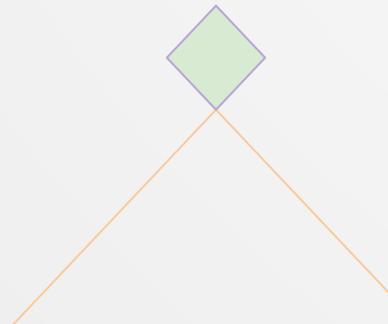


# Deep Learning

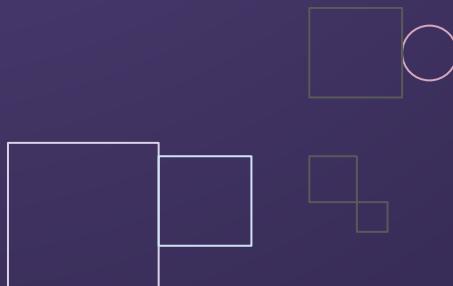
Introduction Series



**Bryan Cardenas**  
Caspar van Leeuwen  
Monica Rotulo  
Robert Jan Schlimbach



# Prerequisites



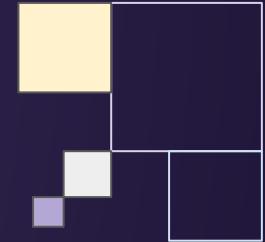
Programming

R / Python

Statistics, Calculus

Machine Learning

Parallel Computing



# Plan for Today

01.

**General Introduction**  
Machine Learning

02.

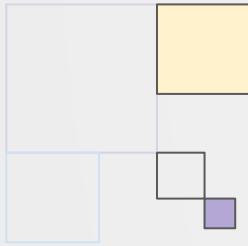
**Neural Networks**  
Dense and Convolutions

03.

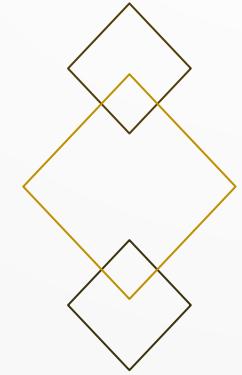
**Language Modelling**  
Recurrence and Transformers

04.

**High Performance**  
How can I train efficiently?



# Machine Learning



High Performance Machine Learning Group

SURF

# What ML is *not*:



Mimicking human intelligence

Robotics

Deep Learning



SURF



# What ML is *not*:



Mimicking human intelligence

Robotics

Deep Learning



“ ML is the study of algorithms that can improve through experience and by the use of data. It is seen as part of Artificial Intelligence

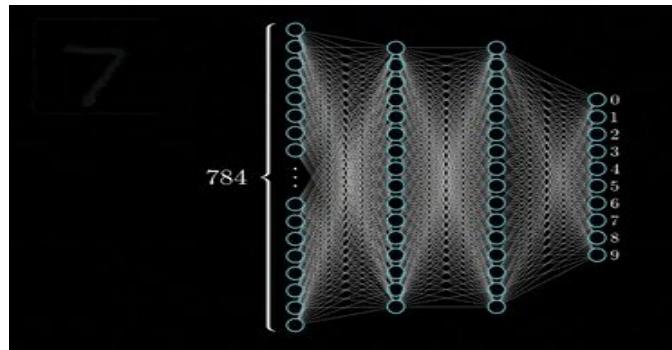
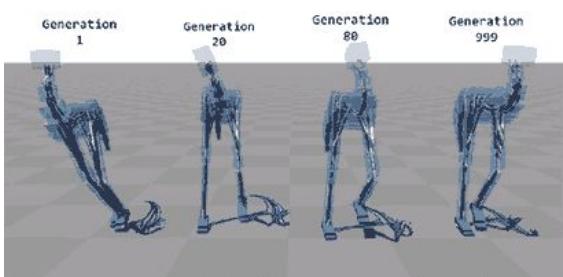


~ Wikipedia

SURF

# Artificial Intelligence

Having computers to exert  
Intelligent behaviour

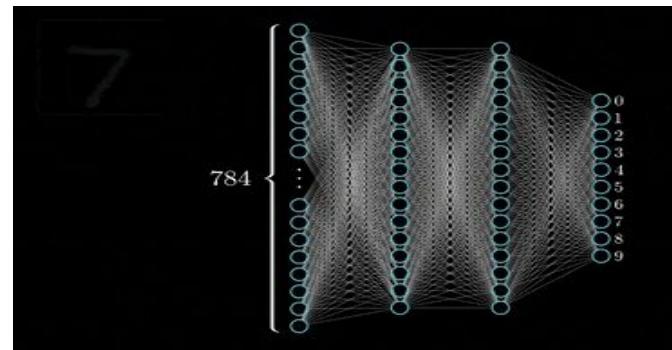
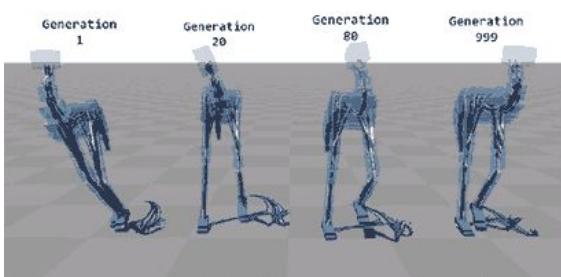
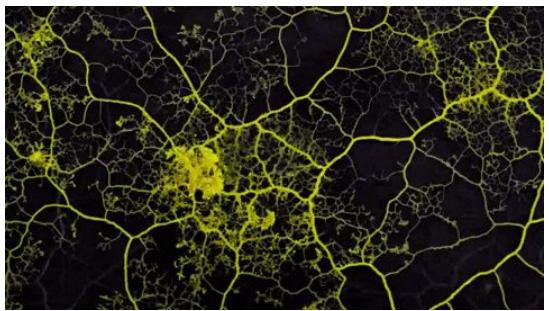


## Artificial Intelligence

Having computers to exert  
Intelligent behaviour

## Machine Learning

Perform tasks without  
Explicitly programmed  
from data



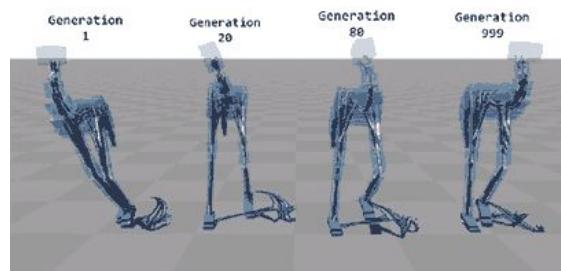
## Artificial Intelligence

Having computers to exert Intelligent behaviour



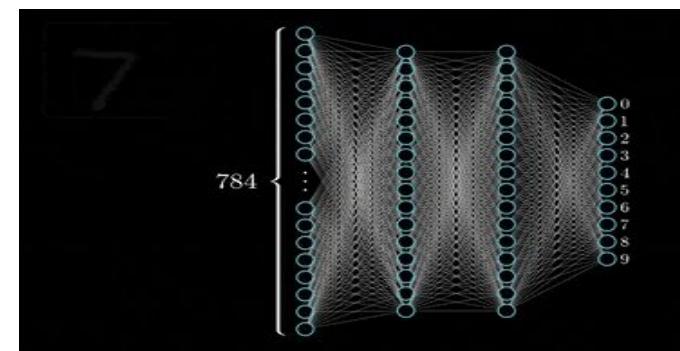
## Machine Learning

Perform tasks without Explicitly programmed from data



## Deep Learning

Use deep neural networks

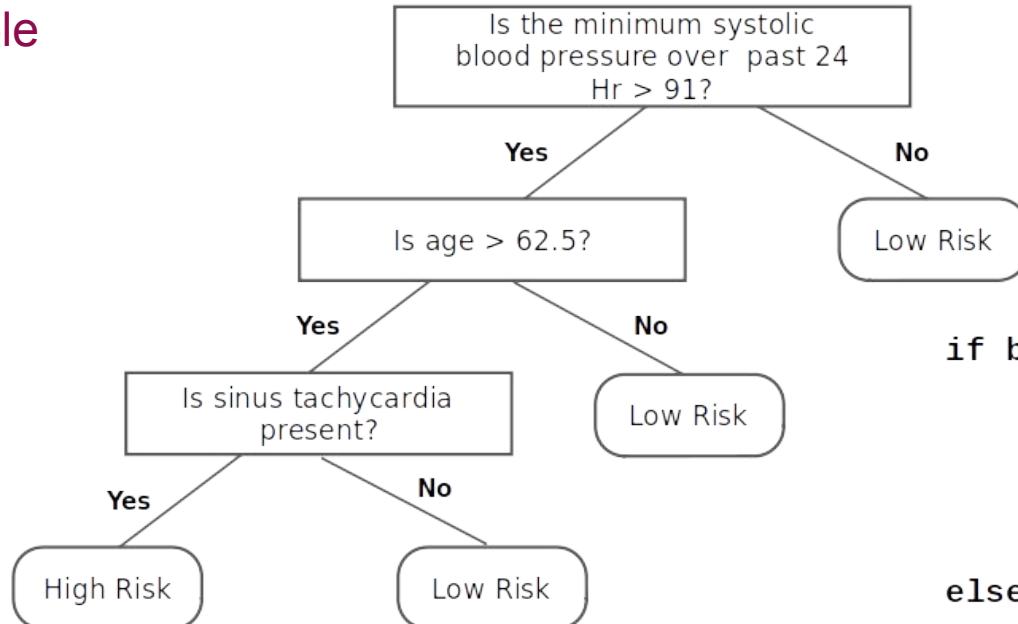


# Why Machine Learning?

Think of a simple  
**decision tree**

# Why Machine Learning?

Think of a simple decision tree



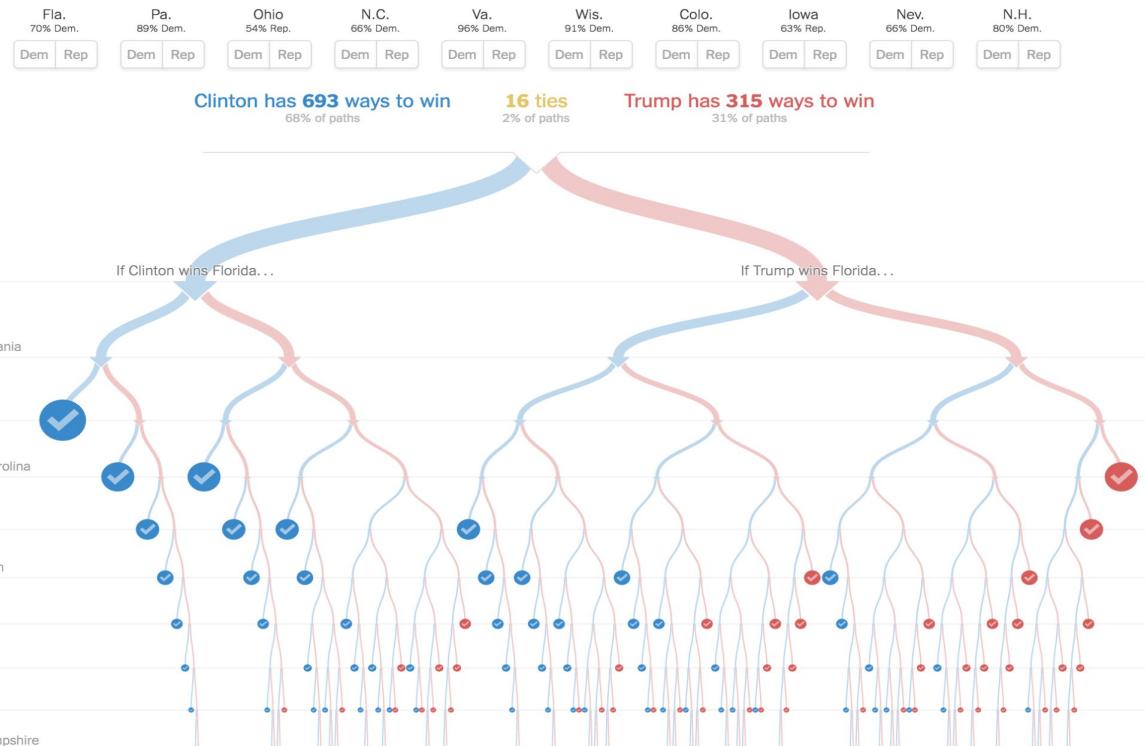
```
if blood_pressure > 91:  
    if age > 62.5:  
        if sinus_tach:
```

...  
else:

...  
else:

# Why Machine Learning?

Think of a *hard* decision tree

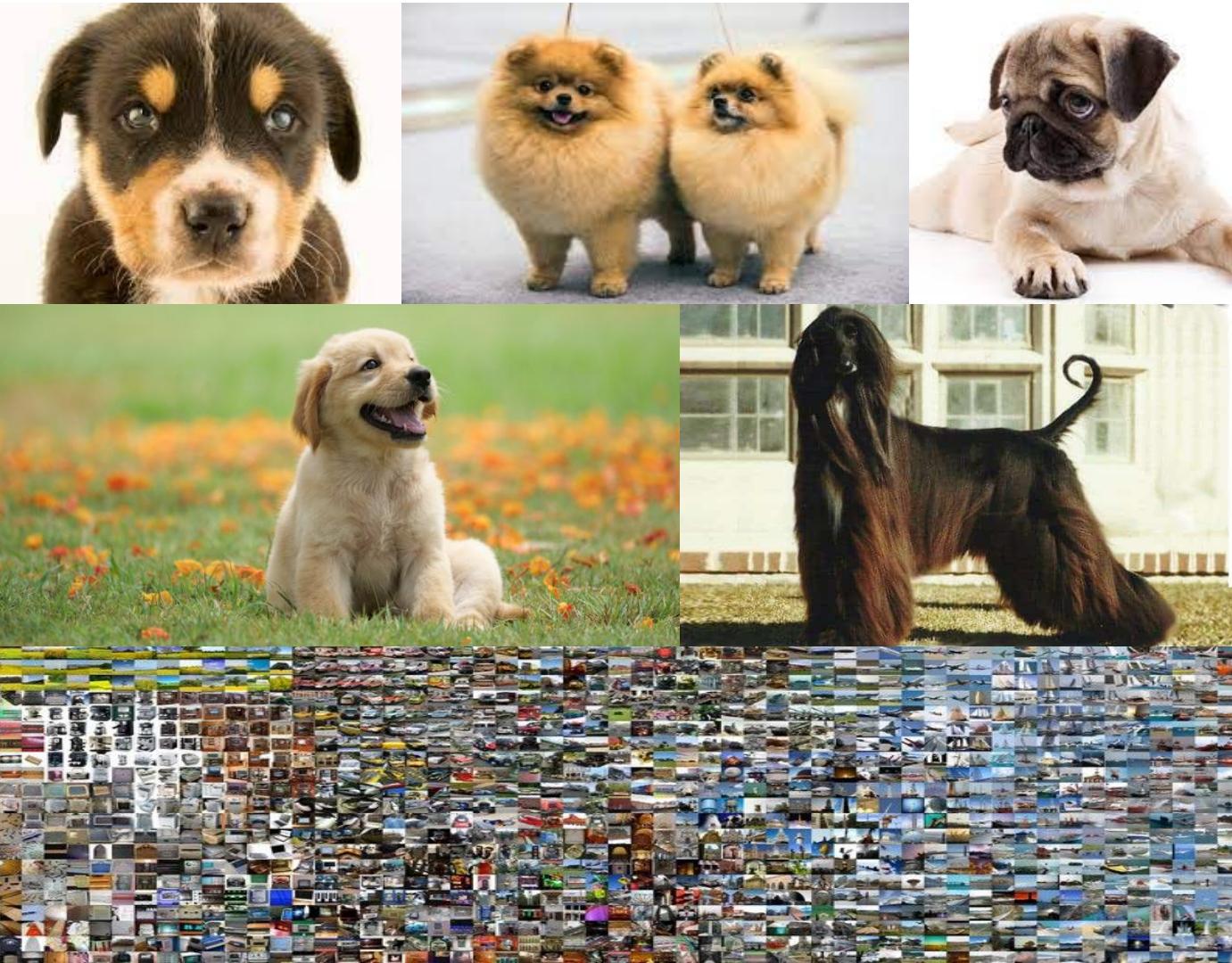


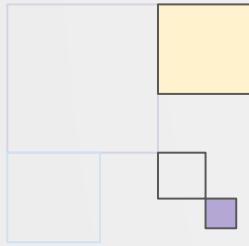
# What is a dog?

**Uncountable** features  
that define a dog

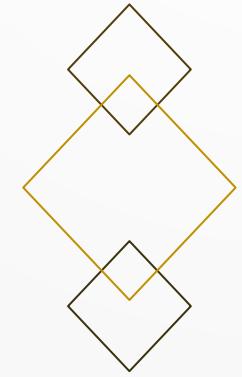
We want an automatic  
way of learning these  
features

Driven by Data





# Neural Networks



High Performance Machine Learning Group



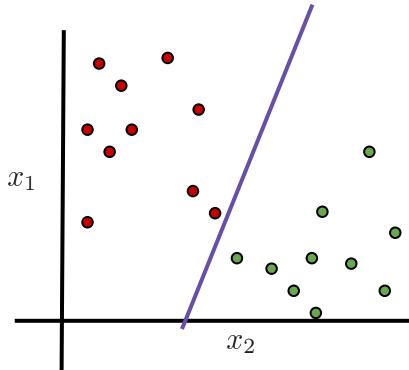
# Categories of Machine Learning

01.

Supervised

Learn from labels

Regression, Classification

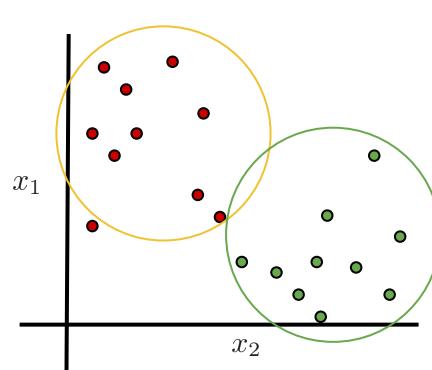


02.

Unsupervised

Detect Patterns in the data

Clustering, Dimensionality Reduction



03.

Reinforcement

Learn from the environment

Control, gaming

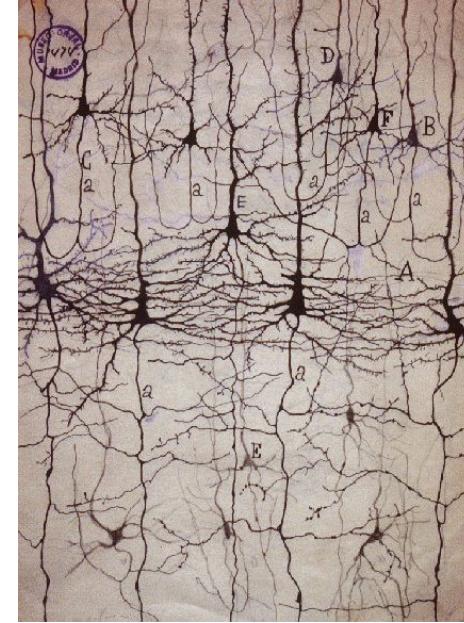
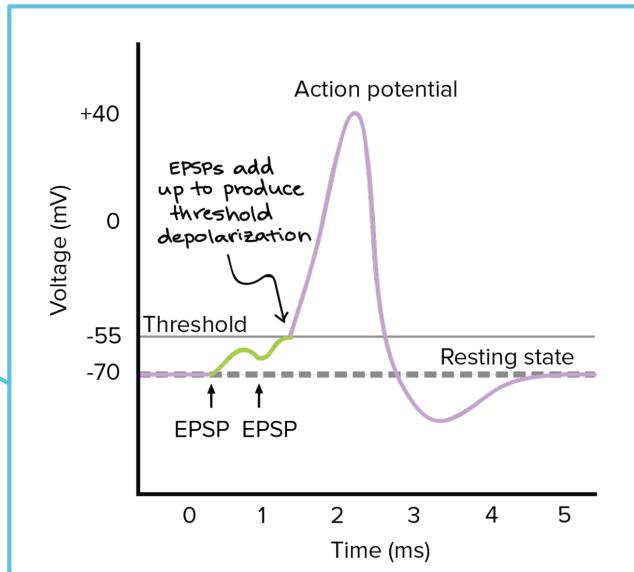
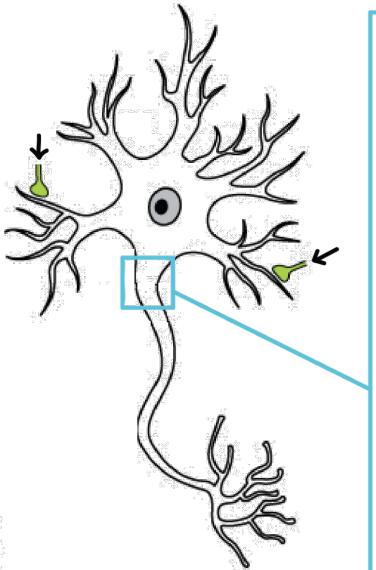


SURF

# Biological Neuron

A neuron inhibits or excites a signal picked up from its receivers

Only fires if a threshold is reached and is connected to thousands of others.

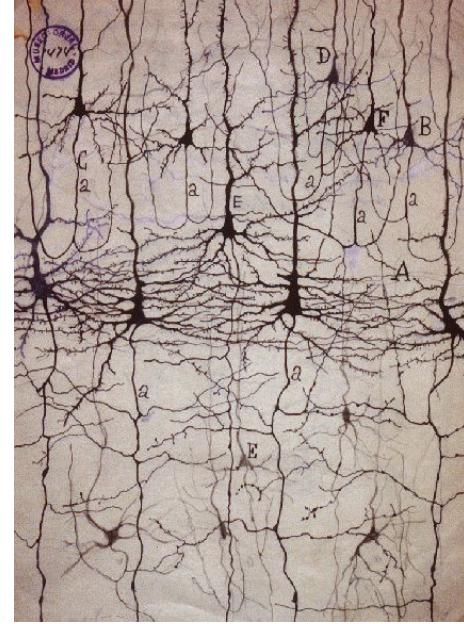
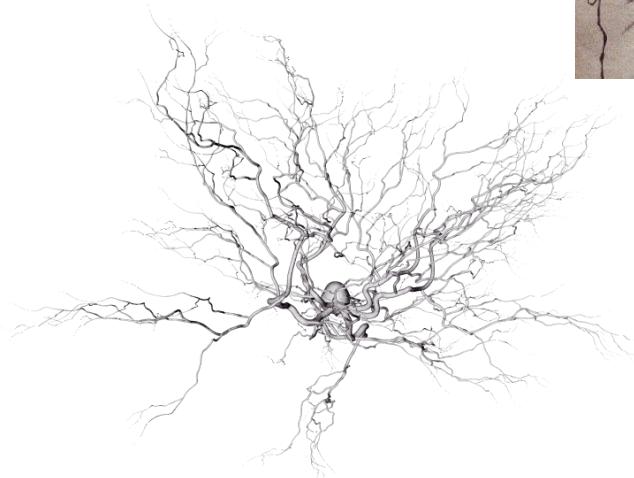
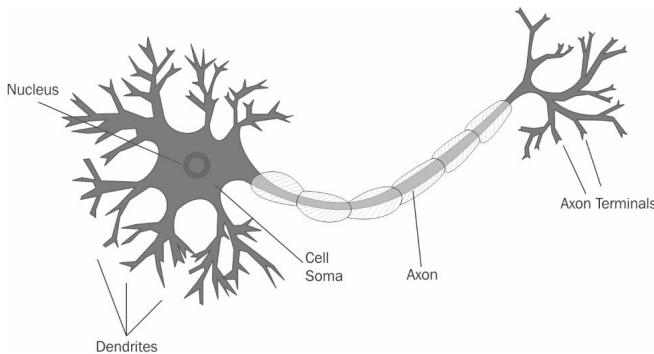


# Biological Neuron

A neuron inhibits or excites a signal picked up from its receivers

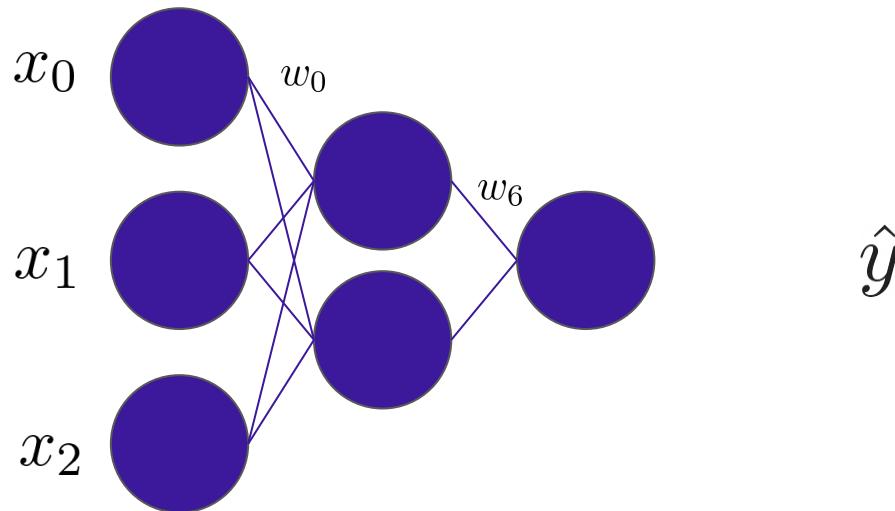
Only fires if a threshold is reached and is connected to thousands of others.

Humans have around 80 billion neurons and trillions of connections



# Anatomy of an Artificial Neural Network

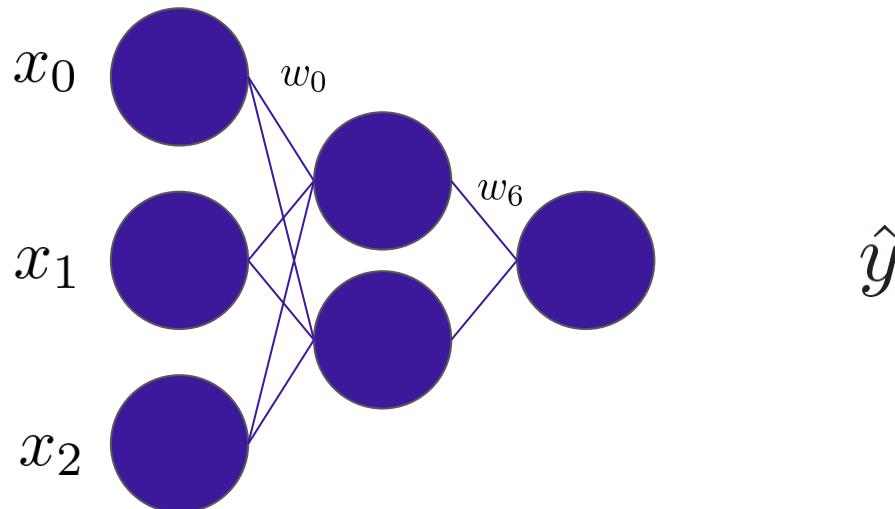
Don't model the biological  
neuron **precisely**



# Anatomy of an Artificial Neural Network

Don't model the biological neuron **precisely**

- Inputs
- Bias
- Weights
- Dot product
- Non-linear activation

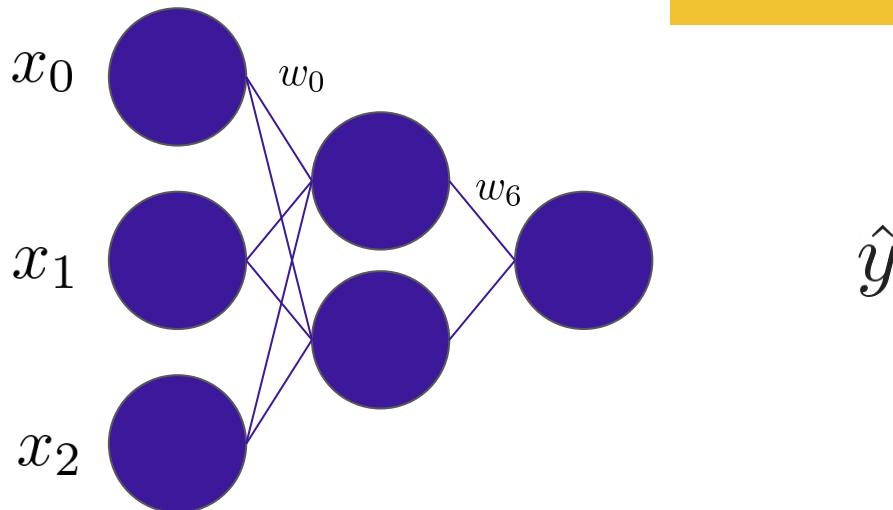


# Anatomy of an Artificial Neural Network

Don't model the biological neuron **precisely**

- Inputs
- Bias
- Weights
- Dot product
- Non-linear activation

Use a (deep) neural network to **approximate** an unknown function

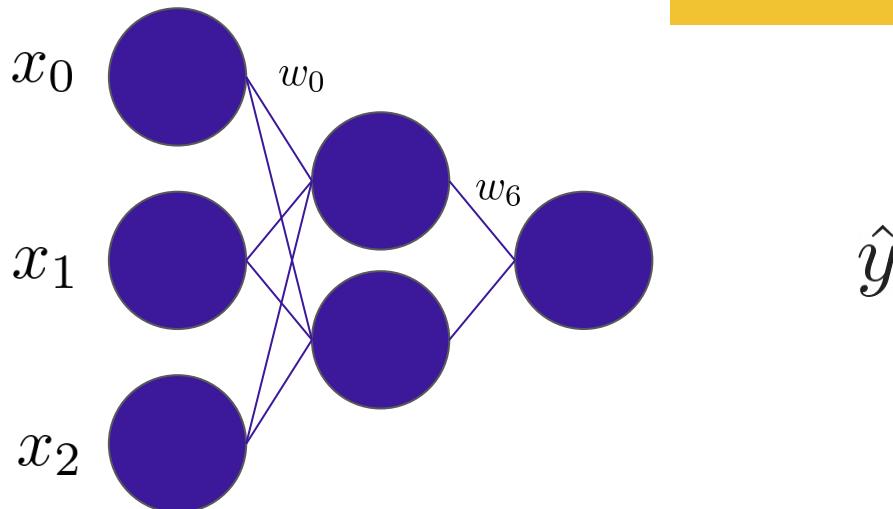


# Anatomy of an Artificial Neural Network

Don't model the biological neuron **precisely**

- Inputs
- Bias
- Weights
- Dot product
- Non-linear activation

Use a (deep) neural network to **approximate** an unknown function



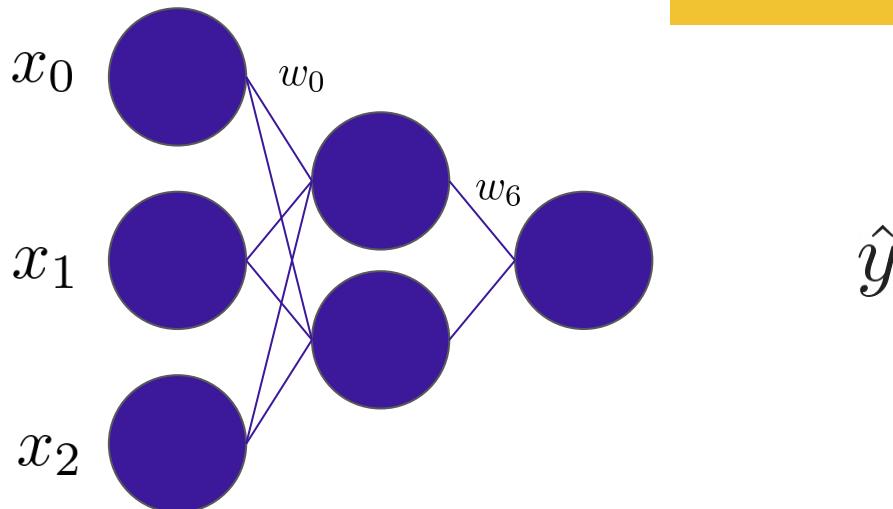
$$\sum_{i=1}^m w_i x_i + b$$

# Anatomy of an Artificial Neural Network

Don't model the biological neuron **precisely**

- Inputs
- Bias
- Weights
- Dot product
- Non-linear activation

Use a (deep) neural network to **approximate** an unknown function



$$\sum_{i=1}^m w_i x_i + b$$

**Wx**



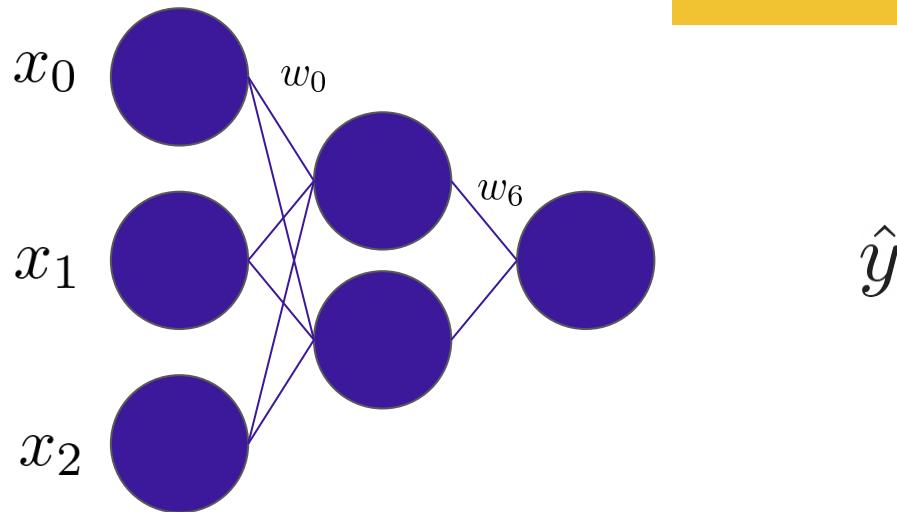
# Anatomy of an Artificial Neural Network

Don't model the biological neuron **precisely**

- Inputs
- Bias
- Weights
- Dot product
- Non-linear activation

- ❖ Easy to **compose** and easy to **vectorize**
- ❖ Fits current compute paradigm

Use a (deep) neural network to **approximate** an unknown function



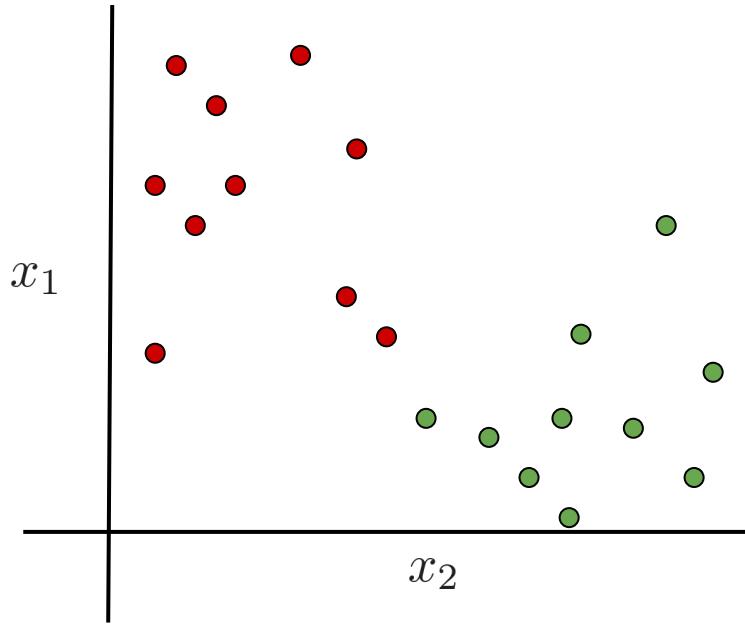
$$\sum_{i=1}^m w_i x_i + b$$

**Wx**



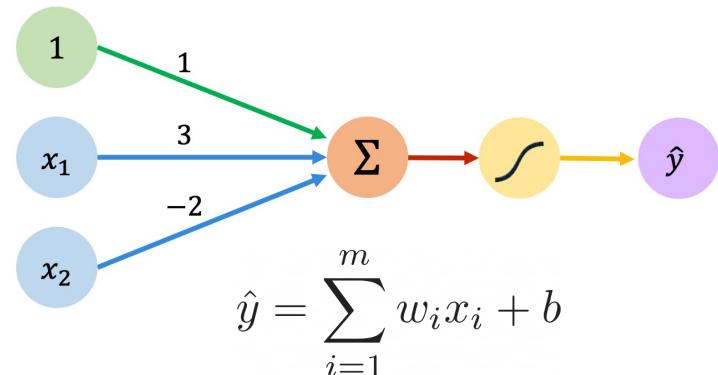
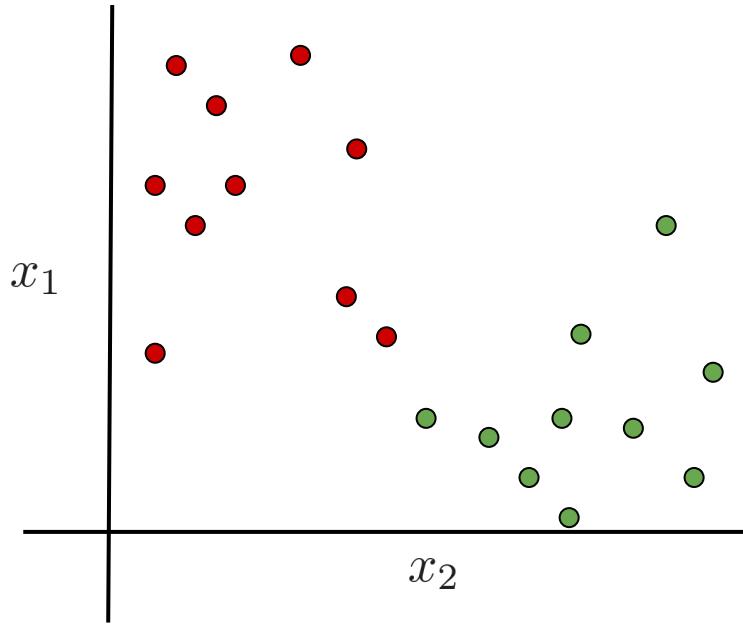
# The Perceptron and Activation

Binary Classification Task



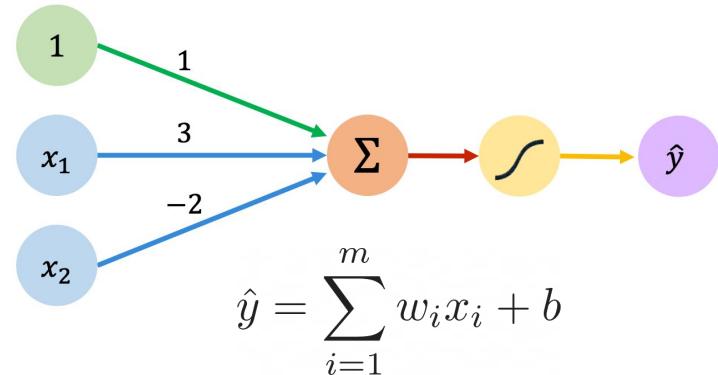
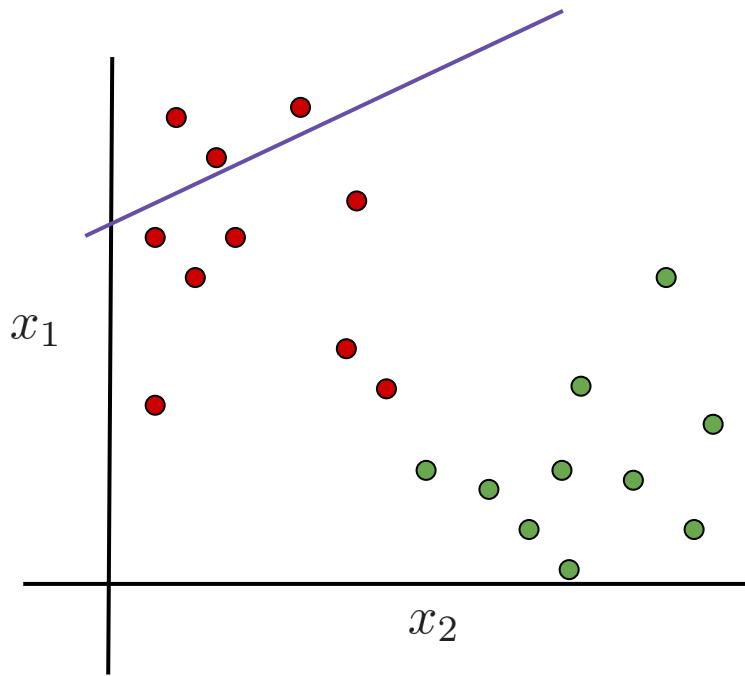
# The Perceptron and Activation

Binary Classification Task



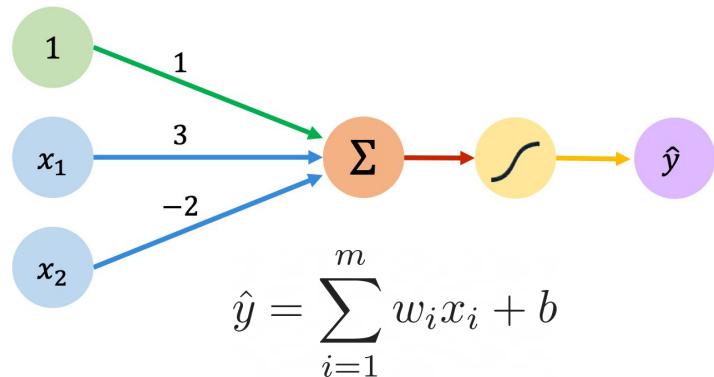
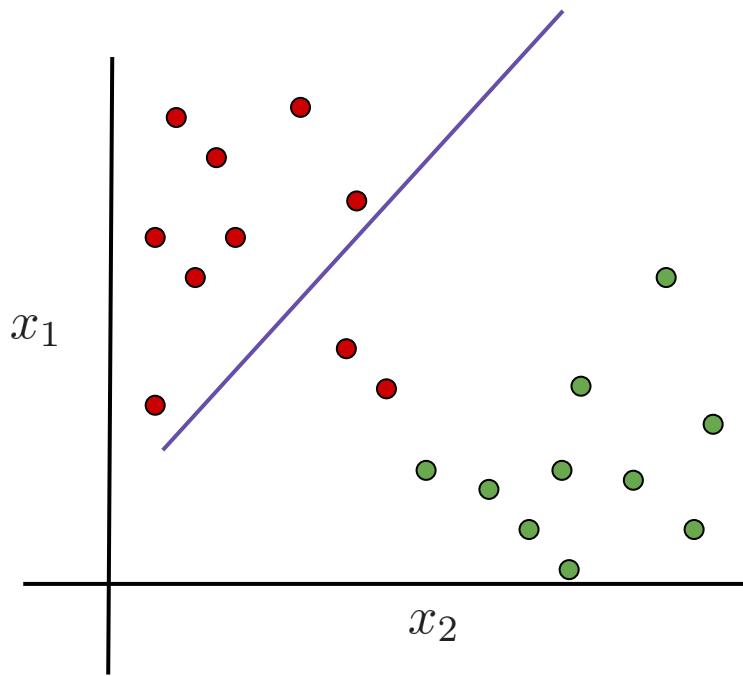
# The Perceptron and Activation

Binary Classification Task



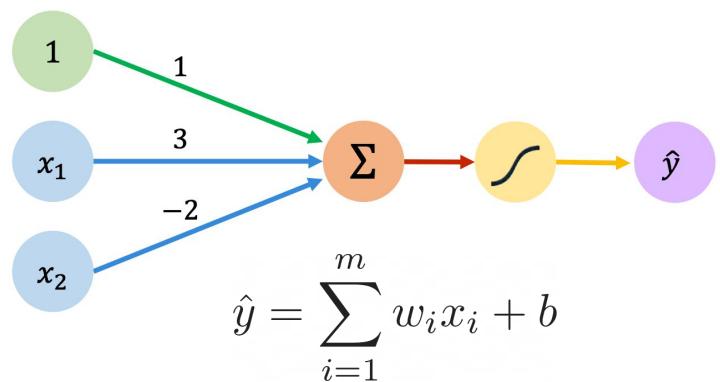
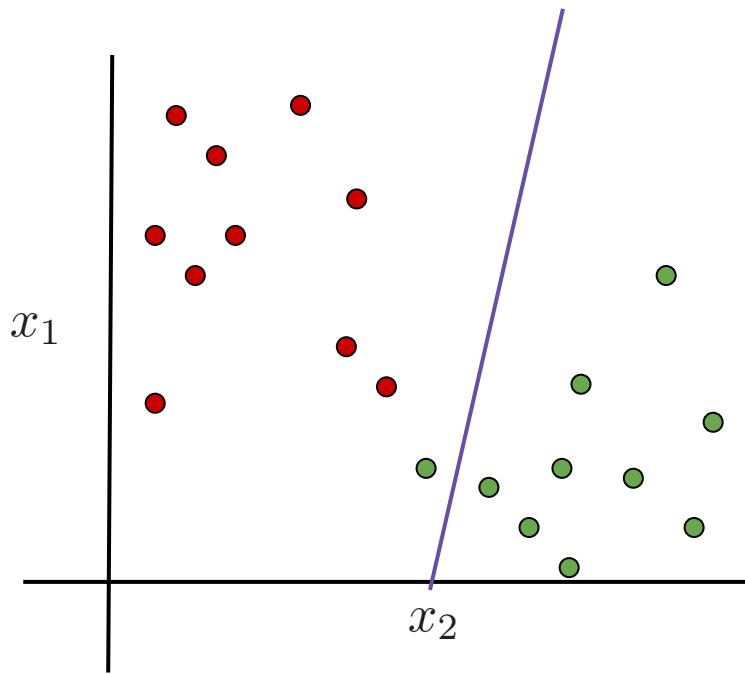
# The Perceptron and Activation

Binary Classification Task



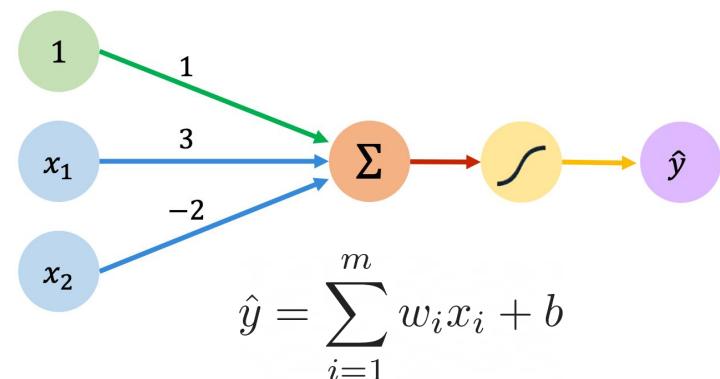
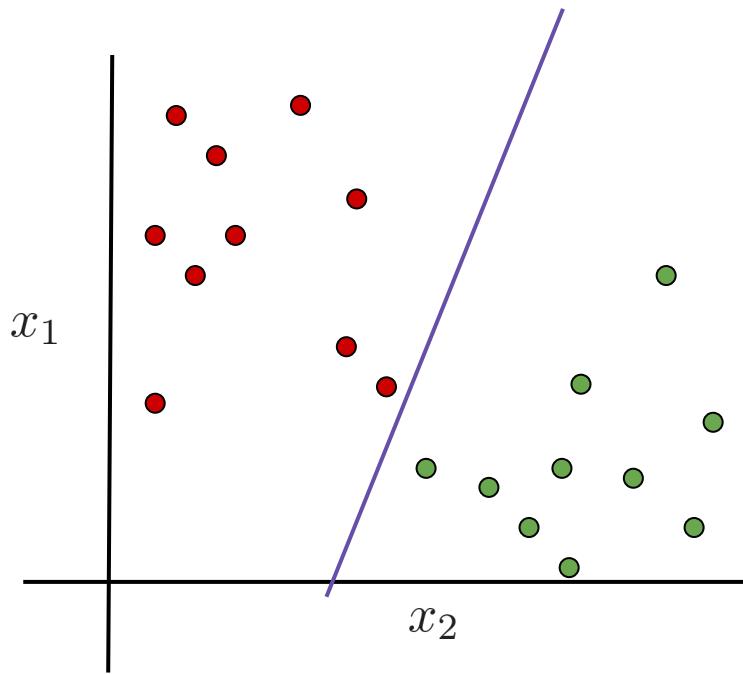
# The Perceptron and Activation

Binary Classification Task



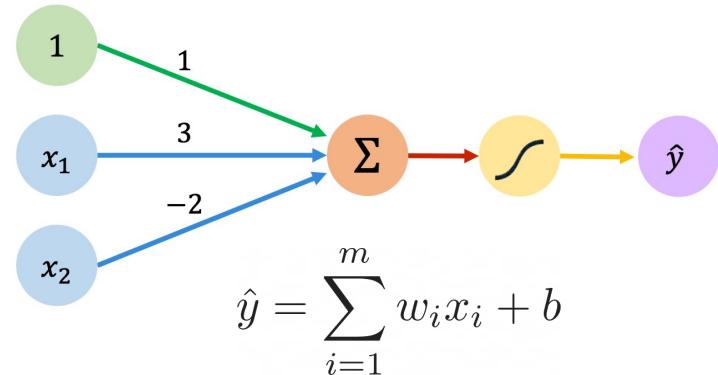
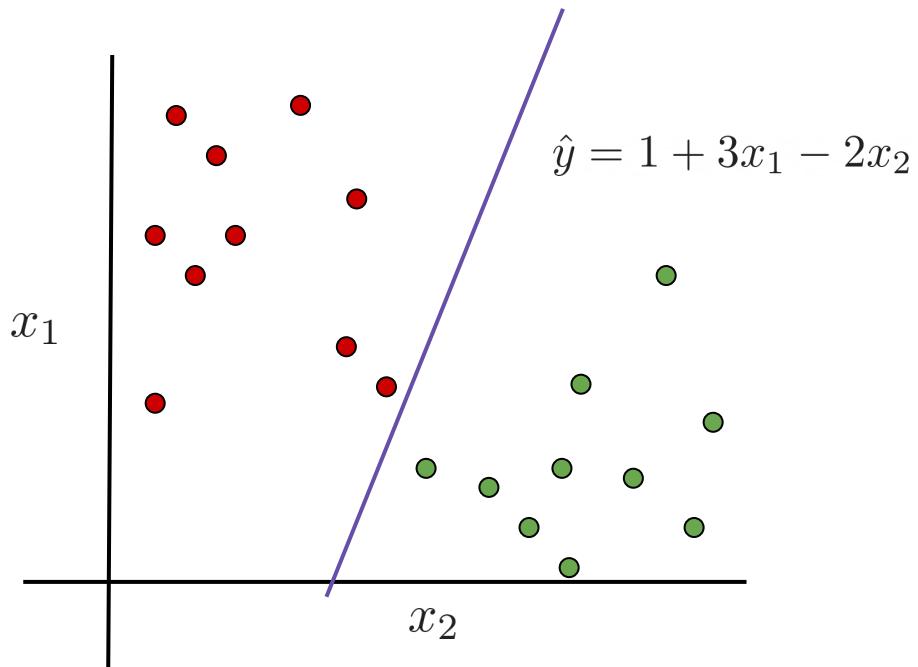
# The Perceptron and Activation

Binary Classification Task



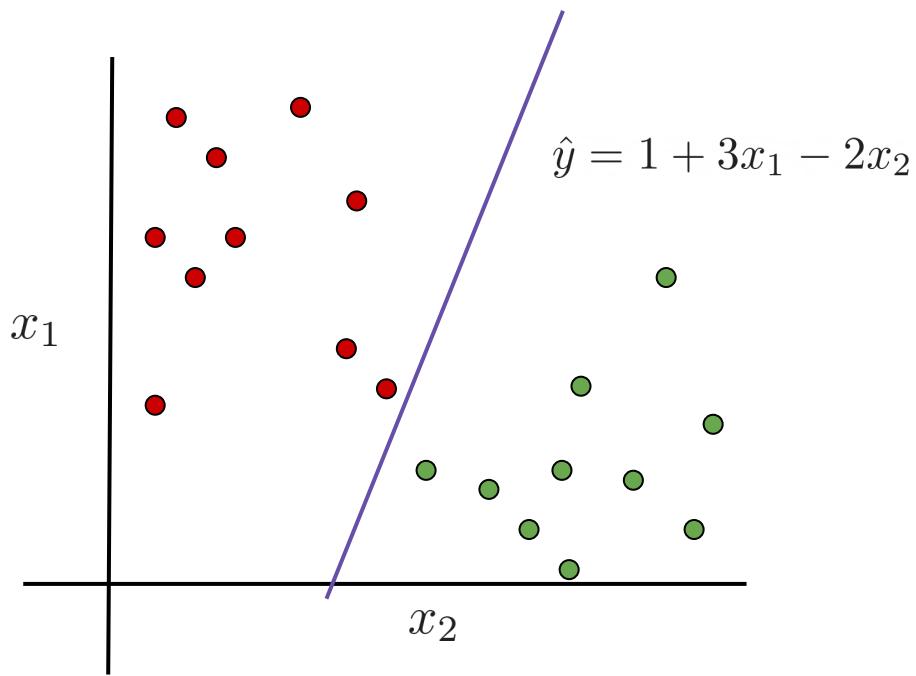
# The Perceptron and Activation

## Binary Classification Task

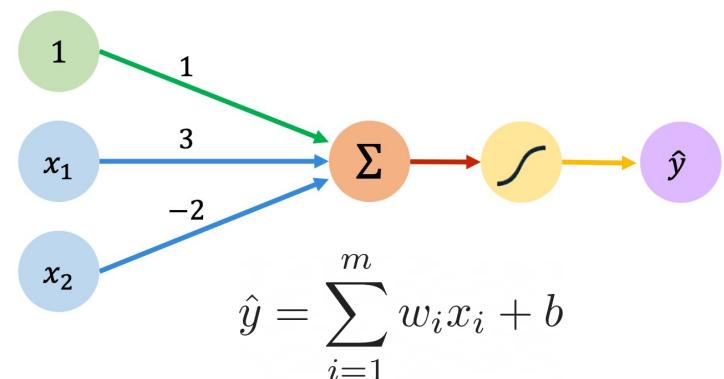
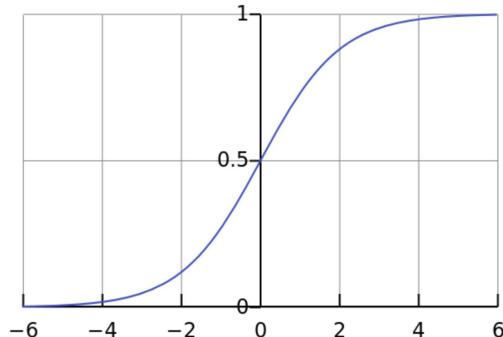


# The Perceptron and Activation

Binary Classification Task

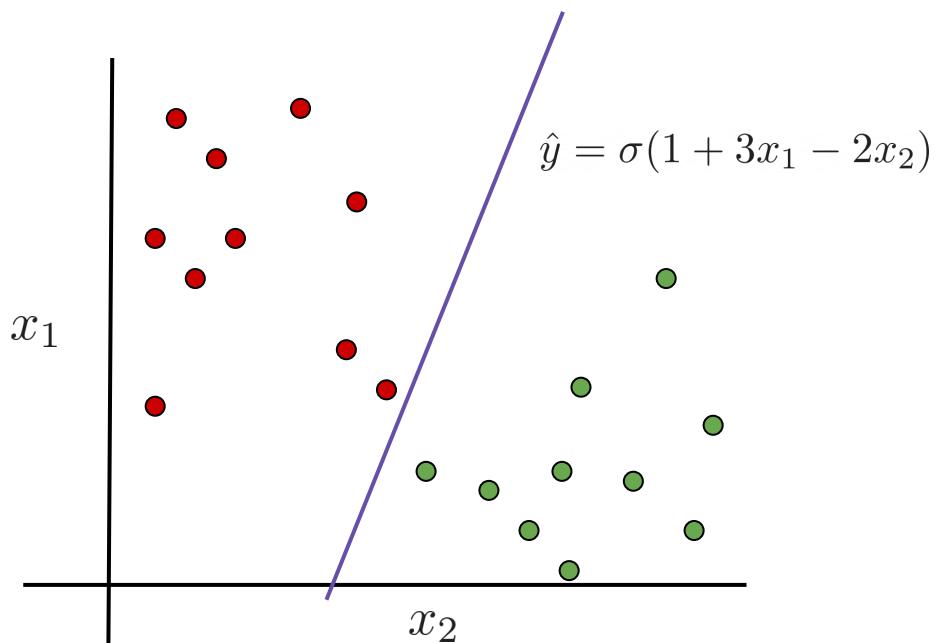


Sigmoid

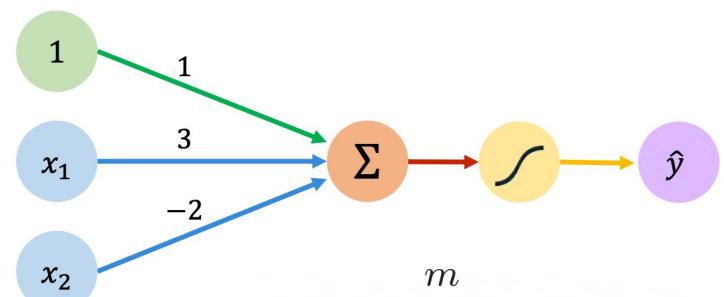
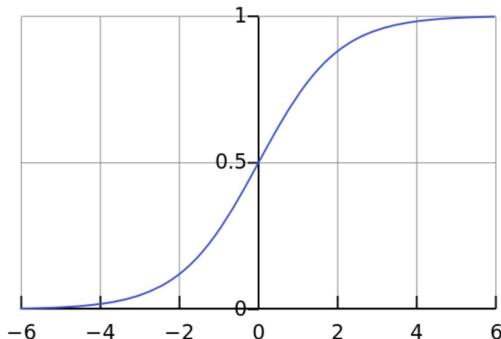


# The Perceptron and Activation

Binary Classification Task

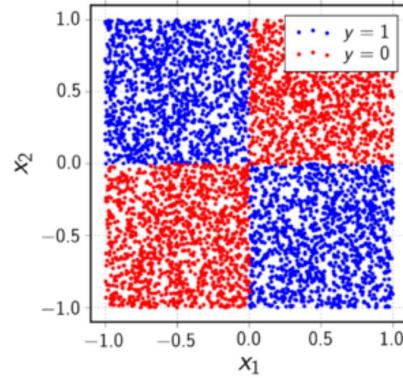
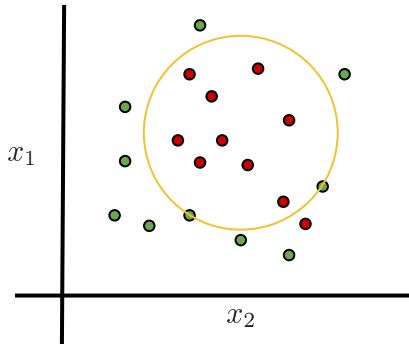
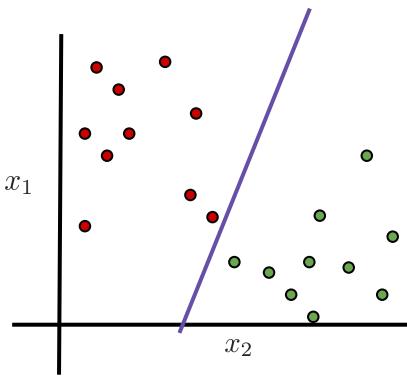


Sigmoid



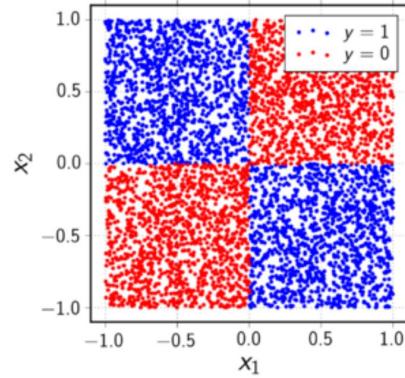
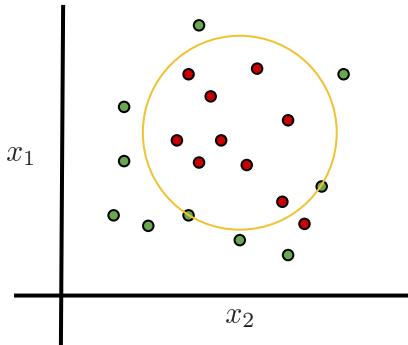
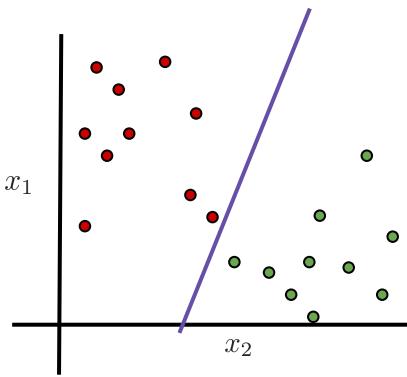
$$\hat{y} = \sigma\left(\sum_{i=1}^m w_i x_i + b\right)$$

# Limitations of Linear Single Layer Classifiers



**XOR Problem**

# Limitations of Linear Single Layer Classifiers



**XOR Problem**

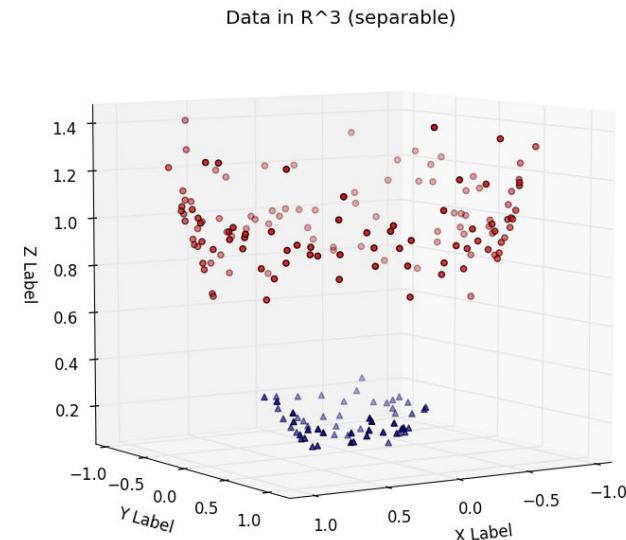
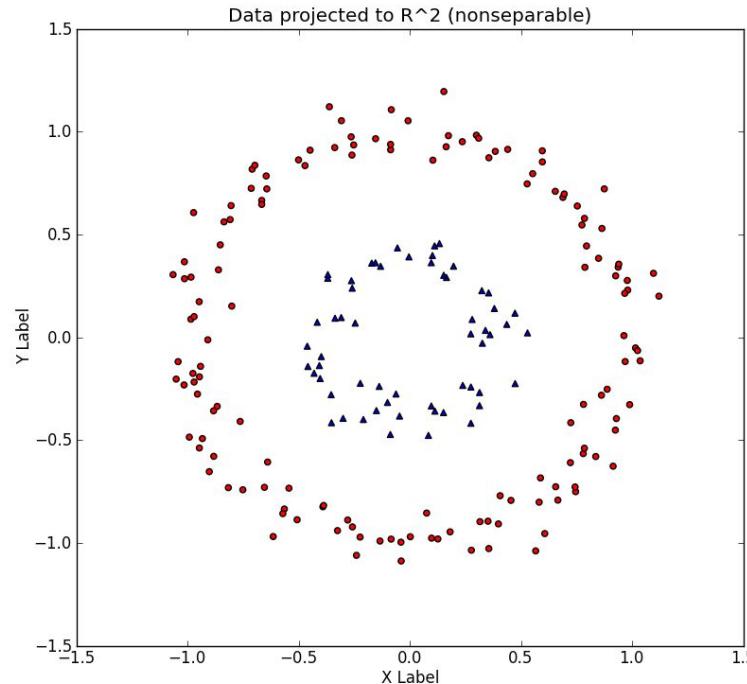
## Possible Solutions

Add more layers (deep learning)

Map into another (higher dimensional) space

We need to be able to automatically extract features

# Limitations of Linear Single Layer Classifiers



# Universal Approximation Theorem

“ A neural network with a **single hidden layer of sufficient size**

Can approximate any continuous function



# Universal Approximation Theorem

“ A neural network with a **single hidden layer of sufficient size**

Can approximate any continuous function

There exists a true function relating the inputs to the outputs

A neural network can approximate this function to arbitrary precision given sufficient layer size

The required layer size can be extremely large and grow rapidly with the dimensionality of the problem



# Universal Approximation Theorem

“

A neural network with a **single hidden layer of sufficient size**

Can approximate any continuous function



There exists a true function relating the inputs to the outputs

A neural network can approximate this function to arbitrary precision given sufficient layer size

The required layer size can be extremely large and grow rapidly with the dimensionality of the problem

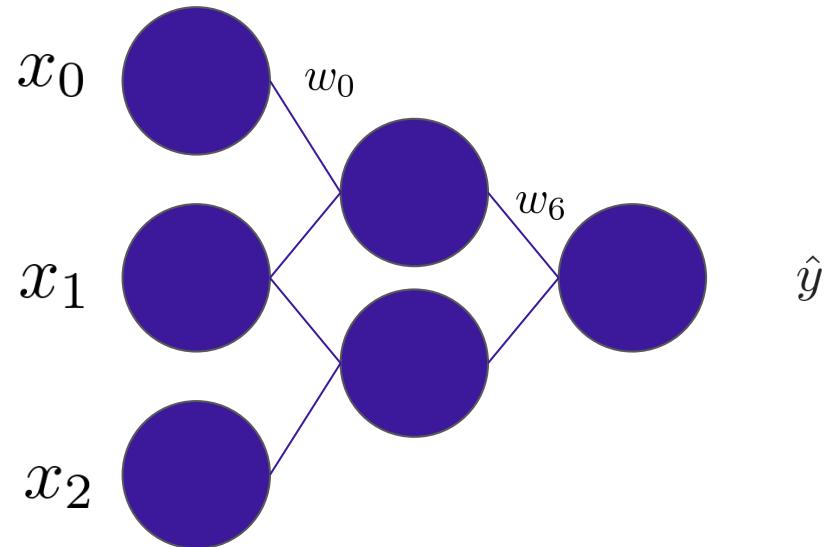
Use of multiple hidden layers makes the NN vector representation of your problem increasingly more abstract

- ❖ How do we train?
- ❖ Compute grows (almost) exponentially

# Predicting Housing Price

During the **optimization** process

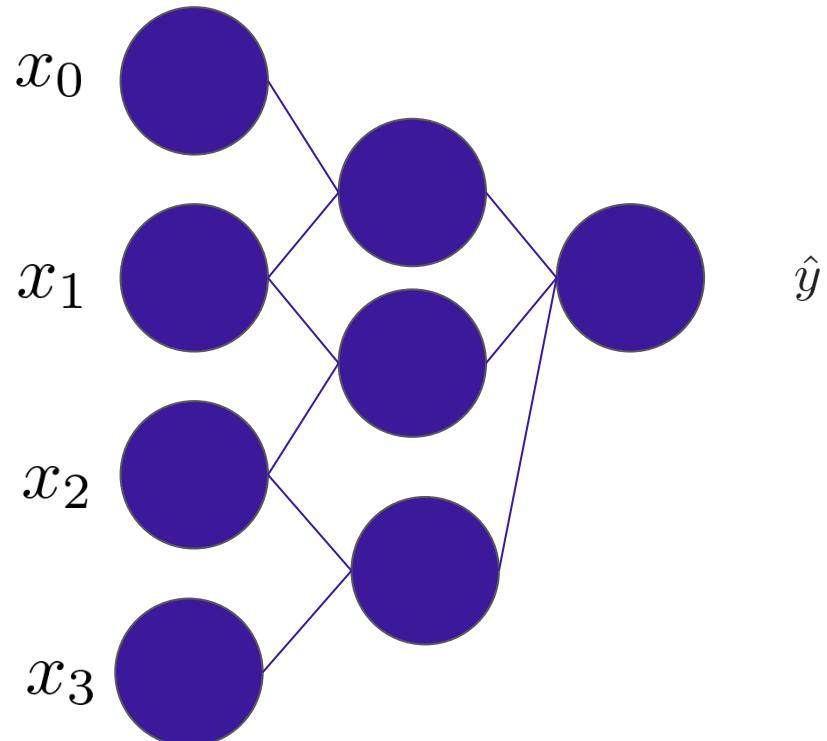
The NN learns to **encode a representation** that maps  
the input to the output



# Predicting Housing Price

During the **optimization** process

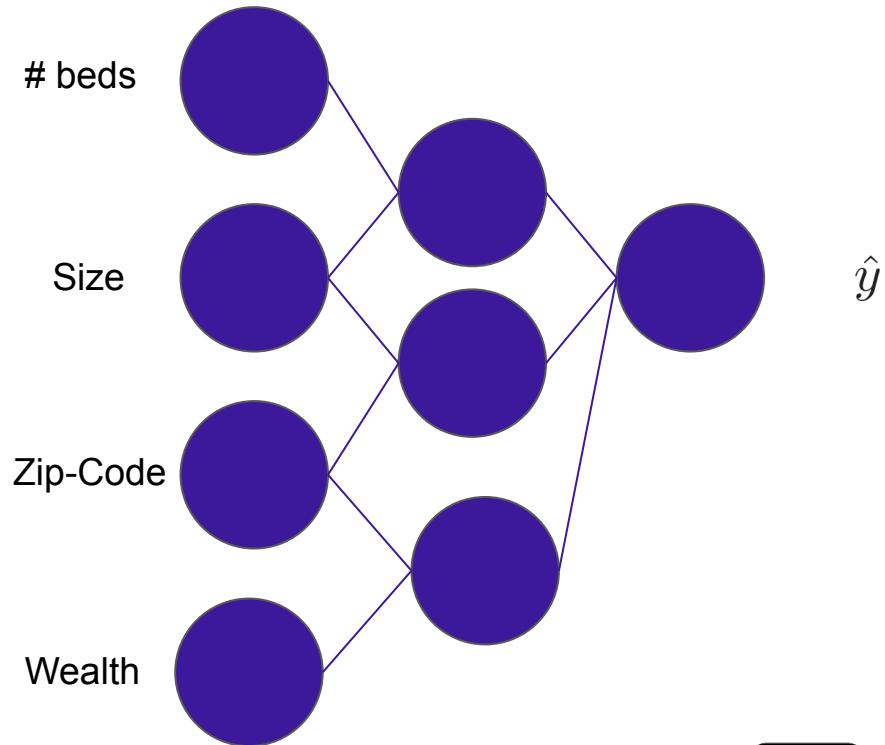
The NN learns to **encode a representation** that maps  
the input to the output



# Predicting Housing Price

During the **optimization** process

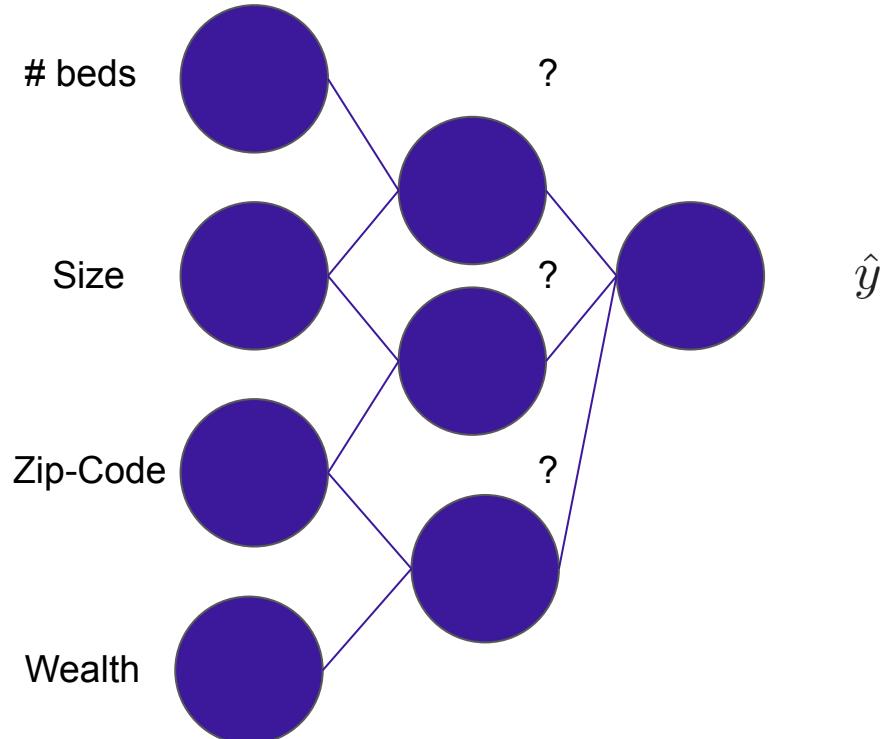
The NN learns to **encode a representation** that maps  
the input to the output



# Predicting Housing Price

During the **optimization** process

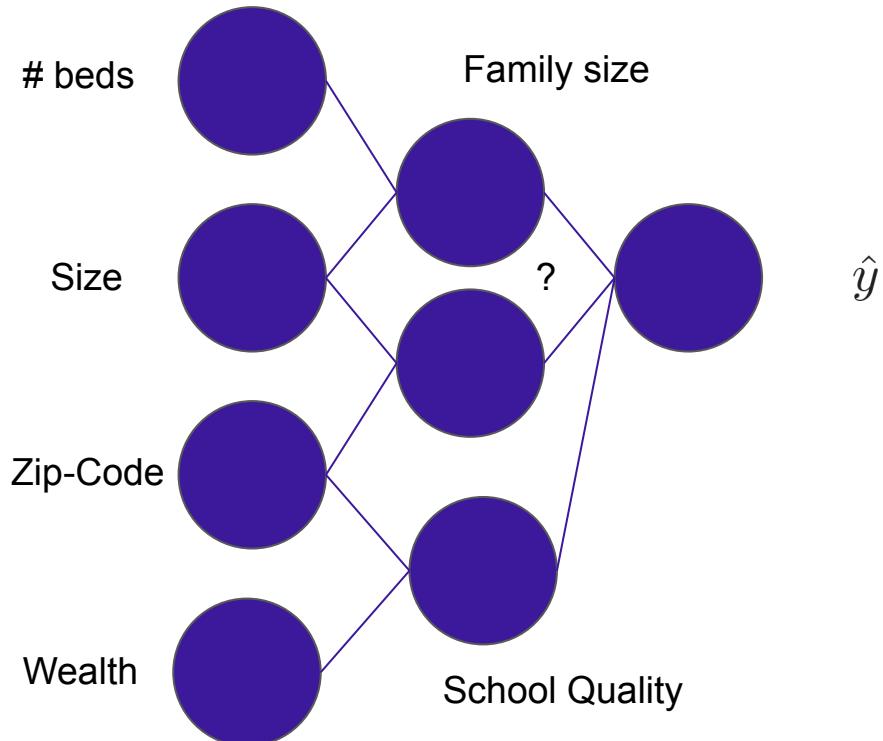
The NN learns to **encode a representation** that maps  
the input to the output



# Predicting Housing Price

During the **optimization** process

The NN learns to **encode a representation** that maps  
the input to the output

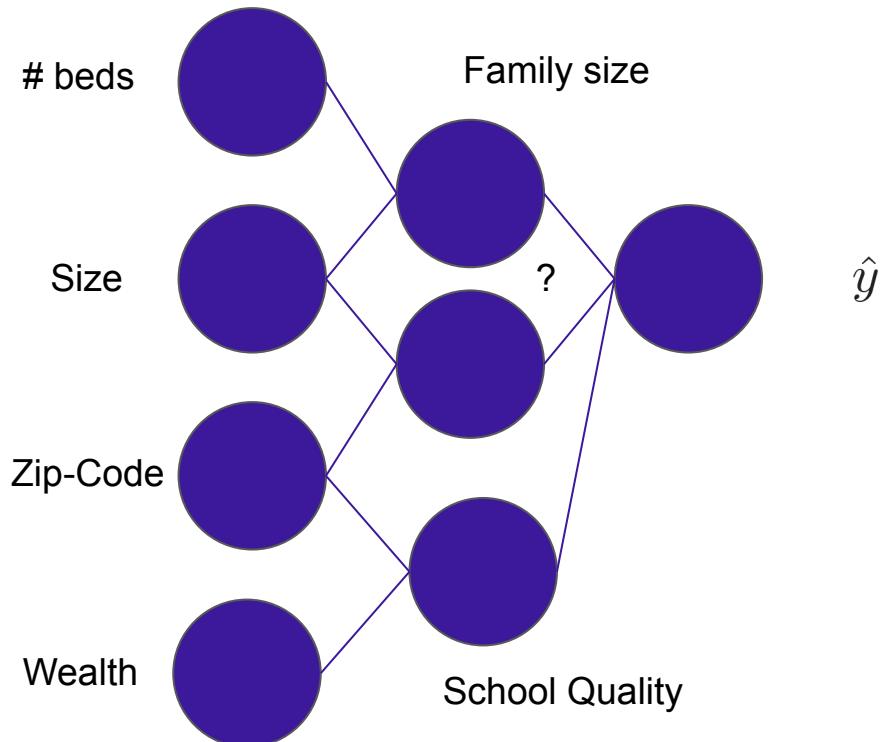


# Predicting Housing Price

During the **optimization** process

The NN learns to **encode a representation** that maps  
the input to the output

Transform the input to a space where we  
are able to **separate** the features

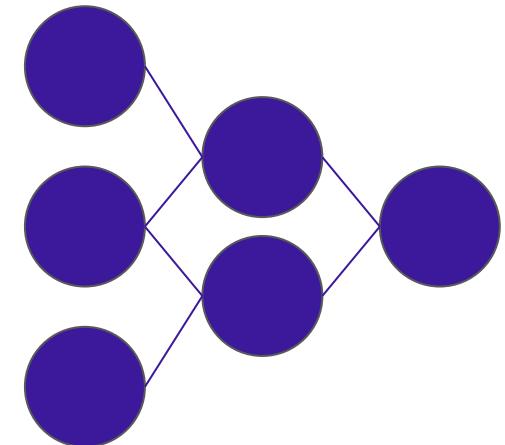


# Predicting Faces

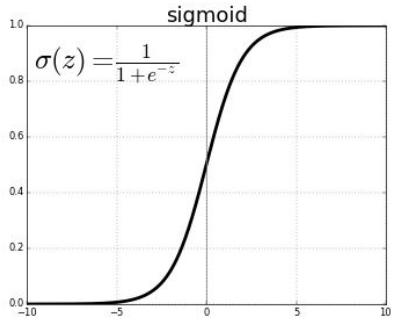
During the **optimization** process

The NN learns to **encode a representation** that maps  
the input to the output

A deep neural network **encodes** the  
**representation** in an increasingly  
abstract way



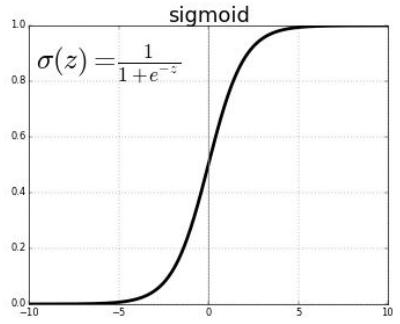
# Activation Functions



One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

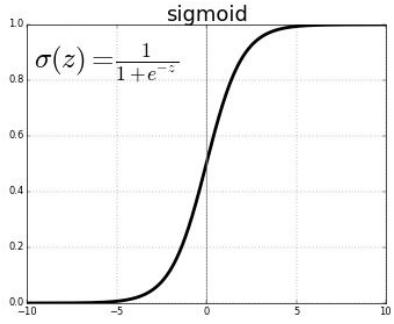
# Activation Functions



One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

# Activation Functions

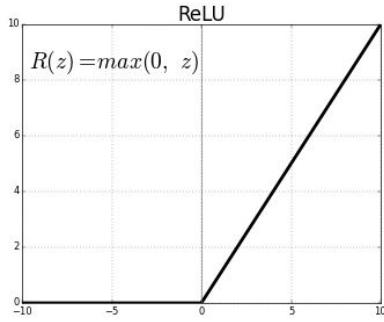
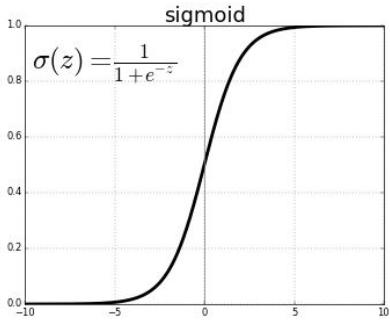


One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

- ❖ Probability Estimate
- ❖ Continuously differentiable
- ❖ Vanishing derivatives due to saturated neurons

# Activation Functions

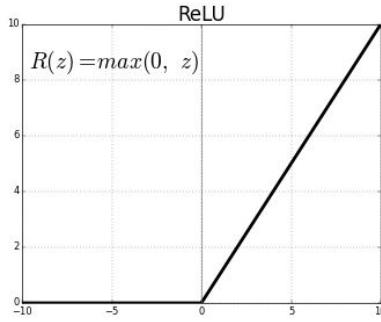
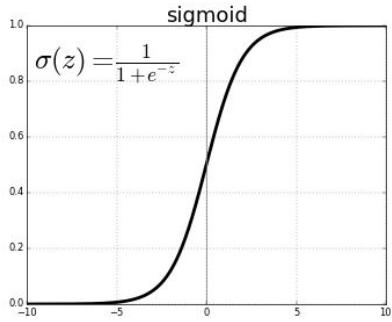


One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

- ❖ Probability Estimate
- ❖ Continuously differentiable
- ❖ Vanishing derivatives due to saturated neurons

# Activation Functions

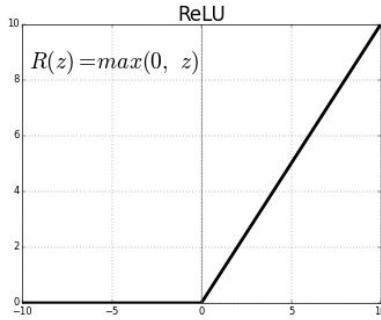
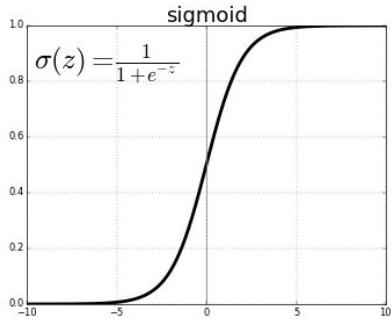


One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

- ❖ Probability Estimate
- ❖ Continuously differentiable
- ❖ Vanishing derivatives due to saturated neurons
- ❖ Very cheap to compute
- ❖ Piece-wise linear functions
- ❖ Dead neurons
- ❖ Not differentiable at 0

# Activation Functions



One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

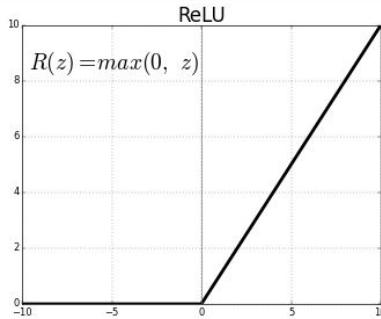
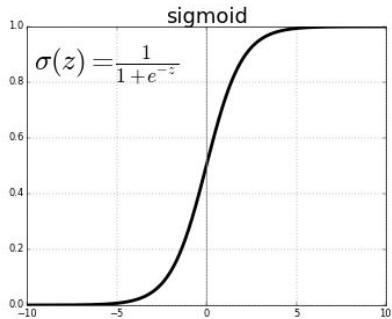
- ❖ Probability Estimate
- ❖ Continuously differentiable
- ❖ Vanishing derivatives due to saturated neurons
- ❖ Very cheap to compute
- ❖ Piece-wise linear functions
- ❖ Dead neurons
- ❖ Not differentiable at 0

Activation functions are applied to the output of each neuron (point-wise)

Simple derivative

Non-linear behaviour

# Activation Functions



One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

- ❖ Probability Estimate
- ❖ Continuously differentiable
- ❖ Vanishing derivatives due to saturated neurons
- ❖ Very cheap to compute
- ❖ Piece-wise linear functions
- ❖ Dead neurons
- ❖ Not differentiable at 0

Activation functions are applied to the output of each neuron (point-wise)

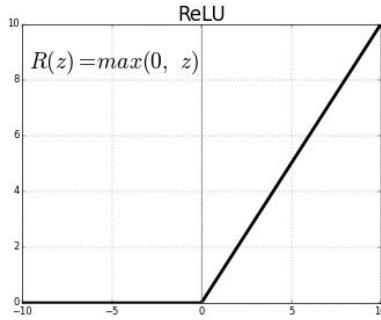
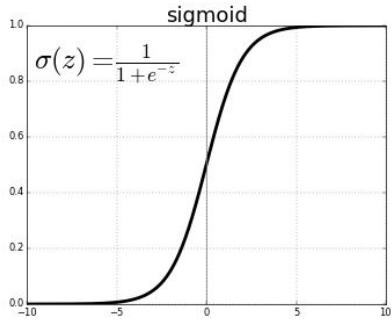
Simple derivative

Non-linear behaviour

ReLU made our lives much easier and faster

Most commonly used activation

# Activation Functions



One of the reasons that enable NNs to encode highly abstract features is the use of **non-linear** activation functions.

Not using non-linearities leads to linear networks

- ❖ Probability Estimate
- ❖ Continuously differentiable
- ❖ Vanishing derivatives due to saturated neurons
- ❖ Very cheap to compute
- ❖ Piece-wise linear functions
- ❖ Dead neurons
- ❖ Not differentiable at 0

Activation functions are applied to the output of each neuron (point-wise)

Simple derivative

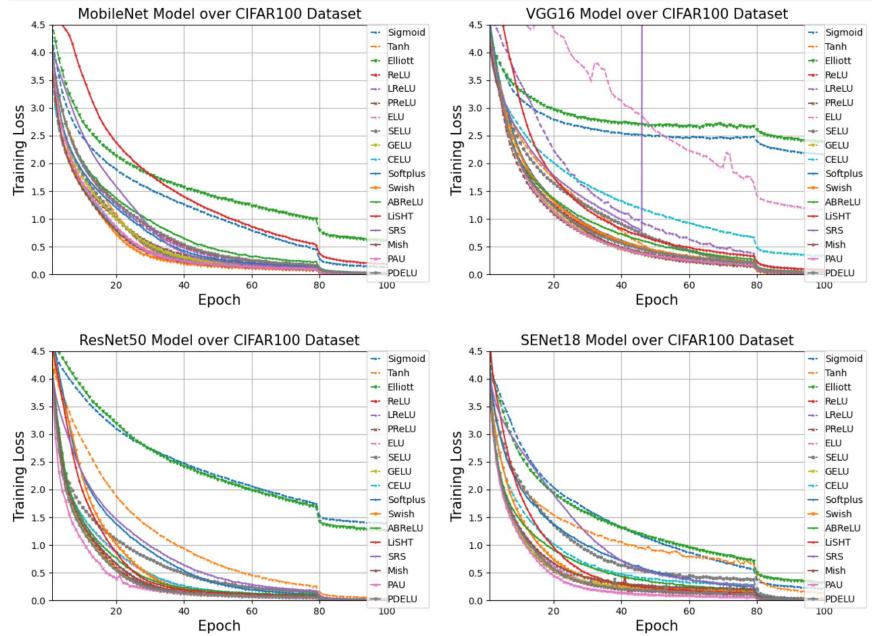
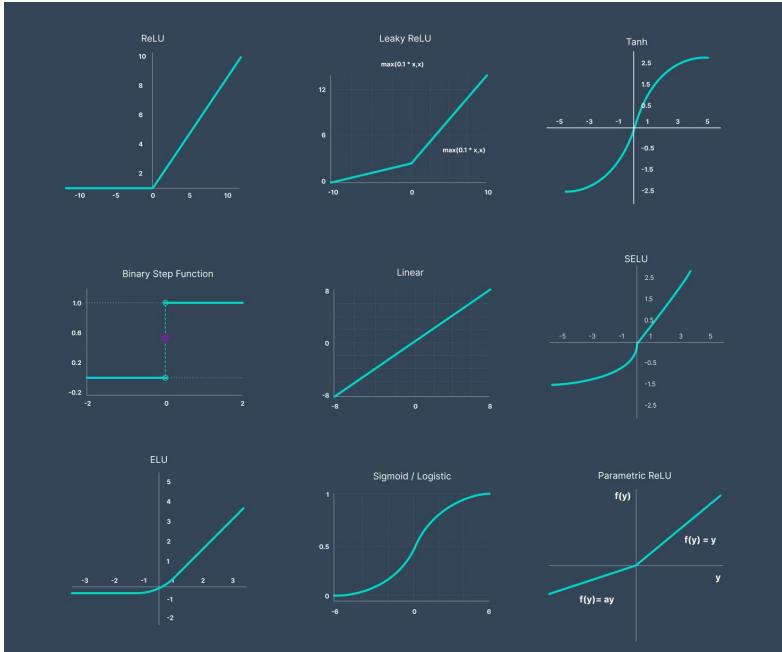
Non-linear behaviour

ReLU made our lives much easier and faster

Most commonly used activation

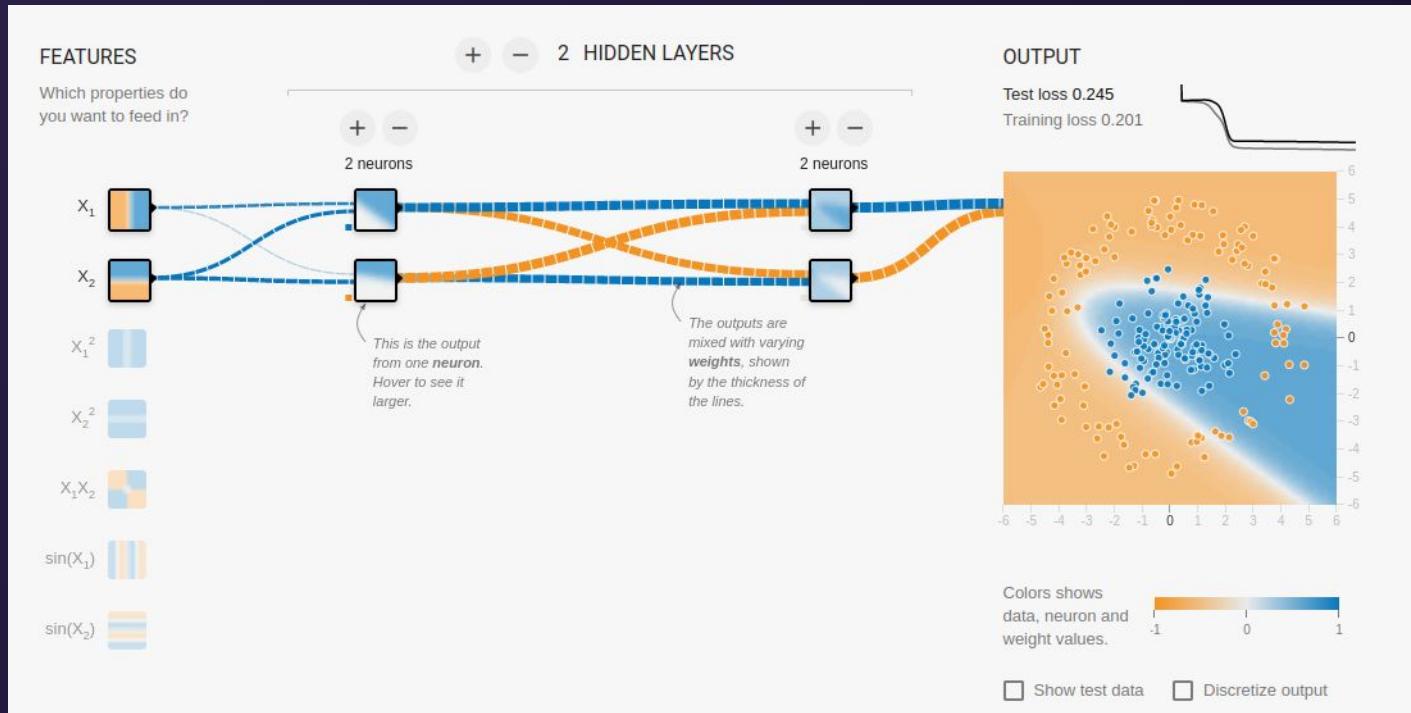
Many more! We can design our own!

# Activation Functions



Many more! We can design our own!

# Neural Network Demo

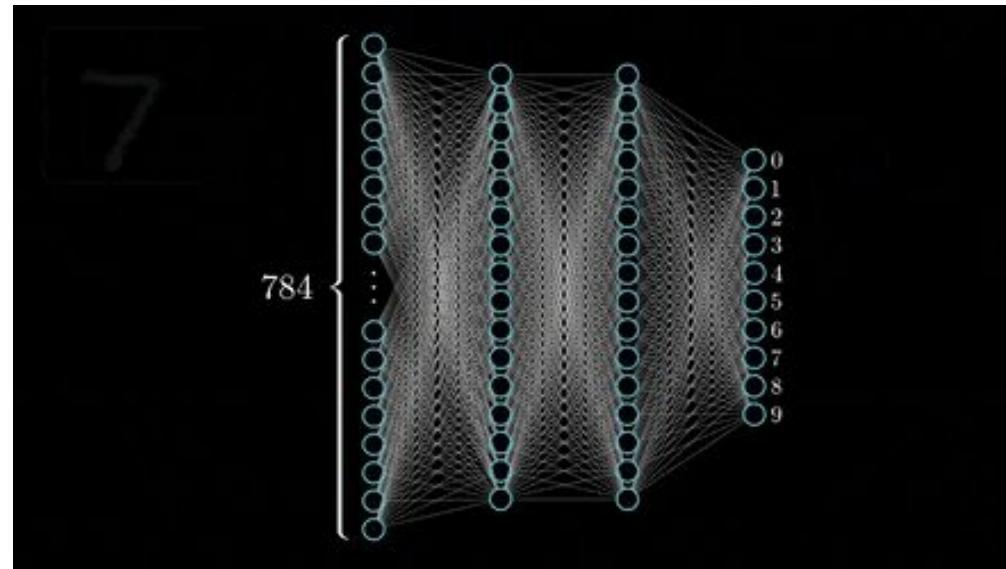


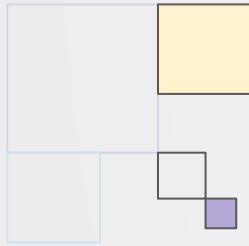
<https://playground.tensorflow.org/>

SURF

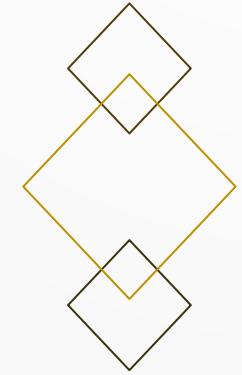
# Neural Network

- ❖ The output of previous layer is used as an input to the next layer
- ❖ The input layer is data input and the output is a prediction
- ❖ Anything in between is **hidden**
- ❖ Layers are represented as vectors
- ❖ Edges are matrices
- ❖ We train the weights





# Neural Network Training



High Performance Machine Learning Group



# Recipe for Training

01.

**Process your data**

Define the data to be used  
Do we have labels?

02.

**Define the Model**

Define the layers and  
The forward propagation

03.

**What function to  
optimize?**

Define the function to  
approximate  
your desired solution

04.

**How to evaluate  
the model?**

Which metrics are going to  
tell us how well we are  
doing on unseen data?

# Recipe for Training

01.

$(x_1, \dots, x_m), y$

# Recipe for Training

01.

$$(x_1, \dots, x_m), y$$

02.

$$f_{NN}(x_1, x_2, \dots, x_n)$$

# Recipe for Training

01.

$$(x_1, \dots, x_m), y$$

02.

$$f_{NN}(x_1, x_2, \dots, x_n)$$

03.

$$\text{MSE} \quad \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

# Recipe for Training

01.

$$(x_1, \dots, x_m), y$$

02.

$$f_{NN}(x_1, x_2, \dots, x_n)$$

03.

MSE       $\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$

CE       $-\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$

# Recipe for Training

01.

$$(x_1, \dots, x_m), y$$

02.

$$f_{NN}(x_1, x_2, \dots, x_n)$$

03.

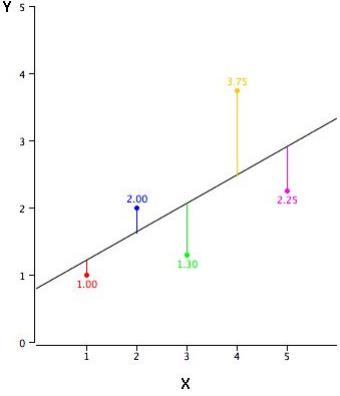
MSE       $\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$

CE       $-\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$

04.

Accuracy, F1-score,  
precision, recall

# Loss Functions

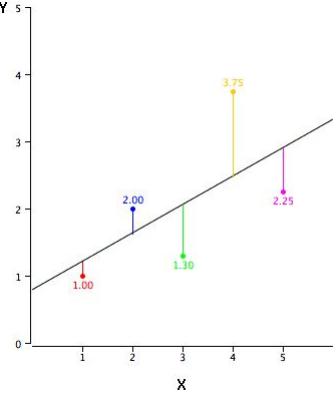


The loss function is used to bridge the gap between your neural network predictions and the true value

We optimize (minimize) the loss to tune the weights  
In the direction of biggest positive change

$$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

# Loss Functions



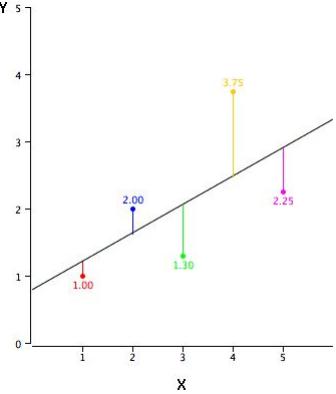
$$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

- ❖ Distance/statistical metric assumes a Gaussian prior
- ❖ Easy to understand, easy to Compute
- ❖ Prone to outliers
- ❖ Not suitable for classification problems

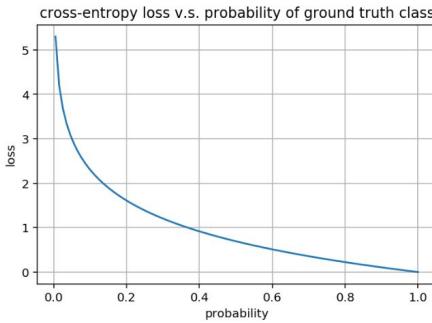
The loss function is used to bridge the gap between your neural network predictions and the true value

We optimize (minimize) the loss to tune the weights  
In the direction of biggest positive change

# Loss Functions



$$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

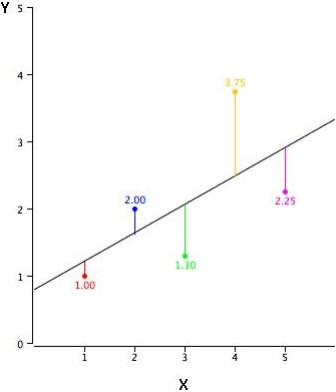


The loss function is used to bridge the gap between your neural network predictions and the true value

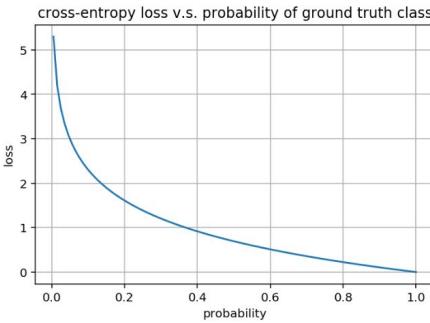
We optimize (minimize) the loss to tune the weights  
In the direction of biggest positive change

- ❖ Distance/statistical metric assumes a Gaussian prior
- ❖ Easy to understand, easy to Compute
- ❖ Prone to outliers
- ❖ Not suitable for classification problems

# Loss Functions



$$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$



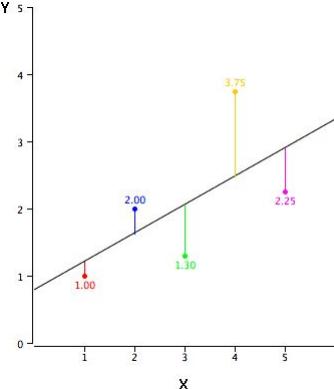
$$-\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

- ❖ Distance/statistical metric assumes a Gaussian prior
- ❖ Easy to understand, easy to Compute
- ❖ Prone to outliers
- ❖ Not suitable for classification problems
- ❖ Suitable for multi-class problems
- ❖ Information theory foundation
- ❖ Not exactly the most stable loss

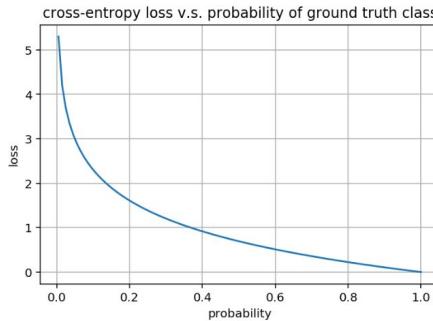
The loss function is used to bridge the gap between your neural network predictions and the true value

We optimize (minimize) the loss to tune the weights  
In the direction of biggest positive change

# Loss Functions



$$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$



$$-\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

- ❖ Distance/statistical metric assumes a Gaussian prior
- ❖ Easy to understand, easy to Compute
- ❖ Prone to outliers
- ❖ Not suitable for classification problems
- ❖ Suitable for multi-class problems
- ❖ Information theory foundation
- ❖ Not exactly the most stable loss

The loss function is used to bridge the gap between your neural network predictions and the true value

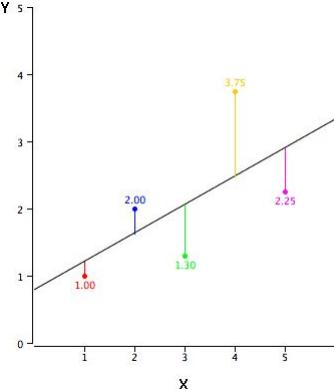
We optimize (minimize) the loss to tune the weights  
In the direction of biggest positive change

CE is easily composed with sigmoid  
Or Softmax activations!

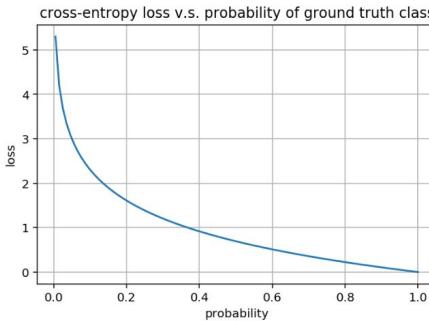
CE and Softmax has better behaved gradients.

Non-linear behaviour

# Loss Functions



$$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$



$$-\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

- ❖ Distance/statistical metric assumes a Gaussian prior
- ❖ Easy to understand, easy to Compute
- ❖ Prone to outliers
- ❖ Not suitable for classification problems
- ❖ Suitable for multi-class problems
- ❖ Information theory foundation
- ❖ Not exactly the most stable loss

The loss function is used to bridge the gap between your neural network predictions and the true value

We optimize (minimize) the loss to tune the weights  
In the direction of biggest positive change

CE is easily composed with sigmoid  
Or Softmax activations!

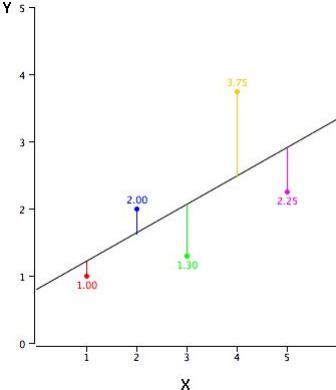
CE and Softmax has better behaved gradients.

Non-linear behaviour

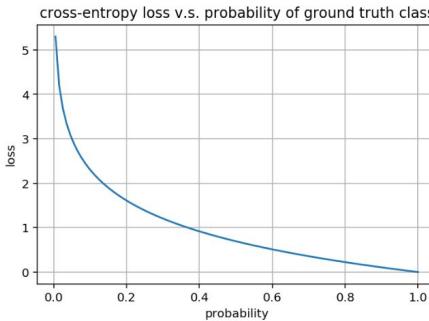
CE is the negative log-likelihood

Most commonly used activation for classification

# Loss Functions



$$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$



$$-\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

- ❖ Distance/statistical metric assumes a Gaussian prior
- ❖ Easy to understand, easy to Compute
- ❖ Prone to outliers
- ❖ Not suitable for classification problems
- ❖ Suitable for multi-class problems
- ❖ Information theory foundation
- ❖ Not exactly the most stable loss

The loss function is used to bridge the gap between your neural network predictions and the true value

We optimize (minimize) the loss to tune the weights  
In the direction of biggest positive change

CE is easily composed with sigmoid  
Or Softmax activations!

CE and Softmax has better behaved gradients.

Non-linear behaviour

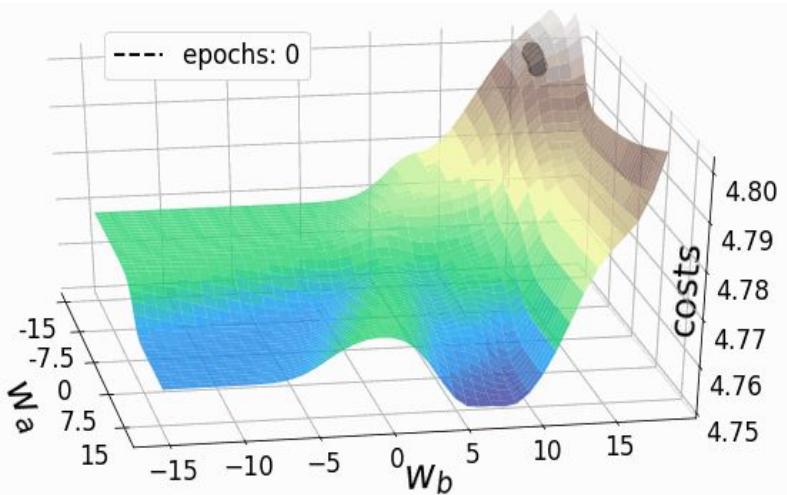
CE is the negative log-likelihood

Most commonly used activation for classification

Many more! We can design our own!

SURF

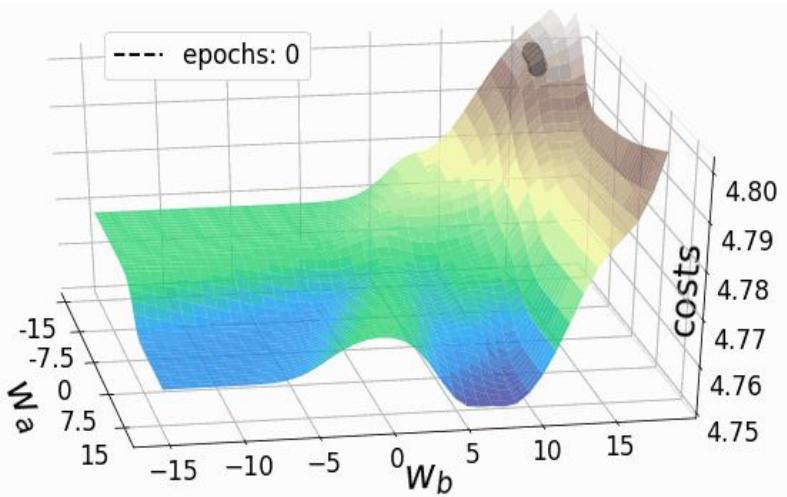
# Stochastic Gradient Descent



01.

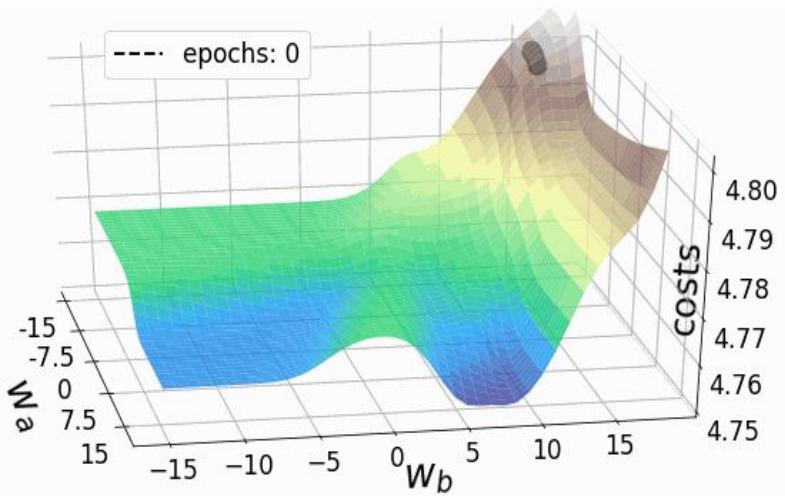
$$L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

# Stochastic Gradient Descent



01.  $L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
02.  $\nabla L(\mathbf{w}_j, b)$

# Stochastic Gradient Descent



01.

$$L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

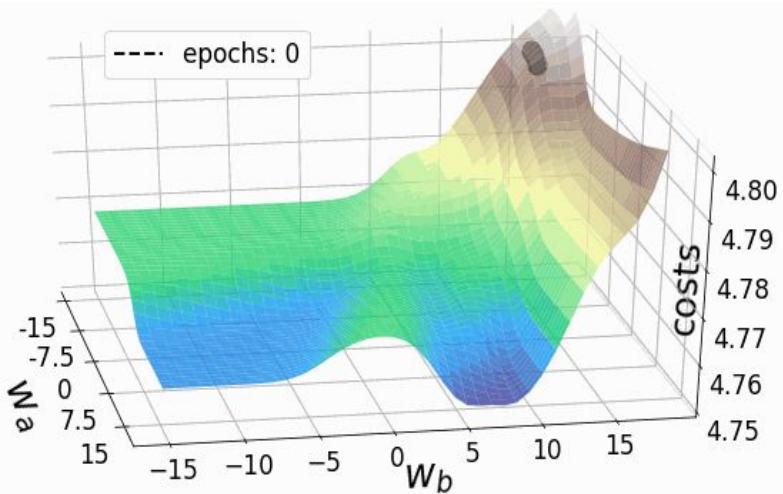
02.

$$\nabla L(\mathbf{w}_j, b)$$

03.

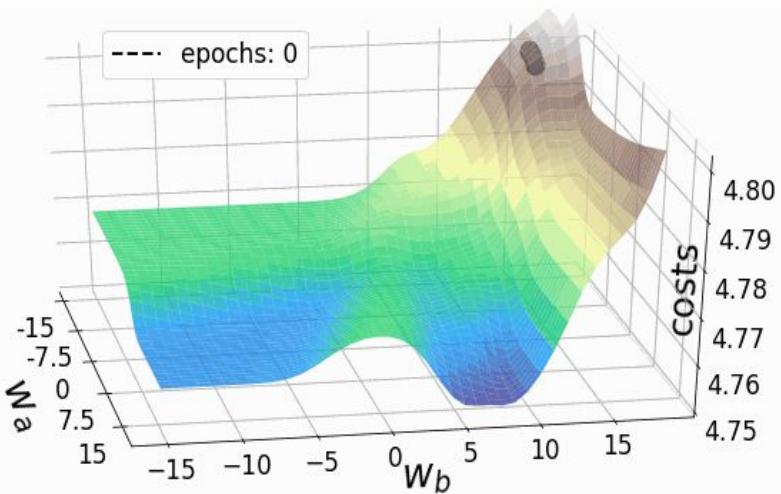
Create batches of  $N$  examples to propagate  
and compute  $\nabla L(\mathbf{w}_j, b)$

# Stochastic Gradient Descent



01.  $L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
02.  $\nabla L(\mathbf{w}_j, b)$
03. Create batches of  $N$  examples to propagate and compute  $\nabla L(\mathbf{w}_j, b)$
04.  $\mathbf{w}_{j+1} = \mathbf{w}_j - \alpha \nabla L(\mathbf{w}_j, b)$

# Stochastic Gradient Descent

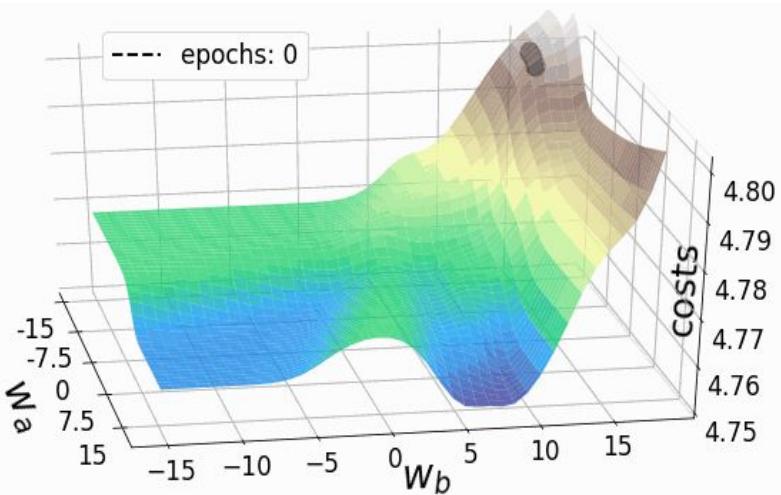


01.  $L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
02.  $\nabla L(\mathbf{w}_j, b)$
03. Create batches of  $N$  examples to propagate and compute  $\nabla L(\mathbf{w}_j, b)$
04.  $\mathbf{w}_{j+1} = \mathbf{w}_j - \alpha \nabla L(\mathbf{w}_j, b)$

**Learning Rate**

Choice of learning rate critical  
SGD is the main engine behind training  
Many variations exist

# Stochastic Gradient Descent



01.  $L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
02.  $\nabla L(\mathbf{w}_j, b)$
03. Create batches of  $N$  examples to propagate and compute  $\nabla L(\mathbf{w}_j, b)$
04.  $\mathbf{w}_{j+1} = \mathbf{w}_j - \alpha \nabla L(\mathbf{w}_j, b)$

Learning Rate

Choice of learning rate critical  
SGD is the main engine behind training  
Many variations exist

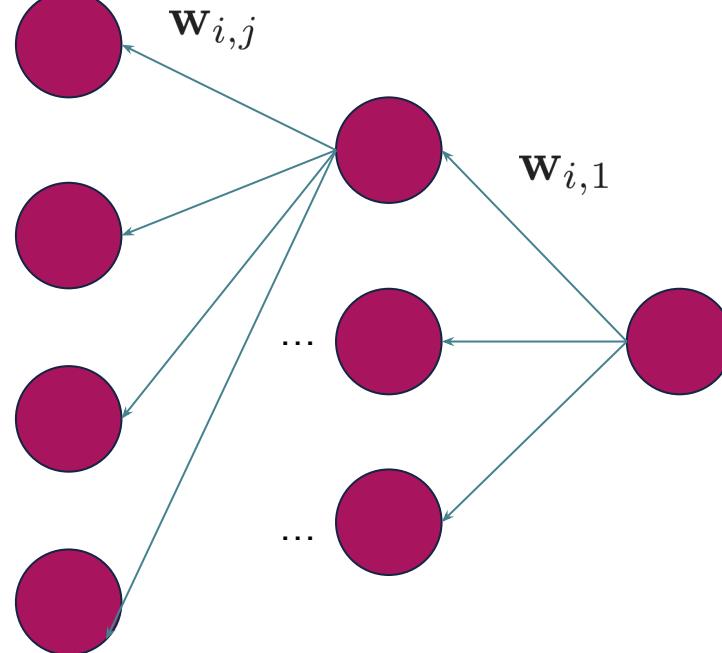
- ❖ Can be used with loss function that are not differentiable
- ❖ No Guarantee that we find the global optimum

# Backpropagation

$$\hat{y} = g(\mathbf{W}_0 f(\mathbf{W}_1 \mathbf{x}))$$

A (deep) neural network is a deeply **nested functions**:

- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**

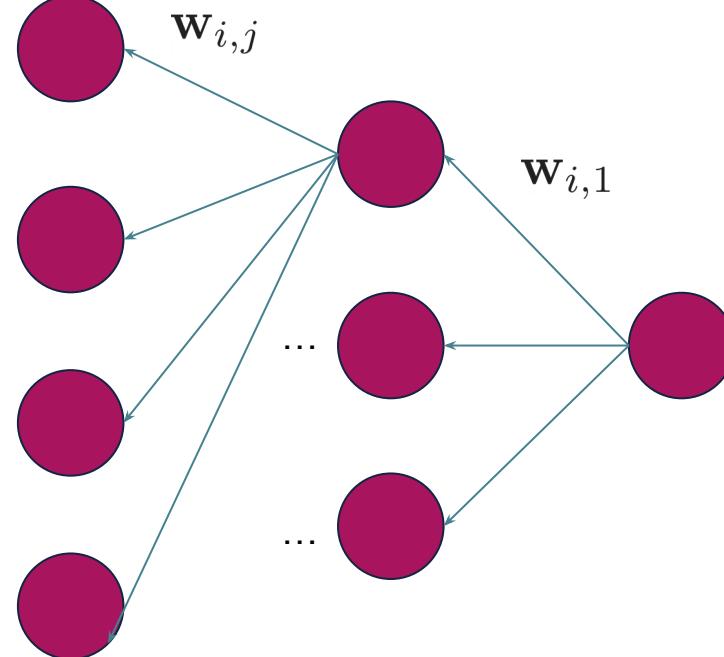


# Backpropagation

$$\hat{y} = g(\mathbf{W}_0 f(\mathbf{W}_1 \mathbf{x}))$$

A (deep) neural network is a deeply **nested functions**:

- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**



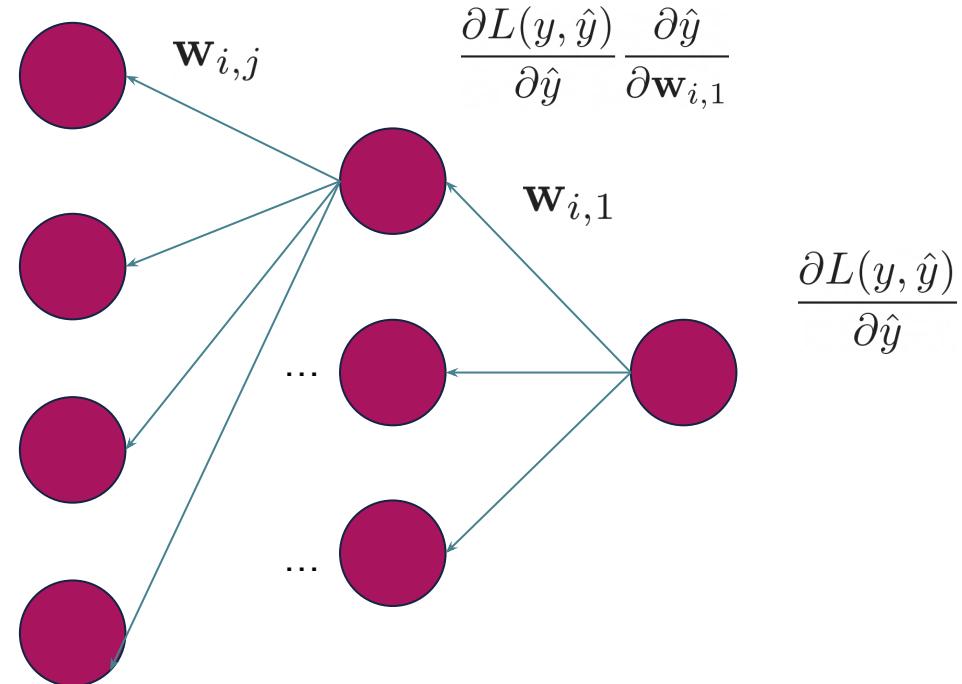
$$\frac{\partial L(y, \hat{y})}{\partial \hat{y}}$$

# Backpropagation

$$\hat{y} = g(\mathbf{W}_0 f(\mathbf{W}_1 \mathbf{x}))$$

A (deep) neural network is a deeply **nested functions**:

- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**



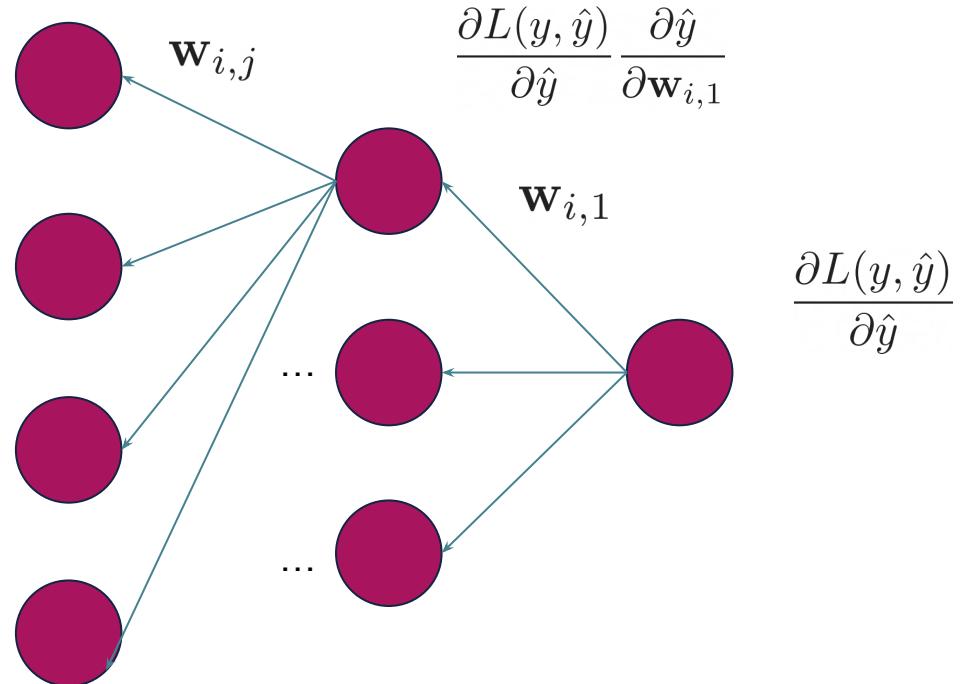
# Backpropagation

$$\hat{y} = g(\mathbf{W}_0 f(\mathbf{W}_1 \mathbf{x}))$$

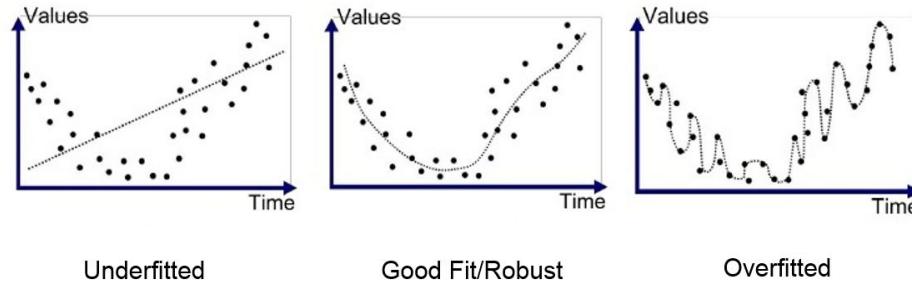
A (deep) neural network is a deeply **nested functions**:

- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**

$$\frac{\partial L(y, \hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{i,j}}$$



# Combatting Overfitting

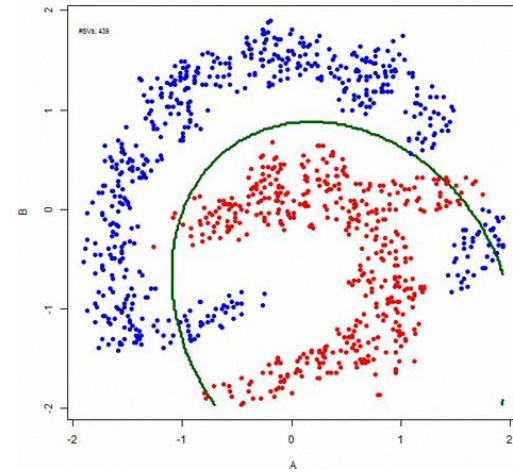
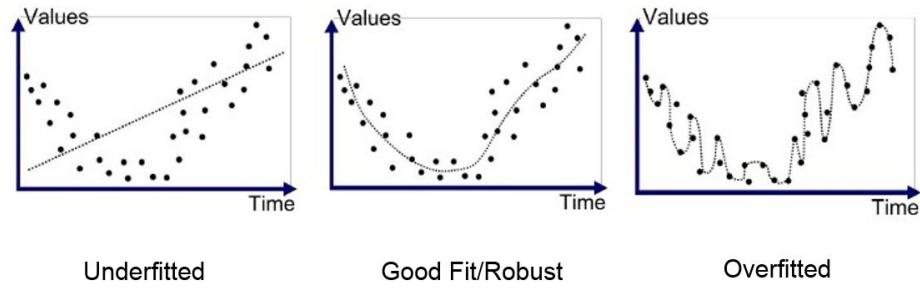


Underfitted

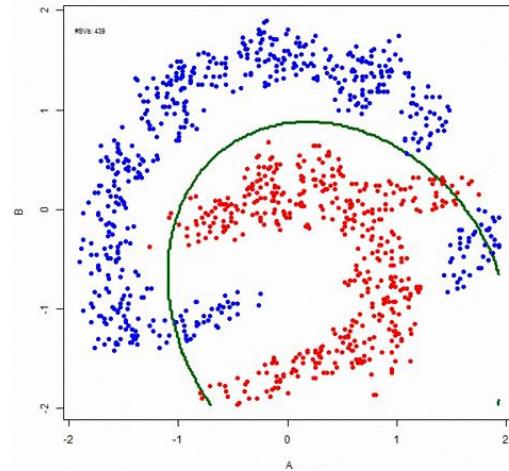
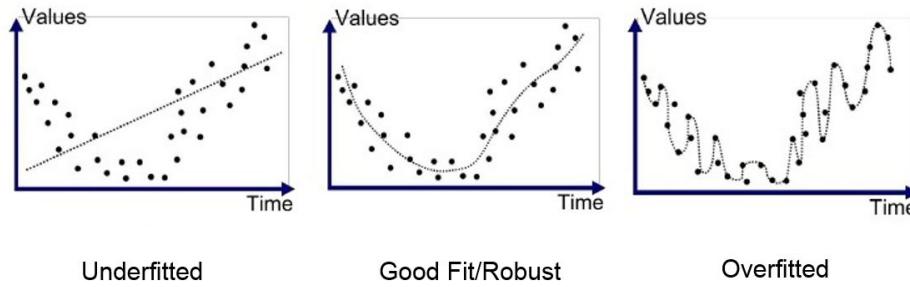
Good Fit/Robust

Overfitted

# Combatting Overfitting

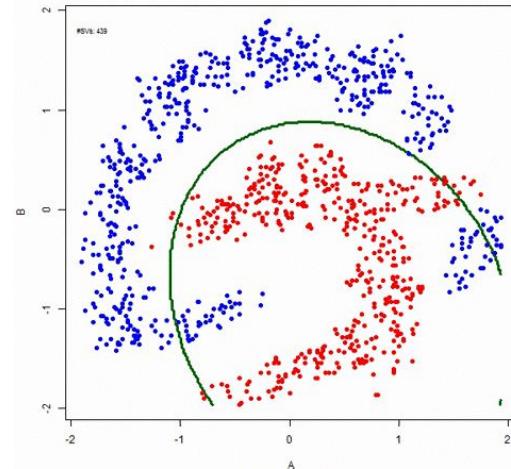
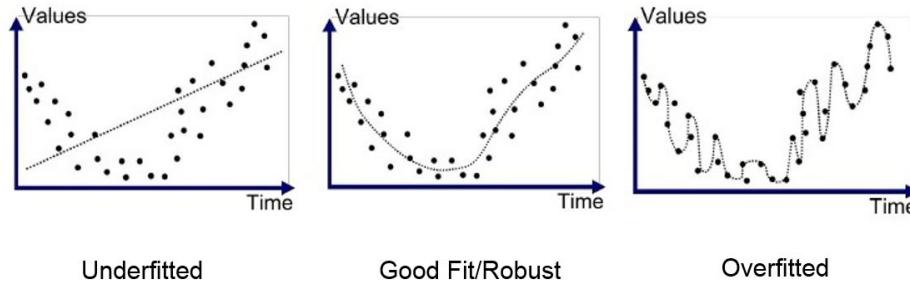


# Combatting Overfitting



- ❖ Dropout prior
- ❖ Weight decay
- ❖ Early stopping
- ❖ Batch Normalization

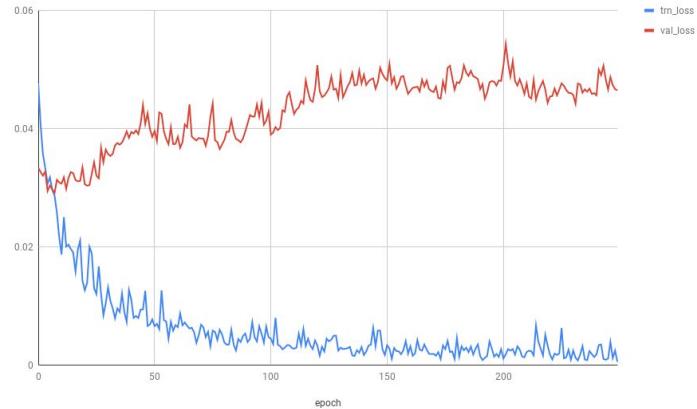
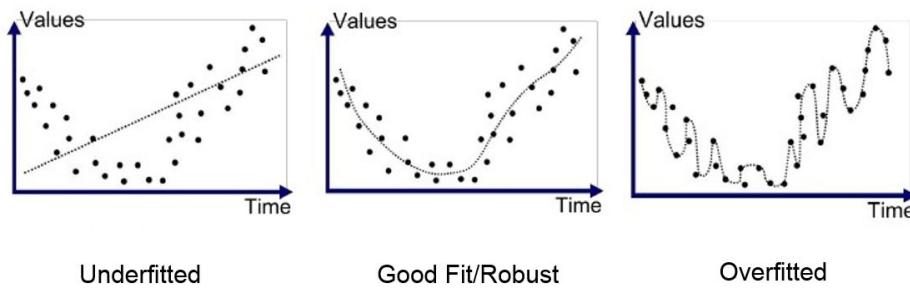
# Combatting Overfitting



- ❖ Dropout prior
- ❖ Weight decay
- ❖ Early stopping
- ❖ Batch Normalization

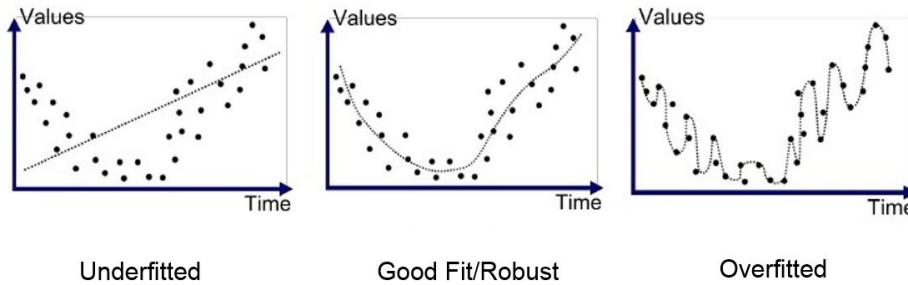
The more weights we need to train, the more complex the model becomes and the sooner it starts to memorize, if we don't have enough data

# Combatting Overfitting

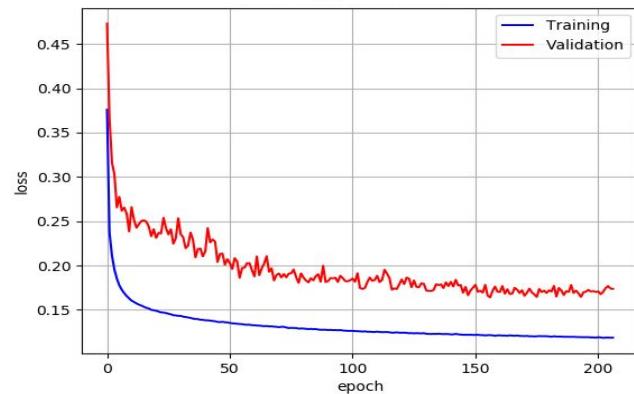
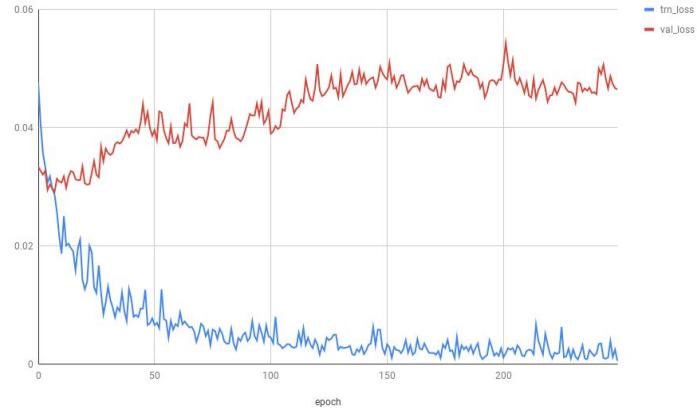


- ❖ Dropout prior
- ❖ Weight decay
- ❖ Early stopping
- ❖ Batch Normalization

# Combatting Overfitting

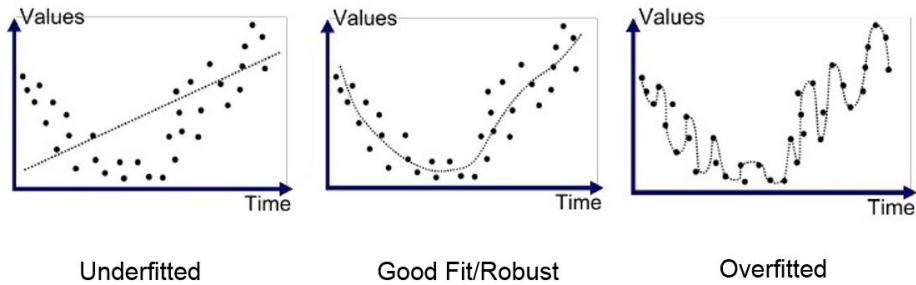


Underfitted      Good Fit/Robust      Overfitted



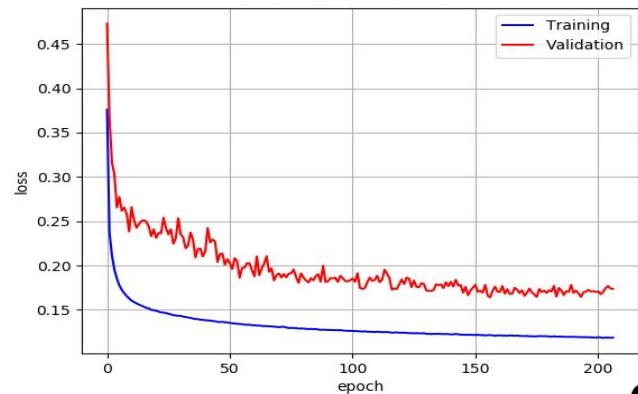
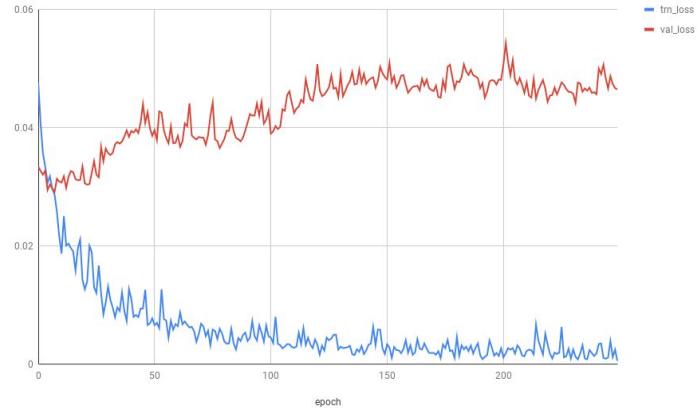
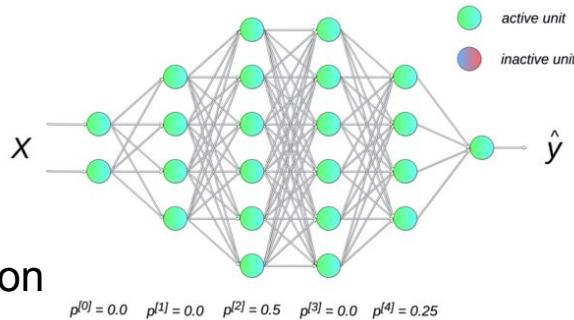
- ❖ Dropout prior
- ❖ Weight decay
- ❖ Early stopping
- ❖ Batch Normalization

# Combatting Overfitting

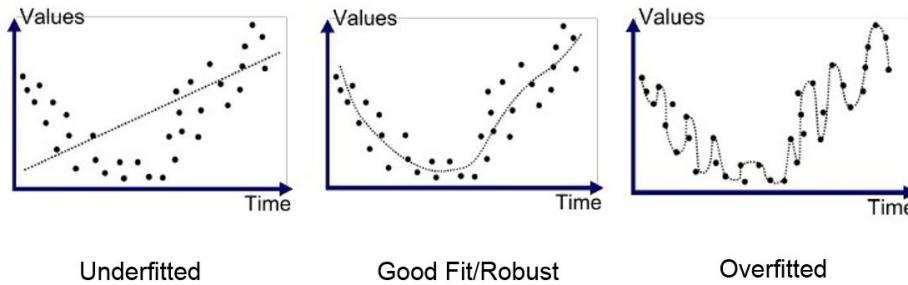


Underfitted      Good Fit/Robust      Overfitted

- ❖ Dropout prior
- ❖ Weight decay
- ❖ Early stopping
- ❖ Batch Normalization

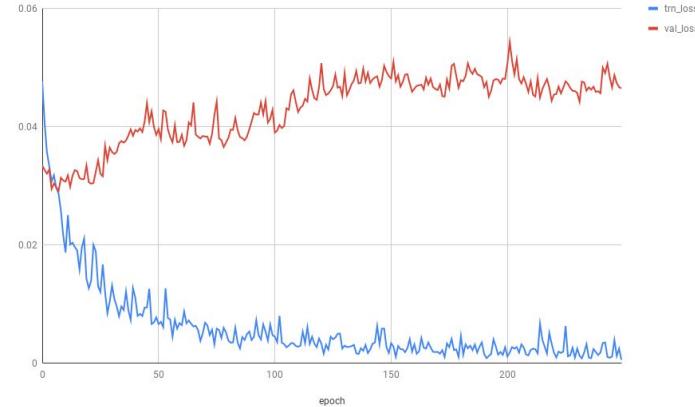
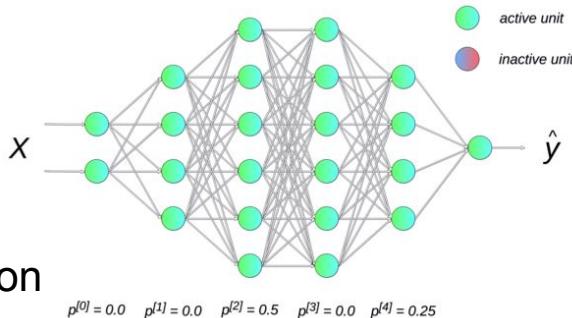


# Combatting Overfitting



Underfitted      Good Fit/Robust      Overfitted

- ❖ Dropout prior
- ❖ Weight decay
- ❖ Early stopping
- ❖ Batch Normalization

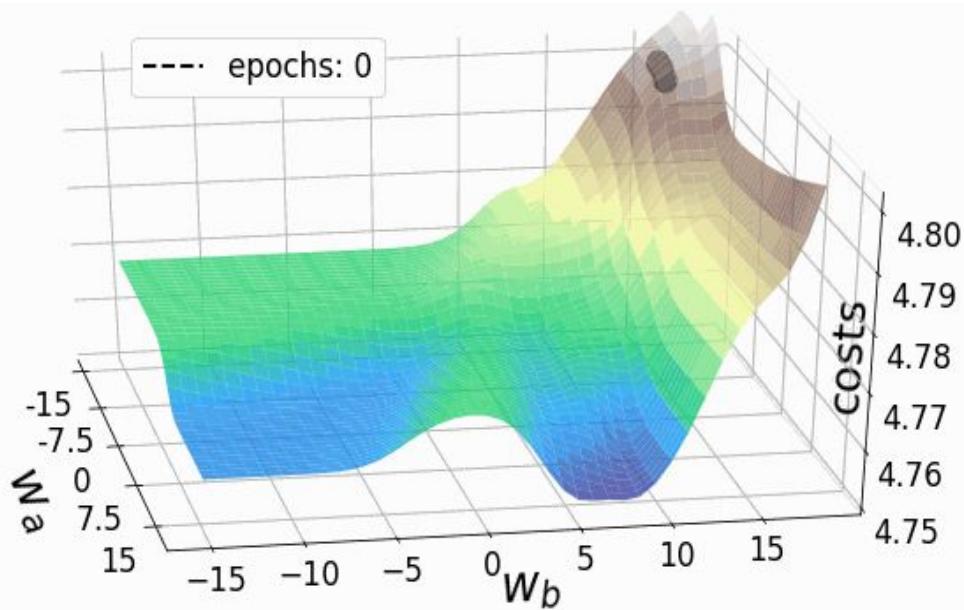
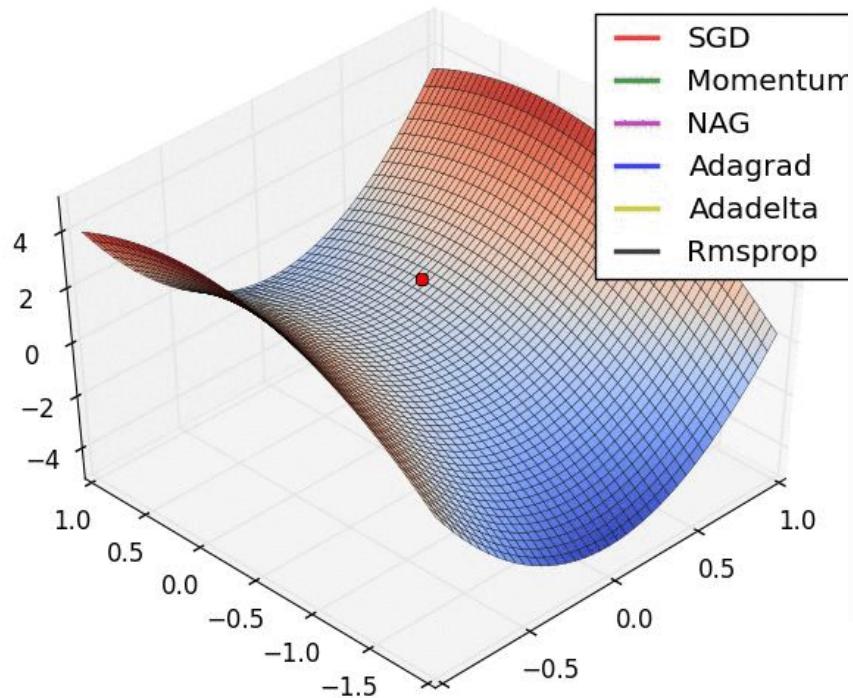


Models always need regularization no matter how big

Not entirely understood how all these tricks amount to a more complex separating hyperplane

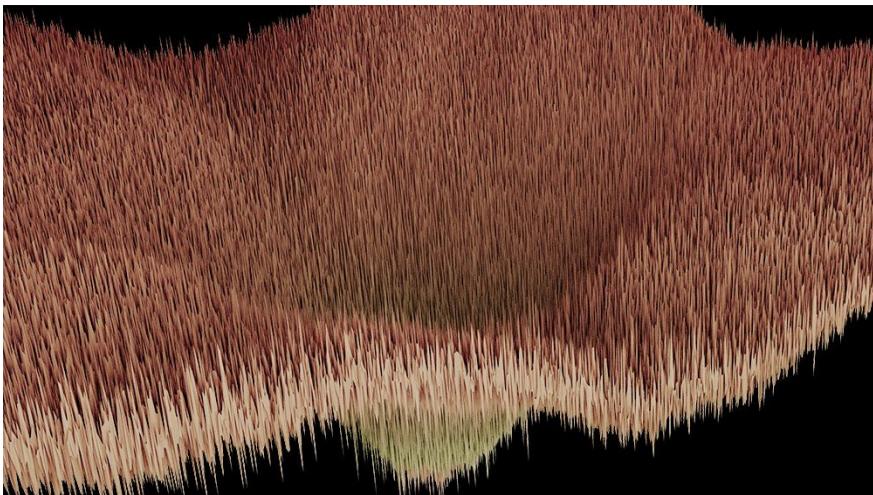
# Optimizers

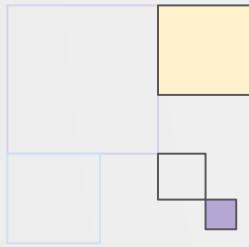
In what way should we **change** the weights?



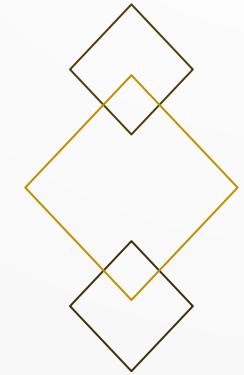
# Optimizers

In what way should we **change** the weights?





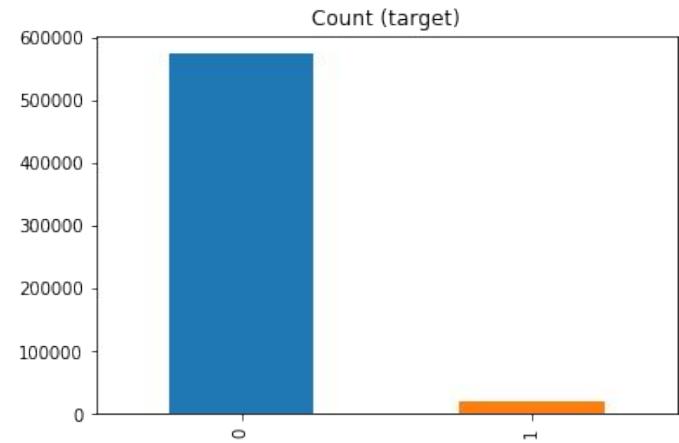
# ML Workflow



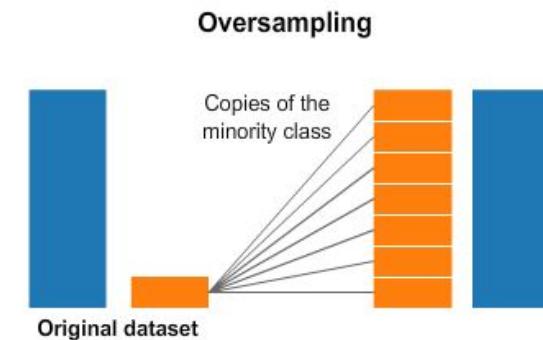
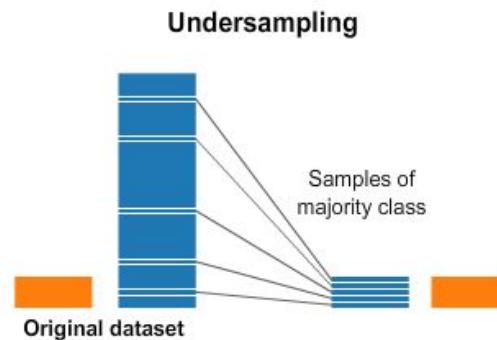
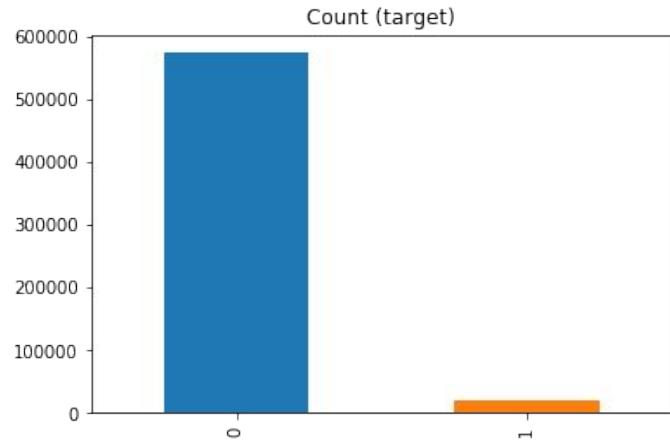
Artificial Intelligence still heavily relies on human intelligence

High Performance Machine Learning Group

# Imbalanced Training set

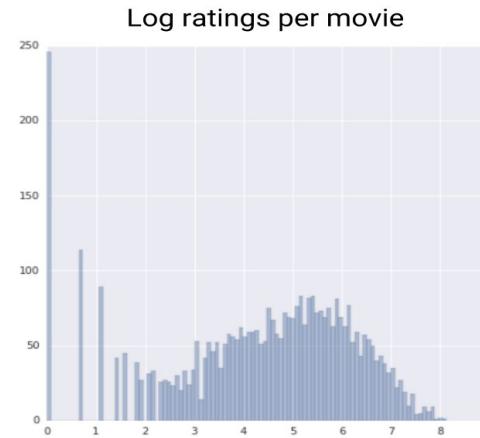
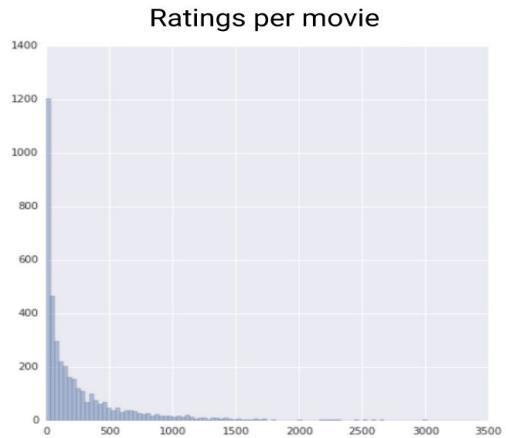
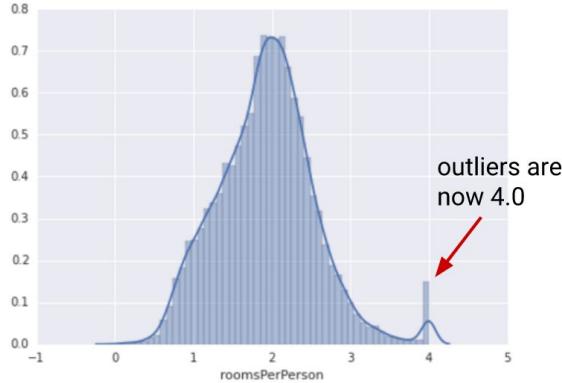
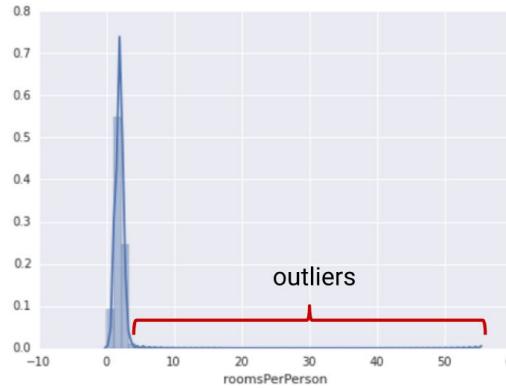


# Imbalanced Training set



# Data normalization

A process to transform the input **data** in a **well-behaved** form



# Open Datasets

## Datasets

Find and use datasets or complete tasks. [Learn more.](#)

+ New Dataset

Help the community by creating and solving Tasks on datasets!



Search 29,853 datasets

Feedback Filter

Sort by: Hottest

PUBLIC



Hotel booking demand

Jesse Mostipak

19 days 1 MB 10.0 1 File (CSV) 1 Task

270



Big Five Personality Test

Bojan Tunguz

14 days 159 MB 9.7 3 Files (CSV, other)

134



StartUp Investments (Crunchbase)

Andy\_M

14 days 3 MB 8.8 1 File (CSV)

92

### Open Tasks

Can we predict the possibility of a bo...

0 Submissions · In Hotel booking demand

Visualize US Accidents Dataset

12 Submissions · In US Accidents (3.0 million...)

What to watch on Netflix ?

4 Submissions · In Netflix Movies and TV Sh...

The state that has the highest number...

5 Submissions · In US Accidents (3.0 million r...)

Processed, balanced,  
well-behaved and labelled  
datasets

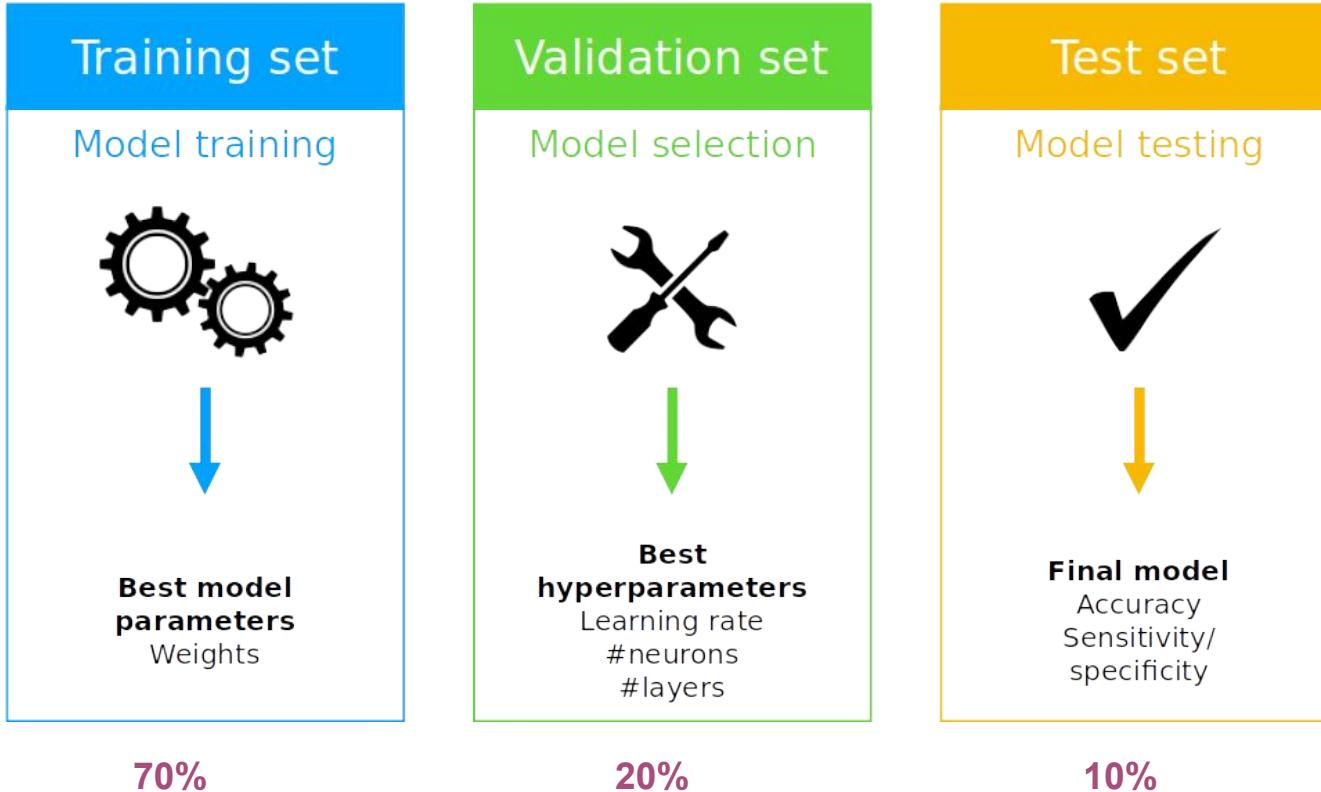
[tensorflow.org/datasets](https://tensorflow.org/datasets)

[kaggle.com/datasets](https://kaggle.com/datasets)

[topepo.github.io/caret/data-sets.html](https://topepo.github.io/caret/data-sets.html)

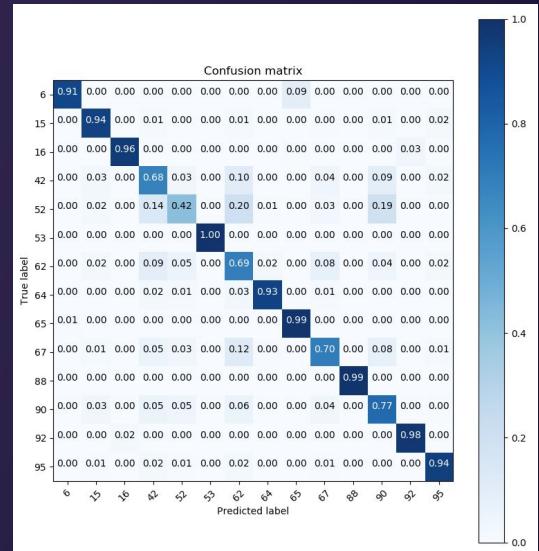
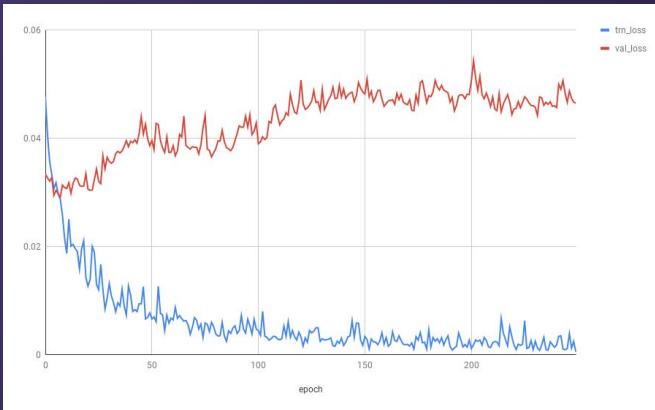
[github.com/awesomedata/awesome-public-datasets](https://github.com/awesomedata/awesome-public-datasets)

# Dataset Splitting



# Network Evaluation

		Prediction outcome		total
		p	n	
actual value	p'	True Positive	False Negative	P'
	n'	False Positive	True Negative	N'
total	P			
	N			



Choose an **appropriate metric** for your own problem

Always **sanity check** your model, is it better than a baseline?

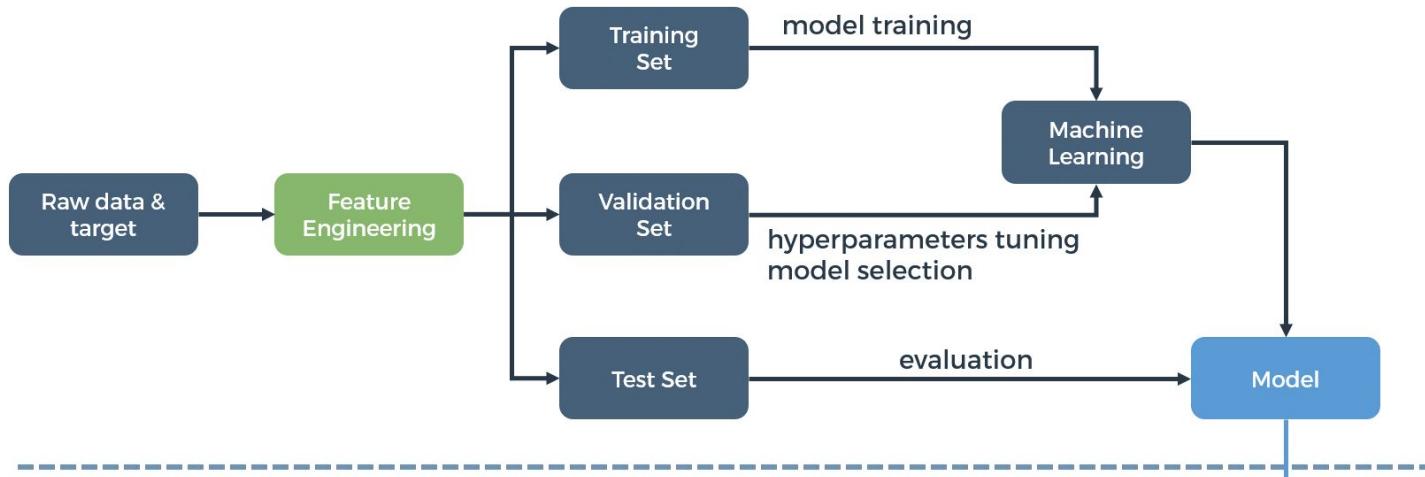
An almost perfect classification score is **always sketchy**

Keep questioning the model, **never trust it**

SURF

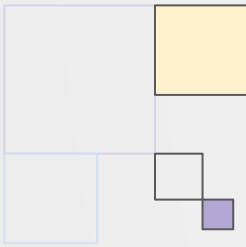
# Workflow

## TRAINING



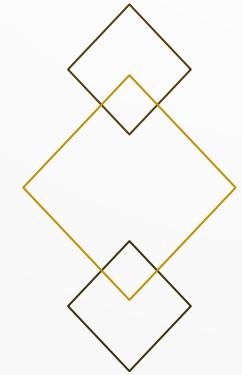
## PREDICTING





# DL Frameworks

Do not compute your own gradients



High Performance Machine Learning Group

SURF

# How to train your NN

- ❖ Define neurons and layers
- ❖ Define loss function
- ❖ Forward propagate and compute loss
- ❖ Compute gradient
- ❖ Propagate backward
- ❖ Update weights



K Keras

theano

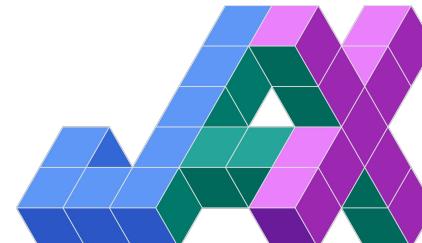


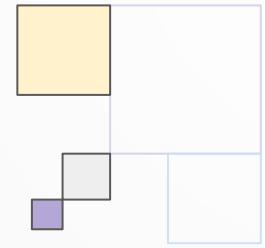
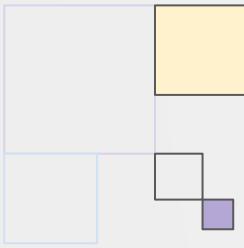
TensorFlow



Microsoft  
Cognitive  
Toolkit

PyTorch





# PyTorch and Modularity



# PyTorch

Three Levels of Abstraction



01. **Tensor:** imperative ndarray,  
possible to run on GPU/TPU
02. (node) **Variable:** Node in the  
built computational graph;  
data, gradient storage
03. (NN) **Module:** A neural  
network layer, store the state  
and the weights of the neural  
network



# PyTorch

Three Levels of Abstraction



01. **Tensor**: imperative ndarray,  
possible to run on GPU/TPU
02. (node) **Variable**: Node in the  
built computational graph;  
data, gradient storage
03. (NN) **Module**: A neural  
network layer, store the state  
and the weights of the neural  
network



# PyTorch

Three Levels of Abstraction



01. **Tensor**: imperative ndarray, possible to run on GPU/TPU
02. (node) **Variable**: Node in the built computational graph; data, gradient storage
03. **(NN) Module**: A neural network layer, store the state and the weights of the neural network



# PyTorch

Three Levels of Abstraction



01. **Tensor**: imperative ndarray,  
possible to run on GPU/TPU
02. (node) **Variable**: Node in the  
built computational graph;  
data, gradient storage
03. (NN) **Module**: A neural  
network layer, store the state  
and the weights of the neural  
network





$$\hat{y} = g(\mathbf{W}_2 f(\mathbf{W}_1 \mathbf{x}))$$

Three Levels of Abstraction



01. **Tensor**: imperative ndarray, possible to run on GPU/TPU
02. (node) **Variable**: Node in the built computational graph; data, gradient storage
03. (NN) **Module**: A neural network layer, store the state and the weights of the neural network



# PyTorch

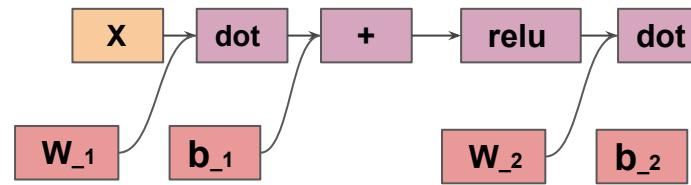
Three Levels of Abstraction



01. **Tensor**: imperative ndarray, possible to run on GPU/TPU
02. (node) **Variable**: Node in the built computational graph; data, gradient storage
03. (NN) **Module**: A neural network layer, store the state and the weights of the neural network



$$\hat{y} = g(\mathbf{W}_2 f(\mathbf{W}_1 \mathbf{x}))$$

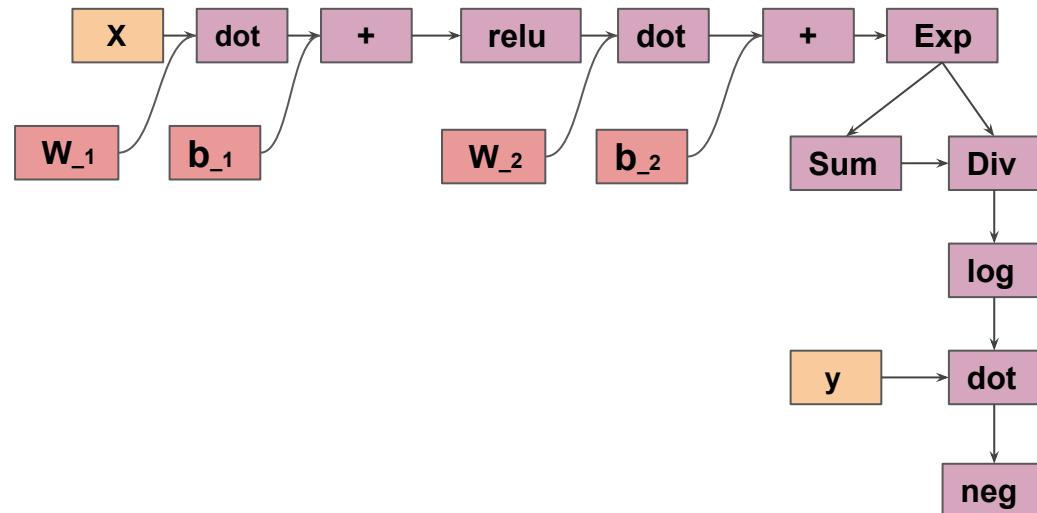


# PyTorch

Three Levels of Abstraction

01. **Tensor**: imperative ndarray, possible to run on GPU/TPU
02. (node) **Variable**: Node in the built computational graph; data, gradient storage
03. (NN) **Module**: A neural network layer, store the state and the weights of the neural network

$$\hat{y} = g(\mathbf{W}_2 f(\mathbf{W}_1 \mathbf{x}))$$



# PyTorch

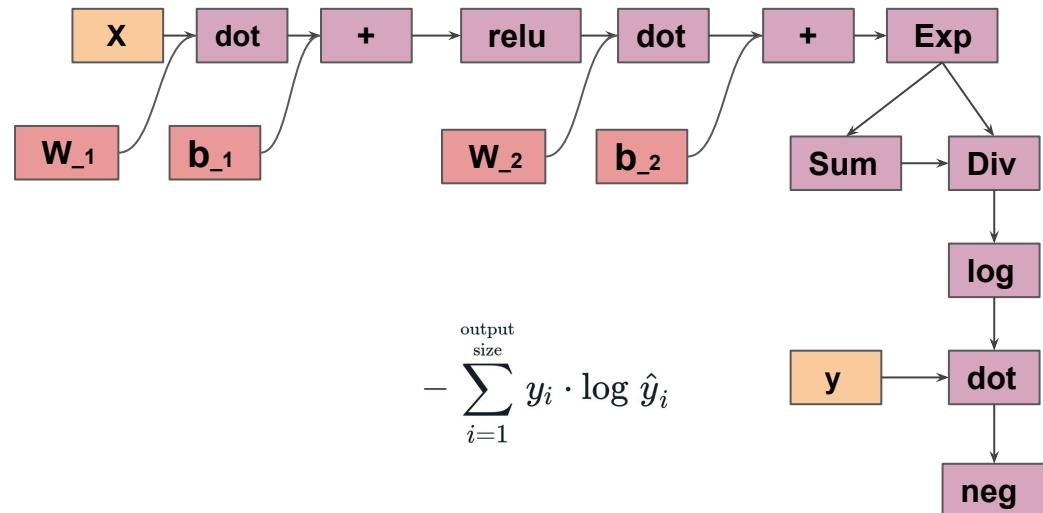
Three Levels of Abstraction



01. **Tensor**: imperative ndarray, possible to run on GPU/TPU
02. (node) **Variable**: Node in the built computational graph; data, gradient storage
03. (NN) **Module**: A neural network layer, store the state and the weights of the neural network



$$\hat{y} = g(\mathbf{W}_2 f(\mathbf{W}_1 \mathbf{x}))$$



# PyTorch

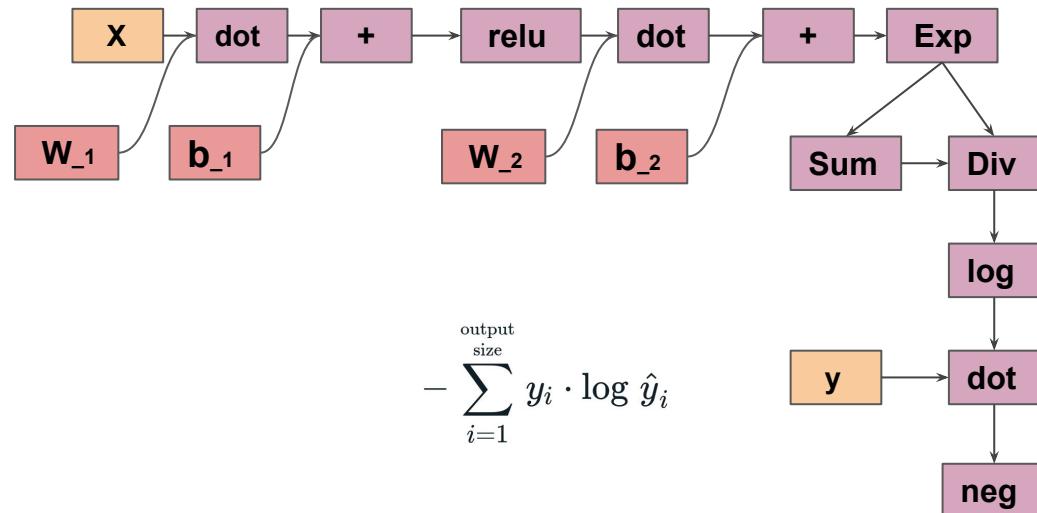
Three Levels of Abstraction



01. **Tensor**: imperative ndarray, possible to run on GPU/TPU
02. (node) **Variable**: Node in the built computational graph; data, gradient storage
03. (NN) **Module**: A neural network layer, store the state and the weights of the neural network



$$\hat{y} = g(\mathbf{W}_2 f(\mathbf{W}_1 \mathbf{x}))$$



Pytorch will help us with

- ❖ Defining a dataset
- ❖ Automatic Gradient Computation
- ❖ Defining Neural Networks
- ❖ Optimization
- ❖ Scheduling
- ❖ Distributing

[Search Docs](#)[Notes \[ + \]](#)[Language Bindings \[ + \]](#)[Python API \[ - \]](#)

torch

torch.nn

torch.nn.functional

torch.Tensor

Tensor Attributes

Tensor Views

torch.autograd

torch.cuda

torch.cuda.amp

torch.backends

torch.distributed

torch.distributed.algorithms.join

torch.distributed.elastic

torch.distributed.fsdp

torch.distributed.optim

torch.distributions

torch.fft

torch.futures

torch.fx

torch.hub

# TORCH.NN

These are the basic building blocks for graphs:

torch.nn

- [Containers](#)
- [Convolution Layers](#)
- [Pooling layers](#)
- [Padding Layers](#)
- [Non-linear Activations \(weighted sum, nonlinearity\)](#)
- [Non-linear Activations \(other\)](#)
- [Normalization Layers](#)
- [Recurrent Layers](#)
- [Transformer Layers](#)
- [Linear Layers](#)
- [Dropout Layers](#)
- [Sparse Layers](#)
- [Distance Functions](#)
- [Loss Functions](#)
- [Vision Layers](#)
- [Shuffle Layers](#)
- [DataParallel Layers \(multi-GPU, distributed\)](#)
- [Utilities](#)
- [Quantized Functions](#)
- [Lazy Modules Initialization](#)

torch.nn

+ Containers

Convolution Layers

Pooling layers

Padding Layers

Non-linear Activations (weighted sum, nonlinearity)

Non-linear Activations (other)

Normalization Layers

Recurrent Layers

Transformer Layers

Linear Layers

Dropout Layers

Sparse Layers

Distance Functions

Loss Functions

Vision Layers

Shuffle Layers

DataParallel Layers (multi-GPU, distributed)

+ Utilities

Quantized Functions

Lazy Modules Initialization

<https://pytorch.org/docs/stable/>



## General Training Structure

**data loader**  
**model**  
**optimizer**  
**loss function**





## General Training Structure

**data loader**  
**model**  
**optimizer**  
**loss function**

For every datapoint,  $y$  in **data\_loader**





## General Training Structure

**data loader**  
**model**  
**optimizer**  
**loss function**

For every datapoint,  $y$  in **data\_loader**  
    **optimizer.zero\_grad()**



## General Training Structure



**data loader**  
**model**  
**optimizer**  
**loss function**

For every datapoint,  $y$  in **data\_loader**  
    **optimizer.zero\_grad()**  
    prediction = **model**(datapoint)





## General Training Structure



**data loader**  
**model**  
**optimizer**  
**loss function**

For every datapoint,  $y$  in **data\_loader**

```
    optimizer.zero_grad()
    prediction = model(datapoint)
    loss = loss_function(prediction, y)
```





## General Training Structure



**data loader**  
**model**  
**optimizer**  
**loss function**

For every datapoint,  $y$  in **data\_loader**

```
optimizer.zero_grad()  
prediction = model(datapoint)  
loss = loss_function(prediction, y)  
loss.backward()
```



# PyTorch

## General Training Structure

**data loader**  
**model**  
**optimizer**  
**loss function**

For every datapoint,  $y$  in **data\_loader**

```
optimizer.zero_grad()  
prediction = model(datapoint)  
loss = loss_function(prediction, y)  
loss.backward()  
optimizer.step()
```

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \alpha \nabla L(\mathbf{w}_j, b)$$



## General Training Structure

**data loader**  
**model**  
**optimizer**  
**loss function**

For every datapoint,  $y$  in **data\_loader**

```
optimizer.zero_grad()
prediction = model(datapoint)
loss = loss_function(prediction, y)
loss.backward()
optimizer.step()
```

```
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)

    optimizer.zero_grad()
    output = model(data)
    loss = F.nll_loss(output, target)
    loss.backward()
    optimizer.step()
```

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \alpha \nabla L(\mathbf{w}_j, b)$$



**Data:**

d\_1 = [0.9, -0.2], y = 0  
d\_2 = [0.75, 0.6],y = 1

**Data:**

d\_1 = [0.9, -0.2], y = 0  
d\_2 = [0.75, 0.6],y = 1

**Define Neural Network**

Input size of 2  
One hidden layer of 8 nodes  
1 output node (binary)

**Data:**

d\_1 = [0.9, -0.2], y = 0  
d\_2 = [0.75, 0.6], y = 1

**Define Neural Network**

Input size of 2  
One hidden layer of 8 nodes  
1 output node (binary)

Learning rate = 0.01

Optimizer = Stochastic Gradient Descent

Loss = Binary Cross Entropy

**Data:**

d\_1 = [0.9, -0.2], y = 0  
d\_2 = [0.75, 0.6], y = 1

**Define Neural Network**

Input size of 2  
One hidden layer of 8 nodes  
1 output node (binary)

Learning rate = 0.01

Optimizer = Stochastic Gradient Descent

Loss = Binary Cross Entropy

$$\text{Loss} = - \sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

# PyTorch

## Data:

$d_1 = [0.9, -0.2], y = 0$   
 $d_2 = [0.75, 0.6], y = 1$

## Define Neural Network

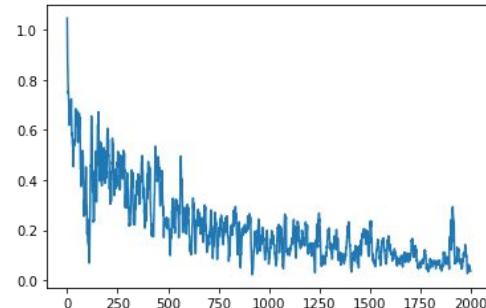
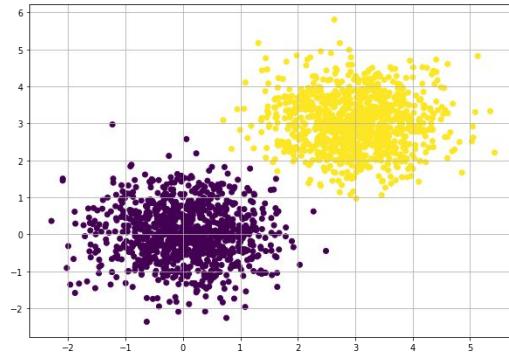
Input size of 2  
One hidden layer of 8 nodes  
1 output node (binary)

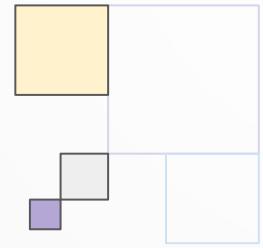
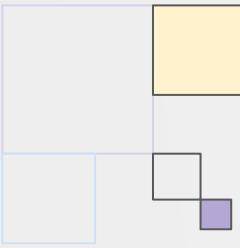
Learning rate = 0.01

Optimizer = Stochastic Gradient Descent

Loss = Binary Cross Entropy

$$\text{Loss} = - \sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$





# Fin

Introduction Series

<https://jupyter.snellius.surf.nl/jhssrf004>

Bryan Cardenas  
Monica Rotulo  
Caspar van Leeuwen  
Robert Jan Schlimbach

