passes these inputs through some Neural Network

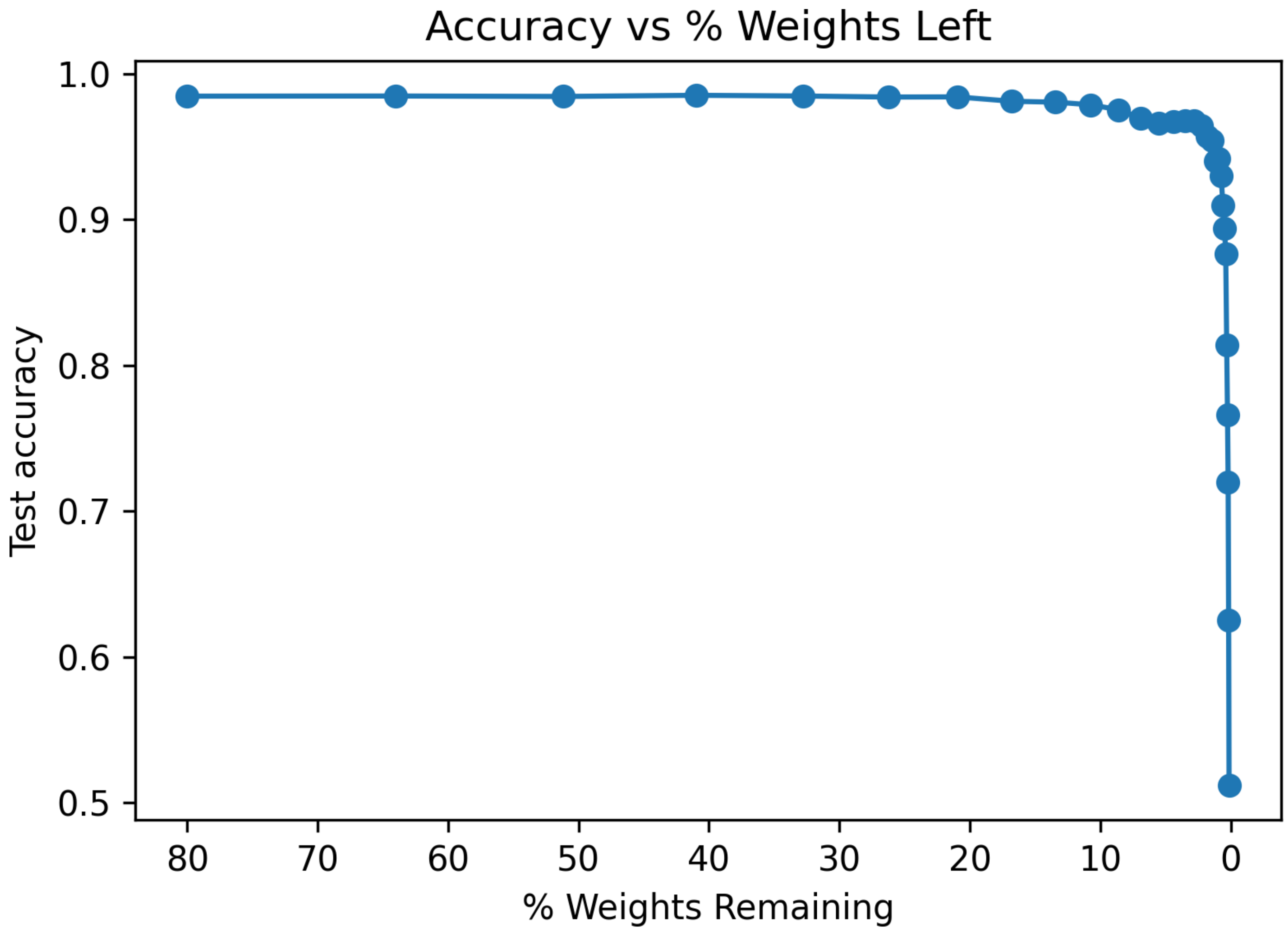
- The output from this network makes a guess at the classification of the input(s)

- By comparing to truth and using back-propagation we train our model to perform better.

Training – Afterwards

- After the model has finished training the weights have been modified and the resulting structure of the DNN is now a good approximate solution for the task that we’ve tasked it with.
- General wisdom is that a well-behaved large model has a distribution of weights centred at zero with a few weights having large values.
- If a weight is $\simeq 0$ it’s possible to remove or “trim” this weight an keep the model performance approximately the same.
- **In the case of a simple DNN classifier how much of the model can we trim and still have a useful model?**

Training – Afterwards



Training – Afterwards

- *These models are random in nature* and built to *approximately solve* complex statistical problems.

- Random means **instability/non-reproducibility**.

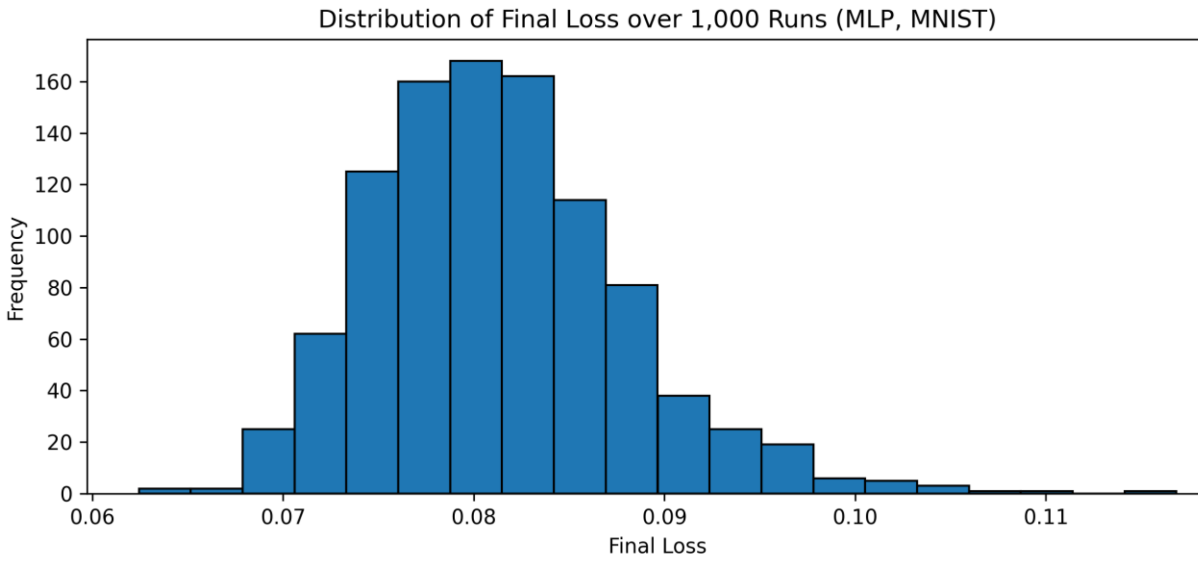
- How reliable is the solution from our model?

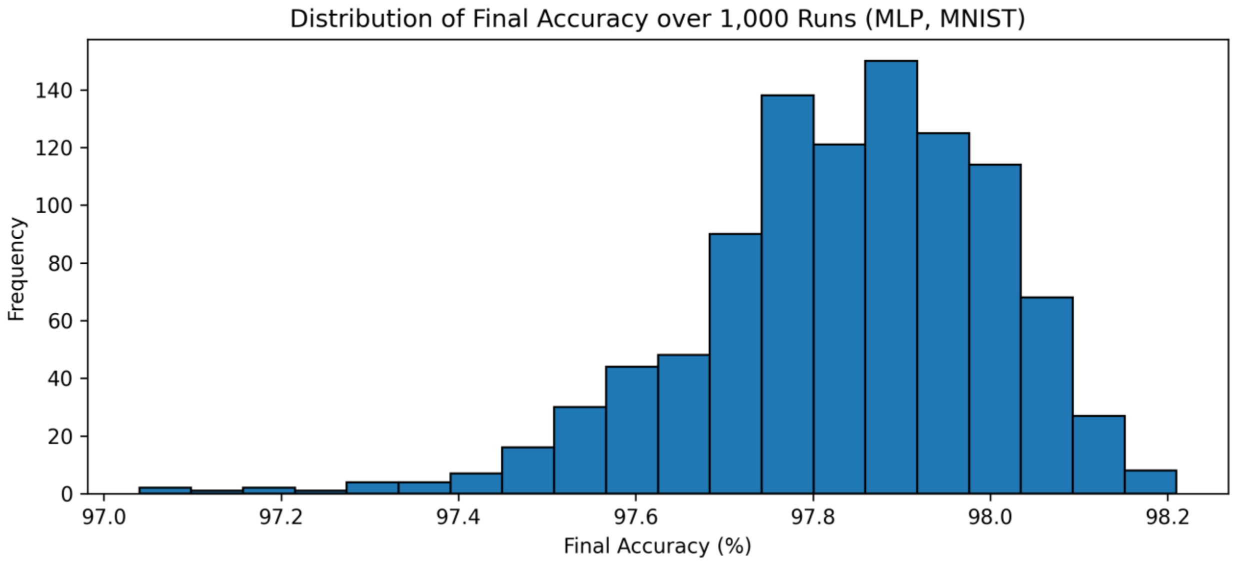
- **How lucky can you get?**

- **How un-lucky could you possibly be?**

Training – Afterwards

- Final model performance is a function of the input initialization weights.
- Given all we care about is model performance we’re free to pick the version which gives us the best outcome each time.


Distribution of Final Loss over 1,000 Runs (MLP, MNIST)


Distribution of Final Accuracy over 1,000 Runs (MLP, MNIST)

Training – Quality of a fit

- When training our model, we want to make sure that we are avoiding over-training on features only present in the training data.

- We also want to not waste computing resources over-training a model which has effectively converged.

- The best way of doing this is to split our data into 3 datasets as with Decision Trees:

  Training, **Validation** & **Testing**

- **More on this next week**

Training – Model Design

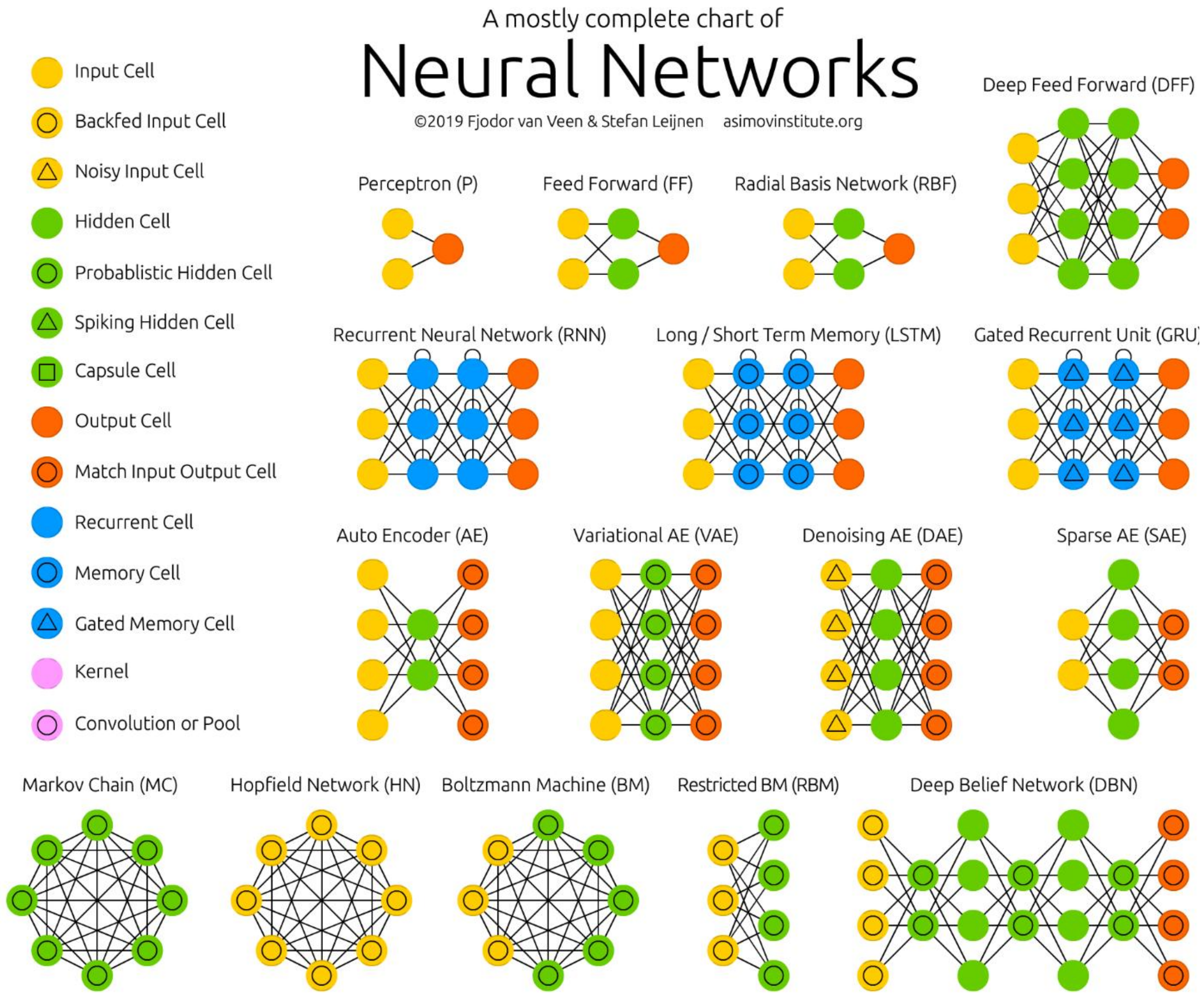
In the workshop following this lecture we will be building models which are designed to work with hundreds of input parameters.

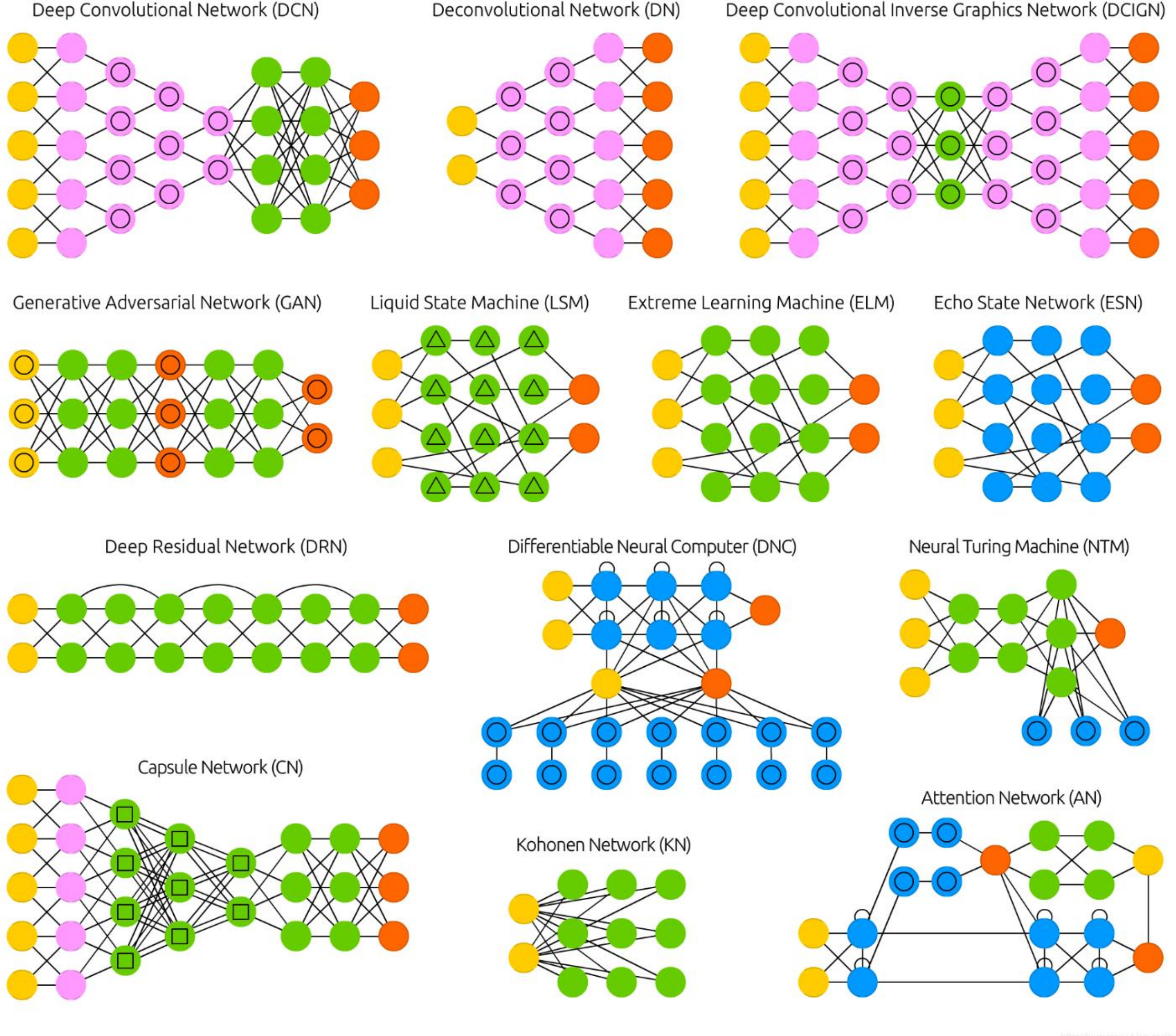
In turn the **DNN** designed to work with this uses hundreds of thousands of free weights.

If it is possible to reduce the number of free weights by being clever in our nodes, then the model is mathematically simpler to work with.

This typically means that we need to do something more complex in our node than just ‘summing’ over the products…

ANN – Zoo




Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Differentiable Neural Computer (DNC)
Neural Turing Machine (NTM)
Capsule Network (CN)
Kohonen Network (KN)
Attention Network (AN)

© Fjodor van Veen, Asimov Institute, 2016

Neural Networks – Deep NN

- In most cases we will be using multiple “hidden layers” within a NN.

- A network with more *<u>layers</u>* is defined as being deeper (***depth***)
- A network with more *<u>nodes</u>* per-layer is described as wider (***width***)

- ANN come in many shapes/sizes

- TensorFlow playground is an excellent online resource for playing with different CNN models and how they train to a given dataset: https://playground.tensorflow.org/

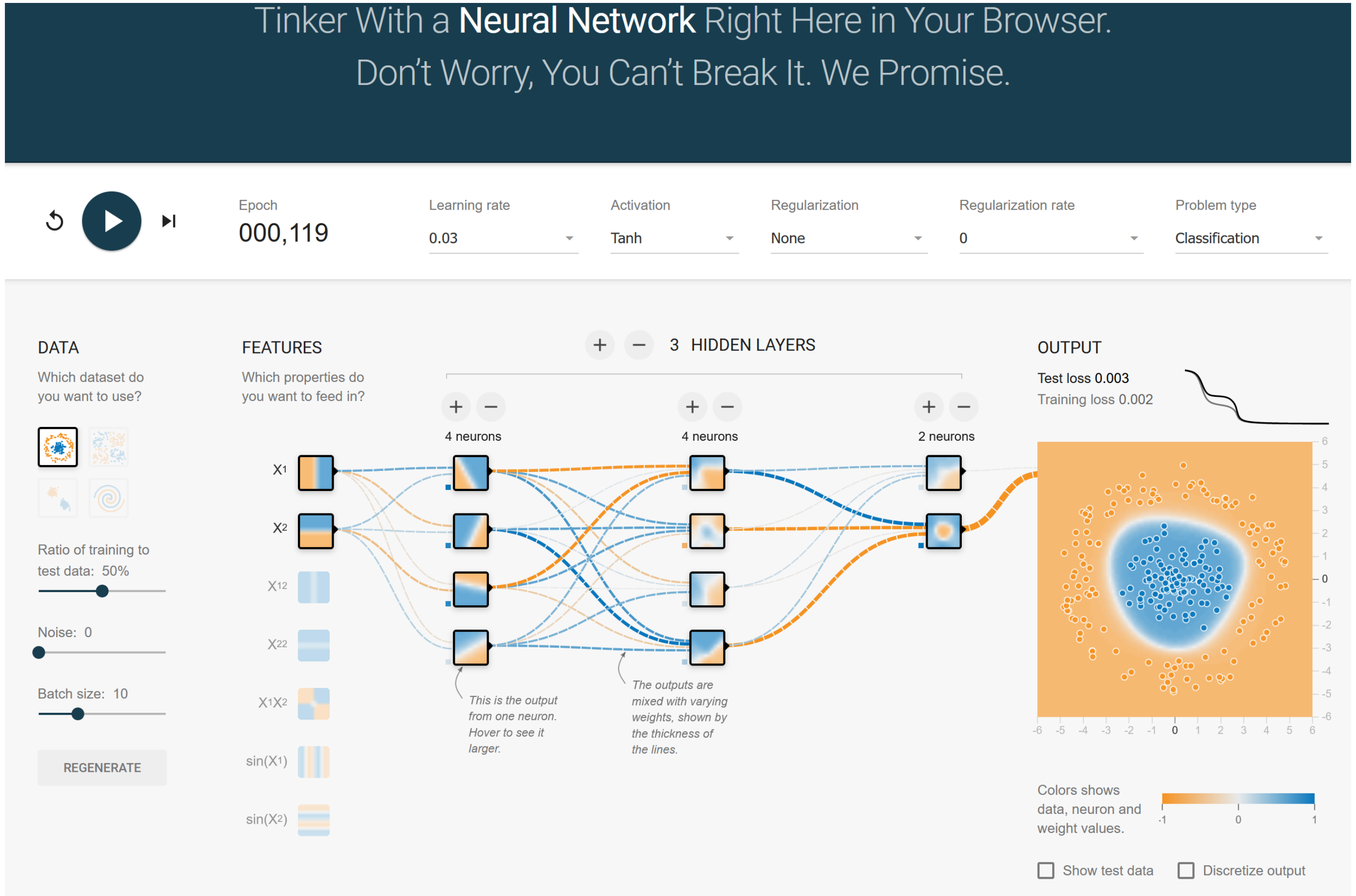
TensorFlow-Playground – (An Aside)

- Can be useful to ‘*get a feel*’ for how changing individual parts of the model can impact training.

- I claim nodes on this page are best thought of as **CNN** neurons. (more on these in my next lecture)

- Weight scales are a good comparator to ‘*how much of feature X*’ is in the dataset?

TensorFlow-Playground – (An aside)



Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 000,119

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

3 HIDDEN LAYERS

This is the output from one neuron. Hover to see it larger.

The outputs are mixed with varying weights, shown by the thickness of the lines.

OUTPUT

Test loss 0.003

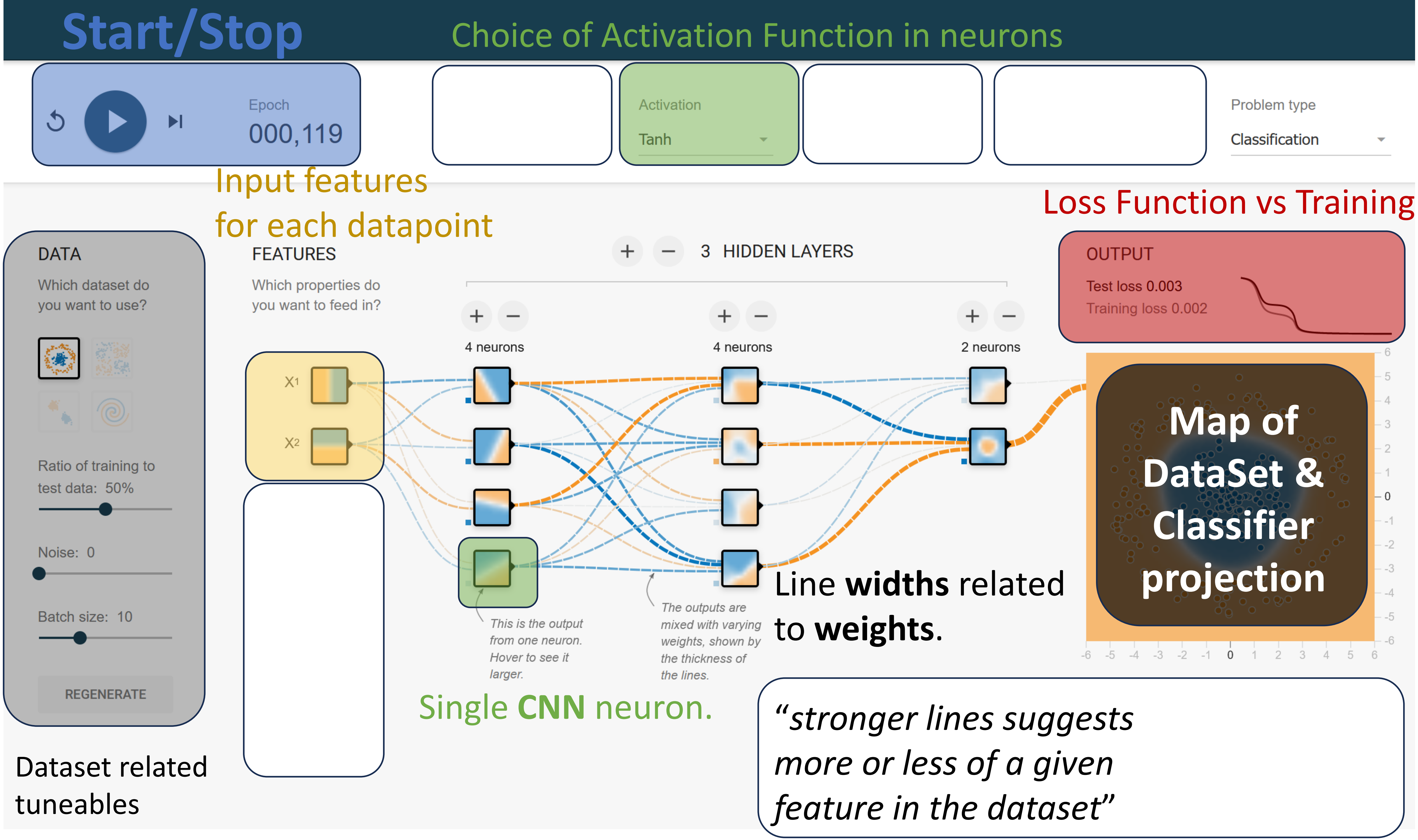
Training loss 0.002

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

Start/Stop

Choice of Activation Function in neurons



Epoch 000,119

Activation: Tanh

Problem type: Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

3 HIDDEN LAYERS

4 neurons

4 neurons

2 neurons

This is the output from one neuron. Hover to see it larger.

The outputs are mixed with varying weights, shown by the thickness of the lines.

OUTPUT

Test loss 0.003

Training loss 0.002

Input features for each datapoint

Loss Function vs Training

Map of DataSet & Classifier projection

Line **widths** related to **weights**.

Single **CNN** neuron.

Dataset related tuneables

“stronger lines suggests more or less of a given feature in the dataset”

Next Lecture

- My next lecture will introduce a few different topics:

1. Evaluating how well has a fit converged?
2. When should we decide to stop training?
3. How can I optimize/improve training?
4. Using more advanced nodes in a PyTorch model(*graph*)
5. Constructing more advanced models in PyTorch

6. Constructing an AutoEncoder for anomaly detection

Today’s Workshop

PyTorch docs if you need them:
https://pytorch.org/docs/stable/nn.html

- Today's workshop starts a bit slower;

1. **Start by build a Deep Neural Network or Multi-Layer Perceptron using just NumPY.**

   Getting hands dirty with Python code.

2. **Second, building the same model again using PyTorch**

   Getting familiar with PyTorch’s API

3. **Building a Classifier using PyTorch**

   Using PyTorch to build a classifier for real data

4. **Extending our DNN beyond it’s training window**