ML Part 2: Intro to Neural networks

Abhijith Gandrakota

CODAS-HEP 2023
Princeton University, NJ

Lecture adapted from J. Ngadiuba's and M. Kagan's courses

Range of ML Algorithms

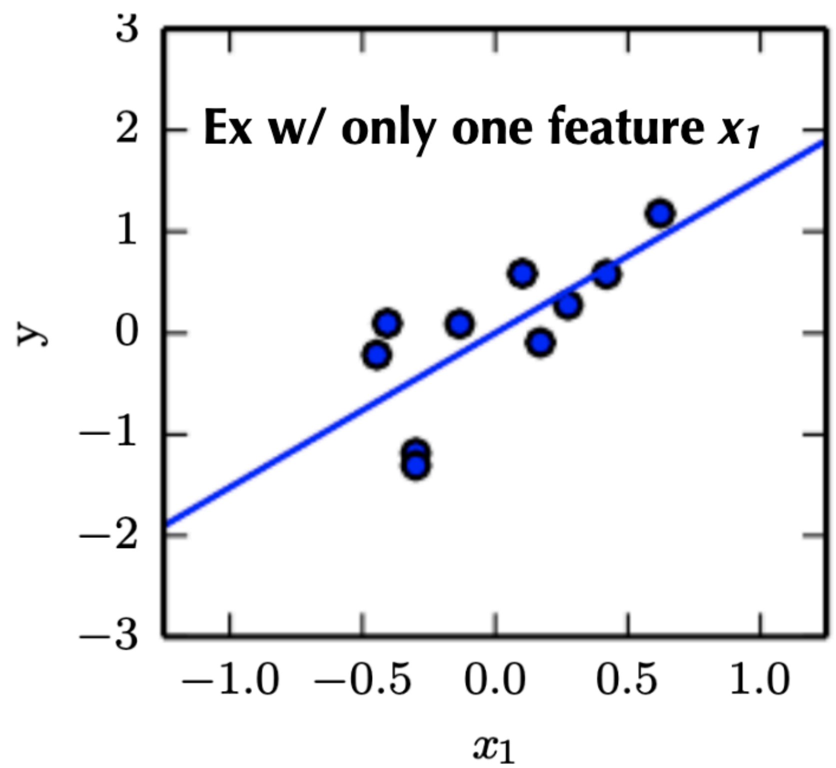
Linear regression

Transformers
Graphs, etc . . . .

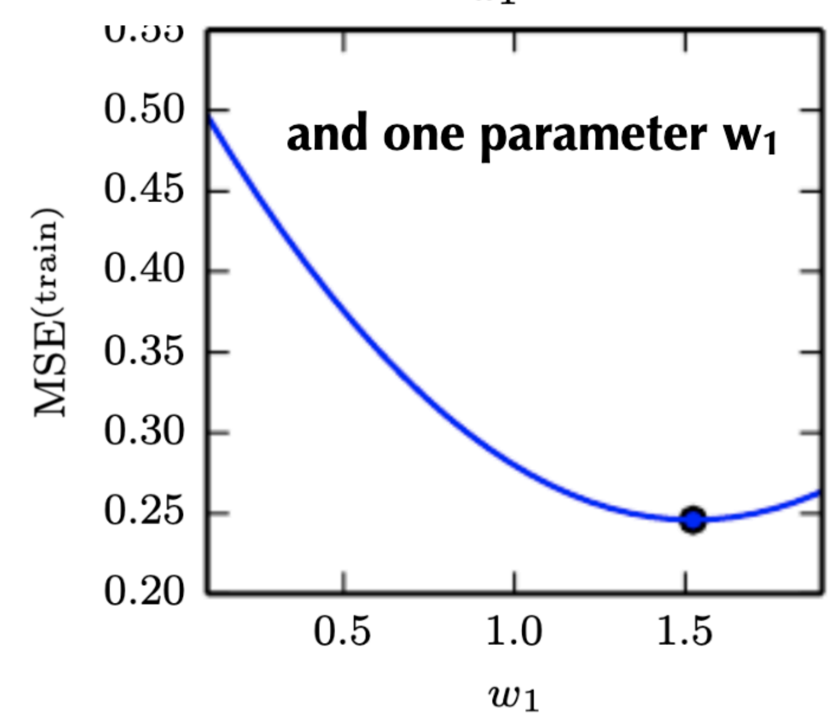
In this session!

Recap: Linear Regression

- Set of inputs (x_i) & Output (y_i) pairs, which comprises our data
  - Inputs: $x_i \in \mathbb{R}^m$ (m is the number of features)
  - Targets: $y_i \in \mathbb{R}^n$ (n is the number of features)



- Model that describes it: $\hat{y} = W^T X$
  - Training was to find the best parameters W That describe the data well
    - Objective: $\mathscr{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h(\mathbf{x}_i; \mathbf{w}))^2$

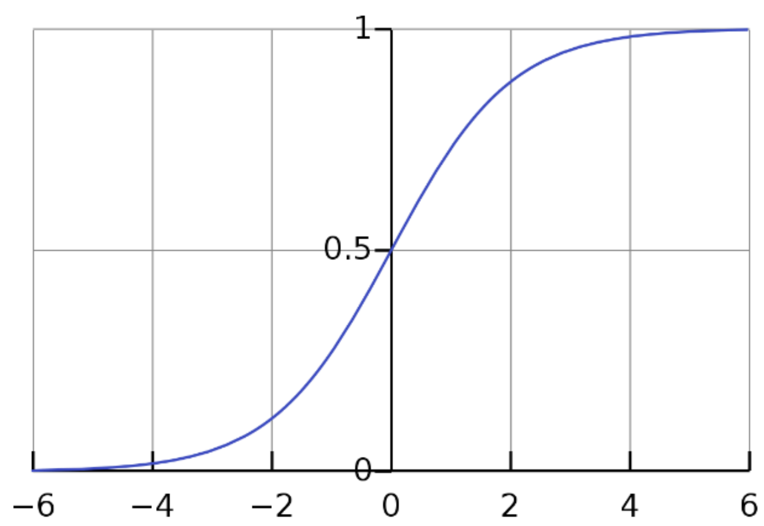


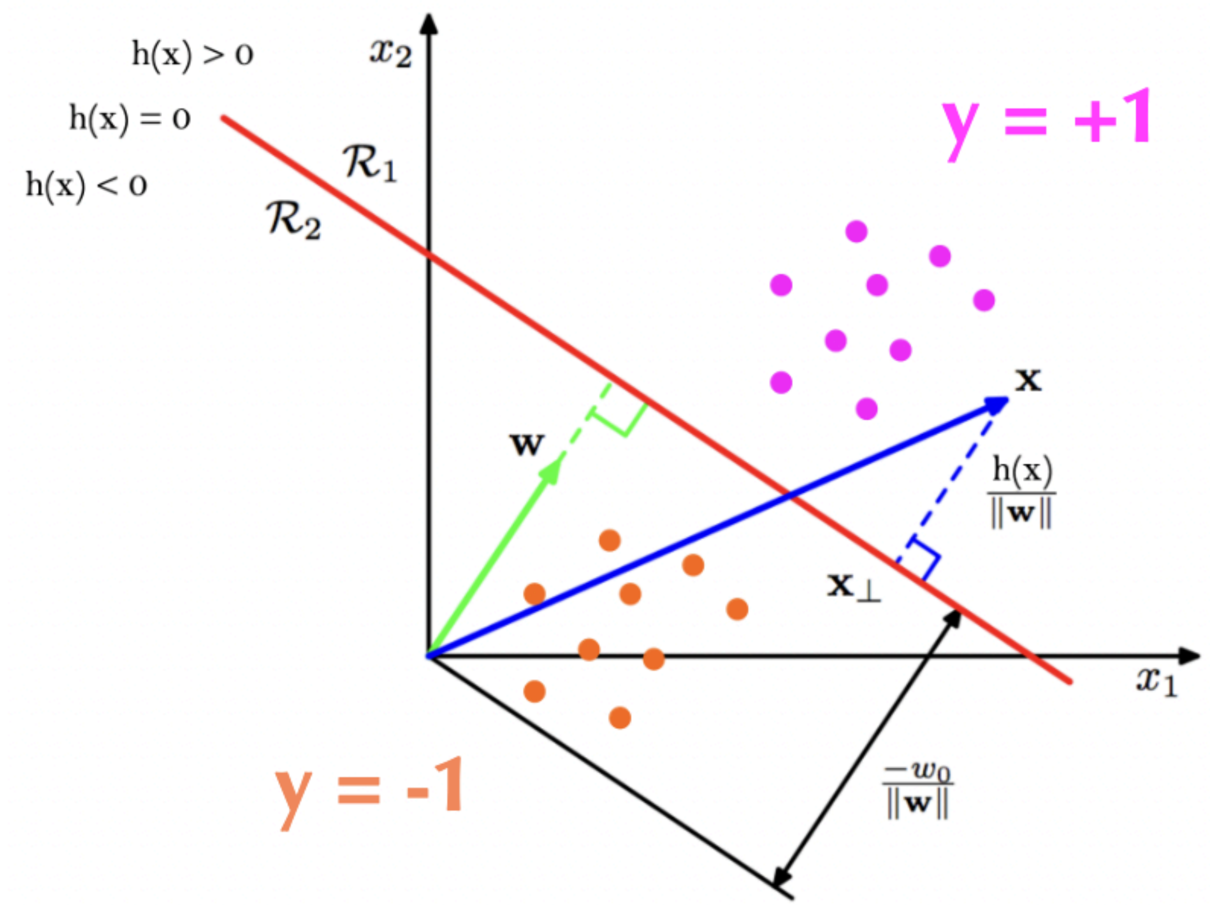
- The model here is linear in weight space

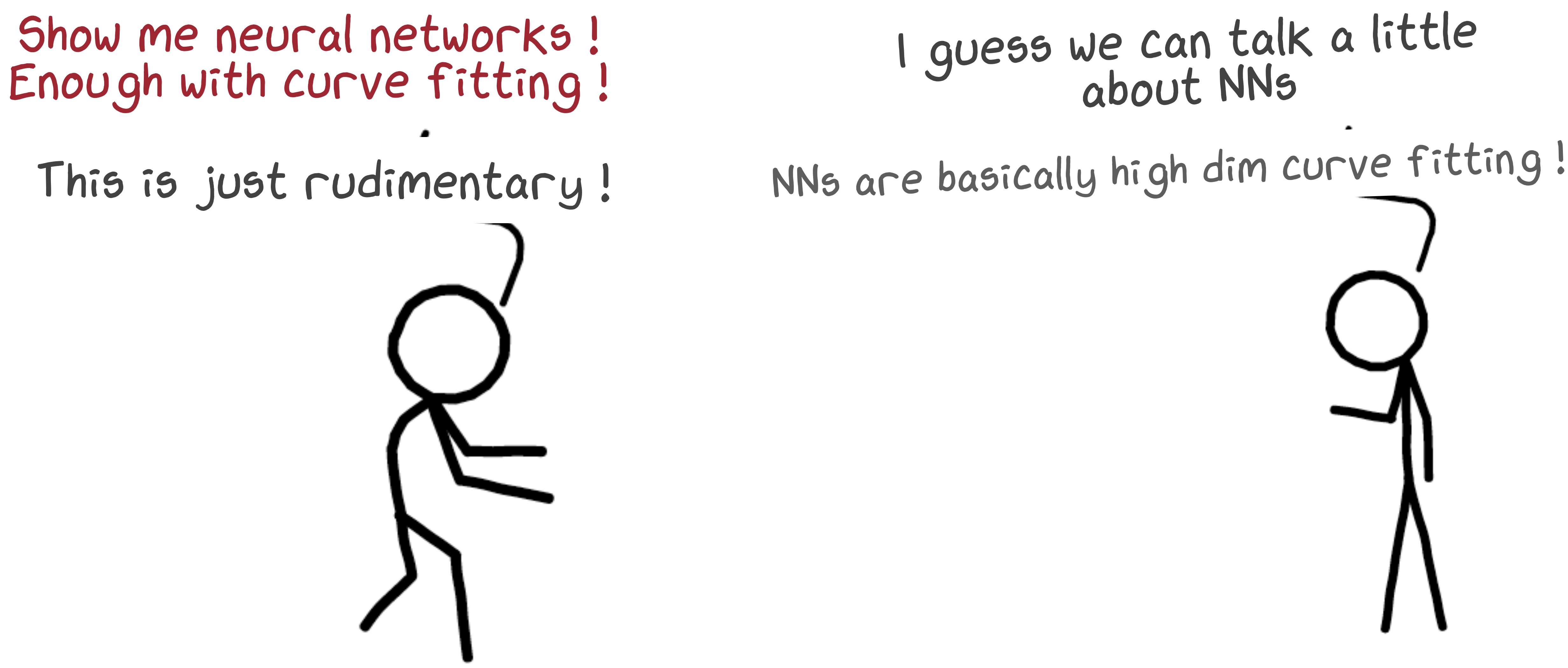
Recap: Logistic Regression

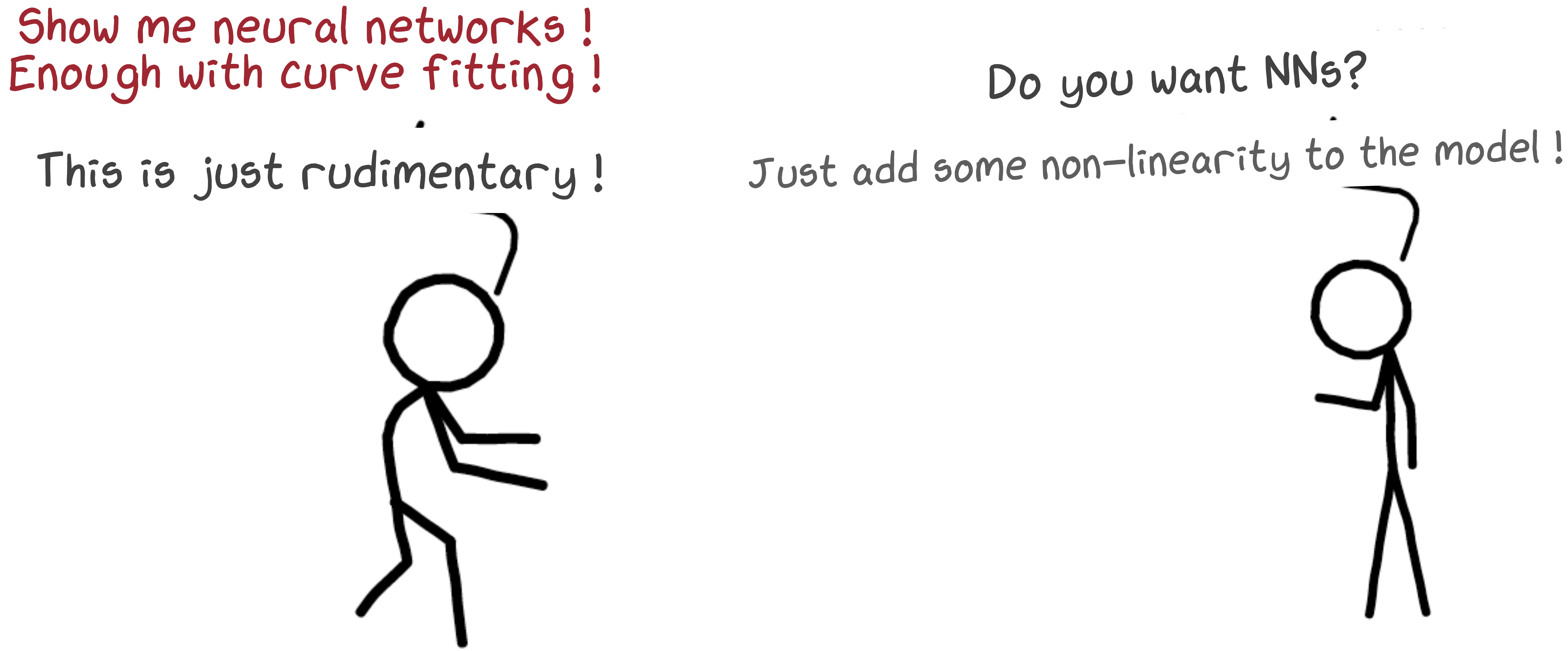
- Set of inputs(x_i) & Output(y_i) pairs, which comprises our data
  - Inputs: $x_i \in \mathbb{R}^m$ (m is the number of features)
  - Targets: $y_i \in \{0,1\}^n$ (n classes)
- Model that describes it: $\hat{y} = W^T X$
  - Map the output to a logistic sigmoid

$$p(y=1\,|\,\mathbf{x}) \equiv p_i = \frac{1}{1+e^{-h(\mathbf{x};\mathbf{w})}} = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$



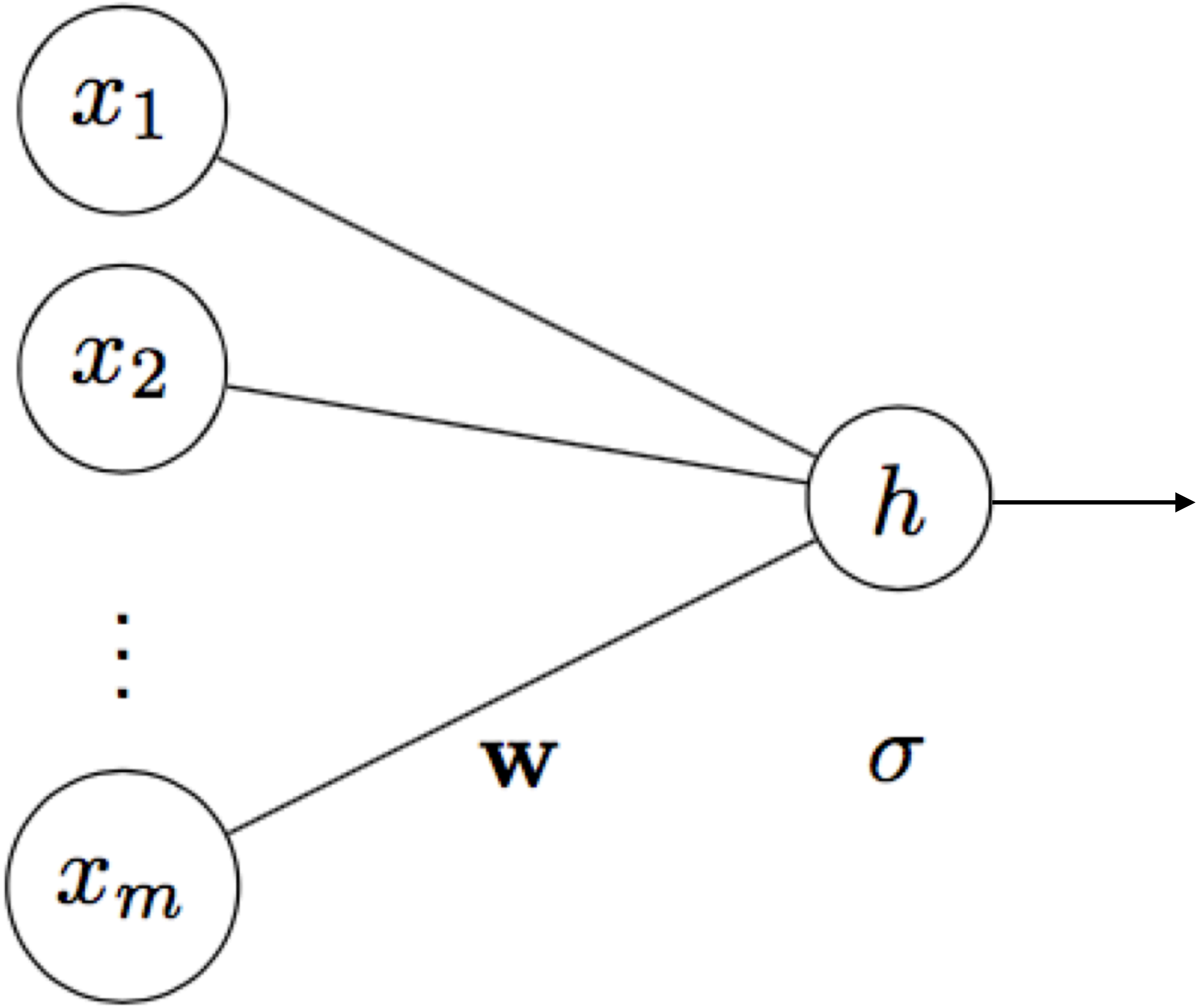



Show me neural networks !
Enough with curve fitting !
This is just rudimentary !
I guess we can talk a little about NNs
NNs are basically high dim curve fitting !


Show me neural networks !
Enough with curve fitting !
This is just rudimentary !
Do you want NNs?
Just add some non-linearity to the model !

Lets take another look . . . .

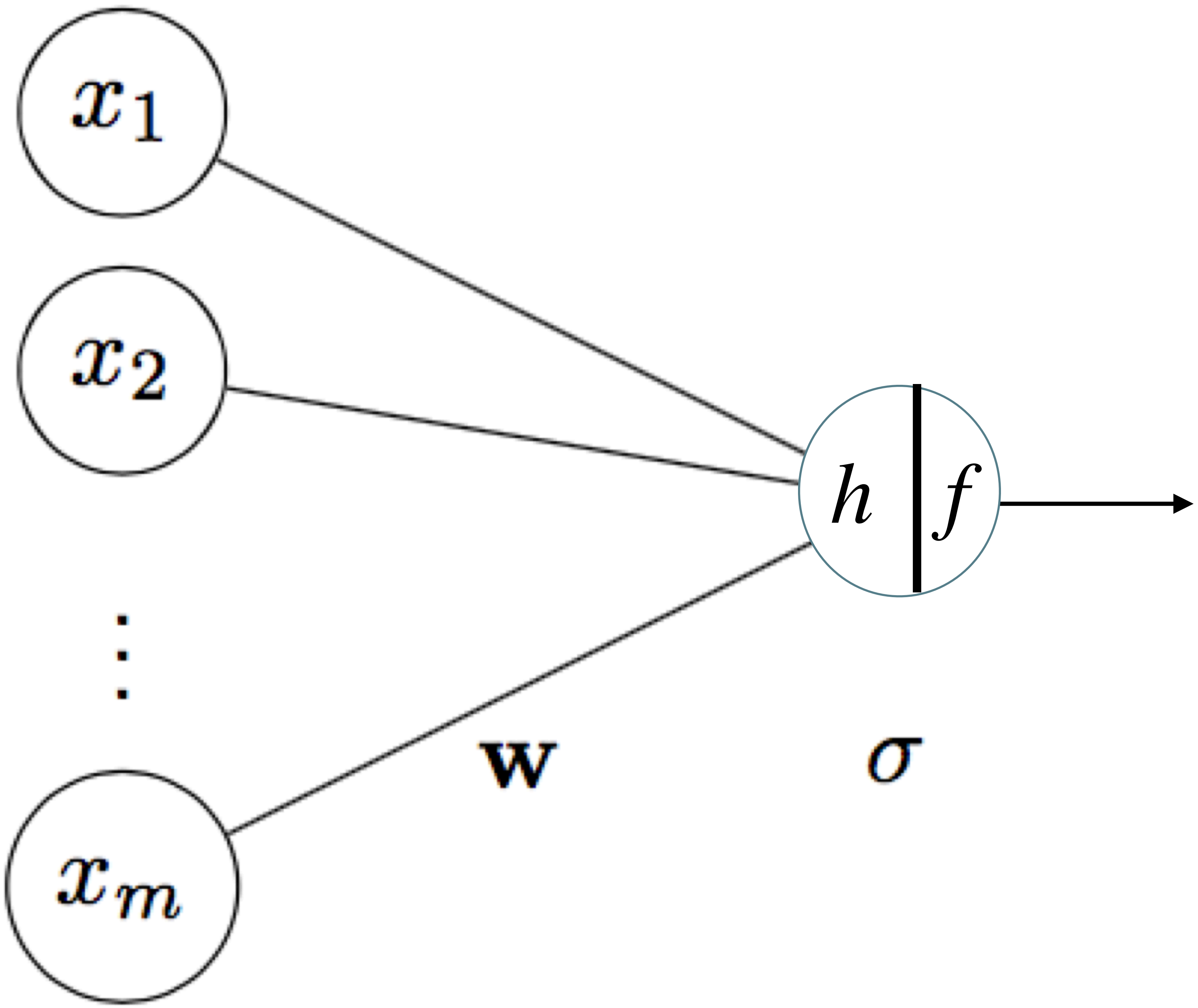
- We can represent Logistic regression as



$$\sigma(h(\mathbf{x}, \mathbf{w})) = \frac{1}{1 + e^{-h(\mathbf{x};\mathbf{w})}}$$

Take inspiration from neurons

- Lets introduce some non-linearity using an additional function



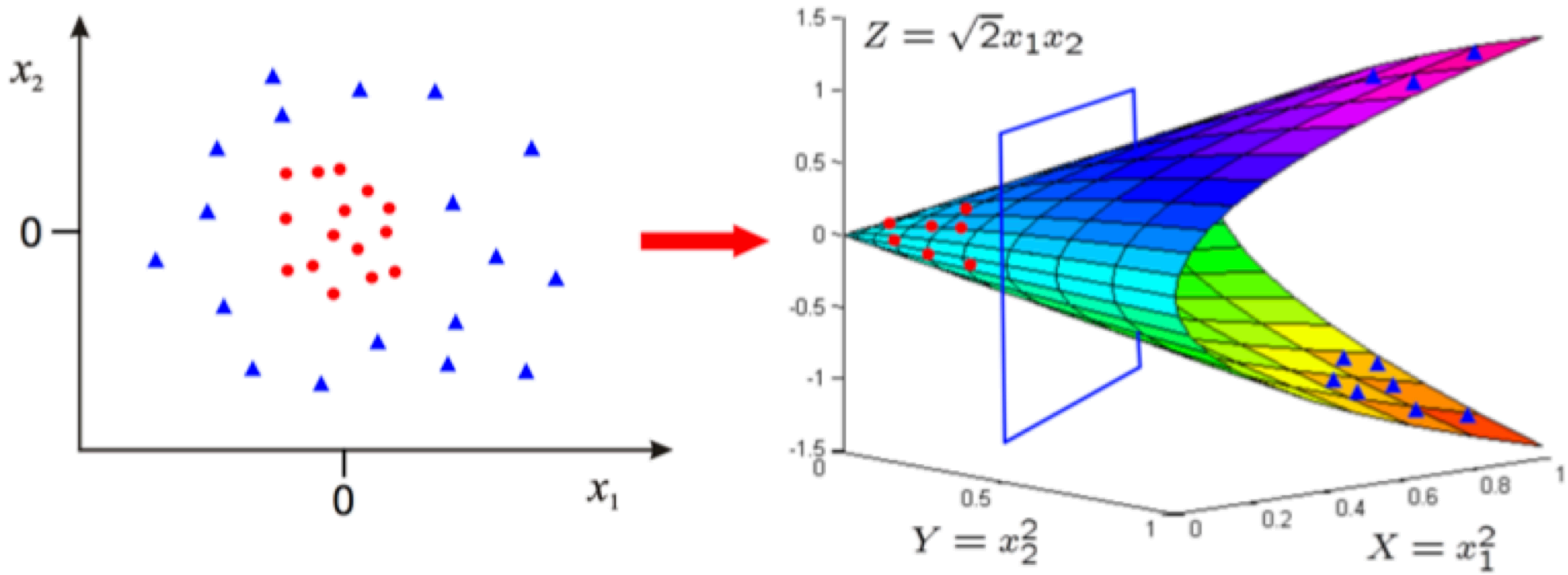
$\sigma\left(\ f(W^T X + b)\ \right)$

f :"Activation" function

Why care about non-linearity ?

- We might require a non-linear decision boundary
  - How do we pick the set of $\phi(x)$? | $\phi(x) \sim \{x^2, sin(x), \dots\}$

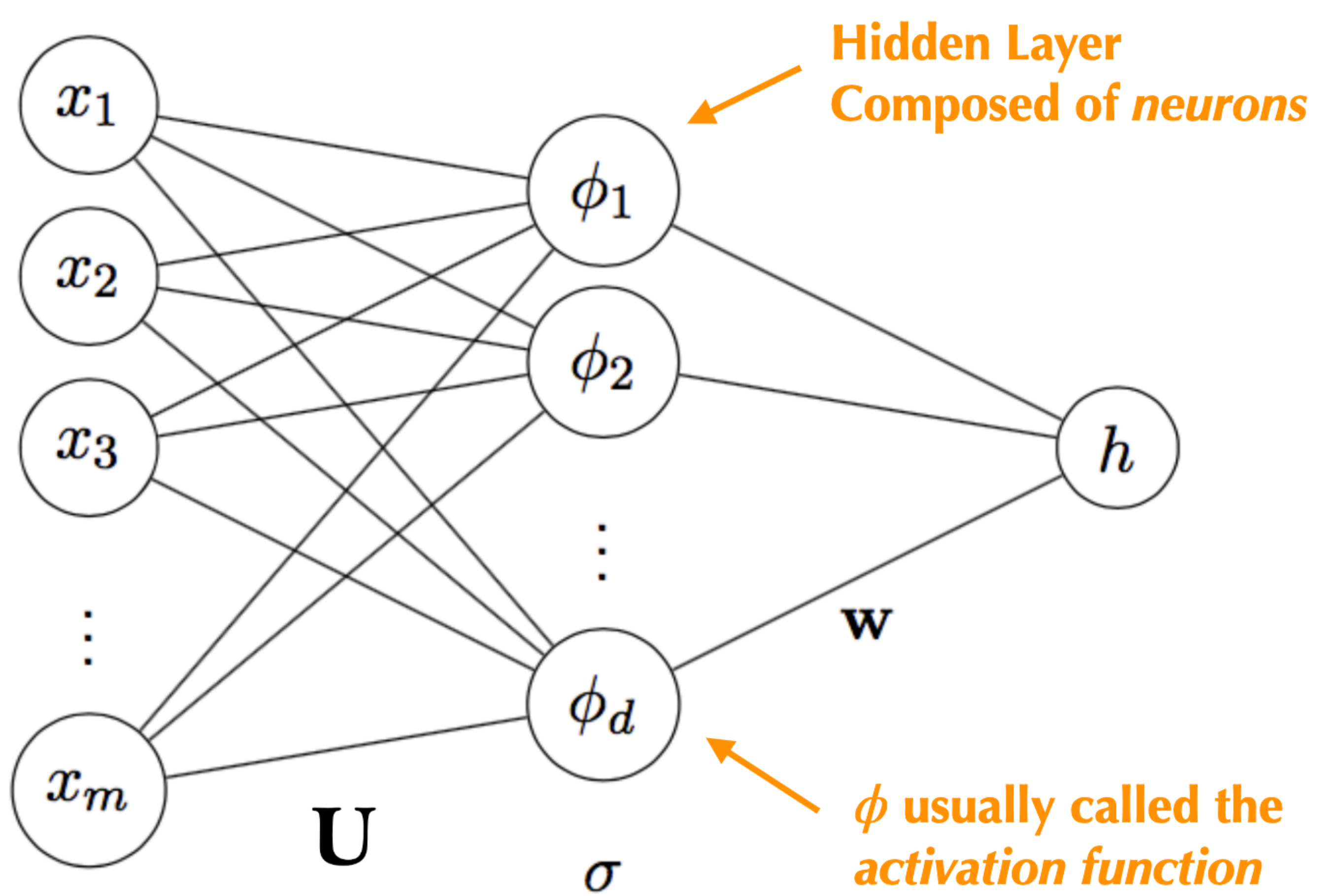
$$\Phi : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \end{pmatrix} \qquad \mathbb{R}^2 \rightarrow \mathbb{R}^3$$



More non-linearity !

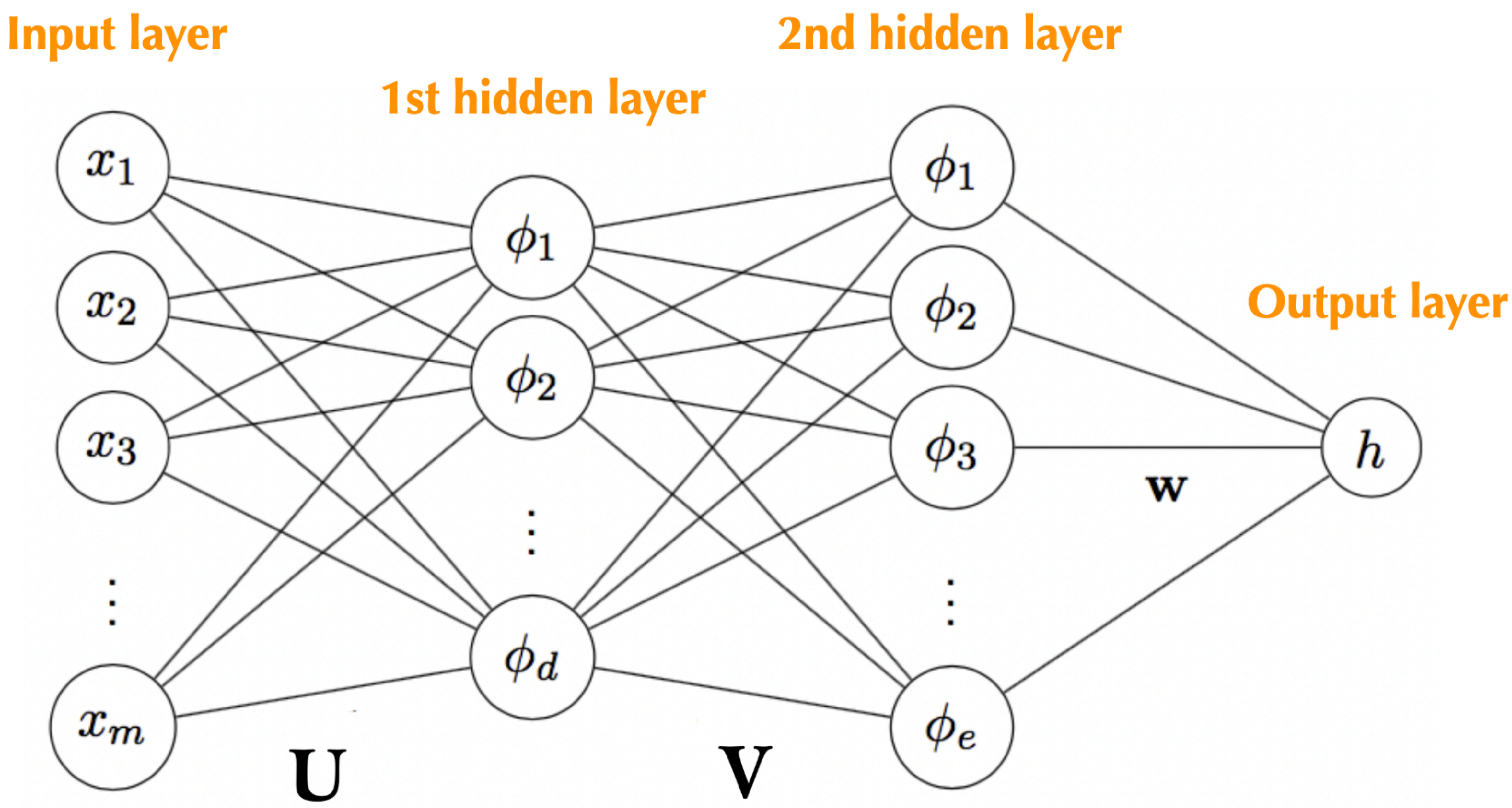
- How do we pick the set of basis functions $\phi(x)$?

- We can learn the basis functions data !
  - We can define the basis functions: $\phi(x; U) : \begin{bmatrix} \sigma(\mathbf{u}_1^\mathrm{T}\mathbf{x}) \\ \sigma(\mathbf{u}_2^\mathrm{T}\mathbf{x}) \\ \vdots \\ \sigma(\mathbf{u}_d^\mathrm{T}\mathbf{x}) \end{bmatrix} \mid \mathbb{R}^m \to \mathbb{R}^d$
  - Now the model is $h(x; U, W) = W^T\phi(x; U)$



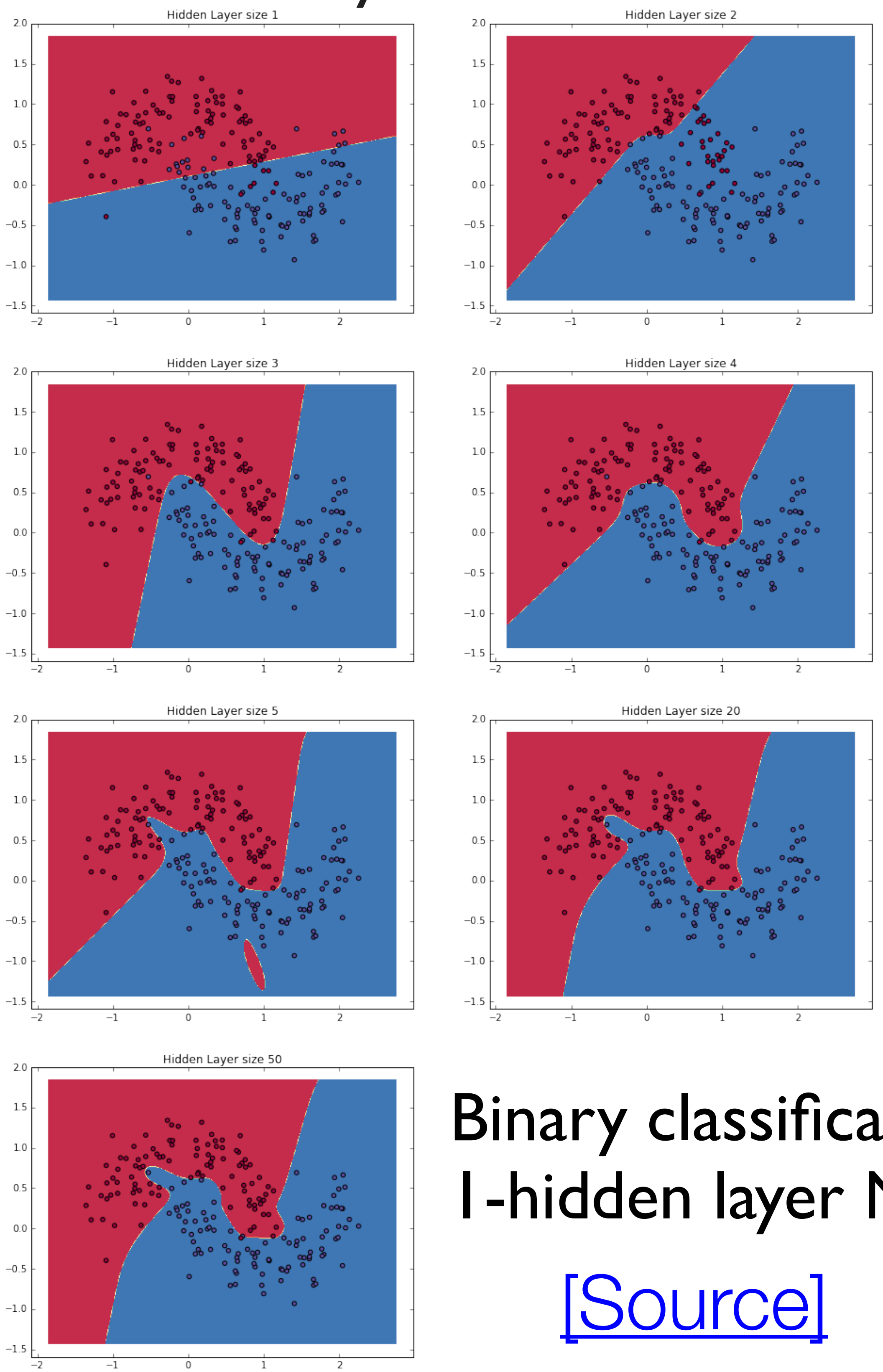
Why stop there ?

- Now we have a "Deep Neural Network"
  - This is what we call it as the *multi layer perceptron* (**MLP**)



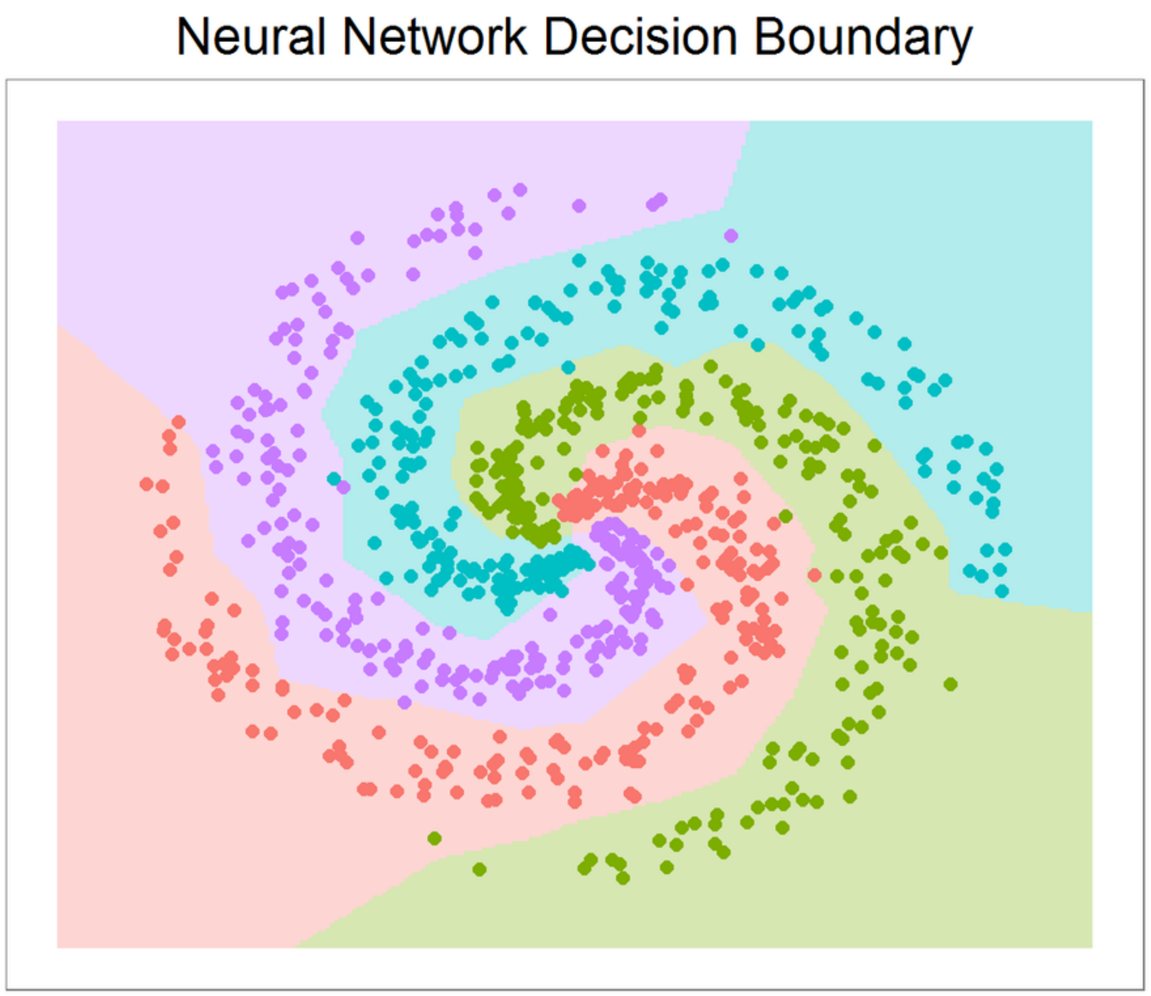
Who do we get ?

- Non-linearity from the MLPs



Binary classification
1-hidden layer NN

[Source]



4-class classification
2-hidden layer NN

[source]

Universal Approximation Theorem

(Feed-forward) NN with a single hidden layer containing a finite number of neurons can approximate continuous functions arbitrarily well on a space

- Only simple assumptions on activation functions
- But no other information are added on how many neurons needed, or how much data!
- How to find the parameters, given a dataset, to perform this approximation?

Universal Approximation Theorem

Fermilab

(Feed-forward) NN with a single hidden layer containing a finite number of neurons can approximate continuous functions arbitrarily well on a space

- Only simple assumptions on activation functions
- But no other information are added on how many neurons needed, or how much data!
- How to find the parameters, given a dataset, to perform this approximation?

Backpropogation !

Optimizing the NNs

- To begin with we need to know the loss or objective to minimize

- For classification: Use cross-entropy

$$p_i = p(y_i = 1|\mathbf{x}_i) = \sigma(h(\mathbf{x}_i))$$

$$L(\mathbf{w}, \mathbf{U}) = -\sum_i y_i \ln(p_i) + (1 - y_i)\ln(1 - p_i)$$

- For regression: Use squared error or something similar

$$L(\mathbf{w}, \mathbf{U}) = \frac{1}{2}\sum_i (y_i - h(\mathbf{x}_i))^2$$

Optimizing the NNs

- We have loss defined, for MLP with many hidden layers

$$L(\phi^a(\ldots\phi^1(\mathbf{x})))$$

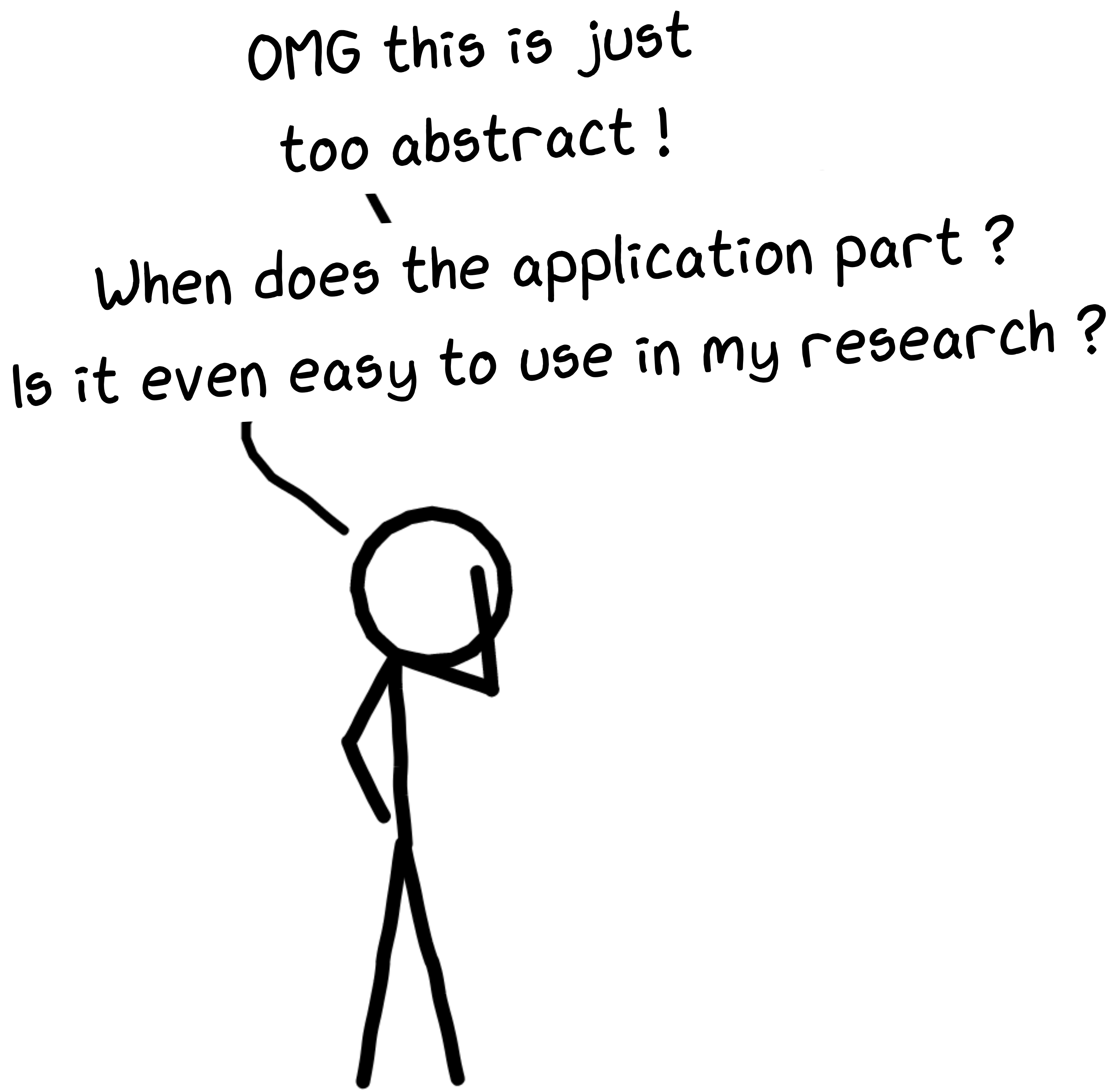
- Forward step / propagation : Compute and save the intermediate hidden layer outputs

$$\phi^a(\ldots\phi^1(\mathbf{x}))$$

- Backward step / propagation: Calculate the derivative with respect to the. input and the hidden layers

$$\frac{\partial L}{\partial \phi^a} = \sum_j \frac{\partial \phi_j^{(a+1)}}{\partial \phi_j^a} \frac{\partial L}{\partial \phi_j^{(a+1)}}$$

- Compute the parameter gradients: $\frac{\partial L}{\partial \mathbf{w}^a} = \sum_j \frac{\partial \phi_j^a}{\partial \mathbf{w}^a} \frac{\partial L}{\partial \phi_j^a}$

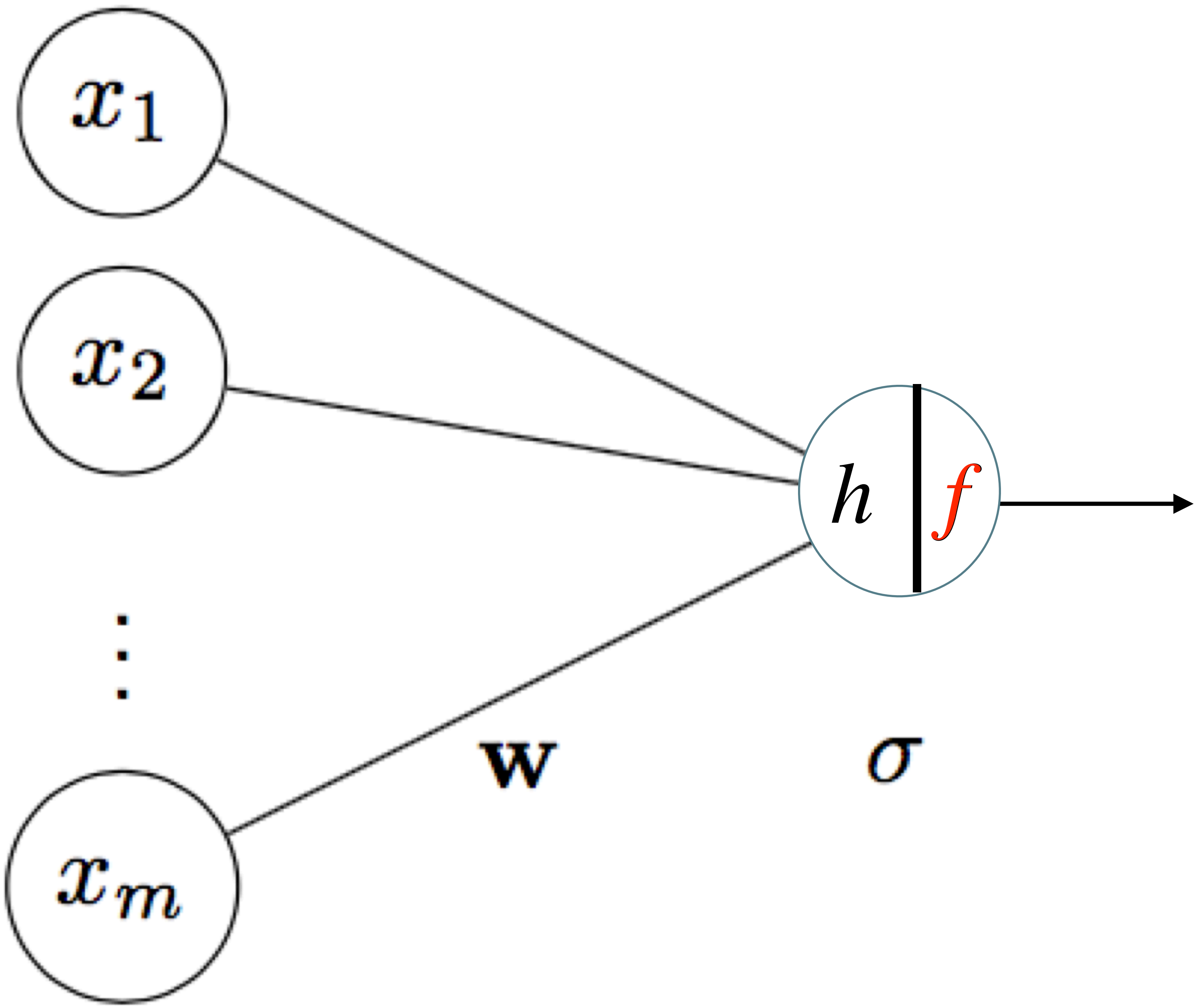

OMG this is just
too abstract !
When does the application part ?
Is it even easy to use in my research ?

We are getting there . . . .

Now let's take another look at everything!

Throwback: Activation functions

- Lets introduce some non-linearity using an additional function

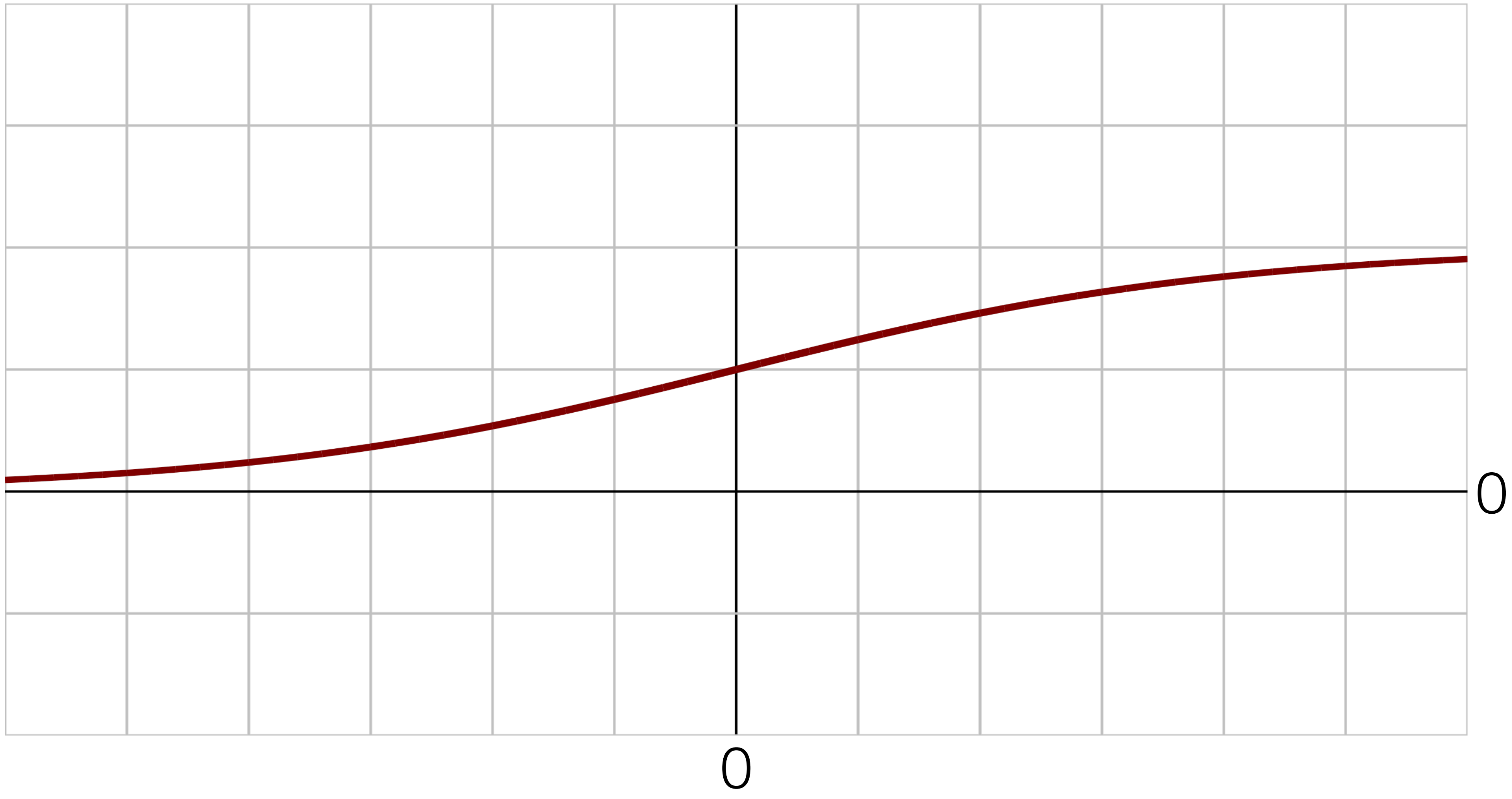


$$\sigma\left(\ f(W^T X + b)\ \right)$$

f : **"Activation" function**

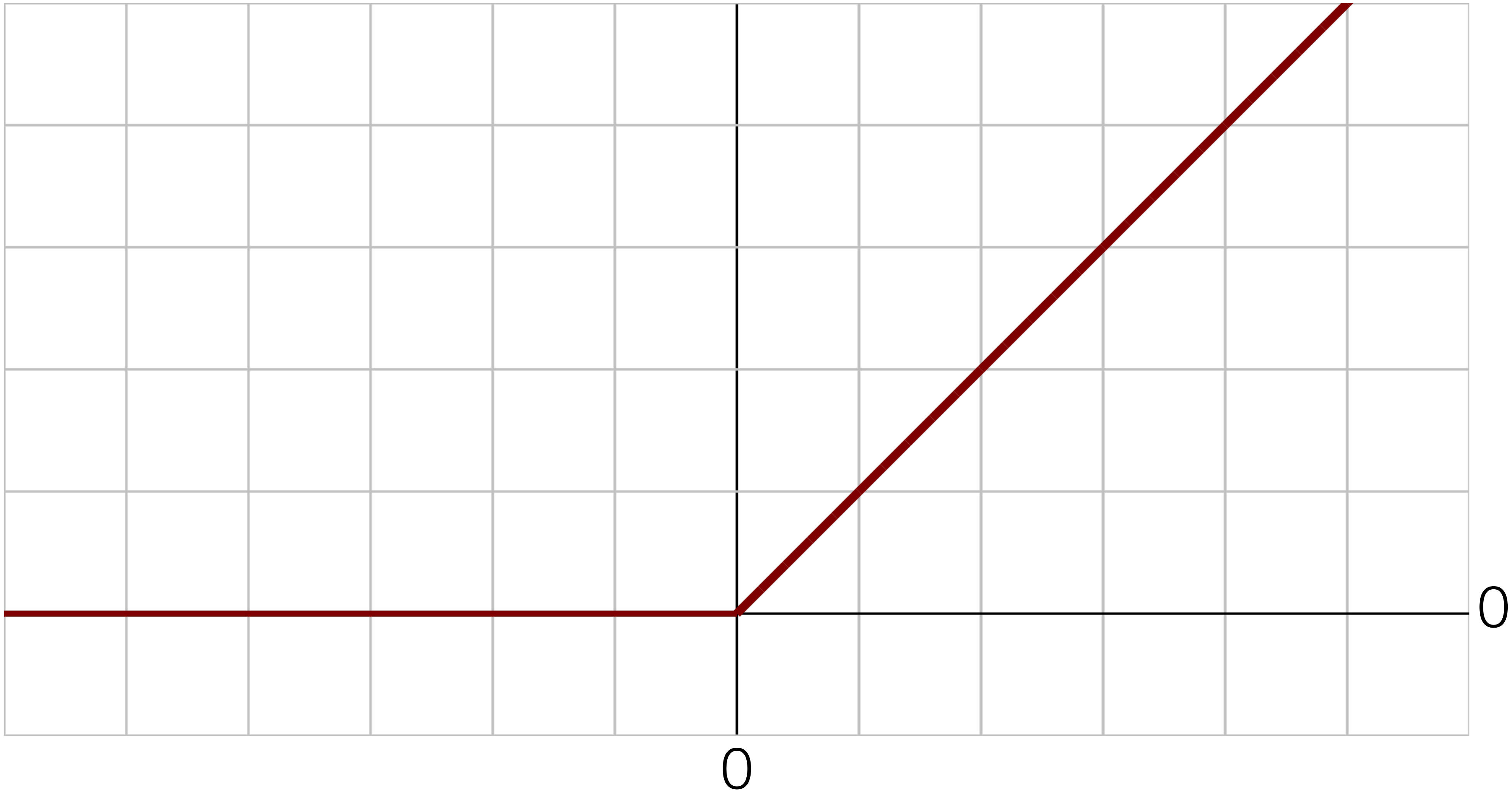
Activation functions

- We could use something like sigmoid as activation (earliest activations)
  - But for values far from 0, gradient vanishes !



Activation functions

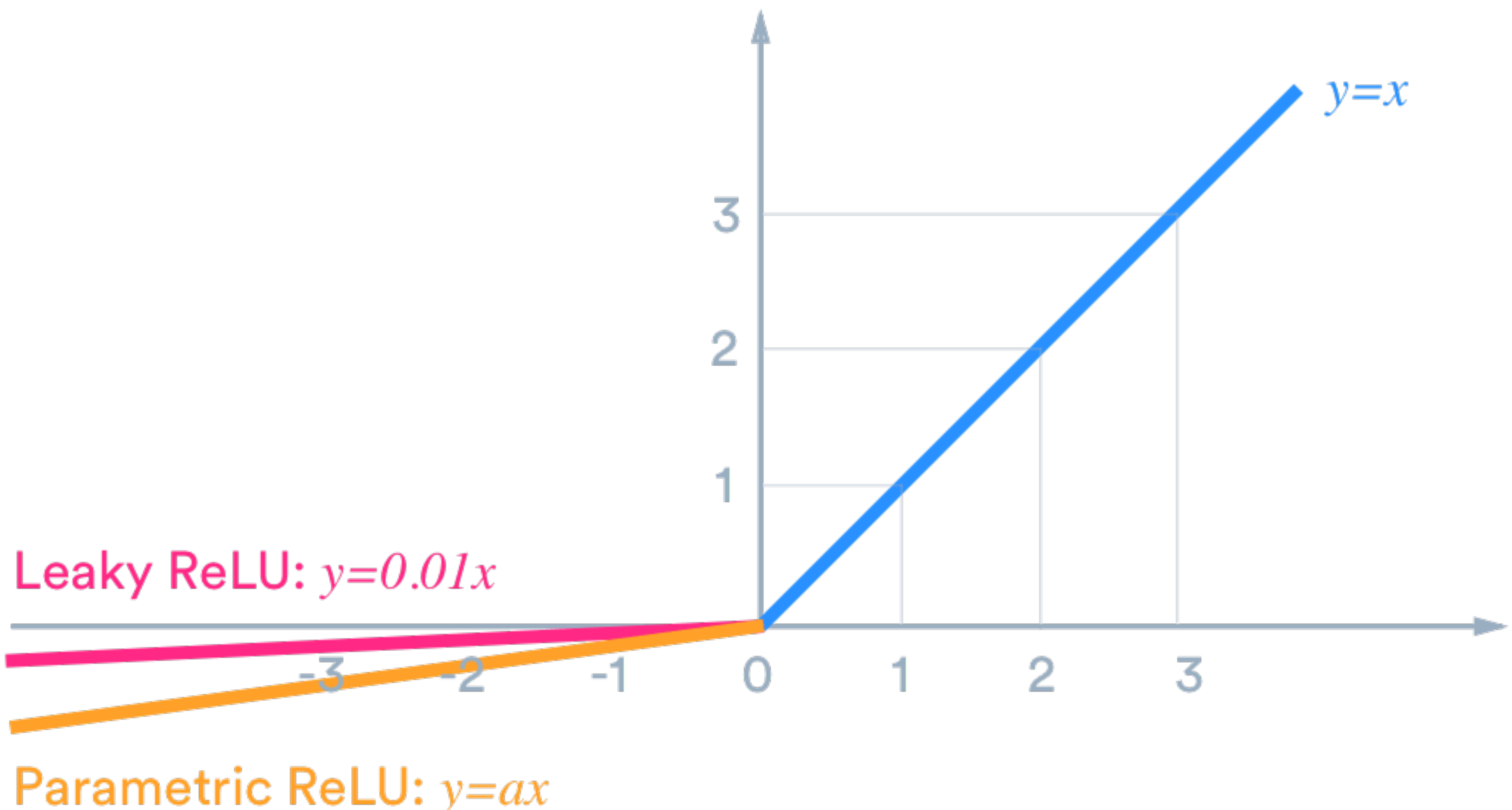
- Alternatively, many modern NNs use Rectified Linear Unit (ReLU)
  - Gradient at 0 is set to 1
  - Gradient ~1 for all positive values, but vanishes for all negative values
    - Useful to induce sparsity in the network !



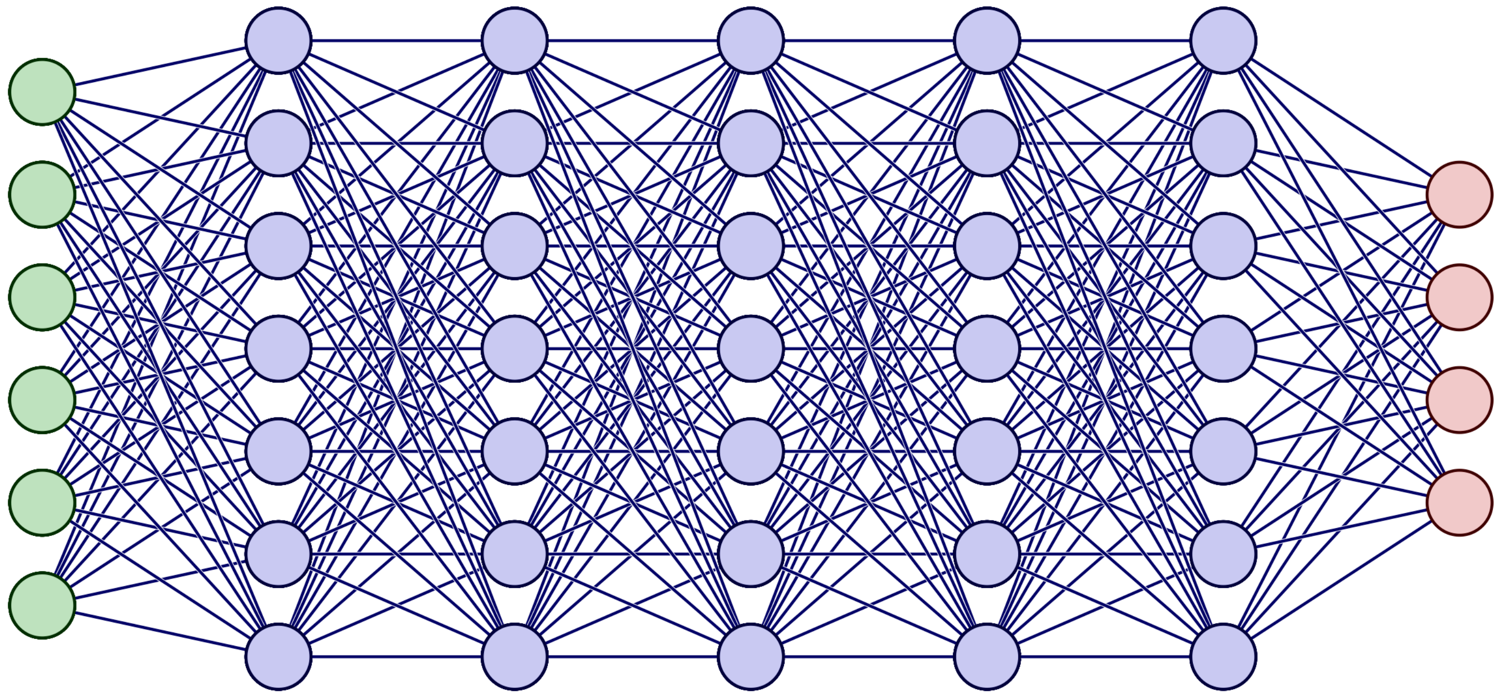
Activation functions

- Sometimes, with bad initialization ReLU can make all of neurons "dead" in the network
  - We could have too much sparsity

- We mitigate this problem with a "Leaky ReLU"



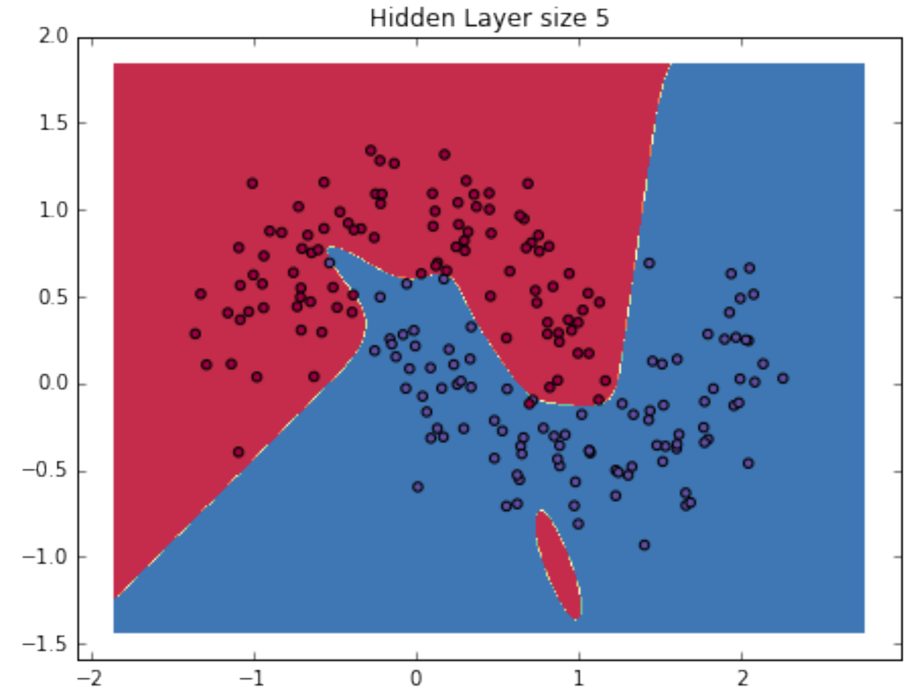
When to use MLPs ?



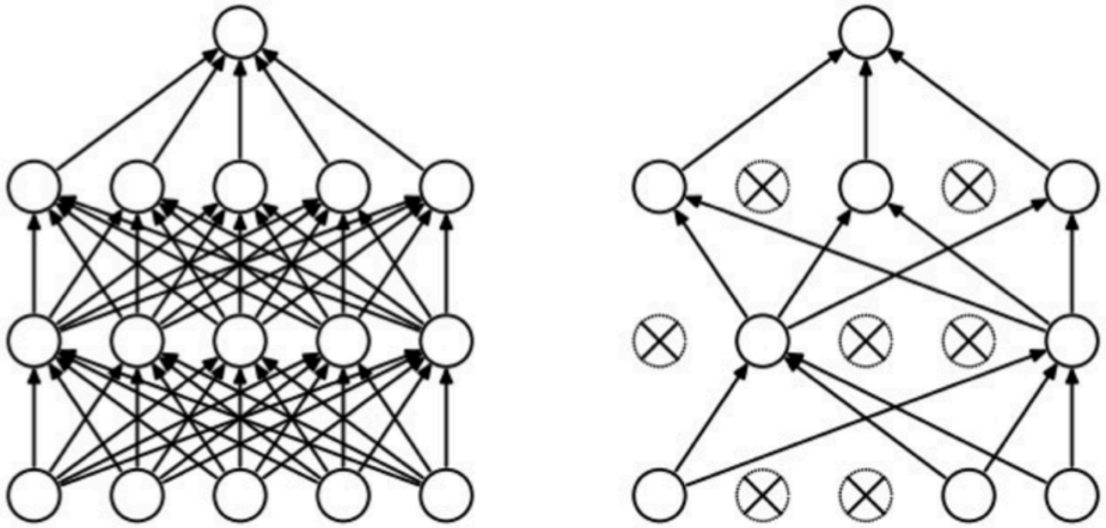
- MLPs: A very generalized way to look at *patterns* in data
  - Not efficient is there is inherent structure that we can use. [e.g: Images]
- Best for distilled inputs or engineered inputs: High-Level features
  - Given sub-structure variables, identifying the jet source
  - Regress the metallicity of the stars from the

Regularization

- DNNs can easily overfit the data !
- We can regularize the network to avoid this problem



- Approach I: L2 regularization
  - Add $||W^2||$ to loss function, avoid large weights saturating network
- Approach II: Drop out / Randomly kill fraction of the nodes during training



(a) Standard Neural Net (b) After applying dropout.

Iterating over the datasets

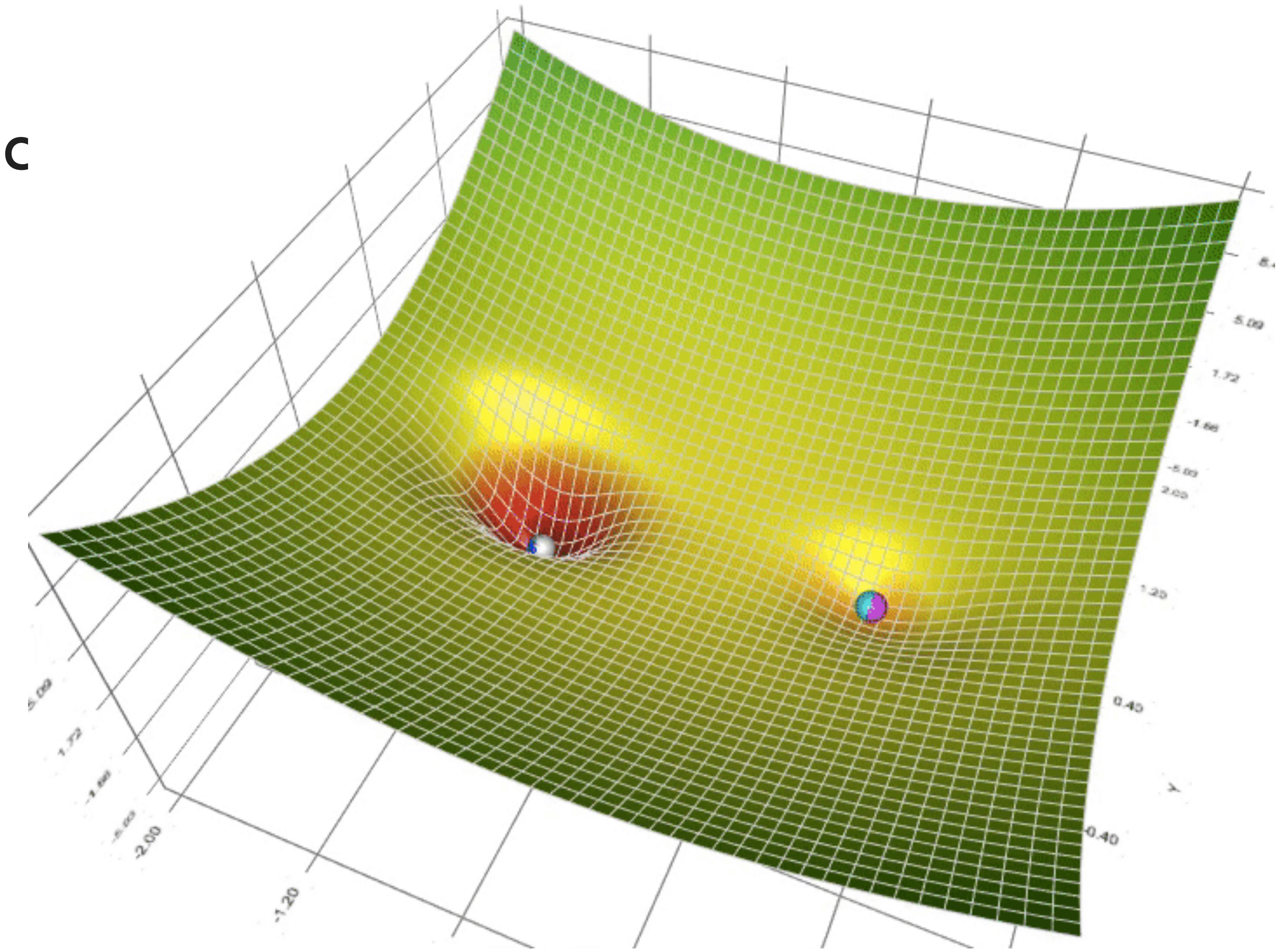
- We have to perform optimization of DNNs until they *converge*
  - How do we do it with limited dataset ?

- We splits the dataset in chunks / batches
  - Compute loss and update the weights with each batch
  - Small batch size results in faster computation but noisy training
  - Large batch size demands more memory, results in sharper gradients

- At the end of one training cycle / epochs, we repeat the process multiple times on the dataset until it reaches convergence

Gradient descent in DNNs

- In training of NNs, we optimize the model paper meters at end of each batch



- So in this case we use the Stochastic Gradient Descent
  - Reduces the very high computational burden
- The most widely adapted method is called *ADAM*
  - Uses momentum fraction of the previous update is added to the current
  - Helps achieve faster convergence of the network

Best practices for best performance

- Make sure that data has no *nan* / *inf* or any unphysical values
  - Many way to take care of them !

- For better classification, standardize the input dataset
  - Typically good for the input features to have $\mu \sim 0, \quad \sigma \sim 1$
  - Backpropagation and activation function don't explicitly require it
  - Helps for a faster and better convergence

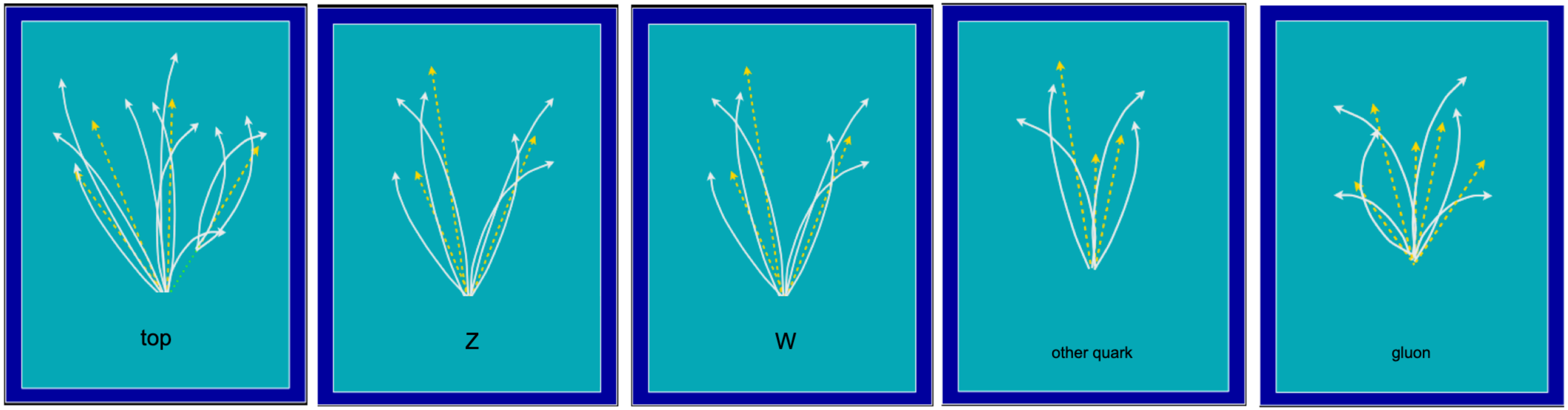
- Check performance and overfitting w/ validation dataset at end of each epoch

- Perform training with multiple seeds, ensure you reach a robust minimum


C'mon,
do something...

Exercise problem

- Identification of jets arising from hadronization of boosted W/Z/H/top
  - A key and important task in high energy physics
  - Analytical *sub-structure*(s) variables contain information about hadronization
    - We are using MLPs to approximate $f(S) \rightarrow$ Jet Flavor



t→bW→bqq	**Z→qq**	**W→qq**	**q/g background**
3-prong jet	2-prong jet	2-prong jet	no substructure and/or mass ~ 0

Reconstructed as one massive jet with substructure

Training dataset

- Input:
    - Various substructure variables of jets
- Objective:
    - Tagging the origin of the jet
- Explore the dataset and get the best performance possible !

Tools for ML

Fermilab

TensorFlow / Keras

- Easy to get started
- Best for simple operations
- Lot of Built-in Fn & documentation
- Hard to customize

PyTorch / PyTorch Lightning

- Also has has lot of libraries
- Very easy to customize
- Needs more lines of code compared to Keras
- Memory efficient

JAX / FLAX

- Extremely versatile
- Can do beyond NNs, use it like accelerated numpy
- Performes Autograd

Tools for ML

Fermilab

TensorFlow

Keras

- Easy to get started
- Best for simple operations
- Lot of Built-in Fn & documentation
- Hard to customize

PyTorch

PyTorch Lightning

- Also has has lot of libraries
- Very easy to customize
- Needs more lines of code compared to Keras
- Memory efficient

- Extremely versatile
- Can do beyond NNs, use it like accelerated numpy
- Performes Autograd

What to do ?

- Identify the best features possible for this task
- Optimize the hyper parameters: learning rate, batch size, Droup out
- Change the architecture, make the network deeper and wider
- Can you plot Signal vs BKG ROC curves ?
    - QCD [Quark/gluon jets] is the background
- Can you look up TF/Keras API and implement weight initialization ?

Callbacks

- Try implementing the callbacks in the network.
  - Reduces the learning rate when the model is getting saturated
  - Stop the training before the model overfits the data
- Refer to Keras API and implement them.
- Has it improved in faster convergence ?

Bayesian optimization

- In a NN / model optimization, we are extremizing a *objective function / loss*

- For a given set of hyper parameters, we have best loss after training

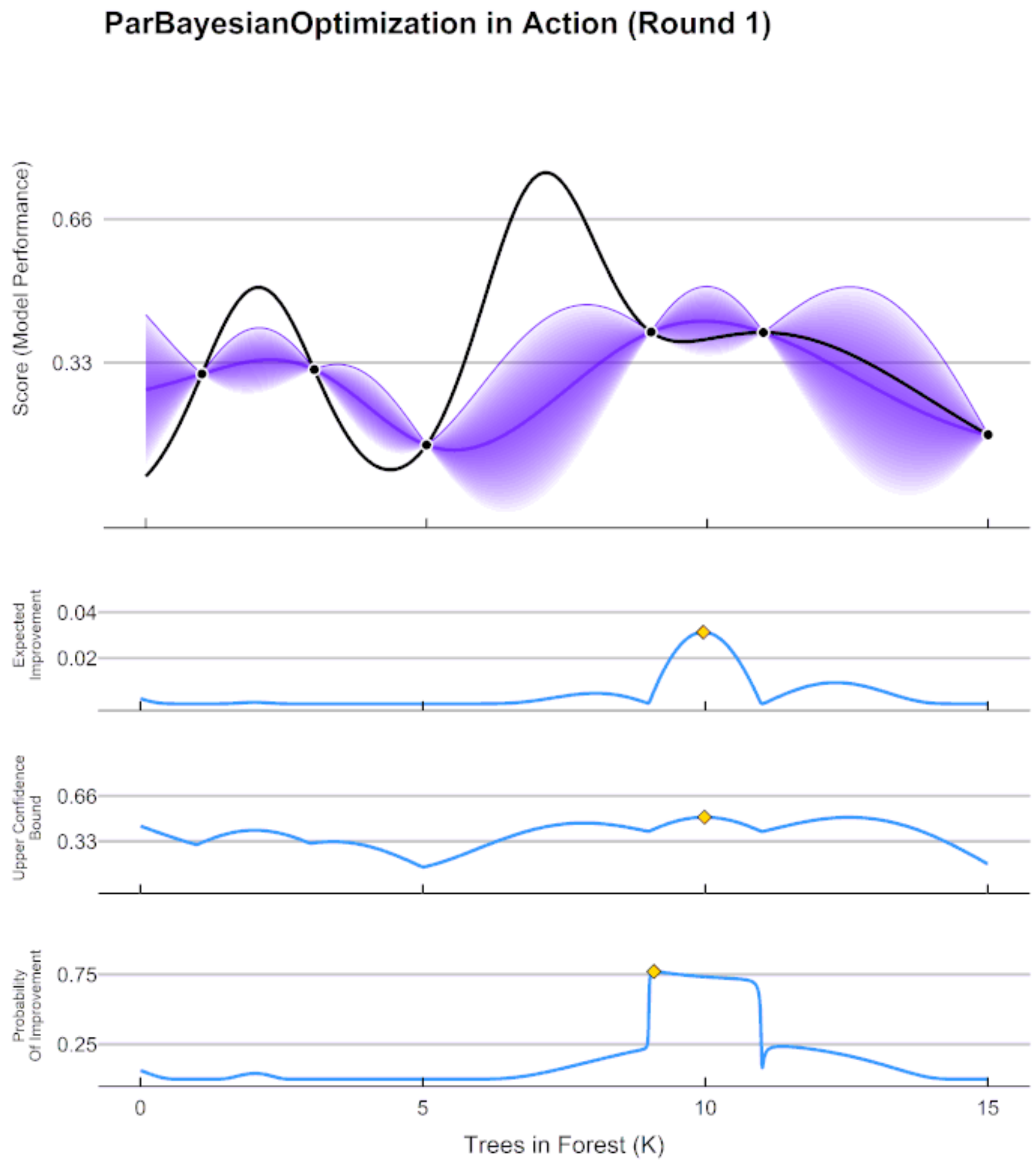
    - Gaussian Process to X (x_1,x_2,..; hyper-parameters), Y (*objective function / loss*)

    - From GP prediction, check where we'd have a extrema from this fit w/ some certainty

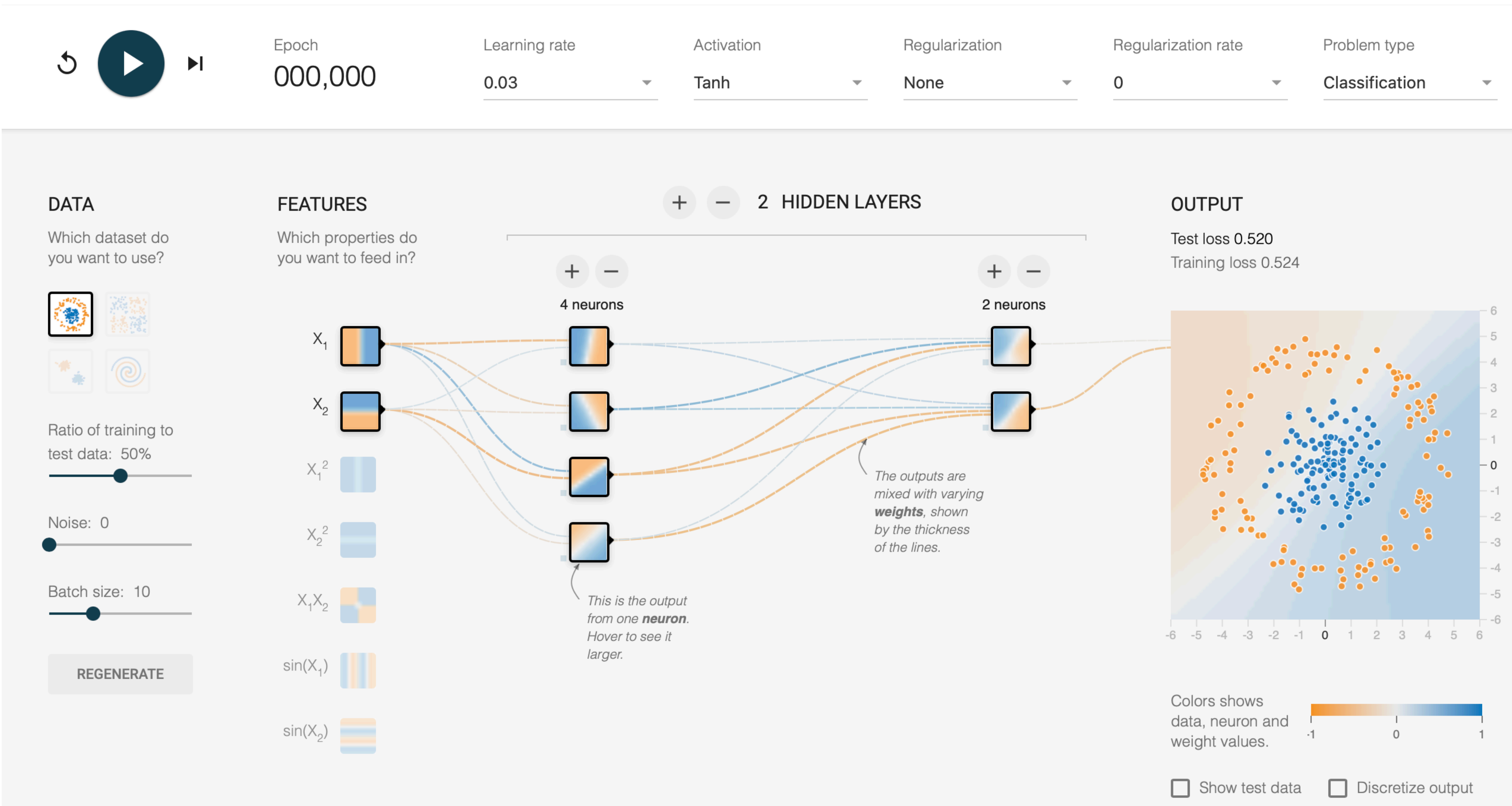
    - Try that point and repeat !

- We map out the for the *objective function* space of HPs

- Try this feature using the Keras Tuner etc . . .



Need Intuition ?



Try : playground.tensorflow.org

Logging your experiments

- Done with exercises ?
  - Can you track your experiments with WandB ?
    - Like GitHub, but you NN weights and tracking multiple trainings
    - https://docs.wandb.ai/tutorials
  - Log your experiments in the WandB
    - Modify the notebook to use WandB logging API
    - Do you see a preference of hyper parameters
  - Launch multiple experiments