

# Today's Resources

## Access the Notebook:

- ▶ Local Clone (**Recommended**)

```
git clone https://github.com/wtsi-hpag/xAIWorkshop
```

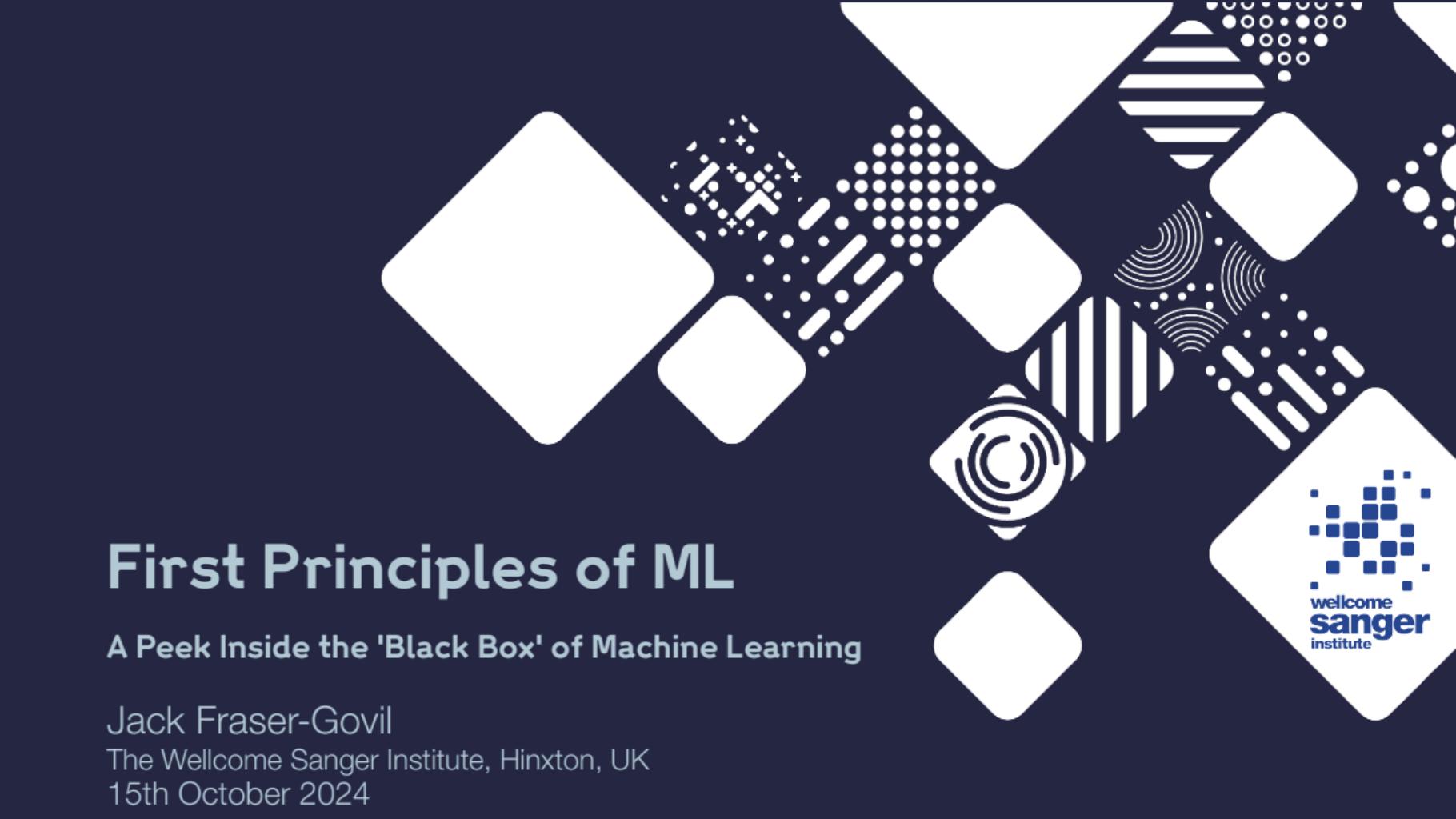
- ▶ Online Hosting (**Slow**) Visit

```
https://githubtocolab.com/wtsi-hpag/xAIWorkshop
```

## Interactive Questions

(Keep open in a browser tab!)

Join [menti.com](https://menti.com) with code 2834 2684



# First Principles of ML

A Peek Inside the 'Black Box' of Machine Learning

Jack Fraser-Govil

The Wellcome Sanger Institute, Hinxton, UK  
15th October 2024



Seem Familiar?

# Seem Familiar?

```
>> import ml_toolbox
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)  
  
Epoch 0: 0% | [0/1000 10it/s, loss = 10.4]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)  
  
Epoch 100: 10%|xx | [100/1000 10it/s, loss = 5.5]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

```
Epoch 200: 20%|xxxxx | [200/1000 12it/s, loss = 3.4]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)  
  
Epoch 300: 30%|xxxxxxxx | [300/1000 9it/s, loss = 3.1]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

```
Epoch 400: 40%|xxxxxx | [400/1000 16it/s, loss = 2.9]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

```
Epoch 500: 50%|xxxxxxxxxx | [500/1000 12it/s, loss = 1.4]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

```
Epoch 700: 70%|xxxxxxxxxxxxx | [700/1000 5it/s, loss = 1.39]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

```
Epoch 900: 90%|xxxxxxxxxxxxxxxxx | [900/1000 8it/s, loss = 1.385]
```

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

Epoch 1000: 100%|xxxxxxxxxxxxxxxxx| [final loss = 1.382]

# Seem Familiar?

```
>> import ml_toolbox  
>> data,labels = load_my_data()  
>> model = ml_toolbox.make_model(layers=5,func=sigmoid)  
>> model.Train(data,labels,epochs=1000)
```

```
Epoch 1000: 100%|xxxxxxxxxxxxxxxxx| [final loss = 1.382]
```

```
>> model.Test()
```

# Today's Agenda

The aim for today is:

# Today's Agenda

The aim for today is:

- ▶ Classic Perceptron

# Today's Agenda

The aim for today is:

- ▶ Classic Perceptron
- ▶ Feedforward Networks

# Today's Agenda

The aim for today is:

- ▶ Classic Perceptron
- ▶ Feedforward Networks
- ▶ Non-Linearity

# Today's Agenda

The aim for today is:

- ▶ Classic Perceptron
- ▶ Feedforward Networks
- ▶ Non-Linearity
- ▶ Optimisation & Backpropagation

# Today's Agenda

The aim for today is:

- ▶ Classic Perceptron
- ▶ Feedforward Networks
- ▶ Non-Linearity
- ▶ Optimisation & Backpropagation

As we progress you will slowly build up your own ML toolkit, built entirely from scratch!

# A Warning

There will be equations.

# A Warning

There will be equations.  
You will need to know what they mean!

# A Warning

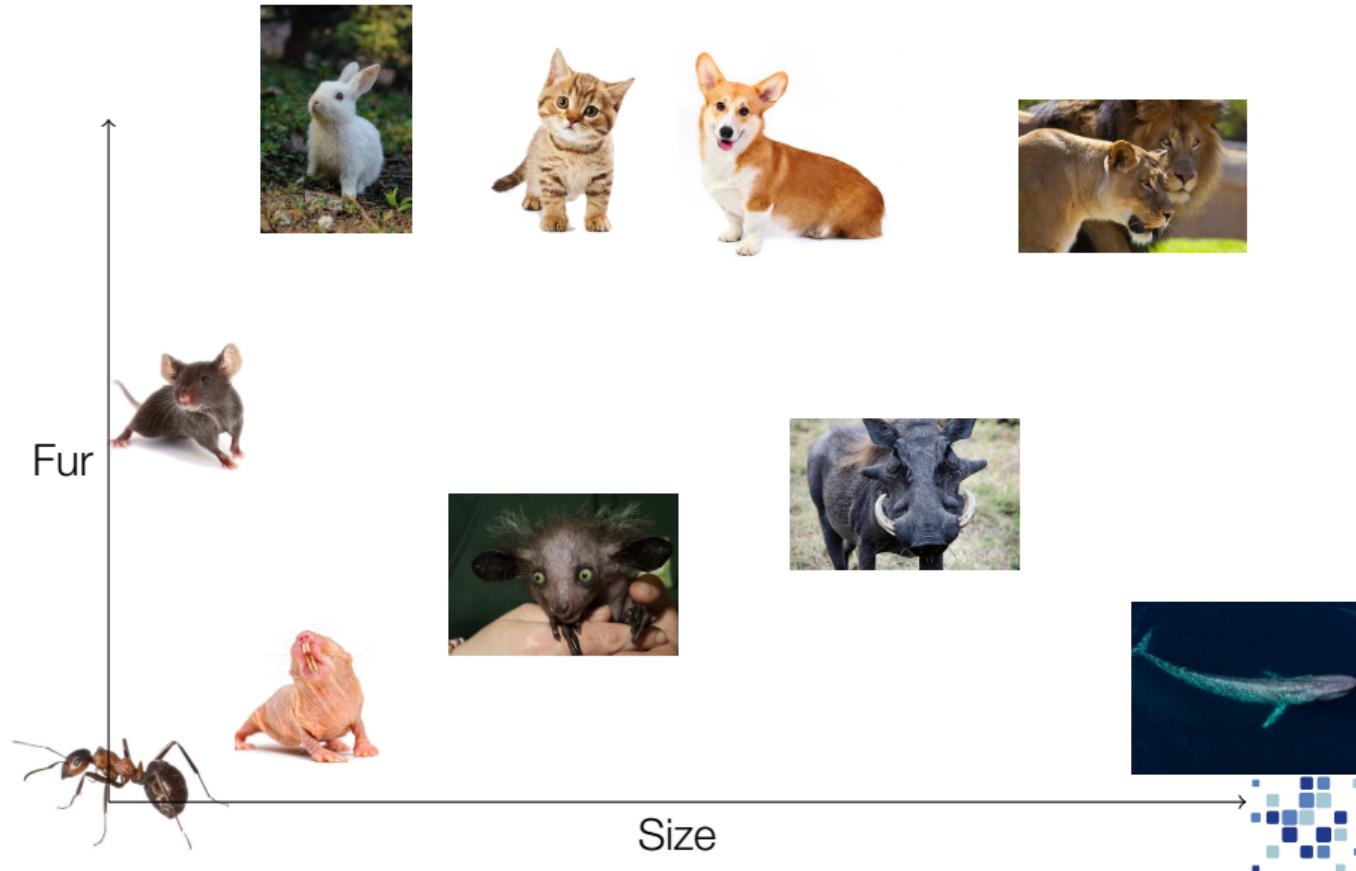
There will be equations.  
You will need to know what they mean!

*Please, please, please, ask if you want clarification on the underlying mathematics  
and theory! That's why you're here today!*

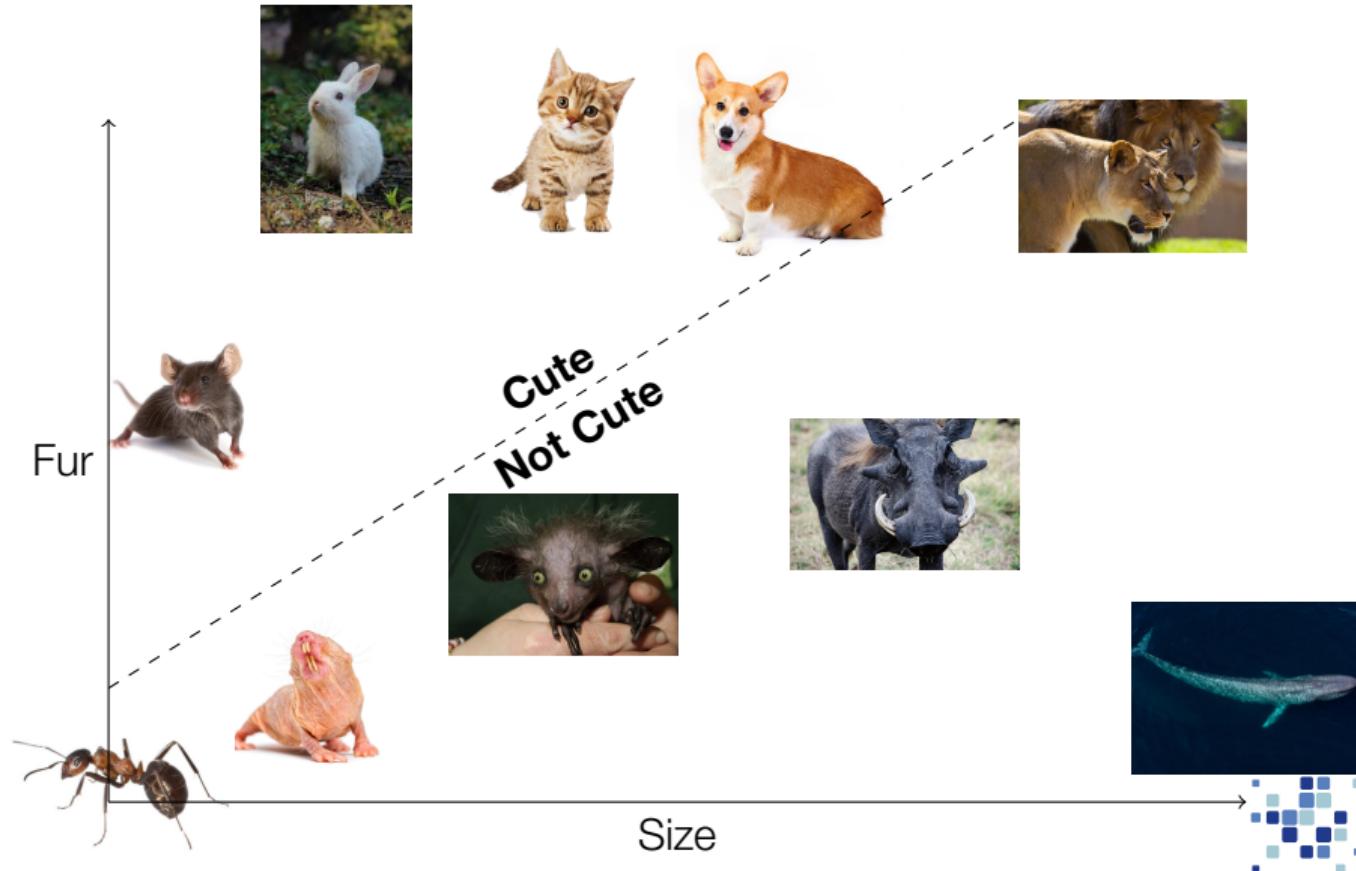
# Part 1

# The Perceptron

# Basic Decision Making: Defining Cuteness

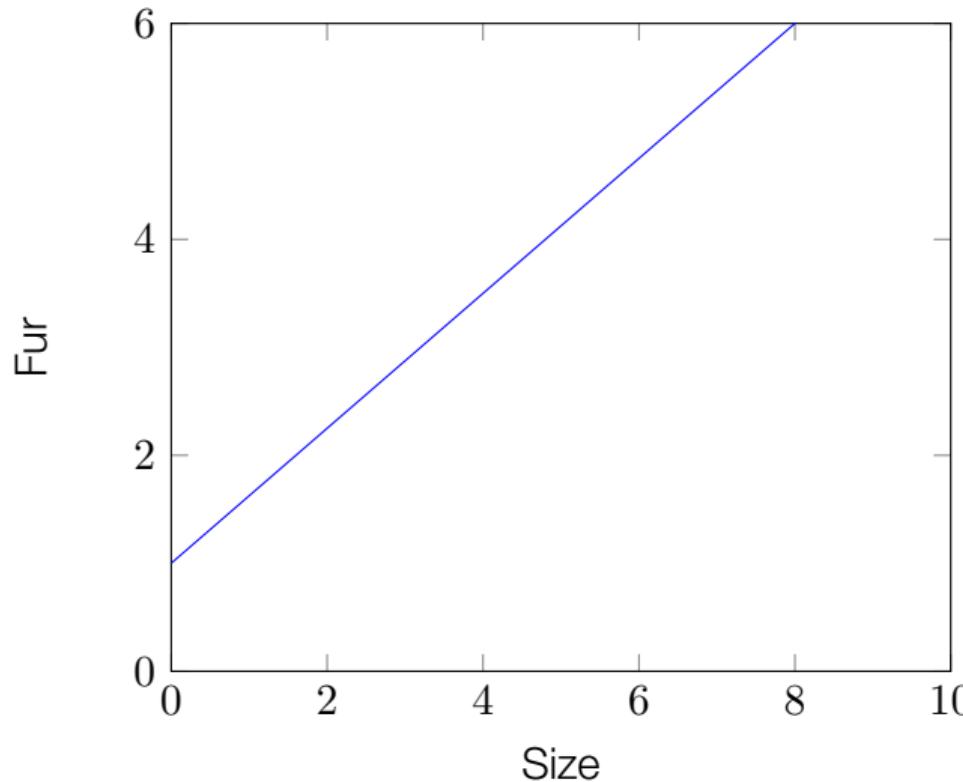


# Basic Decision Making: Defining Cuteness



# Splitting The Plane

In order to split the plane into two parts, we merely need to define a *line*.



# Splitting the plane

# Splitting the plane

**Question:** If we have  $N$  dimensions ( $N = 2$ ), how many parameters do we need to define a line?

# Splitting the plane

**Question:** If we have  $N$  dimensions ( $N = 2$ ), how many parameters do we need to define a line?

**Answer:** The answer is  $N$  - in 2 dimensions, this is friendly  $y = mx + c \rightarrow (m, c)$

# The Perceptron

The Perceptron classifier algorithm is:

$$P(\mathbf{x}) = \begin{cases} 1 & \text{if } b + \mathbf{w} \cdot \mathbf{x}^1 > 0 \\ 0 & \text{else} \end{cases} \quad (1)$$

---

<sup>1</sup>The dot/inner product is covered in Section 3.1 in the notes

# The Perceptron

The Perceptron classifier algorithm is:

$$P(\mathbf{x}) = \begin{cases} 1 & \text{if } \tilde{\mathbf{x}} \cdot \mathbf{w}^1 > 0 \\ 0 & \text{else} \end{cases} \quad (1)$$

Where

$$\tilde{\mathbf{x}} = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}$$

---

<sup>1</sup>The dot/inner product is covered in Section 3.1 in the notes

# The Perceptron

The Perceptron classifier algorithm is:

$$P(\mathbf{x}) = \begin{cases} 1 & \text{if } \tilde{\mathbf{x}} \cdot \mathbf{w}^1 > 0 \\ 0 & \text{else} \end{cases} \quad (1)$$

Where

$$\tilde{\mathbf{x}} = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}$$

$\mathbf{w}$  are the **weights**.

---

<sup>1</sup>The dot/inner product is covered in Section 3.1 in the notes

# The Perceptron: Dimensions

**Question:** What is the dimensionality of the Perceptron? Does this conflict with our earlier statements?

# The Perceptron: Dimensions

**Question:** What is the dimensionality of the Perceptron? Does this conflict with our earlier statements?

**Hint:** It has nothing to do with the ‘bias’!

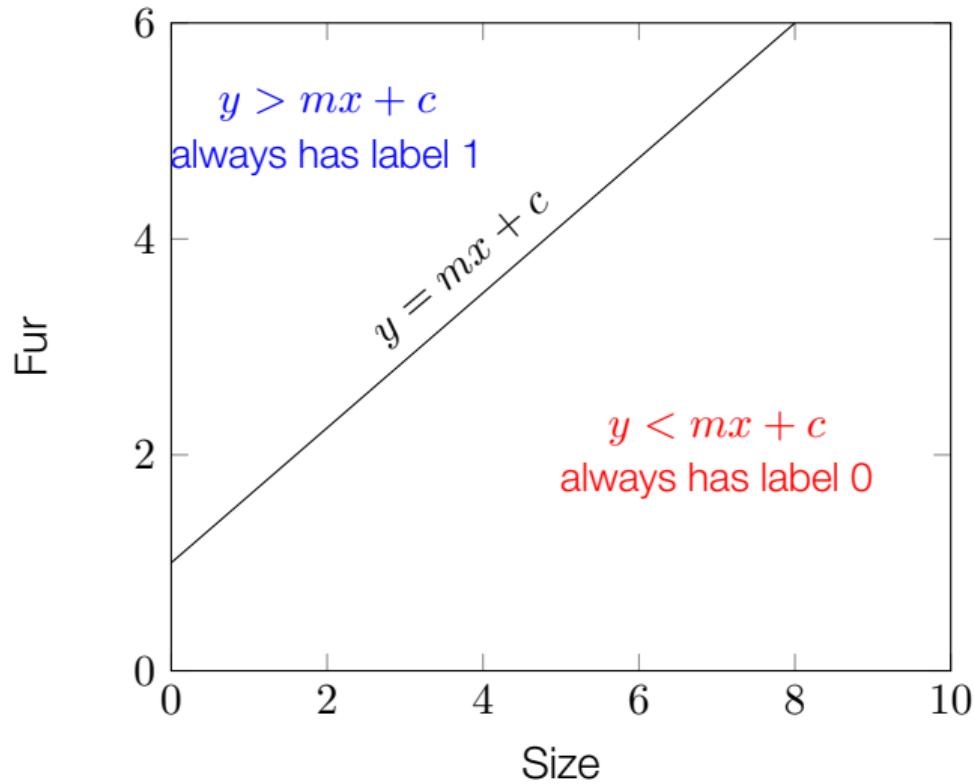
# The Perceptron: Dimensions

**Question:** What is the dimensionality of the Perceptron? Does this conflict with our earlier statements?

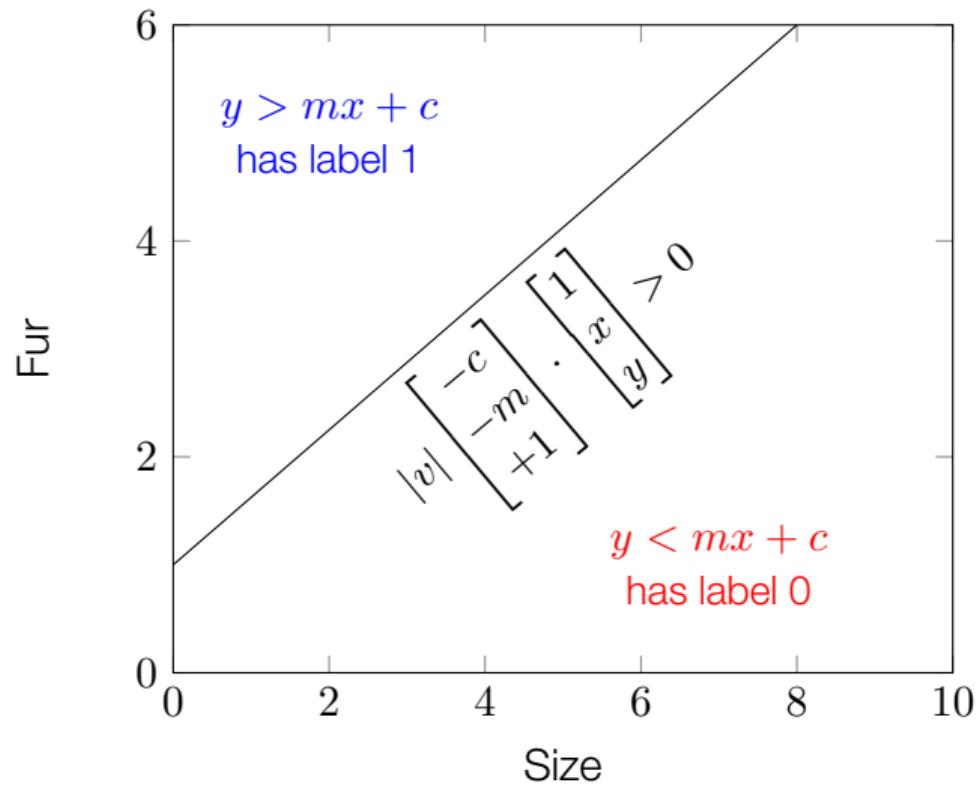
**Hint:** It has nothing to do with the ‘bias’!

**Answer:** We need  $N + 1$  dimensions.  $N$  dimensions define a line - but need an additional dimension to add *directionality*

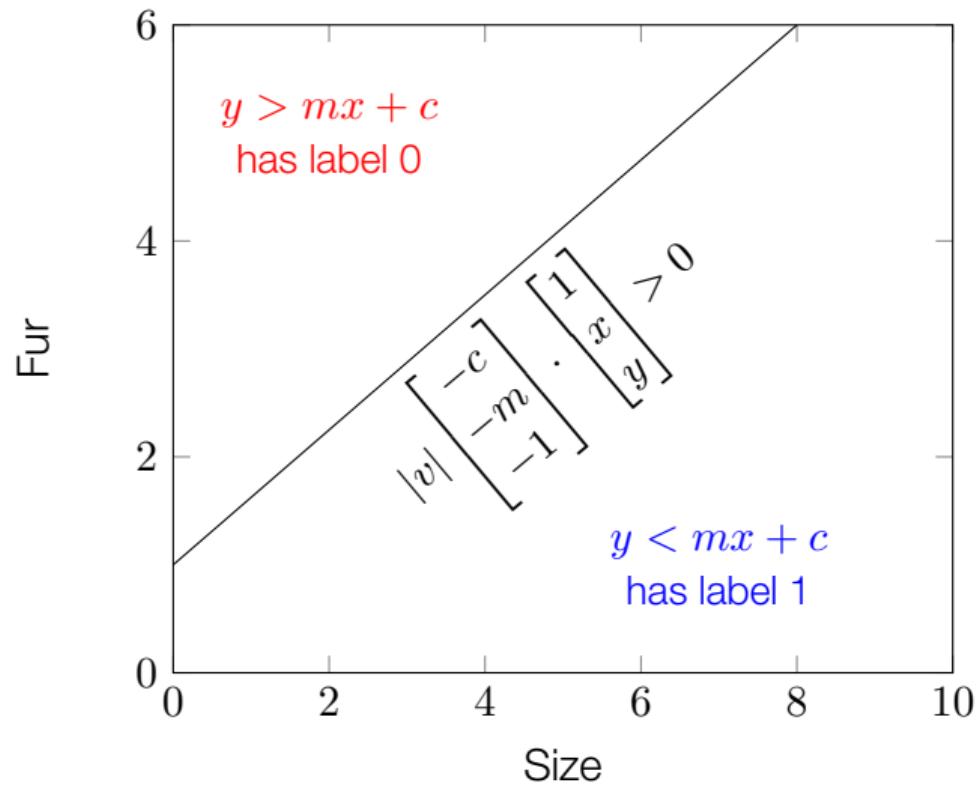
# The Perceptron: Dimensions



# The Perceptron: Dimensions



# The Perceptron: Dimensions



# Exercise 1: Perceptron Classifier

**Task 1:** Write a perceptron classifier for the data in `cuteness.dat`

**Guidance:** Class structure & predetermined weights are provided in the notebook.  
Ignore `Train()` for now

**Question:** The provided weights classify only one animal correctly. Which is it?

# Training A Perceptron

Training a perceptron is easy. Loop over the data, make a prediction ( $P$ ), compare it to the truth ( $T$ ) and if it is wrong:

# Training A Perceptron

Training a perceptron is easy. Loop over the data, make a prediction ( $P$ ), compare it to the truth ( $T$ ) and if it is wrong:

$$\mathbf{w} \rightarrow \mathbf{w} + r \times (T - P) \tilde{\mathbf{x}} \quad (2)$$

# Training A Perceptron

Training a perceptron is easy. Loop over the data, make a prediction ( $P$ ), compare it to the truth ( $T$ ) and if it is wrong:

$$\mathbf{w} \rightarrow \mathbf{w} + r \times (T - P) \tilde{\mathbf{x}} \quad (2)$$

This works because it always makes  $\mathbf{w} \cdot \tilde{\mathbf{x}}$  move closer towards zero (and hence to the tipping point of altering its decision).

# Exercise 2: Train Your Perceptron

**Task 2:** Write the Train() method for your Perceptron, then apply to cuteness data.

# Exercise 2: Train Your Perceptron

**Task 2:** Write the Train() method for your Perceptron, then apply to cuteness data.

**Question:** How long does it take to train to 100% accuracy?

# Exercise 2: Train Your Perceptron

**Task 2:** Write the Train() method for your Perceptron, then apply to cuteness data.

**Question:** How long does it take to train to 100% accuracy?

Give your answers on [menti.com](https://menti.com) with code 2834 2684

# Exercise 3: The Limits of Linearity

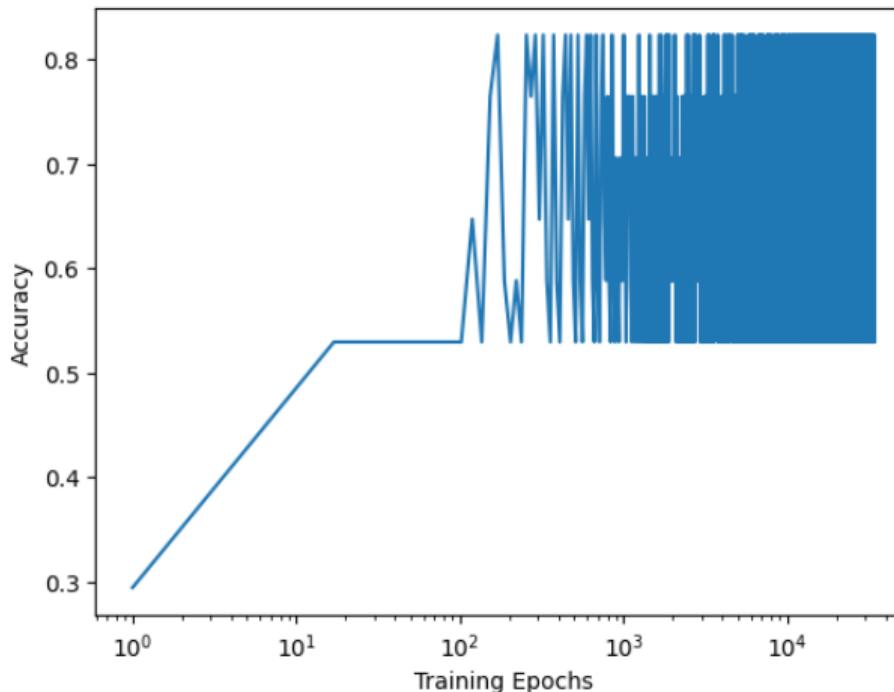
**Task 3:** Apply your method to the data in `cuteness_augmented.dat`, which adds the Axolotl, Otter, Capybara, Quokka, Slug, Tasmanian Devil and Giant Panda.

**Question:** What has gone wrong? Who are the troublemakers?

# Exercise 3: The Limits of Linearity

**Task 3:** Apply your method to the data in `cuteness_augmented.dat`, which adds the Axolotl, Otter, Capybara, Quokka, Slug, Tasmanian Devil and Giant Panda.

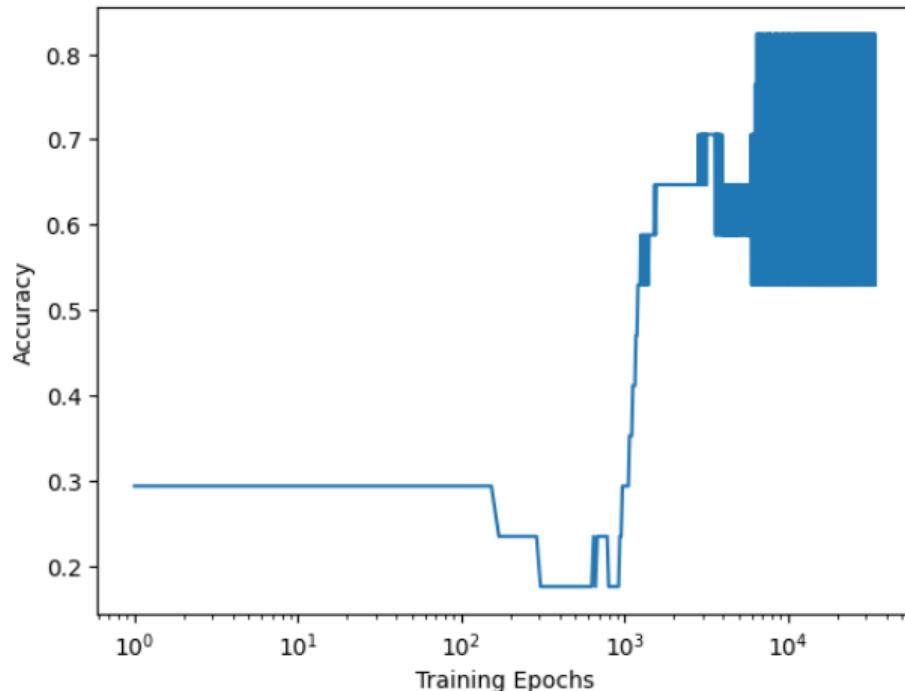
**Question:** What has gone wrong? Who are the troublemakers?



# Exercise 3: The Limits of Linearity

**Task 3:** Apply your method to the data in `cuteness_augmented.dat`, which adds the Axolotl, Otter, Capybara, Quokka, Slug, Tasmanian Devil and Giant Panda.

**Question:** What has gone wrong? Who are the troublemakers?



# The Nonlinear Perceptron

Eminently possible to have a non-linear perceptron.

# The Nonlinear Perceptron

Eminently possible to have a non-linear perceptron.

Pass  $\mathbf{x}$  through a non-linear transform to make it:

$$\mathbf{x}' = \begin{pmatrix} 1 \\ x \\ y \\ x^2 \\ \sin(x) \\ x^2 \exp(y) \\ \cos(\sin(\exp(\sin(\log(x^2 + 9xy)))))) \\ \vdots \end{pmatrix} \quad (3)$$

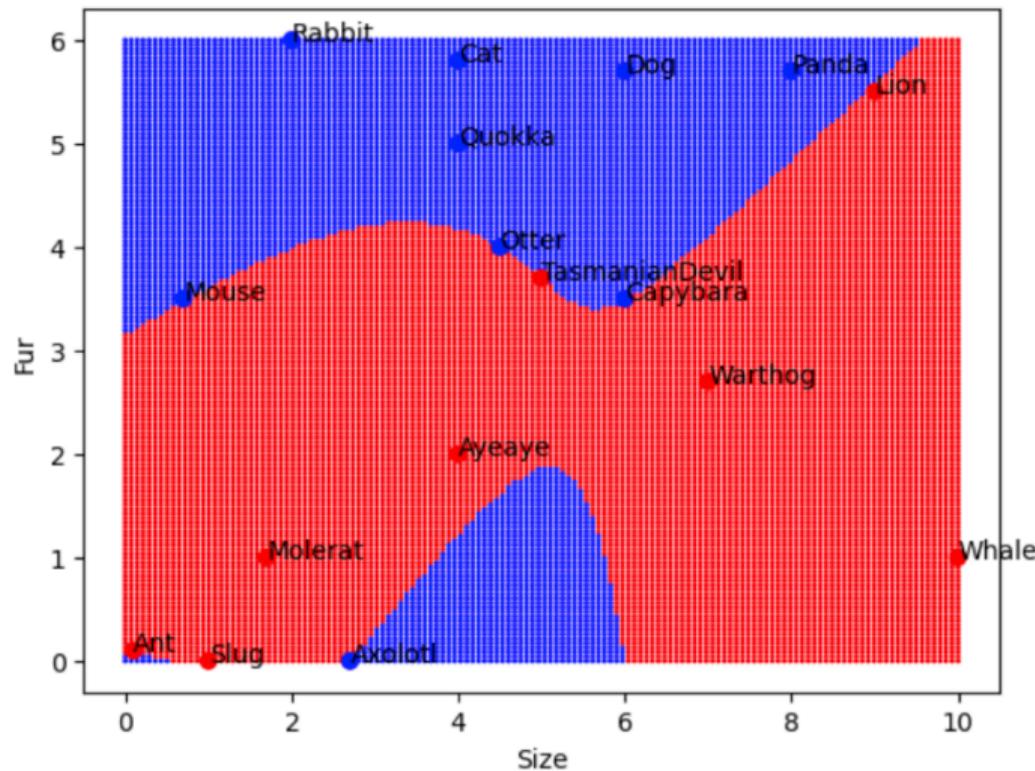
# The Nonlinear Perceptron

Here is the results of a perceptron where:

$$\mathbf{x}'(x, y) = \begin{pmatrix} 1 & x & y & x^2 & xy & y^2 & x^3 & x^2y & xy^2 & y^3 & x^4 & x^3y & x^2y^2 & xy^3 & y^4 \end{pmatrix}^\top \quad (4)$$

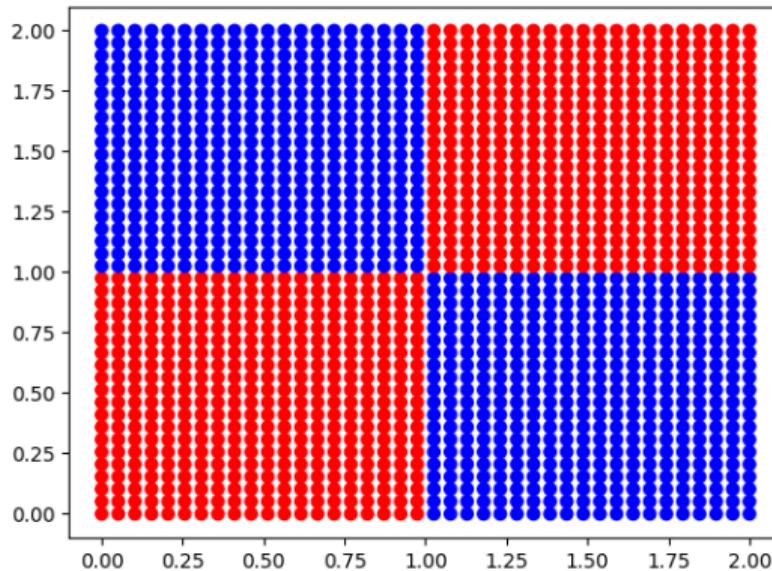
*Bonus Question: Why did I choose this?*

# The Nonlinear Perceptron



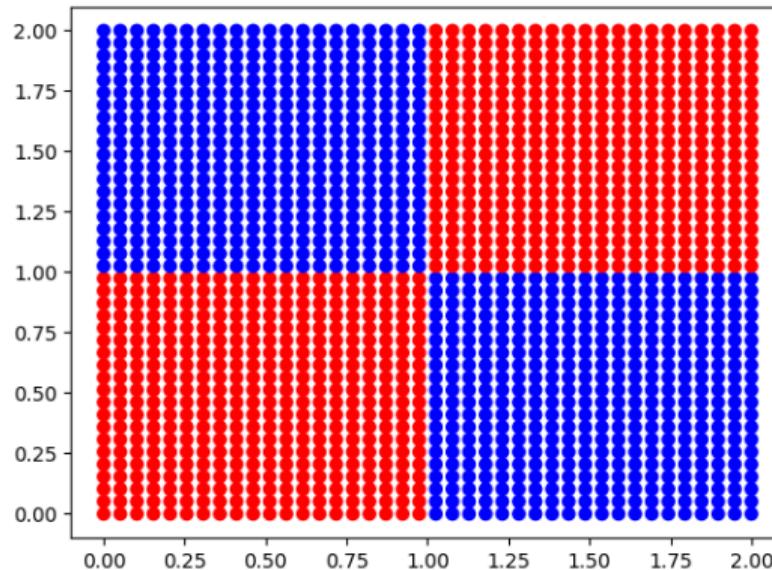
# The Limits of NonLinearity

Target

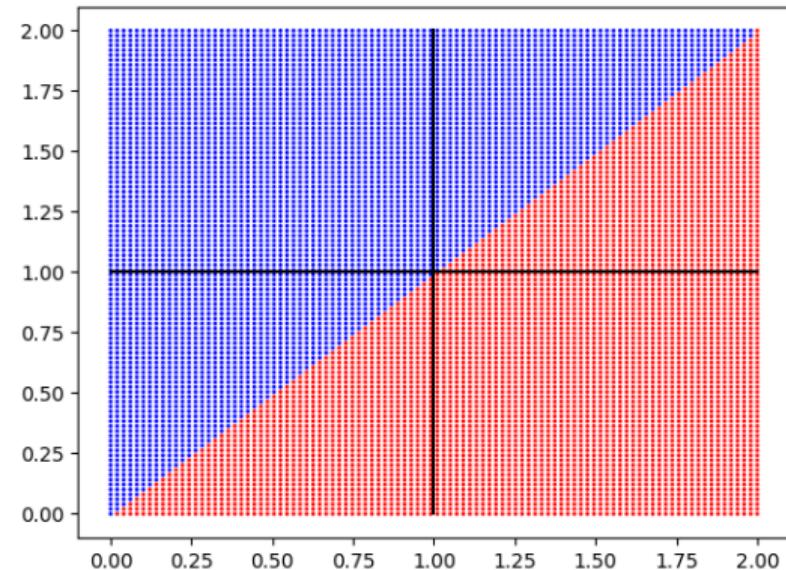


# The Limits of NonLinearity

Target

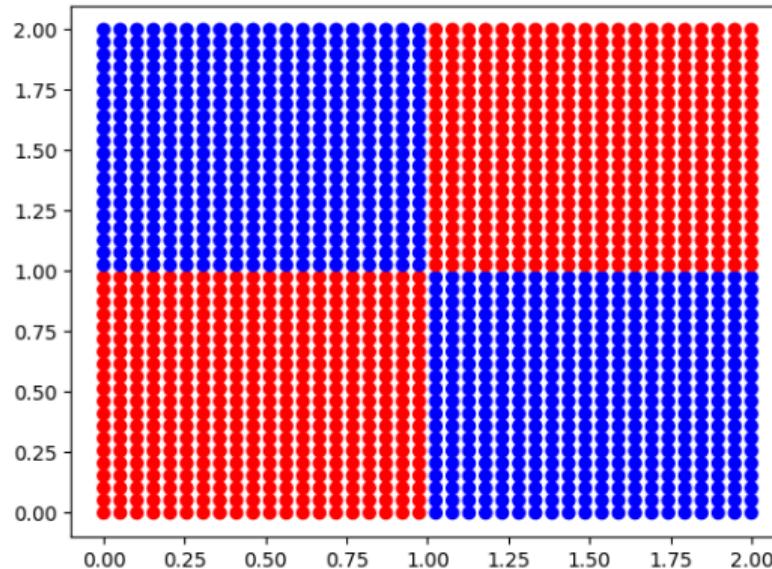


Reality (1st Order - 3 parameters)

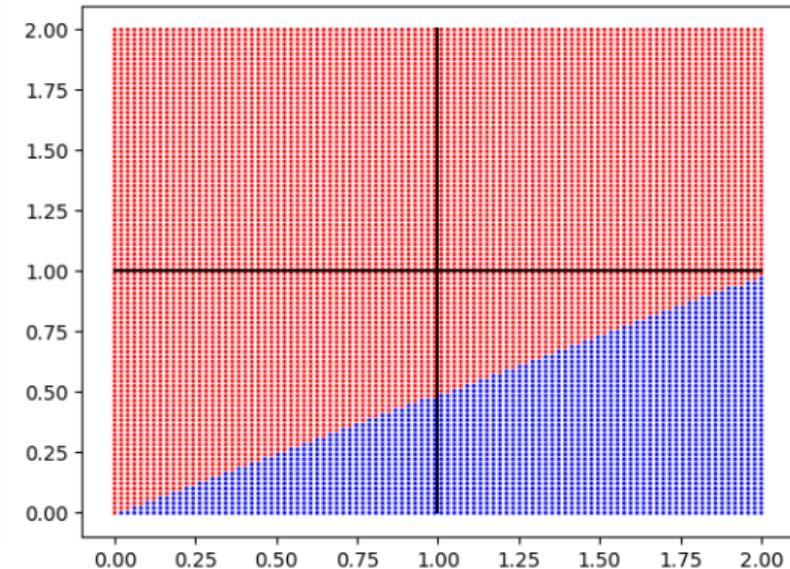


# The Limits of NonLinearity

Target

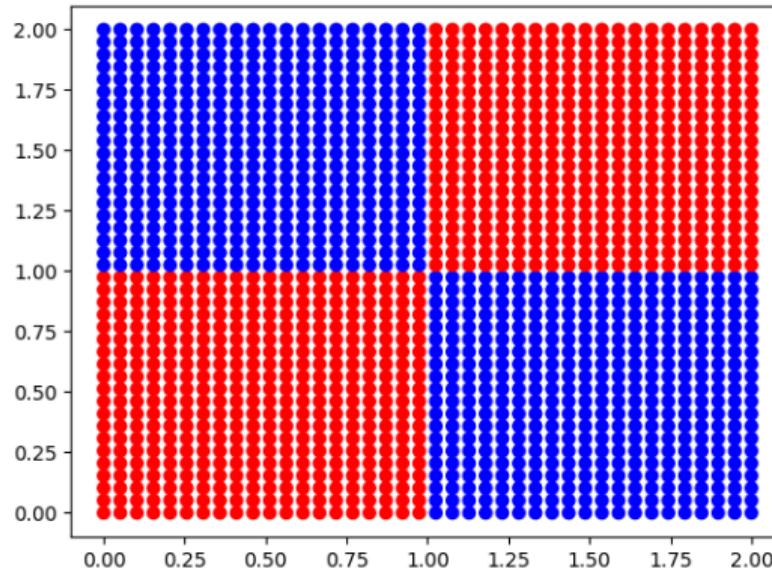


Reality (2nd Order - 6 parameters)

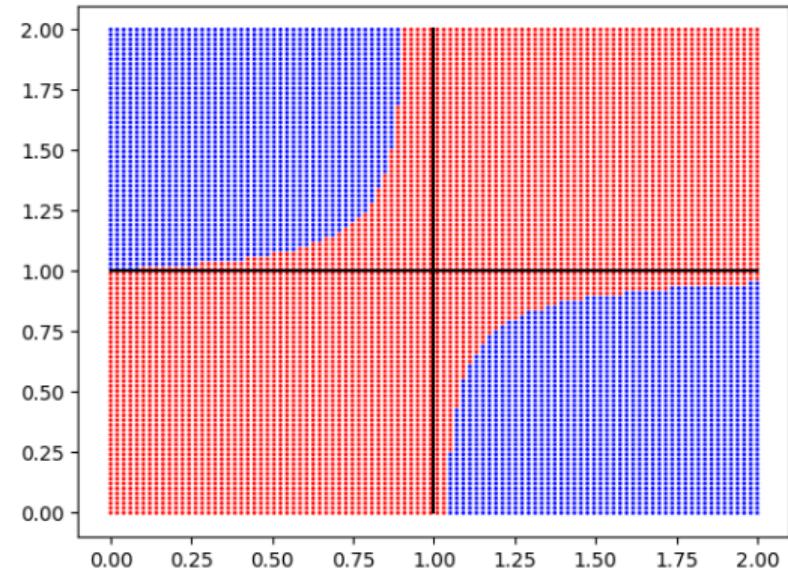


# The Limits of NonLinearity

Target

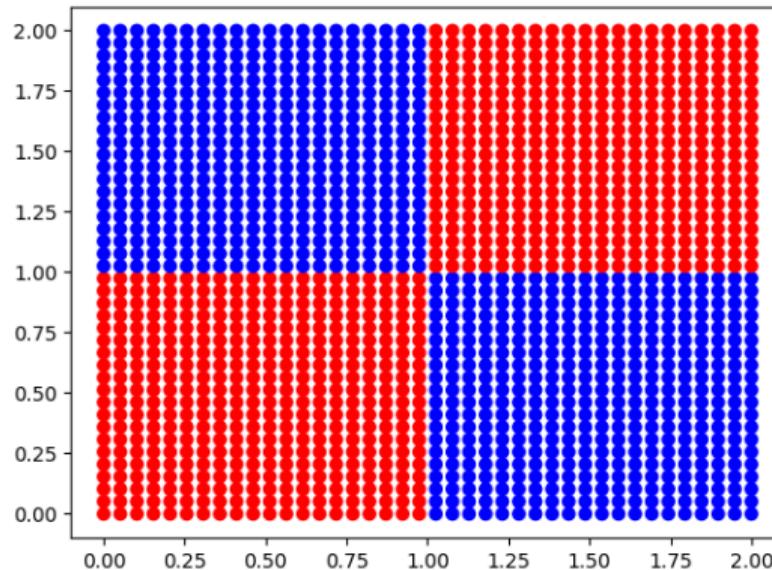


Reality (3rd Order - 10 parameters)

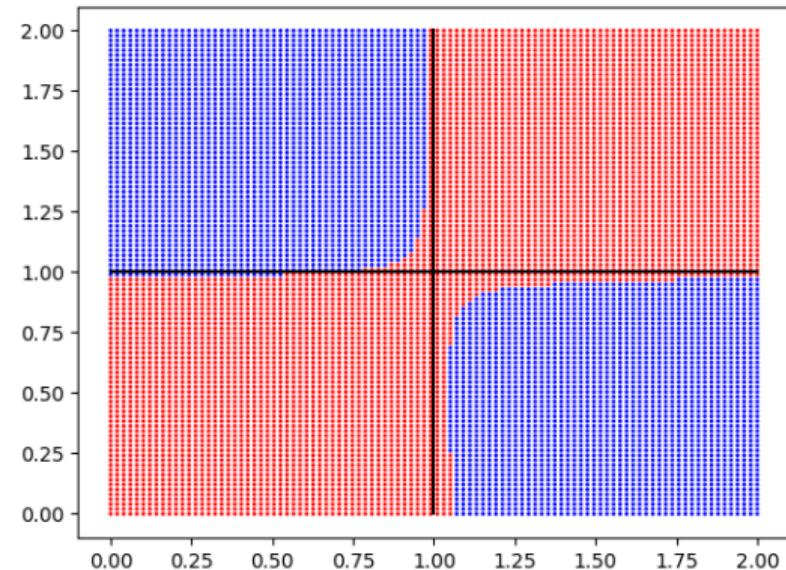


# The Limits of NonLinearity

Target

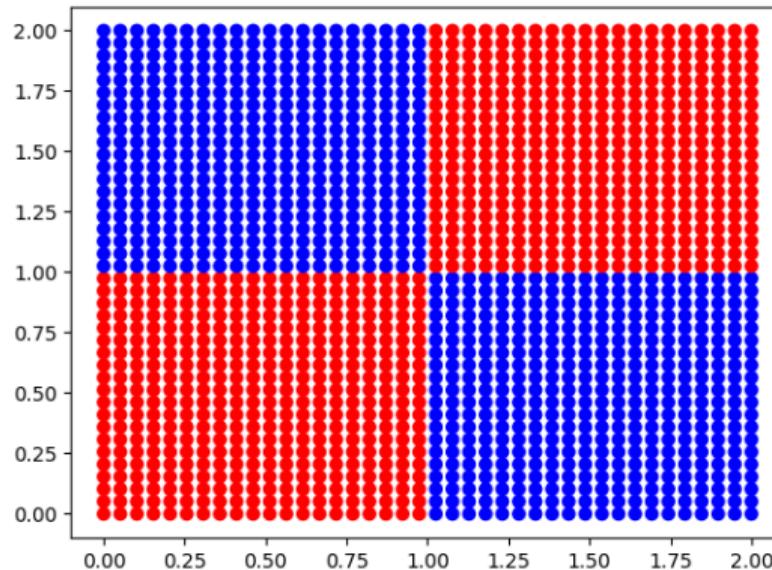


Reality (4th Order - 15 parameters)

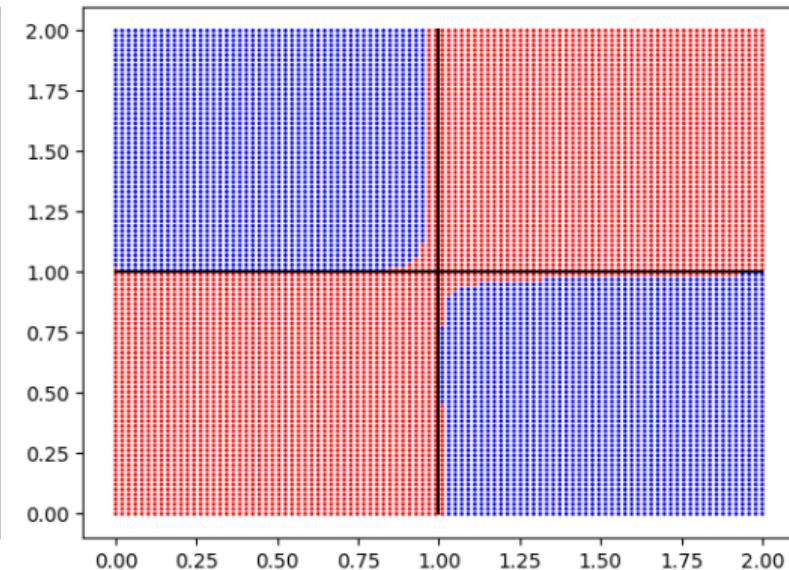


# The Limits of NonLinearity

Target

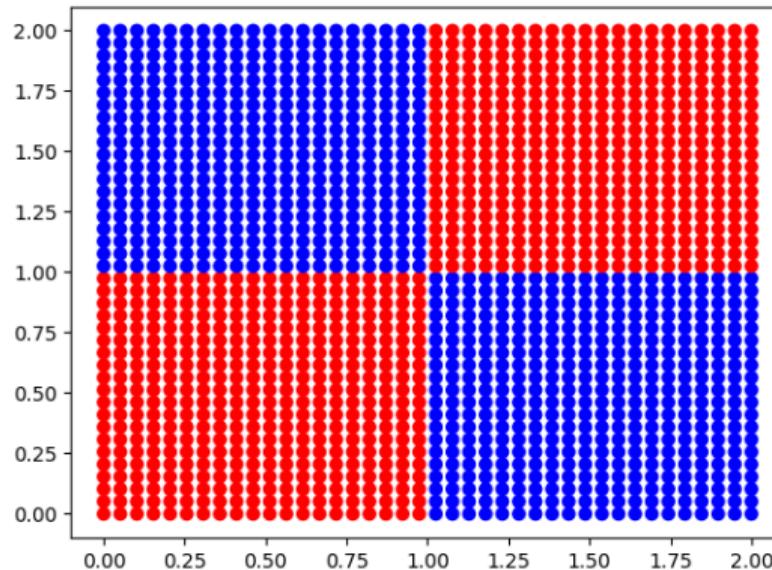


Reality (5th Order - 21 parameters)

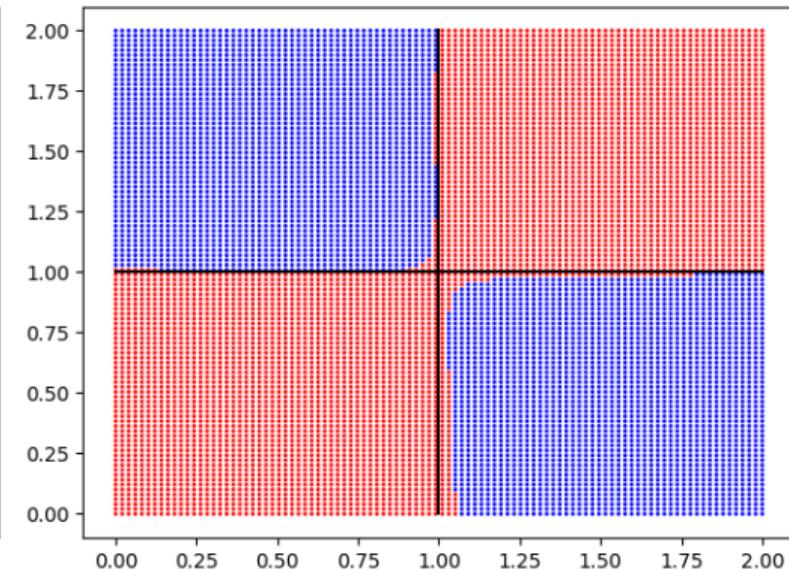


# The Limits of NonLinearity

Target

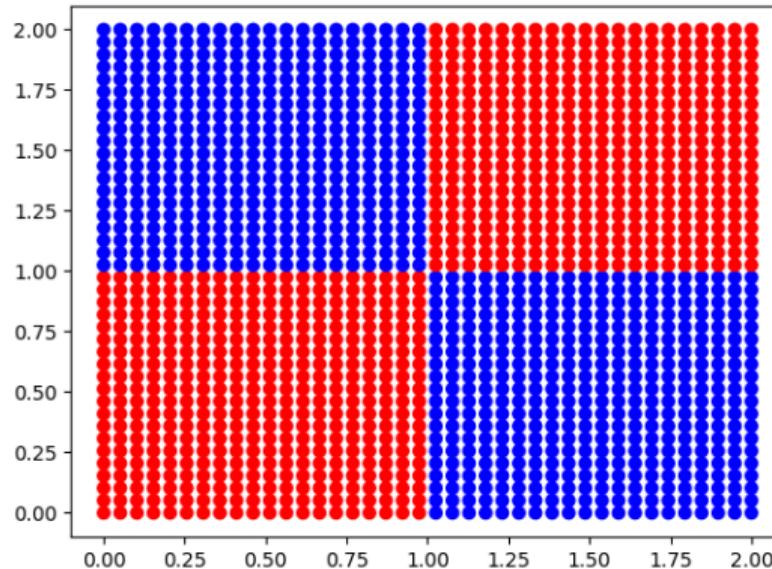


Reality (5th Order - 21 parameters)

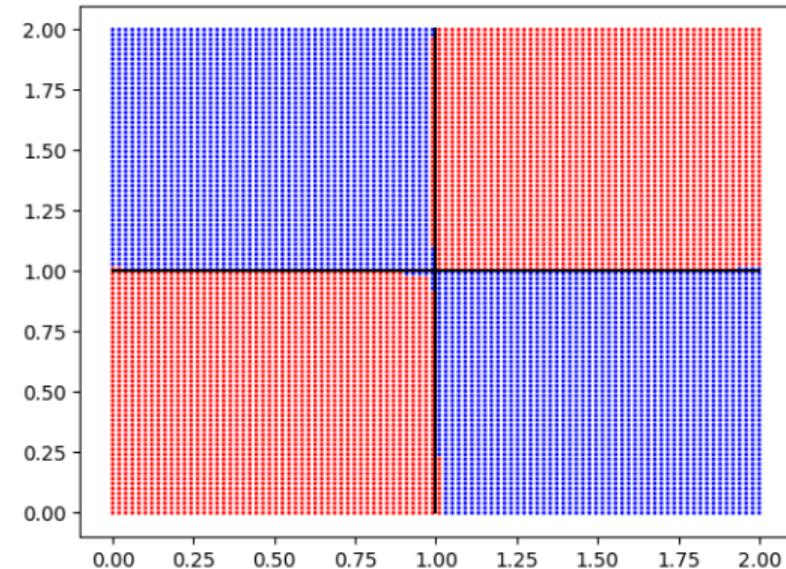


# The Limits of NonLinearity

Target



Reality (20th Order - 231 parameters)



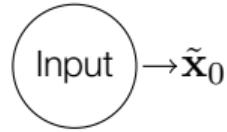
# What do to?

# What do to?

MORE!

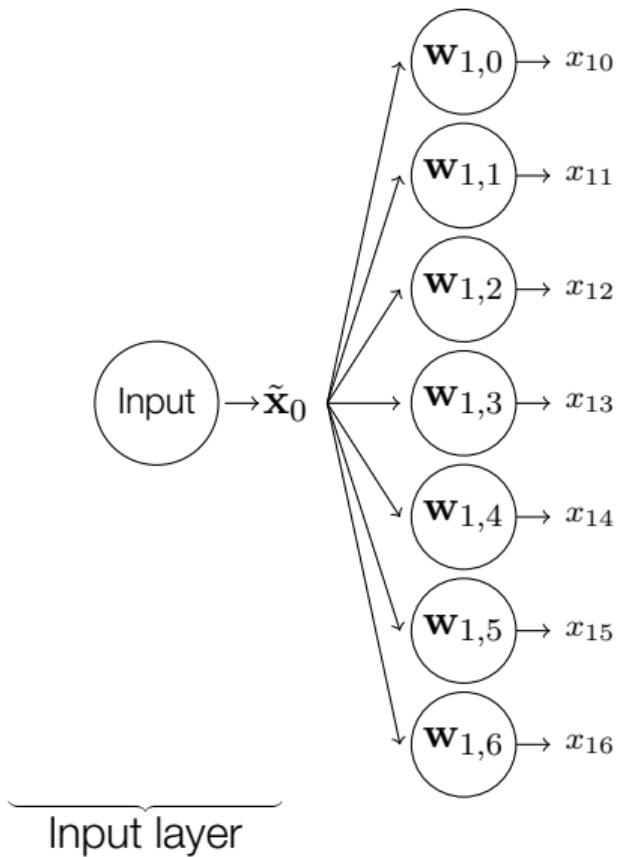
# The Multilayered Perceptron

# The Multilayered Perceptron

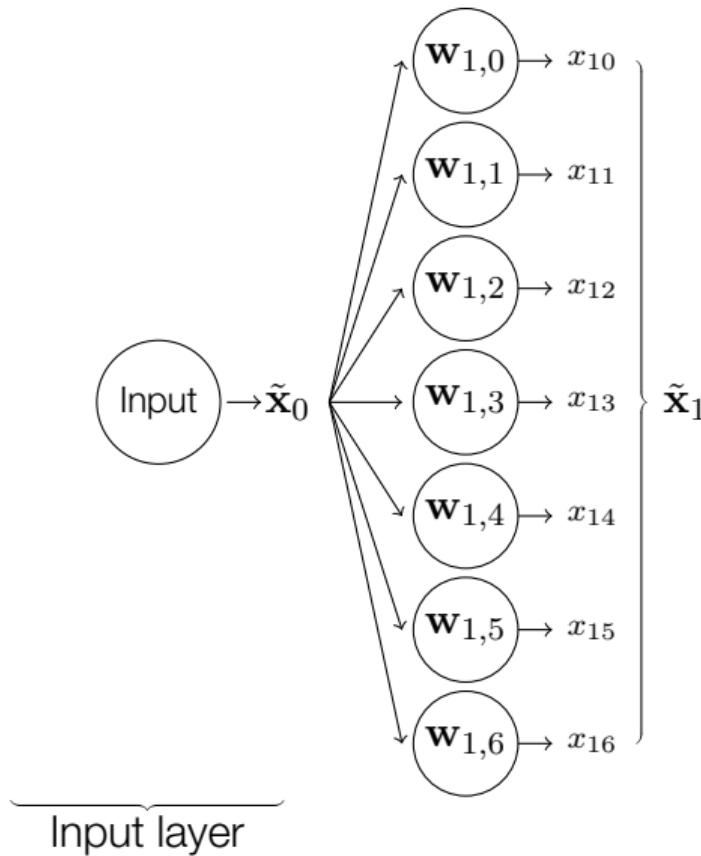


Input layer

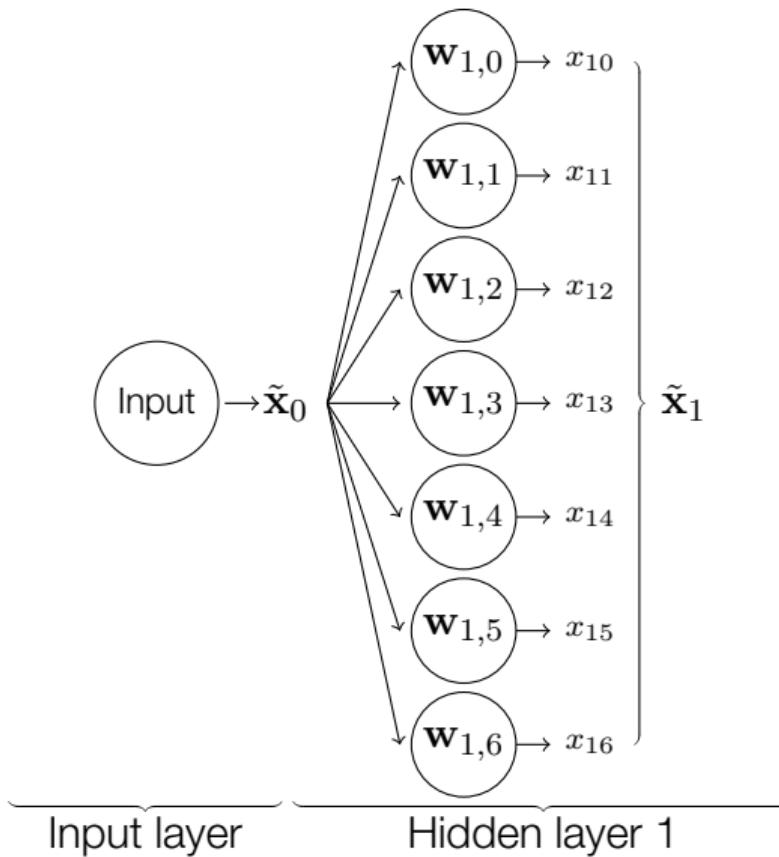
# The Multilayered Perceptron



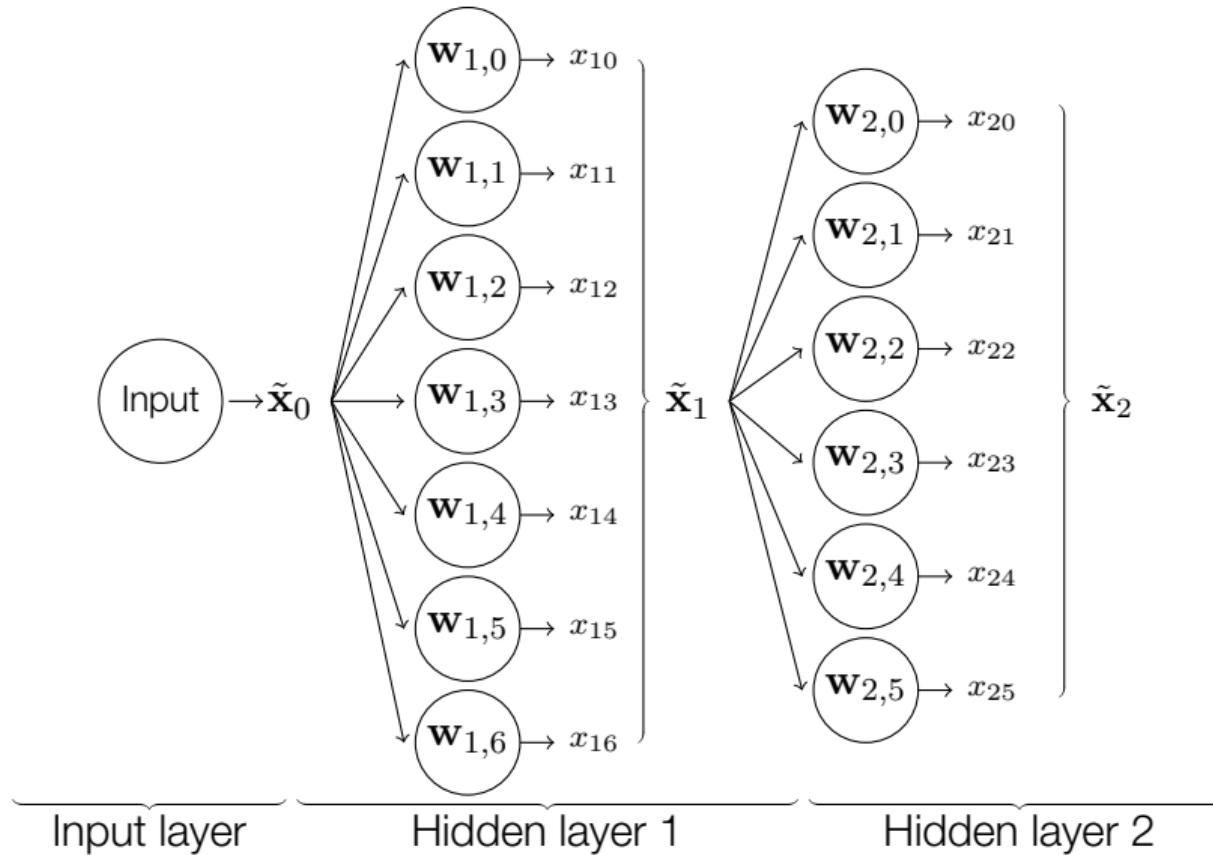
# The Multilayered Perceptron



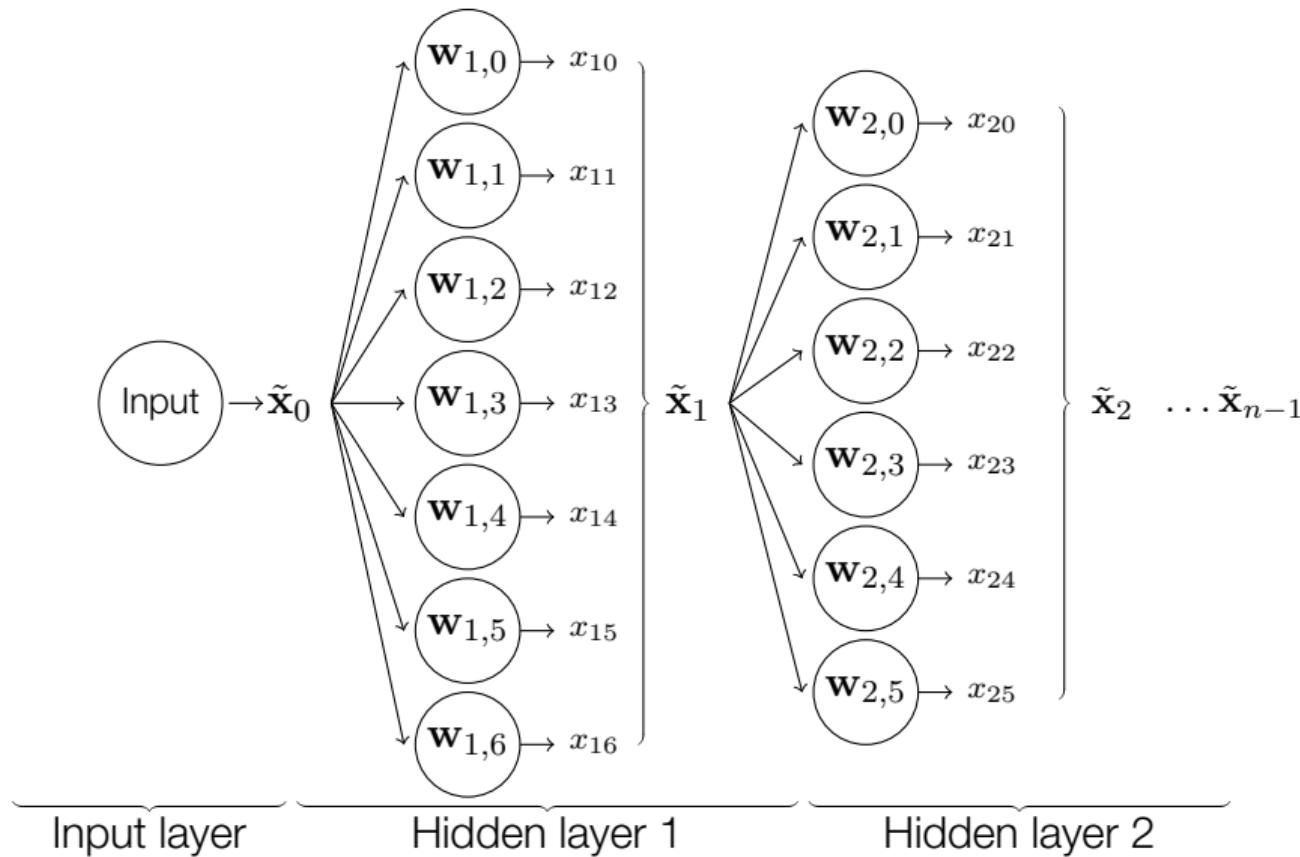
# The Multilayered Perceptron



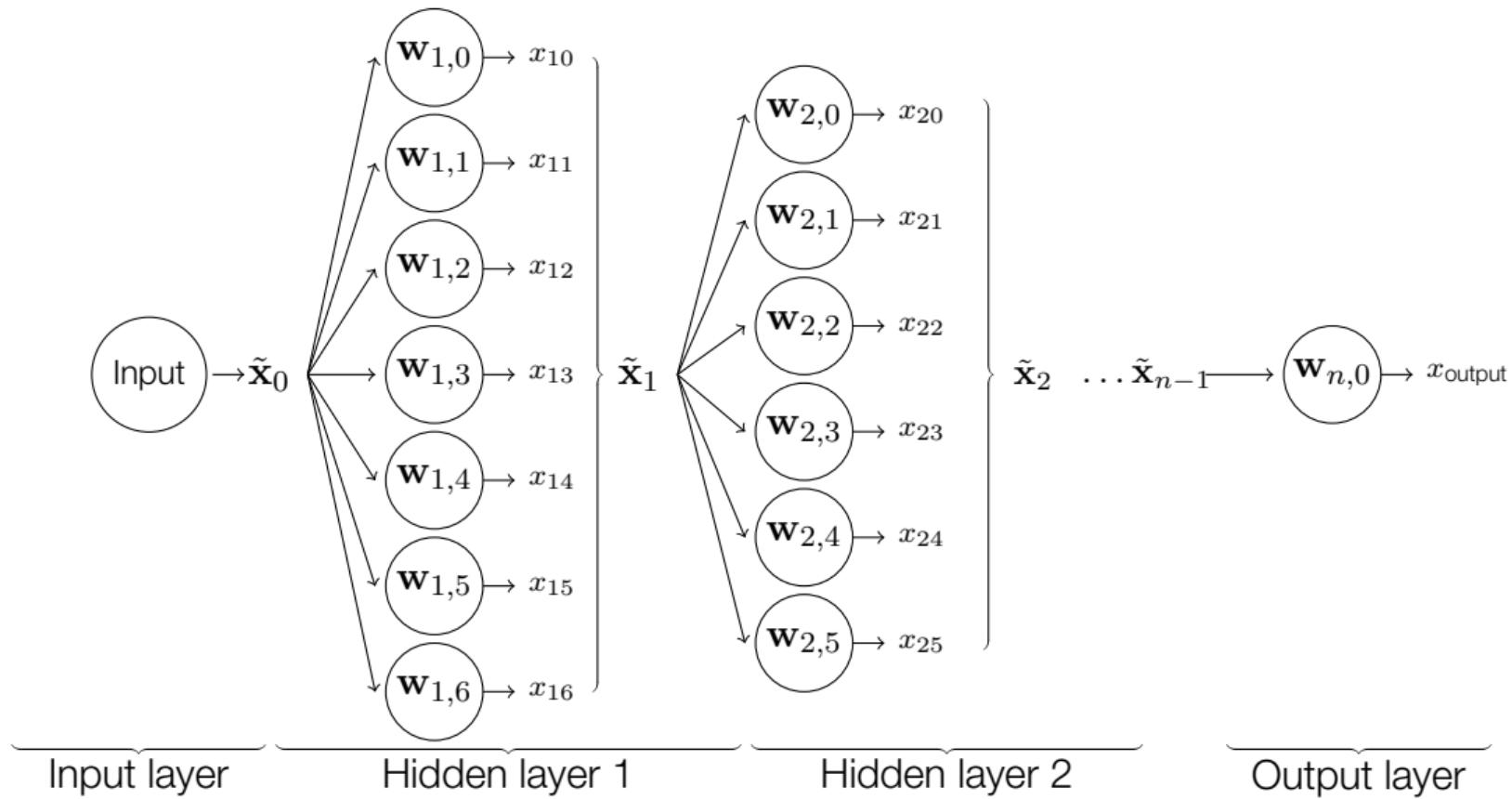
# The Multilayered Perceptron



# The Multilayered Perceptron



# The Multilayered Perceptron



# Exercise 4: The Multilayered Perceptron

**Task 4:** Write a Feedforward Network.

# Exercise 4: The Multilayered Perceptron

**Task 4:** Write a Feedforward Network.

**Guidance:** I have provided the skeleton of a Node, Layer and Network class. Randomly initialise your weights between -1 and 1.

# Exercise 4: The Multilayered Perceptron

**Task 4:** Write a Feedforward Network.

**Guidance:** I have provided the skeleton of a Node, Layer and Network class. Randomly initialise your weights between -1 and 1.

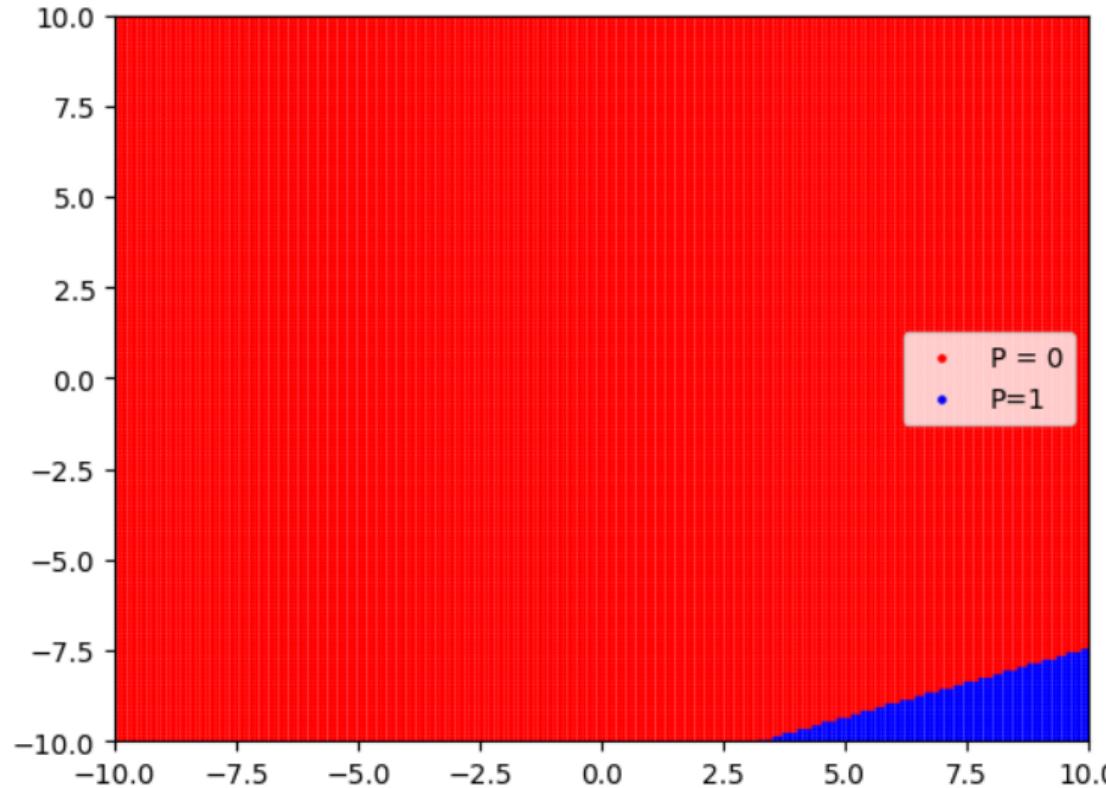
**Question:** Do you notice anything odd about how this predictor behaves?

# All that for nothing?

If everything went well, you should see something like this:

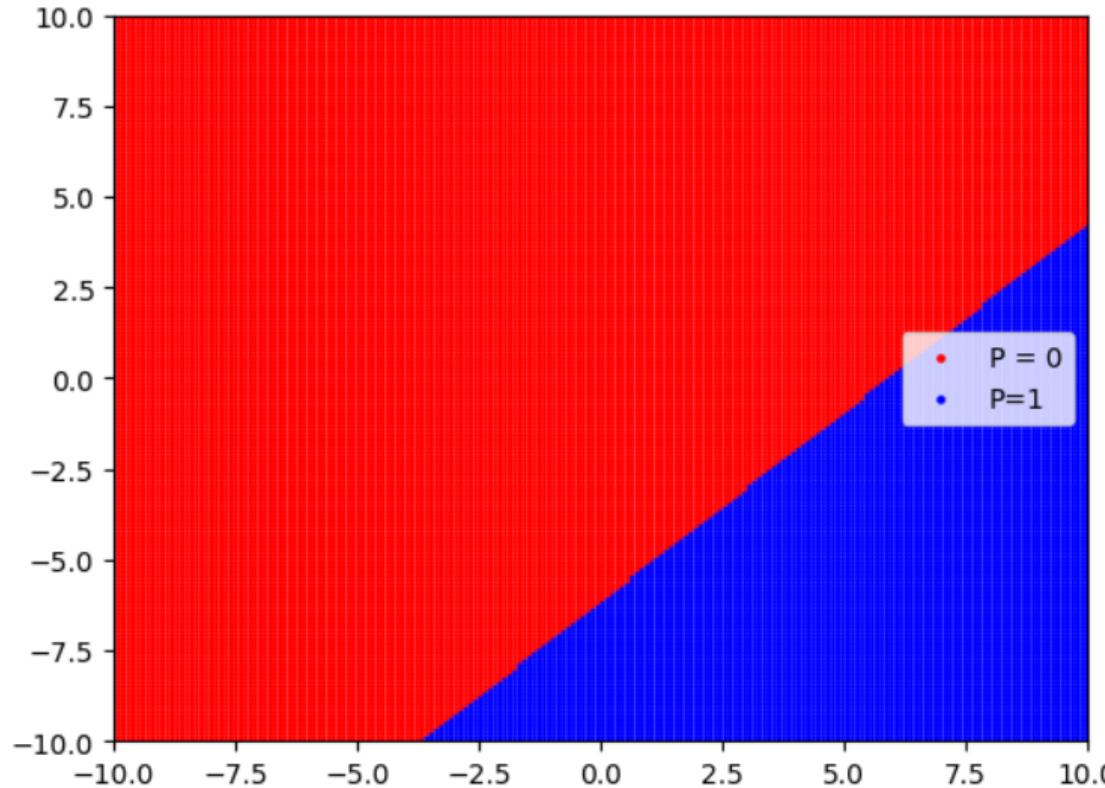
# All that for nothing?

If everything went well, you should see something like this:



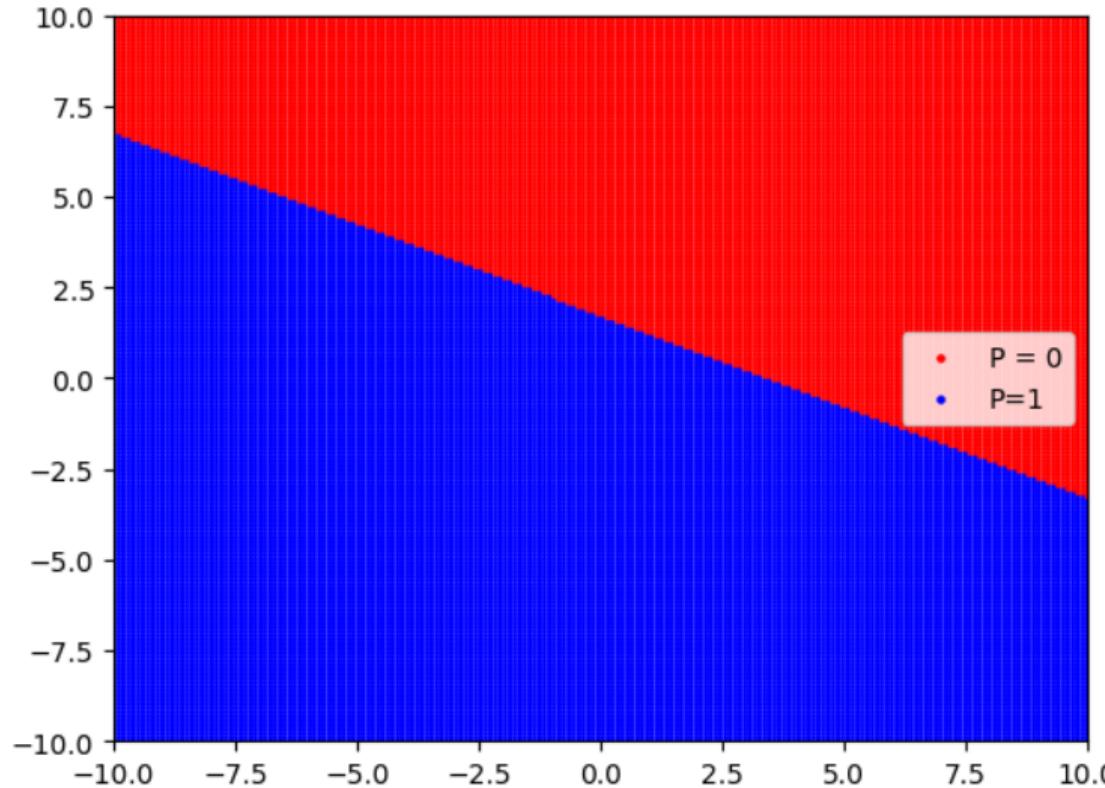
# All that for nothing?

If everything went well, you should see something like this:



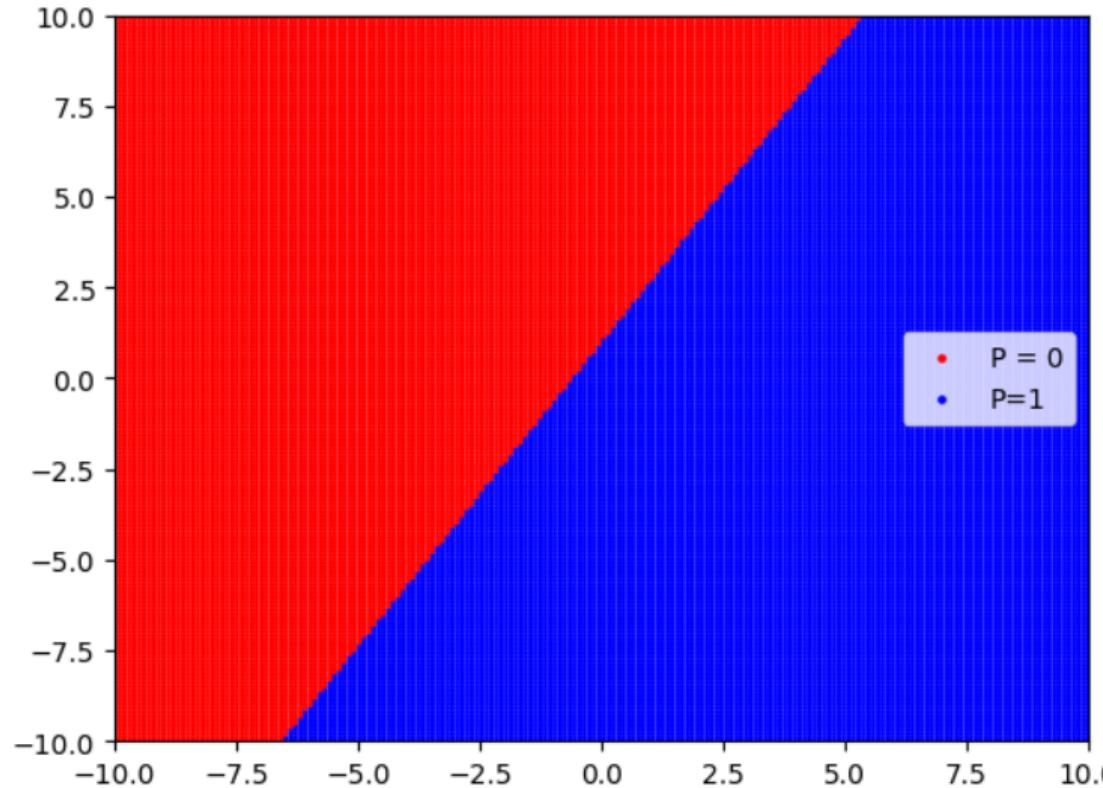
# All that for nothing?

If everything went well, you should see something like this:

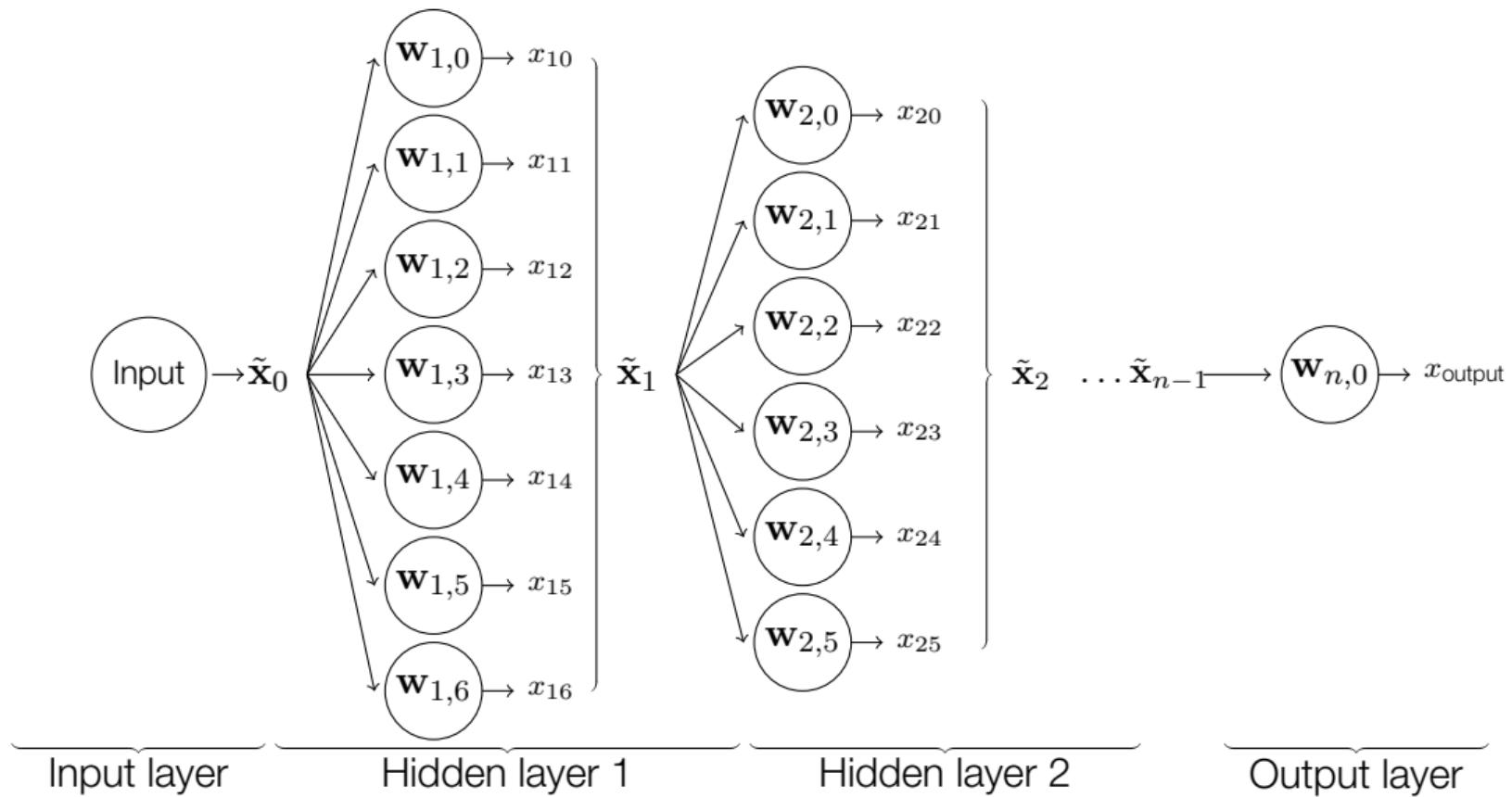


# All that for nothing?

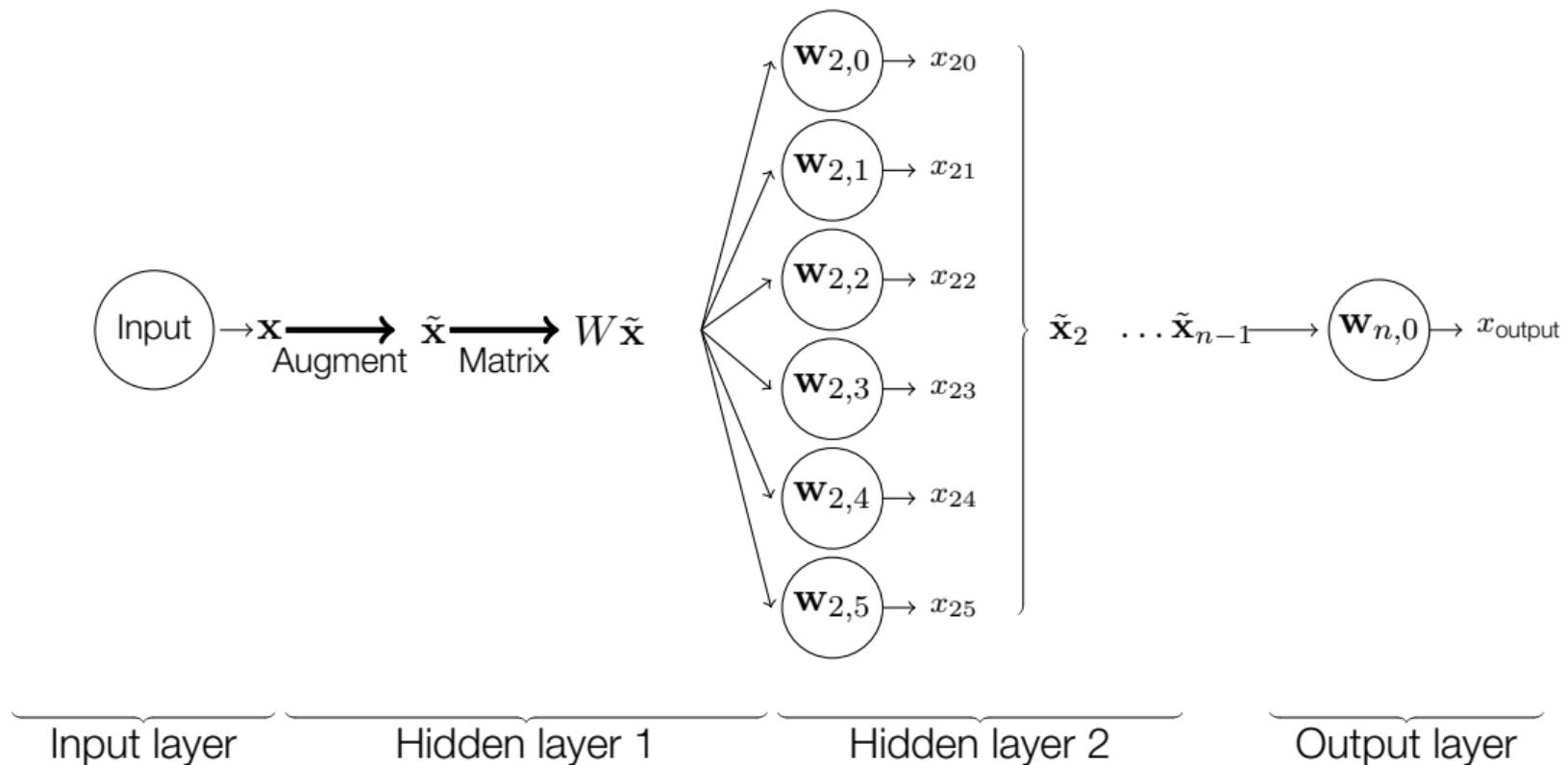
If everything went well, you should see something like this:



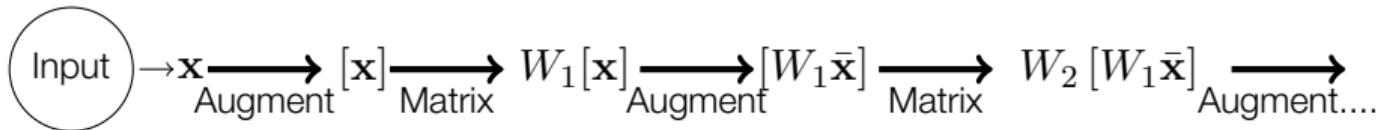
# Affine Mess We Find Ourselves in



# Affine Mess We Find Ourselves in



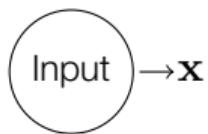
# Affine Mess We Find Ourselves in



# Affine Mess We Find Ourselves in

Or, in terms of **Affine Operators** (Chapter 3.4 in the notes):

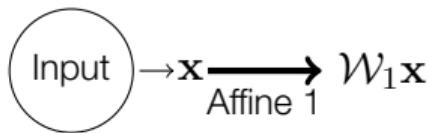
$$\left( \text{Affine} = \text{Matrix multiplication} + \text{translation} \implies \mathcal{W}\mathbf{v} = W\mathbf{v} + \mathbf{w} \right)$$



# Affine Mess We Find Ourselves in

Or, in terms of **Affine Operators** (Chapter 3.4 in the notes):

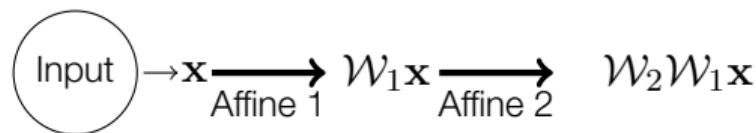
$$\left( \text{Affine} = \text{Matrix multiplication} + \text{translation} \implies \mathcal{W}\mathbf{v} = W\mathbf{v} + \mathbf{w} \right)$$



# Affine Mess We Find Ourselves in

Or, in terms of **Affine Operators** (Chapter 3.4 in the notes):

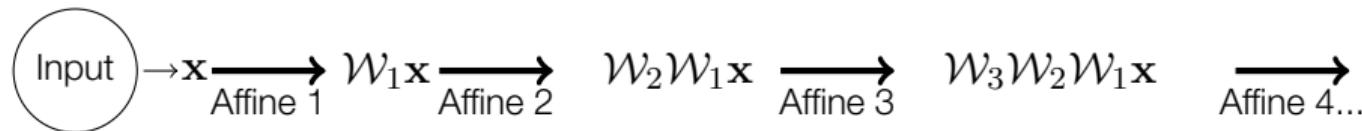
$$\left( \text{Affine} = \text{Matrix multiplication} + \text{translation} \implies \mathcal{W}\mathbf{v} = W\mathbf{v} + \mathbf{w} \right)$$



# Affine Mess We Find Ourselves in

Or, in terms of **Affine Operators** (Chapter 3.4 in the notes):

$$\left( \text{Affine} = \text{Matrix multiplication} + \text{translation} \implies \mathcal{W}\mathbf{v} = W\mathbf{v} + \mathbf{w} \right)$$



# Affine Mess We Find Ourselves in

But...affine operators obey:

$$\hat{\mathcal{W}}_2 \hat{\mathcal{W}}_1 = \hat{\mathcal{V}} \quad (5)$$

(Affine + Affine = Affine)

Which means:

$$\hat{\mathcal{W}}_N \hat{\mathcal{W}}_{N-1} \hat{\mathcal{W}}_{N-2} \dots \hat{\mathcal{W}}_1 \mathbf{x}_{\text{input}} = \hat{\mathcal{V}} \mathbf{x}_{\text{input}} \quad (6)$$

# Affine Mess We Find Ourselves in

But...affine operators obey:

$$\hat{\mathcal{W}}_2 \hat{\mathcal{W}}_1 = \hat{\mathcal{V}} \quad (5)$$

(Affine + Affine = Affine)

Which means:

$$\hat{\mathcal{W}}_N \hat{\mathcal{W}}_{N-1} \hat{\mathcal{W}}_{N-2} \dots \hat{\mathcal{W}}_1 \mathbf{x}_{\text{input}} = \hat{\mathcal{V}} \mathbf{x}_{\text{input}} \quad (6)$$

If you do the maths:

$$\hat{\mathcal{V}} \mathbf{x} = \mathbf{v} \cdot \mathbf{x} + \mathbf{b} \quad (7)$$

# Affine Mess We Find Ourselves in

But...affine operators obey:

$$\hat{\mathcal{W}}_2 \hat{\mathcal{W}}_1 = \hat{\mathcal{V}} \quad (5)$$

(Affine + Affine = Affine)

Which means:

$$\hat{\mathcal{W}}_N \hat{\mathcal{W}}_{N-1} \hat{\mathcal{W}}_{N-2} \dots \hat{\mathcal{W}}_1 \mathbf{x}_{\text{input}} = \hat{\mathcal{V}} \mathbf{x}_{\text{input}} \quad (6)$$

If you do the maths:

$$\hat{\mathcal{V}} \mathbf{x} = \mathbf{v} \cdot \mathbf{x} + \mathbf{b} \quad (7)$$

Which is just....a single perceptron layer

# Affine Mess We Find Ourselves in

If all you do is multiply by matrices & add vectors, you will *never* be doing anything other than an (extremely wasteful) perceptron algorithm

# NonLinearity to the Rescue!

Need to break this affine relationship: enter **activation functions**.

# NonLinearity to the Rescue!

Let:

$$\mathbf{x}_n = \text{activate}(W\mathbf{x}_{n-1} + \mathbf{b}) \quad (8)$$

# NonLinearity to the Rescue!

Let:

$$\mathbf{x}_n = \text{activate}(W\mathbf{x}_{n-1} + \mathbf{b}) \quad (8)$$

One of the simplest ways to break linearity is to add a *elementwise function*:

$$\begin{aligned} \mathbf{v} &= W\mathbf{x}_{n-1} + \mathbf{b} \\ [\mathbf{x}_n]_i &= \sigma(v_i) \end{aligned} \quad (9)$$

# Elementwise Activation Functions

You can probably all name dozens of these functions in common use.

Sigmoid /Logit       $\sigma(x) = \frac{1}{1 + \exp(-x)}$

Rectified Linear Unit (ReLU)       $\text{Relu}(x) = \begin{cases} x & x > 0 \\ 0 & \text{else} \end{cases}$

Leaky ReLU       $\text{Lelu}(x) = \begin{cases} x & x > 0 \\ 0.01x & \text{else} \end{cases}$

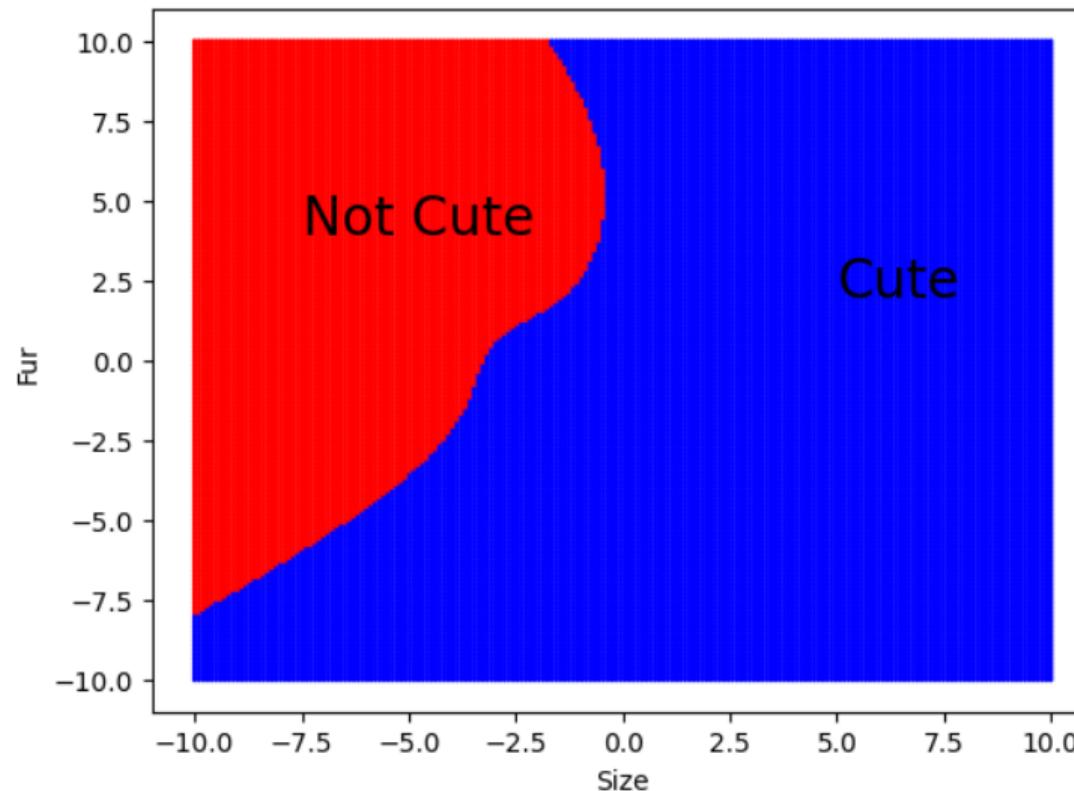
tanh       $\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$

Softplus       $\text{sft}(x) = \log(1 + \exp(x))$

(Bonus Question: Can you name any commonly used functions which **don't** follow this pattern?)

# NonLinearity Ahoy!

By replacing my (randomly initialised) hidden layers with Sigmoid nodes, I get:



# Exercise 5: Activation Functions

**Task 5:** Add in an activation function, and test if the linearity problem persists.

# Exercise 5: Activation Functions

**Task 5:** Add in an activation function, and test if the linearity problem persists.

**Guidance:** It is probably easiest to write “Custom Nodes” as child classes of your Nodes.

# Exercise 5: Activation Functions

**Task 5:** Add in an activation function, and test if the linearity problem persists.

**Guidance:** It is probably easiest to write “Custom Nodes” as child classes of your Nodes.

**Question:** Do you notice anything odd about how this predictor behaves?

# Training the Network

Buckle up, folks. The equations are about to start.

# Cost Function

We suppose we have a cost function,  $C(P, \ell)$  which measures how wrong a prediction  $P$  is, given a true value  $\ell$ :

# Cost Function

We suppose we have a cost function,  $C(P, \ell)$  which measures how wrong a prediction  $P$  is, given a true value  $\ell$ :

- ▶  $C$  should be differentiable,  $\frac{\partial C}{\partial P}$

# Cost Function

We suppose we have a cost function,  $C(P, \ell)$  which measures how wrong a prediction  $P$  is, given a true value  $\ell$ :

- ▶  $C$  should be differentiable,  $\frac{\partial C}{\partial P}$
- ▶  $C$  should be large when  $P \neq \ell$  and small when  $P \approx \ell$

# Cost Function

We suppose we have a cost function,  $C(P, \ell)$  which measures how wrong a prediction  $P$  is, given a true value  $\ell$ :

- ▶  $C$  should be differentiable,  $\frac{\partial C}{\partial P}$
- ▶  $C$  should be large when  $P \neq \ell$  and small when  $P \approx \ell$

The total cost of the network is then:

$$\mathcal{L} = \sum_i C(\mathbf{d}_i, \ell_i) \tag{10}$$

Need to find  $\mathbf{w}$  which minimise this cost.

# Optimisation

Analytical solutions are impossible, but numerically:

$$\mathbf{w} \rightarrow \mathbf{w} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \quad (11)$$

If we iterate this equation enough times we will approach an optimal value. This is **gradient descent**.

# The Chain Rule

Recall that:

$$y_\ell^i = \mathbf{w}_\ell^i \cdot \tilde{\mathbf{x}}_{\ell-1}$$

The chain rule says:

$$\frac{\partial C}{\partial \mathbf{w}_\ell^i} = \frac{\partial C}{\partial y_\ell^i} \times \frac{\partial y_\ell^i}{\partial \mathbf{w}_\ell^i} \quad (12)$$

$$= \frac{\partial C}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1} \quad (13)$$

# The Chain Rule

Recall that:

$$y_\ell^i = \mathbf{w}_\ell^i \cdot \tilde{\mathbf{x}}_{\ell-1}$$

The chain rule says:

$$\frac{\partial C}{\partial \mathbf{w}_\ell^i} = \frac{\partial C}{\partial y_\ell^i} \times \frac{\partial y_\ell^i}{\partial \mathbf{w}_\ell^i} \quad (12)$$

$$= \frac{\partial C}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1} \quad (13)$$

So....what is  $\frac{\partial C}{\partial y_\ell^i}$ ?

# The Chain Rule

This next step is magic.

# The Chain Rule

This next step is magic.

1.  $y_{i+1}^j$  is a function only of the set of  $y_i$  in the lower layer:

$$y_{i+1}^j = \mathbf{w}_{i+1}^j \cdot \begin{bmatrix} 1 \\ a_i(y_i^1) \\ a_i(y_i^2) \\ \dots \end{bmatrix}$$

# The Chain Rule

This next step is magic.

1.  $y_{i+1}^j$  is a function only of the set of  $y_i$  in the lower layer:

$$y_{i+1}^j = \mathbf{w}_{i+1}^j \cdot \begin{bmatrix} 1 \\ a_i(y_i^1) \\ a_i(y_i^2) \\ \dots \end{bmatrix}$$

2. Therefore can compute

$$\frac{\partial y_{i+1}^j}{\partial y_i^k} = d_i(y_i^k) [\mathbf{w}_{i+1}^j]_{k+1} \quad d_i(y) = \frac{da_i}{dy}$$

# The Chain Rule

This next step is magic.

1.  $y_{i+1}^j$  is a function only of the set of  $y_i$  in the lower layer:

$$y_{i+1}^j = \mathbf{w}_{i+1}^j \cdot \begin{bmatrix} 1 \\ a_i(y_i^1) \\ a_i(y_i^2) \\ \vdots \end{bmatrix}$$

2. Therefore can compute

$$\frac{\partial y_{i+1}^j}{\partial y_i^k} = d_i(y_i^k) [\mathbf{w}_{i+1}^j]_{k+1} \quad d_i(y) = \frac{da_i}{dy}$$

3. But our magic chain rule says:

$$\frac{\partial C}{\partial y_i^k} = \sum_j \frac{\partial C}{\partial y_{i+1}^j} \frac{\partial y_{i+1}^j}{\partial y_i^k}$$

# The Chain Rule

To summarise:

# The Chain Rule

To summarise:

- ▶ If we know  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$  we can update our weights

# The Chain Rule

To summarise:

- ▶ If we know  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$  we can update our weights
- ▶ If we know  $\frac{\partial C}{\partial \mathbf{w}_\ell^i}$ , we can compute  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$ :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} = \sum_{\text{data } \mathbf{q}} \frac{\partial C(\mathbf{q})}{\partial \mathbf{w}_\ell^i}$$

# The Chain Rule

To summarise:

- ▶ If we know  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$  we can update our weights
- ▶ If we know  $\frac{\partial C}{\partial \mathbf{w}_\ell^i}$ , we can compute  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$ :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} = \sum_{\text{data } \mathbf{q}} \frac{\partial C(\mathbf{q})}{\partial \mathbf{w}_\ell^i}$$

- ▶ If we know  $\frac{\partial C}{\partial y_\ell^i}$  we can compute

$$\frac{\partial C}{\partial \mathbf{w}_\ell^i} = \frac{\partial C}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1}$$

# The Chain Rule

To summarise:

- ▶ If we know  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$  we can update our weights
- ▶ If we know  $\frac{\partial C}{\partial \mathbf{w}_\ell^i}$ , we can compute  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$ :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} = \sum_{\text{data } \mathbf{q}} \frac{\partial C(\mathbf{q})}{\partial \mathbf{w}_\ell^i}$$

- ▶ If we know  $\frac{\partial C}{\partial y_\ell^i}$  we can compute

$$\frac{\partial C}{\partial \mathbf{w}_\ell^i} = \frac{\partial C}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1}$$

- ▶ If we know  $\frac{\partial C}{\partial y_N^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-1}^j}$

# The Chain Rule

To summarise:

- ▶ If we know  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$  we can update our weights
- ▶ If we know  $\frac{\partial C}{\partial \mathbf{w}_\ell^i}$ , we can compute  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$ :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} = \sum_{\text{data } \mathbf{q}} \frac{\partial C(\mathbf{q})}{\partial \mathbf{w}_\ell^i}$$

- ▶ If we know  $\frac{\partial C}{\partial y_\ell^i}$  we can compute

$$\frac{\partial C}{\partial \mathbf{w}_\ell^i} = \frac{\partial C}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1}$$

- ▶ If we know  $\frac{\partial C}{\partial y_N^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-1}^j}$
- ▶ If we know  $\frac{\partial C}{\partial y_{N-1}^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-2}^j}$

# The Chain Rule

To summarise:

- ▶ If we know  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$  we can update our weights
- ▶ If we know  $\frac{\partial C}{\partial \mathbf{w}_\ell^i}$ , we can compute  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$ :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} = \sum_{\text{data } \mathbf{q}} \frac{\partial C(\mathbf{q})}{\partial \mathbf{w}_\ell^i}$$

- ▶ If we know  $\frac{\partial C}{\partial y_\ell^i}$  we can compute

$$\frac{\partial C}{\partial \mathbf{w}_\ell^i} = \frac{\partial C}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1}$$

- ▶ If we know  $\frac{\partial C}{\partial y_N^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-1}^j}$
- ▶ If we know  $\frac{\partial C}{\partial y_{N-1}^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-2}^j}$
- ▶ If we know  $\frac{\partial C}{\partial y_{N-2}^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-3}^j}$

# The Chain Rule

To summarise:

- ▶ If we know  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$  we can update our weights
- ▶ If we know  $\frac{\partial C}{\partial \mathbf{w}_\ell^i}$ , we can compute  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i}$ :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} = \sum_{\text{data } \mathbf{q}} \frac{\partial C(\mathbf{q})}{\partial \mathbf{w}_\ell^i}$$

- ▶ If we know  $\frac{\partial C}{\partial y_\ell^i}$  we can compute

$$\frac{\partial C}{\partial \mathbf{w}_\ell^i} = \frac{\partial C}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1}$$

- ▶ If we know  $\frac{\partial C}{\partial y_N^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-1}^j}$
- ▶ If we know  $\frac{\partial C}{\partial y_{N-1}^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-2}^j}$
- ▶ If we know  $\frac{\partial C}{\partial y_{N-2}^j}$ , we can compute  $\frac{\partial C}{\partial y_{N-3}^j}$
- ▶ ....

# The Missing Ingredient

So what is  $\frac{\partial C}{\partial y_N^j}$ ?

# The Missing Ingredient

Final layer has:

# The Missing Ingredient

Final layer has:

$$\mathbf{P} = \mathbf{x}_N \quad (14)$$

# The Missing Ingredient

Final layer has:

$$\mathbf{P} = \mathcal{A}(\mathbf{y}_N) \tag{14}$$

# The Missing Ingredient

Final layer has:

$$\mathbf{P} = \begin{pmatrix} a_N(y_N^1) \\ a_N(y_N^2) \\ a_N(y_N^3) \\ \vdots \end{pmatrix} \quad (14)$$

# The Missing Ingredient

Final layer has:

$$\begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} a_N(\mathbf{w}_N^1 \cdot \tilde{x}_{N-1}) \\ a_N(\mathbf{w}_N^2 \cdot \tilde{x}_{N-1}) \\ a_N(\mathbf{w}_N^3 \cdot \tilde{x}_{N-1}) \\ \vdots \end{pmatrix} \quad (14)$$

# Getting the Gradients

# Getting the Gradients

Then:

$$\frac{\partial P_i}{\partial y_N^i} = d_N(y_N^i) \quad (15)$$

# Getting the Gradients

Then:

$$\frac{\partial P_i}{\partial y_N^i} = d_N(y_N^i) \quad (15)$$

Which means:

$$\frac{\partial C}{\partial y_N^i} = \frac{\partial C}{\partial P_i} \times \frac{\partial P_i}{\partial y_N^i} \quad (16)$$

$$= d_N(y_N^i) \frac{\partial C}{\partial P_i} \quad (17)$$

# Backpropagation, Explained (right?)

Collecting this altogether, what we need to compute is:

$$\frac{\partial C_q}{\partial y_\ell^i} = \begin{cases} \frac{\partial C_q}{\partial P_q^i} \times d_N(y_N^i) & \text{if } \ell = N - 1 \\ d_\ell(y_\ell^i) \sum_j [\mathbf{w}_{\ell+1}^j]_{i+1} \frac{\partial C_q}{\partial y_{\ell+1}^j} & \text{otherwise} \end{cases}$$
$$\frac{\partial C_q}{\partial \mathbf{w}_\ell^i} = \frac{\partial C_q}{\partial y_\ell^i} \tilde{\mathbf{x}}_{\ell-1}$$

# Task 6: Training Day

**Task 6:** Assemble the obfuscated code in Data/jigsaw.py into the class structures provided.

# Task 6: Training Day

**Task 6:** Assemble the obfuscated code in Data/jigsaw.py into the class structures provided.

**Guidance:** Hints are available in Hints/jigsaw.py

# Task 6: Training Day

**Task 6:** Assemble the obfuscated code in `Data/jigsaw.py` into the class structures provided.

**Guidance:** Hints are available in `Hints/jigsaw.py`

**Question:** Can the classifier fit the augmented cuteness data? How about the ‘complex classifier’ problem in `Data/complex_classifier??` Can your network do functional regression with `Data/complex_function?`

# Complex Function

The function you're trying to emulate is:

$$f(x, y) = \frac{10}{1 + \exp\left(10\sqrt{x^2 + y^2} - 5\right)} - \frac{15}{1 + \exp\left(5\sqrt{(x - 2)^2 + (y - 2)^2} - 2.5\right)} - \frac{15}{1 + \exp\left(5\sqrt{(x + 2)^2 + (y - 2)^2} - 2.5\right)} - \frac{18}{1 + \exp\left(2 \left[ (0.1 - x/2)^2 + 7 \left(\frac{5}{4}(y + 2.5) - \frac{x^2}{4}\right)^2 \right] - 1\right)}$$

# Thank you!

