



Deep Learning From Scratch

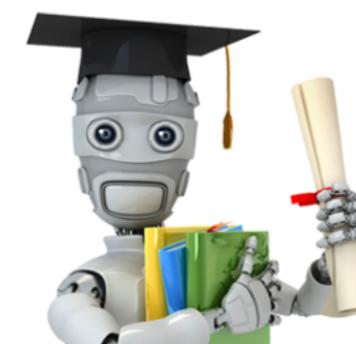
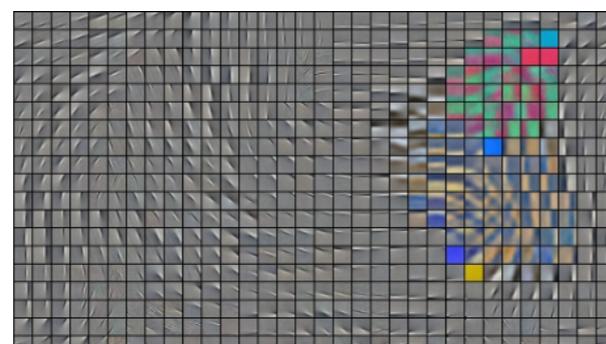
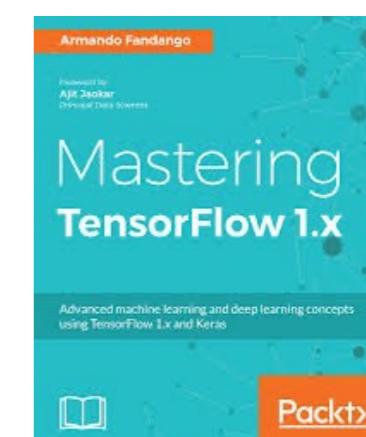
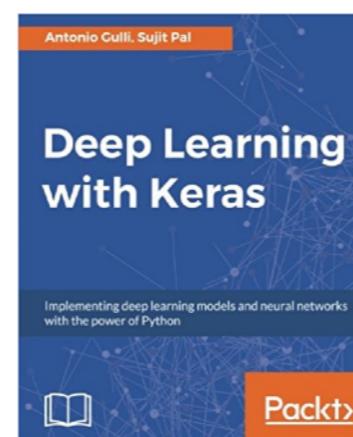
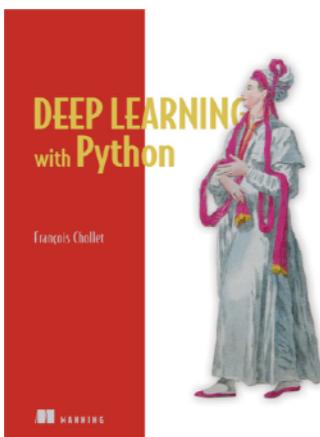
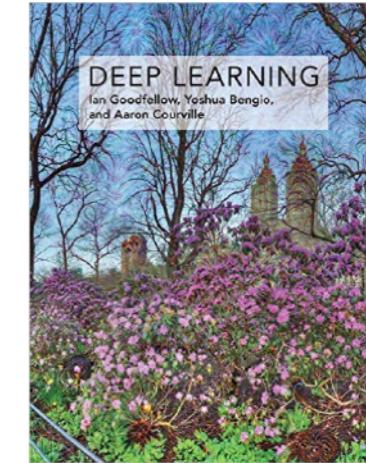
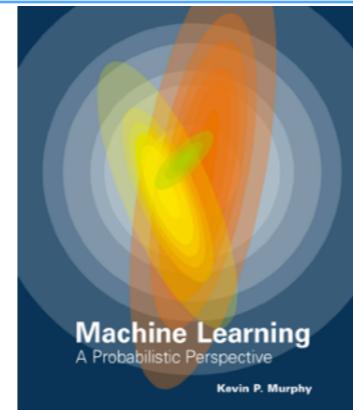
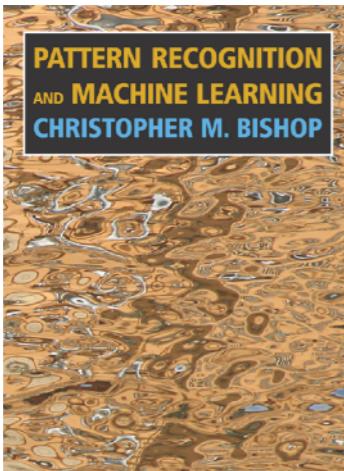
Bruno Gonçalves

www.data4sci.com/newsletter

<https://github.com/DataForScience/DeepLearning>

References

<https://github.com/DataForScience/DeepLearning>



Neural Networks for Machine Learning
Geoff Hinton

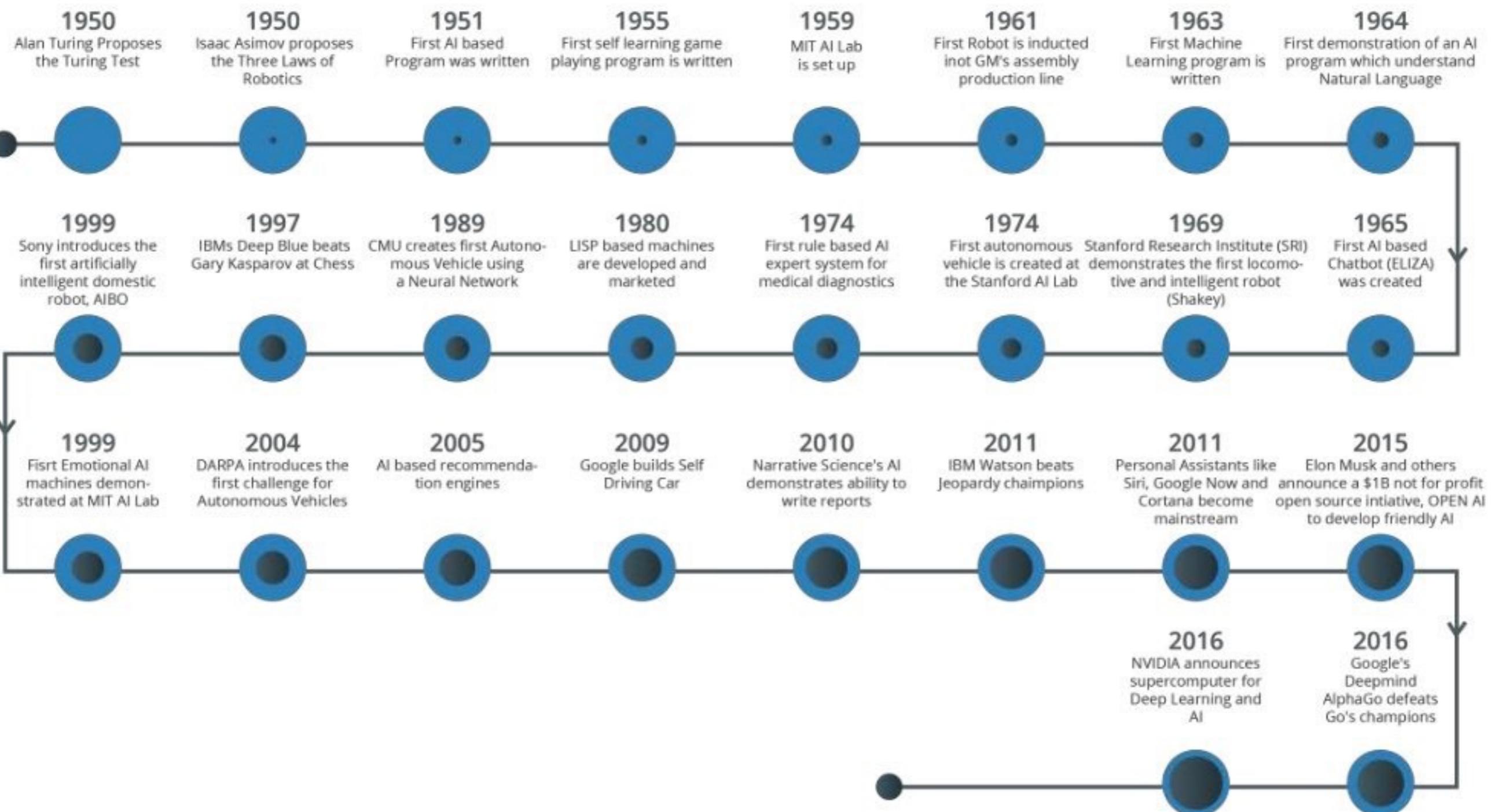
Machine Learning
Andrew Ng

Requirements

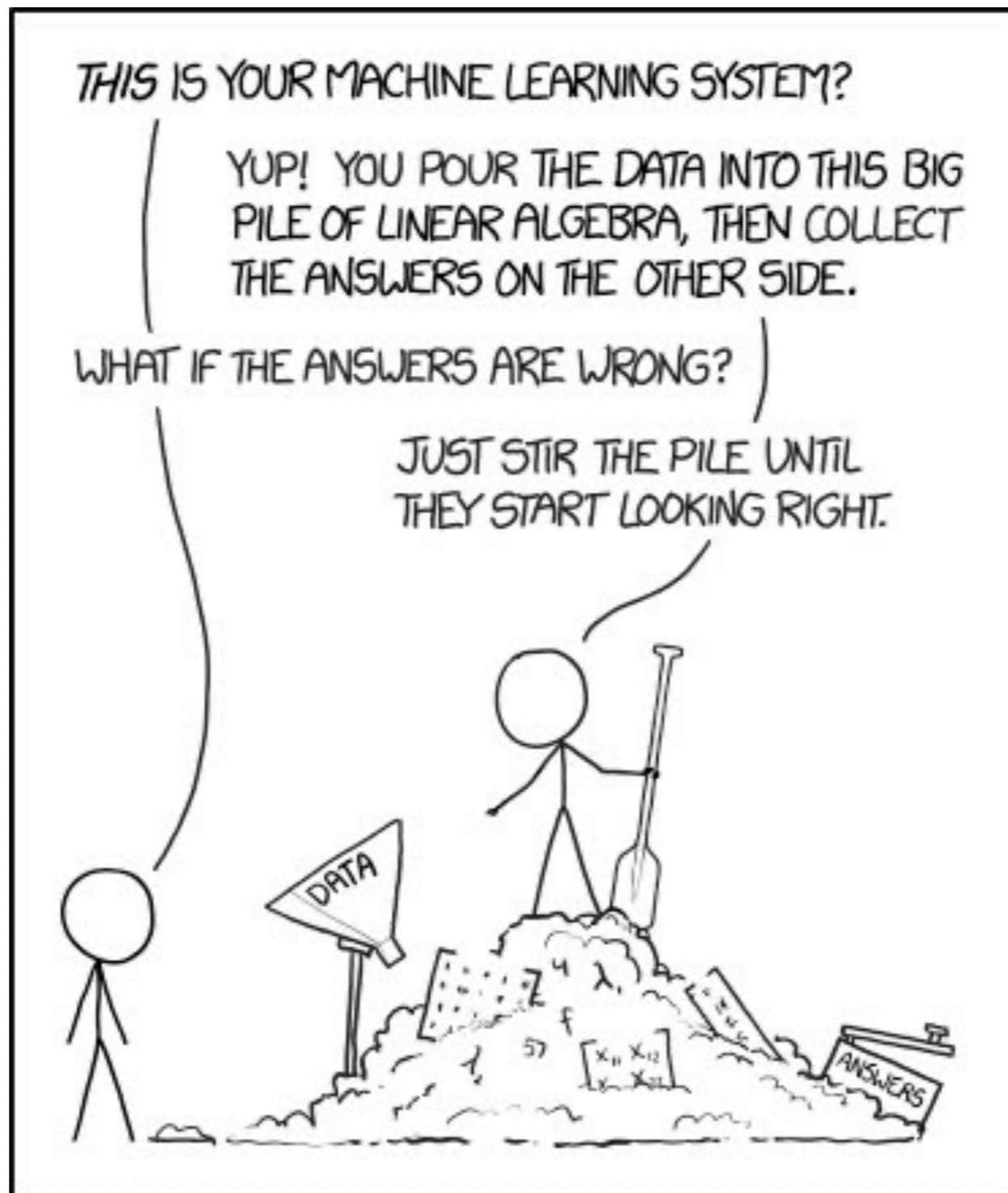
<https://github.com/DataForScience/DeepLearning>

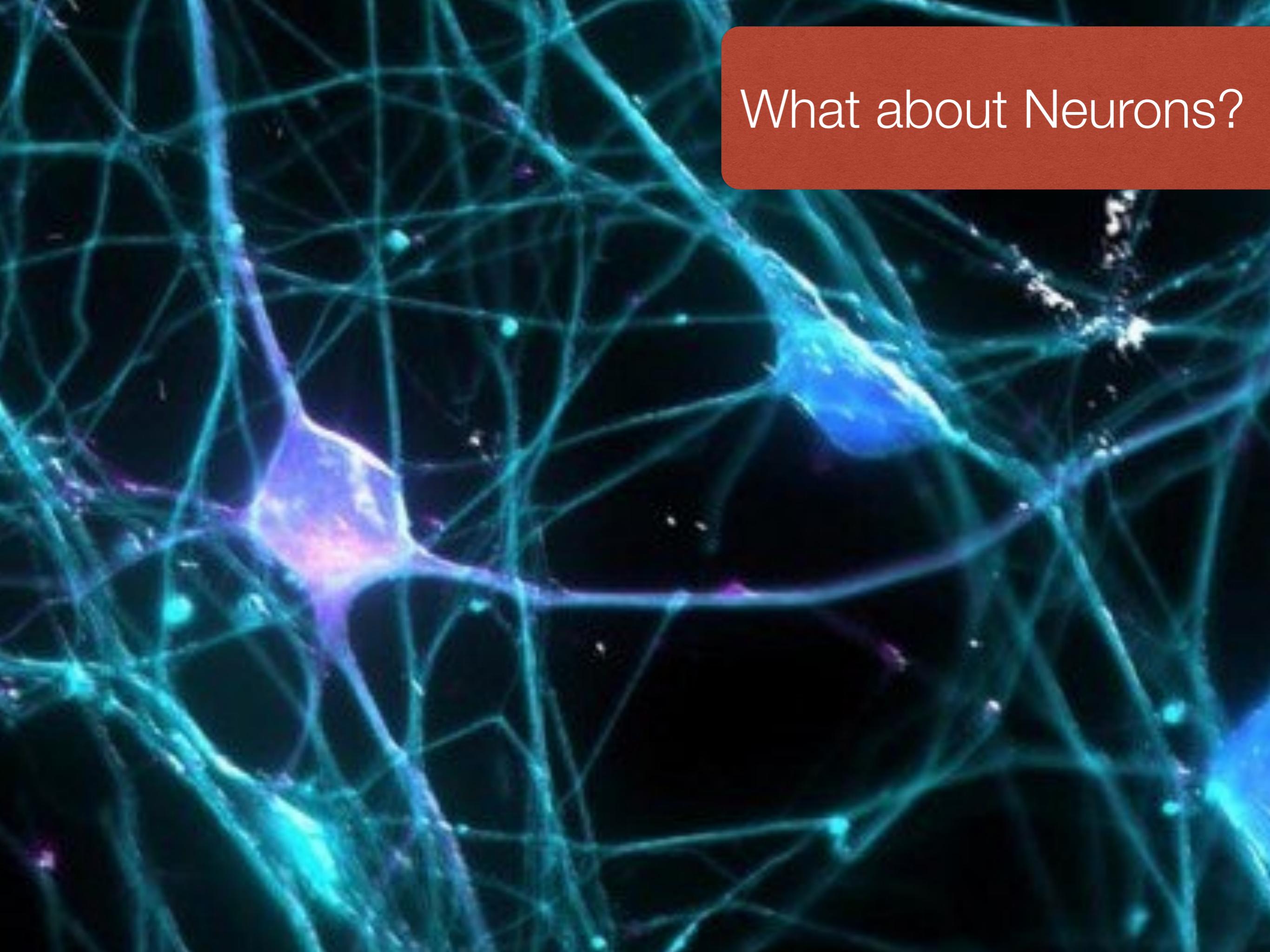


AI Key Milestone Events



Machine Learning



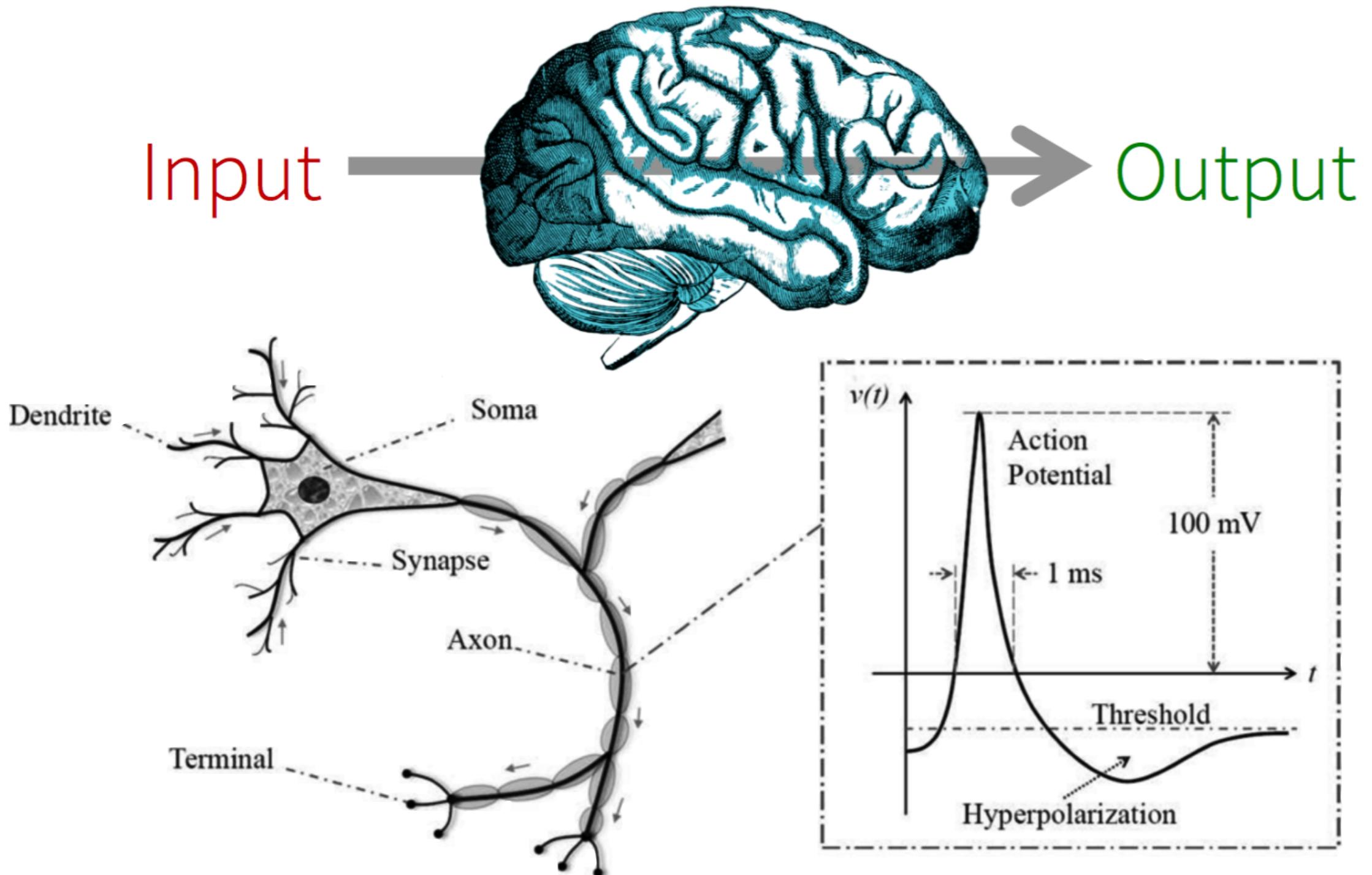
A dense network of glowing blue and purple neurons against a black background.

What about Neurons?

How the Brain “Works” (Cartoon version)

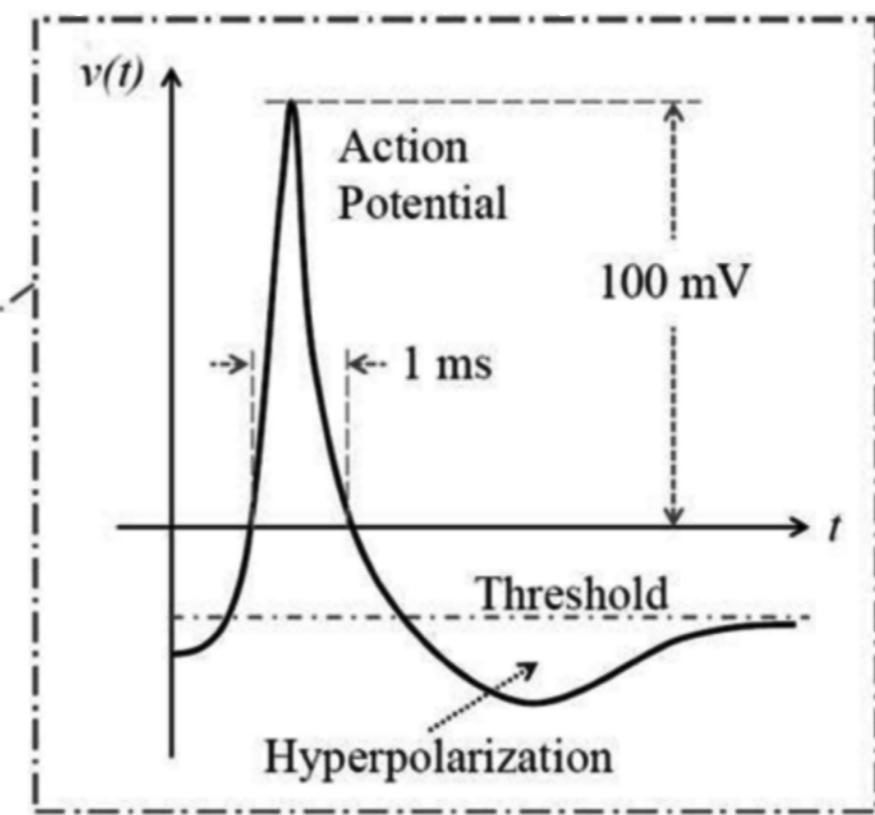
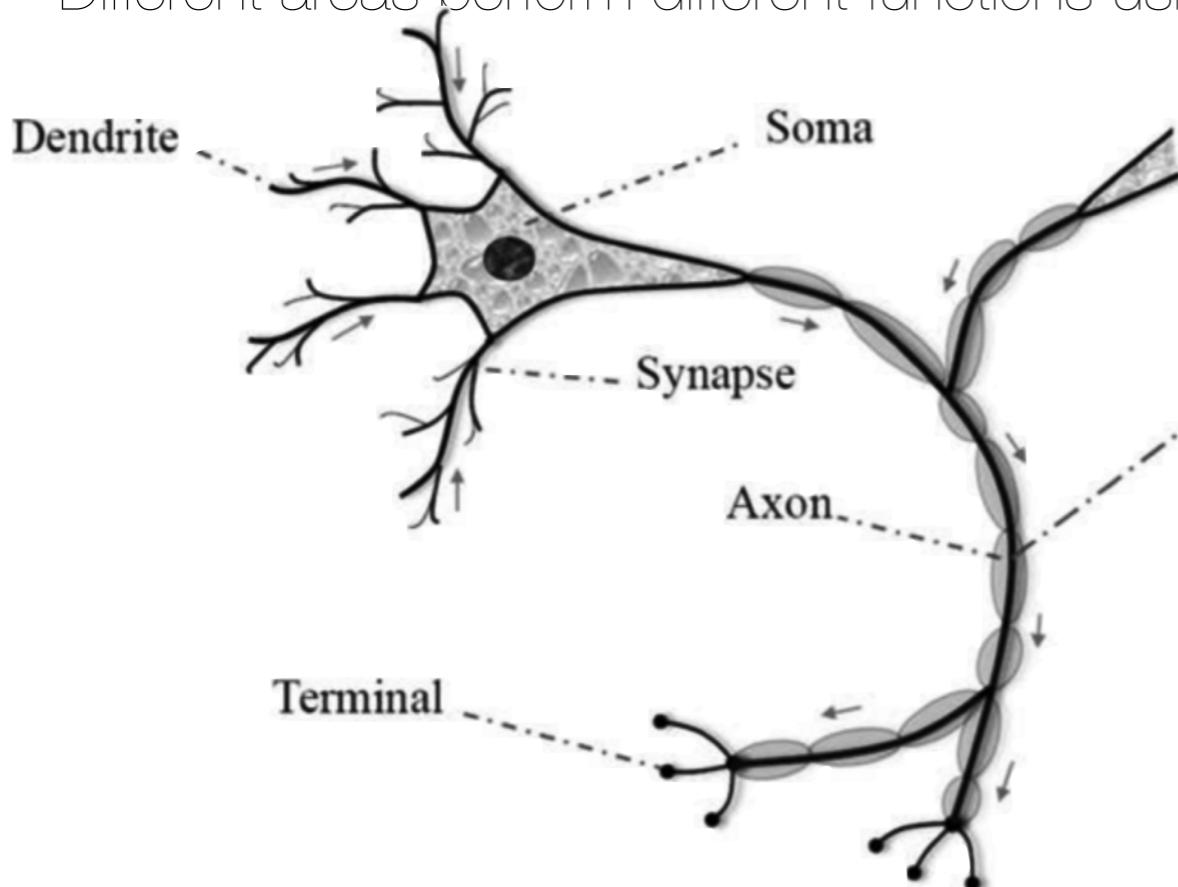


How the Brain “Works” (Cartoon version)

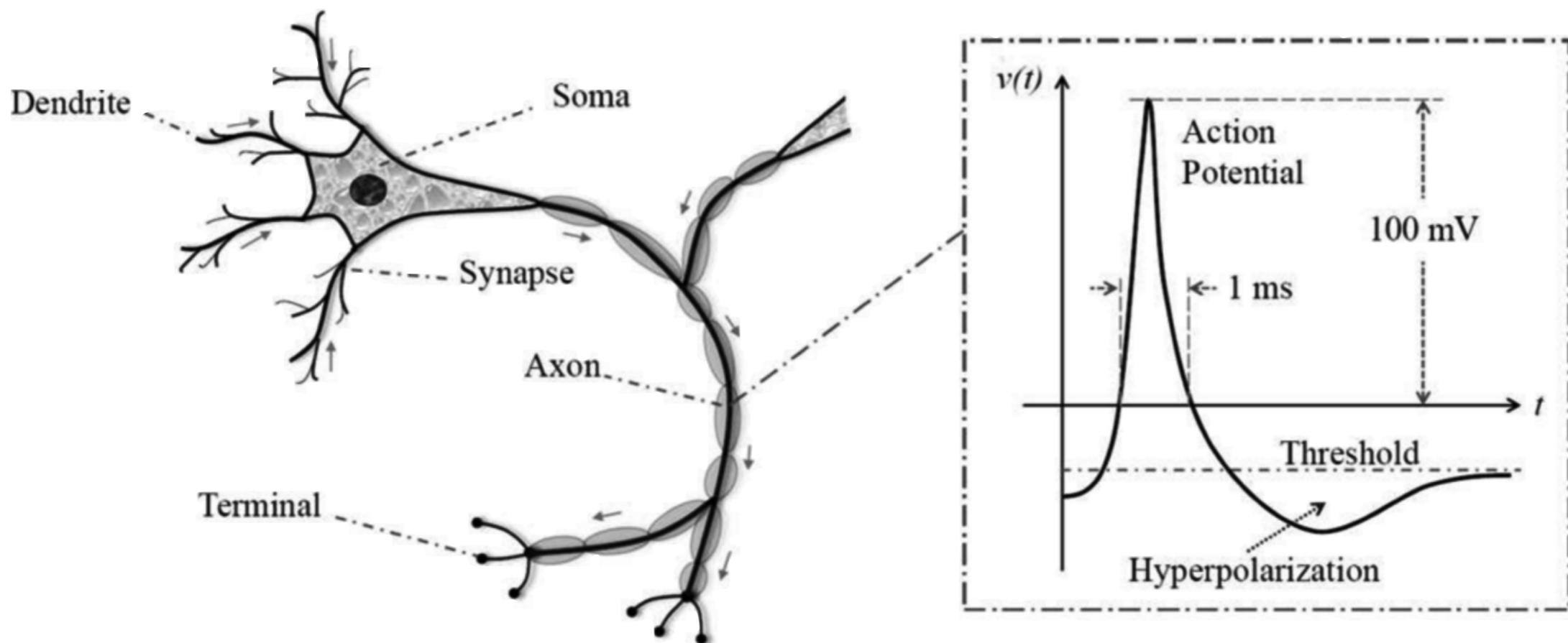
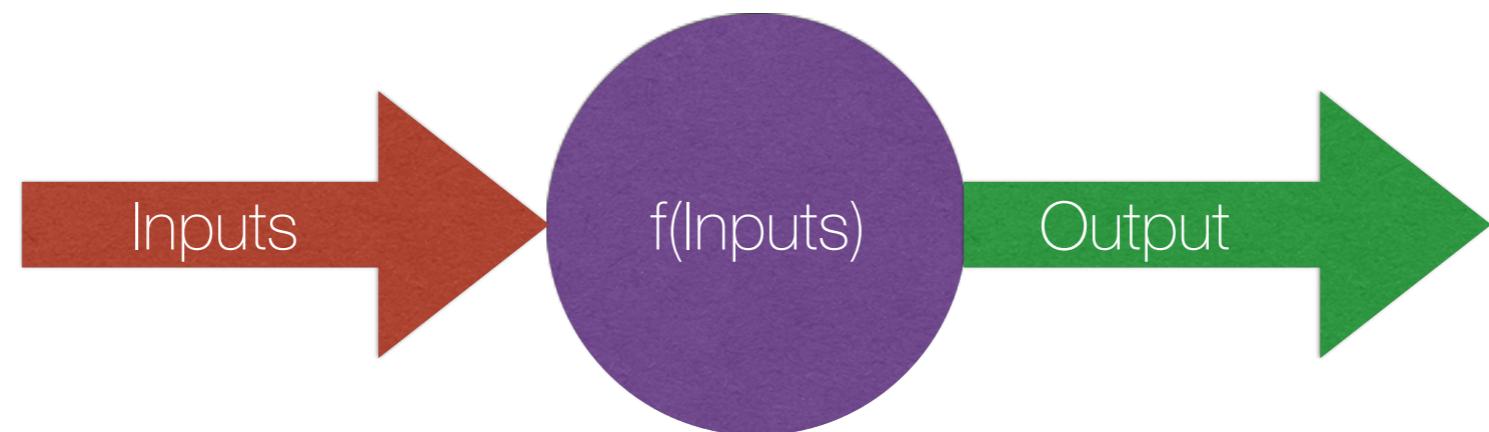


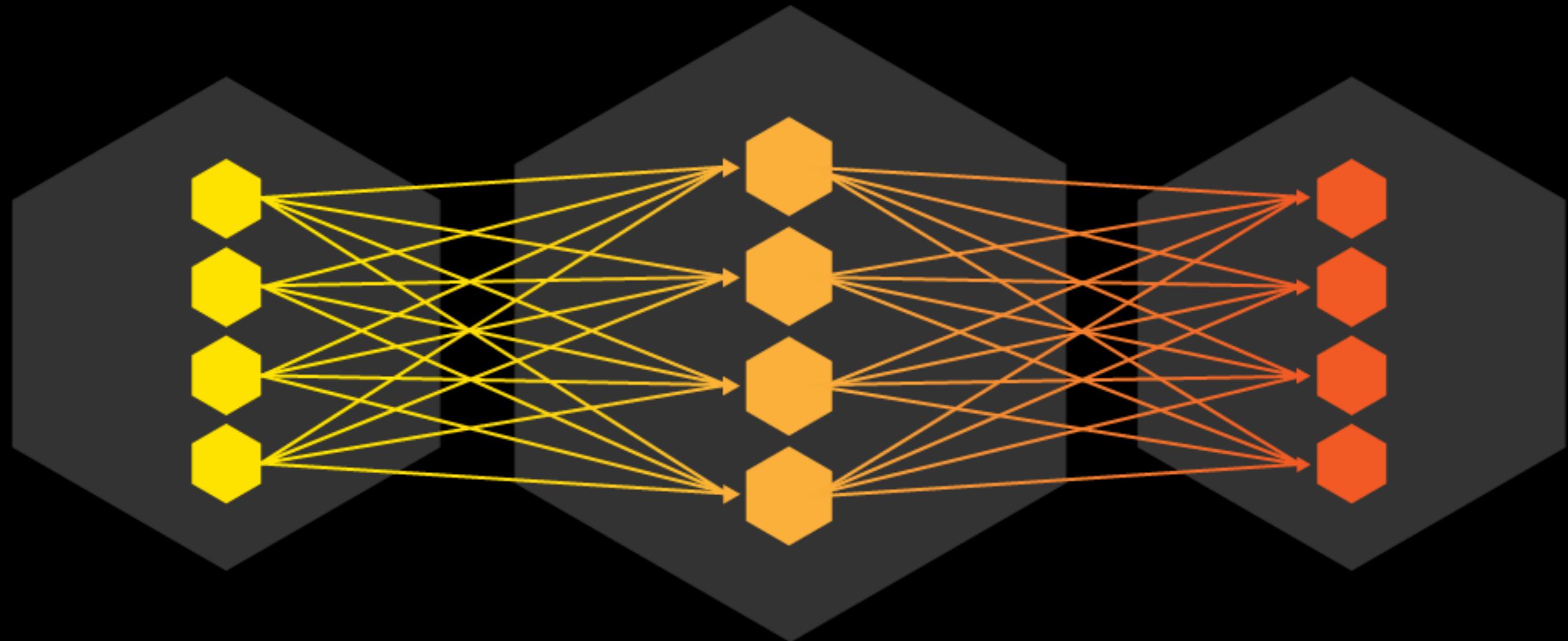
How the Brain “Works” (Cartoon version)

- Each neuron receives input from other neurons
- 10^{11} neurons, each with 10^4 weights
- Weights can be positive or negative
- Weights adapt during the learning process
- “neurons that fire together wire together” (Hebb)
- Different areas perform different functions using same structure (**Modularity**)



How the Brain “Works” (Cartoon version)





INPUT TERMS

FEATURES
PREDICTIONS
ATTRIBUTES
PREDICTABLE VARIABLES

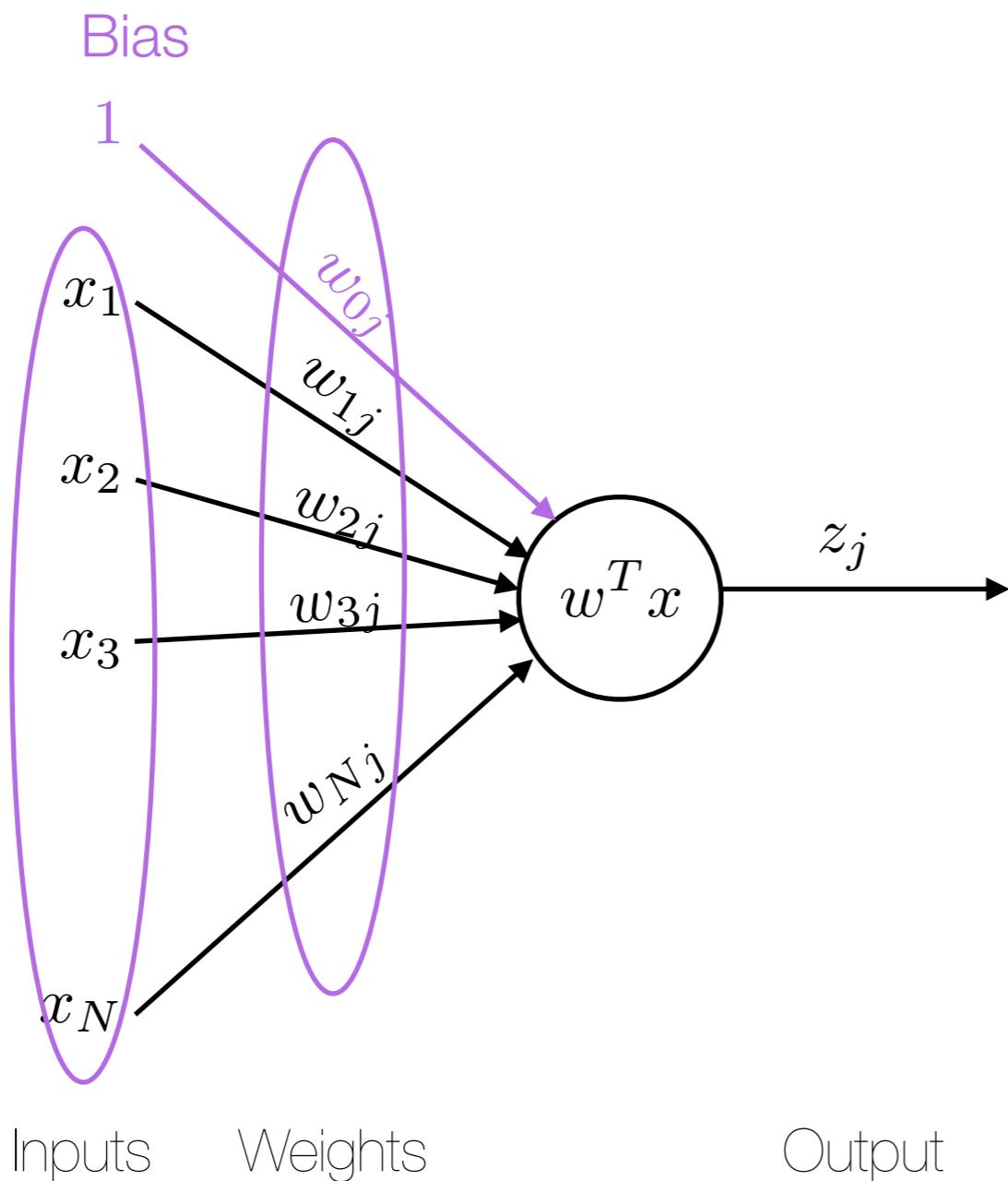
MACHINE

ALGORITHMS
TECHNIQUES
MODELS

OUTPUT TERMS

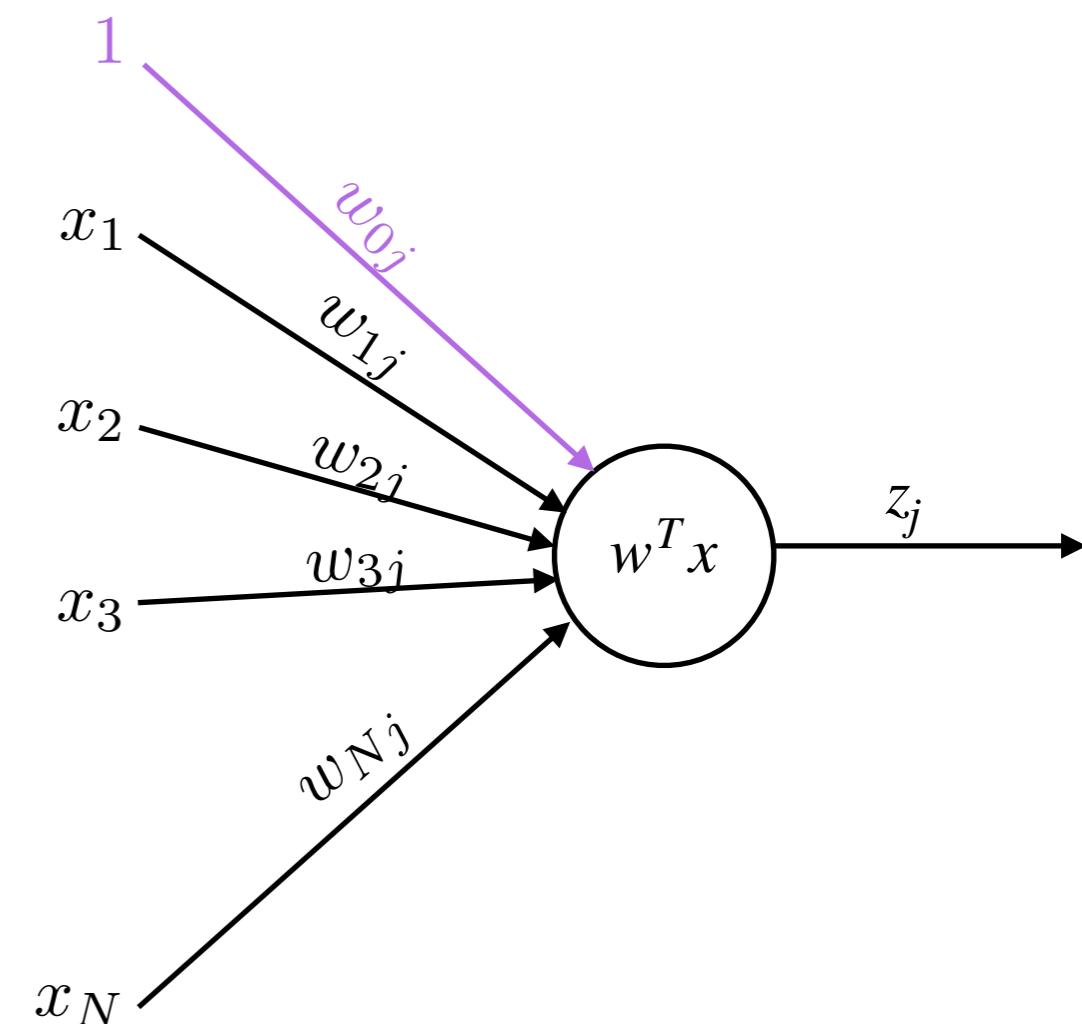
CLASSES
RESPONSES
TARGETS
DEPENDANT VARIABLES

Perceptron



Perceptron - Forward Propagation

- The output of a perceptron is determined by a sequence of steps:
 - obtain the inputs
 - multiply the inputs by the respective weights



Perceptron - Forward Propagation

```
def forward(Theta, X, active):
    N = X.shape[0]

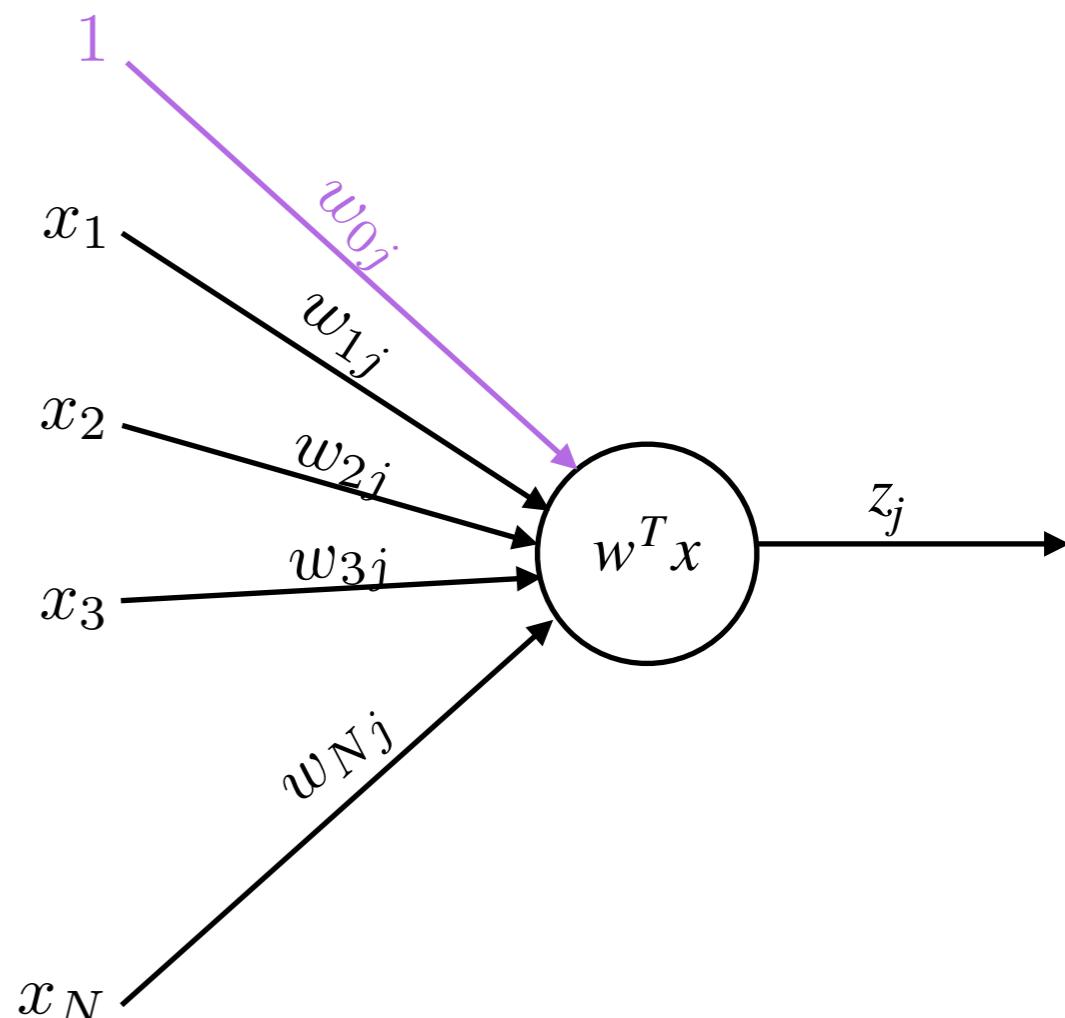
    # Add the bias column
    X_ = np.concatenate((np.ones((N, 1)), X), 1)

    # Multiply by the weights
    z = np.dot(X_, Theta.T)

    return a
```

Perceptron - Training

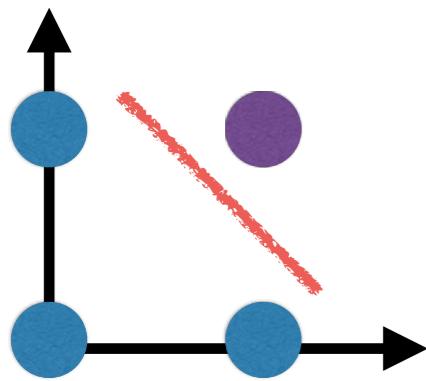
- Training Procedure:
 - If **correct**, do **nothing**
 - If output **incorrectly** outputs 0, **add** input to weight vector
 - if output **incorrectly** outputs 1, **subtract** input to weight vector
- **Guaranteed to converge**, if a correct set of weights exists
- Given enough features, perceptrons **can learn almost anything**
- Specific Features used limit what is possible to learn



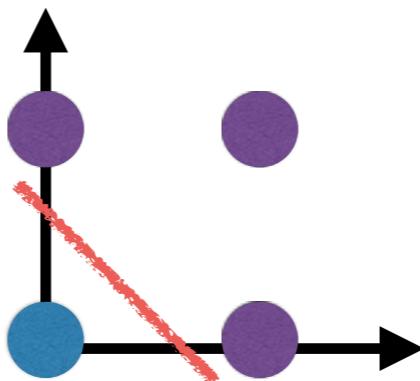
Linear boundaries

1
0

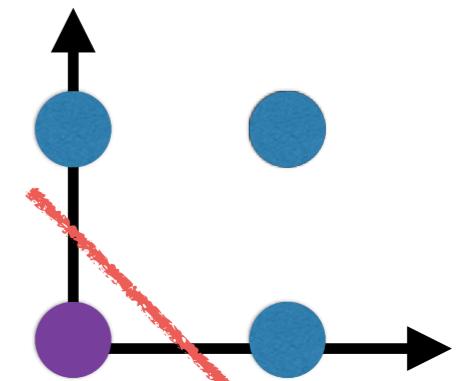
AND



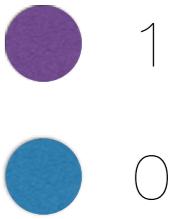
OR



NOR

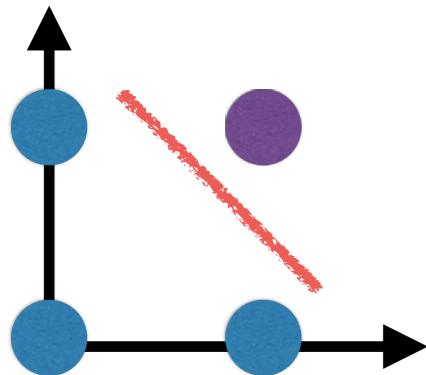


Linear boundaries



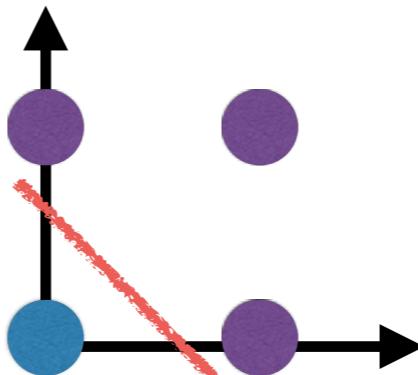
- Perceptrons rely on hyperplanes to separate the data points. Unfortunately, this is not always possible:

AND



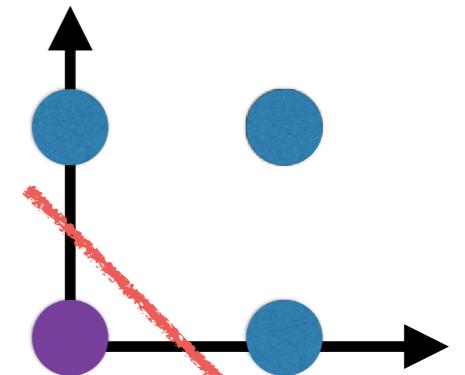
Possible

OR



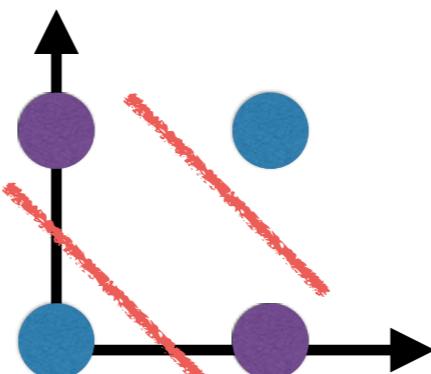
Possible

NOR



Possible

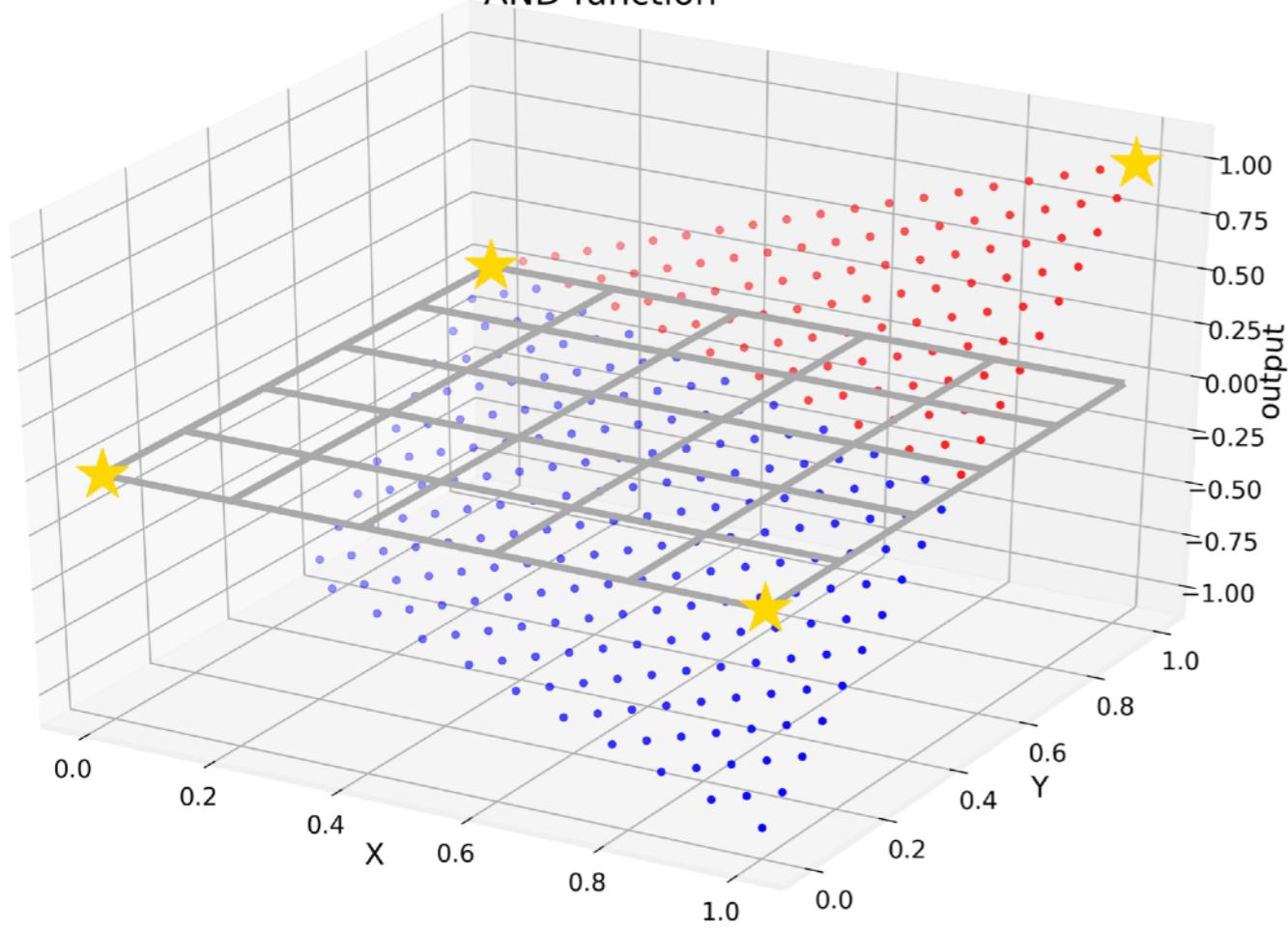
XOR



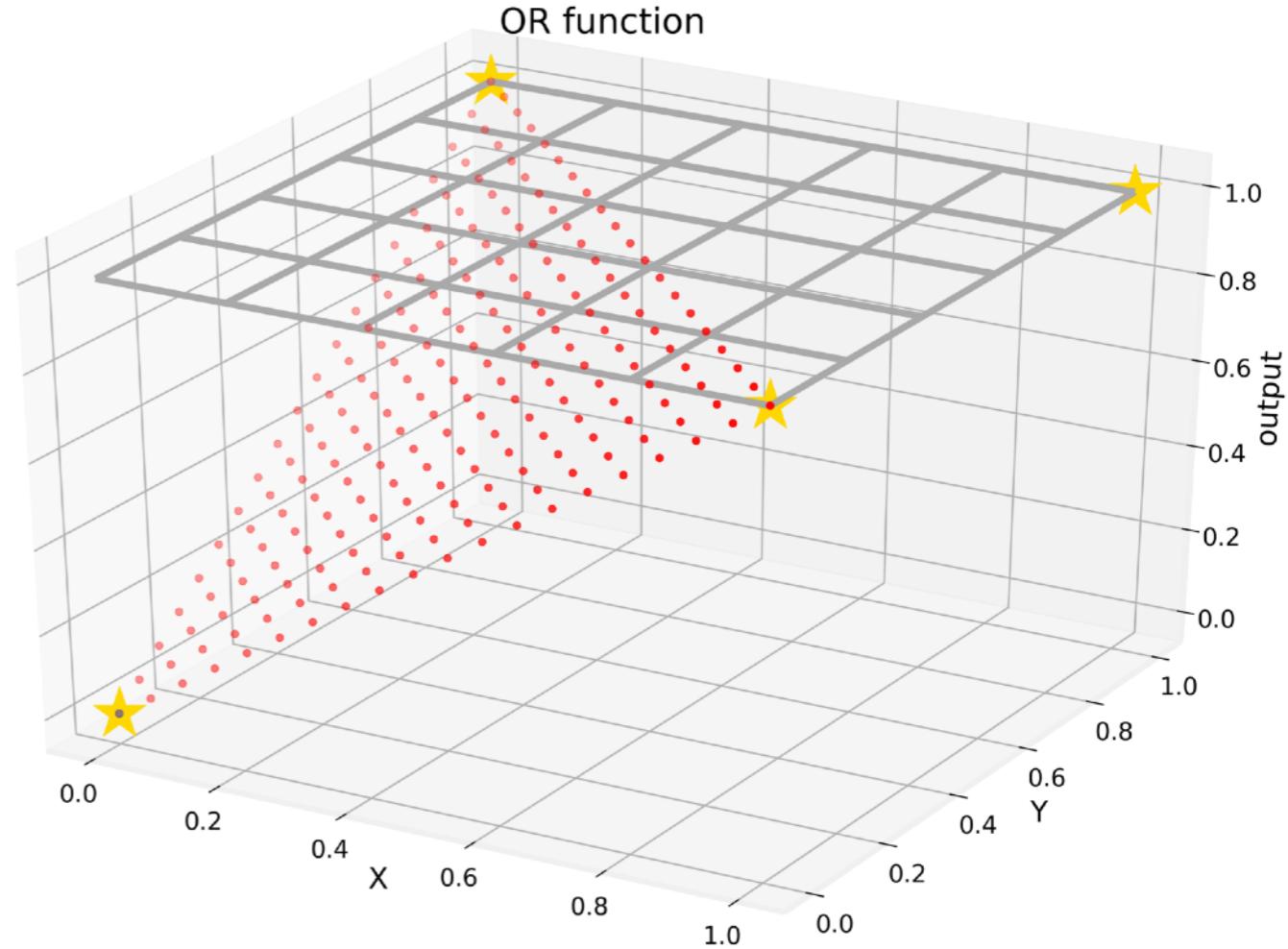
Impossible

Linear Boundaries

AND function



OR function

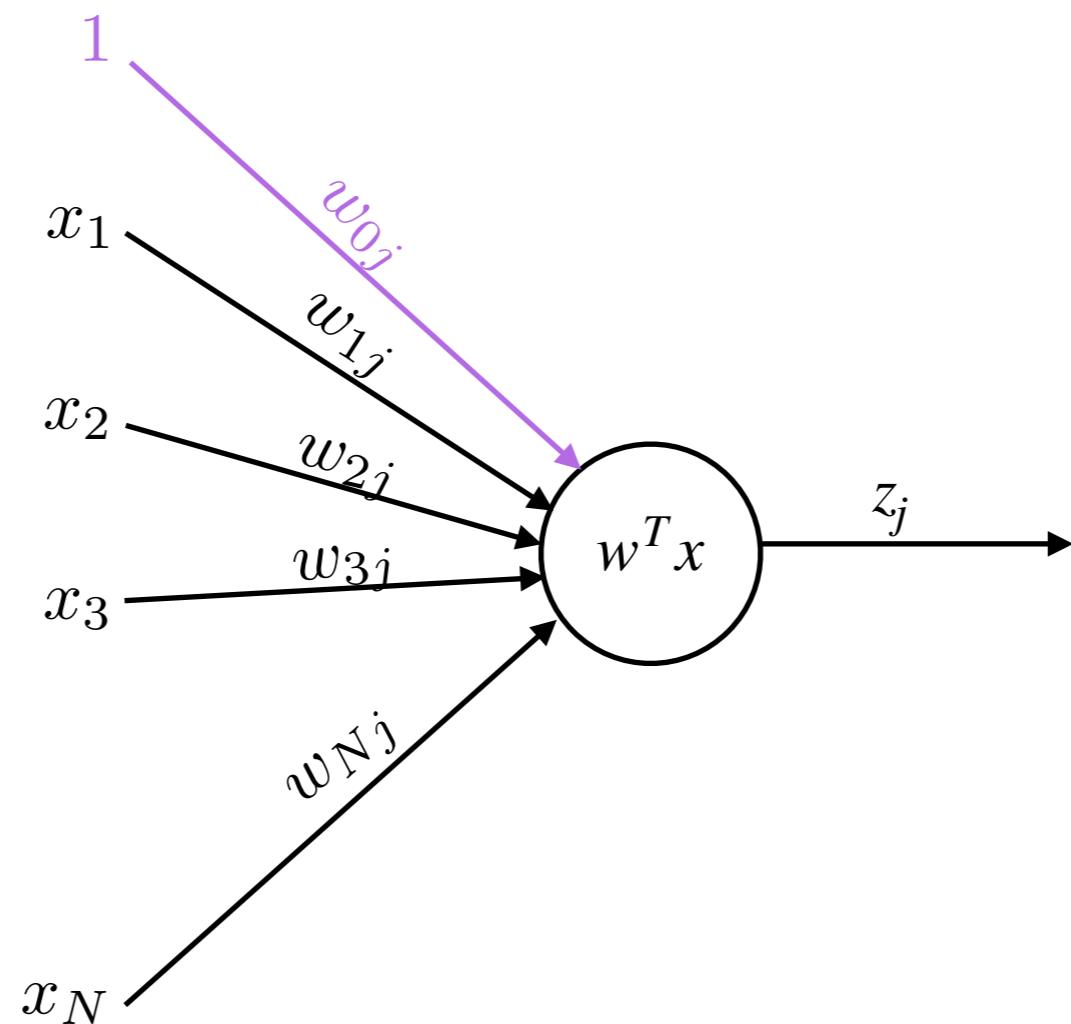




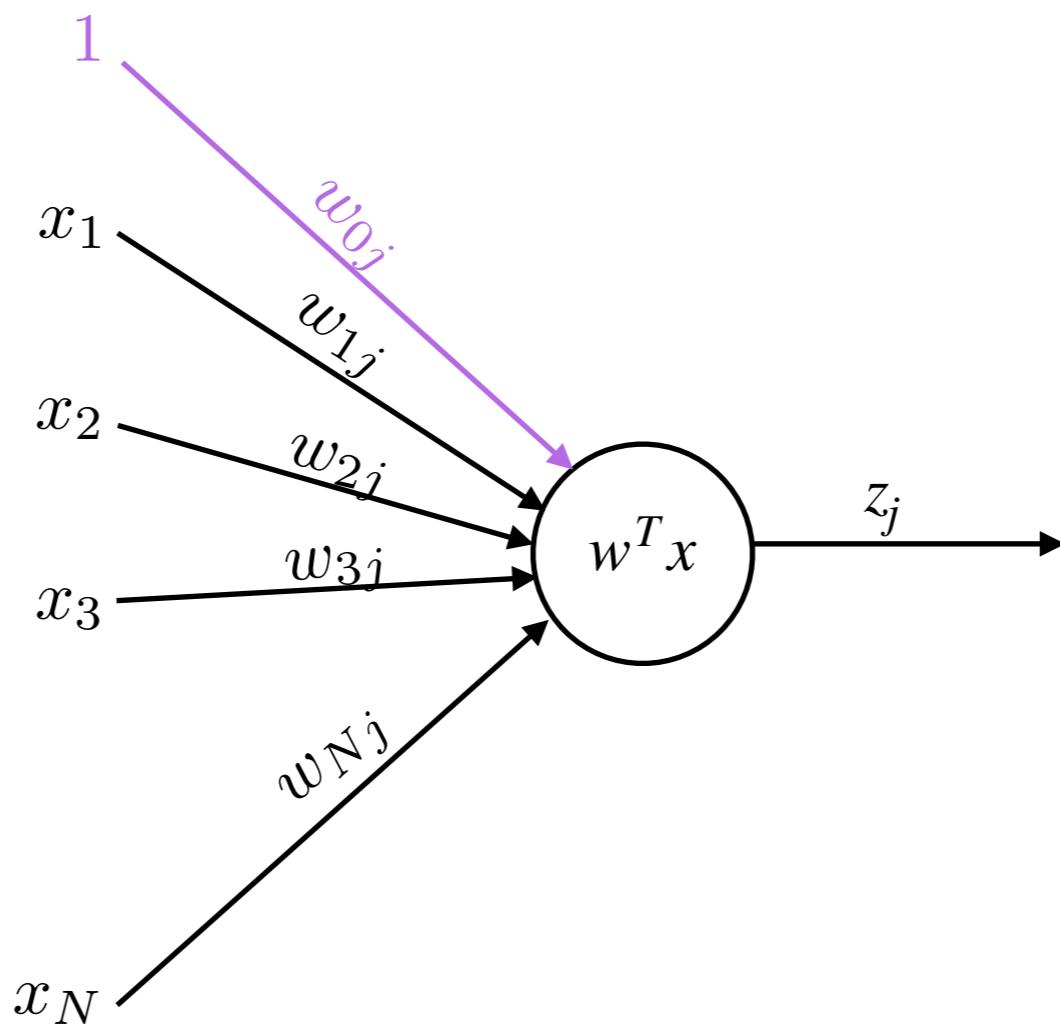
Code - Perceptron / Forward
Propagation

<https://github.com/DataForScience/DeepLearning>

Perceptron



Perceptron

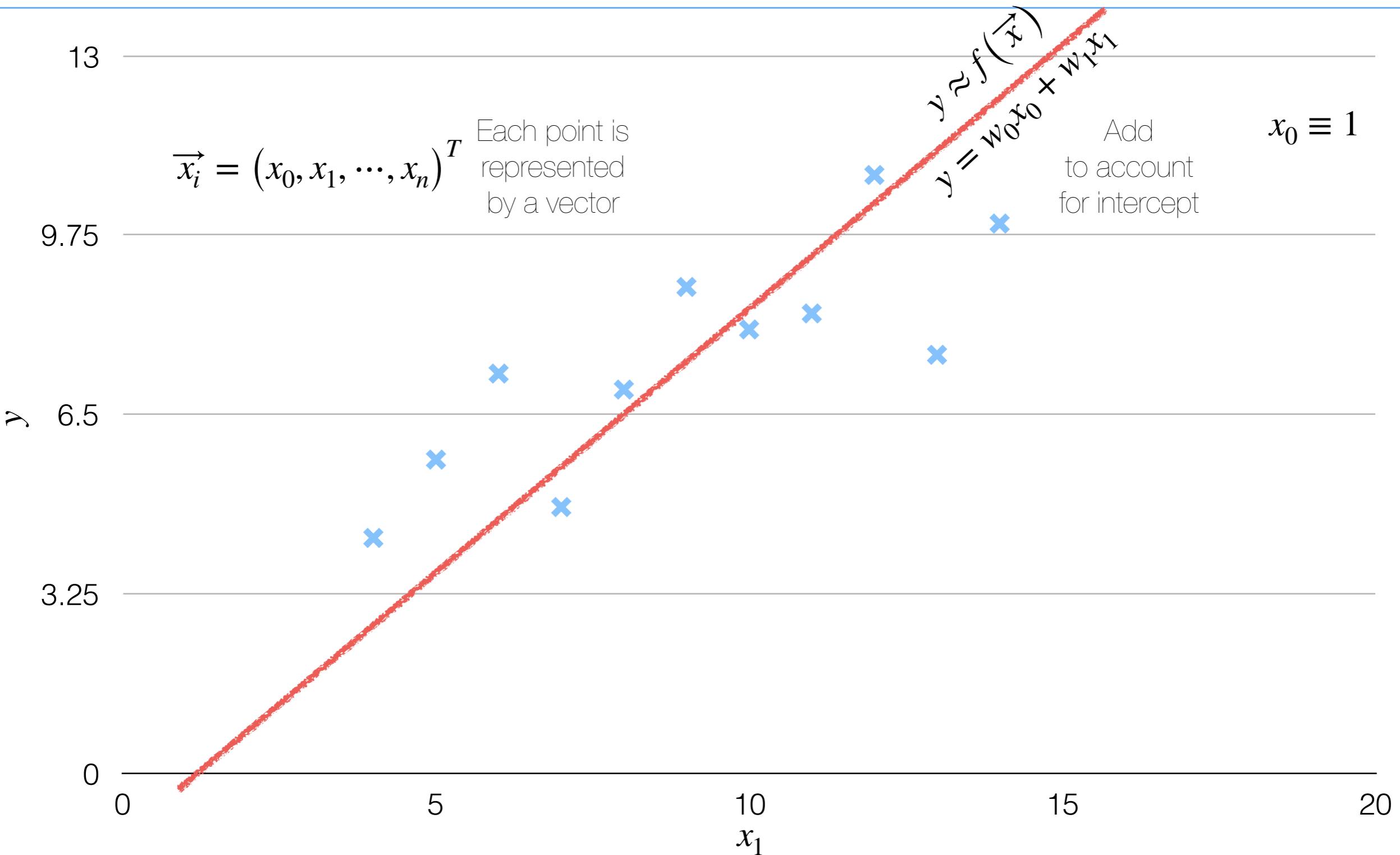


- This is just a graphical representation of:

$$z = w^T x$$

which is just linear regression!

Linear Regression



Optimization Problem

<https://github.com/DataForScience/DeepLearning>

- (Machine) Learning can be thought of as an optimization problem.

- Optimization Problems have 3 distinct pieces:

- The constraints
- The function to optimize
- The optimization algorithm.



Optimization Problem

<https://github.com/DataForScience/DeepLearning>

- (Machine) Learning can be thought of as an optimization problem.

- Optimization Problems have 3 distinct pieces:

- The constraints

Problem Representation

- The function to optimize

Prediction Error

- The optimization algorithm

Gradient Descent



Linear Regression

- We are assuming that our functional dependence is of the form:

$$f(\vec{x}) = w_0 + w_1 x_1 + \cdots + w_n x_n \equiv X \vec{w}$$

- In other words, at each step, our **hypothesis** is:

$$h_w(X) = X \vec{w} \equiv \hat{y}$$

and it imposes a **Constraint** on the solutions that can be found.

- We quantify how far our hypothesis is from the correct value using an **Error Function**:

$$J_w(X, \vec{y}) = \frac{1}{2m} \sum_i \left[h_w(x^{(i)}) - y^{(i)} \right]^2$$

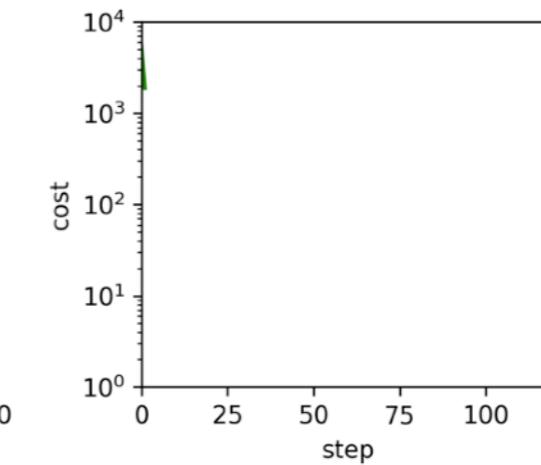
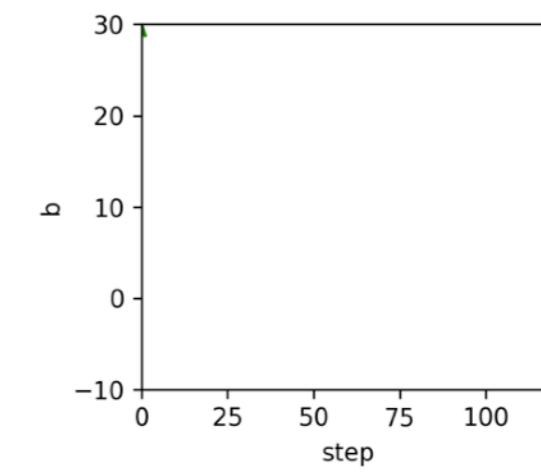
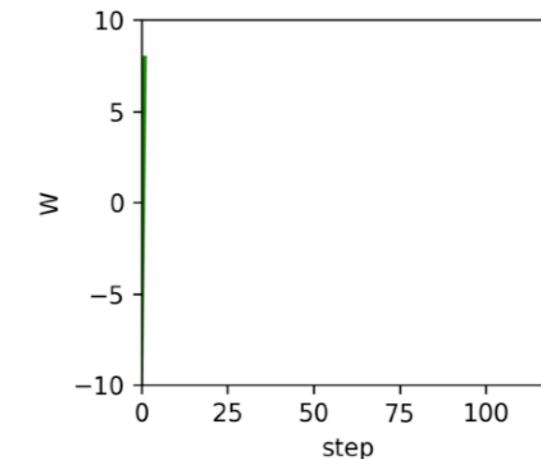
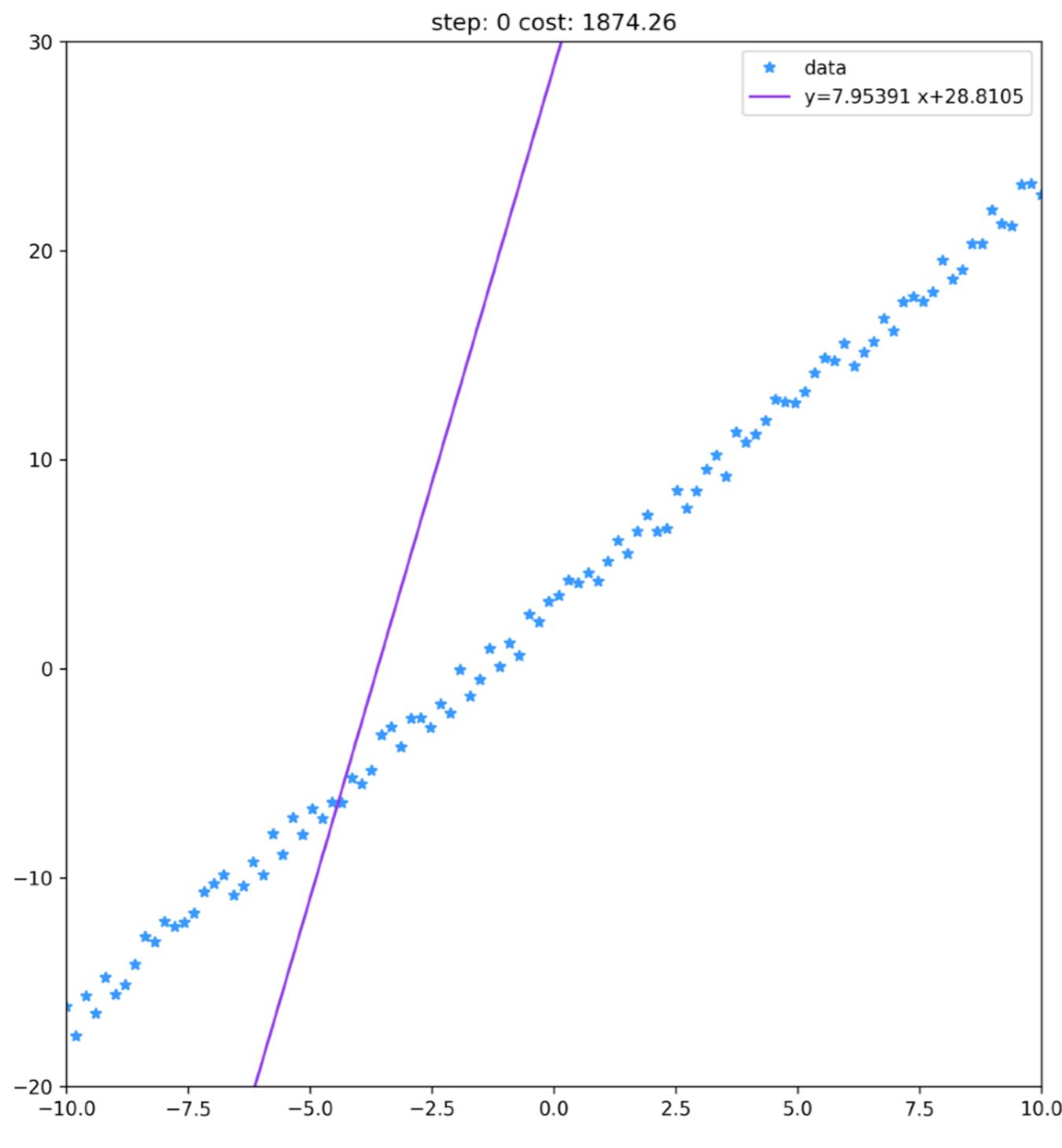
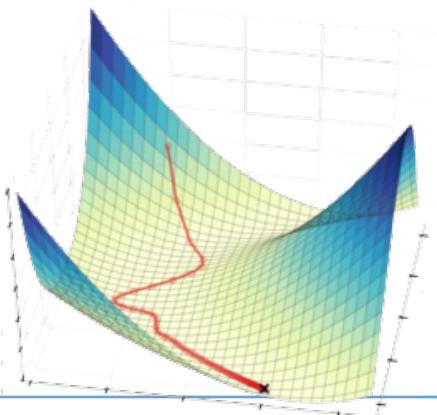
or, vectorially:

$$J_w(X, \vec{y}) = \frac{1}{2m} [X \vec{w} - \vec{y}]^2$$

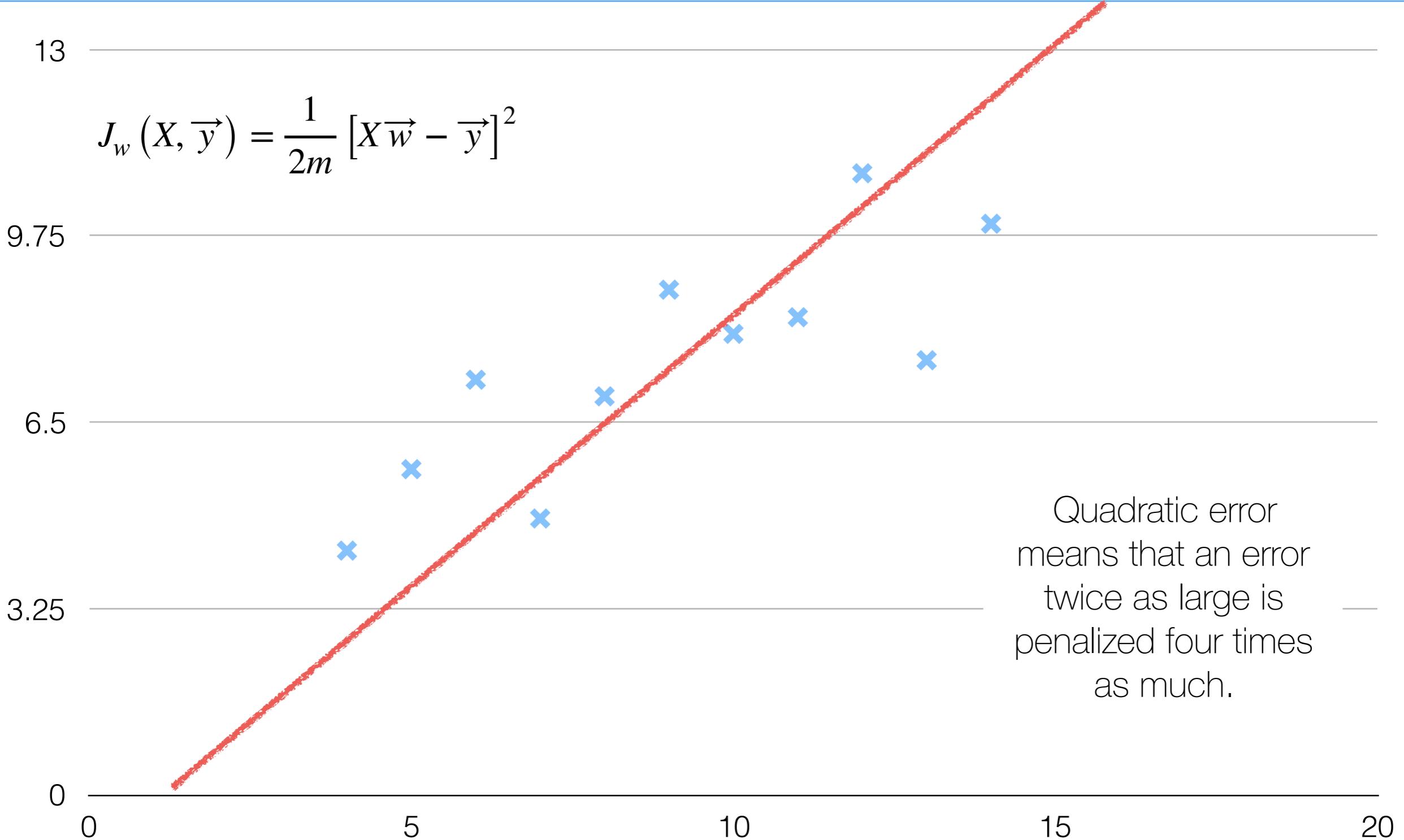


	Feature 1	Feature 2	Feature 3	...	Feature M	value
Sample 1						
Sample 2						
Sample 3						
Sample 4						
Sample 5						
Sample 6						
.						
X						
y						
Sample N						

Linear Regression

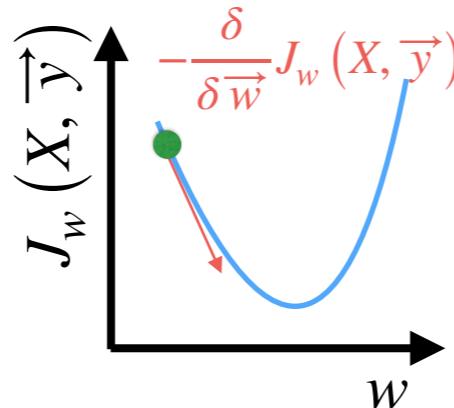


Geometric Interpretation



Gradient Descent

- **Goal:** Find the minimum of $J_w(X, \vec{y})$ by varying the components of \vec{w}
- **Intuition:** Follow the slope of the error function until convergence



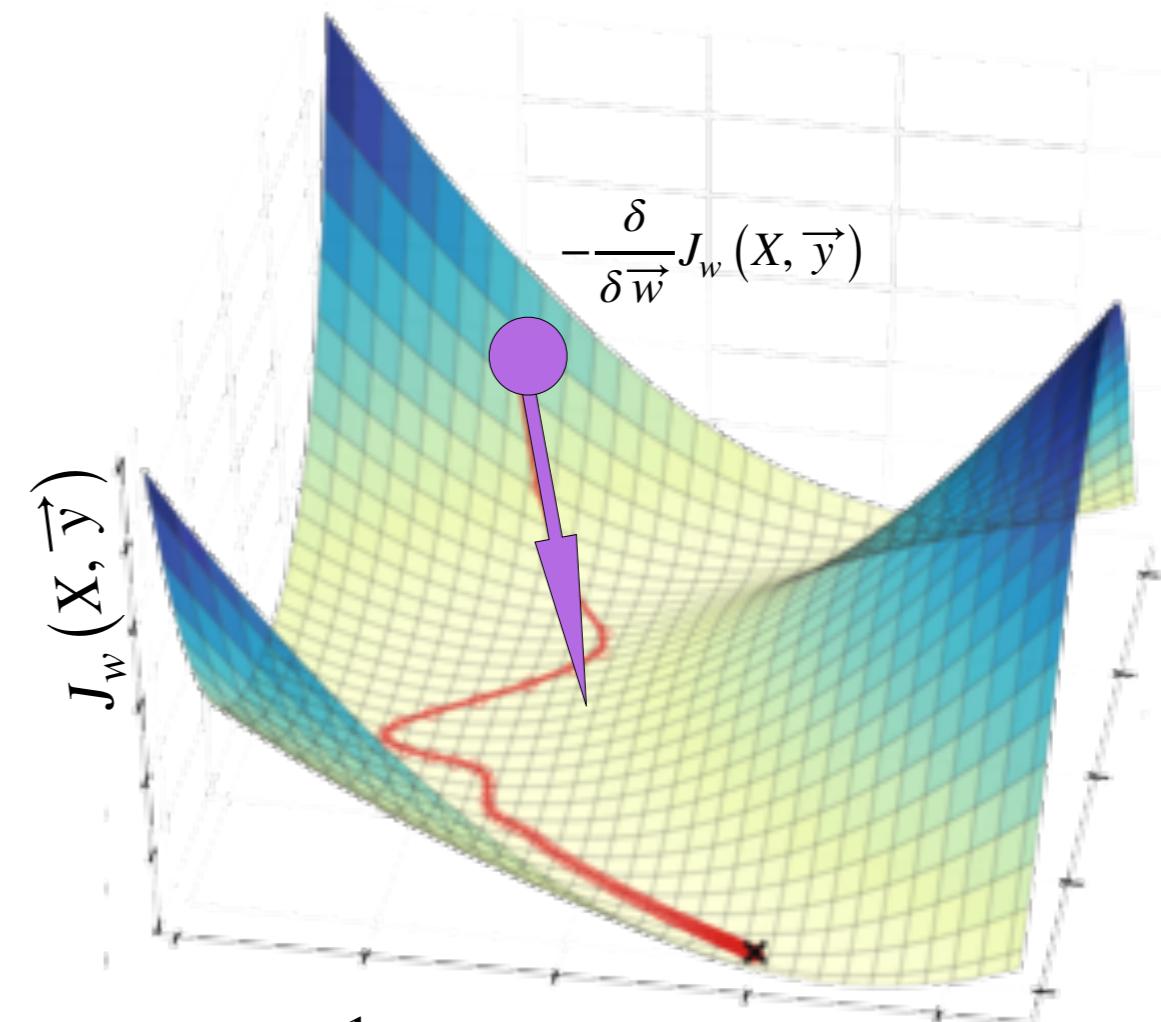
- **Algorithm:**

- Guess $\vec{w}^{(0)}$ (initial values of the parameters)

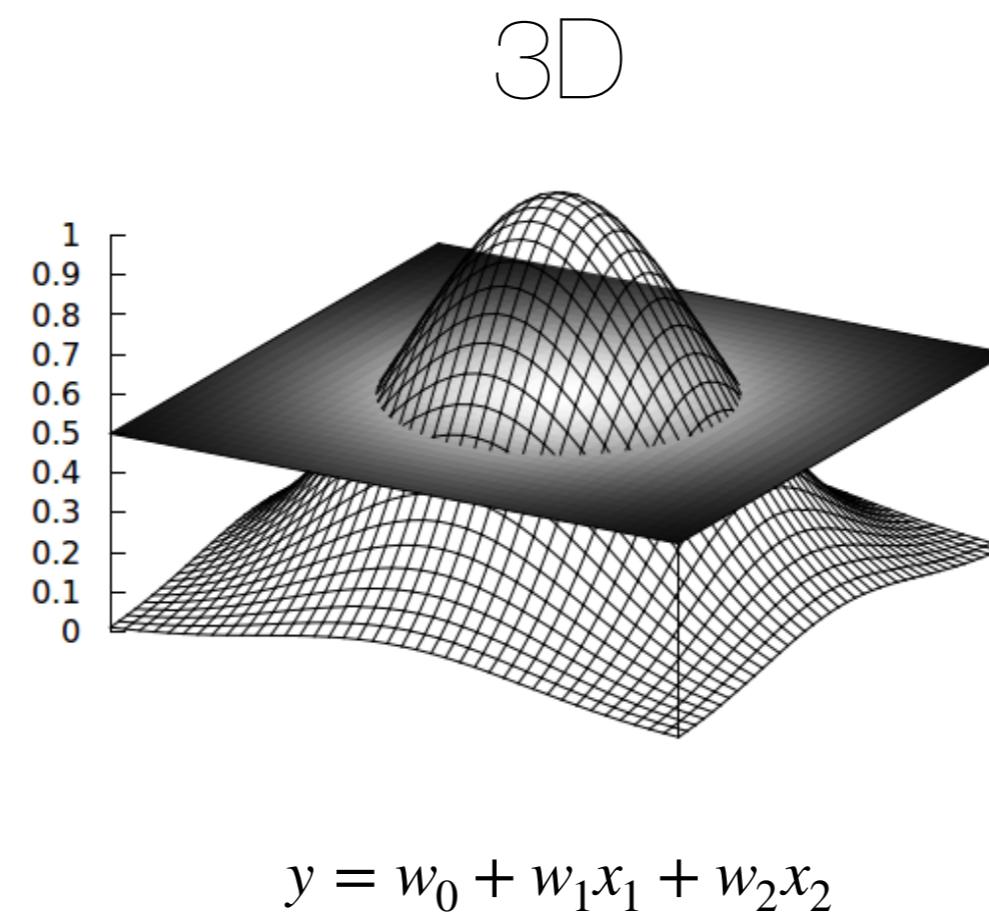
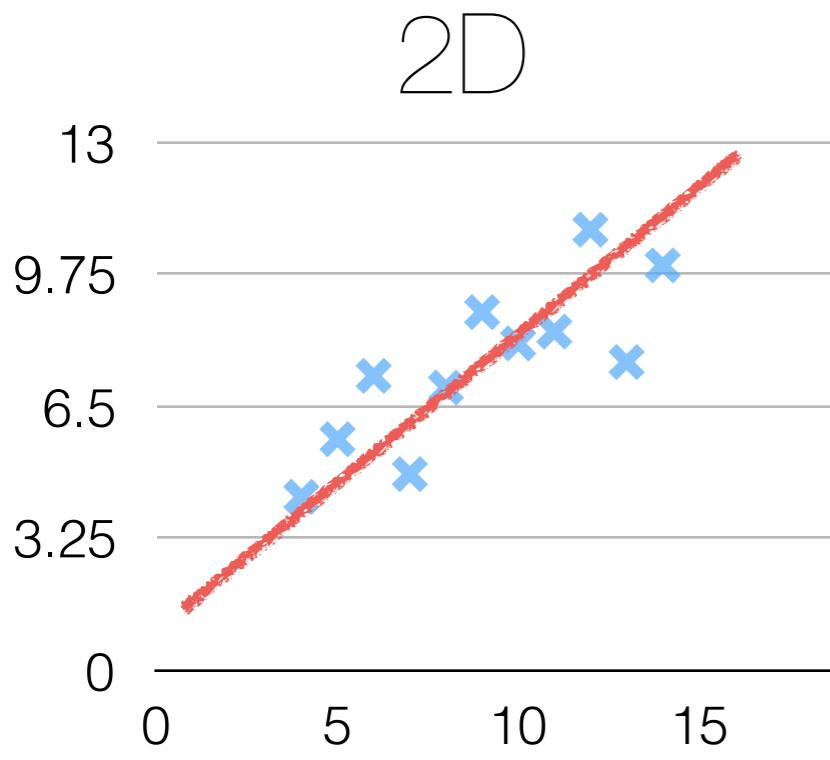
step size

- Update until "convergence":

$$w_j = w_j - \alpha \frac{\delta}{\delta w_j} J_w(X, \vec{y}) \quad \frac{\delta}{\delta w_j} J_w(X, \vec{y}) = \frac{1}{m} X^T \cdot (h_w(X) - \vec{y})$$



Geometric Interpretation



$$y = \vec{X} \vec{w}$$

Finds the hyperplane that splits the points in two such that the errors on each side balance out

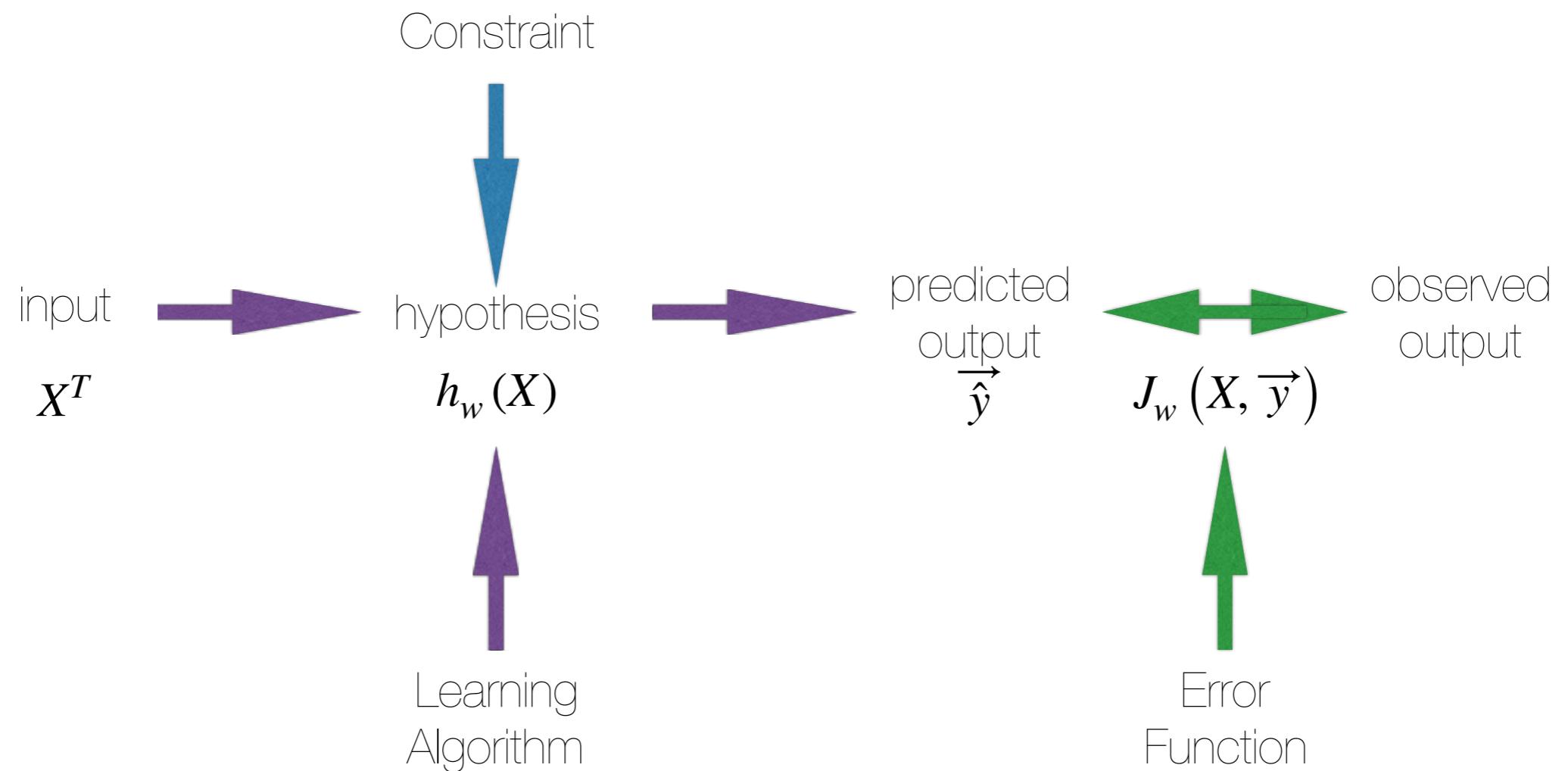
Add to account for intercept

$$x_0 \equiv 1$$

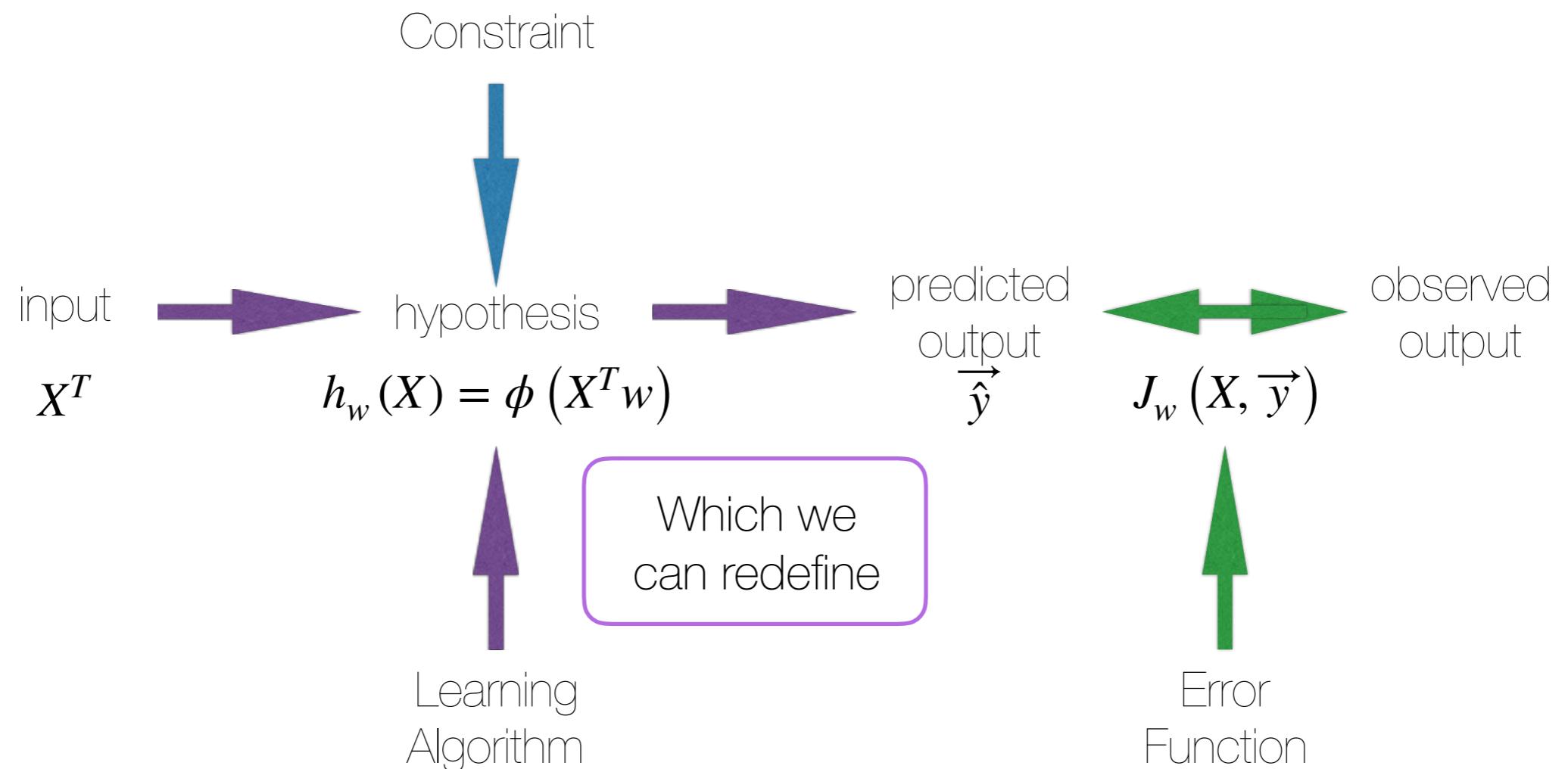


Code - Linear Regression
<https://github.com/DataForScience/DeepLearning>

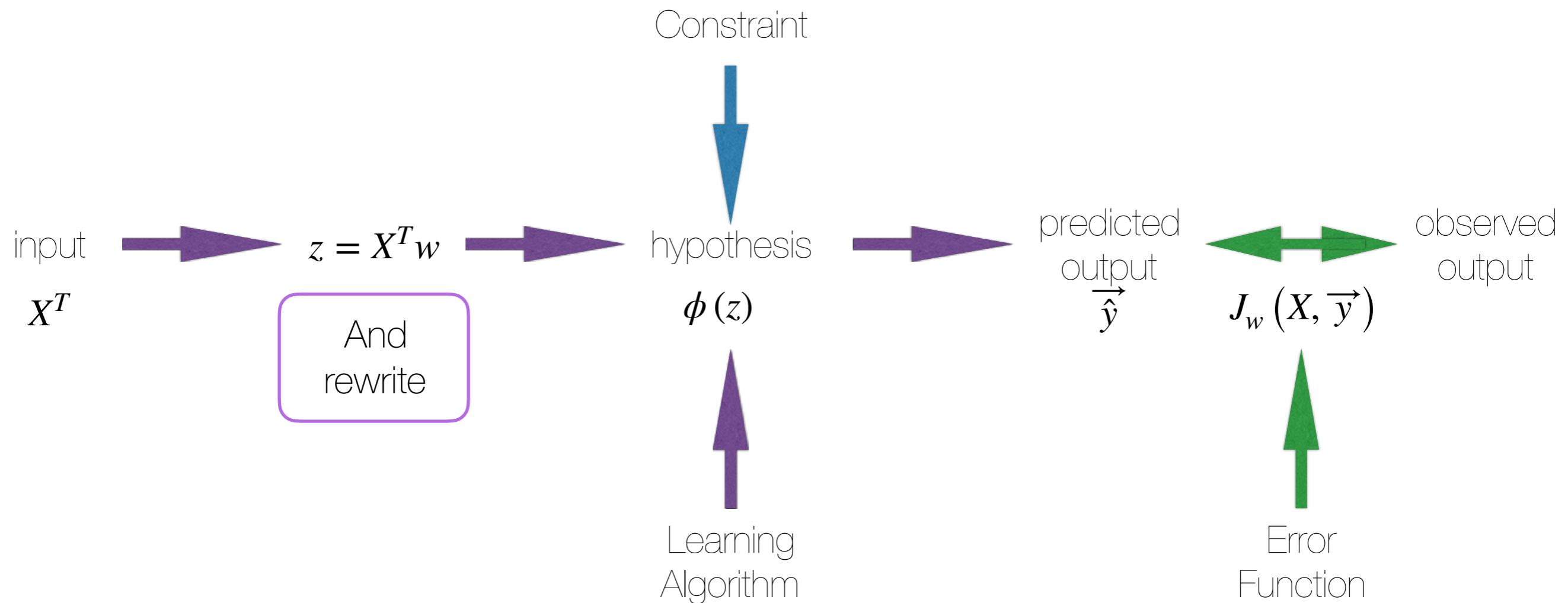
Learning Procedure



Learning Procedure



Learning Procedure



Logistic Regression (Classification)

- Not actually regression, but rather Classification
- Predict the probability of instance belonging to the given class:

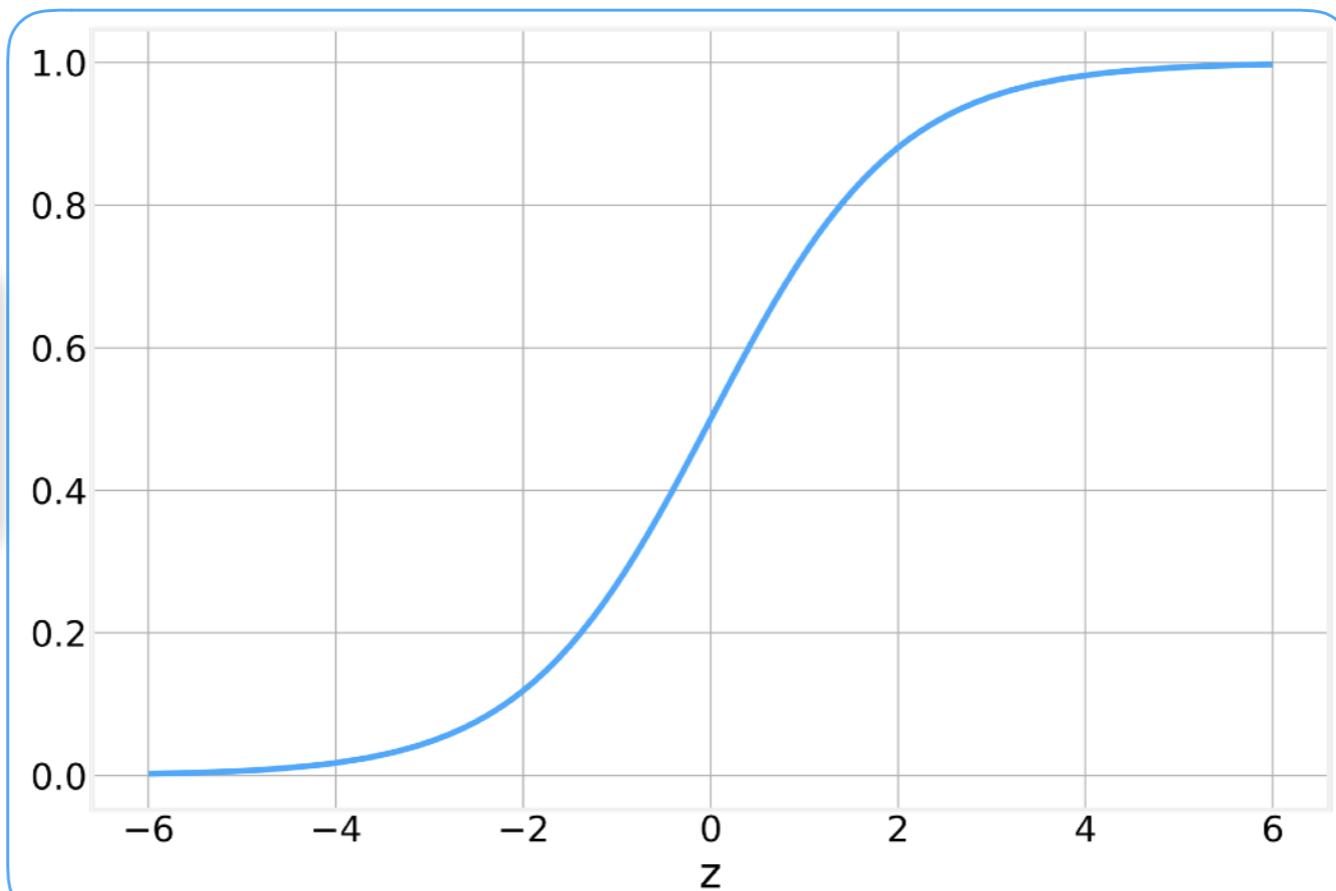
$$h_w(X) \in [0,1]$$

- Use **sigmoid/logistic** function to map weighted inputs to $[0,1]$

$$h_w(X) = \phi(X\vec{w})$$

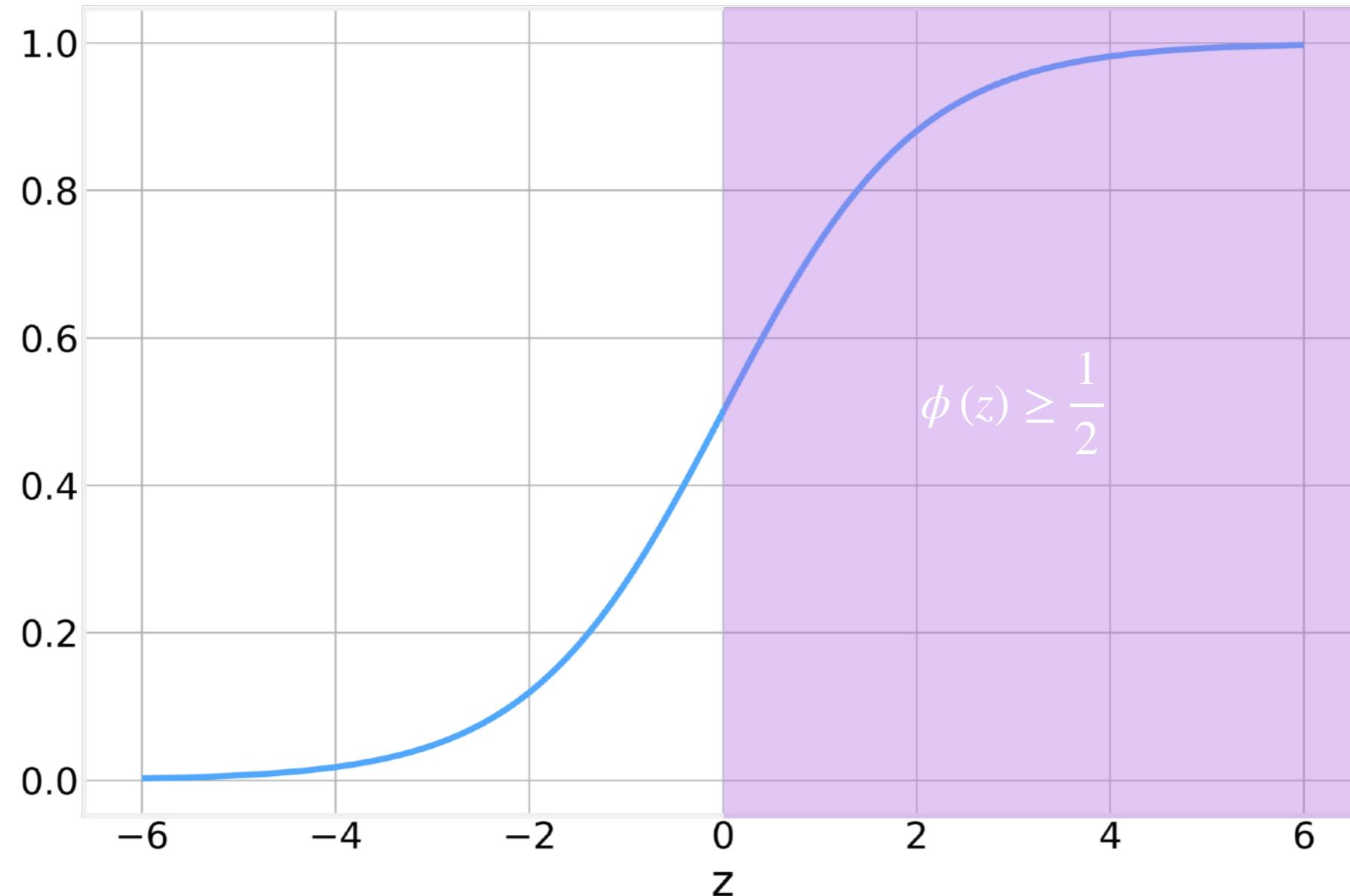
1 - part of the class
0 - otherwise

z encapsulates all the parameters and input values



Geometric Interpretation

maximize the value of z for members of the class



Logistic Regression

- **Error Function** - Cross Entropy

$$J_w(X, \vec{y}) = -\frac{1}{m} \left[y^T \log(h_w(X)) + (1-y)^T \log(1-h_w(X)) \right]$$

measures the “distance” between two probability distributions

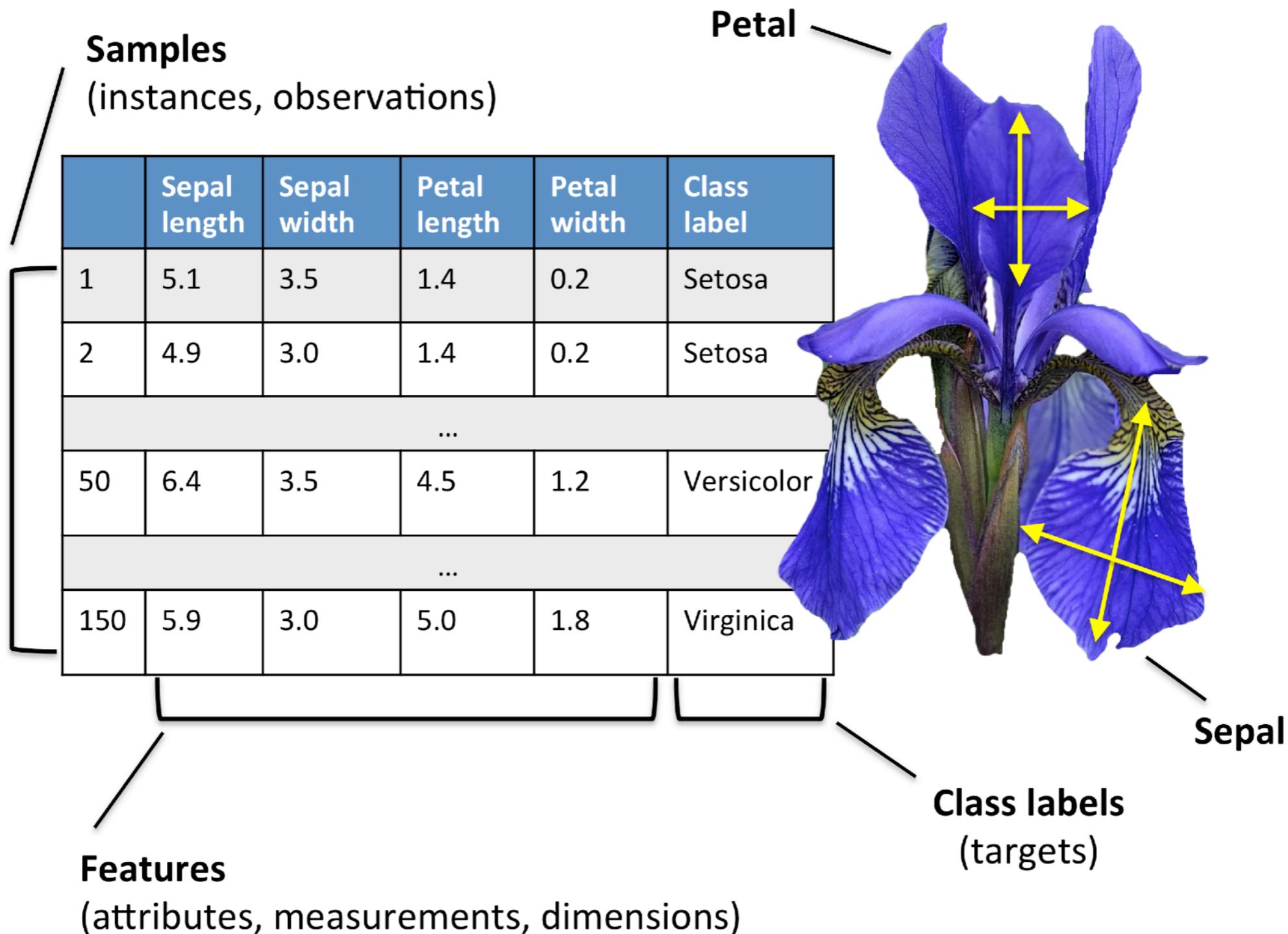
$$h_w(X) = \frac{1}{1 + e^{-X^T \vec{w}}}$$

- Effectively **treating the labels as probabilities** (an instance with label=1 has Probability 1 of belonging to the class).
- **Gradient** - same as Logistic Regression

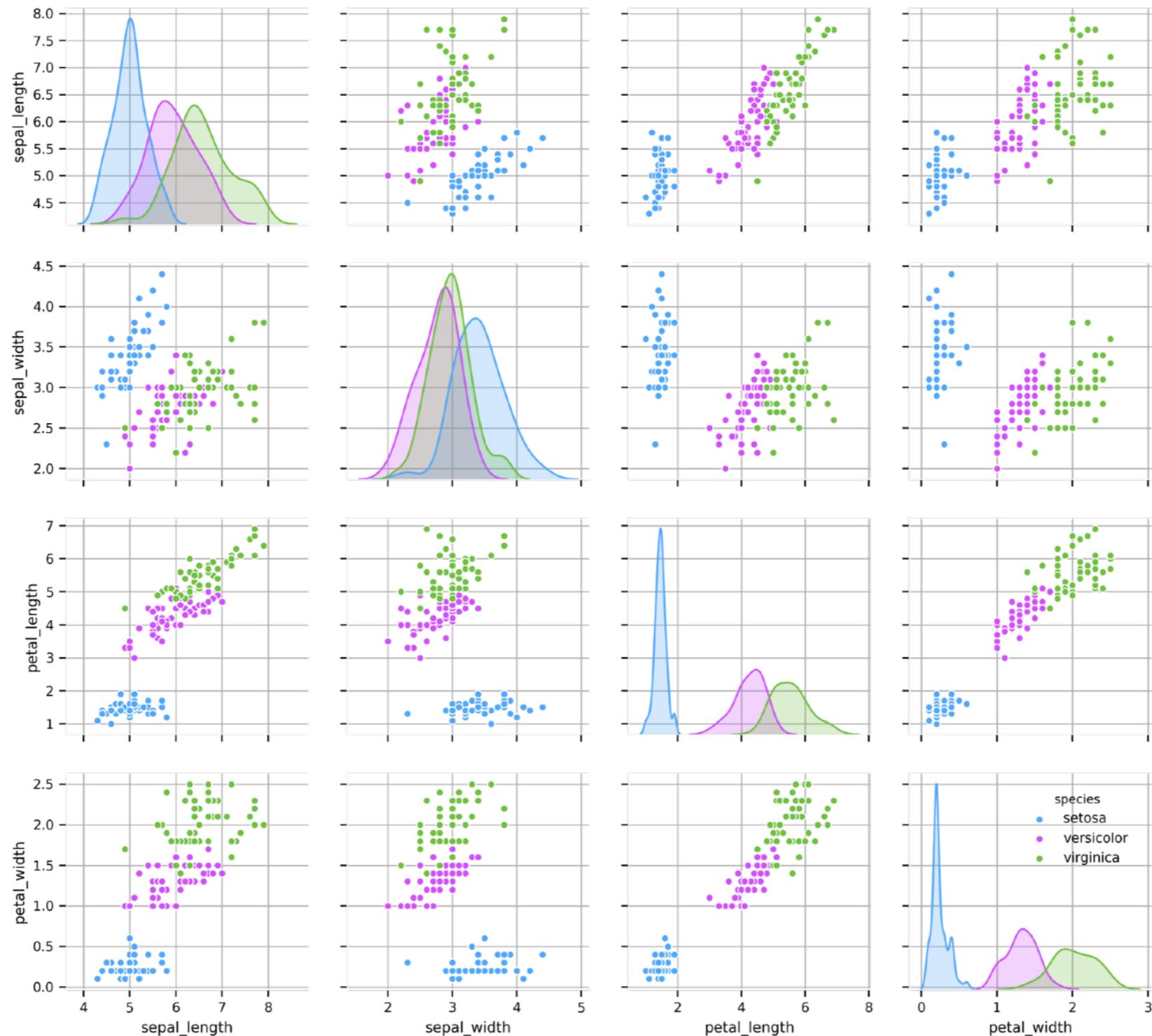
$$w_j = w_j - \alpha \frac{\delta}{\delta w_j} J_w(X, \vec{y})$$

$$\frac{\delta}{\delta w_j} J_w(X, \vec{y}) = \frac{1}{m} X^T \cdot (h_w(X) - \vec{y})$$

Iris dataset



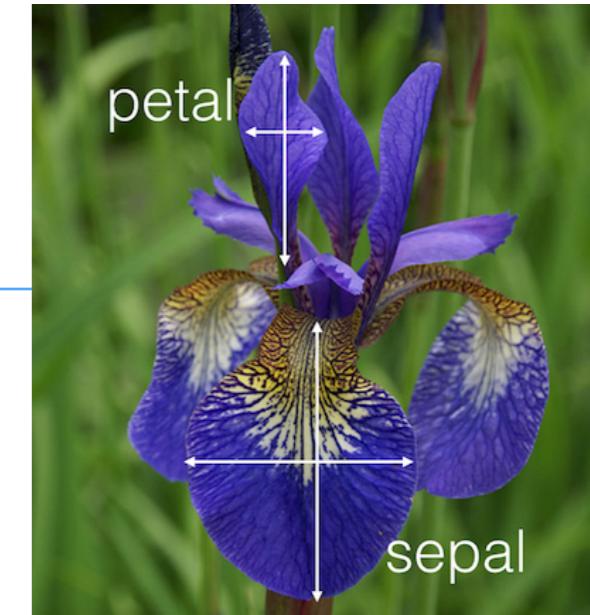
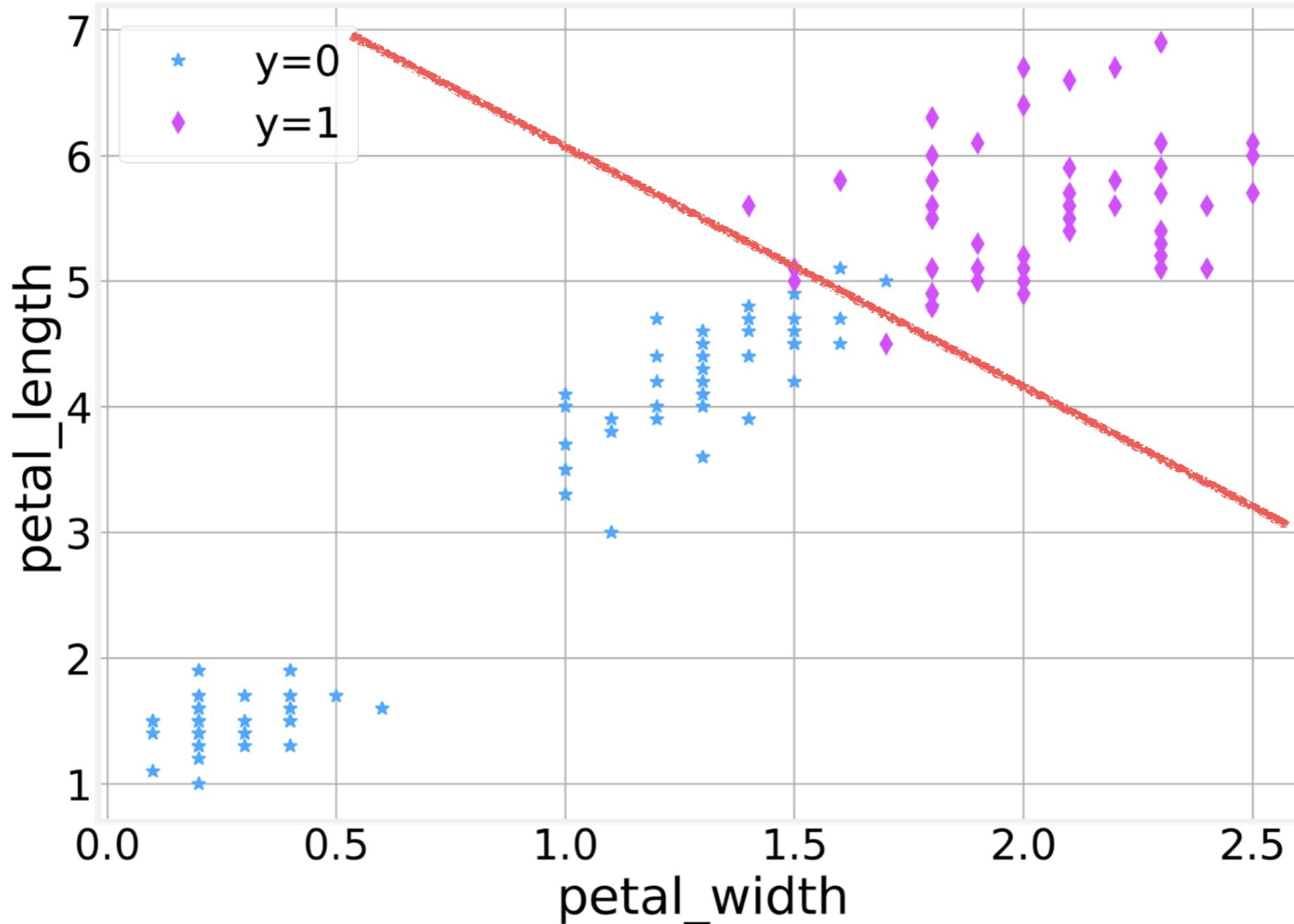
Iris dataset



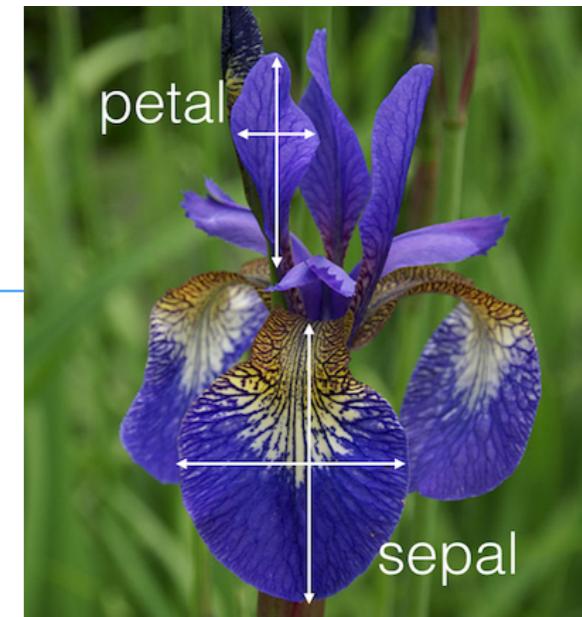
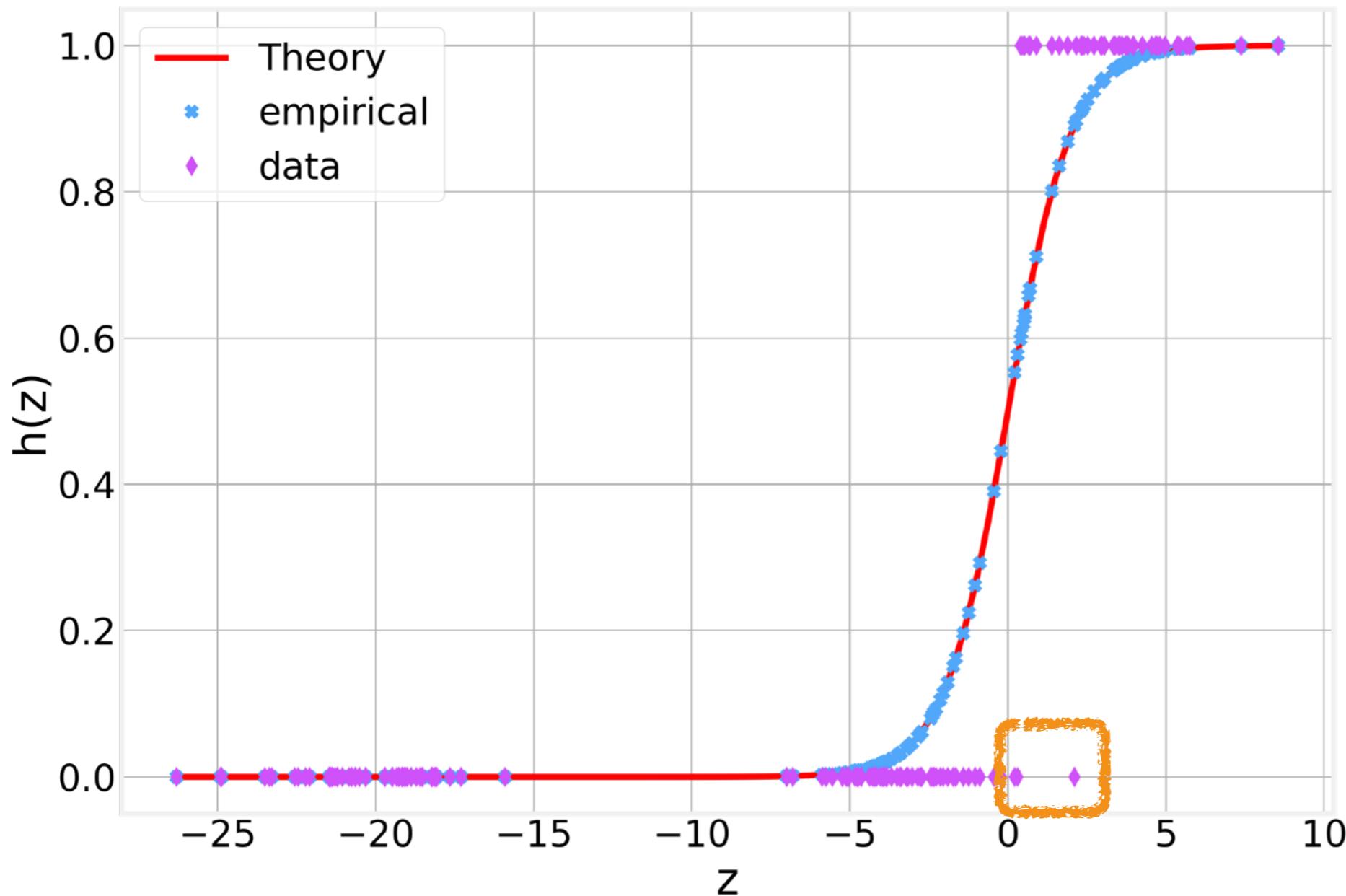


Code - Logistic Regression
<https://github.com/DataForScience/DeepLearning>

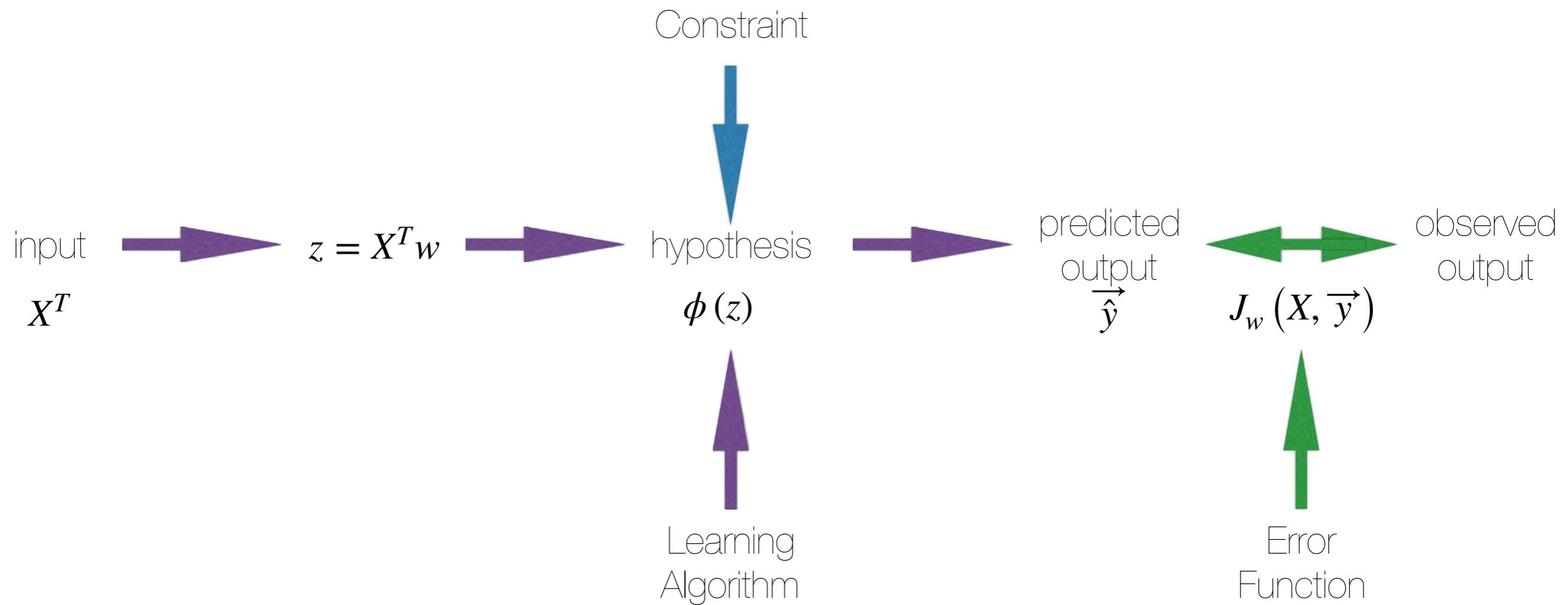
Logistic Regression



Logistic Regression



Learning Procedure



Comparison

- Linear Regression

$$z = X \vec{w}$$

$$h_w(X) = \phi(Z)$$

- Logistic Regression

$$z = X \vec{w}$$

$$h_w(X) = \phi(Z)$$

Map features to a continuous variable

Compare prediction with reality

$$\phi(Z) = Z$$

$$\phi(Z) = \frac{1}{1 + e^{-Z}}$$

Predict based on continuous variable



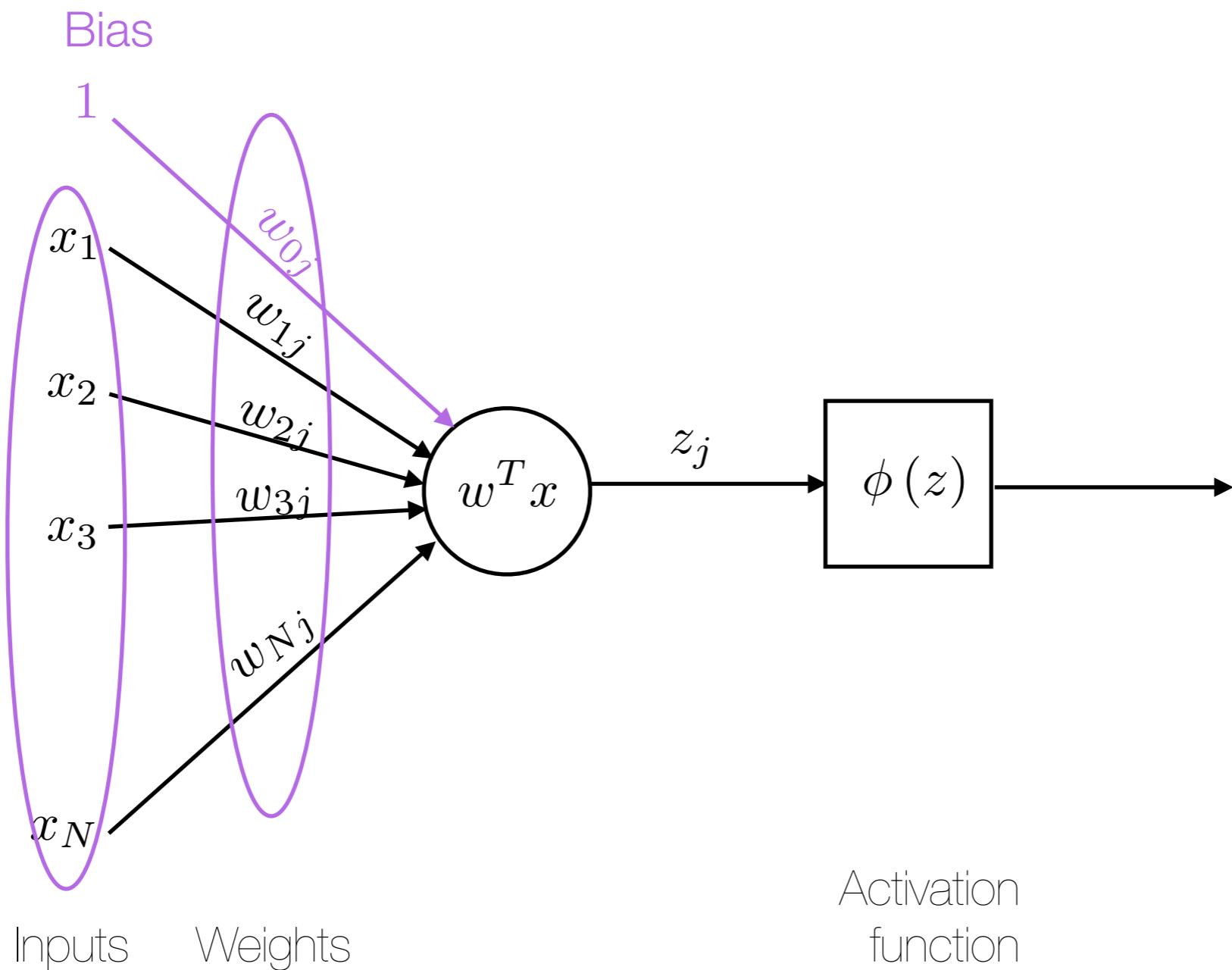
$$J_w(X, \vec{y}) = \frac{1}{2m} [h_w(X) - \vec{y}]^2$$

$$J_w(X, \vec{y}) = -\frac{1}{m} \left[y^T \log(h_w(X)) + (1 - y)^T \log(1 - h_w(X)) \right]$$

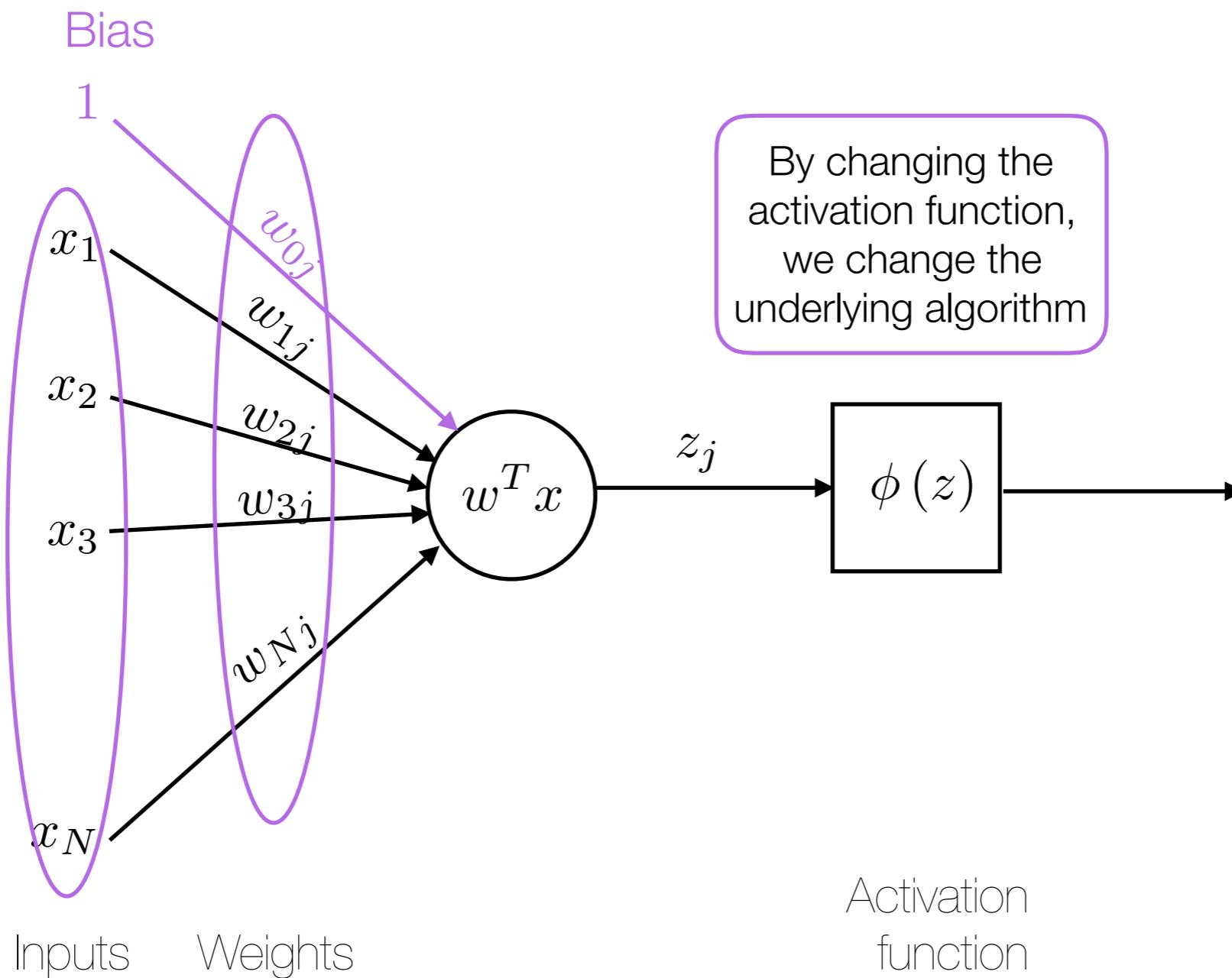
$$\frac{\delta}{\delta w_j} J_w(X, \vec{y}) = \frac{1}{m} X^T \cdot (h_w(X) - \vec{y})$$

$$\frac{\delta}{\delta w_j} J_w(X, \vec{y}) = \frac{1}{m} X^T \cdot (h_w(X) - \vec{y})$$

Learning Procedure



Generalized Perceptron

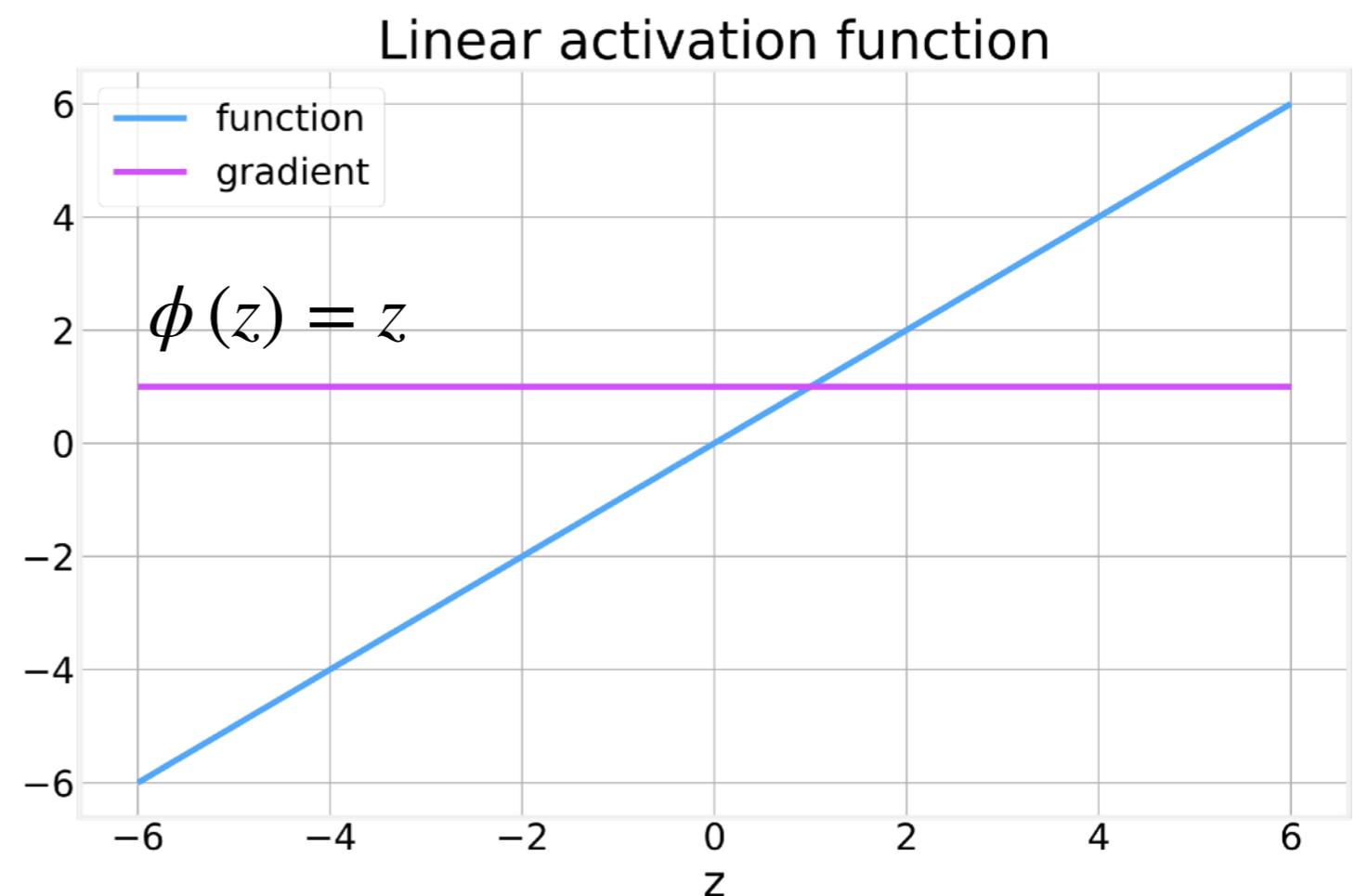


Activation Function

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data

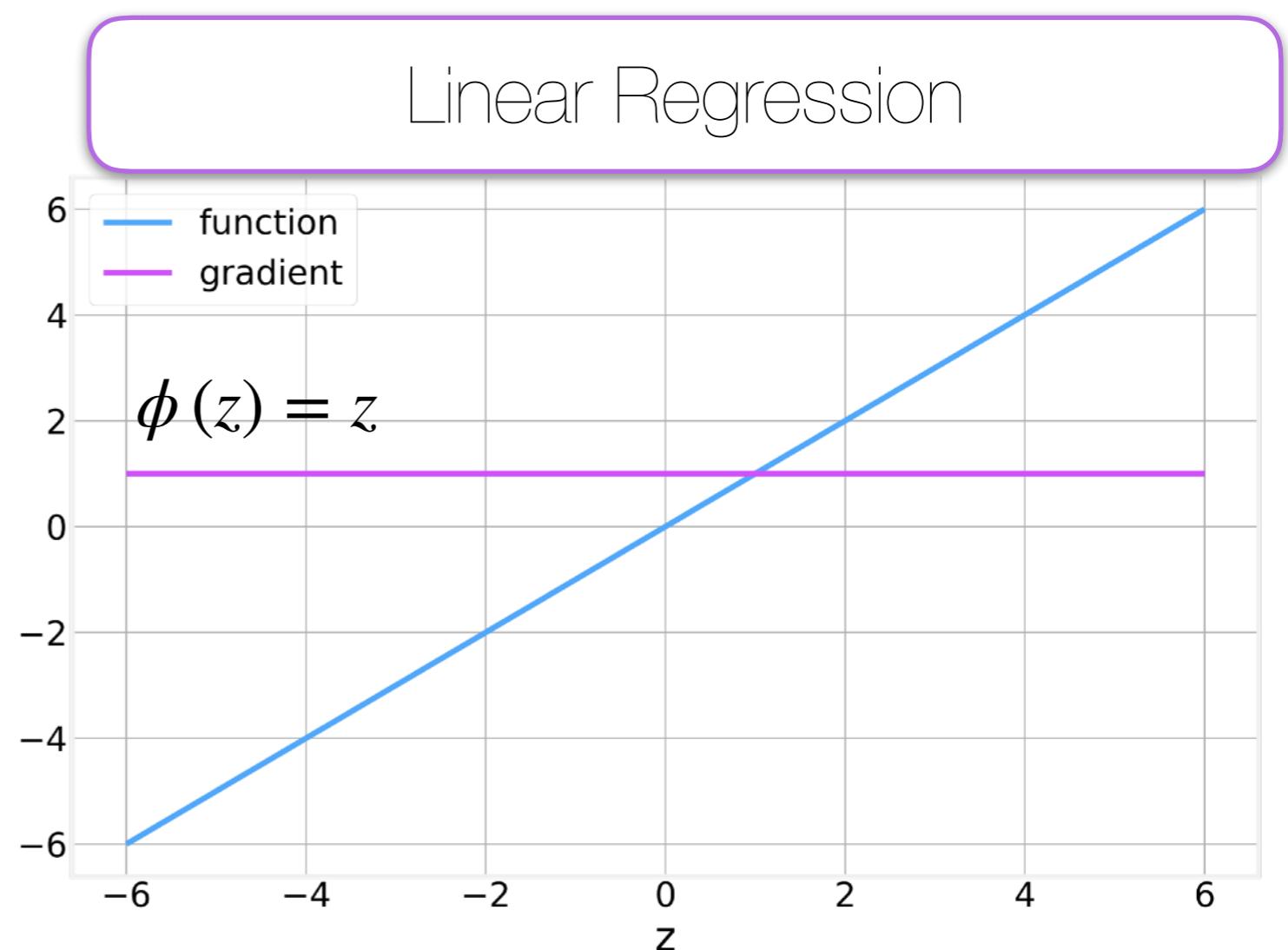
Activation Function - Linear

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- The **simplest**



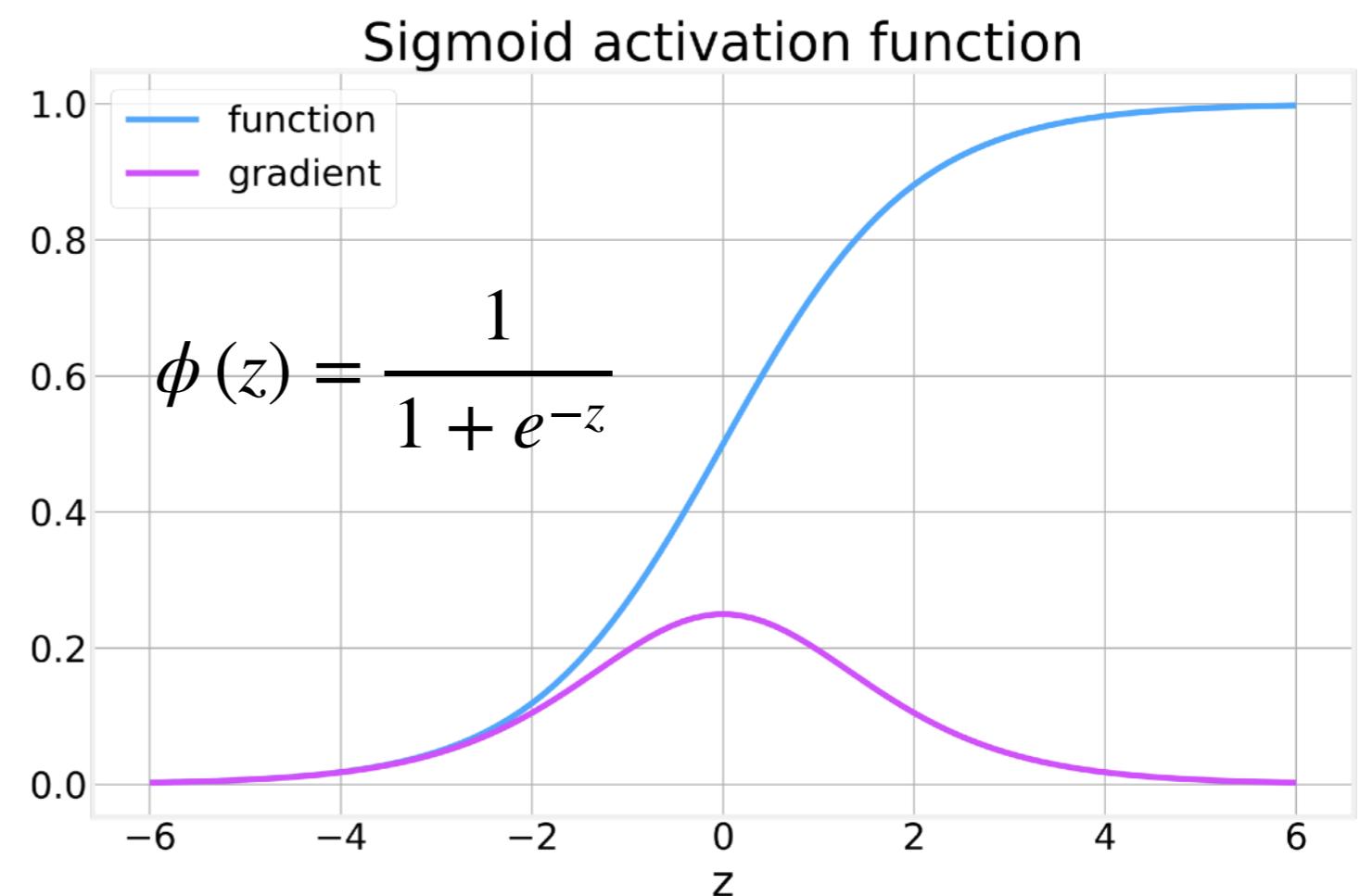
Activation Function - Linear

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- The **simplest**



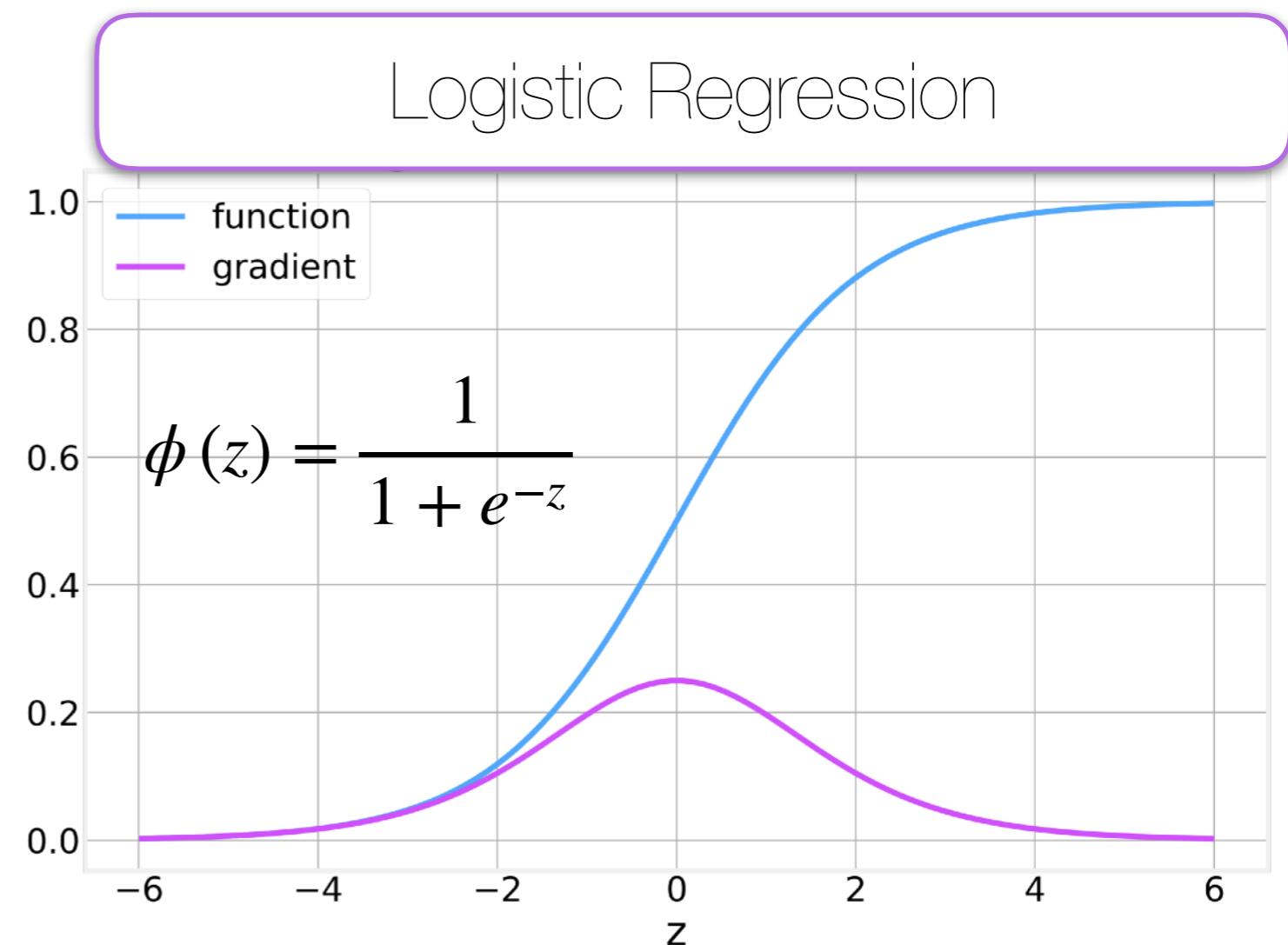
Activation Function - Sigmoid

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- Perhaps the **most common**



Activation Function - Sigmoid

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- Perhaps the **most common**



Forward Propagation

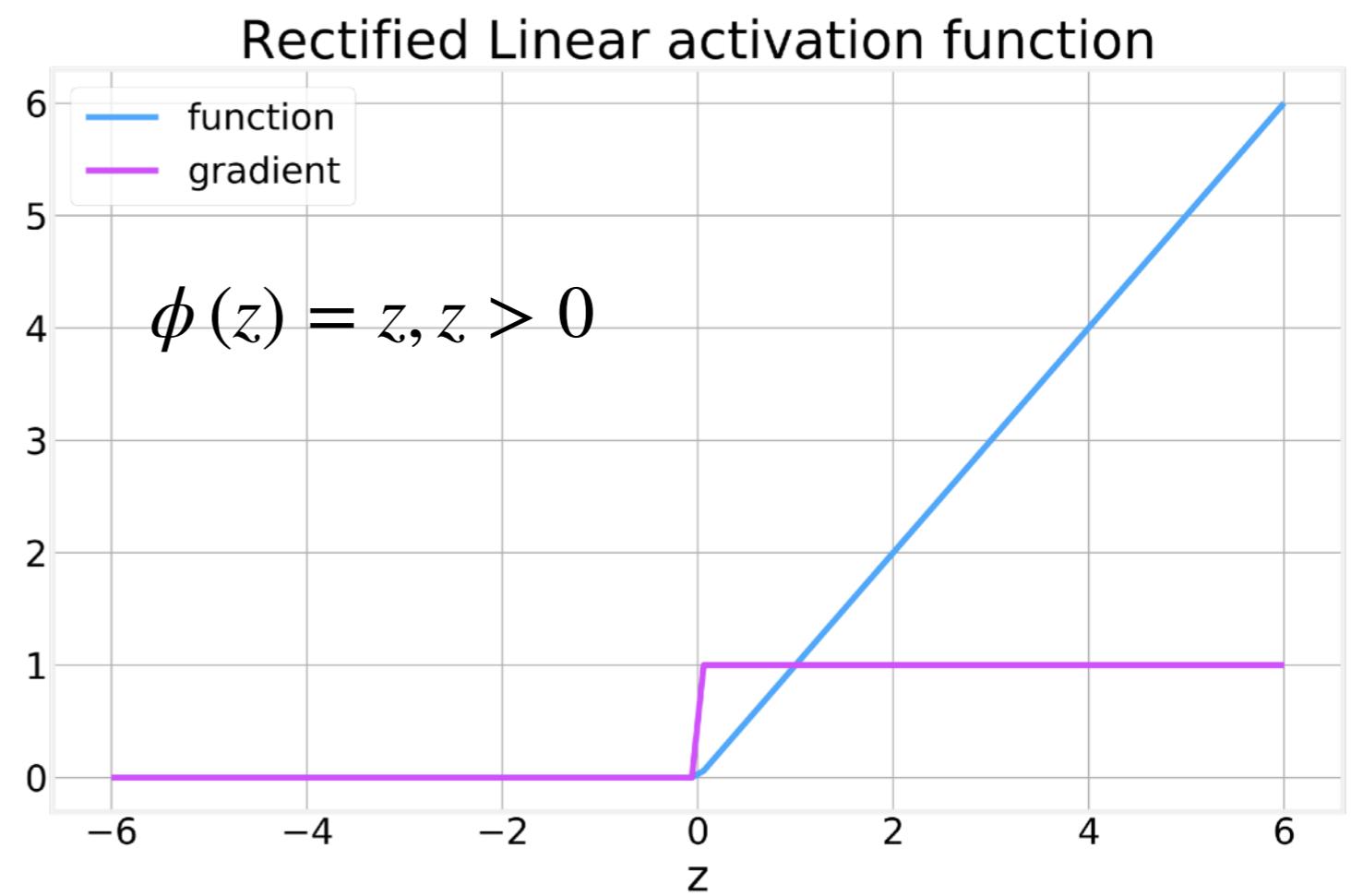
- The output of a perceptron is determined by a sequence of steps:
 - obtain the inputs
 - multiply the inputs by the respective weights
 - calculate output using the activation function



Code - Forward Propagation
<https://github.com/DataForScience/DeepLearning>

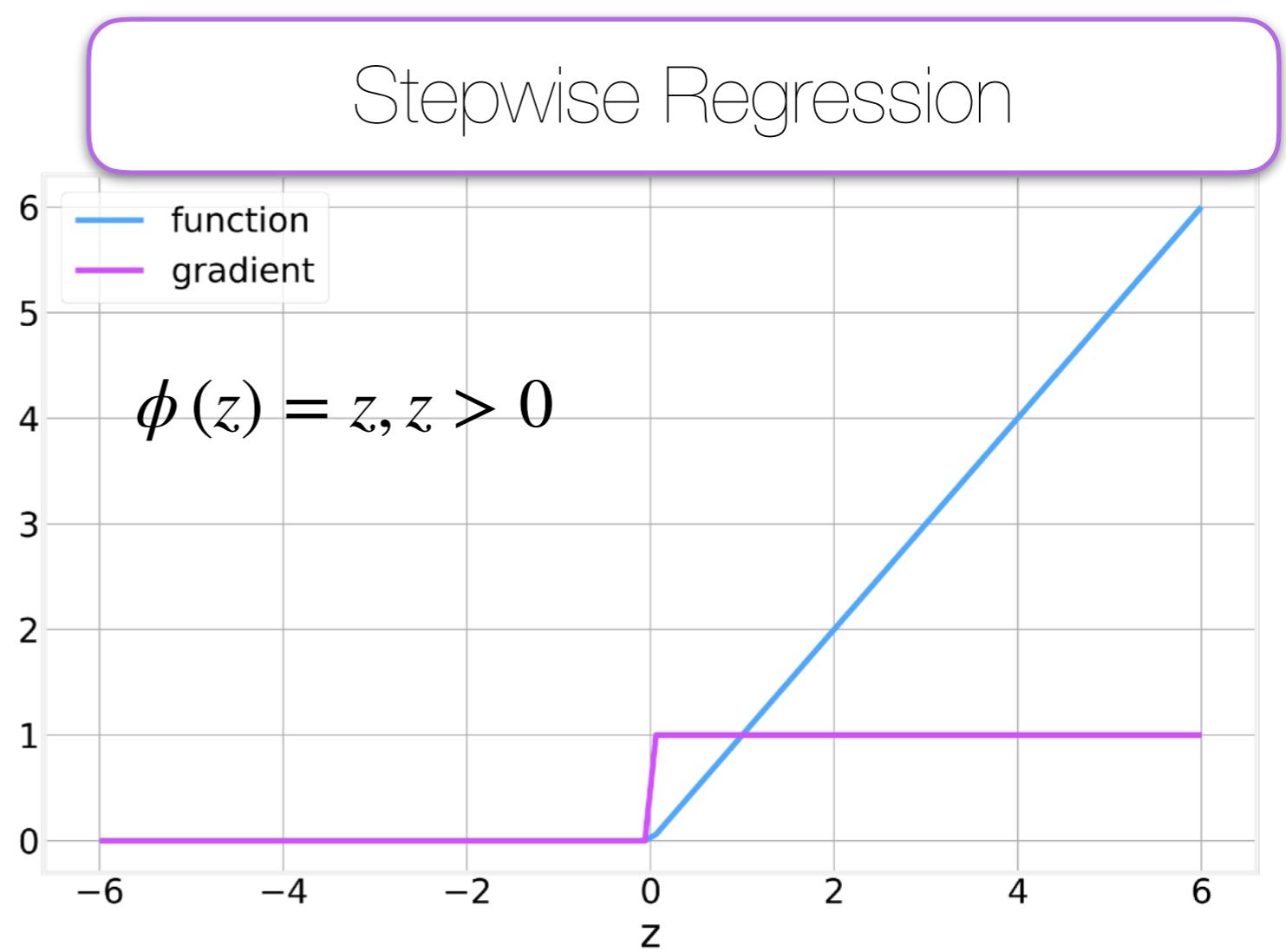
Activation Function - ReLu

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- Results in **faster learning** than with sigmoid



Activation Function - ReLu

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- Results in **faster learning** than with sigmoid



Stepwise Regression

https://en.wikipedia.org/wiki/Multivariate_adaptive_regression_spline

- Multivariate Adaptive Regression Spline (MARS) is the best known example

- Fit curves using a linear combination of:

$$\hat{f}(x) = \sum_i c_i B_i(x)$$

- The basis functions can be:

$$B_i(x)$$

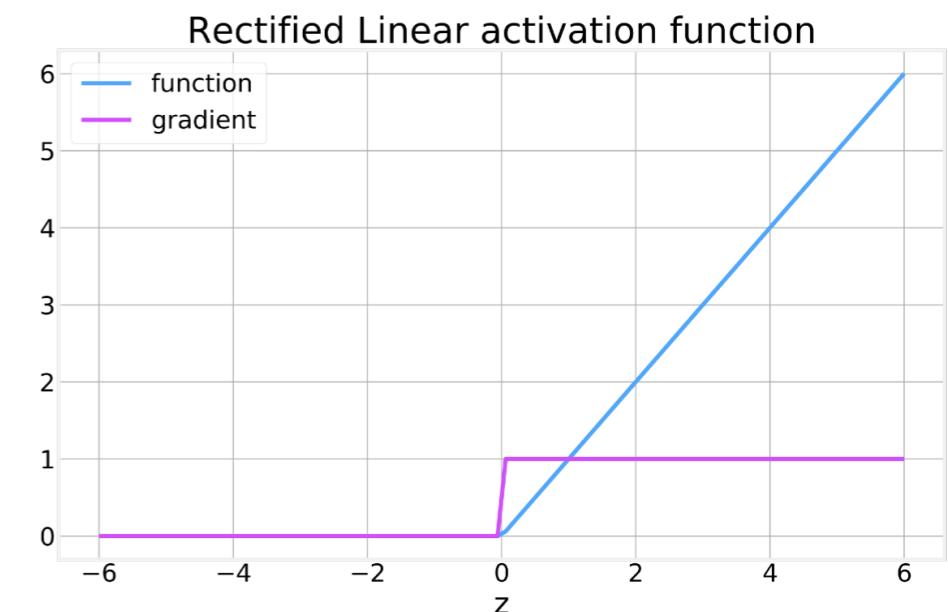
- Constant

- "Hinge" functions of the form:

and

$$\max(0, x - b) \quad \max(0, b - x)$$

- Products of hinges

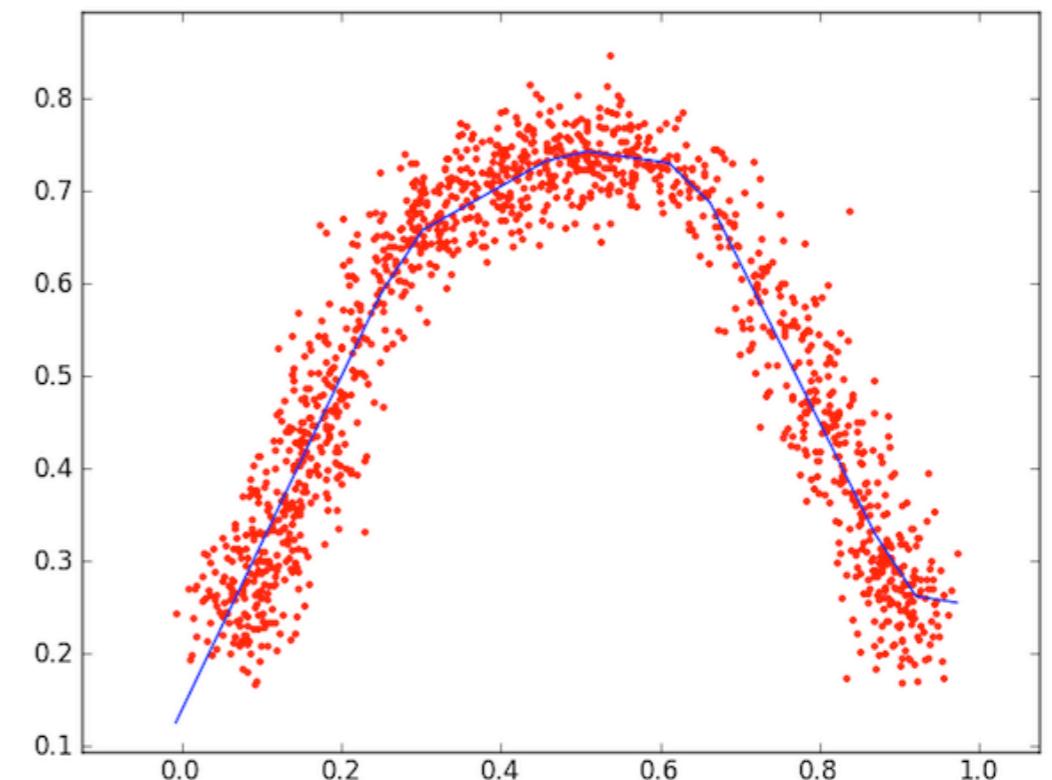


Stepwise Regression

https://en.wikipedia.org/wiki/Multivariate_adaptive_regression_spline

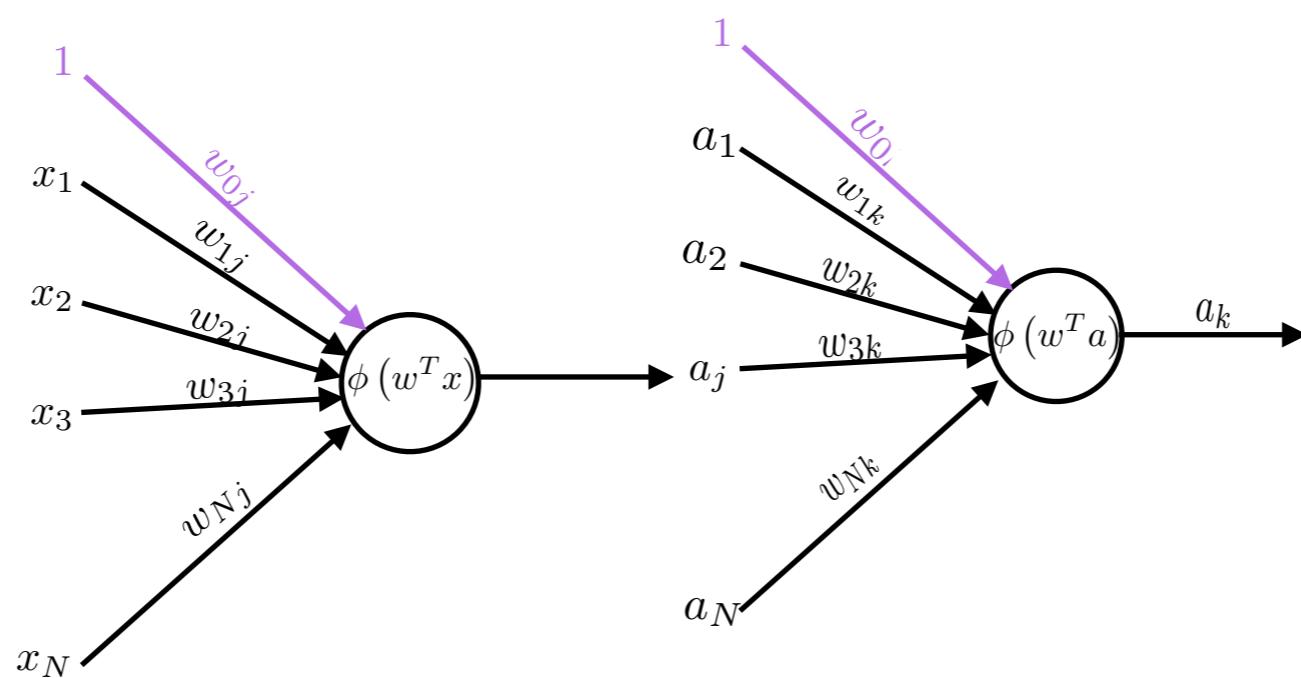
- Multivariate Adaptive Regression Spline (MARS) is the best known example
- Fit curves using a linear combination of: $\hat{f}(x) = \sum_i c_i B_i(x)$

$$\begin{aligned}y(x) &= 1.013 \\&+ 1.198 \max(0, x - 0.485) \\&- 1.803 \max(0, 0.485 - x) \\&- 1.321 \max(0, x - 0.283) \\&- 1.609 \max(0, x - 0.640) \\&+ 1.591 \max(0, x - 0.907)\end{aligned}$$



Forward Propagation

- The output of a perceptron is determined by a sequence of steps:
 - obtain the inputs
 - multiply the inputs by the respective weights
 - calculate output using the activation function
- To create a multi-layer perceptron, you can simply use the output of one layer as the input to the next one.



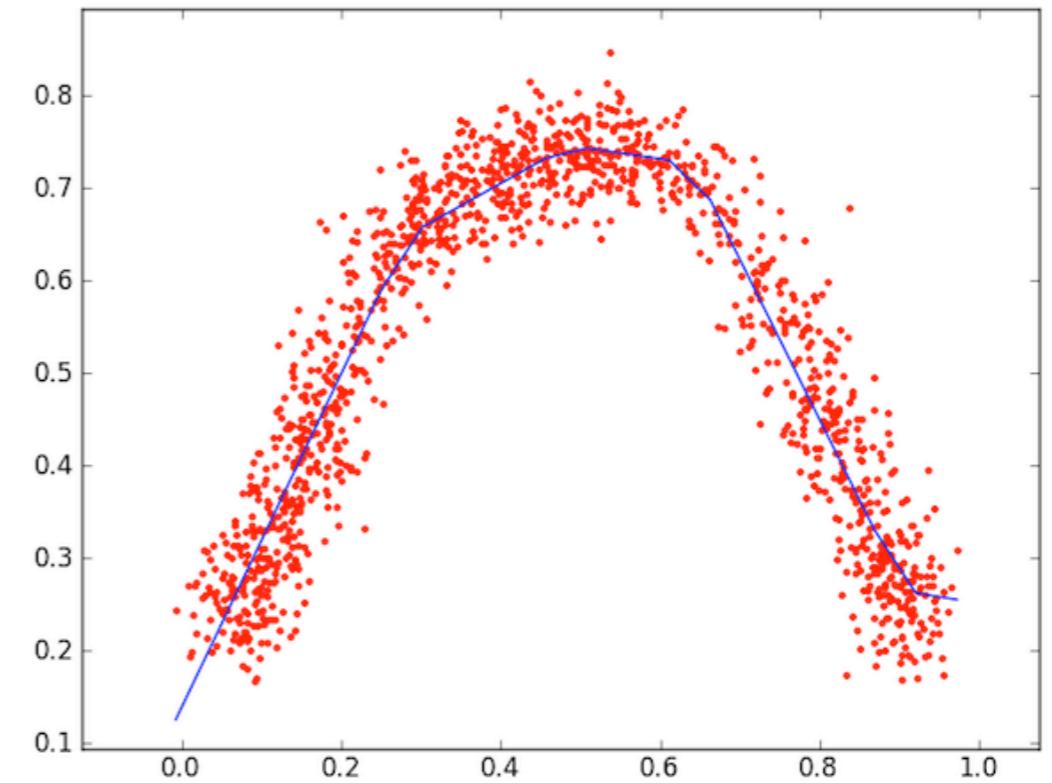
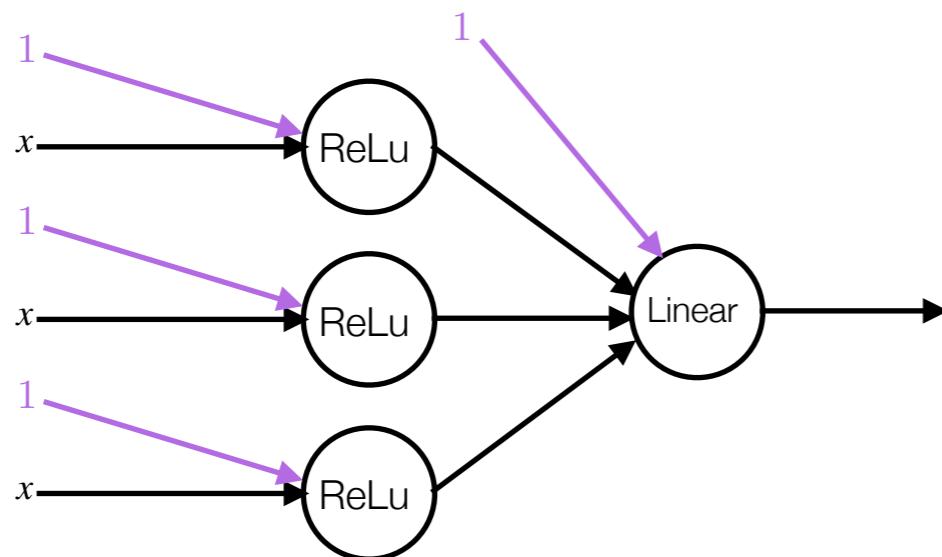
- But how can we propagate back the errors and update the weights?

Stepwise Regression

https://en.wikipedia.org/wiki/Multivariate_adaptive_regression_spline

$$\hat{f}(x) = \sum_i c_i B_i(x)$$

$$\begin{aligned}y(x) &= 1.013 \\&+ 1.198 \max(0, x - 0.485) \\&- 1.803 \max(0, 0.485 - x) \\&- 1.321 \max(0, x - 0.283) \\&- 1.609 \max(0, x - 0.640) \\&+ 1.591 \max(0, x - 0.907)\end{aligned}$$



Loss Functions

- For learning to occur, we must quantify how far off we are from the desired output. There are two common ways of doing this:
 - Quadratic error function:
 - Cross Entropy

$$E = \frac{1}{N} \sum_n |y_n - a_n|^2$$
$$J = -\frac{1}{N} \sum_n \left[y_n^T \log a_n + (1 - y_n)^T \log (1 - a_n) \right]$$

The **Cross Entropy** is complementary to **sigmoid** activation in the output layer and improves its stability.

Regularization

- Helps keep weights relatively small by adding a penalization to the cost function.
- Two common choices:

$$\hat{J}_w(X) = J_w(X) + \lambda \sum_{ij} |w_{ij}| \quad \text{"Lasso"}$$

$$\hat{J}_w(X) = J_w(X) + \lambda \sum_{ij} w_{ij}^2 \quad \text{L2}$$

- Lasso helps with feature selection by driving less important weights to zero

Backward Propagation of Errors (BackProp)

- BackProp operates in two phases:
 - Forward propagate the inputs and calculate the deltas
 - Update the weights
- The error at the output is a **weighted average difference** between predicted output and the observed one.
- For inner layers there is no “real output”!

BackProp

- Let $\delta^{(l)}$ be the error at each of the total L layers:

- Then:

$$\delta^{(L)} = h_w(X) - y$$

- And for every other layer, **in reverse order**:

$$\delta^{(l)} = W^{(l)T} \delta^{(l+1)} * \phi^\dagger(z^{(l)})$$

- Until:

$$\delta^{(1)} \equiv 0$$

as there's no error on the input layer.

- And finally:

$$\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial w_{ij}^{(l)}} J_w(X, \vec{y}) = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda w_{ij}^{(l)}$$

A practical example - MNIST

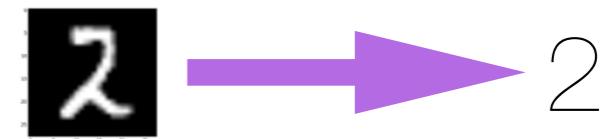
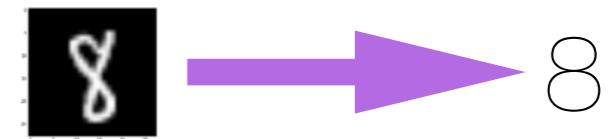
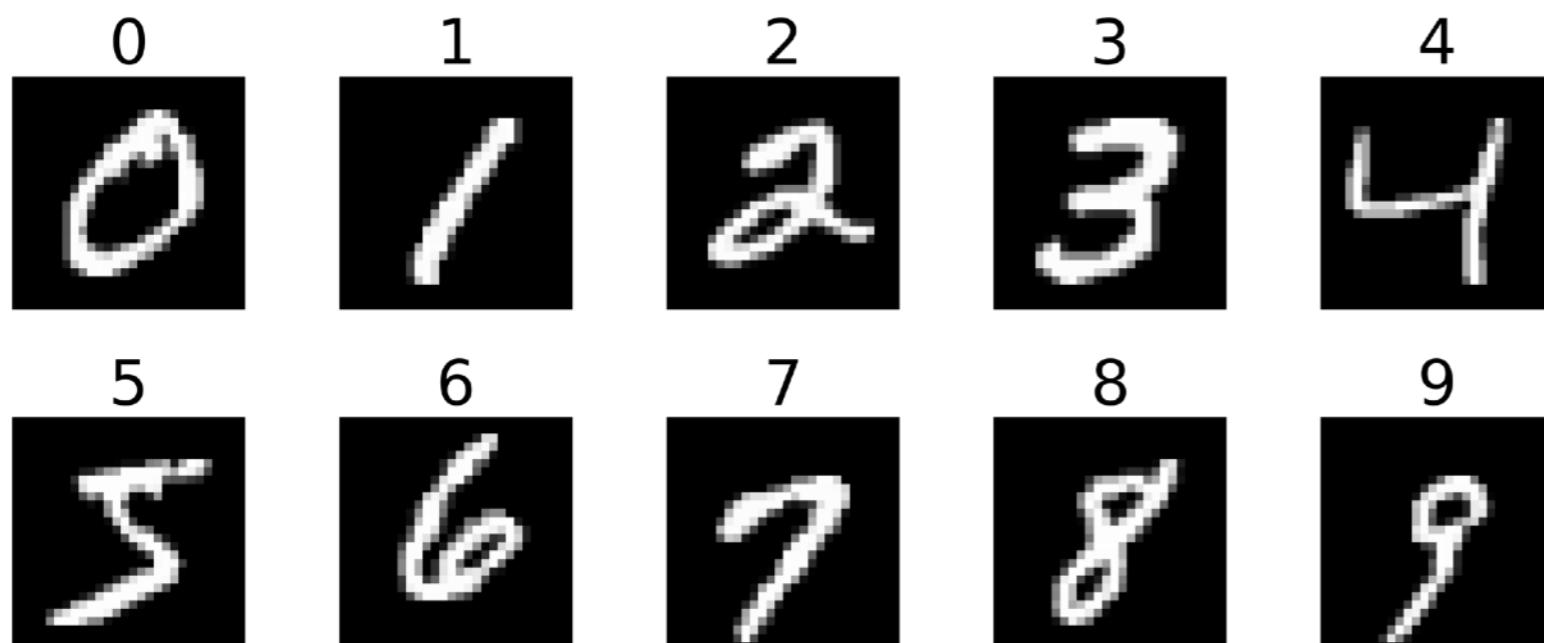
THE MNIST DATABASE

of handwritten digits

[Yann LeCun](#), Courant Institute, NYU

[Corinna Cortes](#), Google Labs, New York

[Christopher J.C. Burges](#), Microsoft Research, Redmond

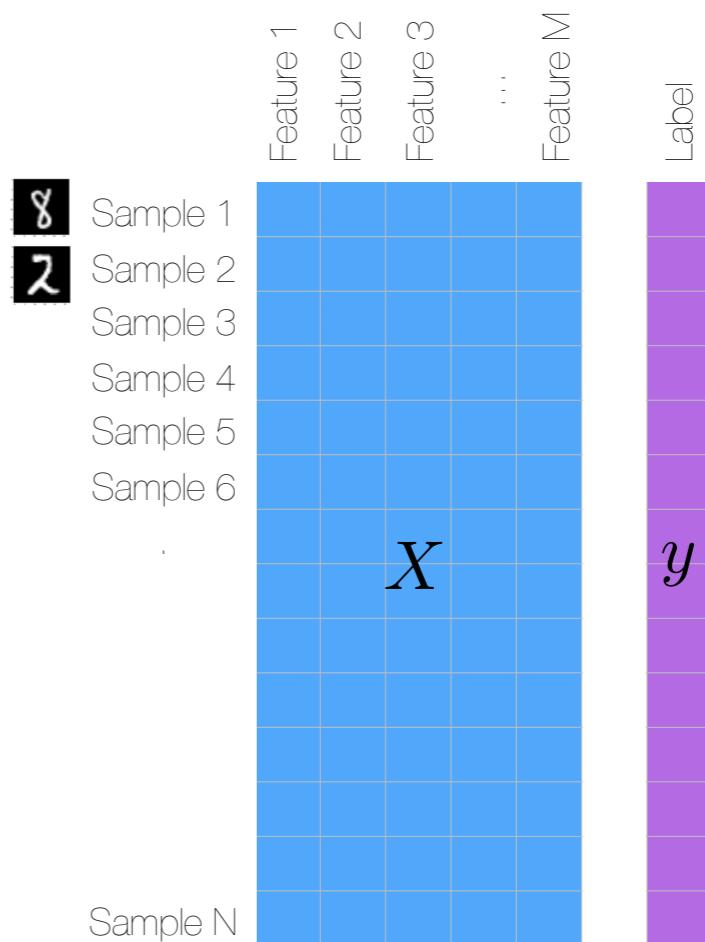
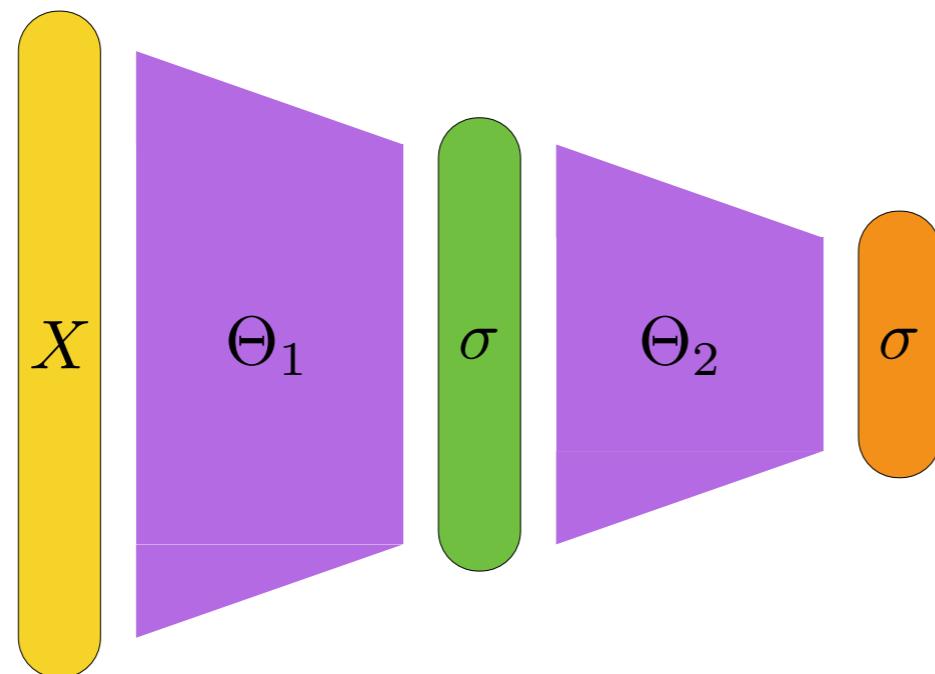


yann.lecun.com/exdb/mnist/

A practical example - MNIST

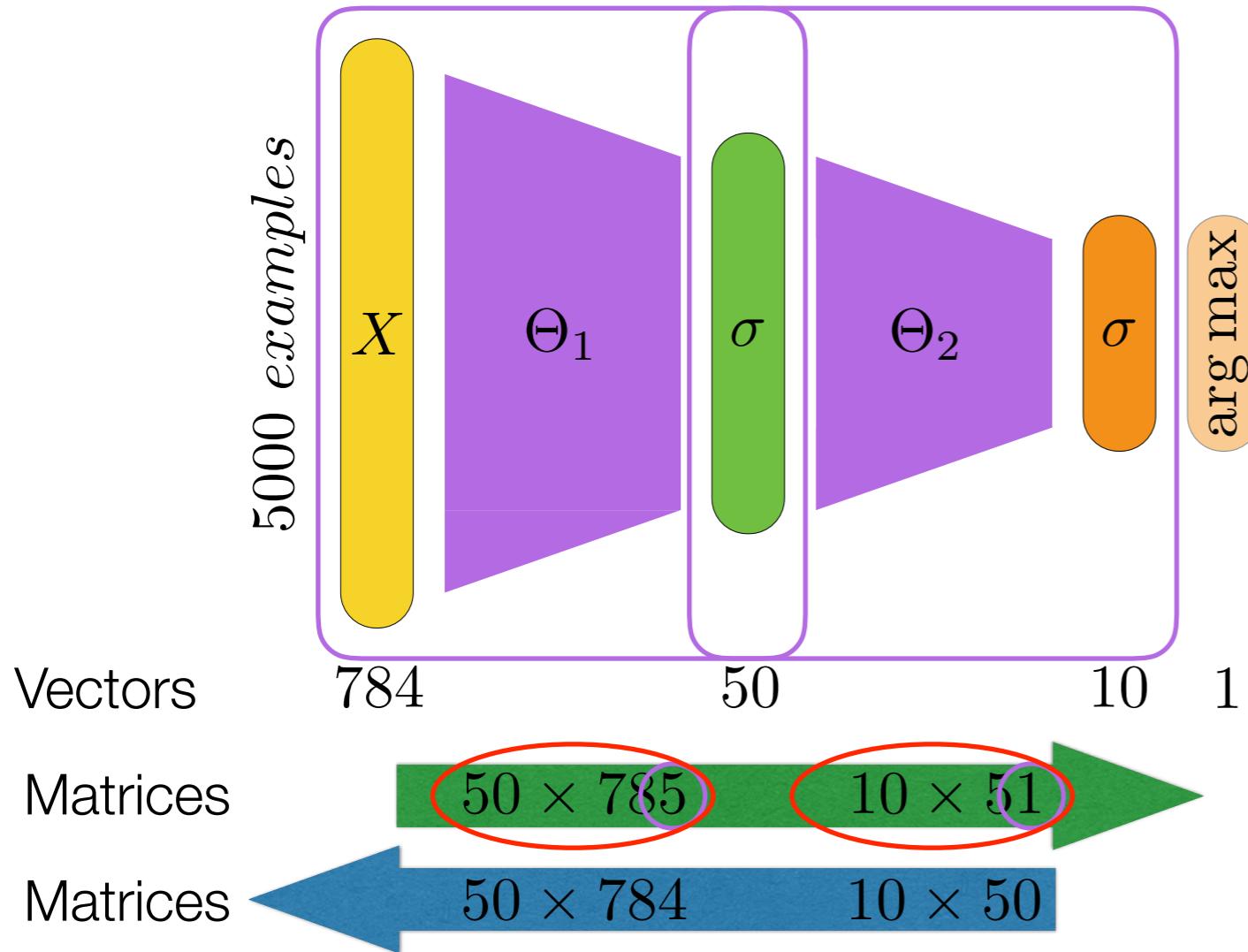
THE MNIST DATABASE of handwritten digits

[Yann LeCun](#), Courant Institute, NYU
[Corinna Cortes](#), Google Labs, New York
[Christopher J.C. Burges](#), Microsoft Research, Redmond



yann.lecun.com/exdb/mnist/

A practical example - MNIST



```
def forward(Theta, X, active):  
    N = X.shape[0]  
  
    # Add the bias column  
    X_ = np.concatenate((np.ones((N, 1)), X), 1)  
  
    # Multiply by the weights  
    z = np.dot(X_, Theta.T)  
  
    # Apply the activation function  
    a = active(z)  
  
    return a
```

Forward Propagation

Backward Propagation

```
def predict(Theta1, Theta2, X):  
    h1 = forward(Theta1, X, sigmoid)  
    h2 = forward(Theta2, h1, sigmoid)  
  
    return np.argmax(h2, 1)
```



Code - Simple Network
<https://github.com/DataForScience/DeepLearning>

Practical Considerations

- So far we have looked at very idealized cases. Reality is never this simple!
- In practice, many details have to be considered:
 - Data normalization
 - Overfitting
 - Hyperparameters
 - Bias, Variance tradeoffs
 - etc...

Data Normalization

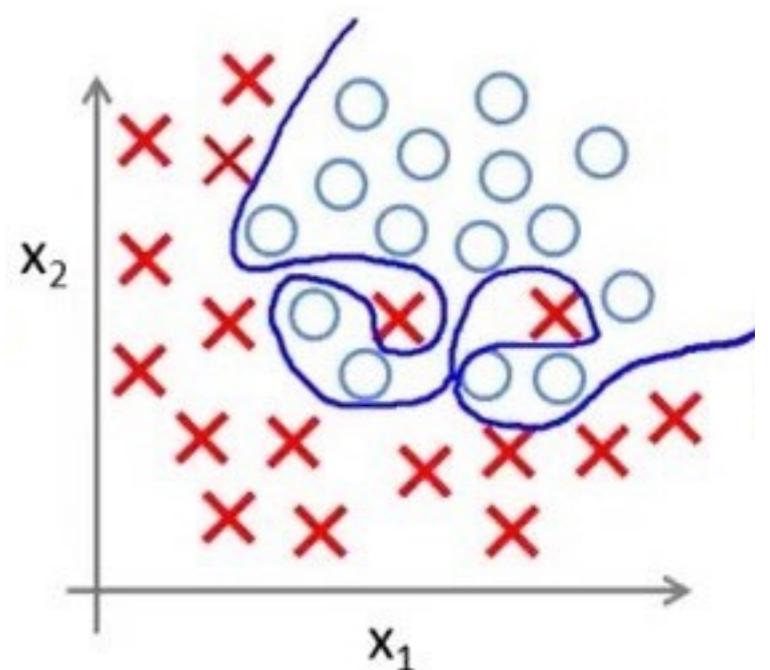
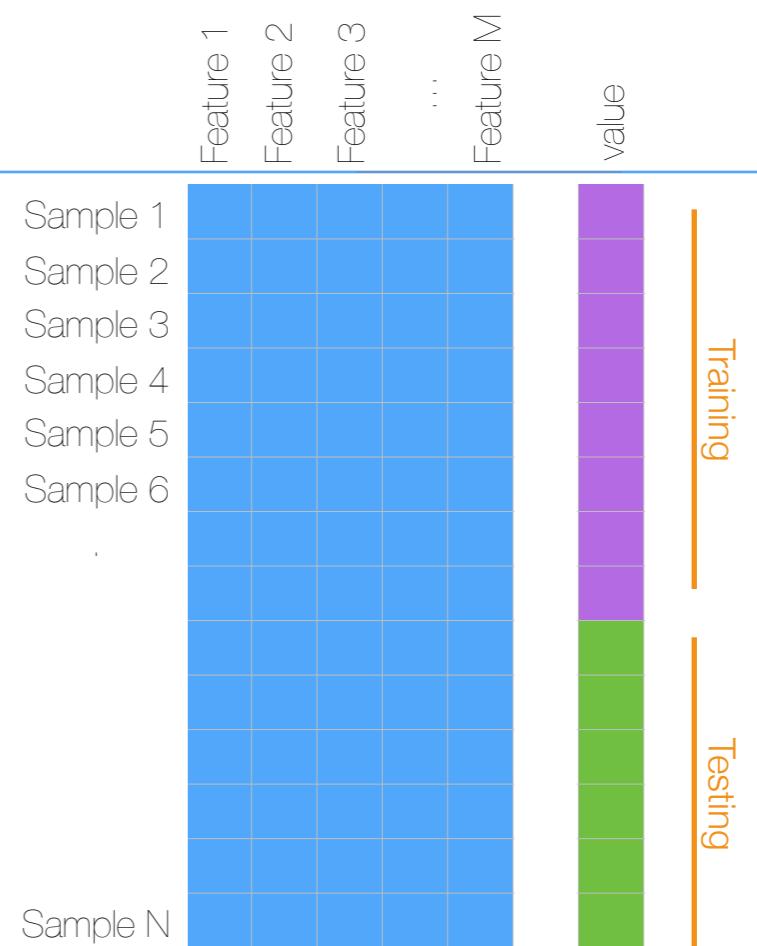
- The range of raw data values can vary widely.
- Using feature with very different ranges in the same analysis can cause numerical problems.
Many algorithms are linear or use euclidean distances that are heavily influenced by the numerical values used (cm vs km, for example)
- To avoid difficulties, it's common to rescale the range of all features in such a way that each feature follows within the same range.
- Several possibilities:
 - Rescaling - $\hat{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$
 - Standardization - $\hat{x} = \frac{x - \mu_x}{\sigma_x}$
 - Normalization - $\hat{x} = \frac{x}{\|x\|}$
- In the rest of the discussion we will assume that the data has been normalized in some

Data Normalization

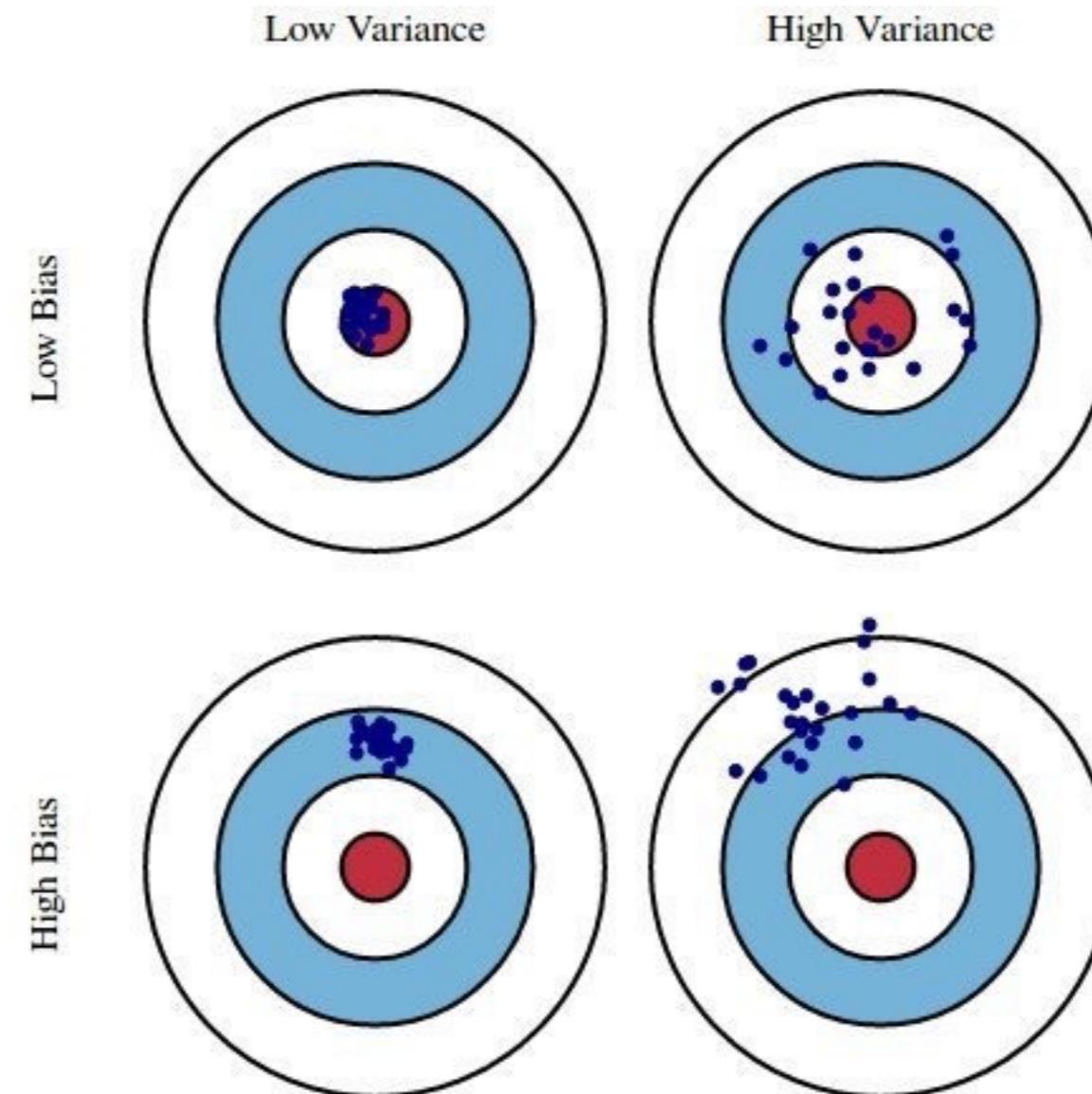
- The range of raw data values can vary widely.
- Using feature with very different ranges in the same analysis can cause numerical problems.
Many algorithms are linear or use euclidean distances that are heavily influenced by the numerical values used (cm vs km, for example)
- To avoid difficulties, it's common to rescale the range of all features in such a way that each feature follows within the same range.
- Several possibilities:
 - Rescaling - $\hat{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$
 - Standardization - $\hat{x} = \frac{x - \mu_x}{\sigma_x}$
 - Normalization - $\hat{x} = \frac{x}{\|x\|}$
- In the rest of the discussion we will assume that the data has been normalized in some

Supervised Learning - Overfitting

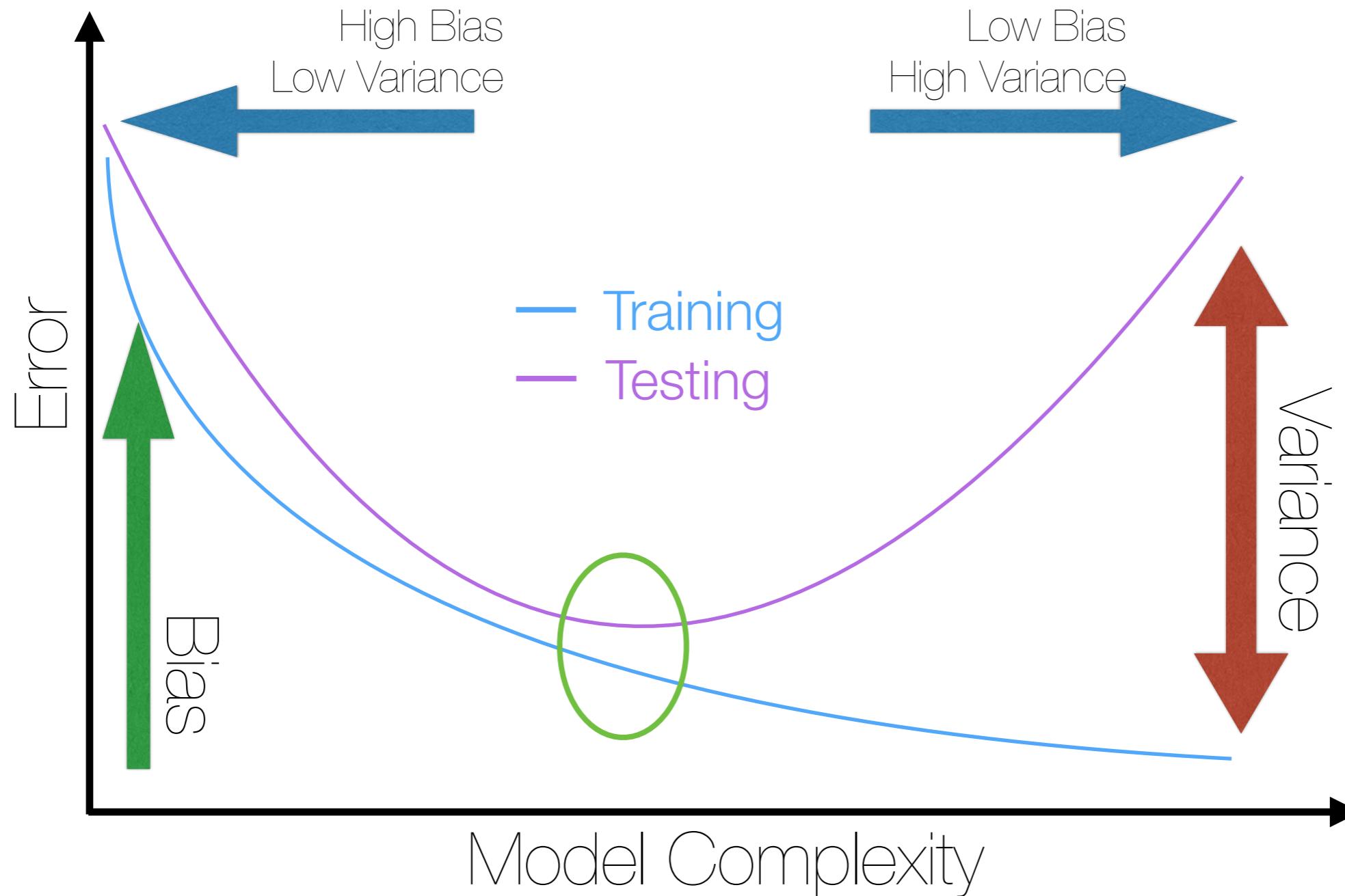
- "Learning the noise"
- "**Memorization**" instead of "generalization"
- How can we prevent it?
 - Split dataset into two subsets: **Training** and **Testing**
 - Train model using only the **Training** dataset and evaluate results in the previously unseen **Testing** dataset.
- Different heuristics on how to split:
 - Single split
 - k-fold cross validation: split dataset in **k** parts, train in **k-1** and evaluate in **1**, repeat **k** times and average results.



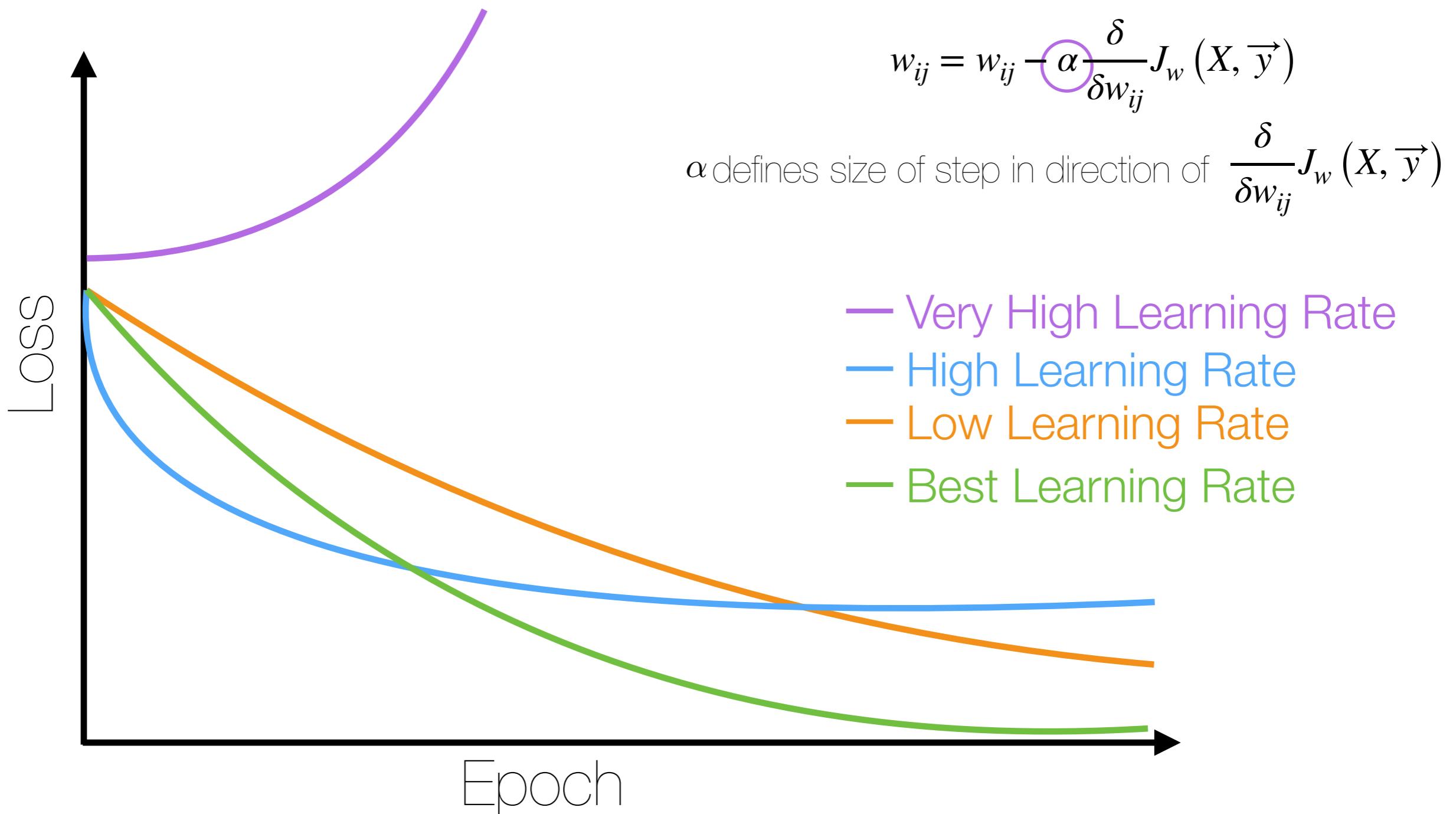
Bias-Variance Tradeoff



Bias-Variance Tradeoff



Learning Rate



Tips

- **online learning** - update weights after **each** case
 - might be useful to update model as new data is obtained
 - subject to fluctuations
- **mini-batch** - update weights after a “**small**” number of cases
 - batches should be balanced
 - if dataset is redundant, the gradient estimated using only a fraction of the data is a good approximation to the full gradient.
- **momentum** - let gradient change the **velocity** of weight change instead of the value directly
- **rmsprop** - divide learning rate for each weight by a **running average** of “recent” gradients
- **learning rate** - vary over the course of the training procedure and use different learning rates for each weight

Generalization

- Neural Networks are extremely modular in their design with
- Fortunately, we can write code that is also modular and can easily handle arbitrary numbers of layers
- Let's describe the structure of our network as a list of weight matrices and activation functions
- We also need to keep track of the gradients of the activation functions so let us define a simple class:

```
class Activation(object):  
    def f(z):  
        pass  
  
    def df(z):  
        pass  
  
class Linear(Activation):  
    def f(z):  
        return z  
  
    def df(z):  
        return np.ones(z.shape)  
  
class Sigmoid(Activation):  
    def f(z):  
        return 1./(1+np.exp(-z))  
  
    def df(z):  
        h = Sigmoid.f(z)  
        return h*(1-h)
```

Generalization

- Now we can describe our simple MNIST model with:

```
Thetas = []
Thetas.append(init_weights(input_layer_size, hidden_layer_size))
Thetas.append(init_weights(hidden_layer_size, num_labels))

model = []

model.append(Thetas[0])
model.append(Sigmoid)
model.append(Thetas[1])
model.append(Sigmoid)
```

- Where **Sigmoid** is an object that contains both the sigmoid function and its gradient as was defined in the previous slide.

Generalization - Forward propagation

```
def forward(Theta, X, active):
    N = X.shape[0]

    # Add the bias column
    X_ = np.concatenate((np.ones((N, 1)), X), 1)

    # Multiply by the weights
    z = np.dot(X_, Theta.T)

    # Apply the activation function
    a = active.f(z)

    return a

def predict(model, X):
    h = X.copy()

    for i in range(0, len(model), 2):
        theta = model[i]
        activation = model[i+1]

        h = forward(theta, h, activation)

    return np.argmax(h, 1)
```

```

def backprop(model, X, y):
    M = X.shape[0]

    Thetas = model[0::2]
    activations = model[1::2]
    layers = len(Thetas)

    K = Thetas[-1].shape[0]
    J = 0
    Deltas = []

    for i in range(layers):
        Deltas.append(np.zeros(Thetas[i].shape))

    deltas = [0, 0, 0, 0]

    for i in range(M):
        As = []
        Zs = [0]
        Hs = [X[i]]
        # Forward propagation, saving intermediate results
        As.append(np.concatenate(([1], Hs[0]))) # Input layer

        for l in range(1, layers+1):
            Zs.append(np.dot(Thetas[l-1], As[l-1]))
            Hs.append(activations[l-1].f(Zs[l]))
            As.append(np.concatenate(([1], Hs[l])))

        y0 = one_hot(K, y[i])

        # Cross entropy
        J -= np.dot(y0.T, np.log(Hs[2]))+np.dot((1-y0).T, np.log(1-Hs[2]))

        # Calculate the weight deltas
        deltas[layers] = Hs[layers]-y0

        for l in range(layers-1, 1, -1):
            deltas[l] = np.dot(Thetas[l-1].T, deltas[l+1])[1:]*activations[l-1].df(Zs[l-1])
            Deltas[l] += np.outer(deltas[l+1], As[l])

    J /= M
    grads = []
    grads.append(Deltas[0]/M)
    grads.append(Deltas[1]/M)

return [J, grads]

```

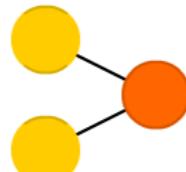


Code - Modular Network
<https://github.com/DataForScience/DeepLearning>

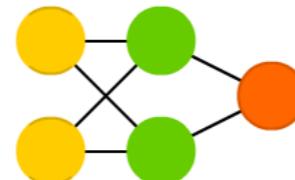
Neural Network Architectures

- (○) Backfed Input Cell
- (○) Input Cell
- (△) Noisy Input Cell
- (●) Hidden Cell
- (○) Probabilistic Hidden Cell
- (△) Spiking Hidden Cell
- (●) Output Cell
- (○) Match Input Output Cell
- (●) Recurrent Cell
- (○) Memory Cell
- (△) Different Memory Cell
- (●) Kernel
- (○) Convolution or Pool

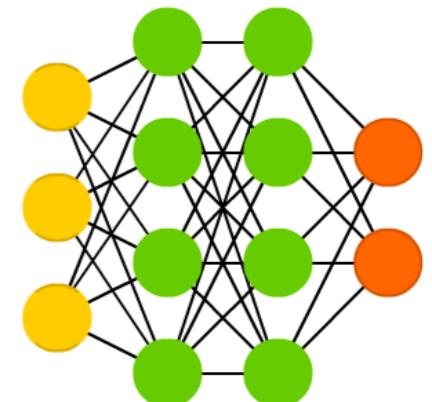
Perceptron (P)



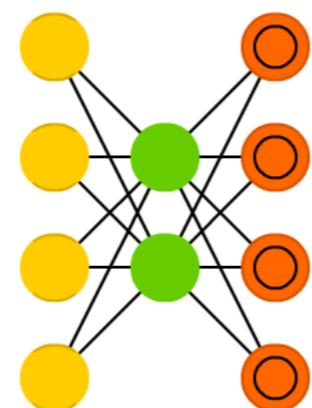
Feed Forward (FF)



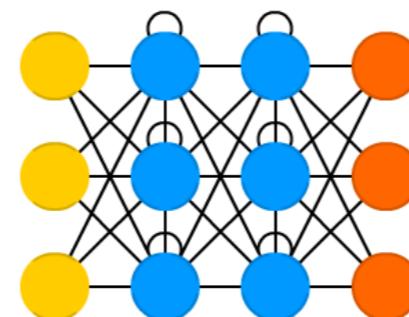
Deep Feed Forward (DFF)



Auto Encoder (AE)

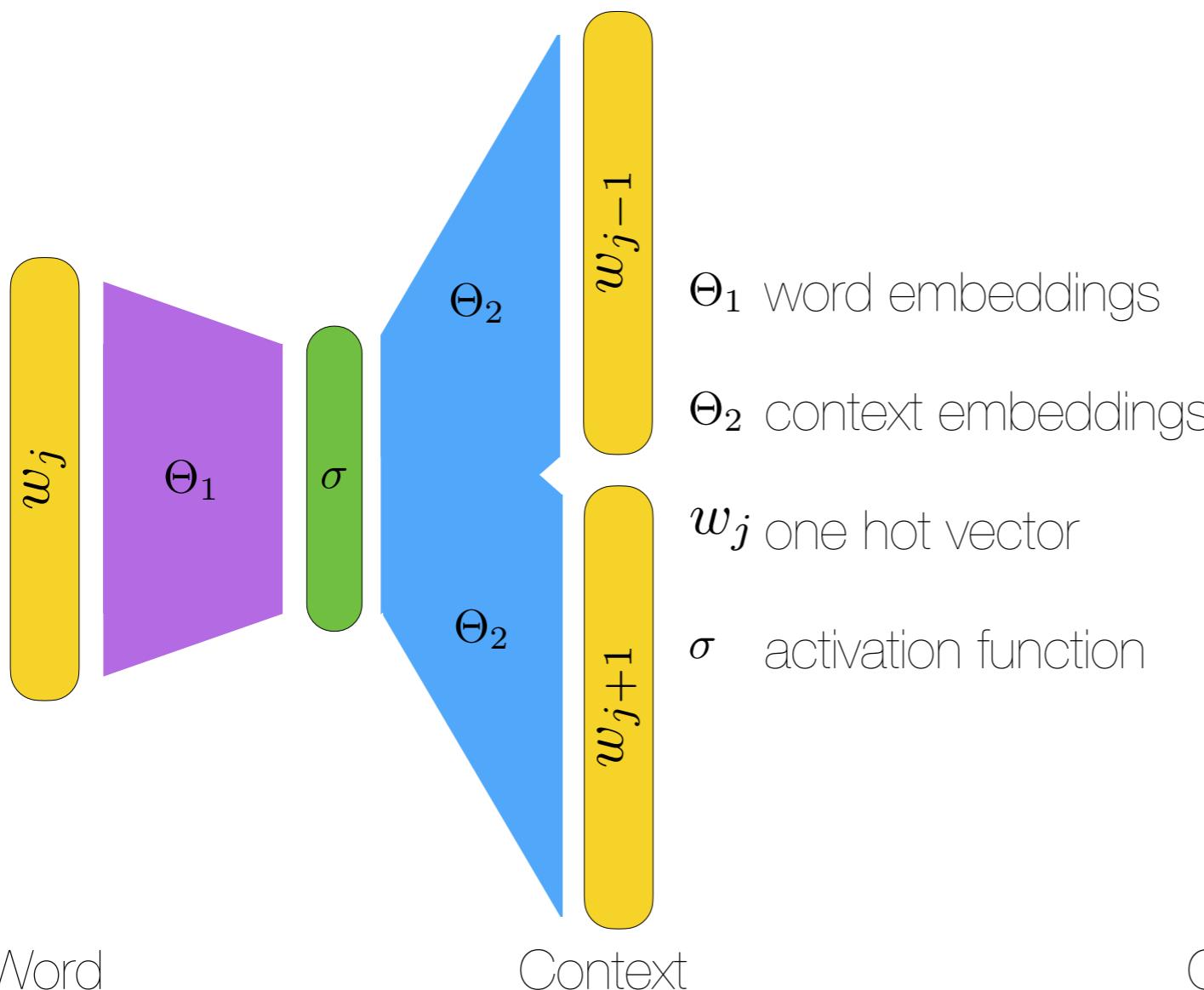


Recurrent Neural Network (RNN)



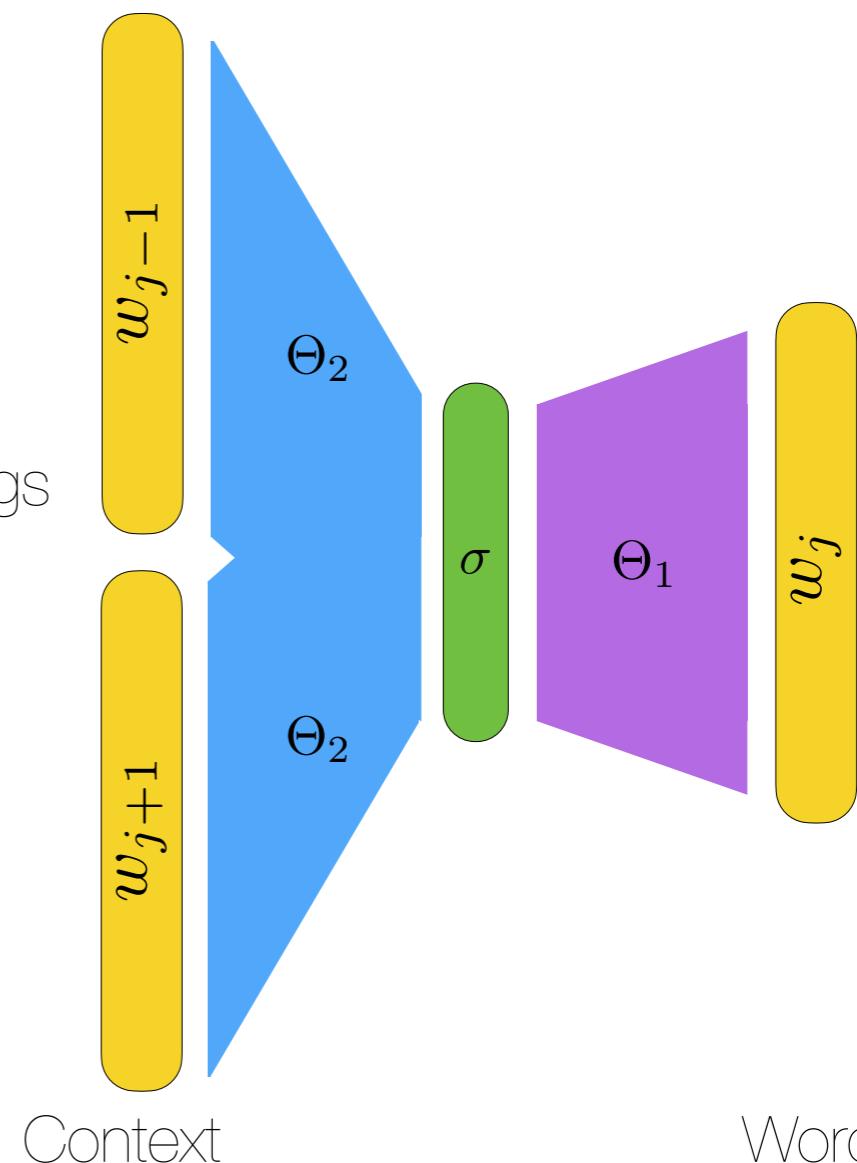
Skipgram

$$\max p(C|w)$$

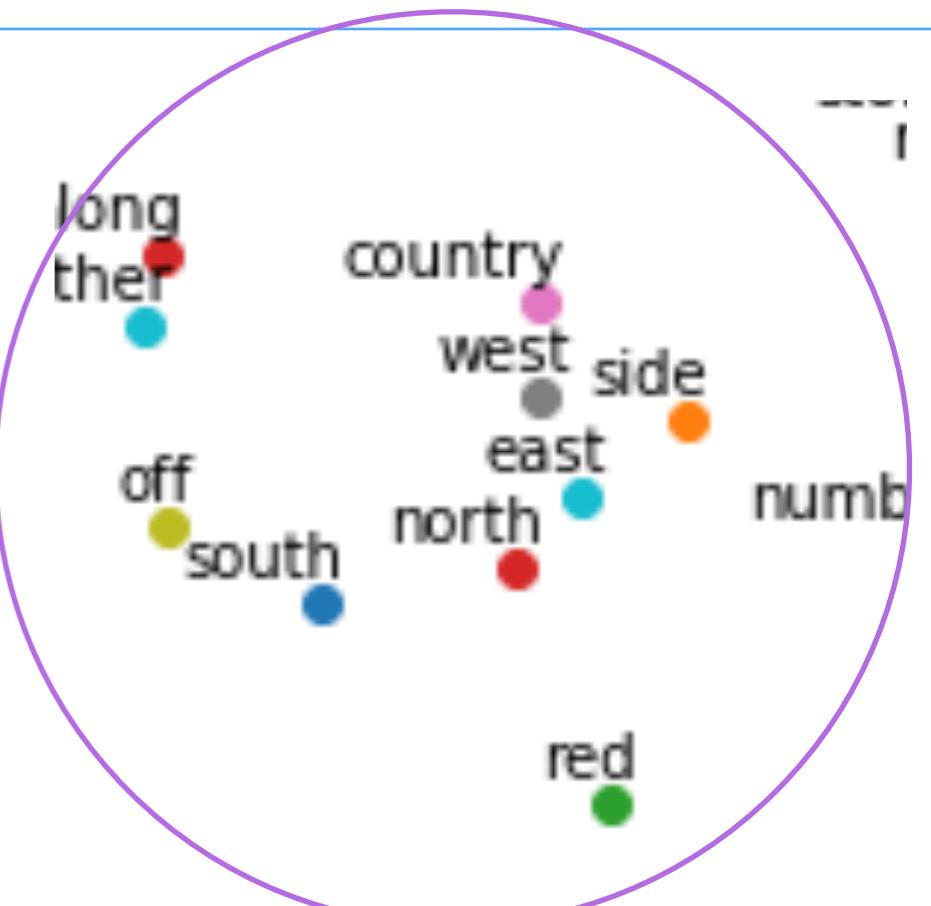
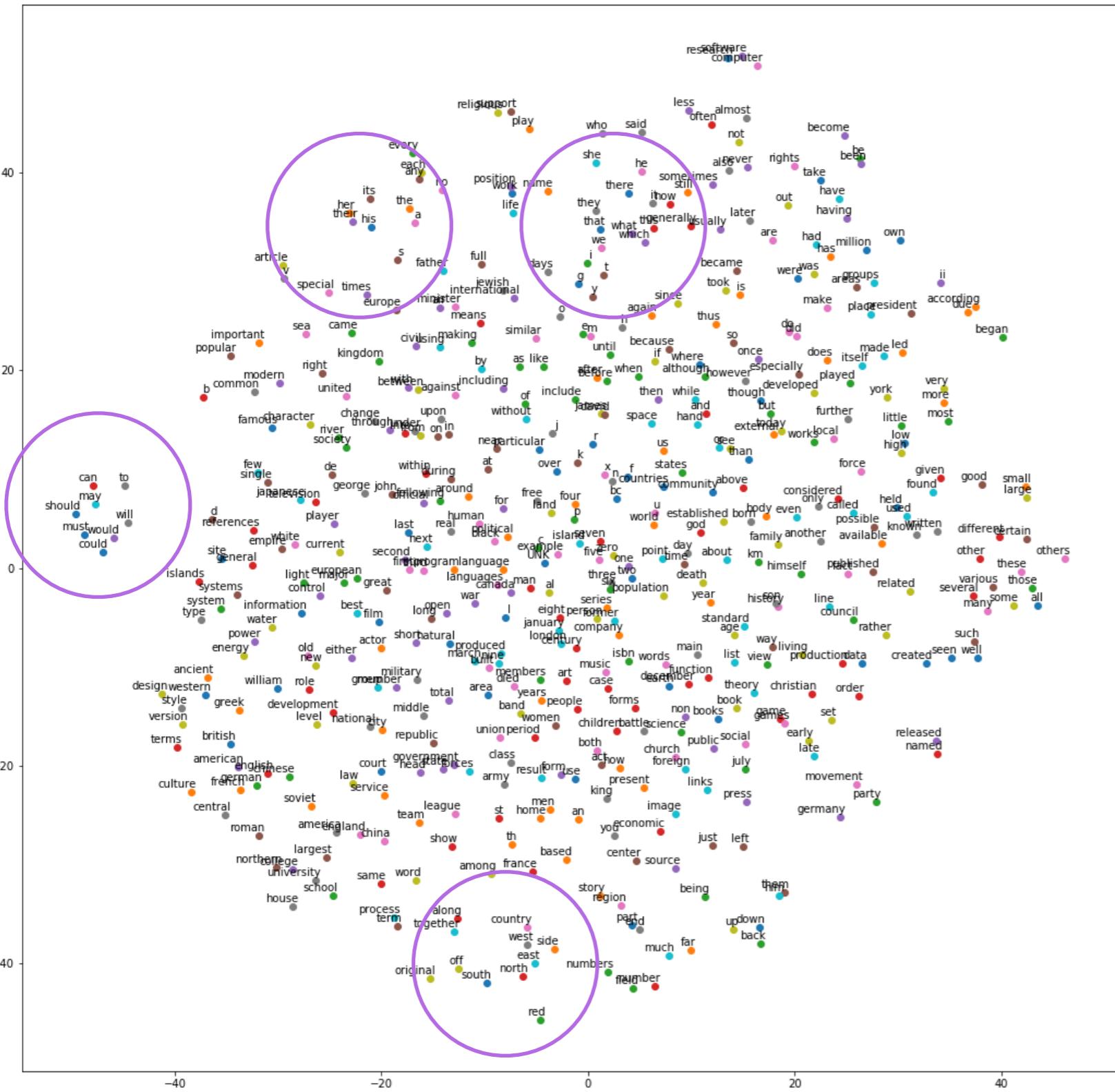


Continuous Bag of Words

$$\max p(w|C)$$



Visualization





"You shall know a word by the company it keeps"
(J. R. Firth)

Analogies

- The embedding of each word is a function of the context it appears in:

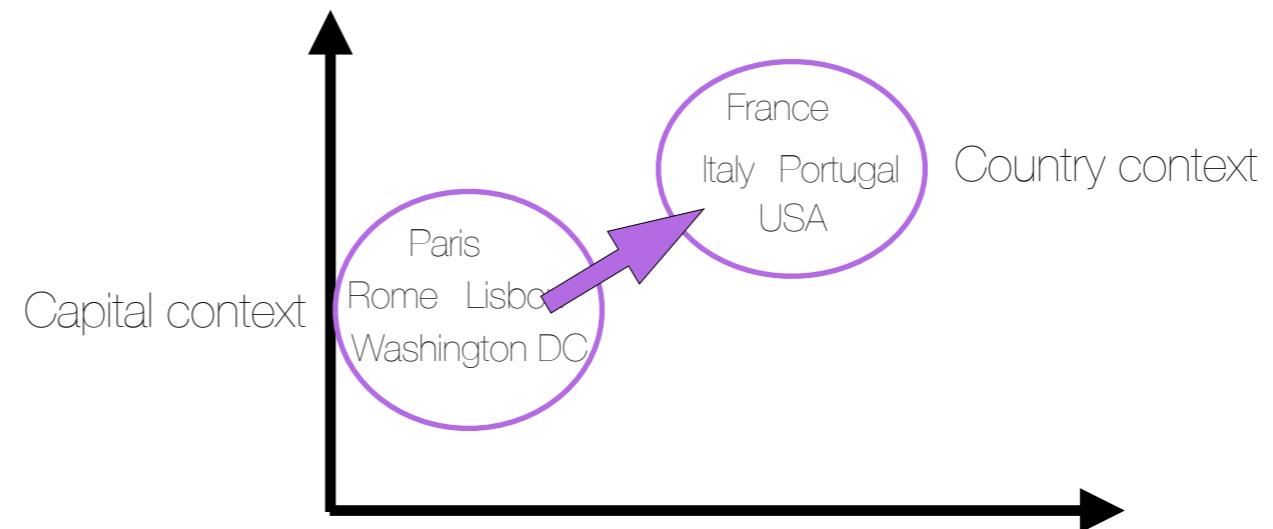
$$\sigma(\text{red}) = f(\text{context}(\text{red}))$$

- words that appear in similar contexts will have similar embeddings:

$$\text{context}(\text{red}) \approx \text{context}(\text{blue}) \implies \sigma(\text{red}) \approx \sigma(\text{blue})$$

- "Distributional hypothesis" in linguistics

Geometrical relations
between contexts imply
semantic relations
between words!



$$\sigma(\text{France}) - \sigma(\text{Paris}) + \sigma(\text{Rome}) = \sigma(\text{Italy})$$

$$\vec{b} - \vec{a} + \vec{c} = \vec{d}$$

Analogies

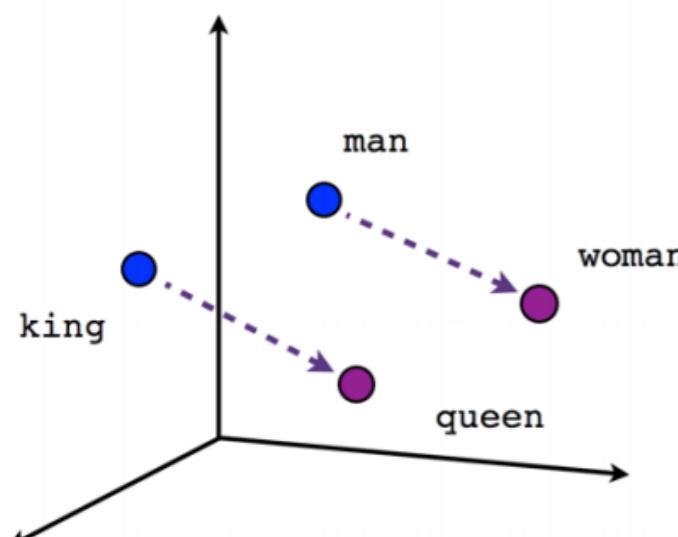
<https://www.tensorflow.org/tutorials/word2vec>

What is the word **d** that is most **similar** to **b** and **c** and most **dissimilar** to **a**?

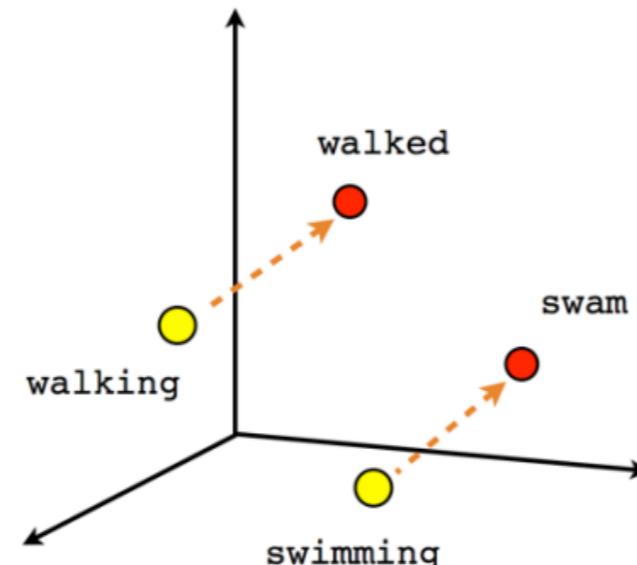
$$\vec{b} - \vec{a} + \vec{c} = \vec{d}$$

$$d^\dagger = \operatorname{argmax}_x \frac{(\vec{b} - \vec{a} + \vec{c})^T}{\|\vec{b} - \vec{a} + \vec{c}\|} \vec{x}$$

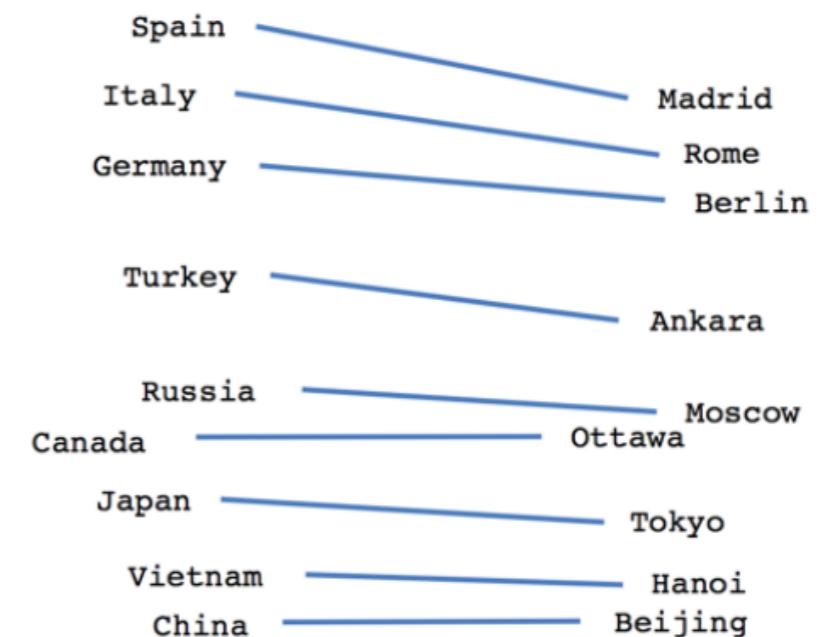
$$d^\dagger \sim \operatorname{argmax}_x (\vec{b}^T \vec{x} - \vec{a}^T \vec{x} + \vec{c}^T \vec{x})$$



Male-Female

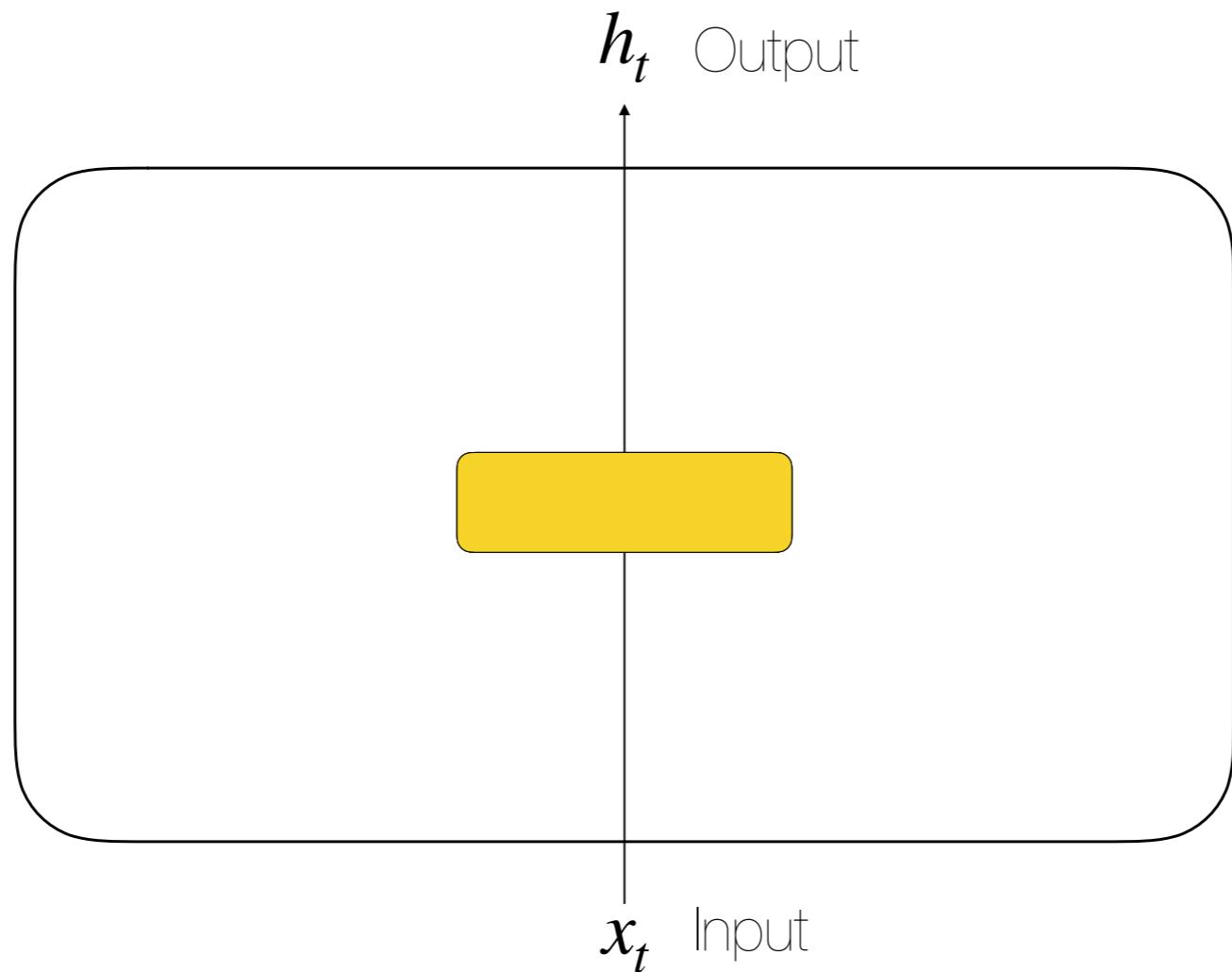


Verb tense



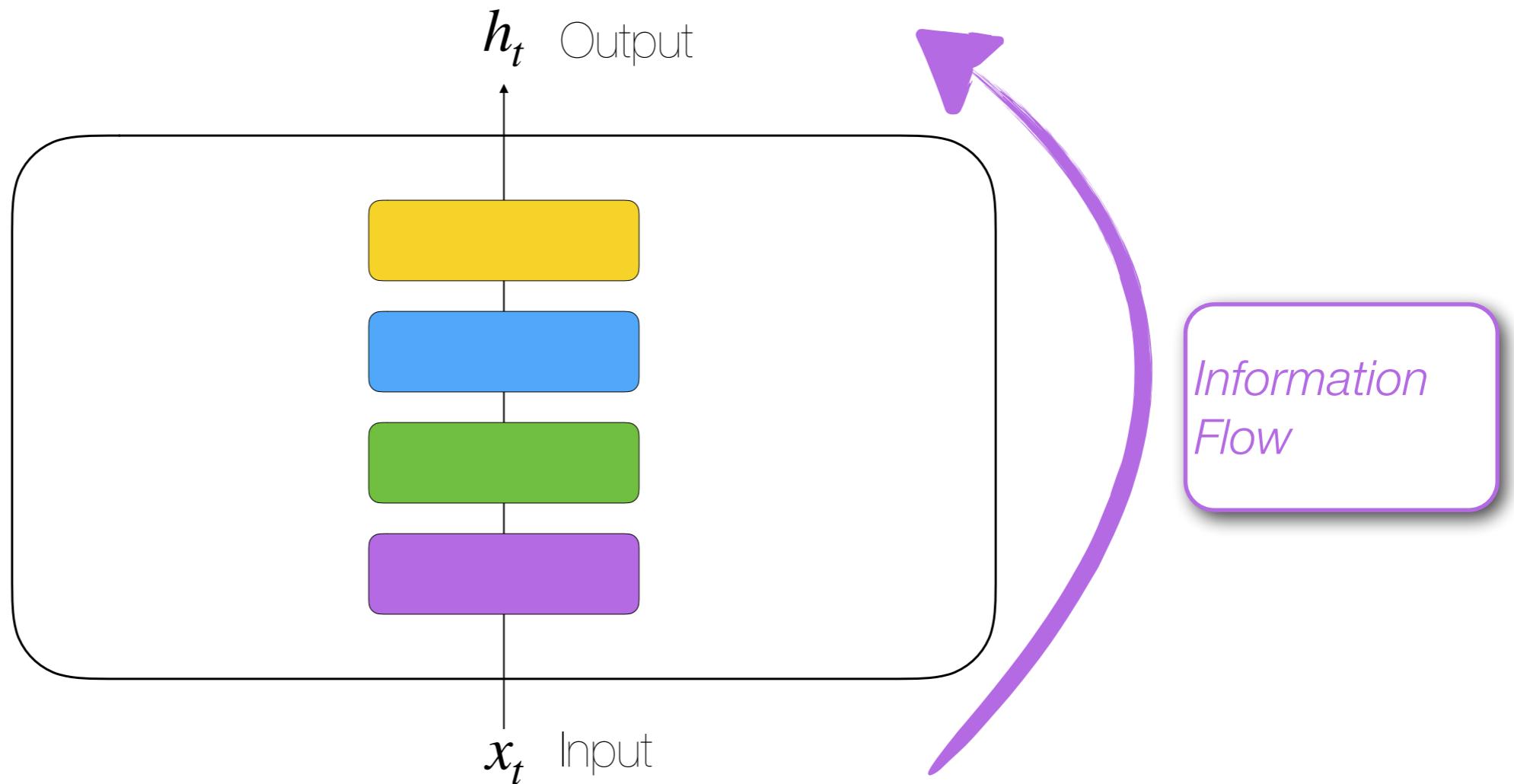
Country-Capital

Feed Forward Networks



$$h_t = f(x_t)$$

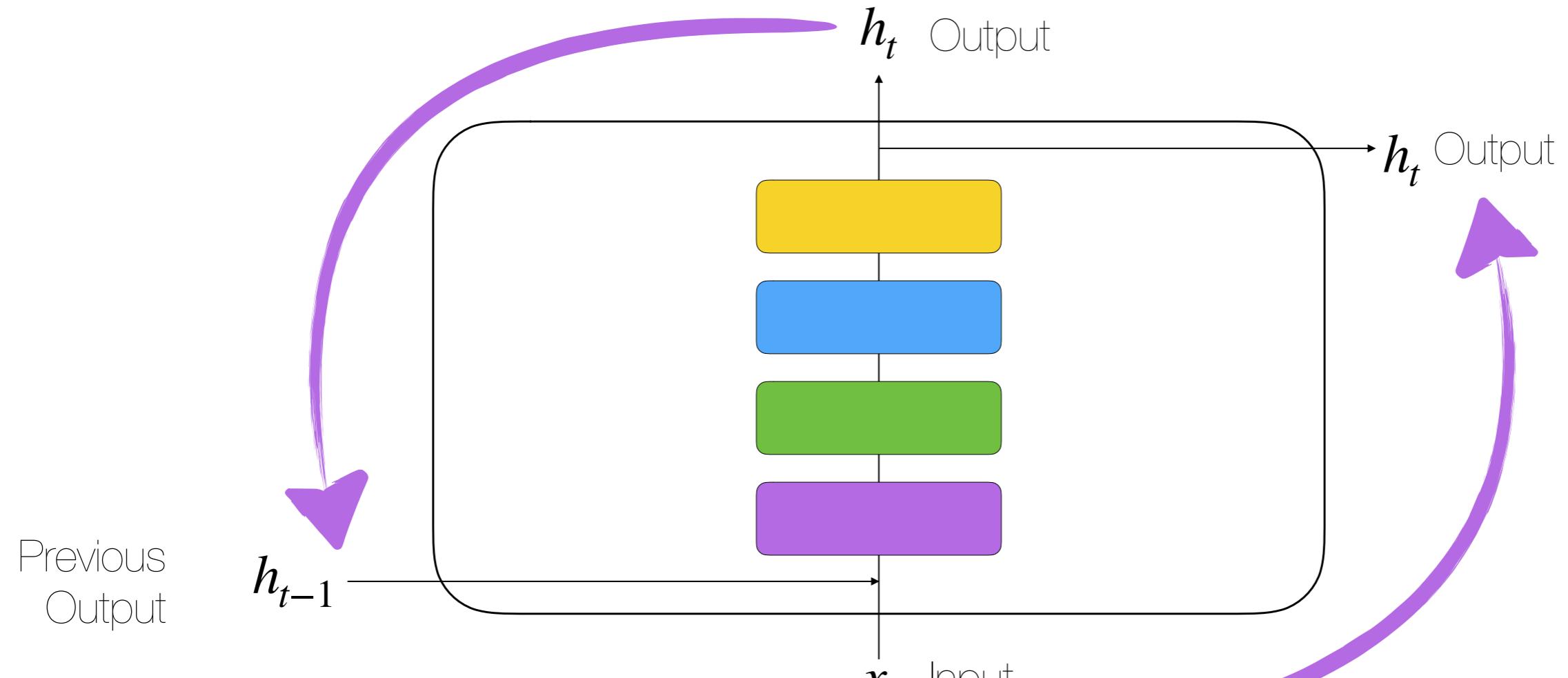
Feed Forward Networks



$$h_t = f(x_t)$$

Information
Flow

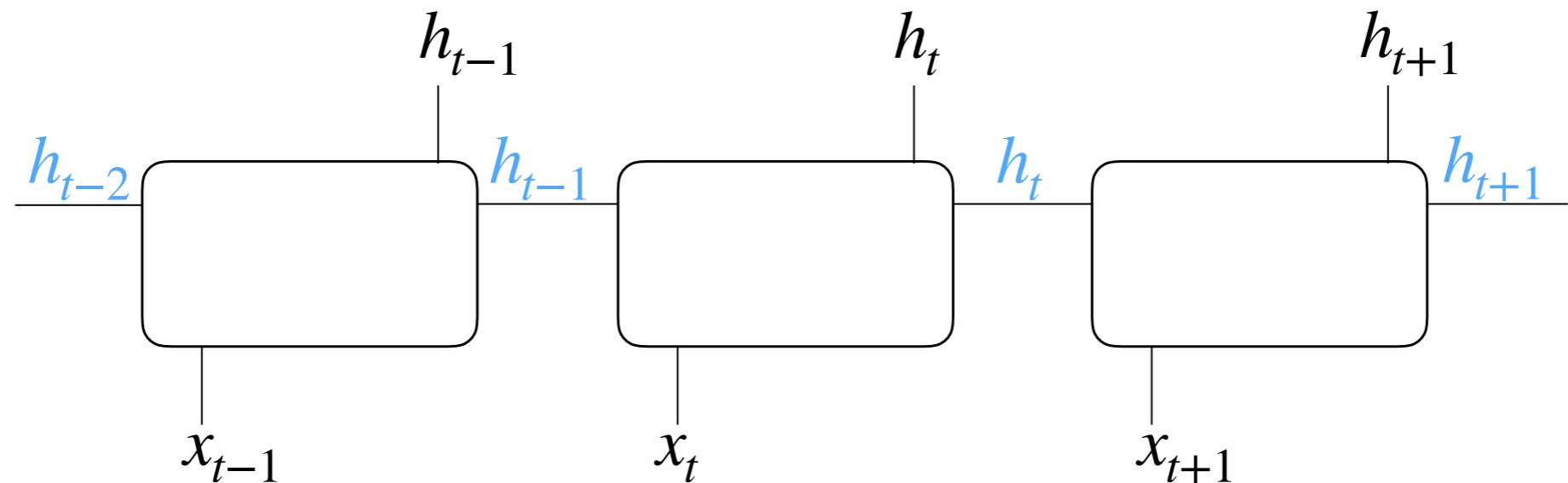
Recurrent Neural Network (RNN)



$$h_t = f(x_t, h_{t-1})$$

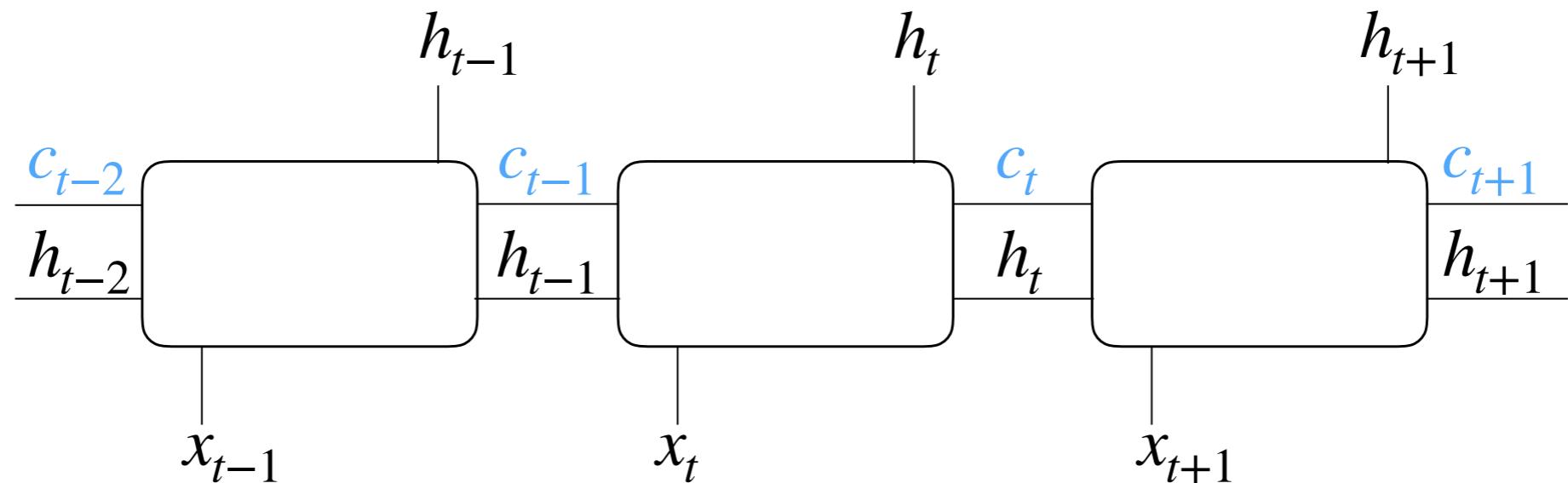
Recurrent Neural Network (RNN)

- Each output depends (implicitly) on all previous **outputs**.
- Input sequences generate output sequences (**seq2seq**)

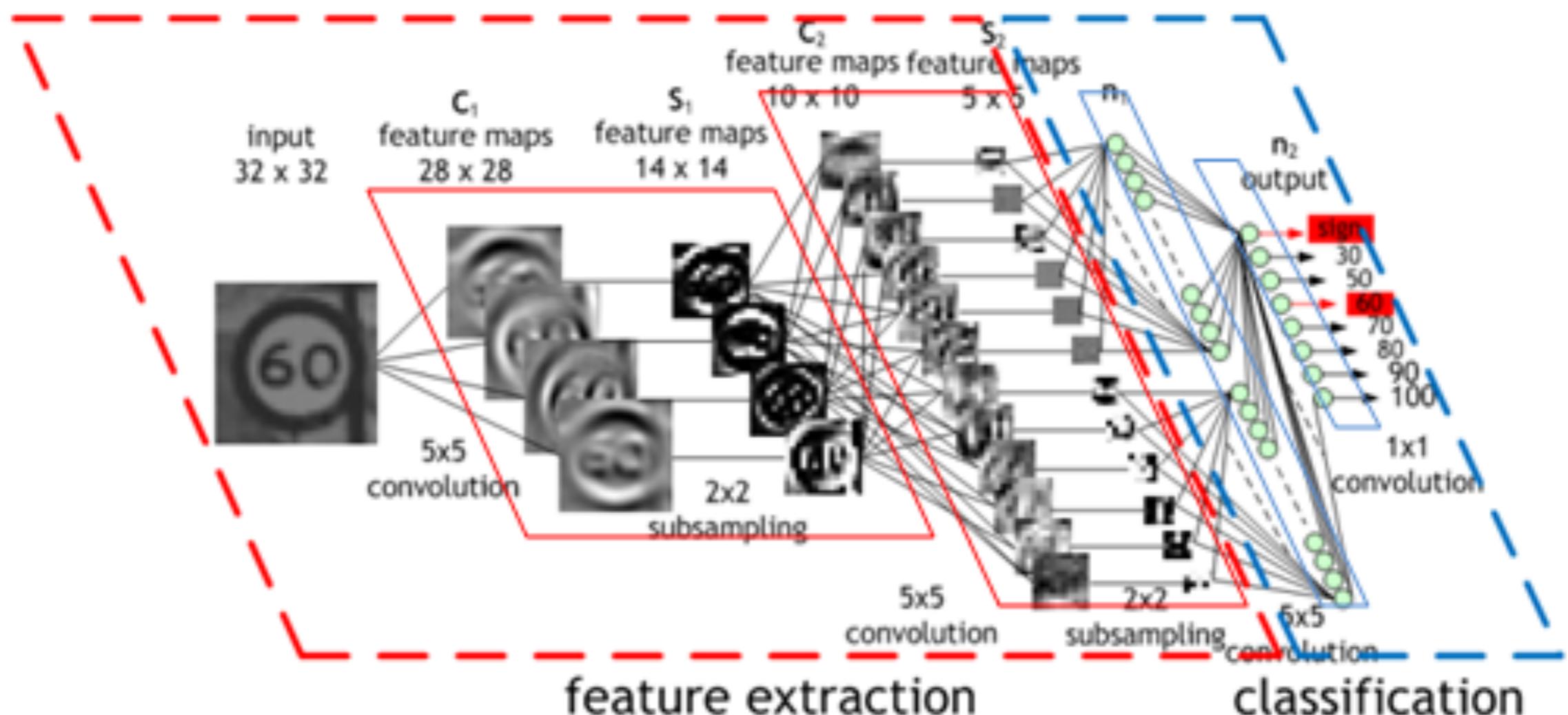


Long-Short Term Memory (LSTM)

- What if we want to keep explicit information about previous states (**memory**)?
- How much information is kept, can be controlled through gates.
- LSTMs were first introduced in [1997](#) by Hochreiter and Schmidhuber



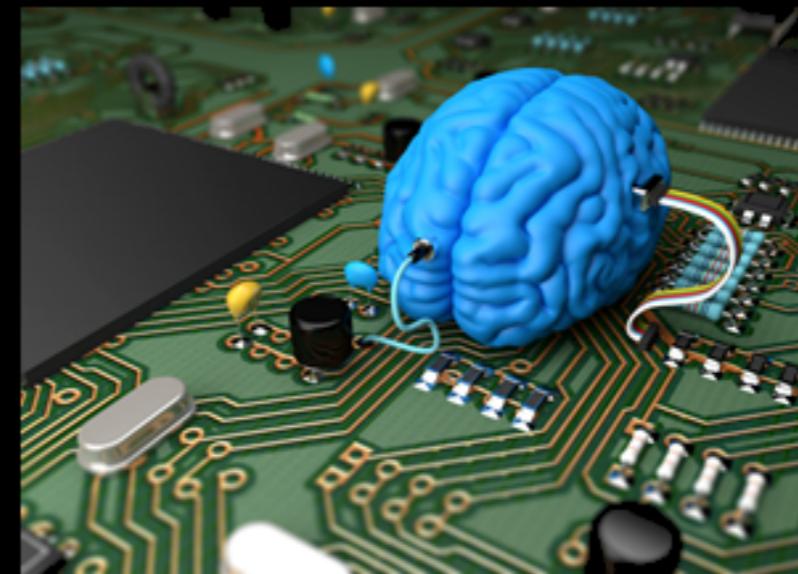
Convolutional Neural Networks



Deep Learning



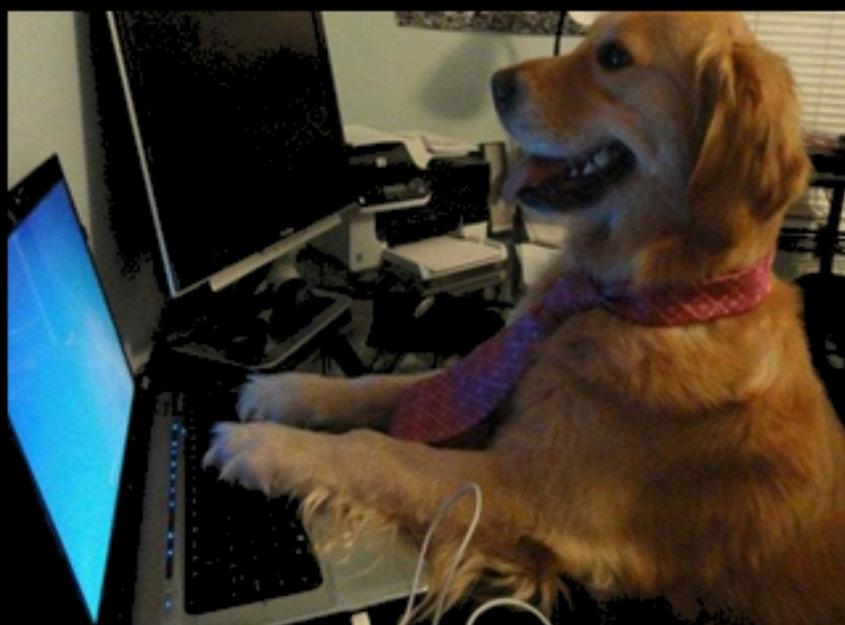
What society thinks I do



What my friends think I do



What other computer
scientists think I do



What mathematicians think I do

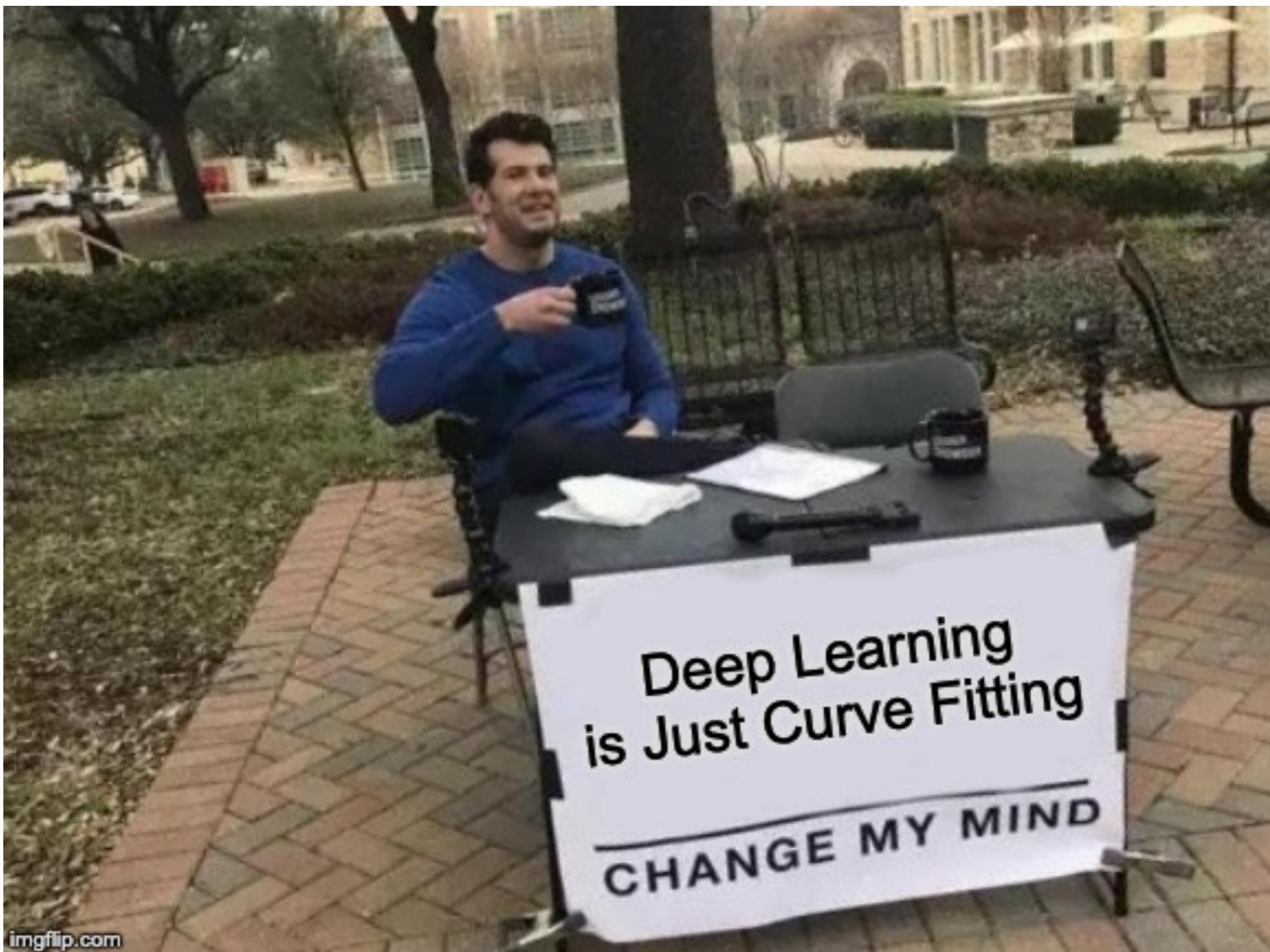


What I think I do

```
In [1]:  
import keras  
Using TensorFlow backend.
```

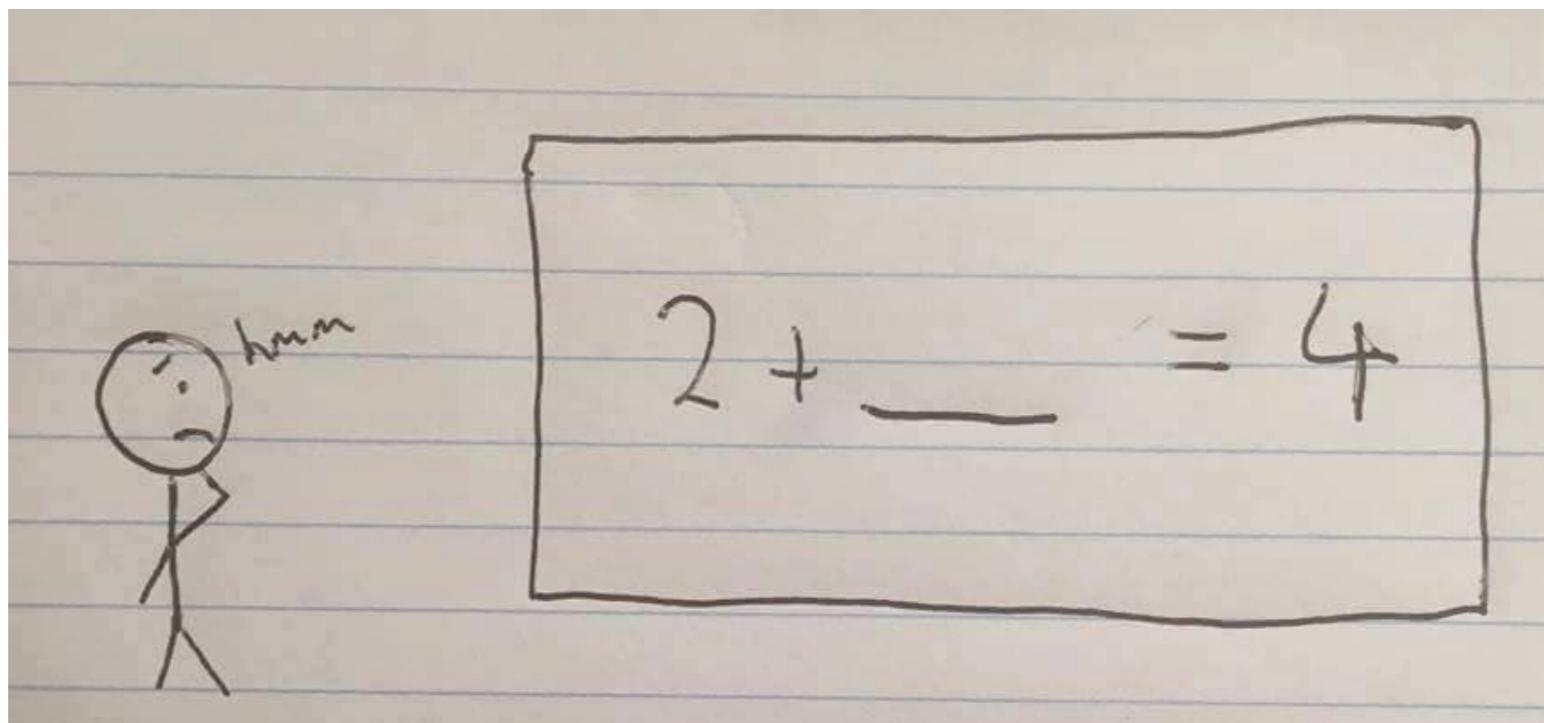
What I actually do

Curve Fitting?

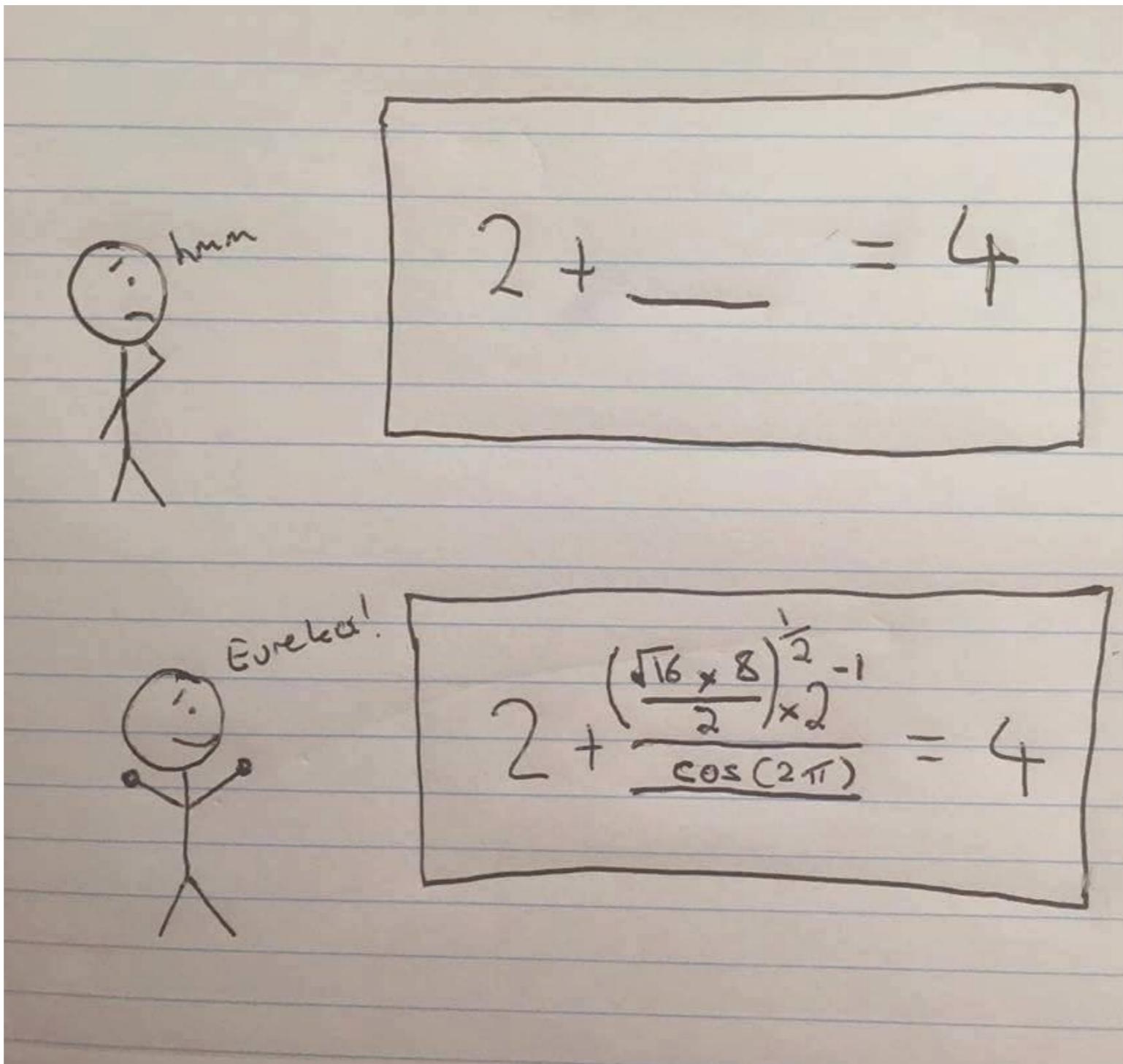


imgflip.com

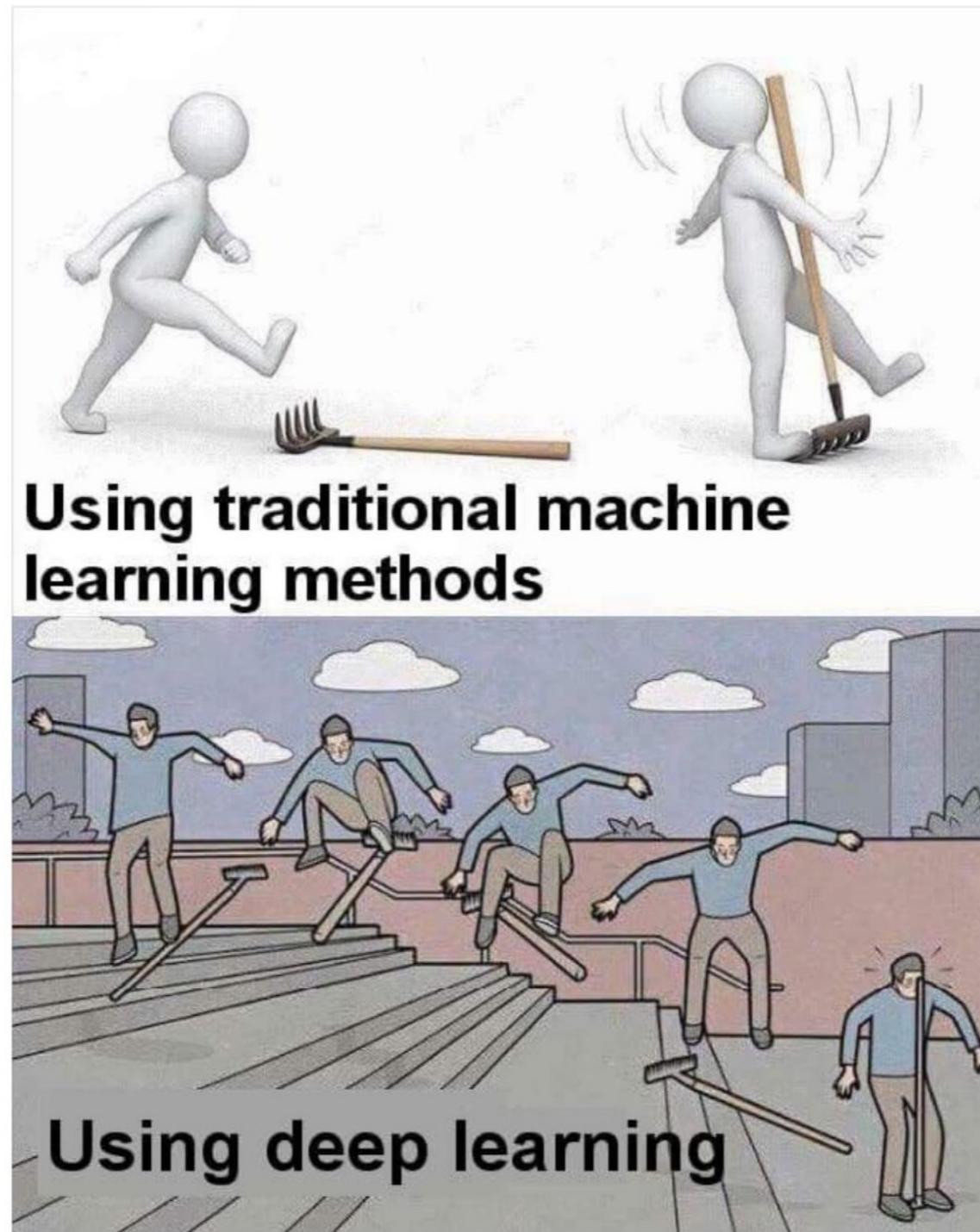
Interpretability?



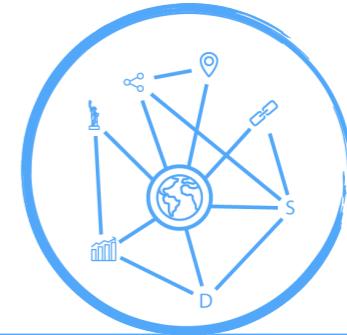
Interpretability?



“Deep” learning



Events



www.data4sci.com/newsletter

Time Series for Everyone

Apr 08, 2020 - 5am-9am (PST)

Applied Probability Theory for Everyone

Apr 29, 2020 - 5am-9am (PST)



Natural Language Processing (NLP) from Scratch

<http://bit.ly/LiveLessonNLP> - On Demand