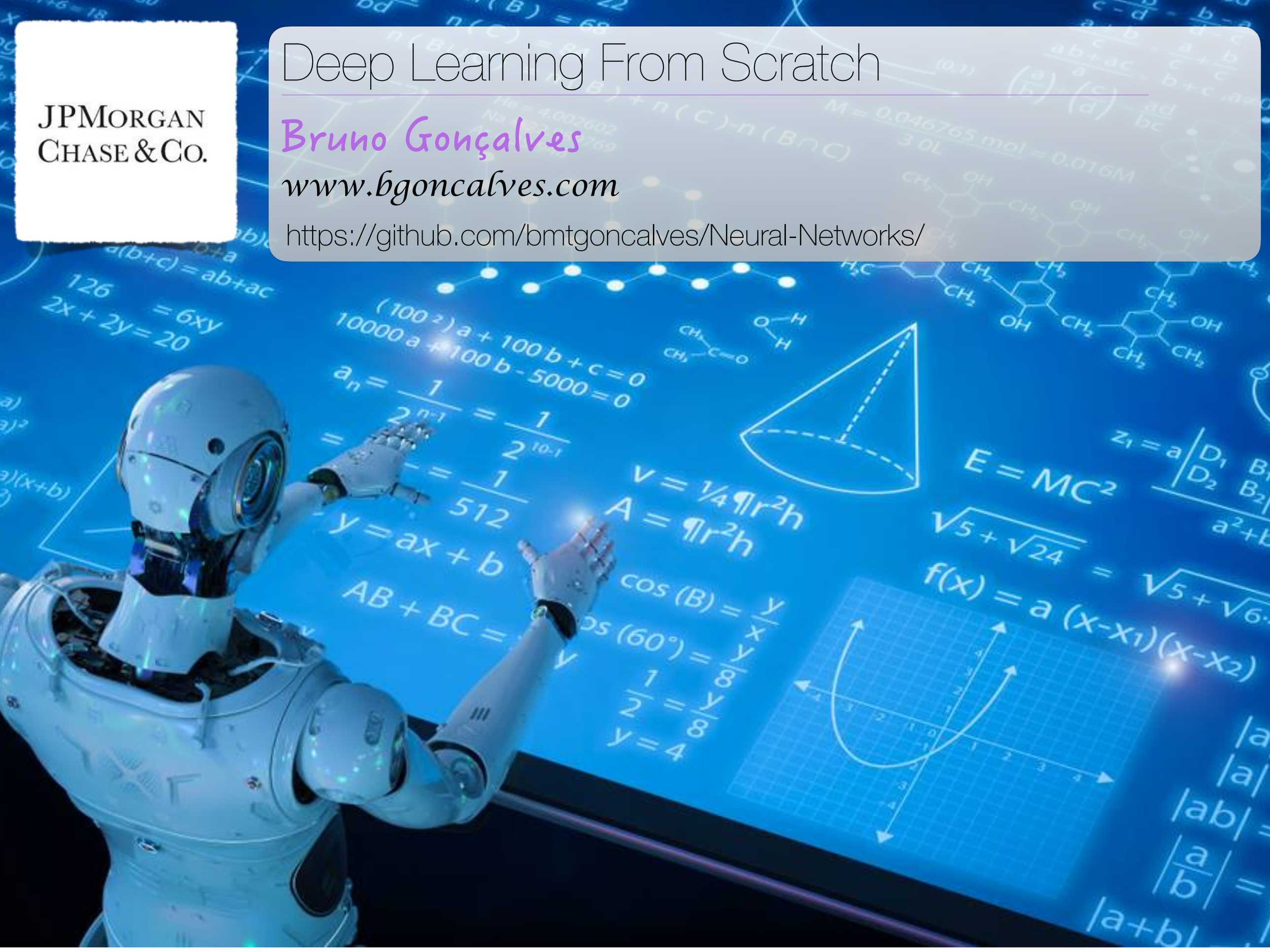
JPMorgan Chase & Co.

Deep Learning From Scratch

Bruno Gonçalves

www.bgoncalves.com

https://github.com/bmtgoncalves/Neural-Networks/

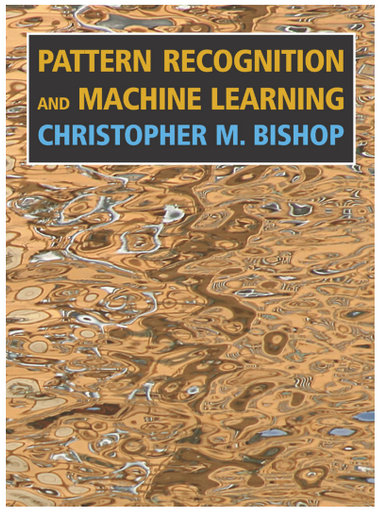


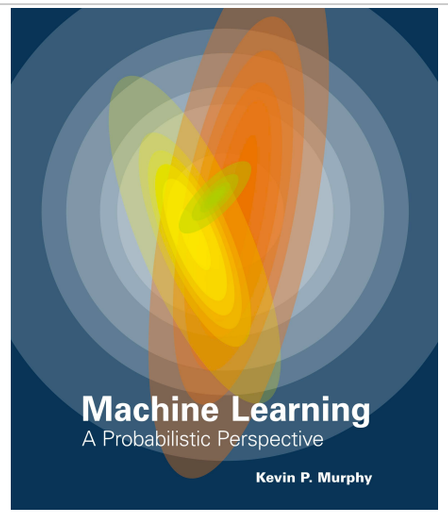
Disclaimer

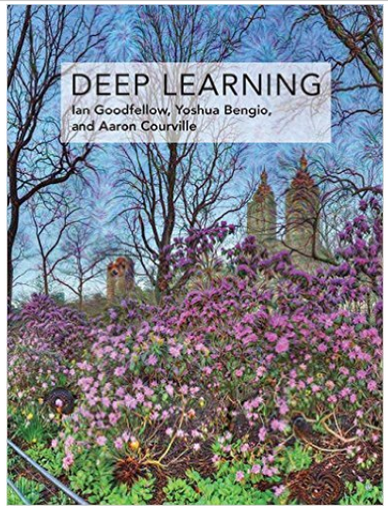
The views and opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of my employer. The examples provided with this tutorial were chosen for their didactic value and are not mean to be representative of my day to day work.

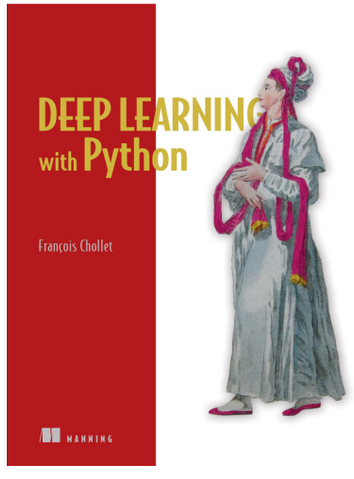
References

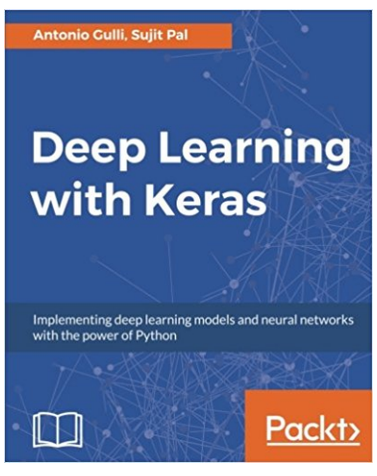
https://github.com/bmtgoncalves/Neural-Networks/

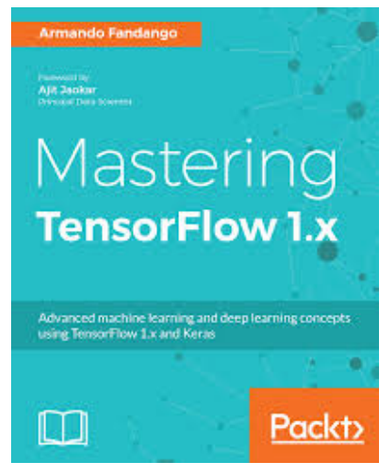


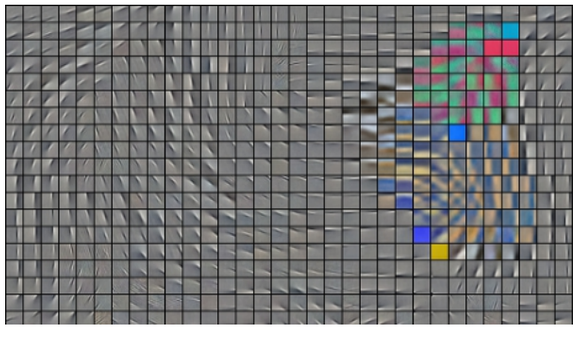












Neural Networks for Machine Learning

Geoff Hinton

Machine Learning

Andrew Ng

Requirements

https://github.com/bmtgoncalves/Neural-Networks/

AI Key Milestone Events

1950 — Alan Turing Proposes the Turing Test

1950 — Isaac Asimov proposes the Three Laws of Robotics

1951 — First AI based Program was written

1955 — First self learning game playing program is written

1959 — MIT AI Lab is set up

1961 — First Robot is inducted inot GM's assembly production line

1963 — First Machine Learning program is written

1964 — First demonstration of an AI program which understand Natural Language

1965 — First AI based Chatbot (ELIZA) was created

1969 — Stanford Research Institute (SRI) demonstrates the first locomotive and intelligent robot (Shakey)

1974 — First autonomous vehicle is created at the Stanford AI Lab

1974 — First rule based AI expert system for medical diagnostics

1980 — LISP based machines are developed and marketed

1989 — CMU creates first Autonomous Vehicle using a Neural Network

1997 — IBMs Deep Blue beats Gary Kasparov at Chess

1999 — Sony introduces the first artificially intelligent domestic robot, AIBO

1999 — Fisrt Emotional AI machines demonstrated at MIT AI Lab

2004 — DARPA introduces the first challenge for Autonomous Vehicles

2005 — AI based recommendation engines

2009 — Google builds Self Driving Car

2010 — Narrative Science's AI demonstrates ability to write reports

2011 — IBM Watson beats Jeopardy chaimpions

2011 — Personal Assistants like Siri, Google Now and Cortana become mainstream

2015 — Elon Musk and others announce a $1B not for profit open source intiative, OPEN AI to develop friendly AI

2016 — NVIDIA announces supercomputer for Deep Learning and AI

2016 — Google's Deepmind AlphaGo defeats Go's champions

No. of transistors per sq. inch

Machine Learning

3 Types of Machine Learning

https://github.com/bmtgoncalves/Neural-Networks/

- Unsupervised Learning
  - Autonomously learn an good representation of the dataset
  - Find clusters in input
- Supervised Learning
  - Predict output given input
  - Training set of known inputs and outputs is provided
- Reinforcement Learning
  - Learn sequence of actions to maximize payoff
  - Discount factor for delayed rewards

3 Types of Machine Learning

https://github.com/bmtgoncalves/Neural-Networks/

- Unsupervised Learning
  - Autonomously learn an good representation of the dataset
  - Find clusters in input
- Supervised Learning
  - Predict output given input
  - Training set of known inputs and outputs is provided
- ~~Reinforcement Learning~~
  - Learn sequence of actions to maximize payoff
  - Discount factor for delayed rewards

Optimization Problem

https://github.com/bmtgoncalves/Neural-Networks/

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

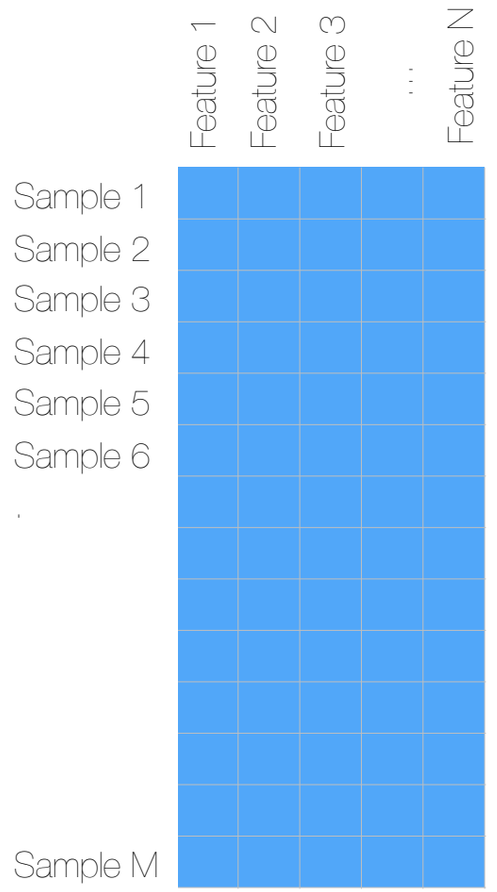
- Optimization Problems have 3 distinct pieces:

  - The constraints — Problem Representation
  - The function to optimize — Prediction Error
  - The optimization algorithm. — Gradient Descent

Supervised Learning

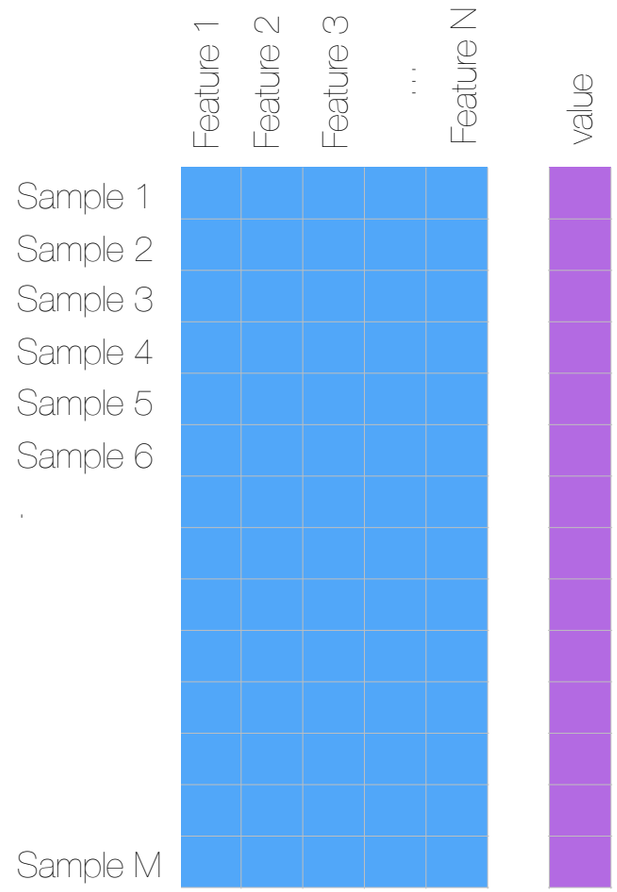
Supervised Learning - Classification

- Two fundamental types of problems:
  - Regression (continuous output value)
  - Classification (discrete output value)
- Dataset formatted as an MxN matrix of M samples and N features
- Each sample corresponds to a specific value of the target variable
- The goal of regression is to predict or approximate the value of a function at previously unseen points



Supervised Learning - Classification

- Two fundamental types of problems:
  - Regression (continuous output value)
  - Classification (discrete output value)
- Dataset formatted as an MxN matrix of M samples and N features
- Each sample corresponds to a specific value of the target variable
- The goal of regression is to predict or approximate the value of a function at previously unseen points
  - Linear Regression
  - Neural Networks



Linear Regression

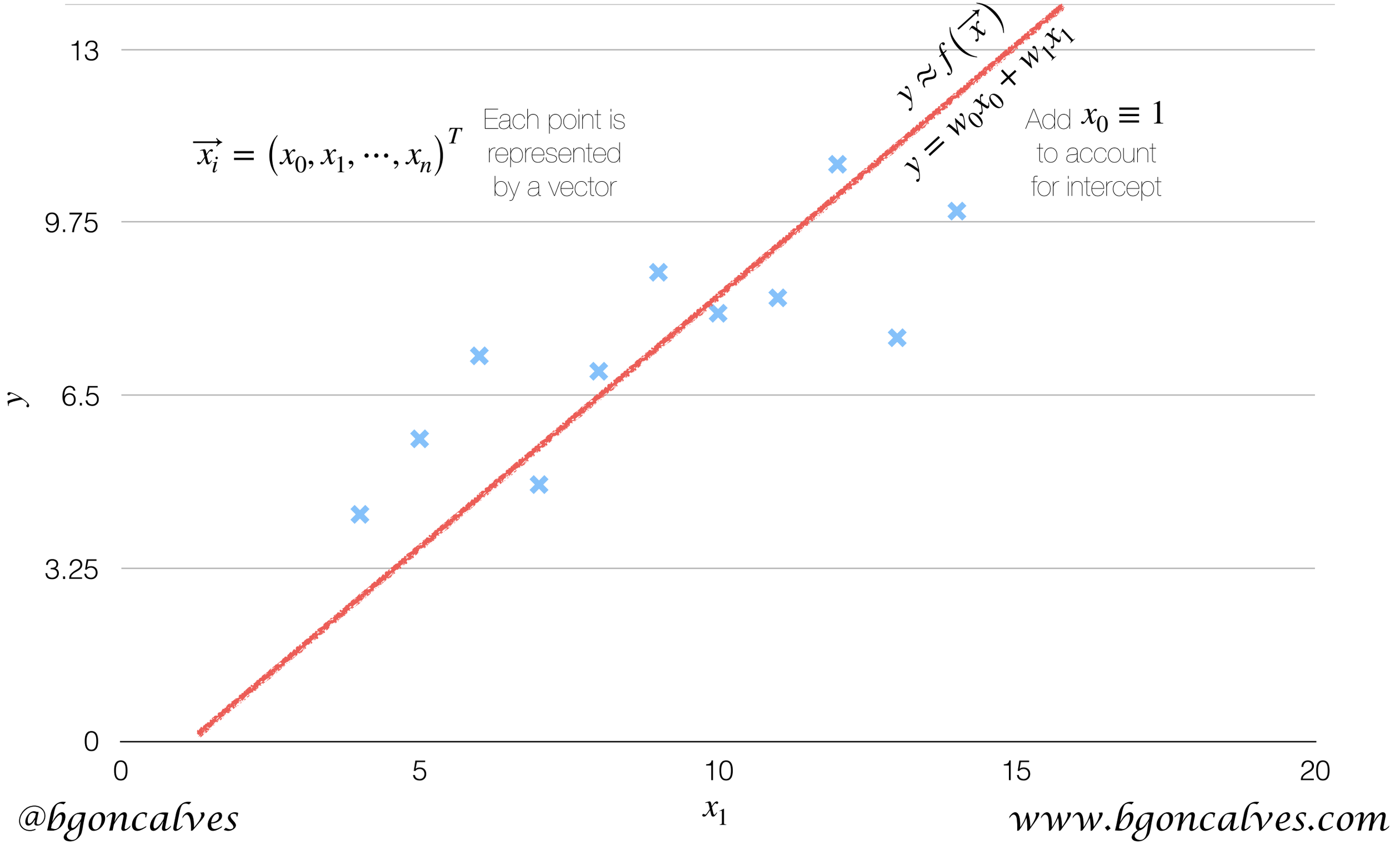
$$\vec{x_i} = (x_0, x_1, \cdots, x_n)^T$$

Each point is represented by a vector

$$y \approx f(\vec{x})$$

$$y = w_0 x_0 + w_1 x_1$$

Add $x_0 \equiv 1$ to account for intercept



Linear Regression

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

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

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

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

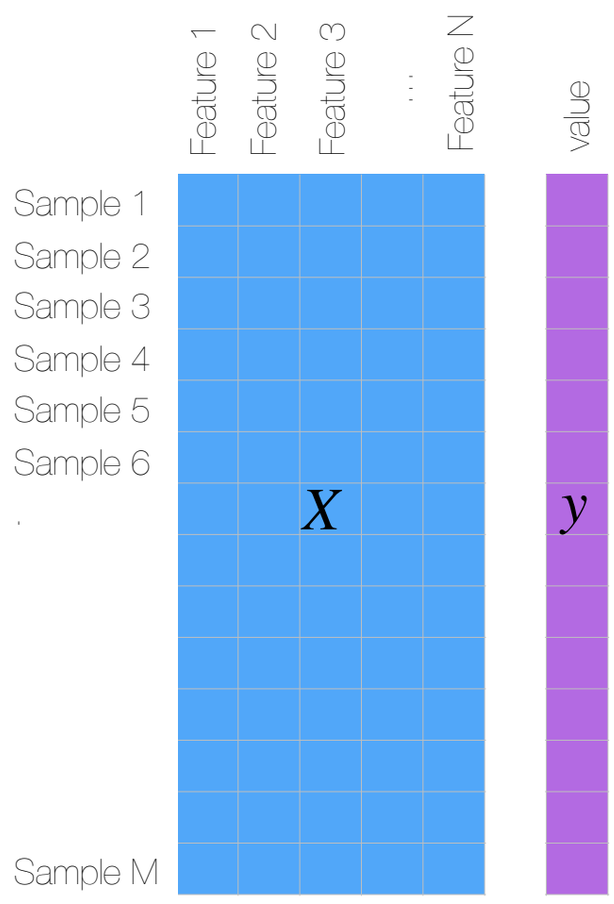
and it imposes a Constraint on the solutions that can be found.

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

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

or, vectorially:

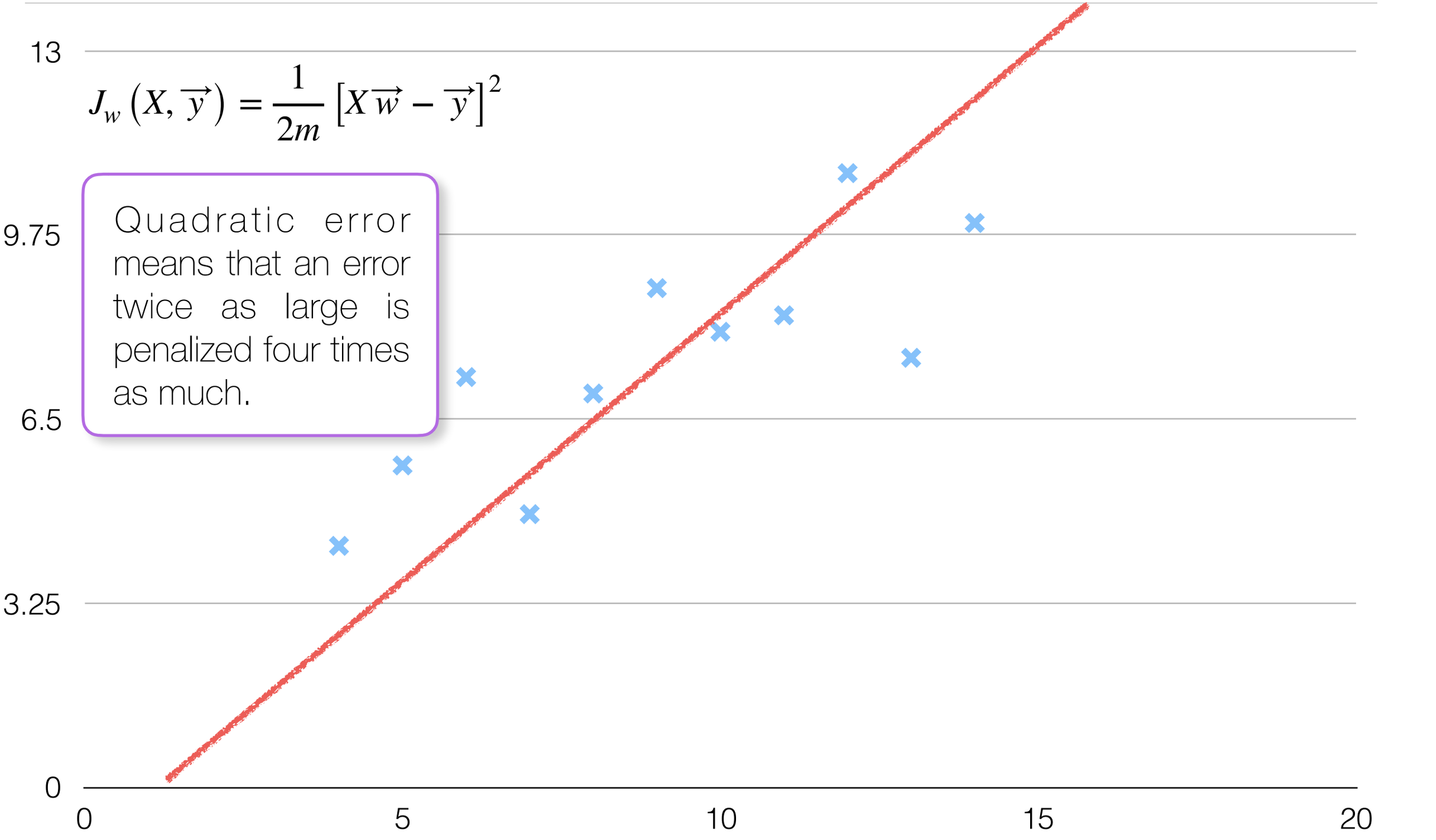
$$J_w\left(X, \vec{y}\right) = \frac{1}{2m}\left[X\vec{w} - \vec{y}\right]^2$$



Geometric Interpretation

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

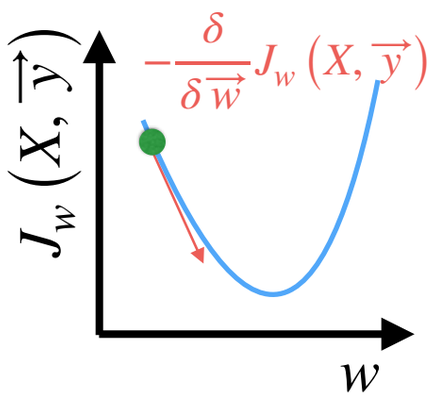
Quadratic error means that an error twice as large is penalized four times as much.

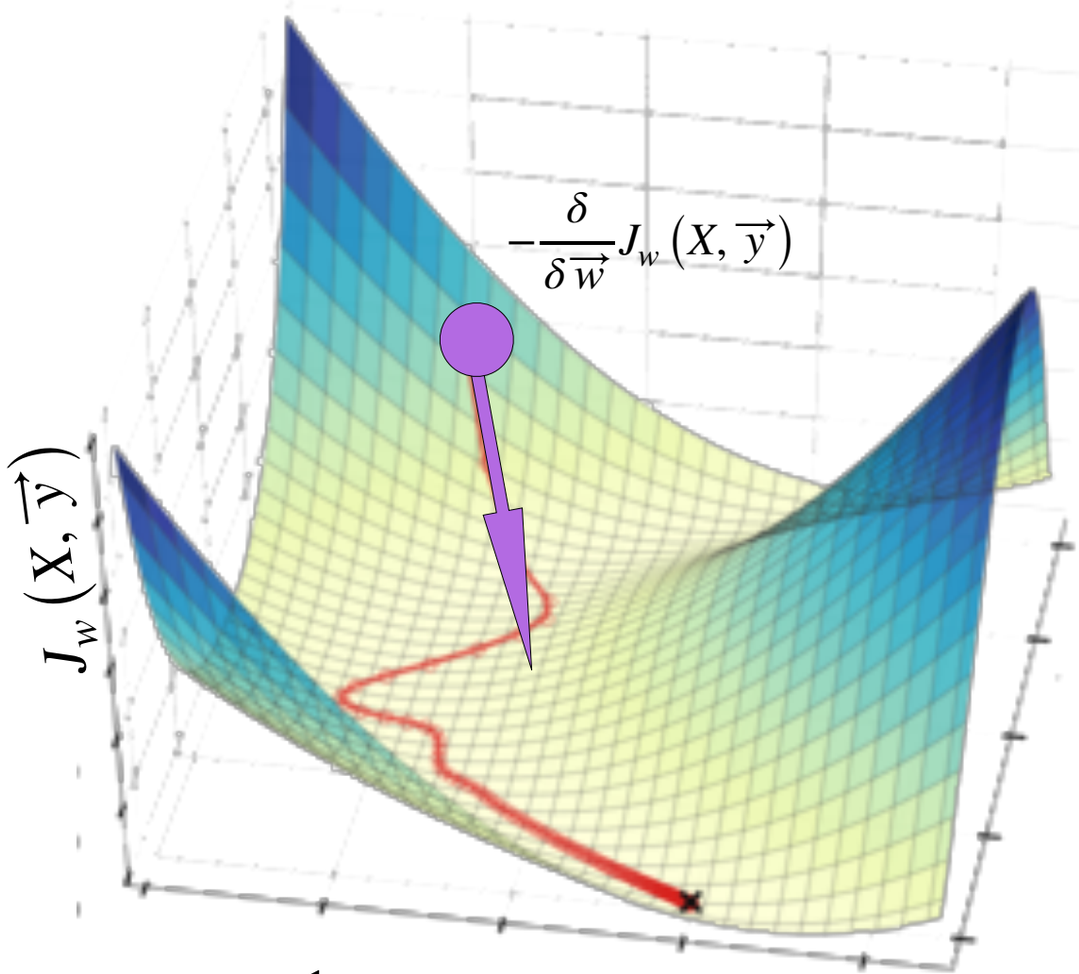


Gradient Descent

- Goal: Find the minimum of $J_w\left(X, \vec{y}\right)$ by varying the components of $\vec{w}$

- Intuition: Follow the slope of the error function until convergence





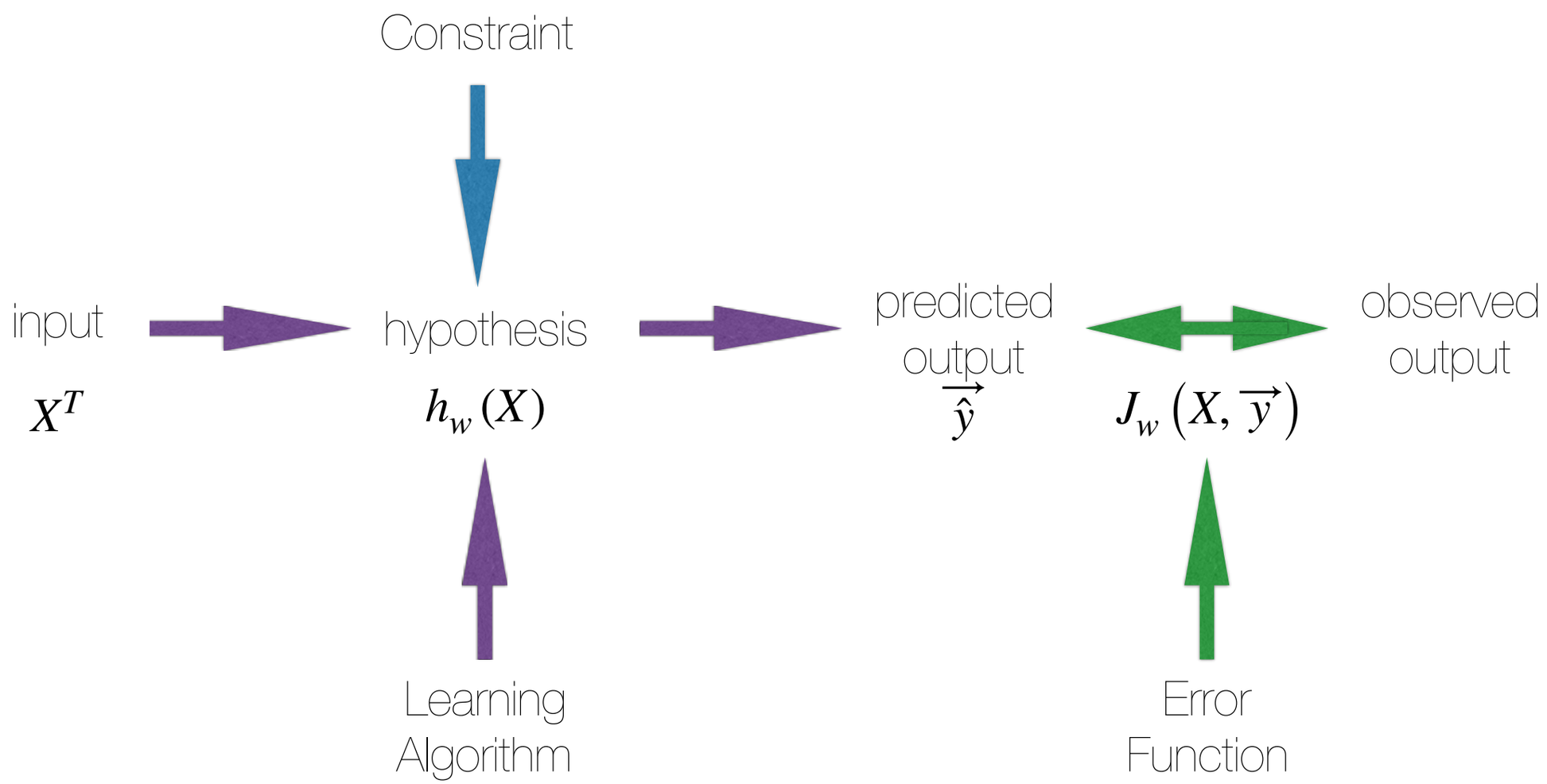
- Algorithm:

  - Guess $\vec{w}^{(0)}$ (initial values of the parameters)
  - Update until "convergence":

step size

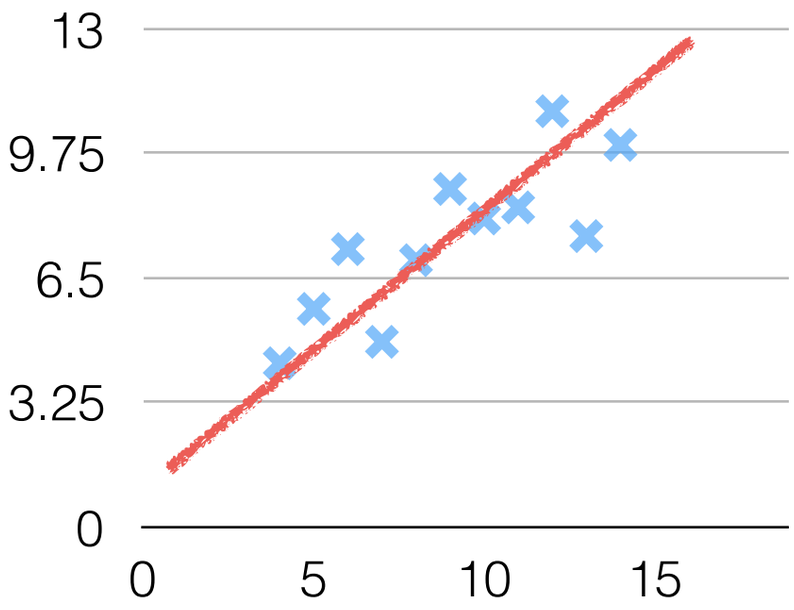
$$w_j = w_j - \alpha \frac{\delta}{\delta w_j} J_w\left(X, \vec{y}\right) \qquad \frac{\delta}{\delta w_j} J_w\left(X, \vec{y}\right) = \frac{1}{m} X^T \cdot \left(h_w(X) - \vec{y}\right)$$

Learning Procedure



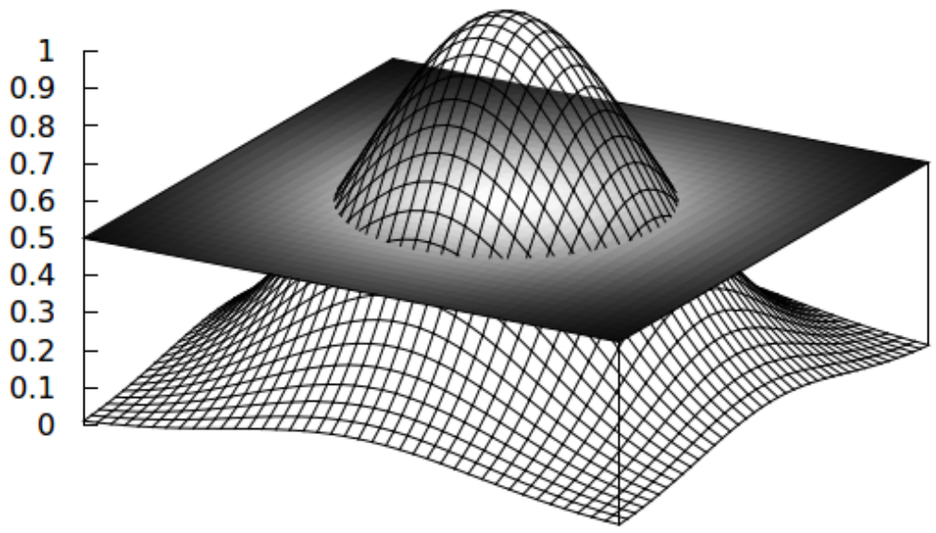
Geometric Interpretation

2D



$$y = w_0 + w_1 x_1$$

3D



$$y = w_0 + w_1 x_1 + w_2 x_2$$

nD

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

Add $x_0 \equiv 1$ to account for intercept

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

Linear Regression

```python
import sys
import numpy as np

data = np.matrix(np.loadtxt("data/Anscombe1.dat"))

X = data[:, 0].reshape(-1, 1)
y = data[:, 1].reshape(-1, 1)

M, N = X.shape
X = np.concatenate((np.ones((M, 1)), X), axis=1) #Add x0

alpha = 0.01
epsilon = 0.12

weights = 2*np.random.rand(N+1, 1)*epsilon - epsilon
count = 0

oldJ = 0
err = 1

while err > 1e-6:
    Hs = np.dot(X, weights)
    deltas = alpha/M*np.dot(X.T, (Hs-y))

    count += 1
    weights -= deltas

    J = np.sum(np.power(Hs-y, 2.))/(2*M)
    err = np.abs(oldJ-J)
    oldJ = J

    print(count, J, err, weights.flatten())
```

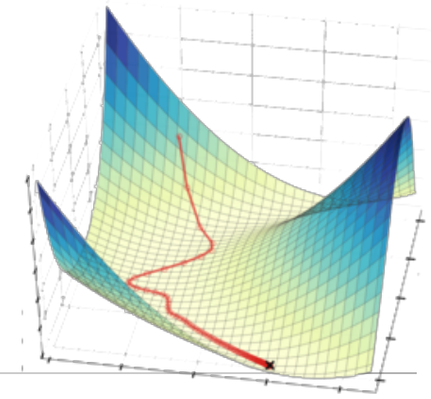
$$h_w(X) = X\vec{w}$$

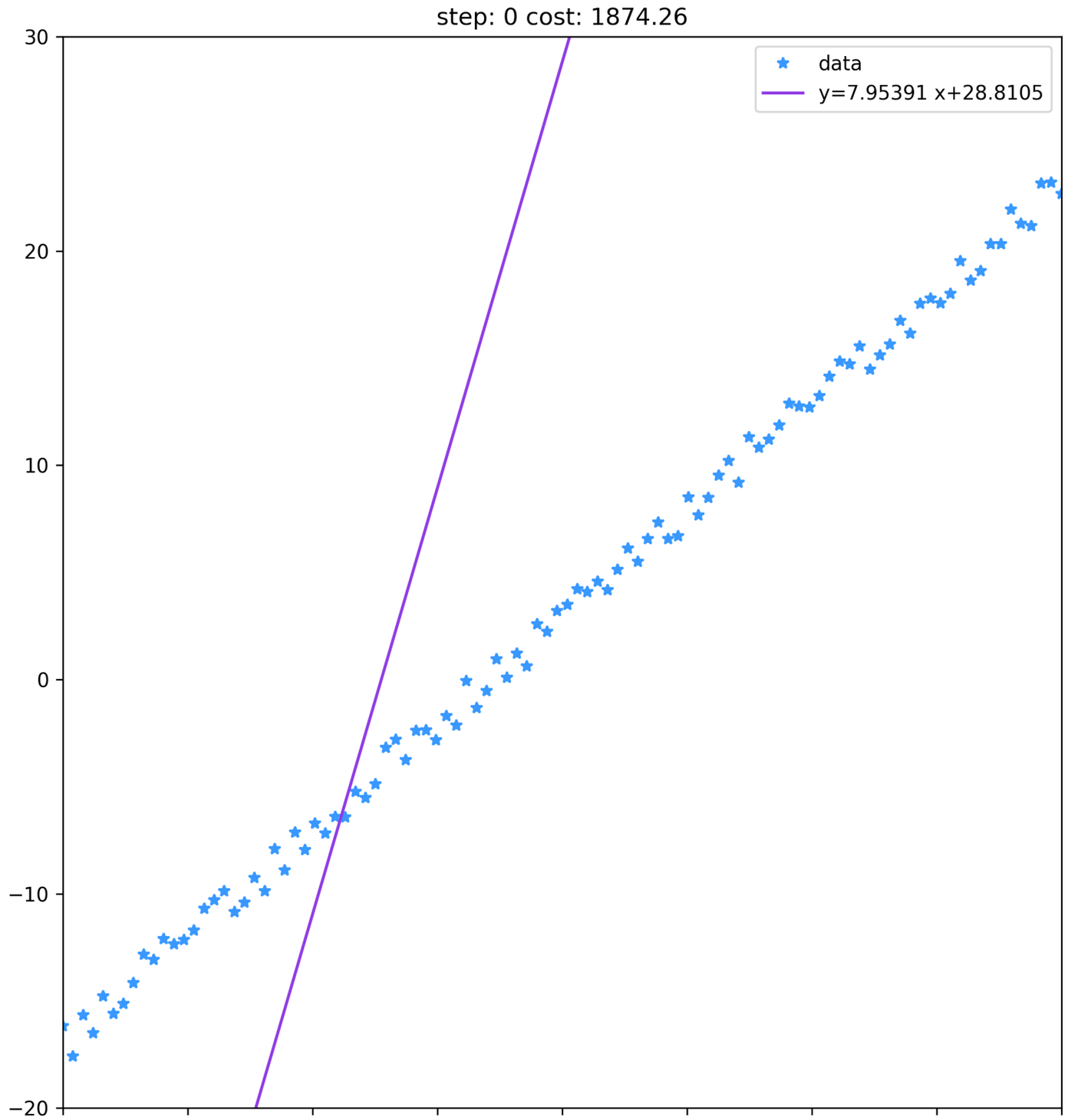
$$\frac{\delta}{\delta w_j} J_w\left(X, \vec{y}\right) = \frac{1}{m} X^T \cdot \left(h_w(X) - \vec{y}\right)$$

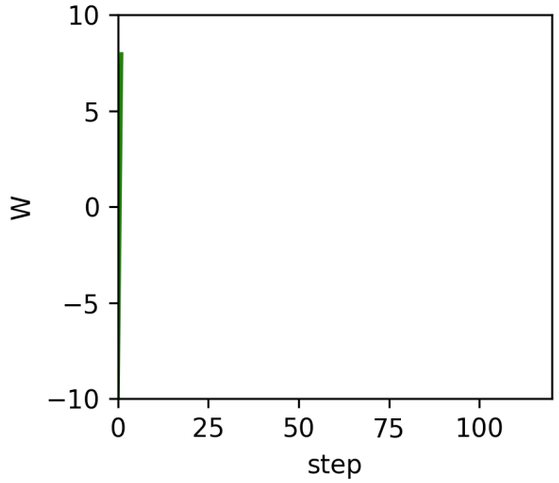
$$J_w\left(X, \vec{y}\right) = \frac{1}{2m}\left[X\vec{w} - \vec{y}\right]^2$$

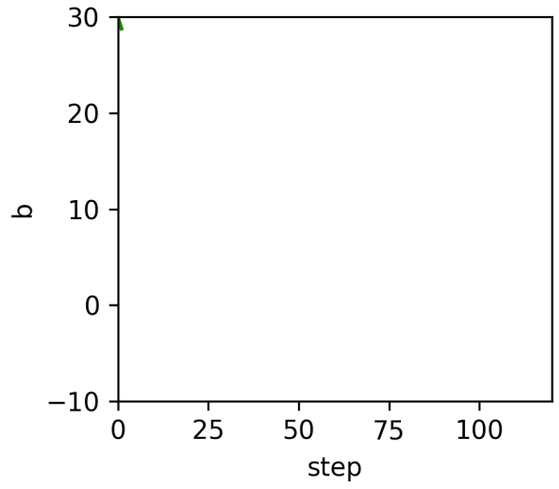
linear.py

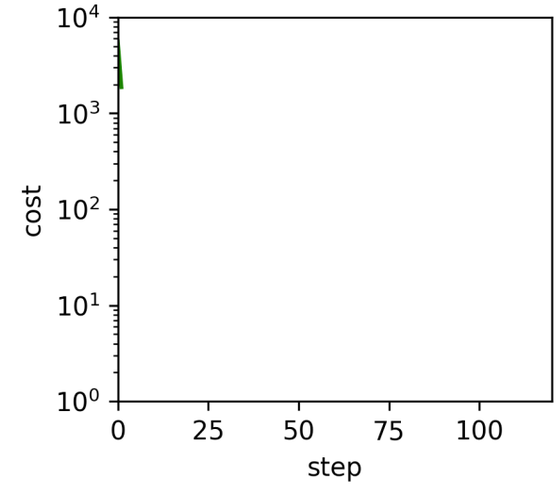
Linear Regression





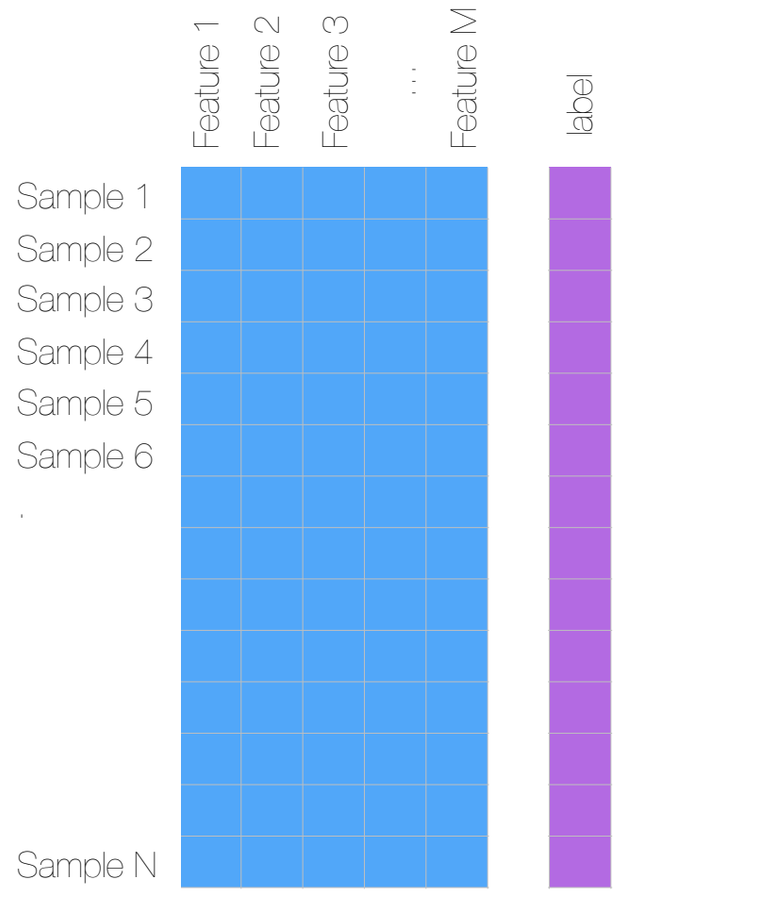






Supervised Learning - Classification

- Two fundamental types of problems:
  - Regression (continuous output value)
  - Classification (discrete output value)
- Dataset formatted as an NxM matrix of N samples and M features
- Each sample belongs to a specific class or has a specific label.
- The goal of classification is to predict to which class a previously unseen sample belongs to by learning defining regularities of each class
  - Logistic Regression
  - Neural Networks



Logistic Regression (Classification)

- Not actually regression, but rather Classification

- Predict the probability of instance belonging to the given class:

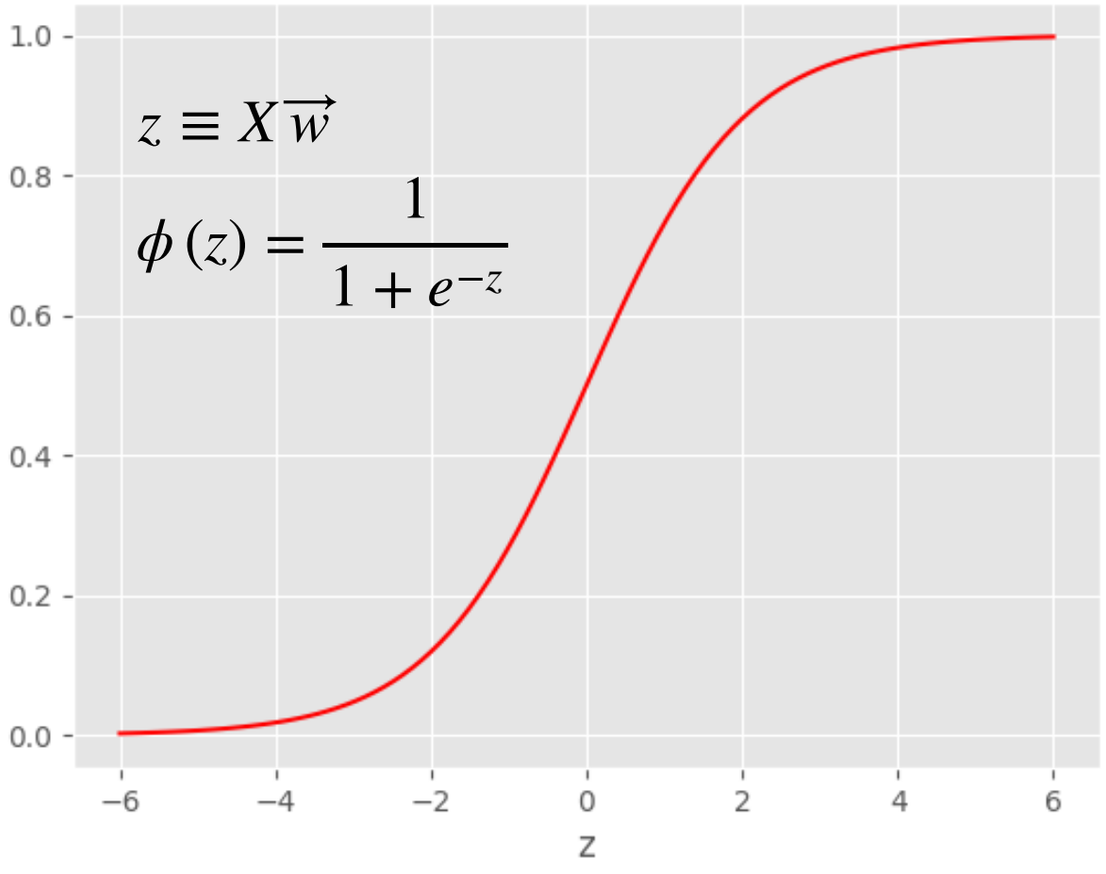
$$h_w(X) \in [0,1]$$

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

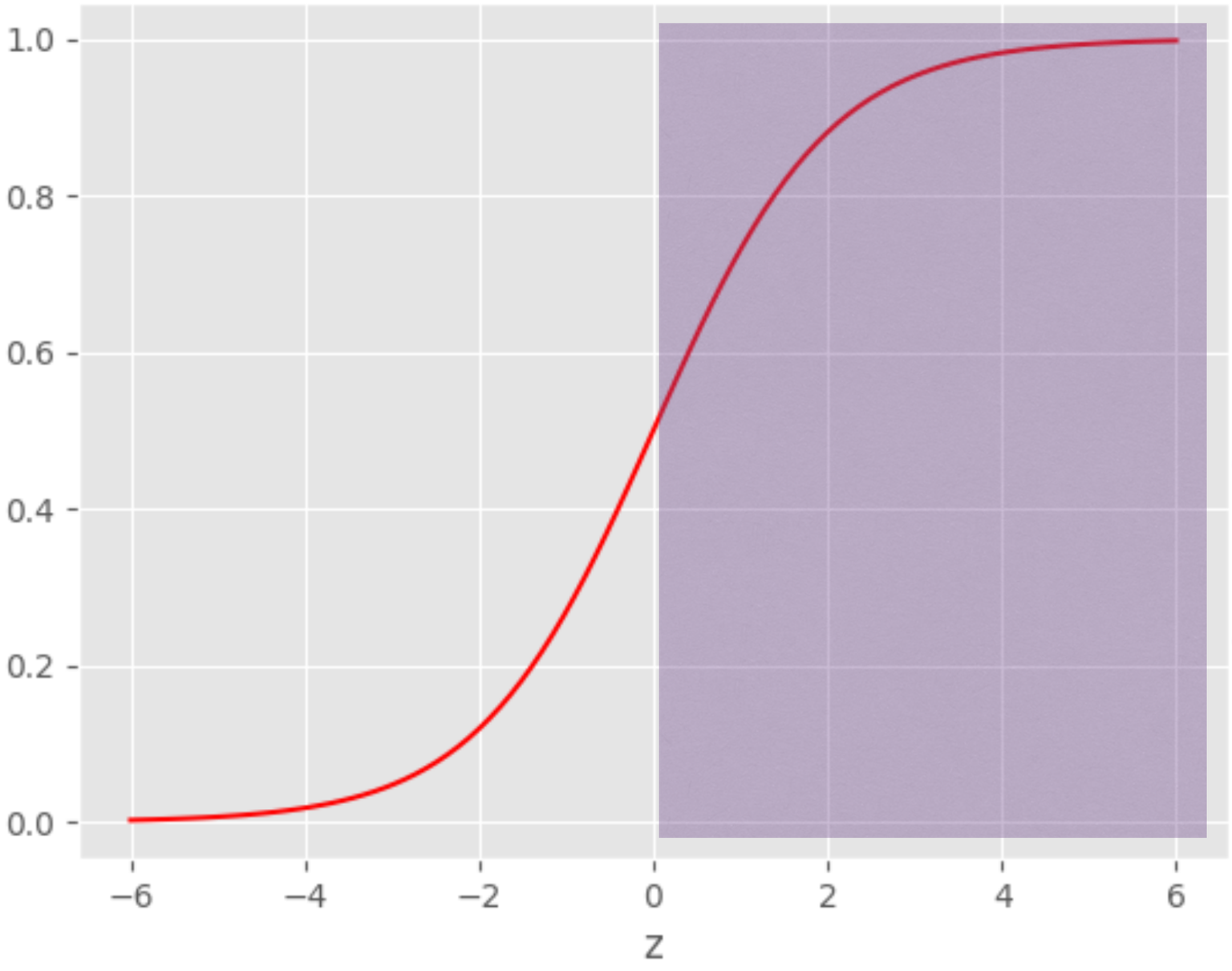
$$h_w(X) = \phi\left(X\vec{w}\right)$$

1 - part of the class
0 - otherwise

z encapsulates all the parameters and input values



Geometric Interpretation



$$\phi(z) \geq \frac{1}{2}$$

maximize the value of z for members of the class

Logistic Regression

- Error Function - Cross Entropy

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

measures the "distance" between two probability distributions

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

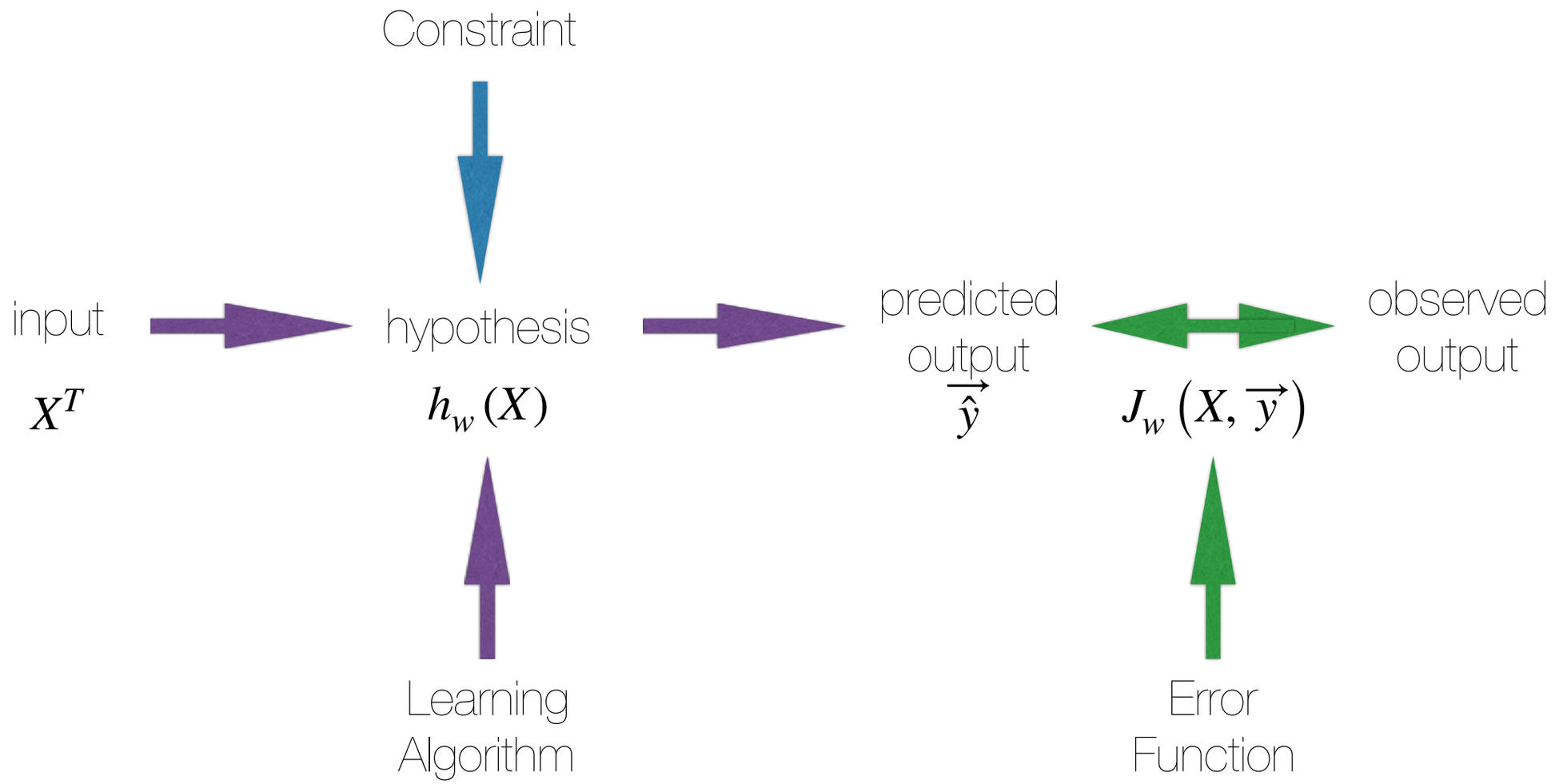
- Effectively treating the labels as probabilities (an instance with label=1 has Probability 1 of belonging to the class).

- Gradient - same as Logistic Regression

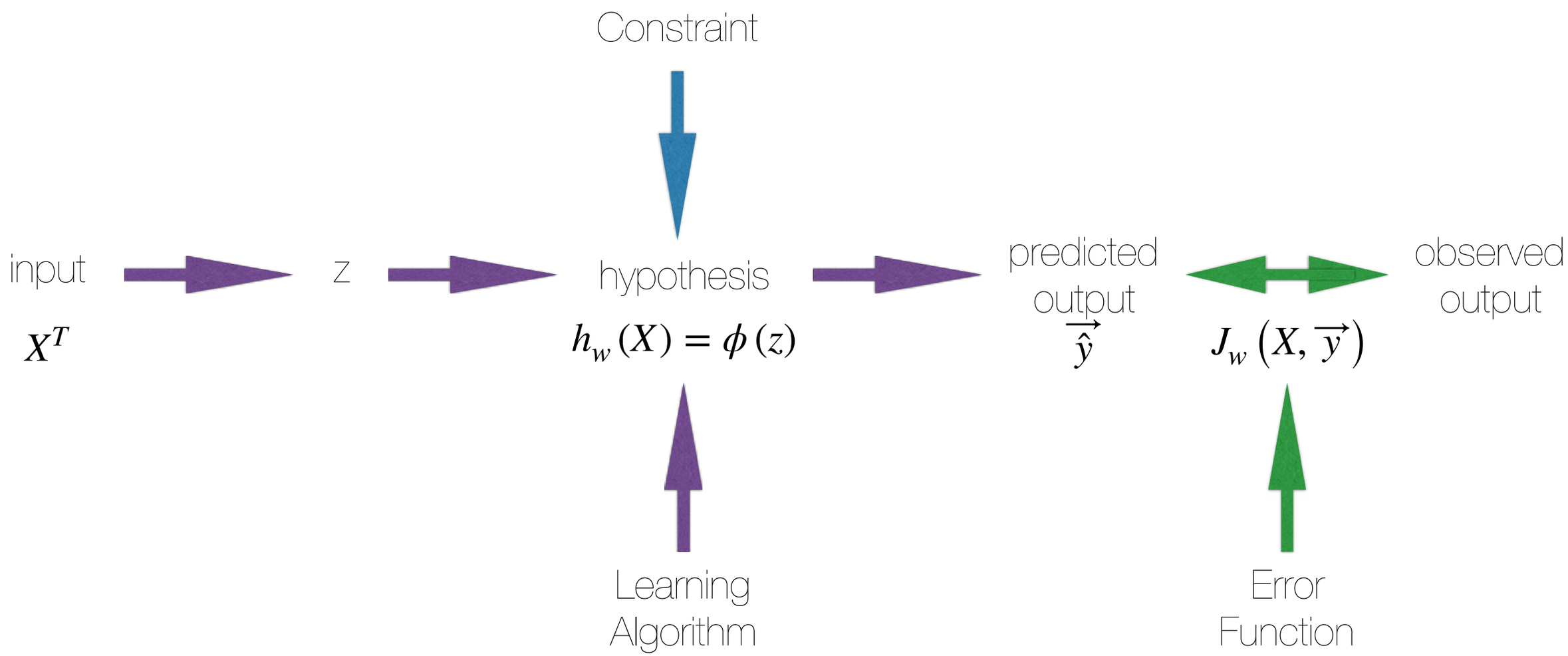
$$w_j = w_j - \alpha \frac{\delta}{\delta w_j} J_w\left(X, \vec{y}\right)$$

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

Learning Procedure



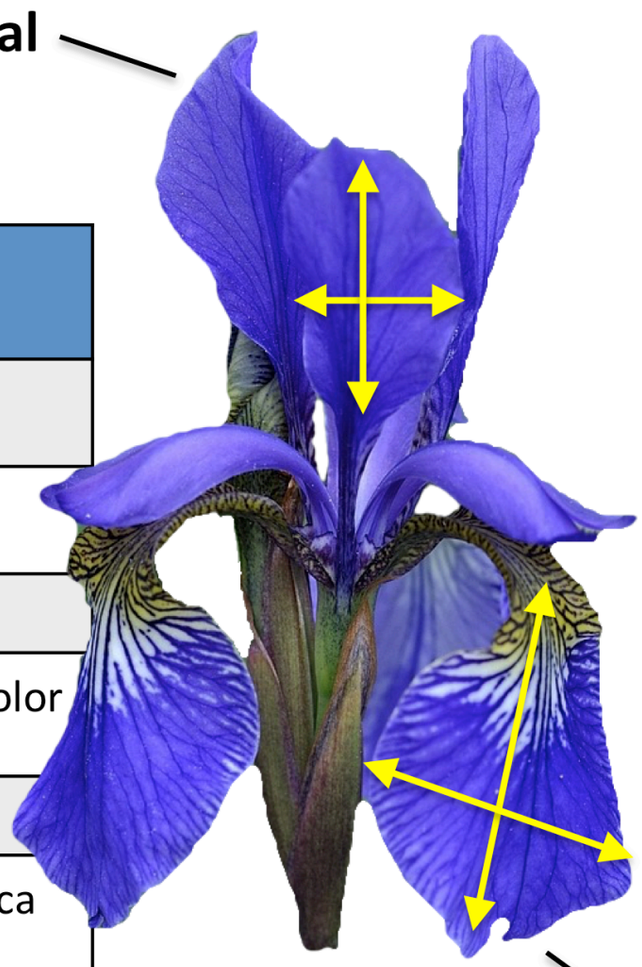
Learning Procedure



Iris dataset

Samples
(instances, observations)

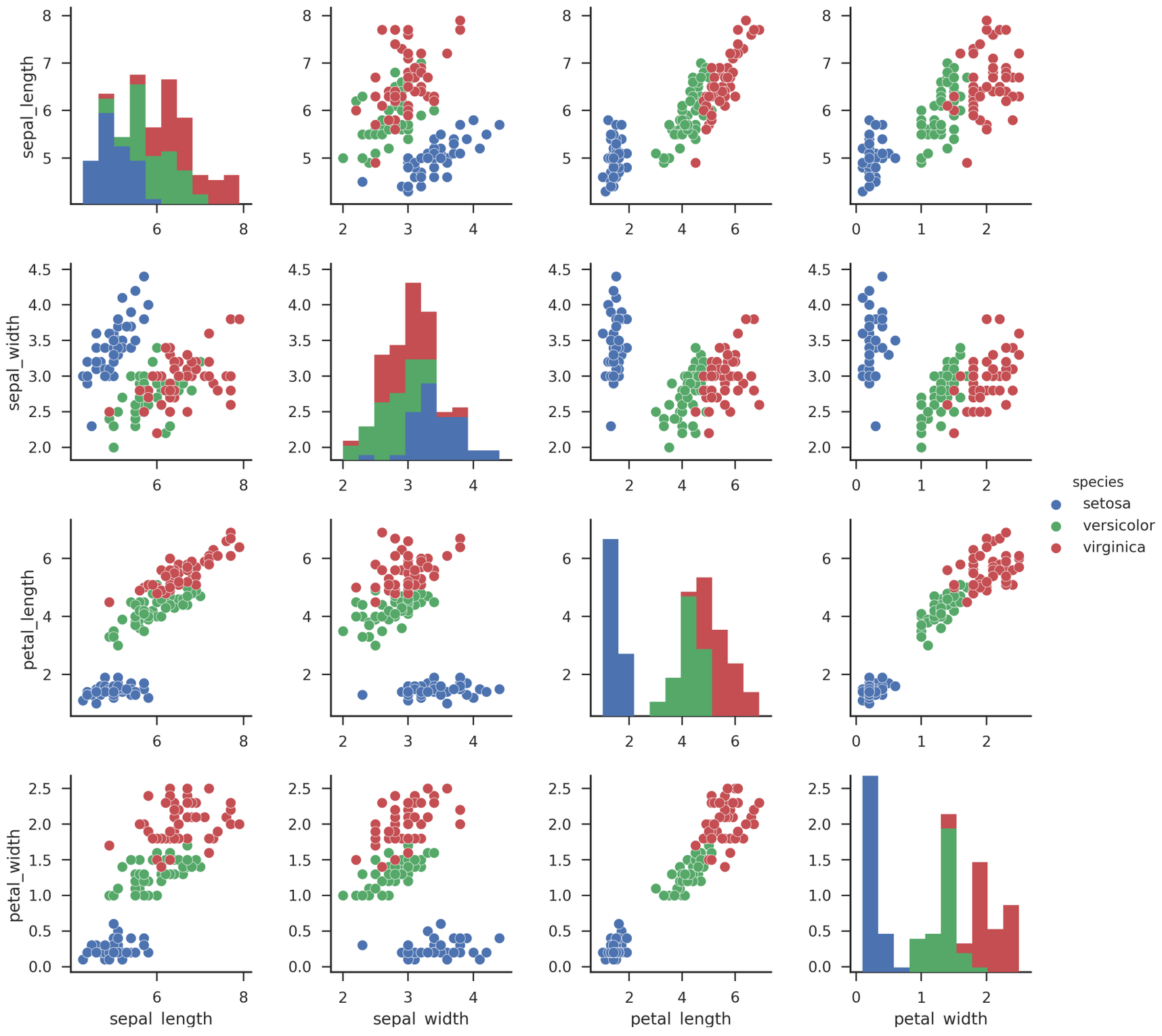
	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica



Features
(attributes, measurements, dimensions)

Class labels
(targets)

Iris dataset



Logis

```python
import sys
import numpy as np
import pandas as pd

data = pd.read_csv('data/iris.csv', delimiter=',', header=0)

data['y'] = 0
data.loc[data['species'] == 'setosa', 'y'] = 1

X = np.matrix(data[['sepal_length','sepal_width','petal_length','petal_width']])
y = np.matrix(data['y']).T

def logistic(z):
    return 1./(1+np.exp(-z))

alpha = 0.05
lamb = .1
M, N = X.shape

X = np.concatenate((np.ones((M, 1)), X), axis=1)
weights = 2*np.random.rand(N+1, 1)*epsilon - epsilon

epsilon = 0.12
count = 0
oldJ = 0
err = 1

while err > 1e-6:
    Zs = np.dot(X, weights)
    Hs = logistic(Zs)
    deltas = alpha/M*np.dot(X.T, (Hs-y))

    count += 1
    weights -= deltas

    J = -1/M*np.dot(y.T, np.log(Hs)) - np.dot(1-y.T, np.log(1-Hs))

    err = np.abs(oldJ-J)
    oldJ = J

    print(count, J, err, weights.flatten())
```

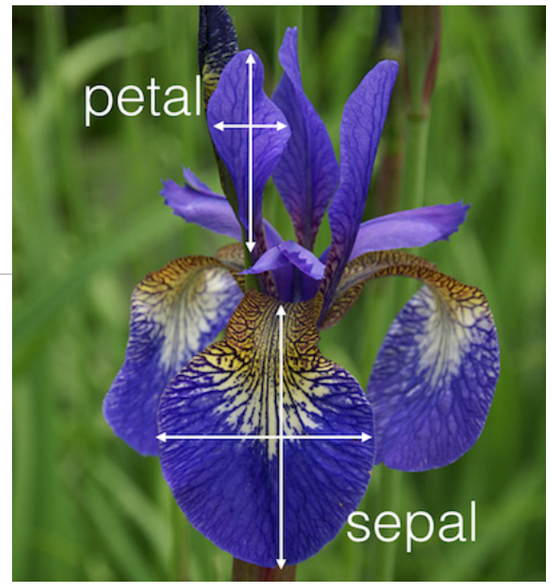
$$h_w(X) = \frac{1}{1 + e^{-X\vec{w}}}$$

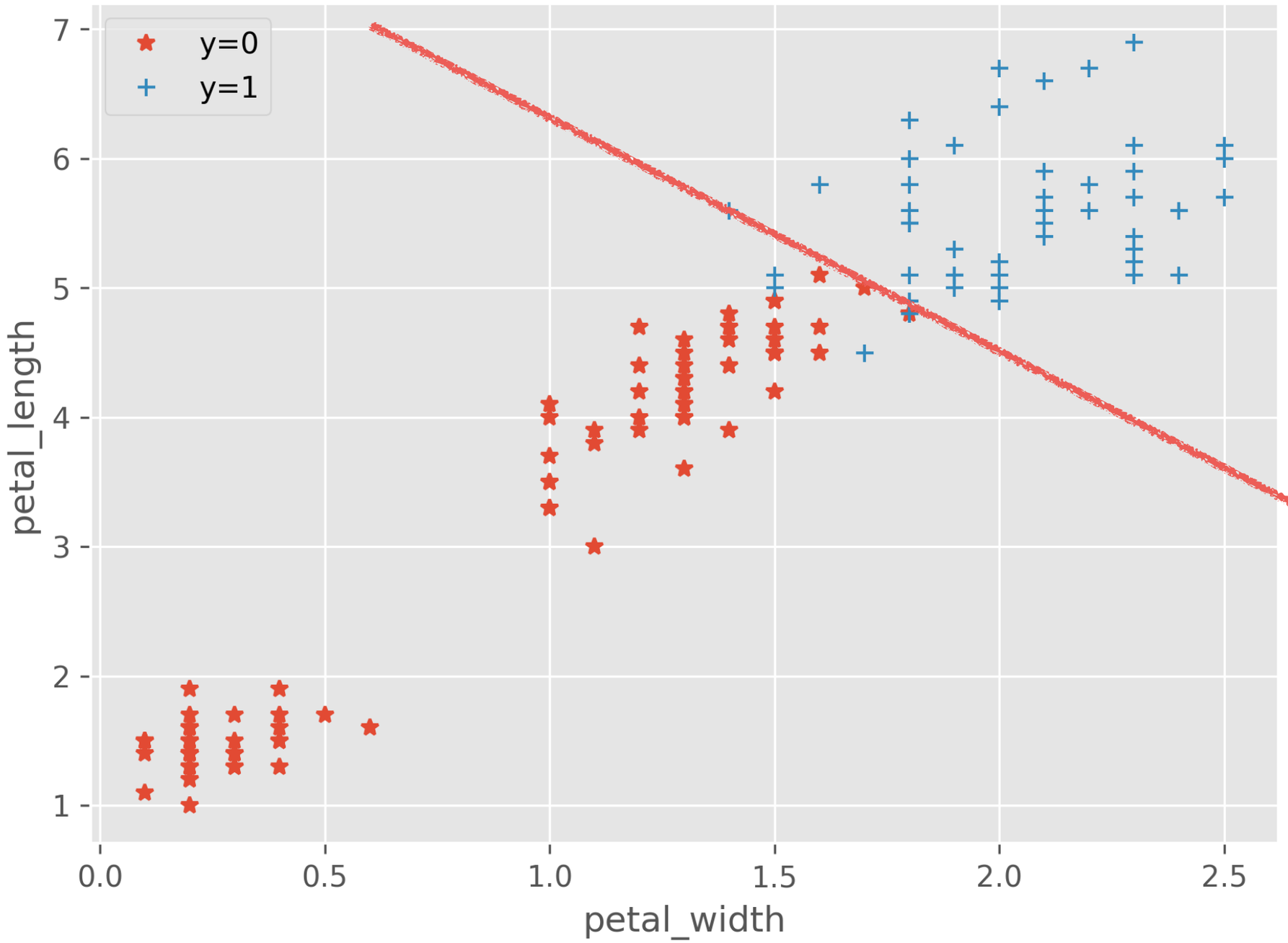
$$\frac{\delta}{\delta w_j} J_w\left(X, \vec{y}\right) = \frac{1}{m} X^T \cdot \left(h_w(X) - \vec{y}\right)$$

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

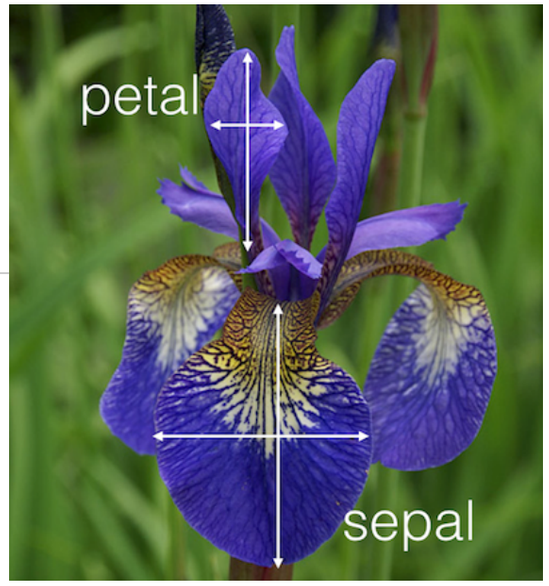
logistic.py

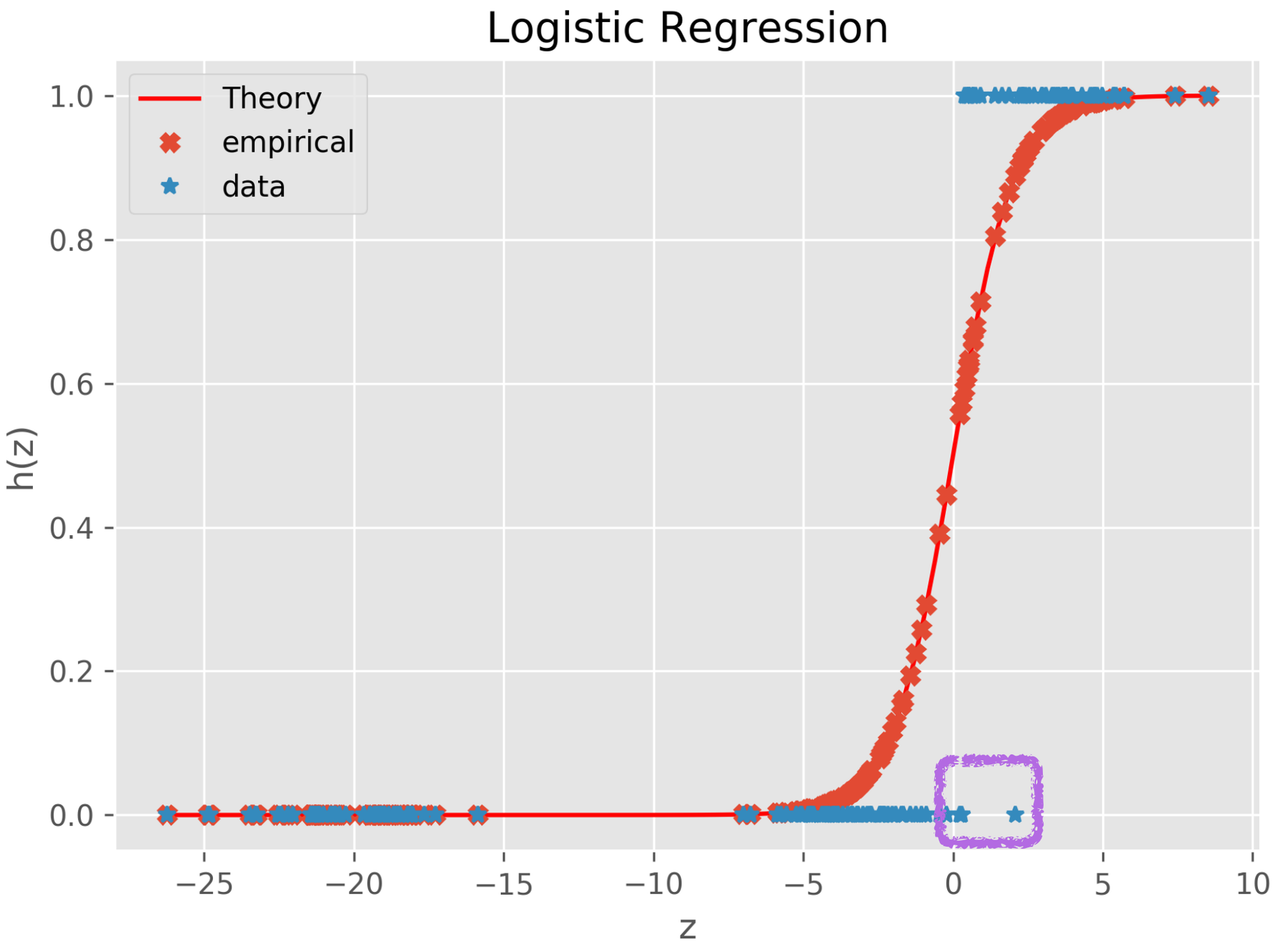
Logistic Regression





Logistic Regression





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

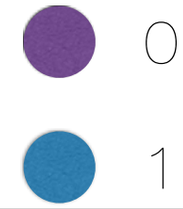
    - Overfitting

    - Hyperparameters

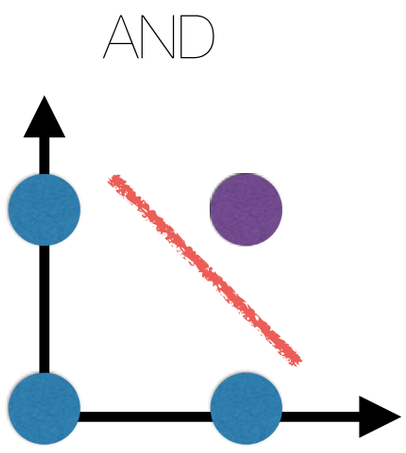
    - Bias, Variance tradeoffs

    - etc...

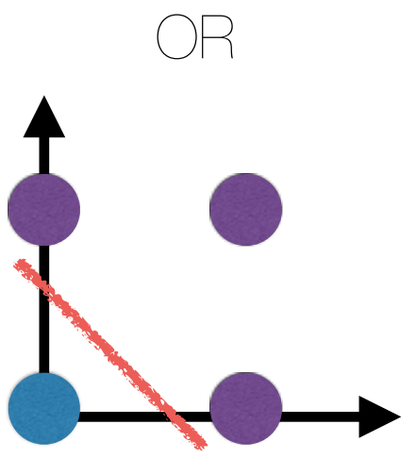
Linear boundaries



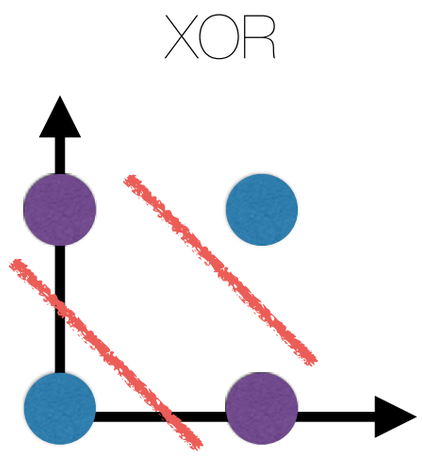
- Both Linear Regression and Logistic Regression rely on hyperplanes to separate the data points. Unfortunately, this is not always possible:



Possible



Possible



Impossible

Data Normalization

https://github.com/bmtgoncalves/Neural-Networks/

- 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 suitable way

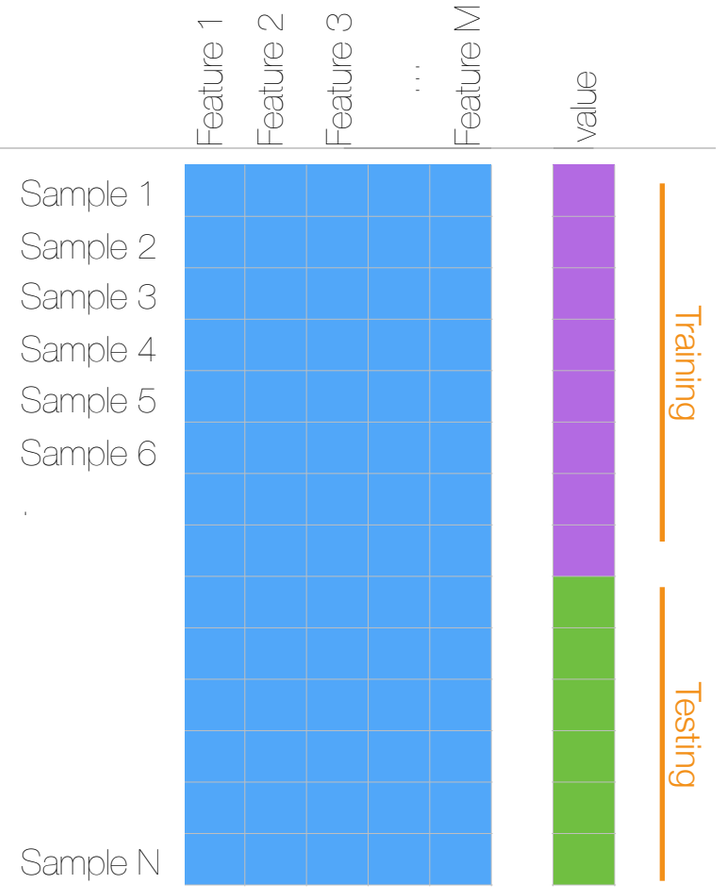
Data Normalization

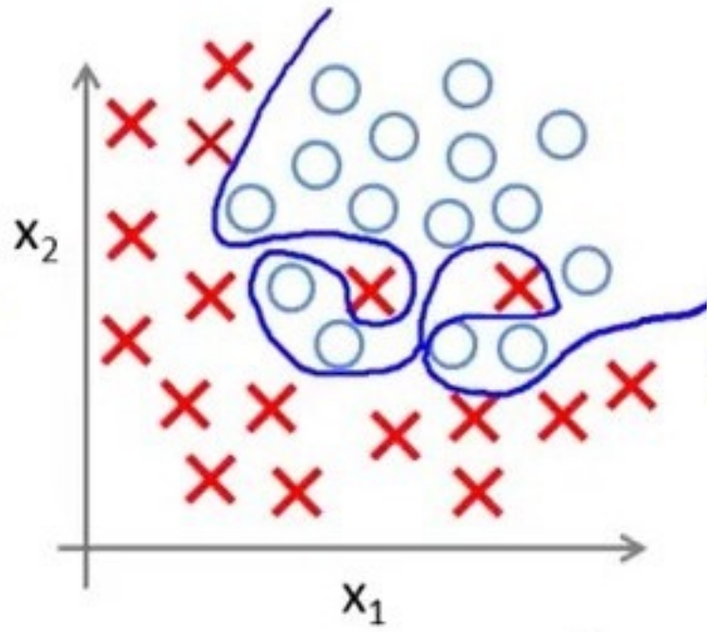
https://github.com/bmtgoncalves/Neural-Networks/

- 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 suitable way

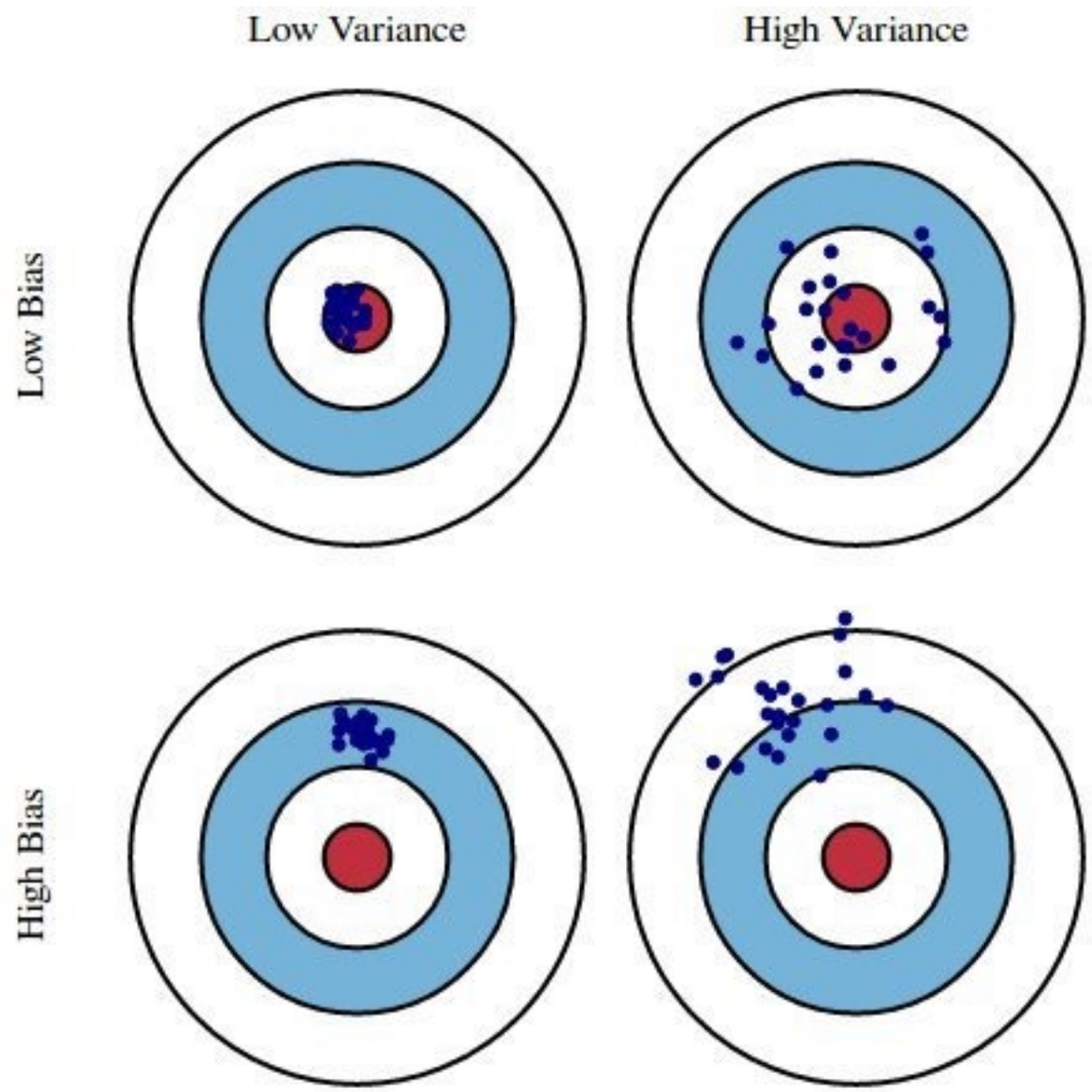
Supervised Learning - Overfitting

- "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.

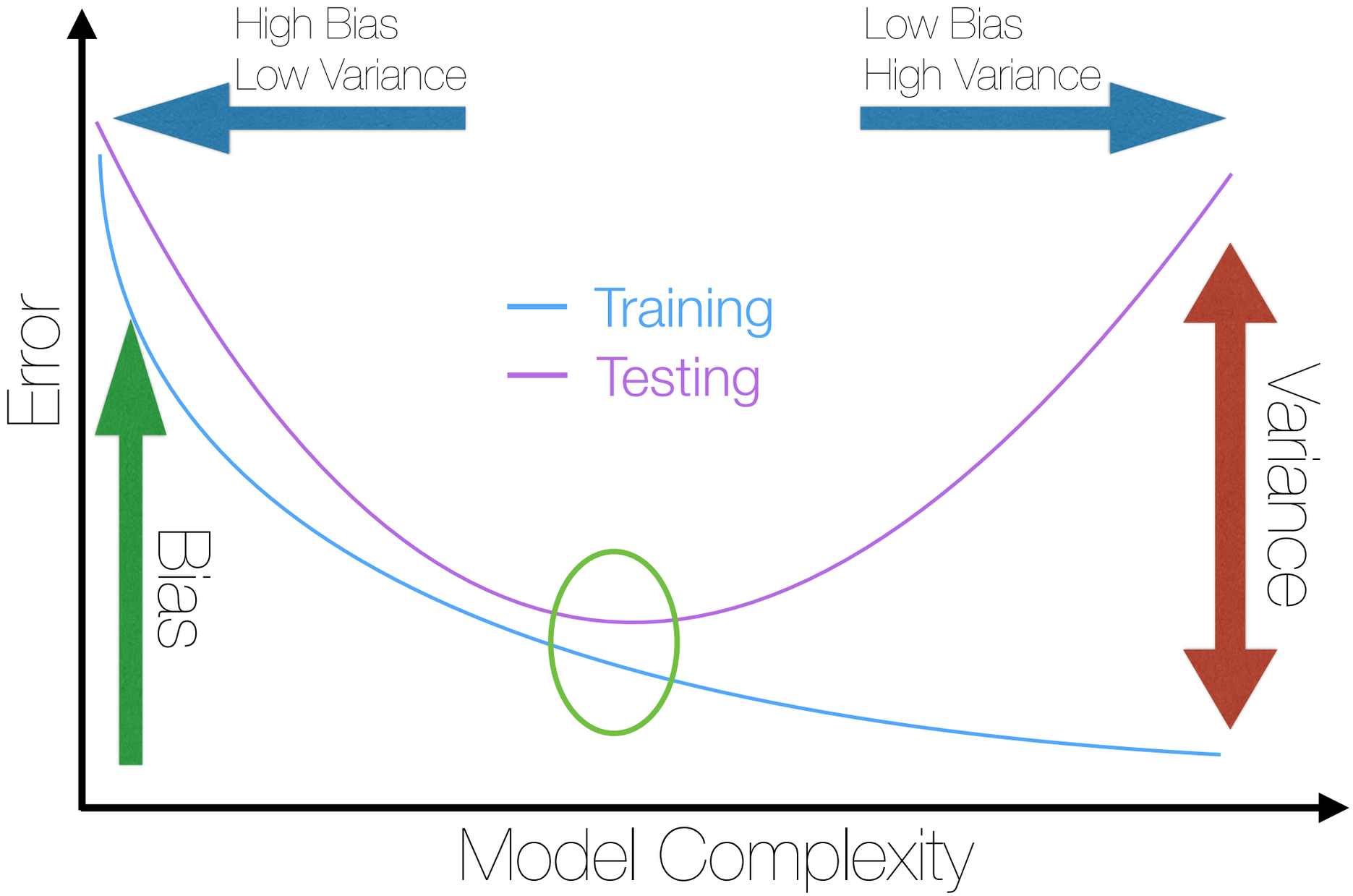




Bias-Variance Tradeoff



Bias-Variance Tradeoff



Comparison

- Linear Regression

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

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

$$\phi(Z) = Z$$

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

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

- Logistic Regression

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

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

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

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

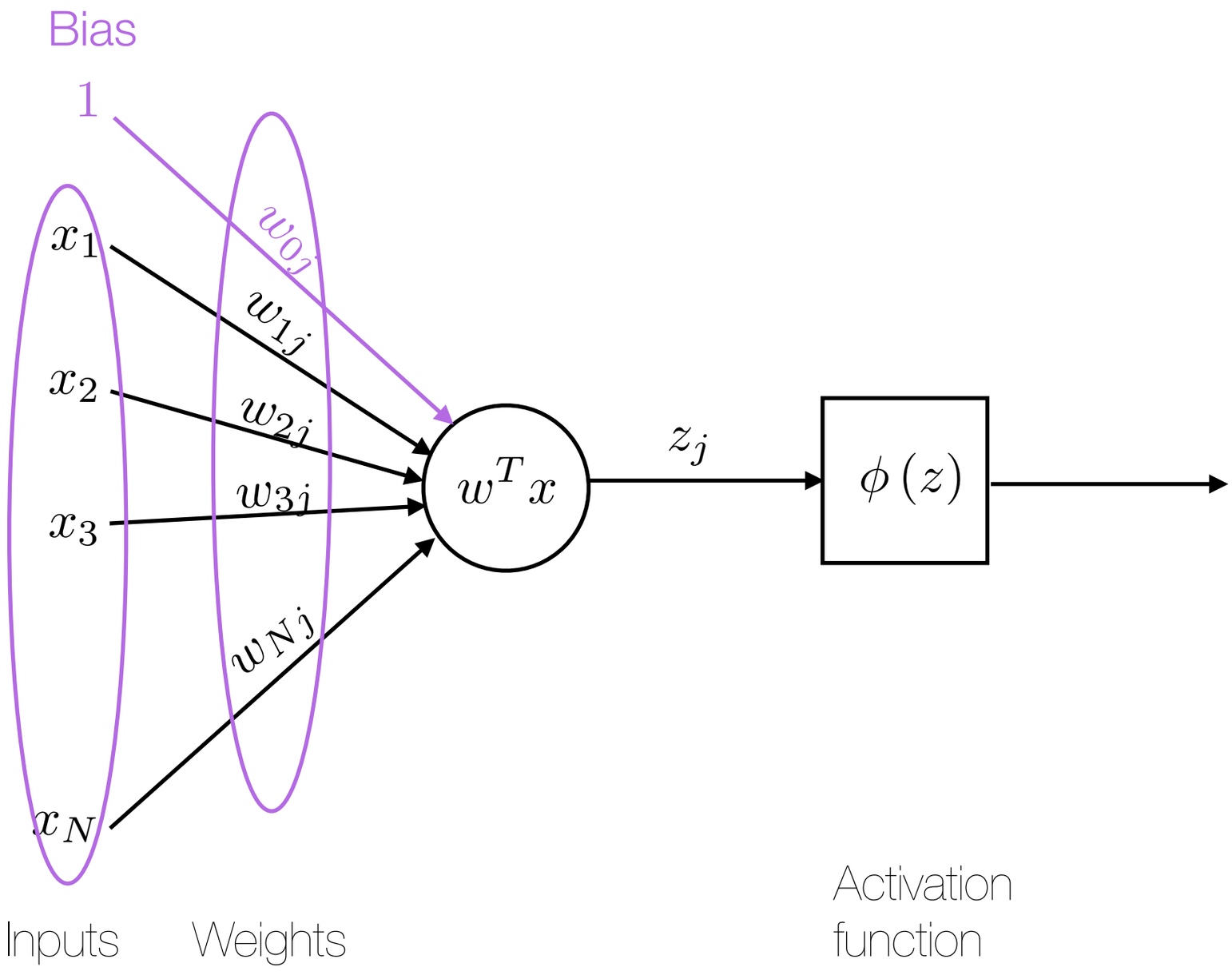
$$\frac{\delta}{\delta w_j} J_w\left(X, \vec{y}\right) = \frac{1}{m} X^T \cdot \left(h_w(X) - \vec{y}\right)$$

Map features to a continuous variable

Predict based on continuous variable

Compare prediction with reality

Similar Structure



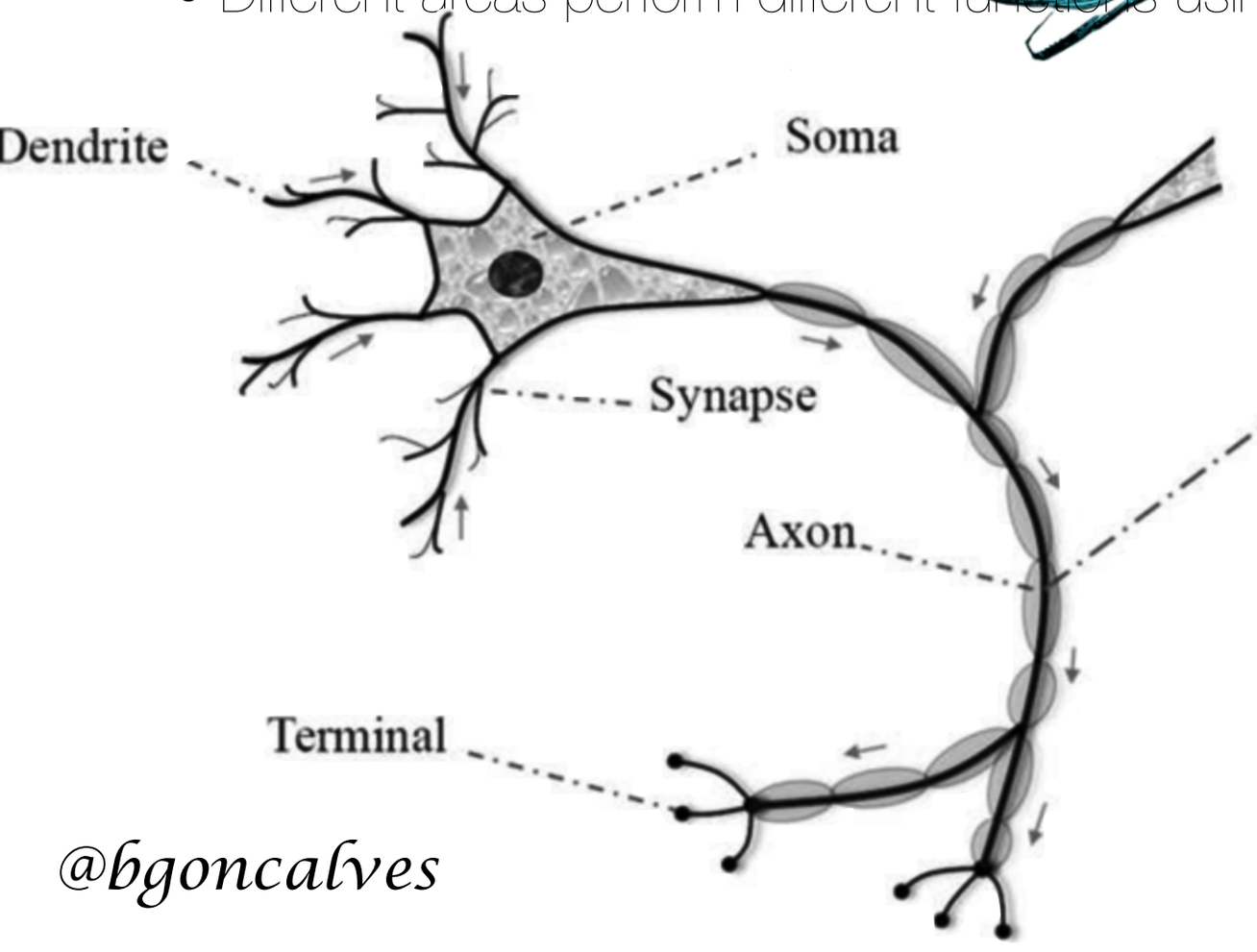
What about Neurons?

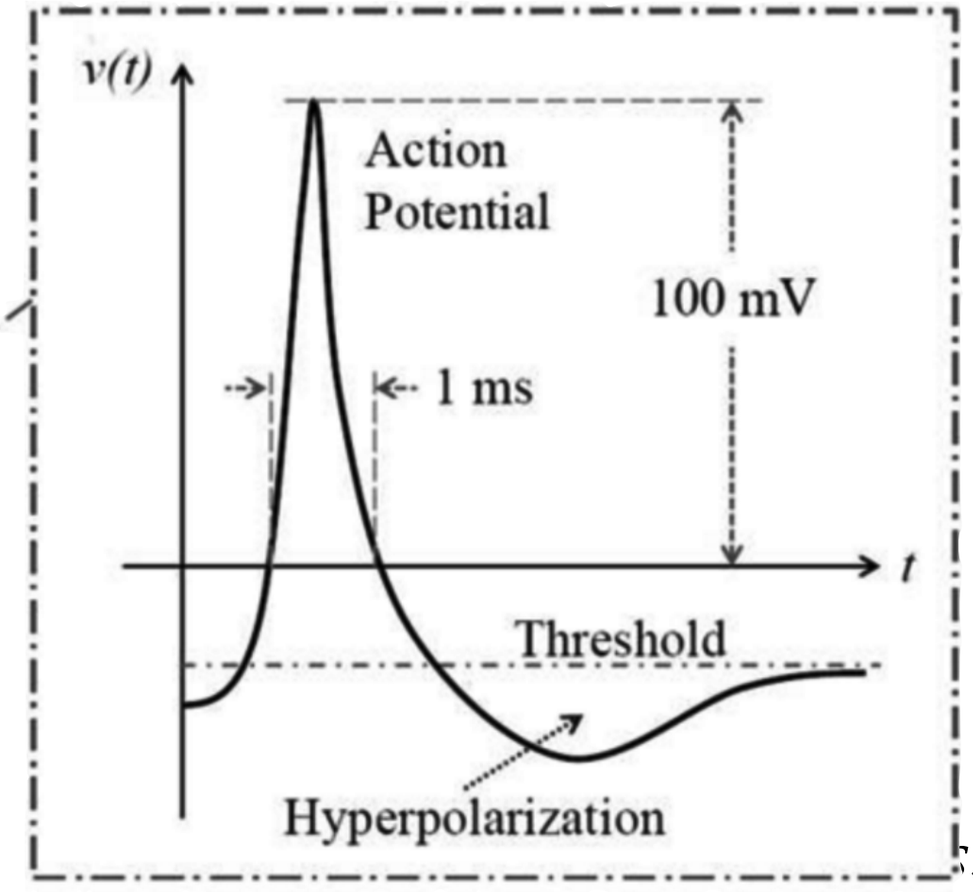
How the Brain "Works" (Cartoon version)

How the Brain "Works" (Cartoon version)

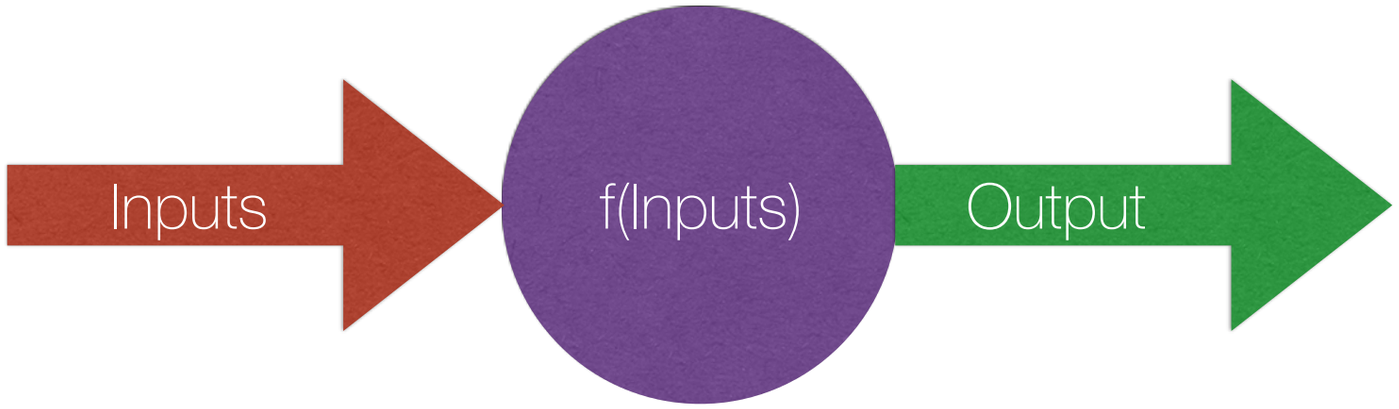
- Each neuron receives input from other neurons
- 10^{11} neurons, each with with 10^{4} weights
- Weights can be positive or negative
- Weights adapt during the learning process
- "neurons that fire together wire together" (Hebb)
- Different areas perform different functions using same structure (Modularity)

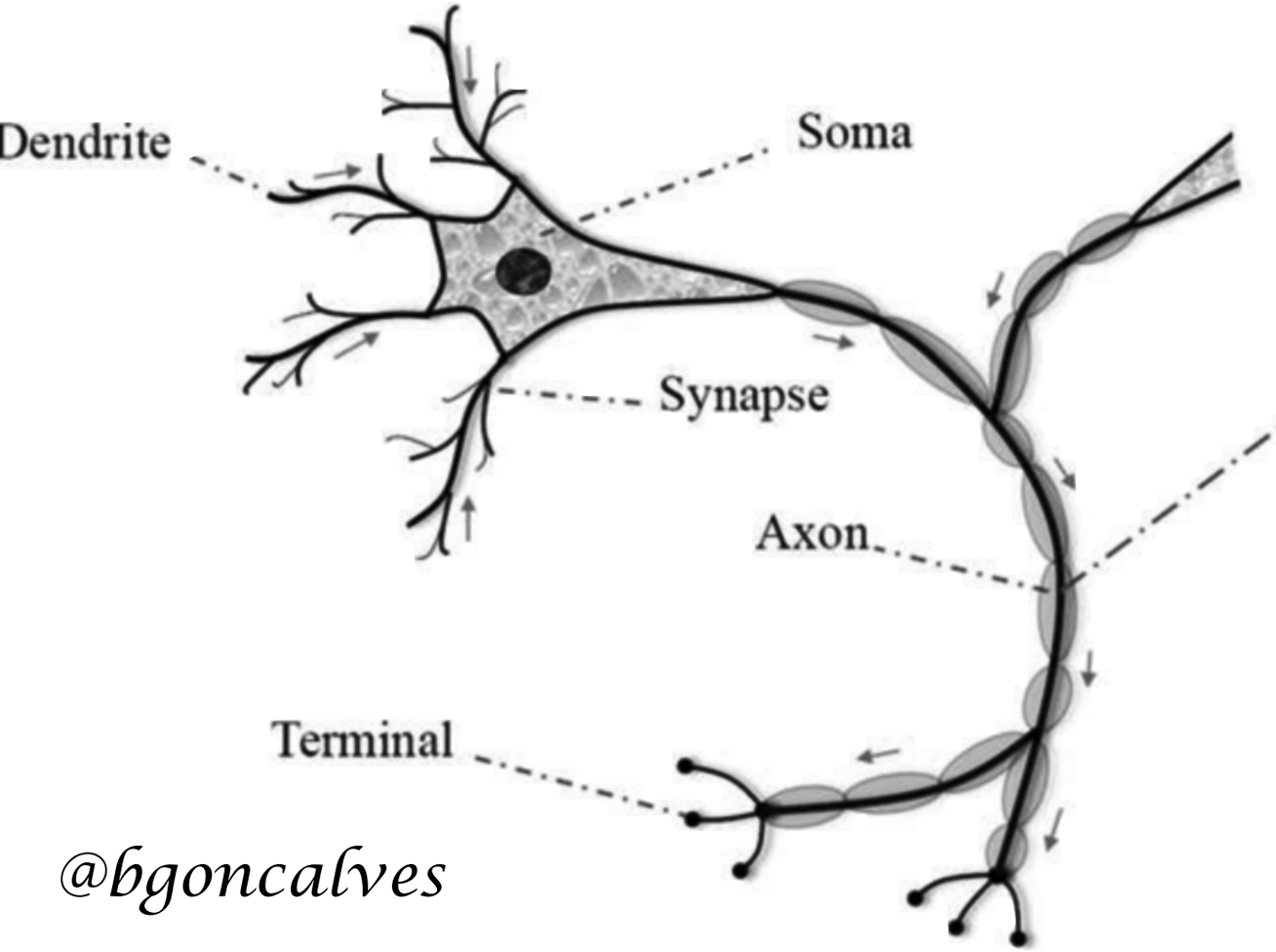
Input → Output

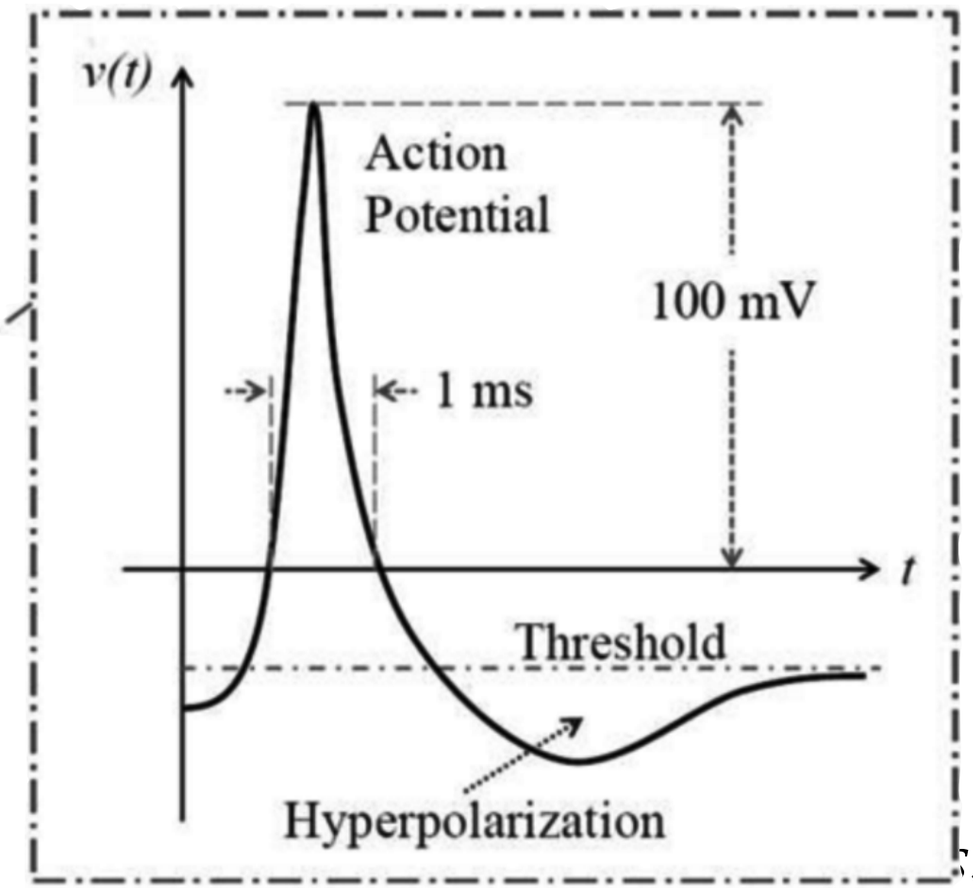


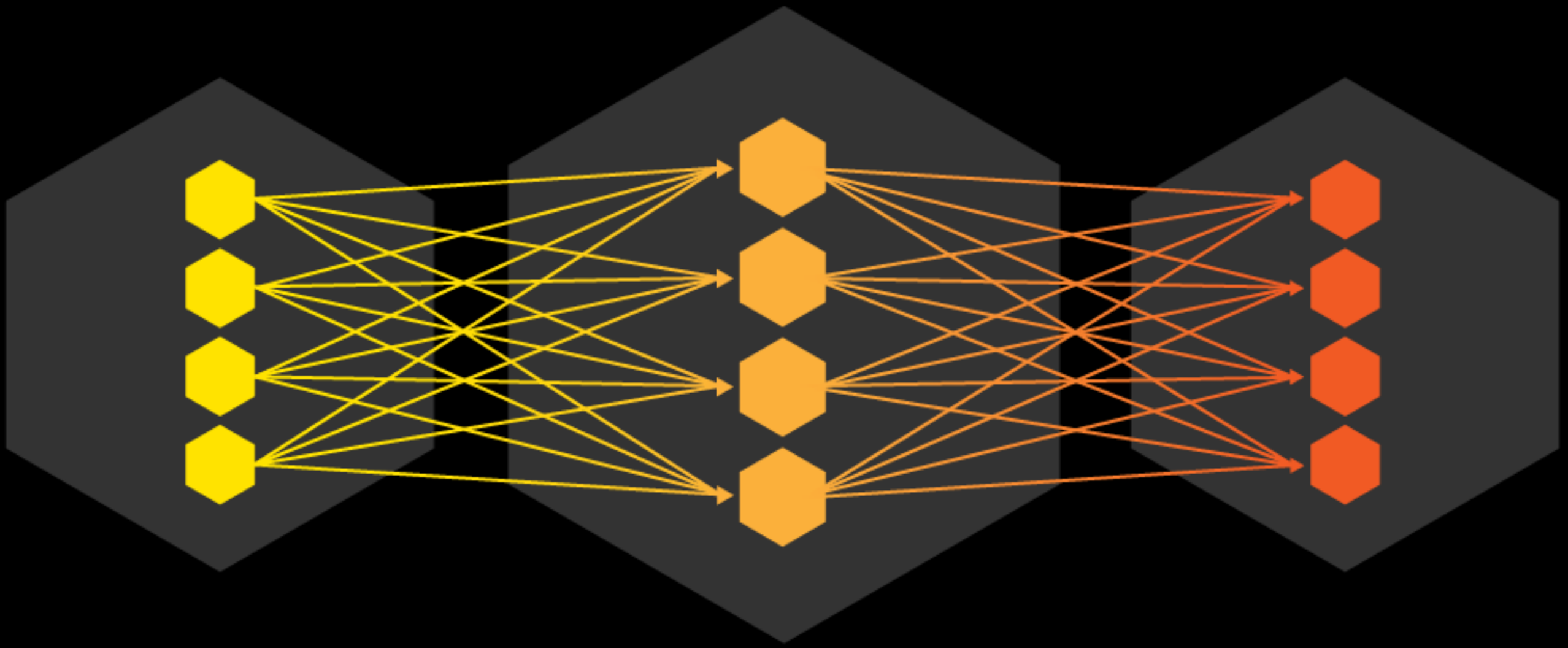


How the Brain "Works" (Cartoon version)









INPUT TERMS

FEATURES
PREDICTIONS
ATTRIBUTES
PREDICTABLE VARIABLES

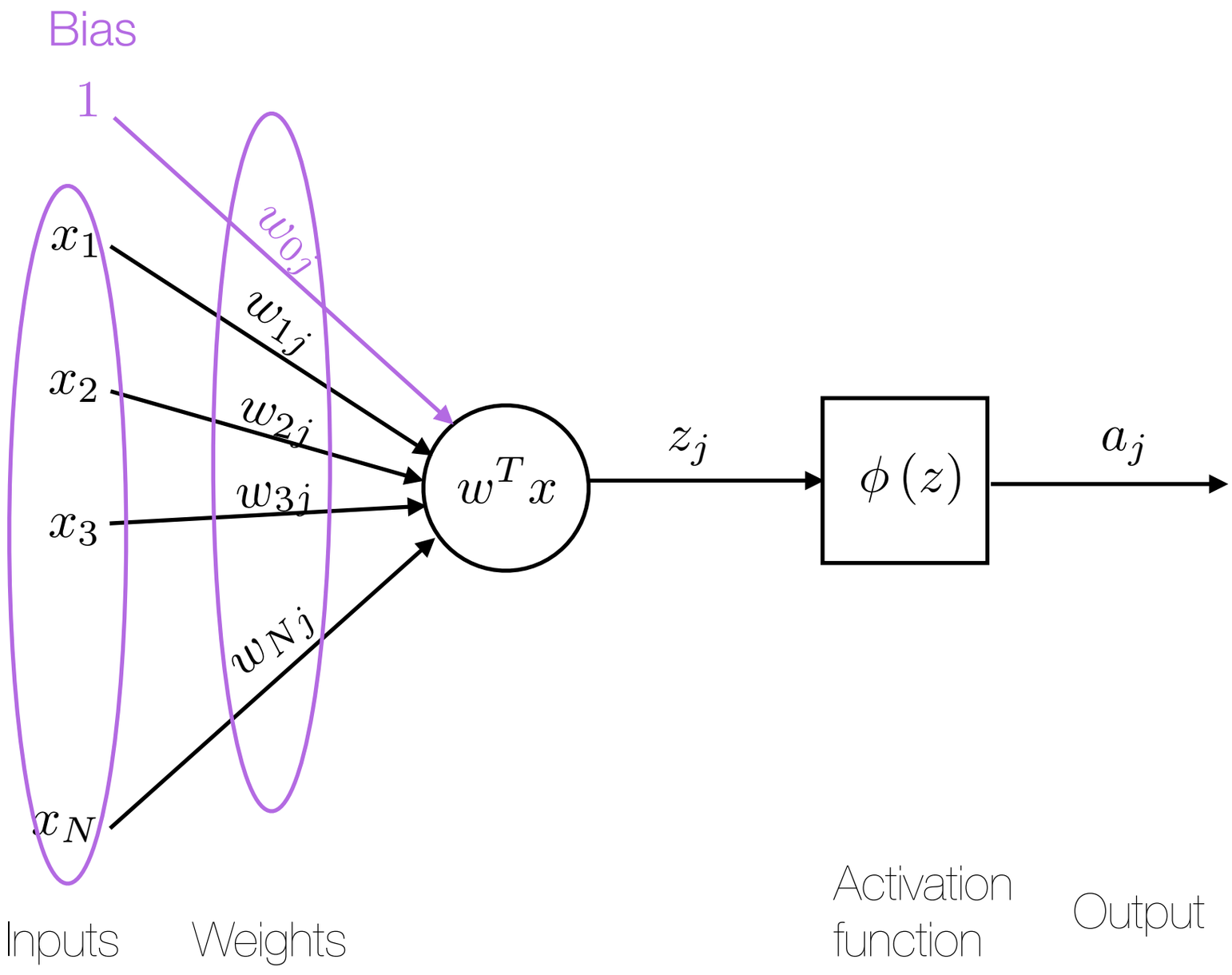
MACHINE

ALGORITHMS
TECHNIQUES
MODELS

OUTPUT TERMS

CLASSES
RESPONSES
TARGETS
DEPENDANT VARIABLES

Perceptron


Bias
1
w_{0j}
x_1
w_{1j}
x_2
w_{2j}
x_3
w_{3j}
w_{Nj}
x_N
$w^T x$
z_j
$\phi(z)$
a_j
Inputs
Weights
Activation function
Output

Activation Function

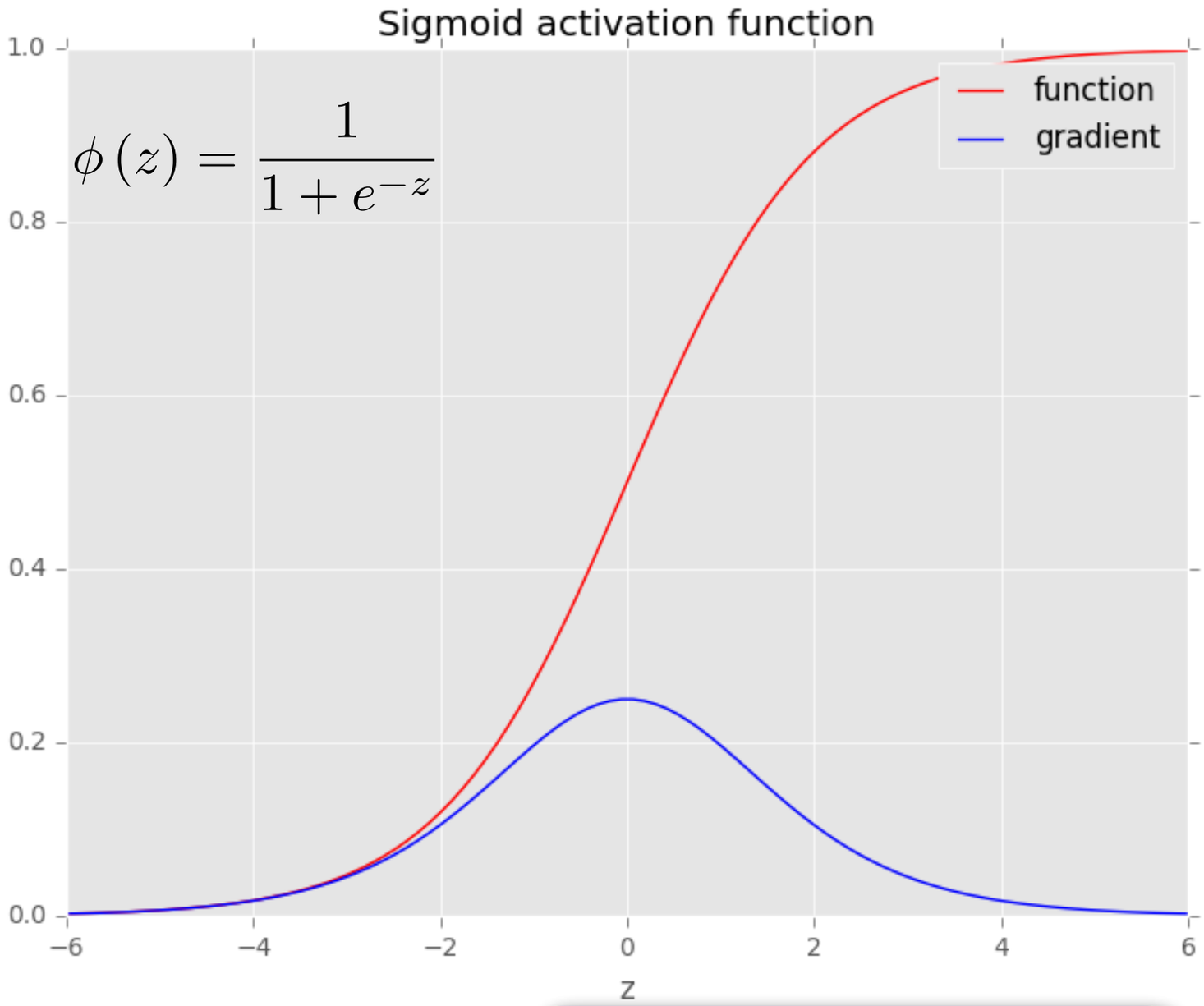
http://github.com/bmtgoncalves/Neural-Networks

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

Activation Function - Sigmoid

http://github.com/bmtgoncalves/Neural-Networks

- 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

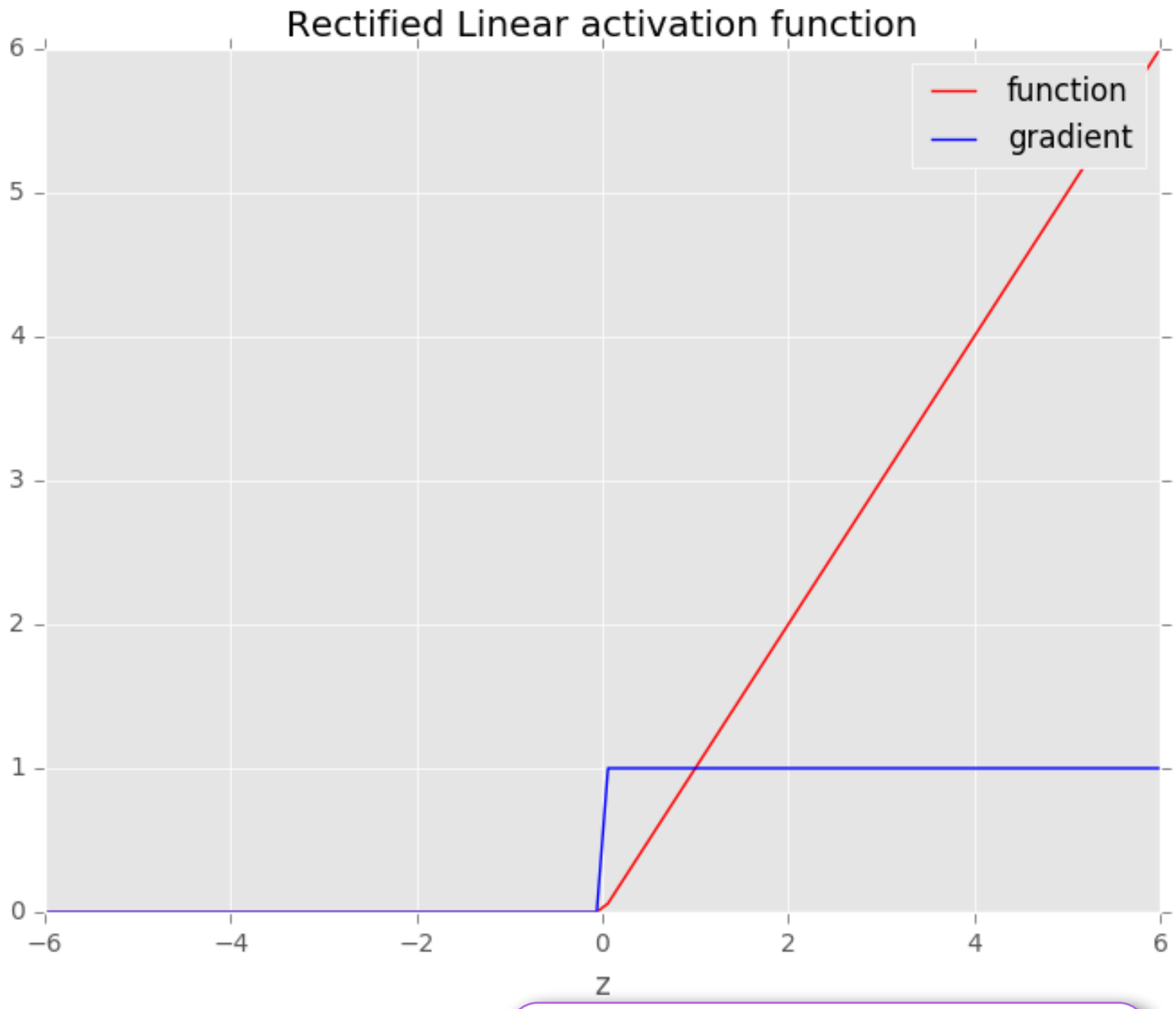


activation_simple.py

Activation Function - ReLu

http://github.com/bmtgoncalves/Neural-Networks

- 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

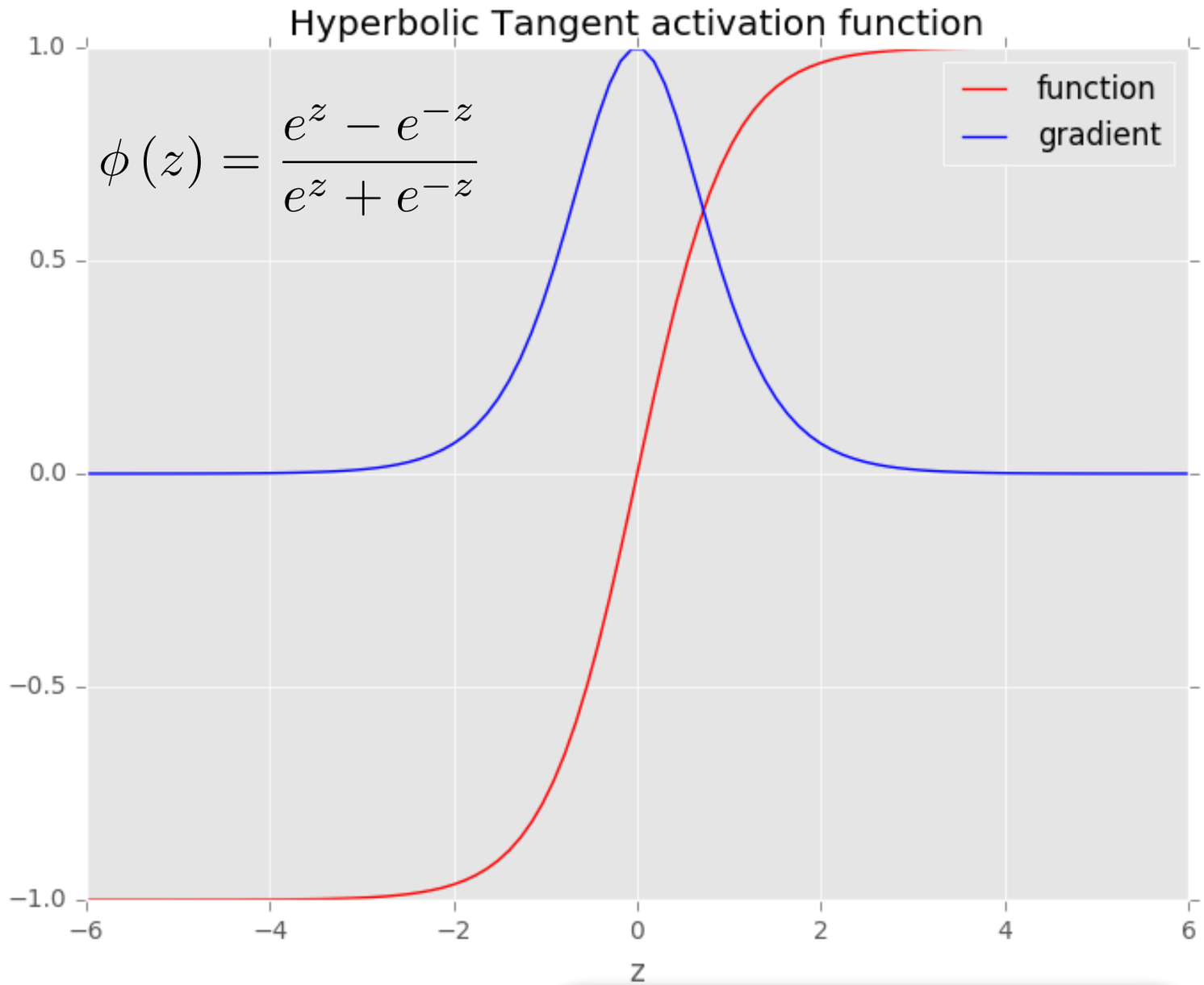


activation_simple.py

Activation Function - tanh

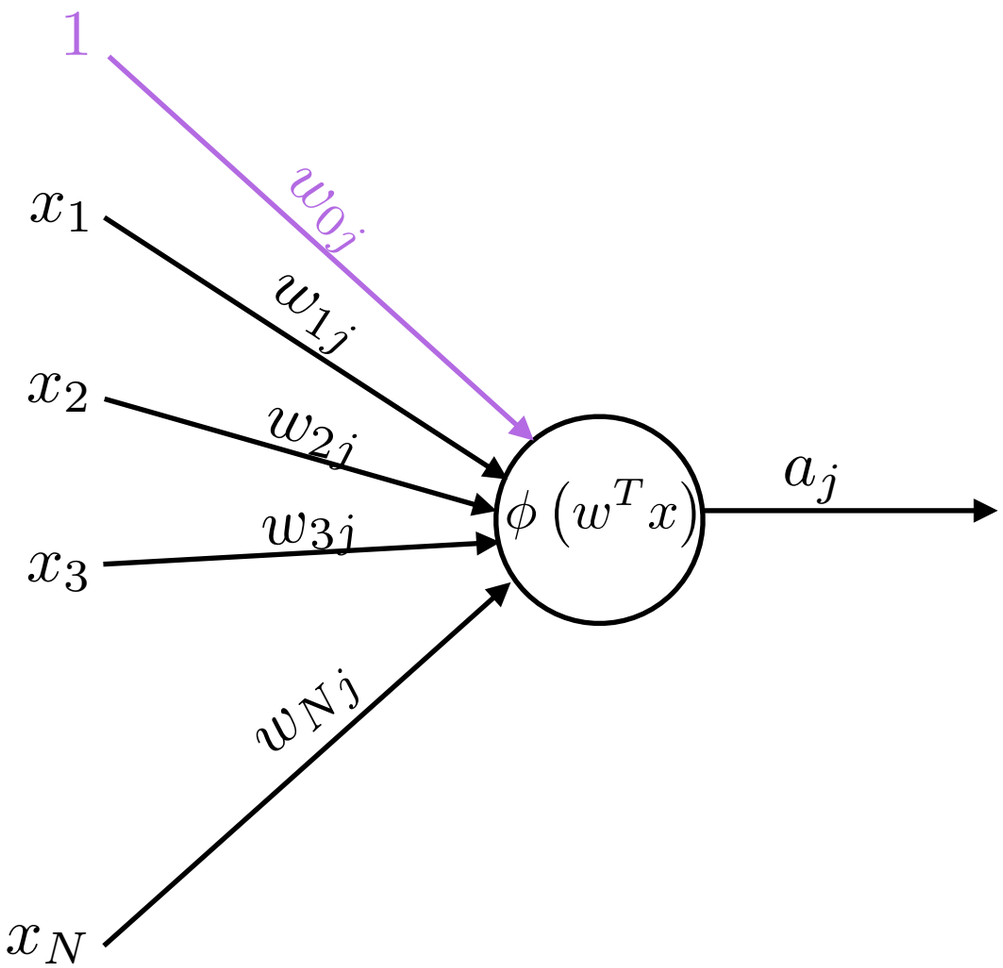
http://github.com/bmtgoncalves/Neural-Networks

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



activation_simple.py

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
  - calculate value of the activation function
  - (map activation value to predicted output)

$$a_j = \phi\left(\omega^T x\right)$$

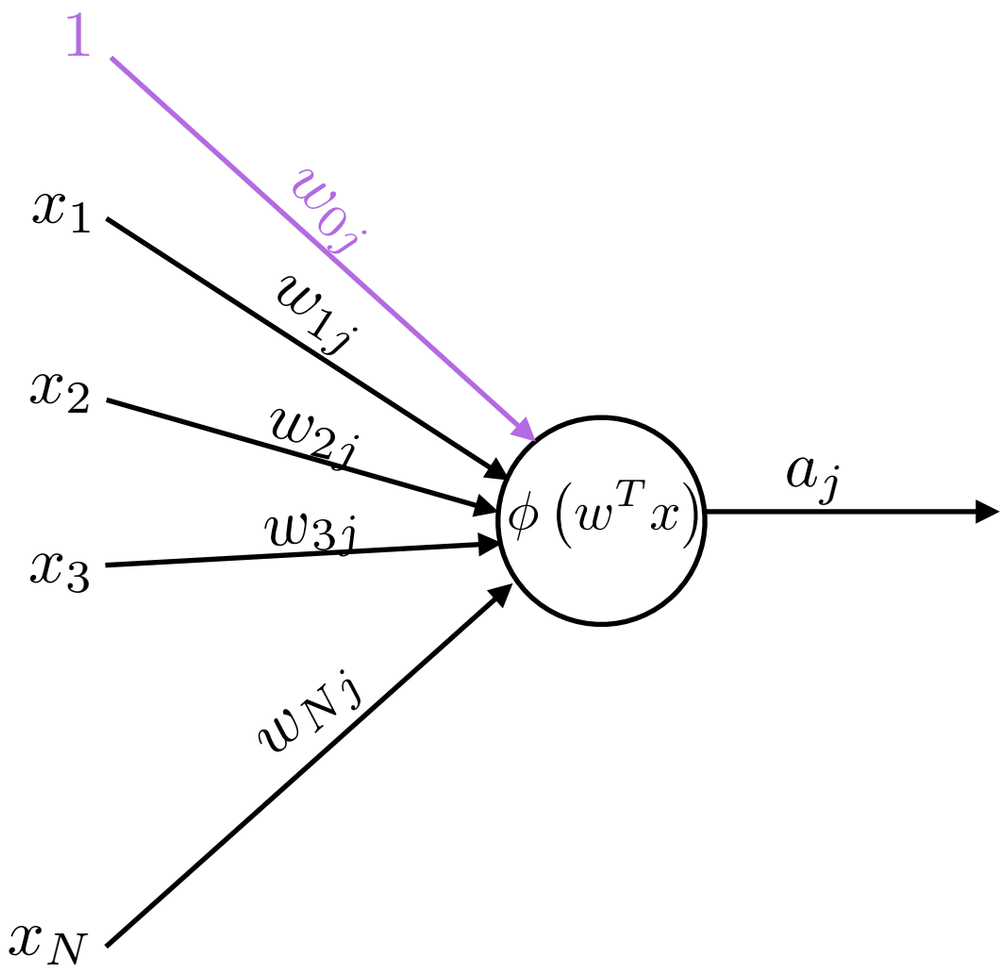
Perceptron - Forward Propagation

http://github.com/bmtgoncalves/Neural-Networks

```python
import numpy as np

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

if __name__ == "__main__":
    Theta1 = np.load('input/Theta1.npy')
    X = np.load('input/X_train.npy')[:10]

    from activation import sigmoid

    active_value = forward(Theta1, X, sigmoid)
```


forward_simple.py

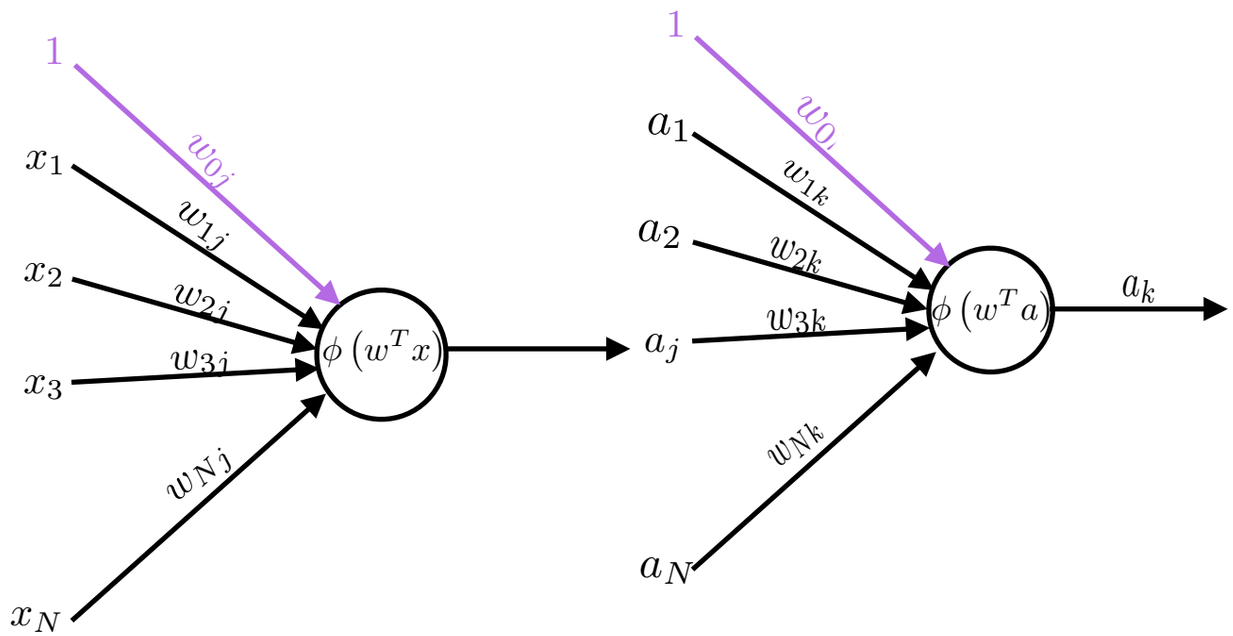
Perceptron - Training



- 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

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?

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:

$$E = \frac{1}{N} \sum_n |y_n - a_n|^2$$

  - Cross Entropy

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

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:

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

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

- 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:

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

- Until:

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

as there's no error on the input layer.

- And finally:

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

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

A practical example - MNIST

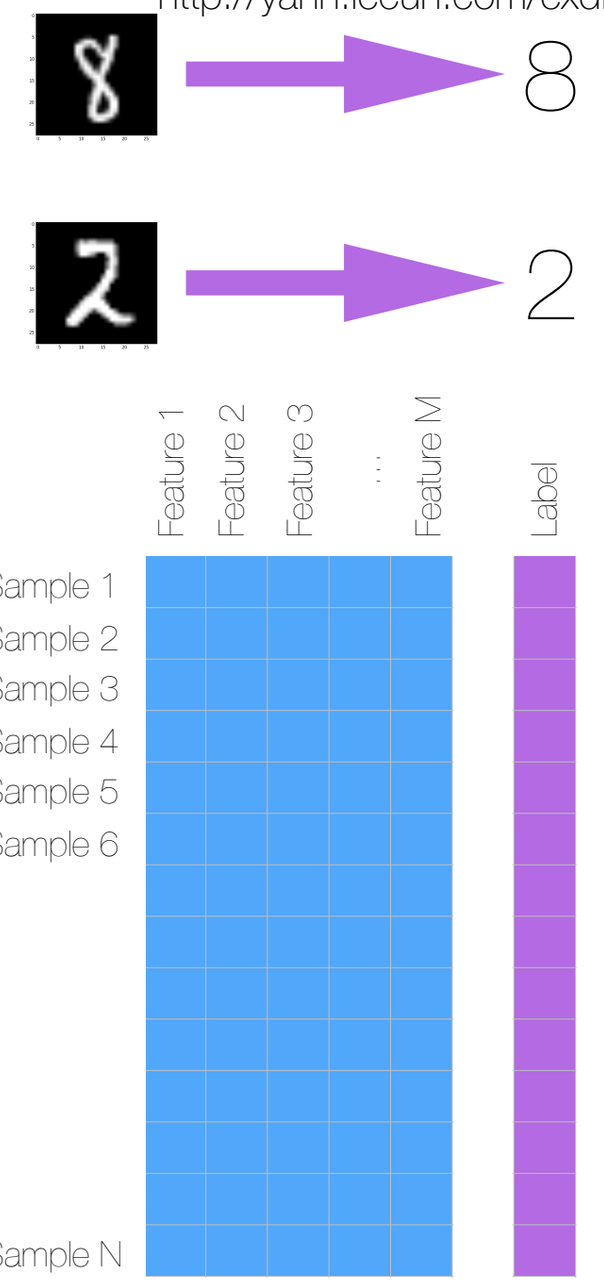
http://github.com/bmtgoncalves/Neural-Networks

http://yann.lecun.com/exdb/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



visualize_digits.py
convert_input.py

A practical example - MNIST

http://github.com/bmtgoncalves/Neural-Networks

http://yann.lecun.com/exdb/mnist/

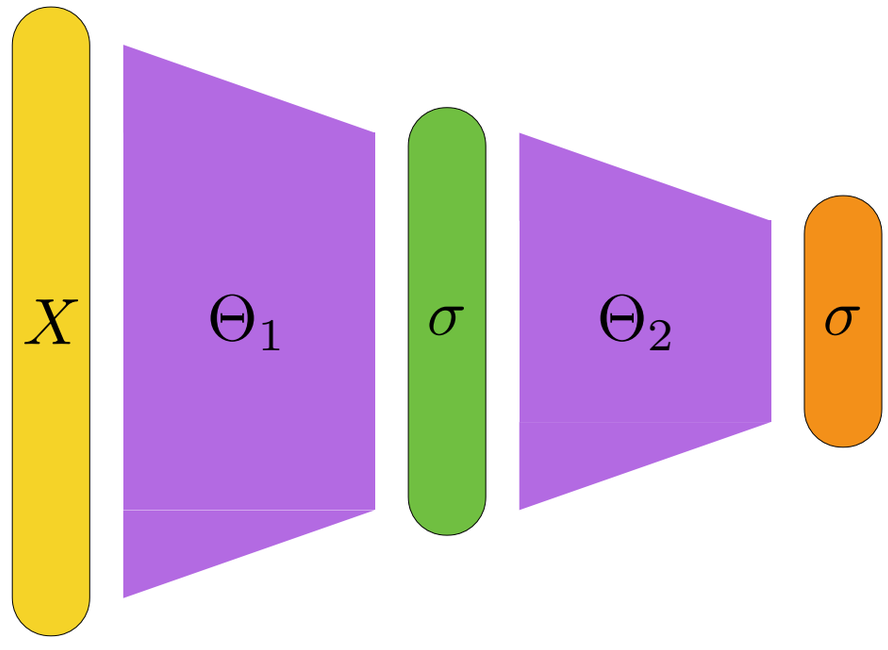
THE MNIST DATABASE

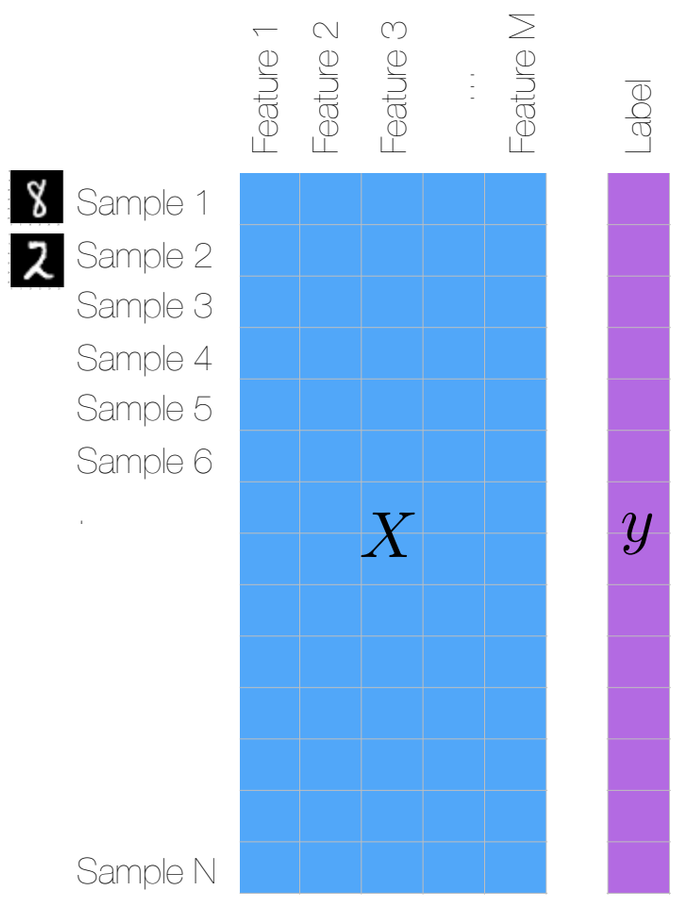
of handwritten digits

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

3 layers:

- 1 input layer
- 1 hidden layer
- 1 output layer



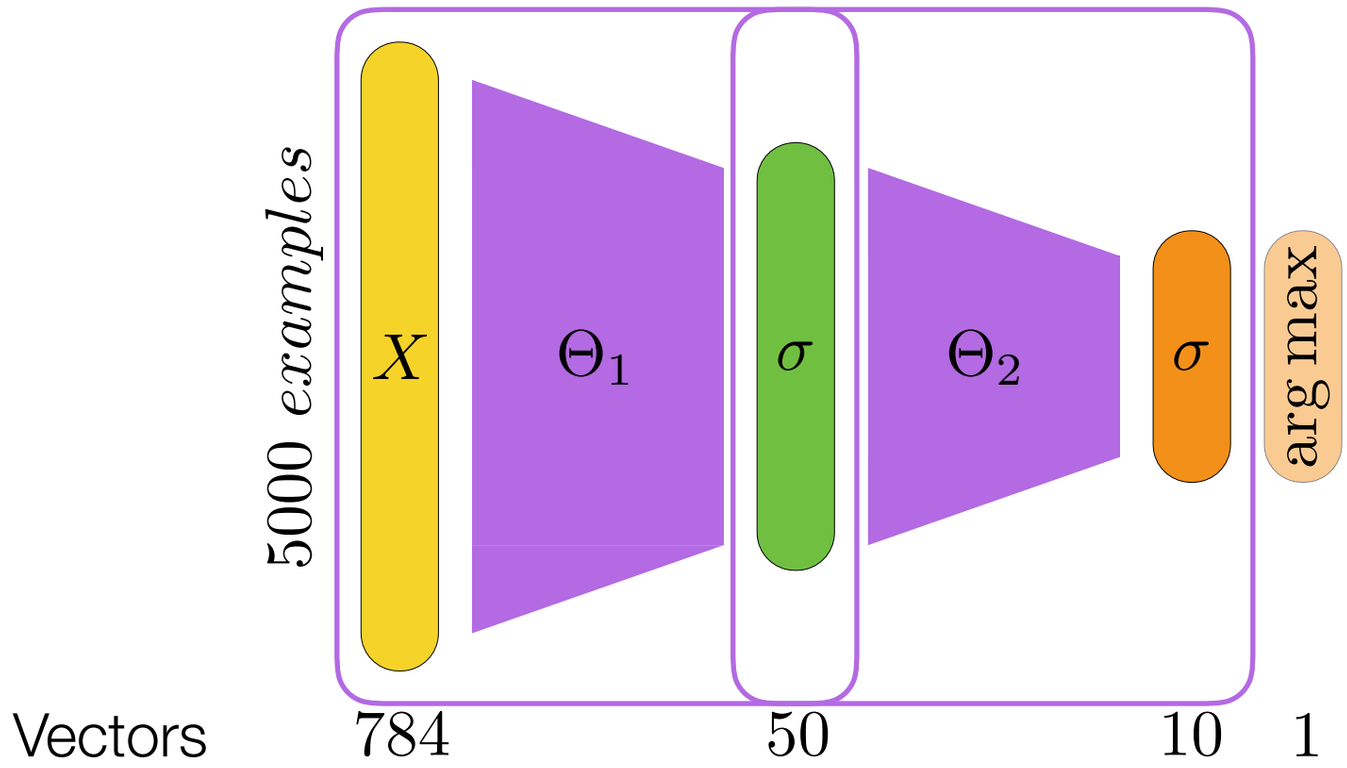


visualize_digits.py
convert_input.py

@bgoncalves

A practical example - MNIST

http://github.com/bmtgoncalves/Neural-Networks




```python
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
```

```python
def predict(Theta1, Theta2, X):
    h1 = forward(Theta1, X, sigmoid)
    h2 = forward(Theta2, h1, sigmoid)

    return np.argmax(h2, 1)
```

nn_simple.py
forward_simple.py

Backprop

http://github.com/bmtgoncalves/Neural-Networks

```python
def backprop(Theta1, Theta2, X, y):
    N = X.shape[0]
    K = Theta2.shape[0]
    J = 0

    Delta2 = np.zeros(Theta2.shape)
    Delta1 = np.zeros(Theta1.shape)

    for i in range(N): # Forward propagation, saving intermediate results
        a1 = np.concatenate(([1], X[i]))  # Input layer
        z2 = np.dot(Theta1, a1)
        a2 = np.concatenate(([1], sigmoid(z2)))  # Hidden Layer
        z3 = np.dot(Theta2, a2)
        a3 = sigmoid(z3)  # Output layer
        y0 = one_hot(K, y[i])

        # Cross entropy
        J -= np.dot(y0.T, np.log(a3))+np.dot((1-y0).T, np.log(1-a3))

        # Calculate the weight deltas
        delta_3 = a3-y0
        delta_2 = np.dot(Theta2.T, delta_3)[1:]*sigmoidGradient(z2)
        Delta2 += np.outer(delta_3, a2)
        Delta1 += np.outer(delta_2, a1)

J /= N

Theta1_grad = Delta1/N
Theta2_grad = Delta2/N

return [J, Theta1_grad, Theta2_grad]
```

nn_simple.py

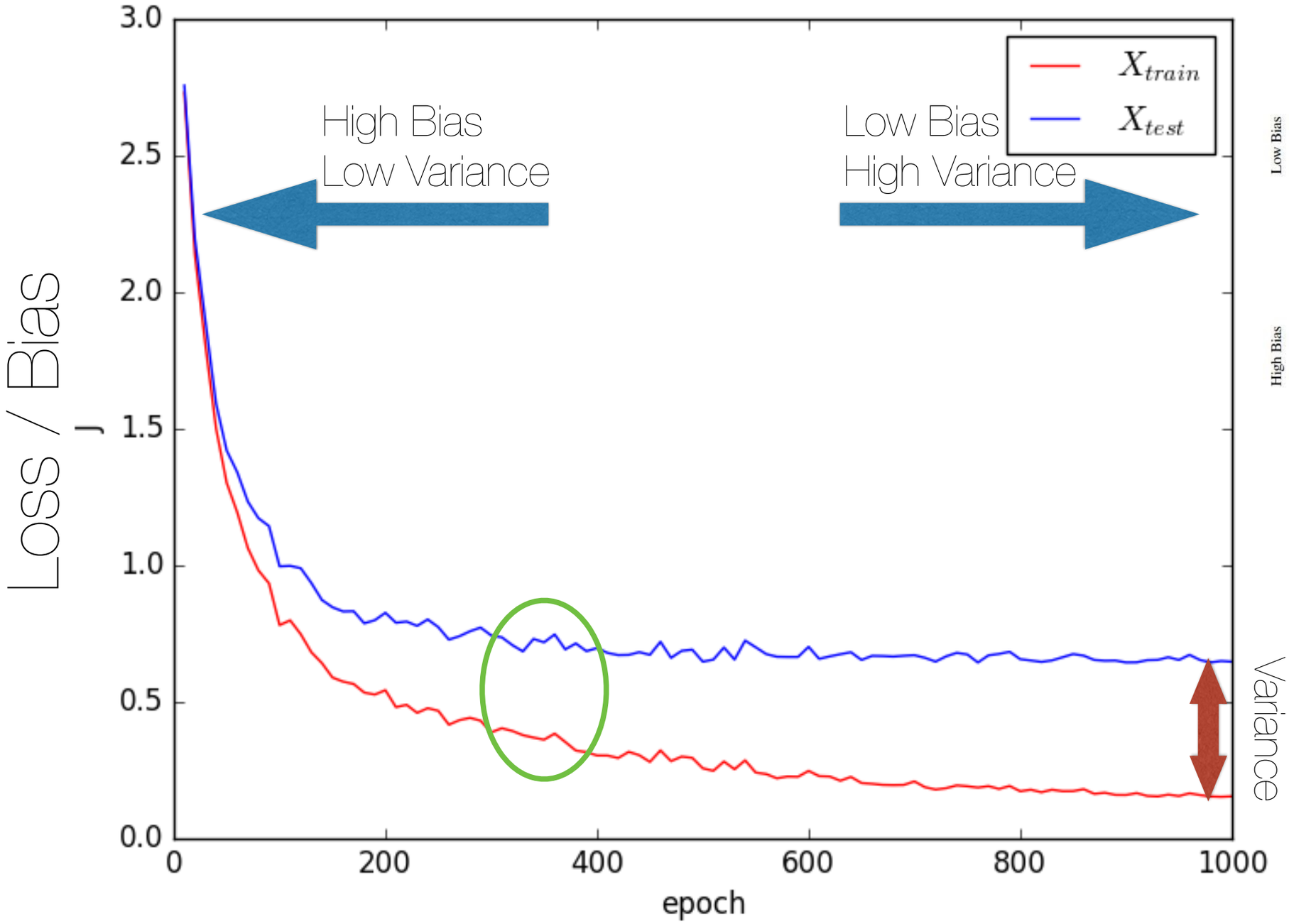
Training

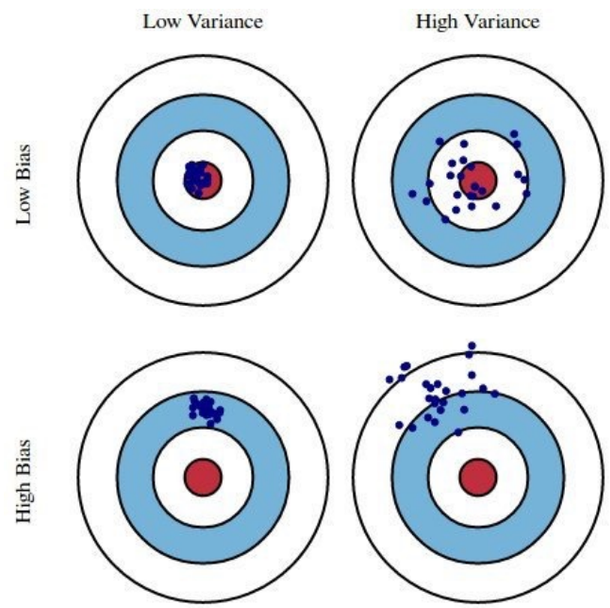
http://github.com/bmtgoncalves/Neural-Networks

```
Theta1 = init_weights(input_layer_size, hidden_layer_size)
Theta2 = init_weights(hidden_layer_size, num_labels)

iter = 0
tol = 1e-3
J_old = 1/tol
diff = 1
alpha = 0.5

while diff > tol:
    J_train, Theta1_grad, Theta2_grad = backprop(Theta1, Theta2, X_train, y_train)

    diff = abs(J_old-J_train)
    J_old = J_train

    Theta1 -= alpha*Theta1_grad
    Theta2 -= alpha*Theta2_grad
```


nn_simple.py

Bias-Variance Tradeoff

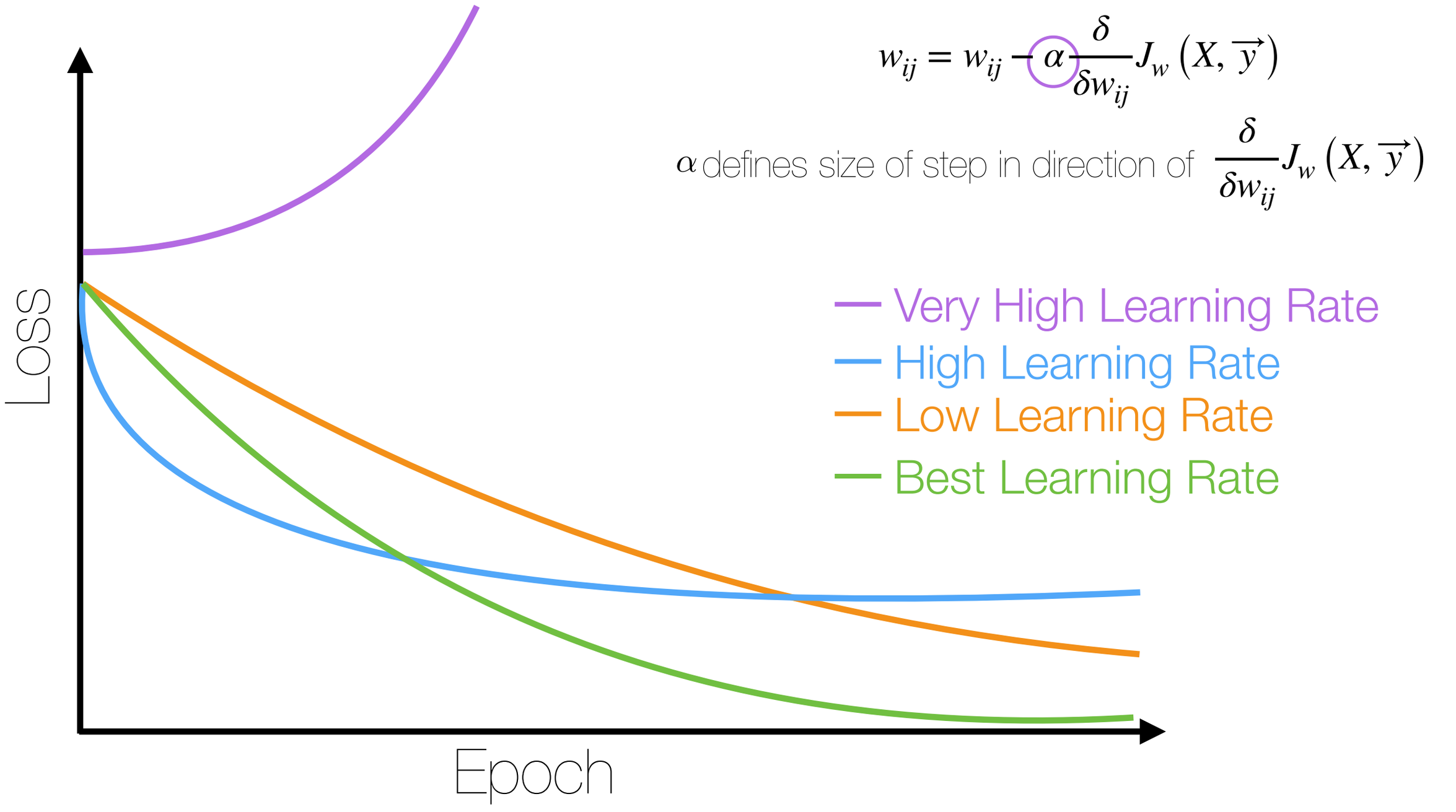




Learning Rate

$$w_{ij} = w_{ij} - \alpha \frac{\delta}{\delta w_{ij}} J_w\left(X, \vec{y}\right)$$

α defines size of step in direction of $\frac{\delta}{\delta w_{ij}} J_w\left(X, \vec{y}\right)$



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:

```python
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)
```

activation.py

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.

nn_new.py

Generalization - Forward propagation

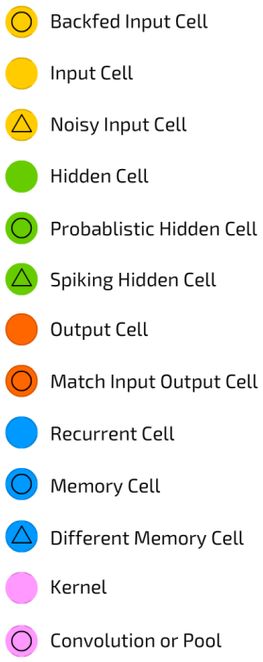
```python
def forward(Theta, X, active):
    N = X.shape[0]

    # 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)
```

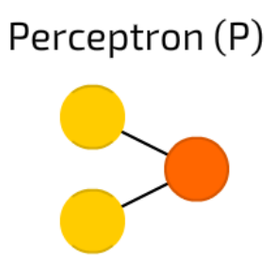
forward.py

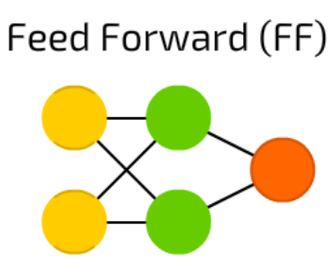
```python
def backprop(model, X, y):
    M = X.shape[0]

    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]
```

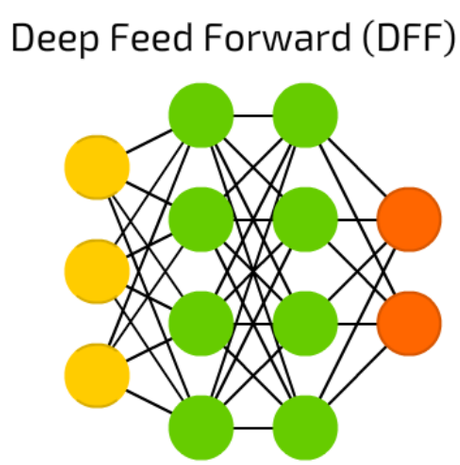
nn_simple.py

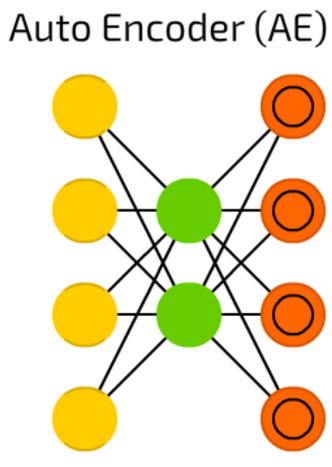
Neural Network Architectures

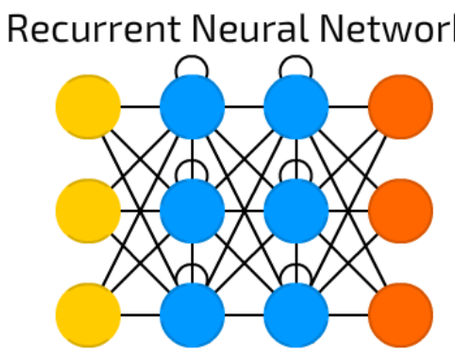












word2vec

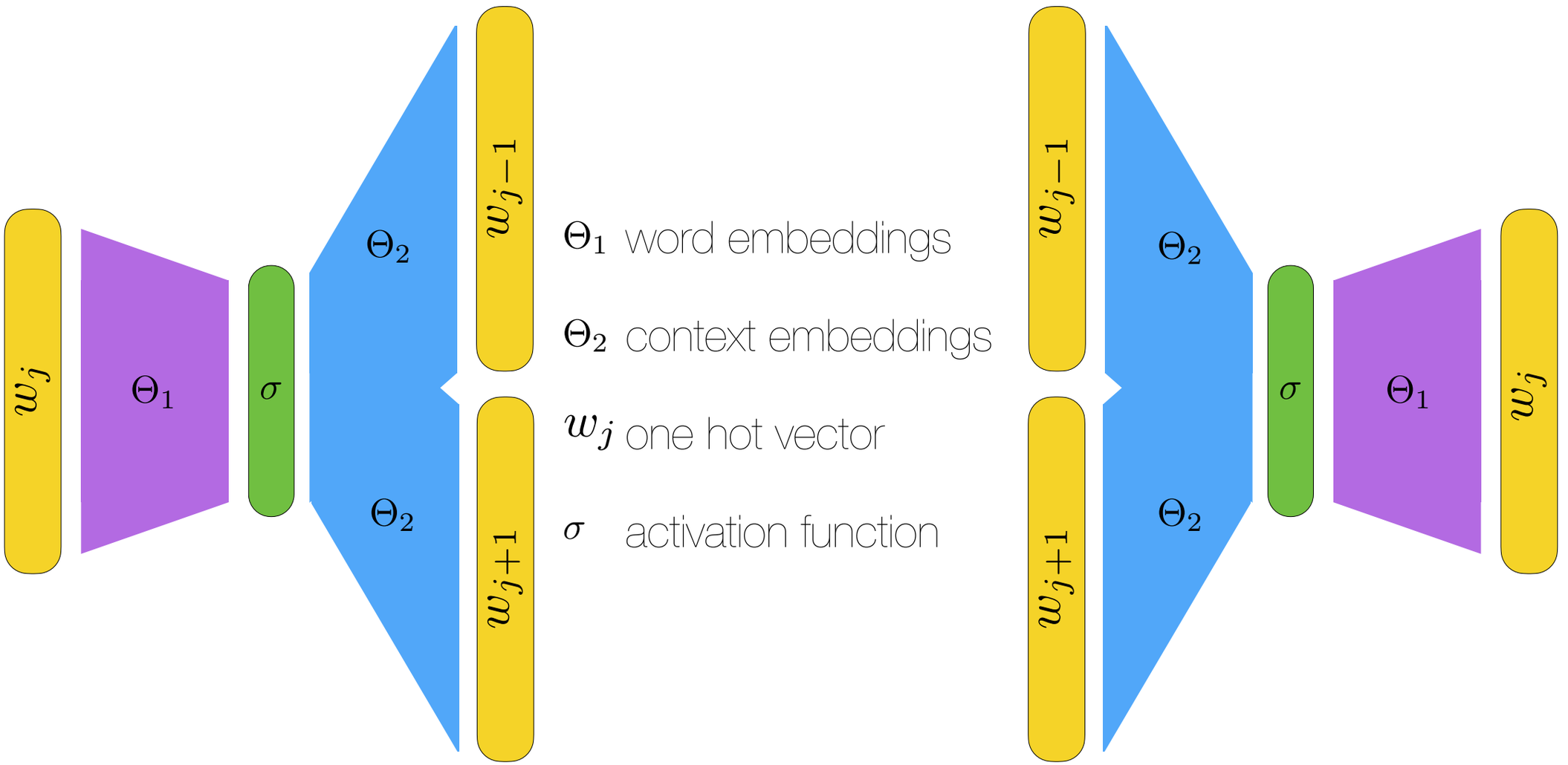
Mikolov 2013

Skipgram

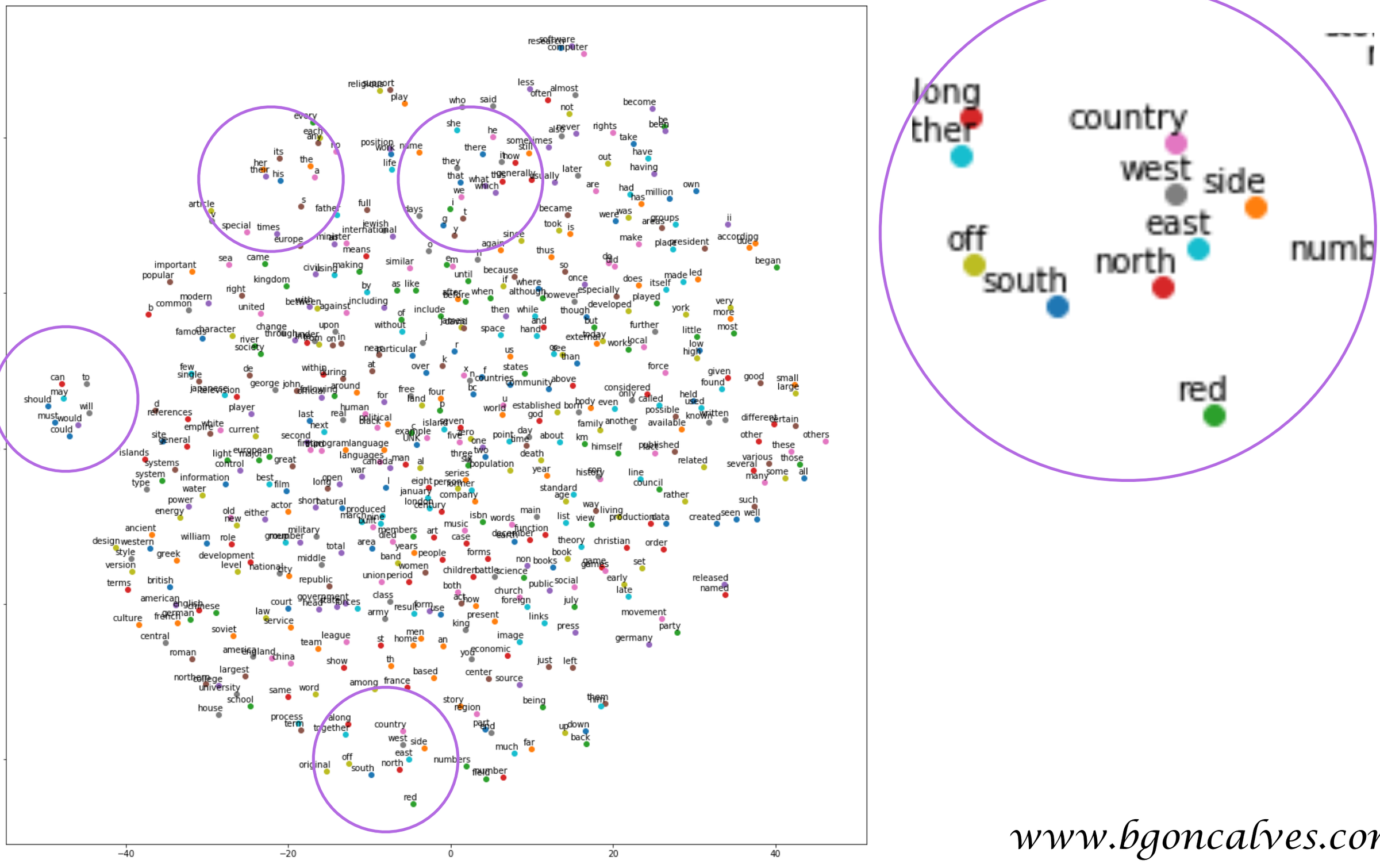
$$\max p\left(C|w\right)$$

Continuous Bag of Words

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

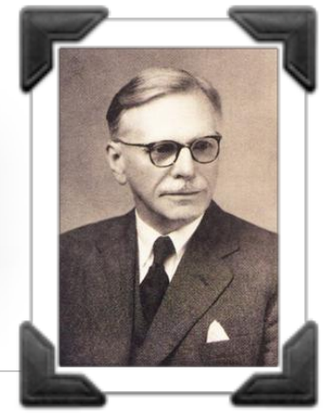


Visualization



Analogies

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



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

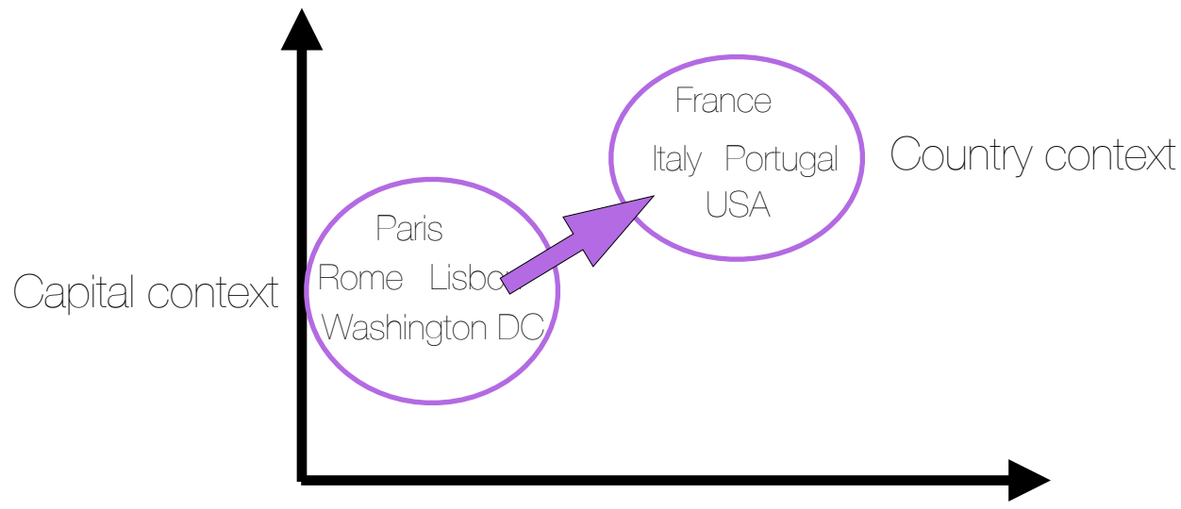
$$\sigma\left(red\right) = f\left(context\left(red\right)\right)$$

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

$$context\left(red\right) \approx context\left(blue\right) \implies \sigma\left(red\right) \approx \sigma\left(blue\right)$$

- "Distributional hypotesis" in linguistics

Geometrical relations between contexts imply semantic relations between words!



$$\sigma\left(France\right) - \sigma\left(Paris\right) + \sigma\left(Rome\right) = \sigma\left(Italy\right)$$

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

Analogies

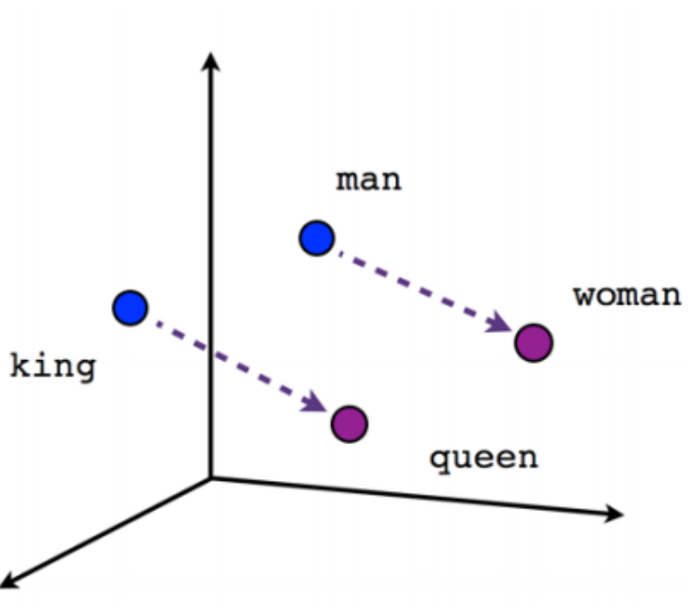
https://www.tensorflow.org/tutorials/word2vec

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

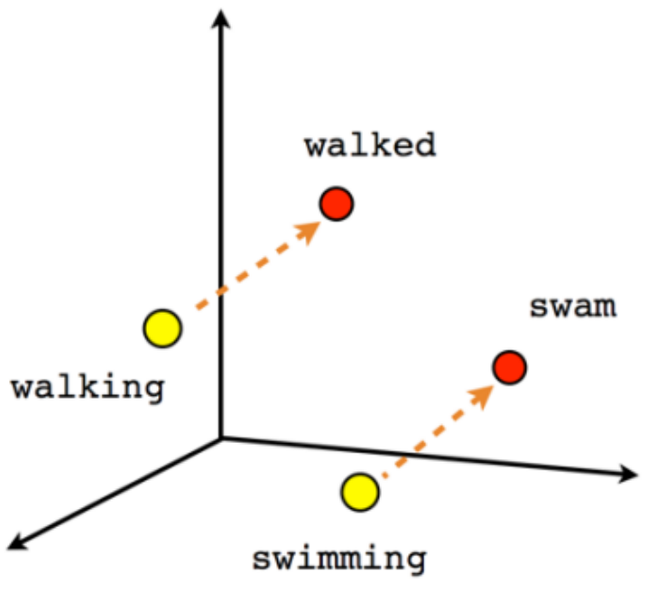
$$d^{\dagger} = \underset{x}{\text{argmax}} \frac{\left(\vec{b} - \vec{a} + \vec{c}\right)^{T}}{\left|\left|\vec{b} - \vec{a} + \vec{c}\right|\right|} \vec{x}$$

$$d^{\dagger} \sim \underset{x}{\text{argmax}} \left(\vec{b}^{T}\vec{x} - \vec{a}^{T}\vec{x} + \vec{c}^{T}\vec{x}\right)$$

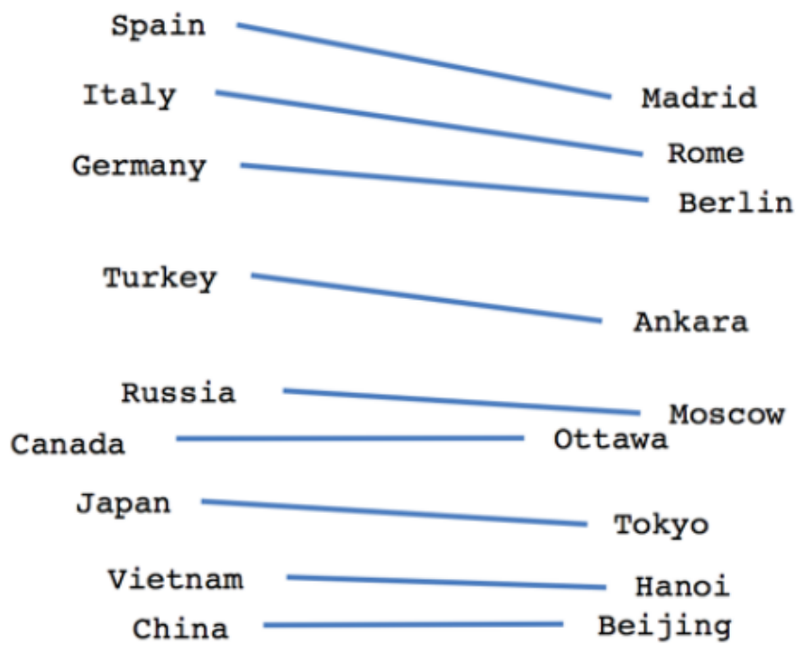
What is the word d that is most similar to b and c and most dissimilar to a?



Male-Female

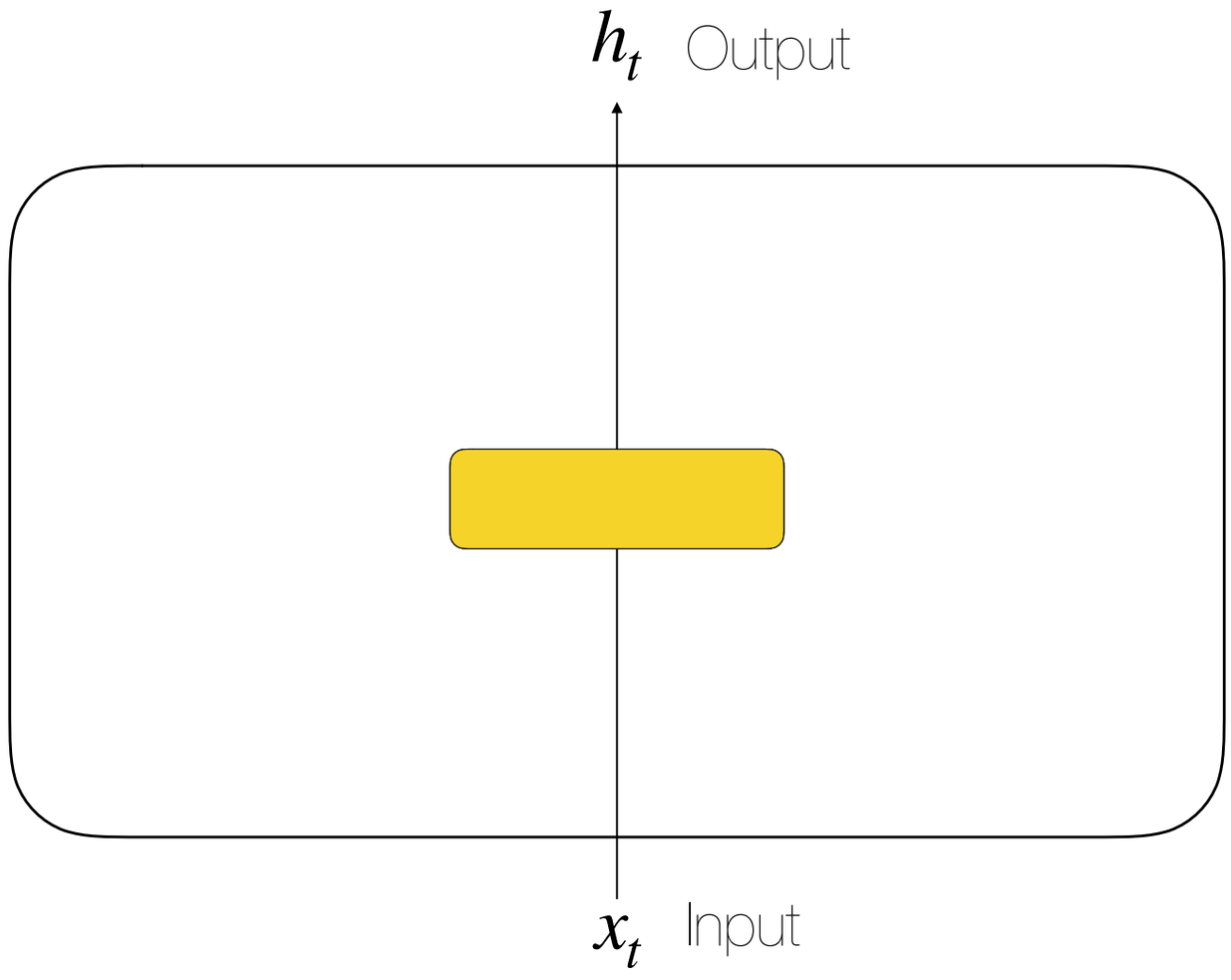


Verb tense



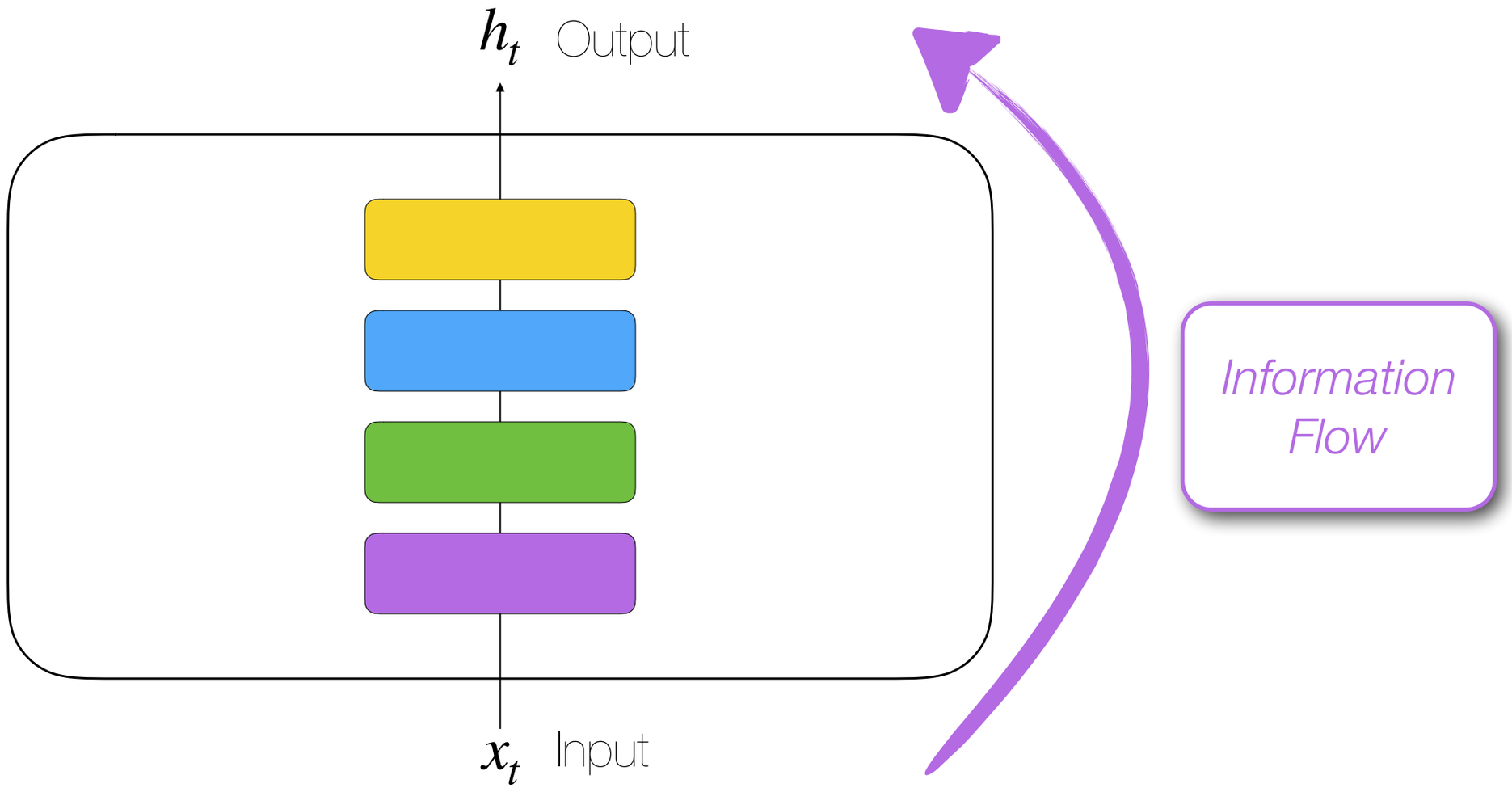
Country-Capital

Feed Forward Networks



$$h_t = f\left(x_t\right)$$

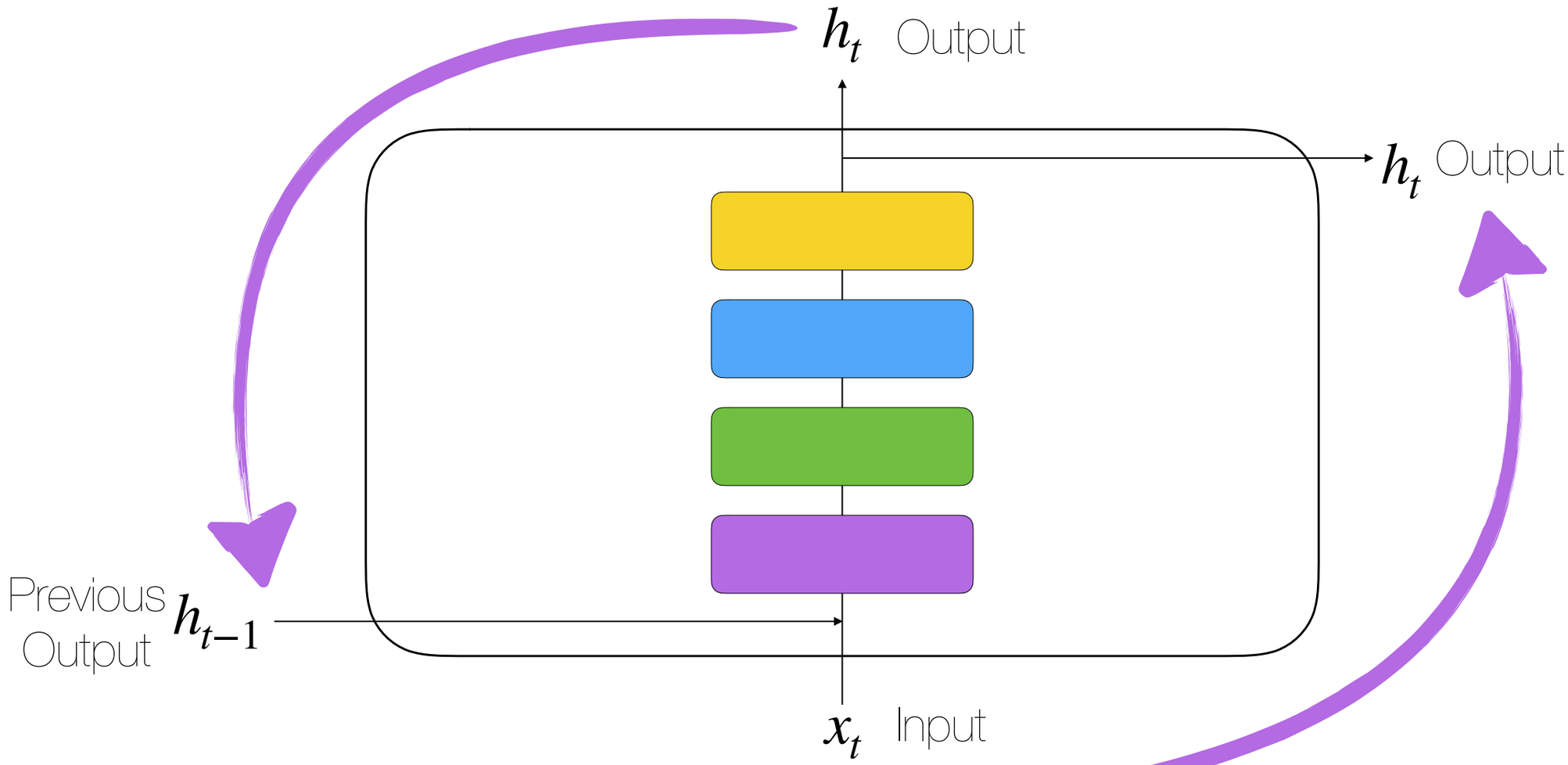
Feed Forward Networks



$$h_t = f\left(x_t\right)$$

Recurrent Neural Network (RNN)

Information Flow

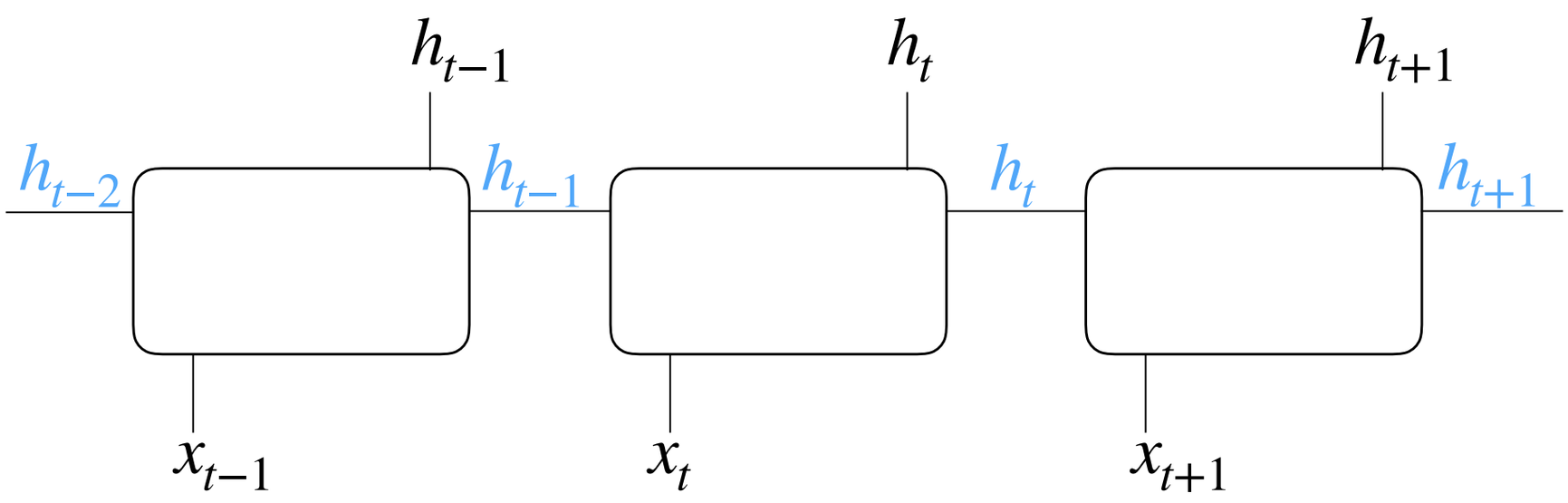


$$h_t = f\left(x_t, h_{t-1}\right)$$

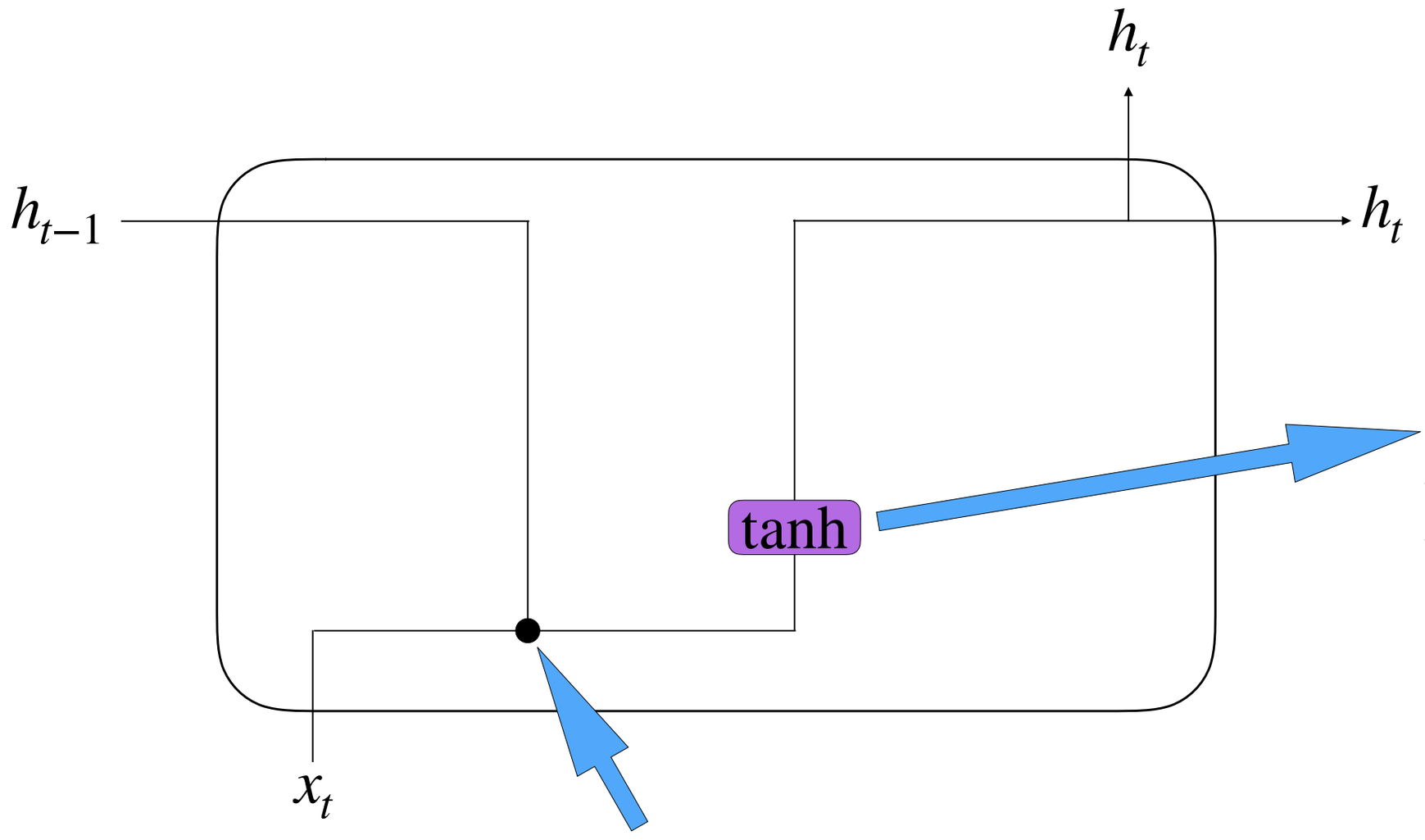
Recurrent Neural Network (RNN)

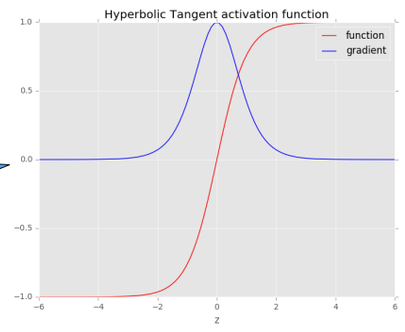
- Each output depends (implicitly) on all previous outputs.

- Input sequences generate output sequences (seq2seq)



Recurrent Neural Network (RNN)

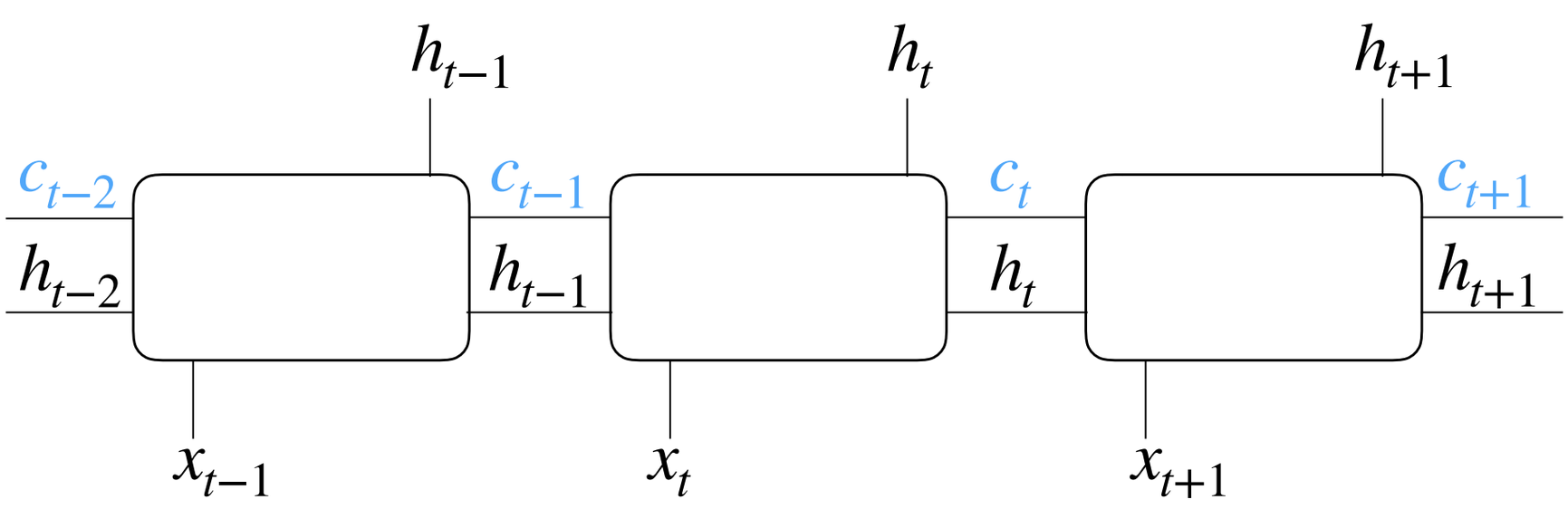




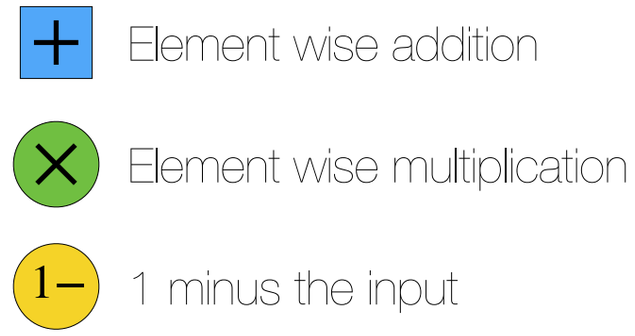
$$h_t = \tanh\left(Wh_{t-1} + Ux_t\right)$$

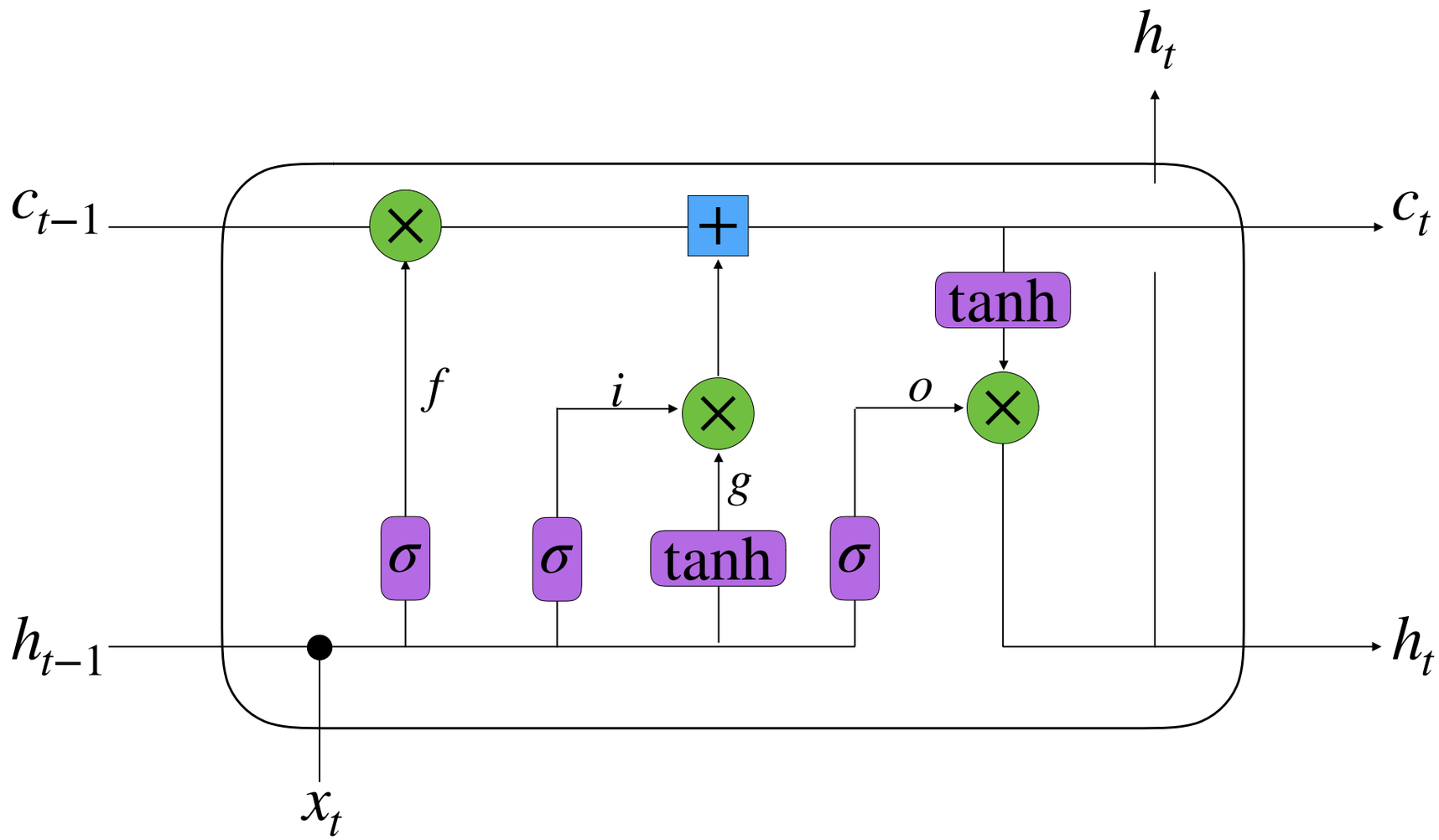
Long-Short Term Memory (LSTM)

- 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



Long-Short Term Memory (LSTM)





$$f = \sigma\left(W_f h_{t-1} + U_f x_t\right) \qquad g = \tanh\left(W_g h_{t-1} + U_g x_t\right)$$

$$i = \sigma\left(W_i h_{t-1} + U_i x_t\right) \qquad c_t = \left(c_{t-1} \otimes f\right) + \left(g \otimes i\right)$$

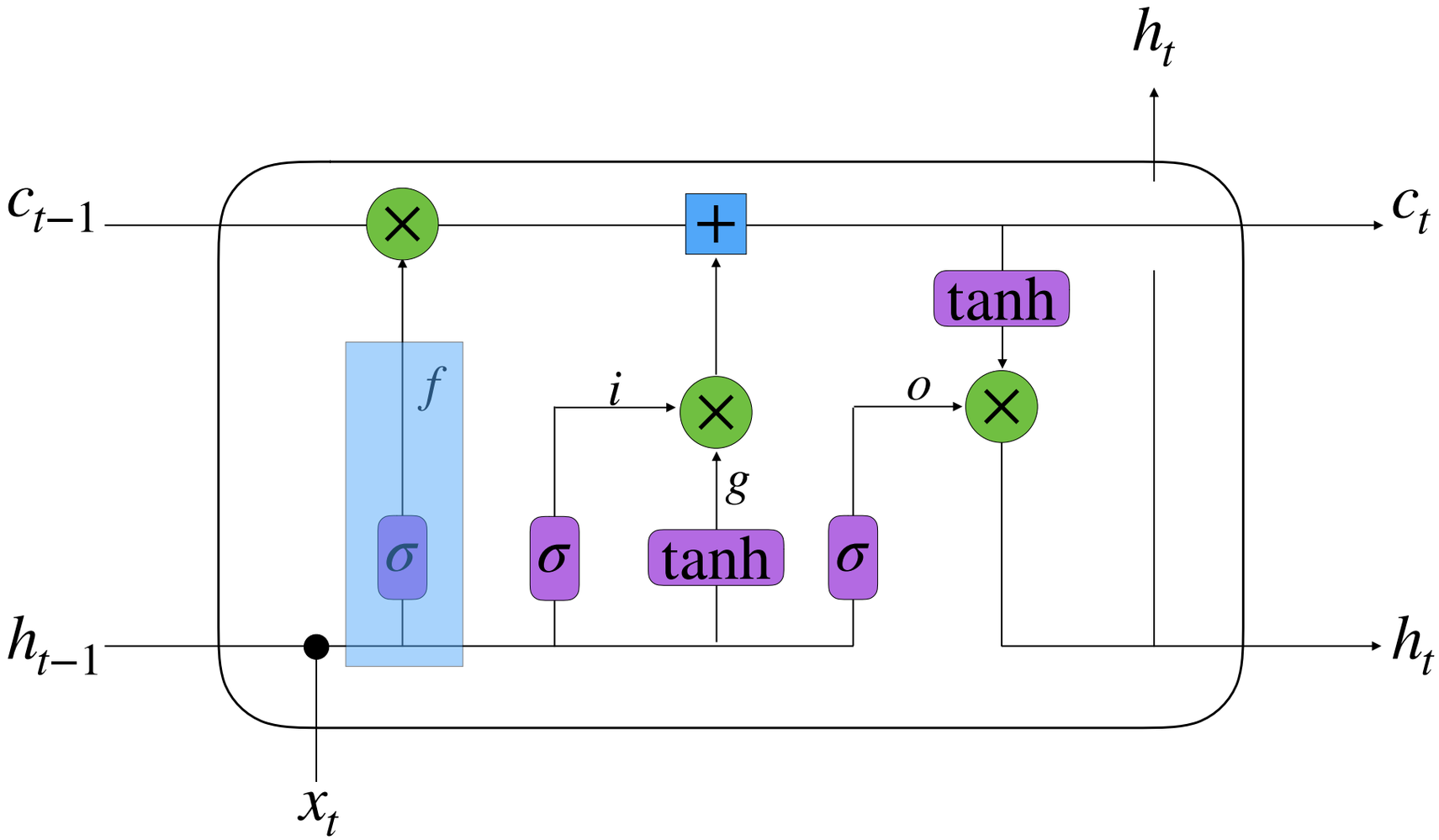
$$o = \sigma\left(W_o h_{t-1} + U_o x_t\right) \qquad h_t = \tanh\left(c_t\right) \otimes o$$

Long-Short Term Memory (LSTM)

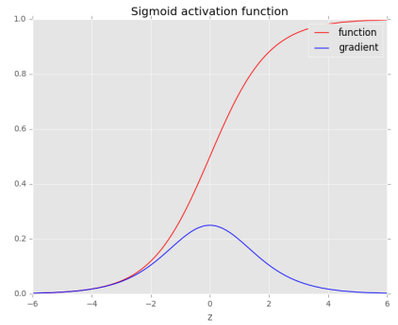
Element wise addition

Element wise multiplication

1− 1 minus the input



Forget gate: How much of the previous state should be kept?



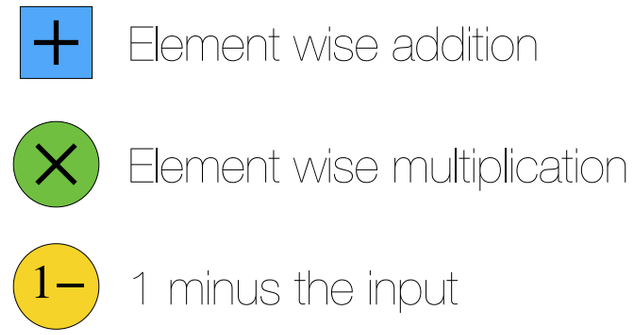
$$f = \sigma\left(W_f h_{t-1} + U_f x_t\right) \qquad g = \tanh\left(W_g h_{t-1} + U_g x_t\right)$$

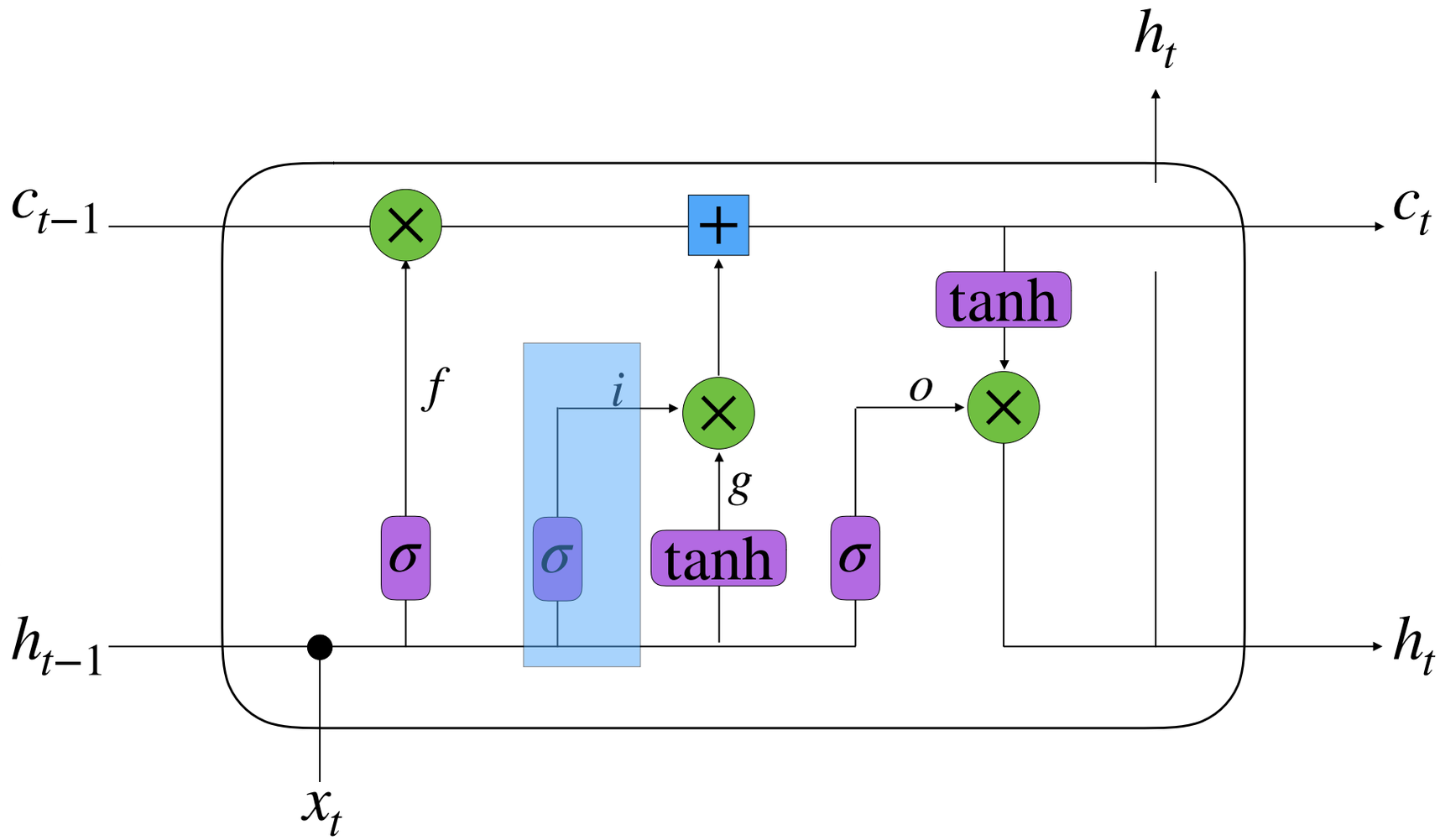
$$i = \sigma\left(W_i h_{t-1} + U_i x_t\right) \qquad c_t = \left(c_{t-1} \otimes f\right) + \left(g \otimes i\right)$$

$$o = \sigma\left(W_o h_{t-1} + U_o x_t\right) \qquad h_t = \tanh\left(c_t\right) \otimes o$$

www.bgoncalves.com

Long-Short Term Memory (LSTM)





Input gate: How much of the previous output should be remembered?

$$f = \sigma\left(W_f h_{t-1} + U_f x_t\right) \qquad g = \tanh\left(W_g h_{t-1} + U_g x_t\right)$$

$$i = \sigma\left(W_i h_{t-1} + U_i x_t\right) \qquad c_t = \left(c_{t-1} \otimes f\right) + \left(g \otimes i\right)$$

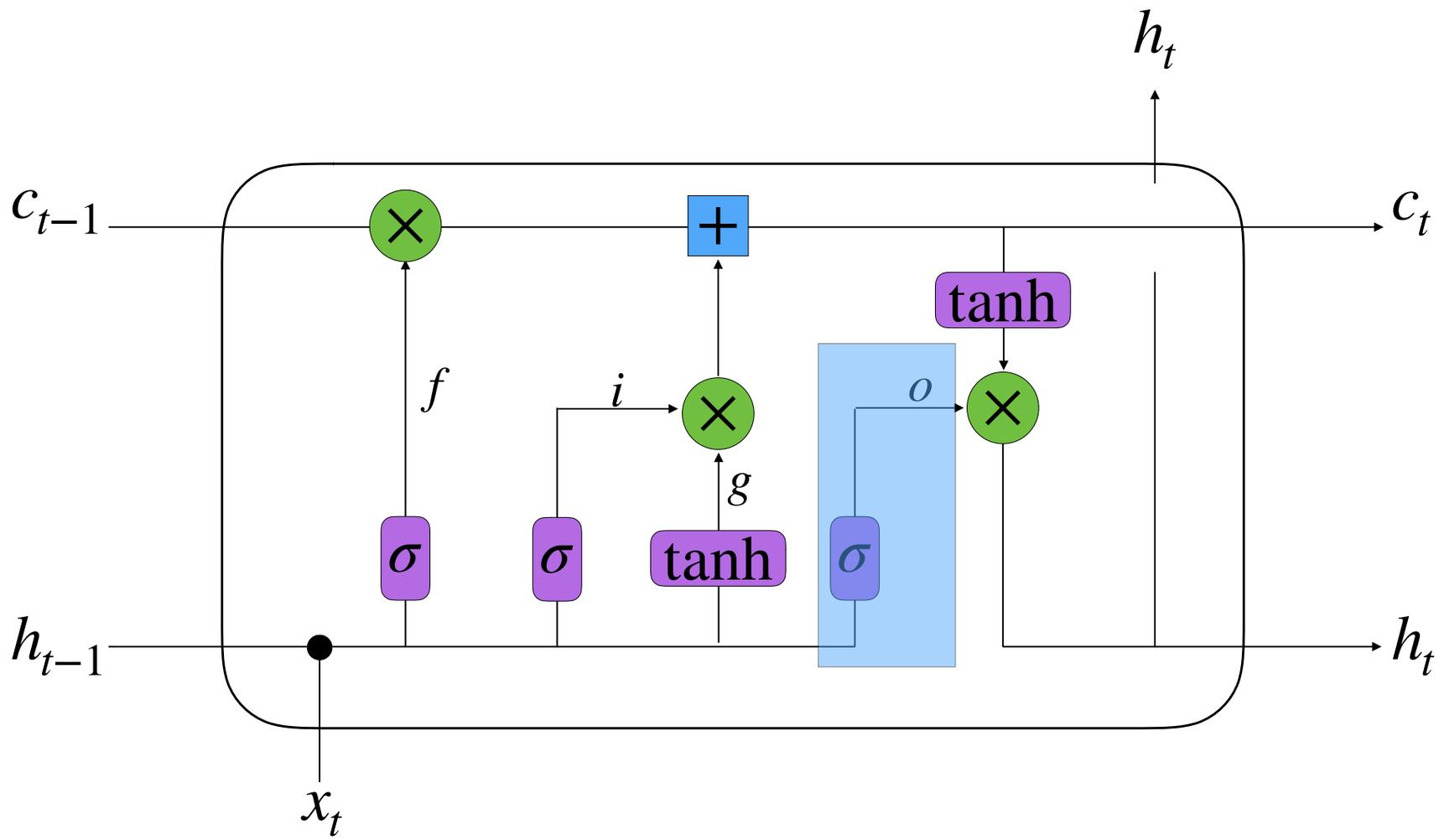
$$o = \sigma\left(W_o h_{t-1} + U_o x_t\right) \qquad h_t = \tanh\left(c_t\right) \otimes o$$

Long-Short Term Memory (LSTM)

Element wise addition

Element wise multiplication

1– 1 minus the input



Output gate: How much of the previous output should contribute?

All gates use the same inputs and activation functions, but different weights

$$f = \sigma\left(W_f h_{t-1} + U_f x_t\right) \qquad g = \tanh\left(W_g h_{t-1} + U_g x_t\right)$$

$$i = \sigma\left(W_i h_{t-1} + U_i x_t\right) \qquad c_t = \left(c_{t-1} \otimes f\right) + \left(g \otimes i\right)$$

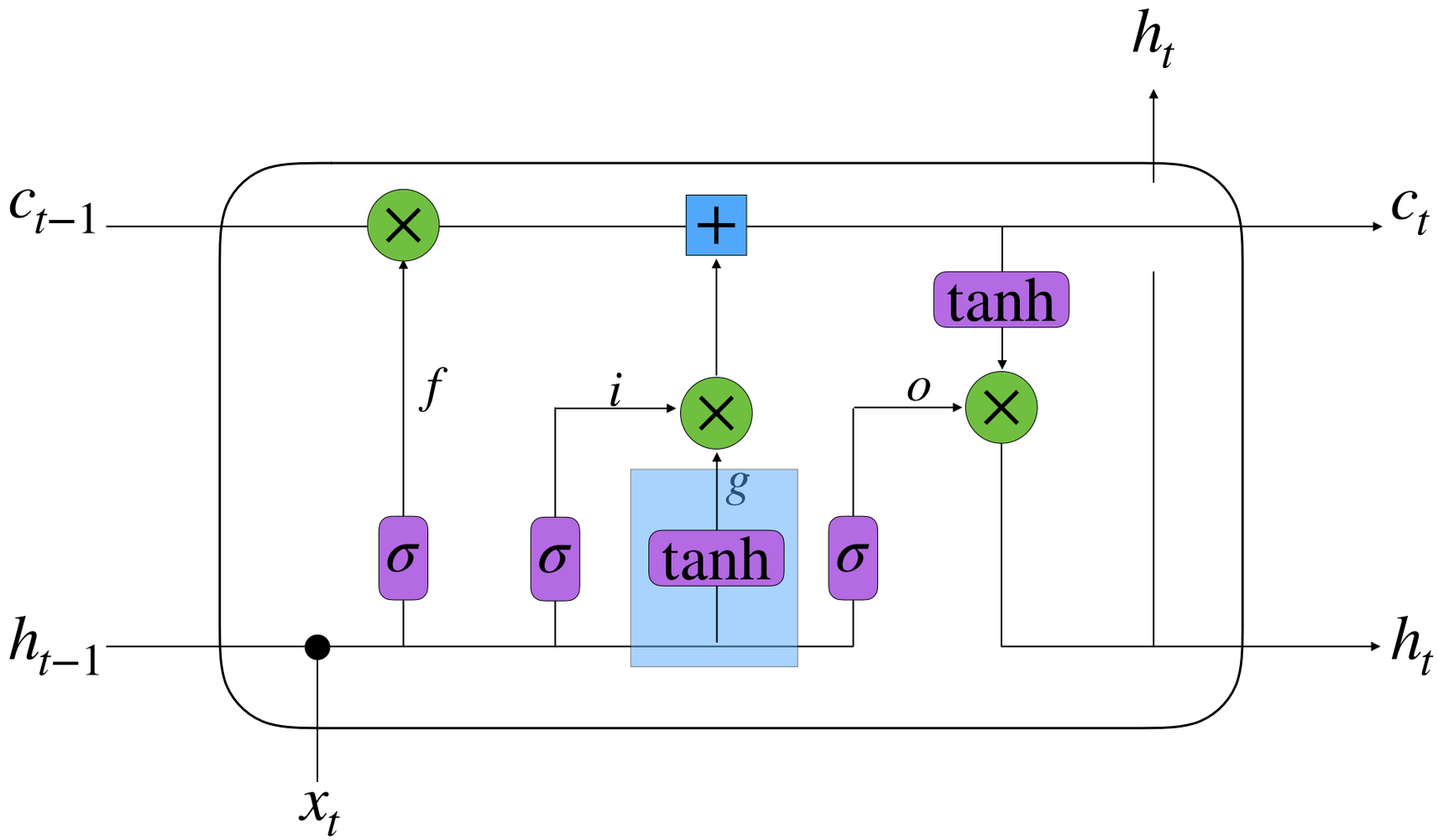
$$o = \sigma\left(W_o h_{t-1} + U_o x_t\right) \qquad h_t = \tanh\left(c_t\right) \otimes o$$

Long-Short Term Memory (LSTM)

Element wise addition

Element wise multiplication

1– 1 minus the input



Output gate: How much of the previous output should contribute?

$$f = \sigma\left(W_f h_{t-1} + U_f x_t\right) \qquad g = \tanh\left(W_g h_{t-1} + U_g x_t\right)$$

$$i = \sigma\left(W_i h_{t-1} + U_i x_t\right) \qquad c_t = \left(c_{t-1} \otimes f\right) + \left(g \otimes i\right)$$

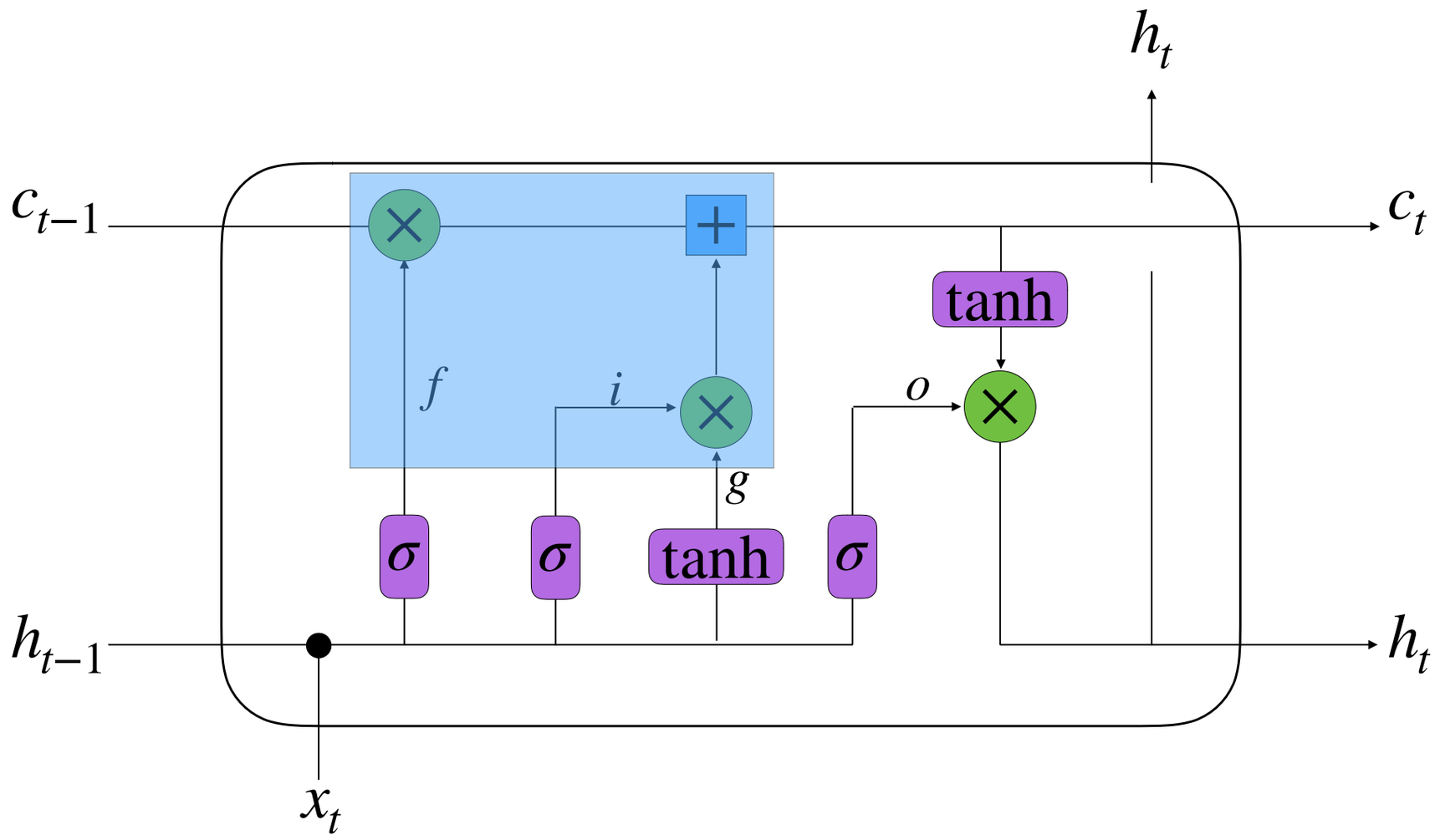
$$o = \sigma\left(W_o h_{t-1} + U_o x_t\right) \qquad h_t = \tanh\left(c_t\right) \otimes o$$

Long-Short Term Memory (LSTM)

Element wise addition

Element wise multiplication

1– 1 minus the input



State: Update the current state

$$f = \sigma\left(W_f h_{t-1} + U_f x_t\right) \qquad g = \tanh\left(W_g h_{t-1} + U_g x_t\right)$$

$$i = \sigma\left(W_i h_{t-1} + U_i x_t\right) \qquad c_t = \left(c_{t-1} \otimes f\right) + \left(g \otimes i\right)$$

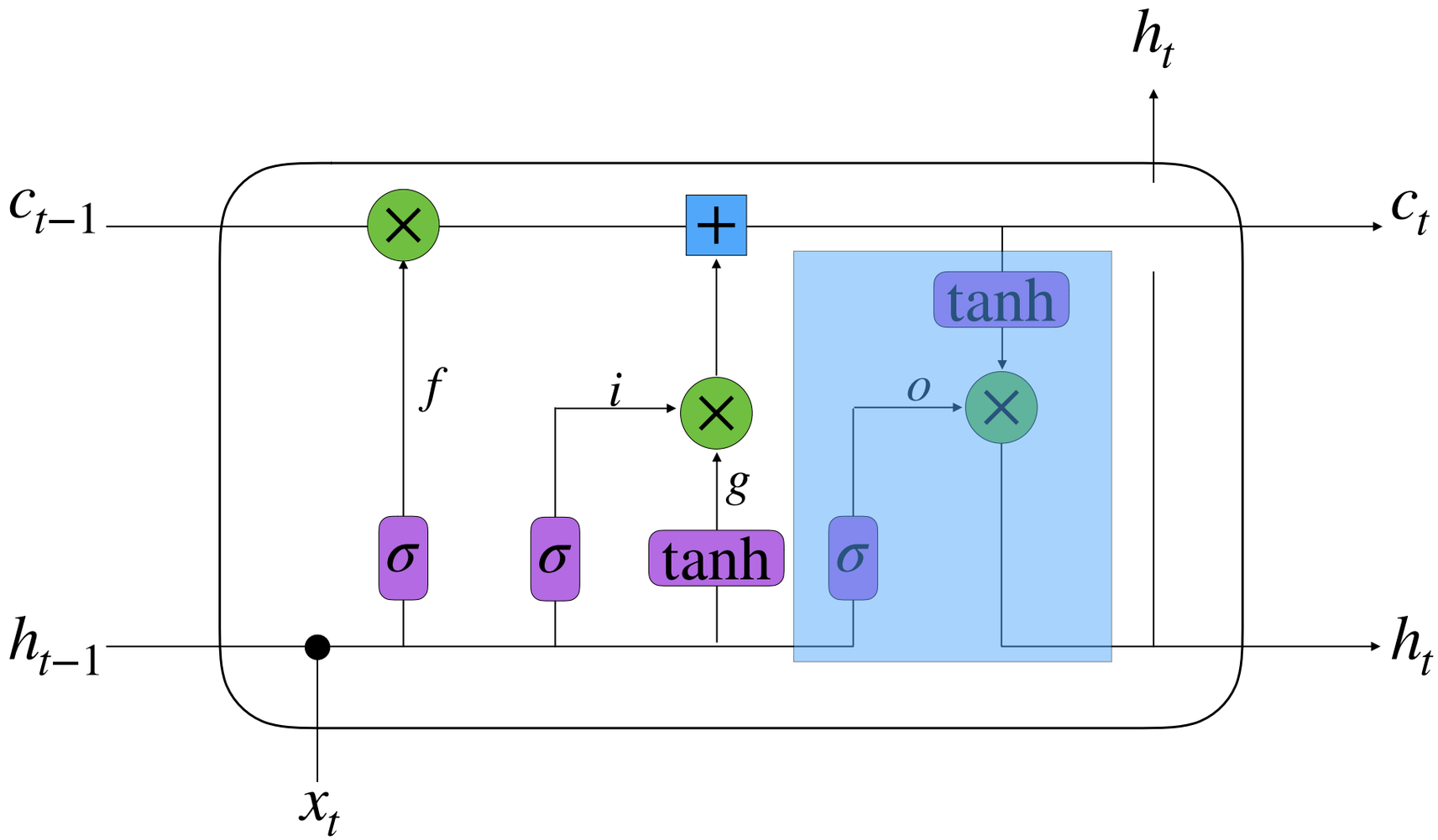
$$o = \sigma\left(W_o h_{t-1} + U_o x_t\right) \qquad h_t = \tanh\left(c_t\right) \otimes o$$

Long-Short Term Memory (LSTM)

Element wise addition

Element wise multiplication

1− 1 minus the input



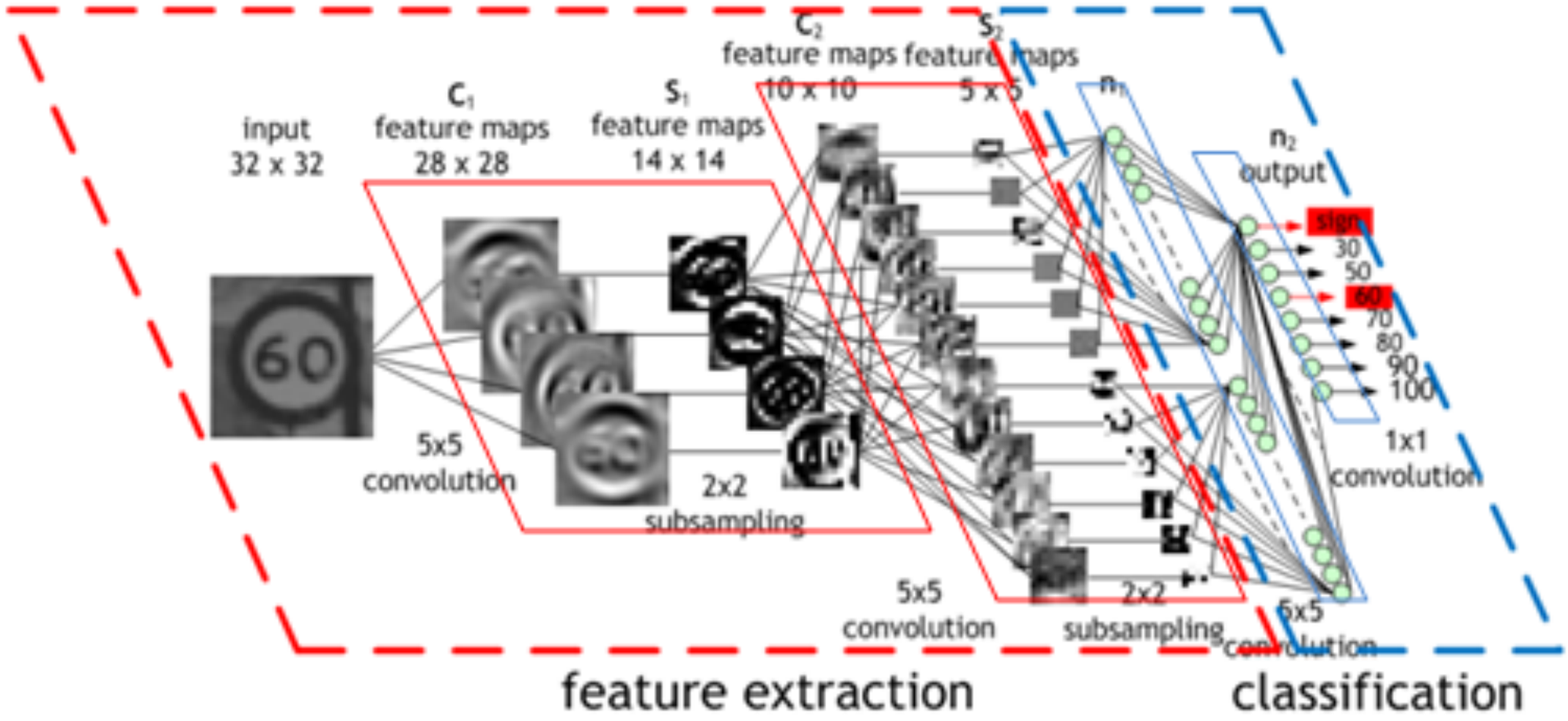
Output: Combine all available information.

$$f = \sigma\left(W_f h_{t-1} + U_f x_t\right) \qquad g = \tanh\left(W_g h_{t-1} + U_g x_t\right)$$

$$i = \sigma\left(W_i h_{t-1} + U_i x_t\right) \qquad c_t = \left(c_{t-1} \otimes f\right) + \left(g \otimes i\right)$$

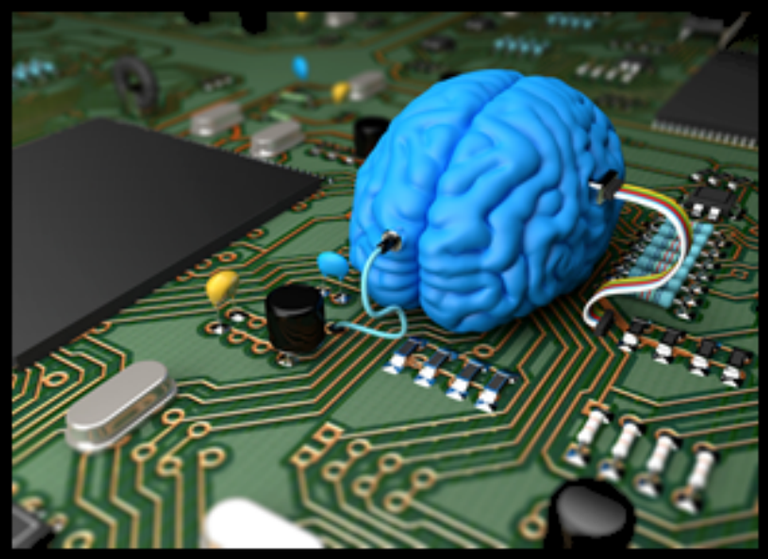
$$o = \sigma\left(W_o h_{t-1} + U_o x_t\right) \qquad h_t = \tanh\left(c_t\right) \otimes o$$

Convolutional Neural Networks



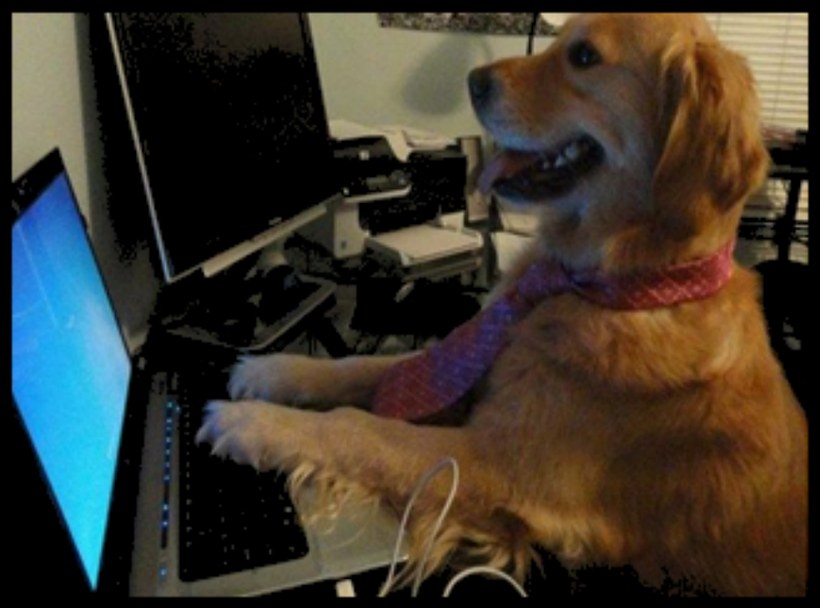
Deep Learning

What society thinks I do



What my friends think I do

What other computer scientists think I do



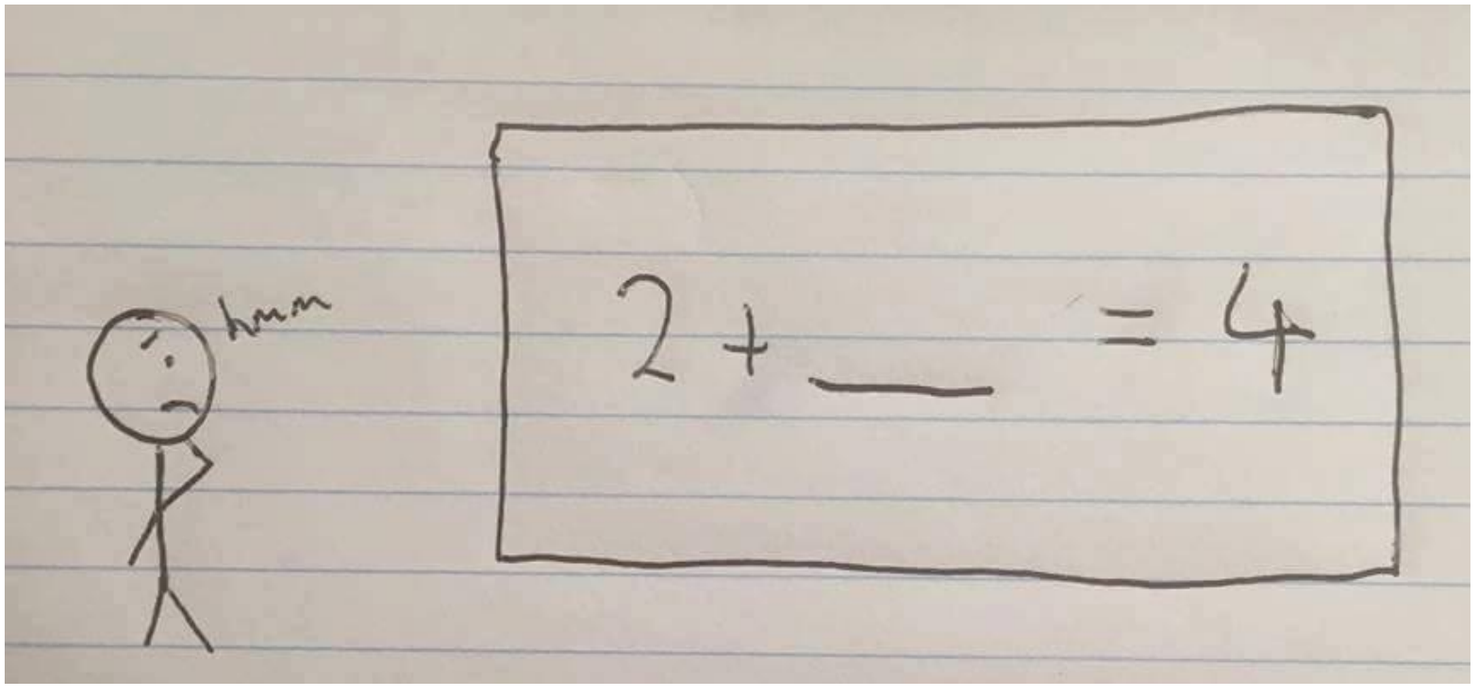
What mathematicians think I do

What I think I do

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

What I actually do

Interpretability



"Deep" learning