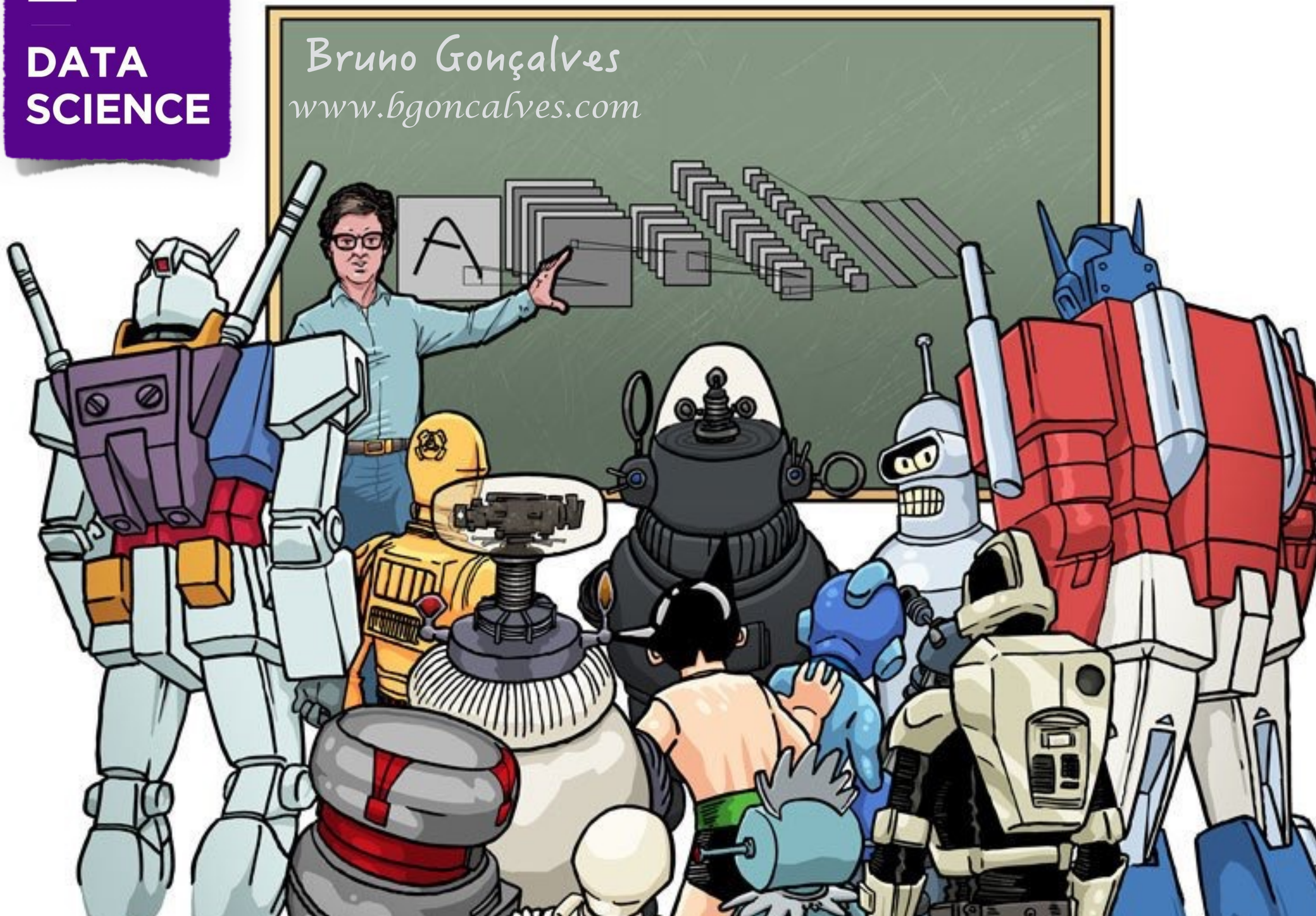
NYU

DATA SCIENCE

Machine(s) Learning and Data Science

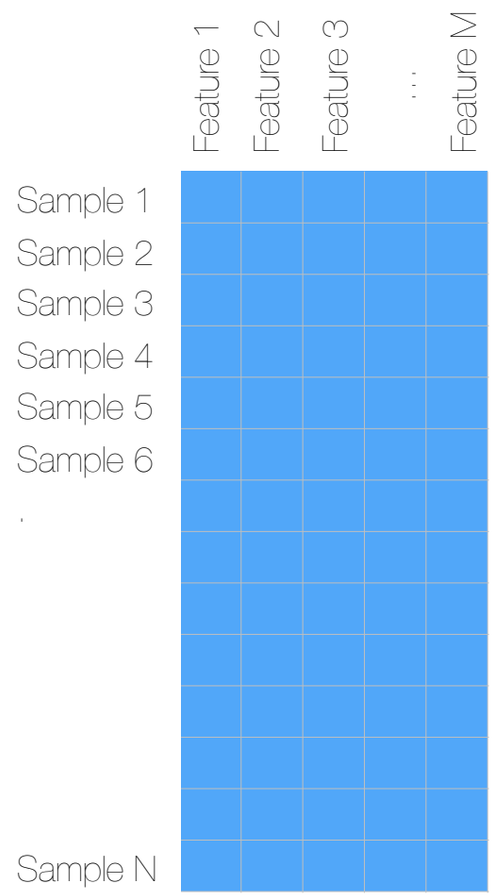
Bruno Gonçalves
www.bgoncalves.com



Supervised Learning

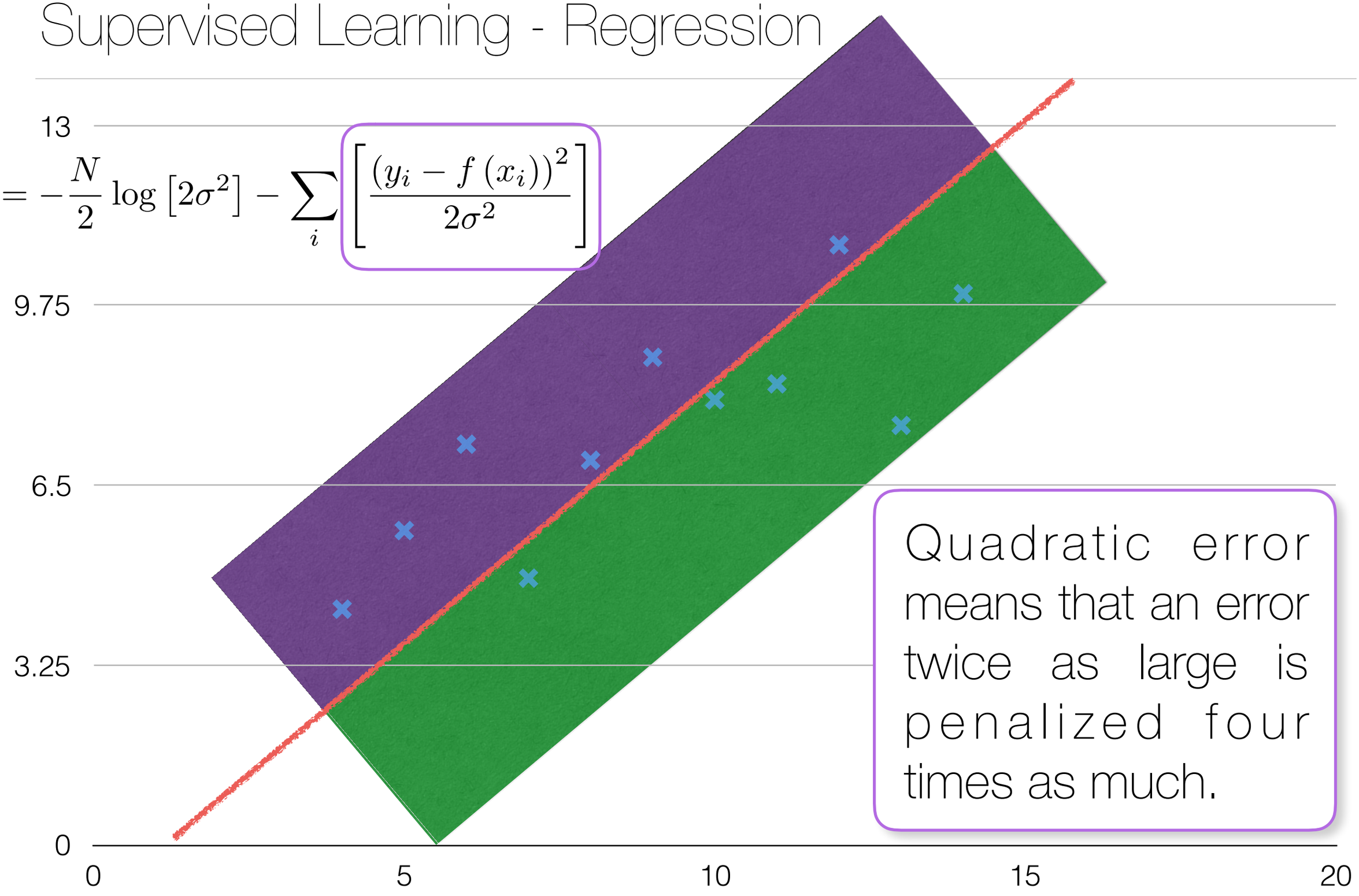
Supervised Learning - Regression

- Two fundamental types of problems:
  - Classification (discrete output value)
  - Regression (continuous output value)
- Dataset formatted as an NxM matrix of N samples and M features



Supervised Learning - Regression

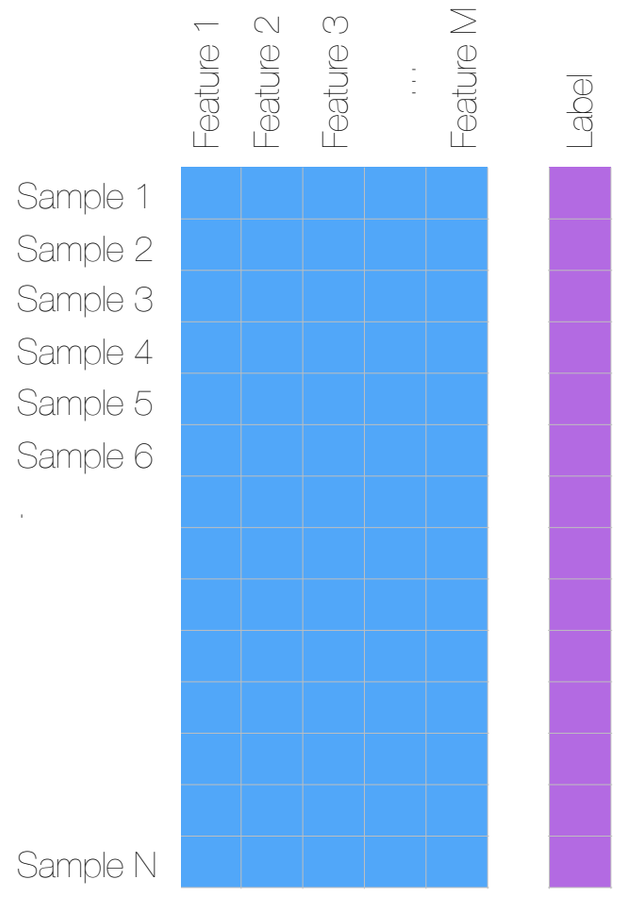
$$\mathcal{L} = -\frac{N}{2}\log\left[2\sigma^2\right] - \sum_i \left[\frac{\left(y_i - f\left(x_i\right)\right)^2}{2\sigma^2}\right]$$



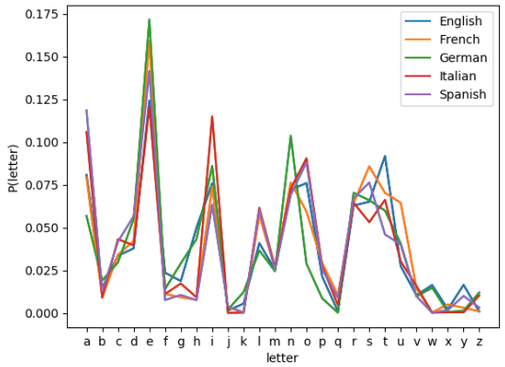
Quadratic error means that an error twice as large is penalized four times as much.

Supervised Learning - Classification

- Two fundamental types of problems:
  - Classification (discrete output value)
  - Regression (continuous 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
  - K-Nearest Neighbor
  - Support Vector Machine
  - Neural Networks



Language Detection



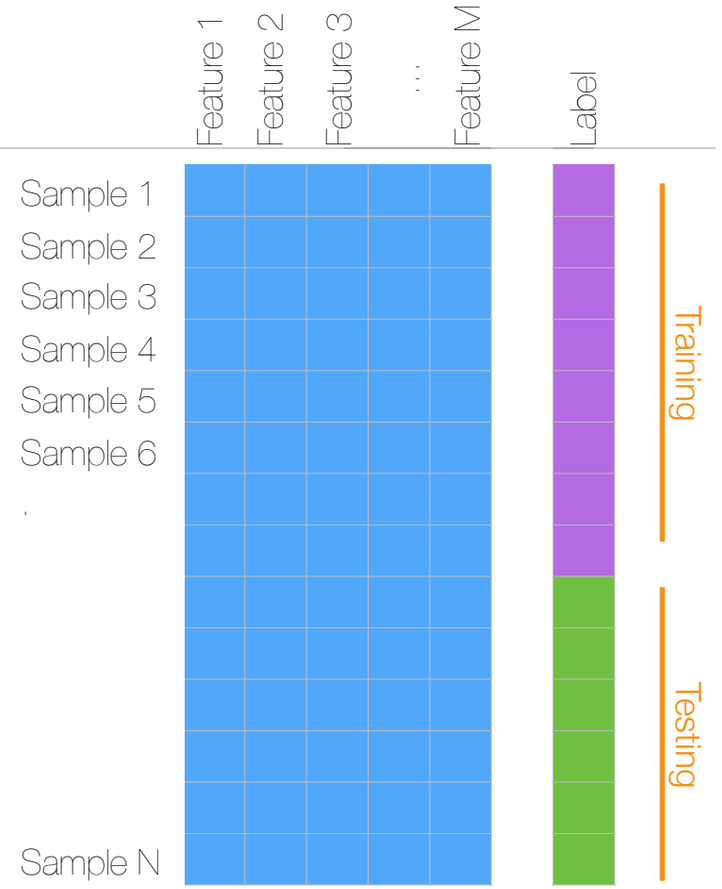

```python
import pandas as pd
import numpy as np
from collections import Counter

def load_data():
    P_letter_lang = pd.read_csv('table_langs.dat', sep=' ', header=0, index_col = 0)

    langs = list(P_letter_lang.columns)

    P_letter = P_letter_lang.mean(axis=1)
    P_letter /= P_letter.sum()

    P_lang_letter = np.array(P_letter_lang)/(P_letter_lang.shape[1]*P_letter.T[:,None])

    L_lang_letter = np.log(P_lang_letter.T)

    return langs, P_letter, L_lang_letter

def detect_lang(langs, P_letter, L_lang_letter, text):
    counts = np.zeros(26, dtype='int')
    pos = dict(zip(P_letter.index, range(26)))

    text_counts =  Counter(text).items()

    for letter, count in text_counts:
        if letter in pos:
            counts[pos[letter]] += count

    L_text = np.dot(L_lang_letter, counts)
    index = np.argmax(L_text)
    lang_text = langs[index]
    prob = np.exp(L_text[index])/np.sum(np.exp(L_text))*100

    return lang_text, prob, L_text
```

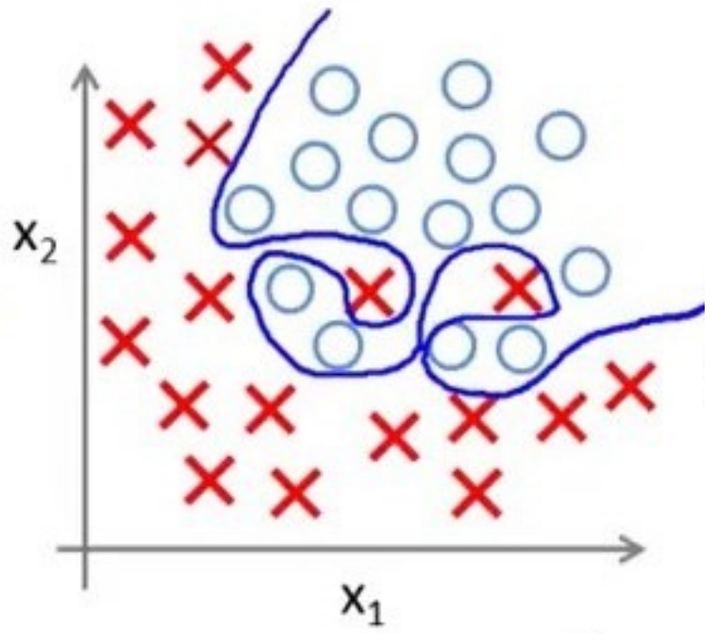
$$P\left(language|text\right) = \frac{\exp\left[\mathcal{L}\left(language|text\right)\right]}{\sum_{language}\exp\left[\mathcal{L}\left(language|text\right)\right]}$$

detect.py

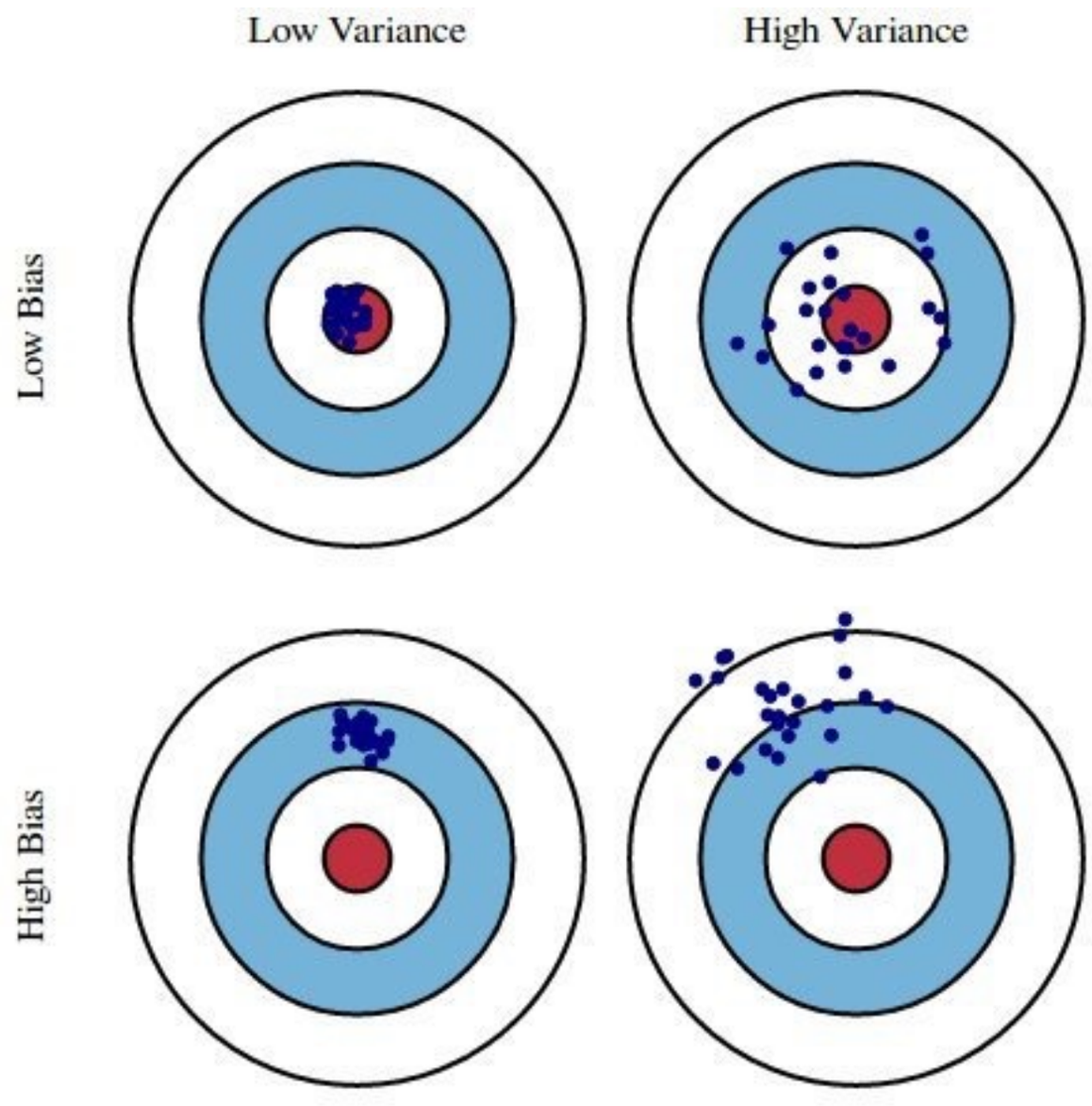
Supervised Learning - Overfitting

- "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.

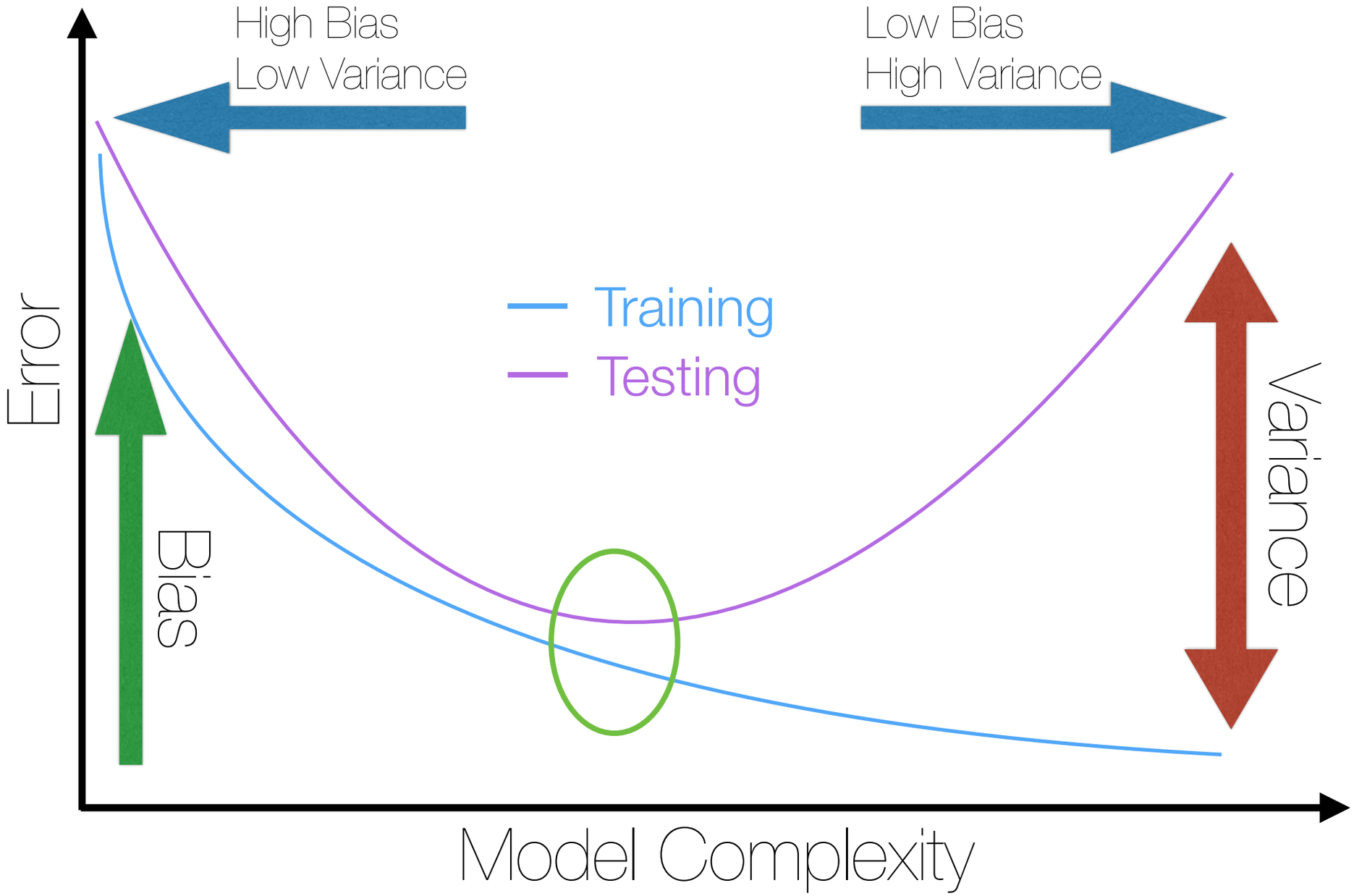




Bias-Variance Tradeoff



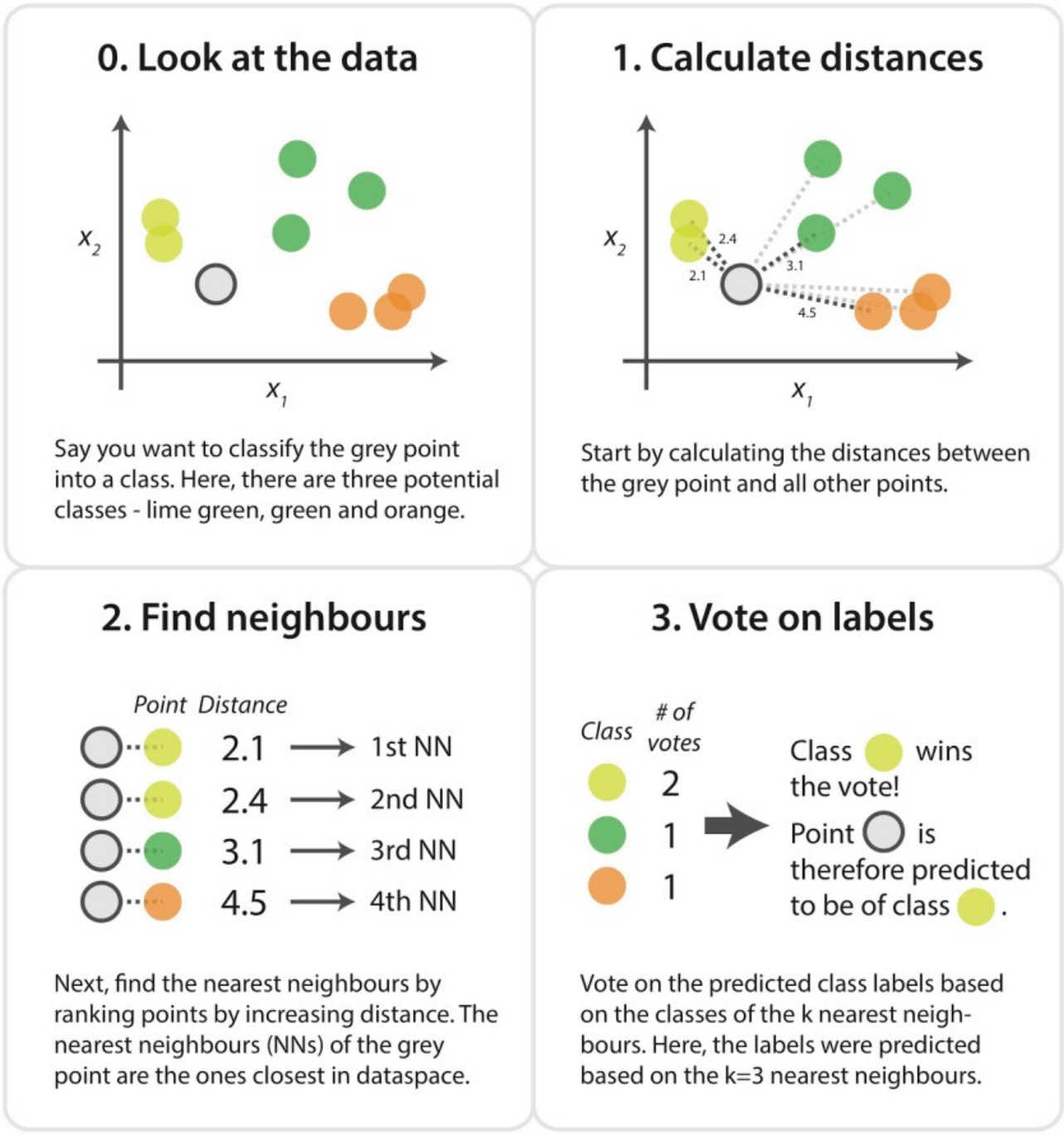
Bias-Variance Tradeoff



K-nearest neighbors

- Perhaps the simplest of supervised learning algorithms
- Effectively memorizes all previously seen data
- Intuitively takes advantage of natural data clustering
- Define that the class of any datapoint is given by the plurality of it's k nearest neighbors
- Many variations using:
    - different distance metrics,
    - weighting procedures,
    - etc…

K-nearest neighbors



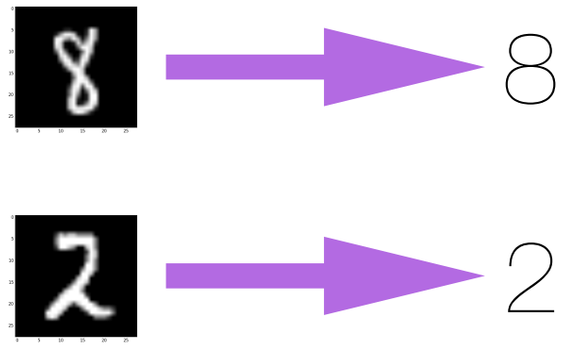
A practical example - MNIST

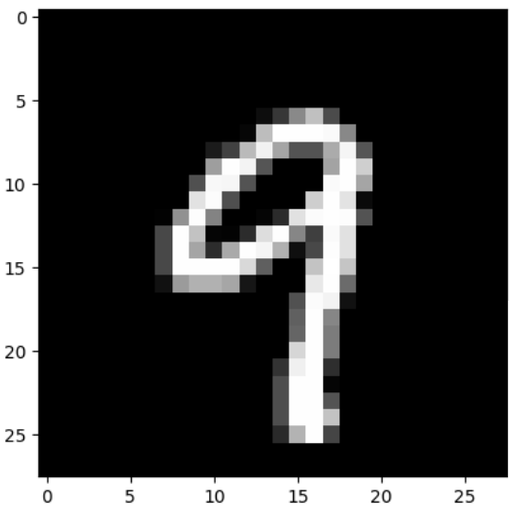
http://yann.lecun.com/exdb/mnist/

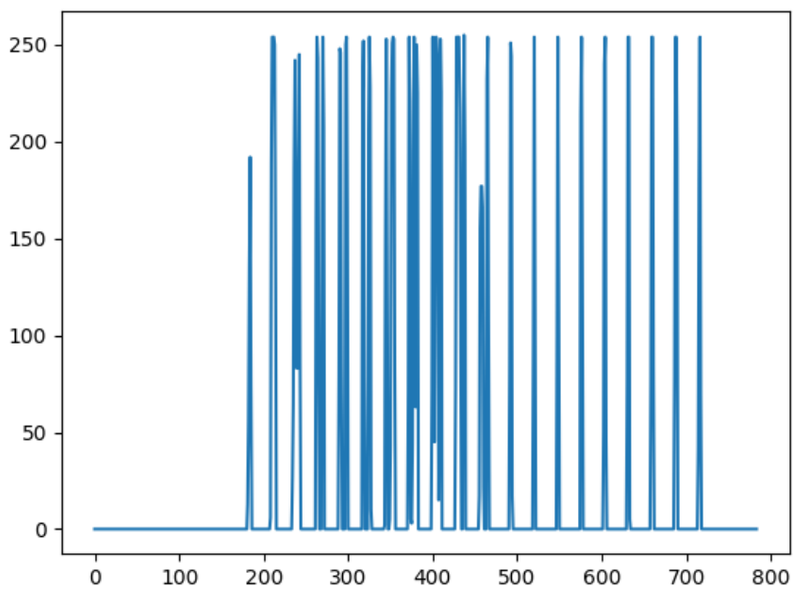
THE MNIST DATABASE

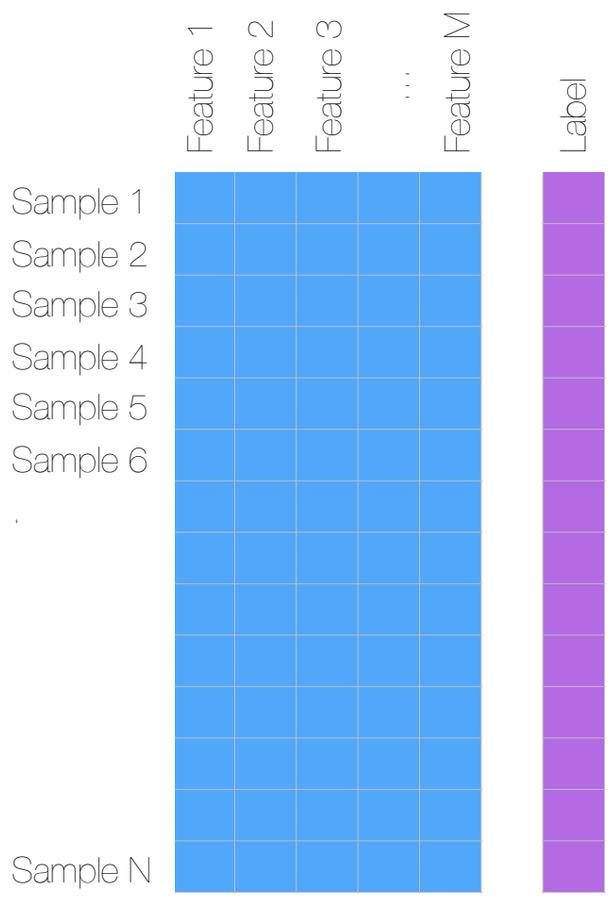
of handwritten digits

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









visualize_digits.py
convert_input.py

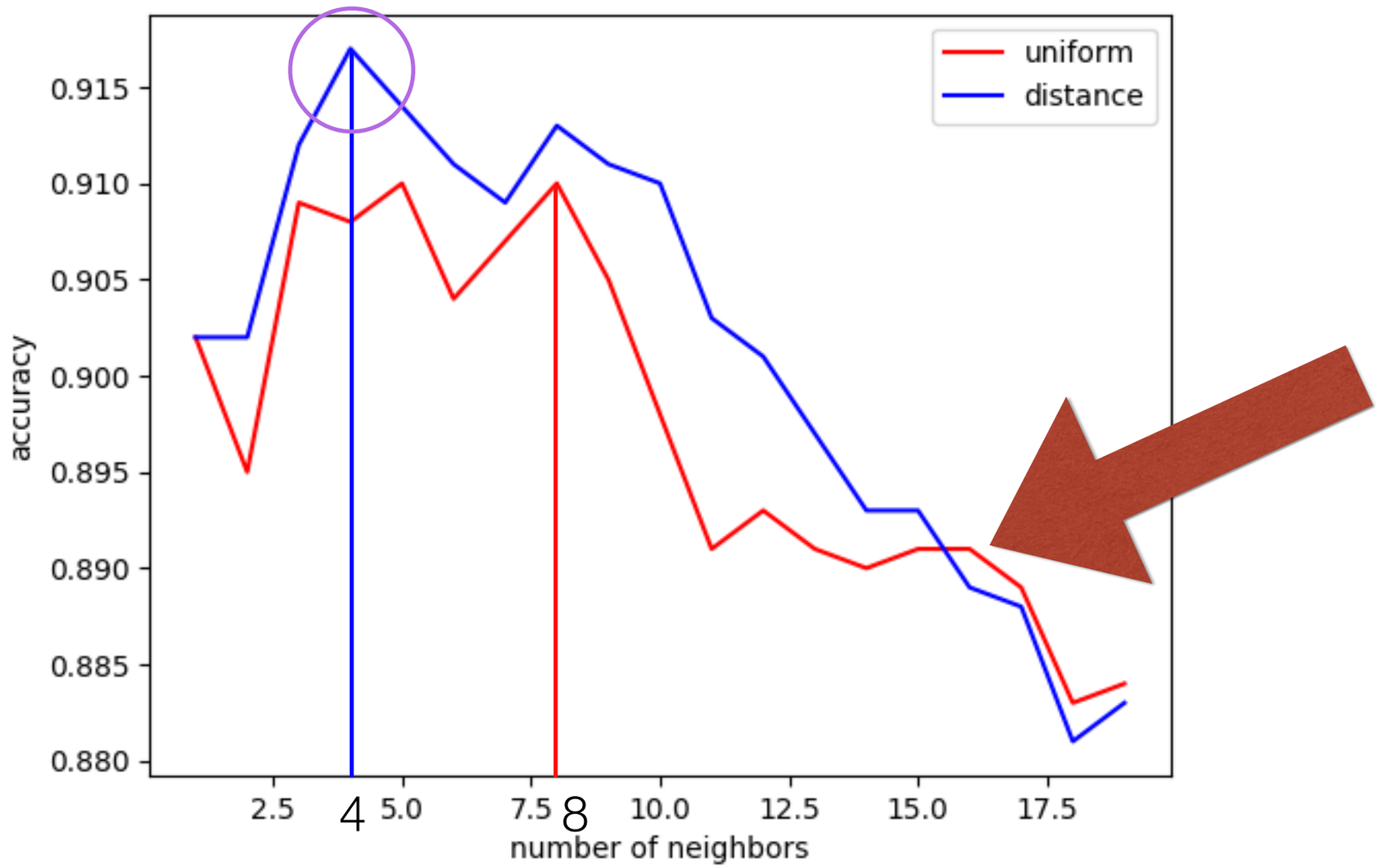
@bgoncalves

K-nearest neighbors

```python
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

X_train = np.load('input/X_train.npy')
X_test = np.load('input/X_test.npy')
y_train = np.load('input/y_train.npy')
y_test = np.load('input/y_test.npy')

input_layer_size = X_train.shape[1]

X_train /= 255.
X_test /= 255.

def accuracy(y_test, y_pred):
    return np.sum(y_test==y_pred)/len(y_test)

results_distance = []
results_uniform = []

for k in range(1, 21):
    neigh = KNeighborsClassifier(n_neighbors=k, metric='euclidean', weights='uniform')
    neigh.fit(X_train, y_train)

    y_pred = neigh.predict(X_test)

    acc = accuracy(y_test, y_pred)
    results_distance.append([k, acc])

    print(k, acc)
```

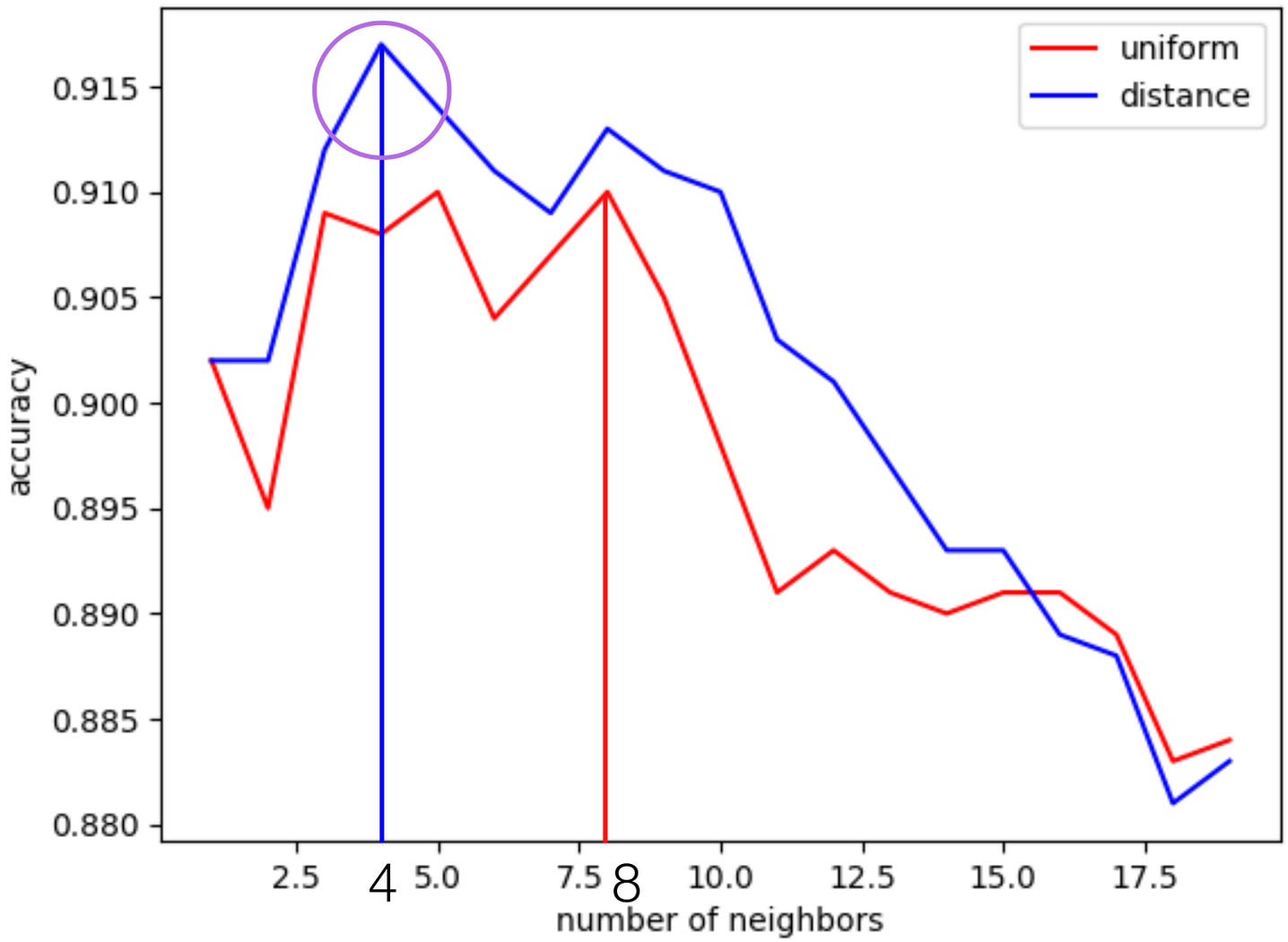
knn.py

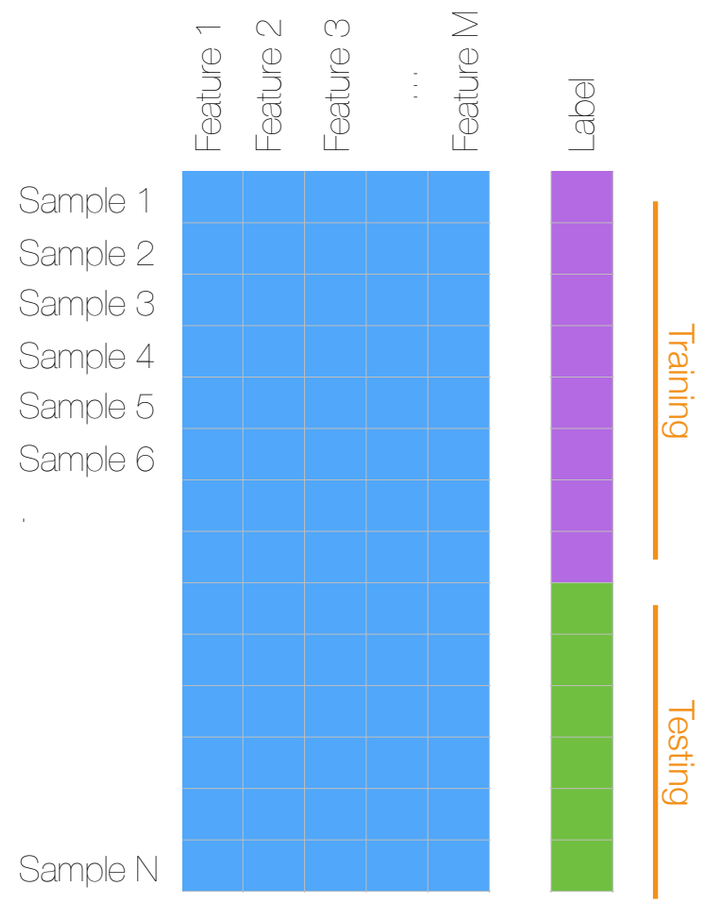
K-nearest neighbors



knn.py

K-nearest neighbors

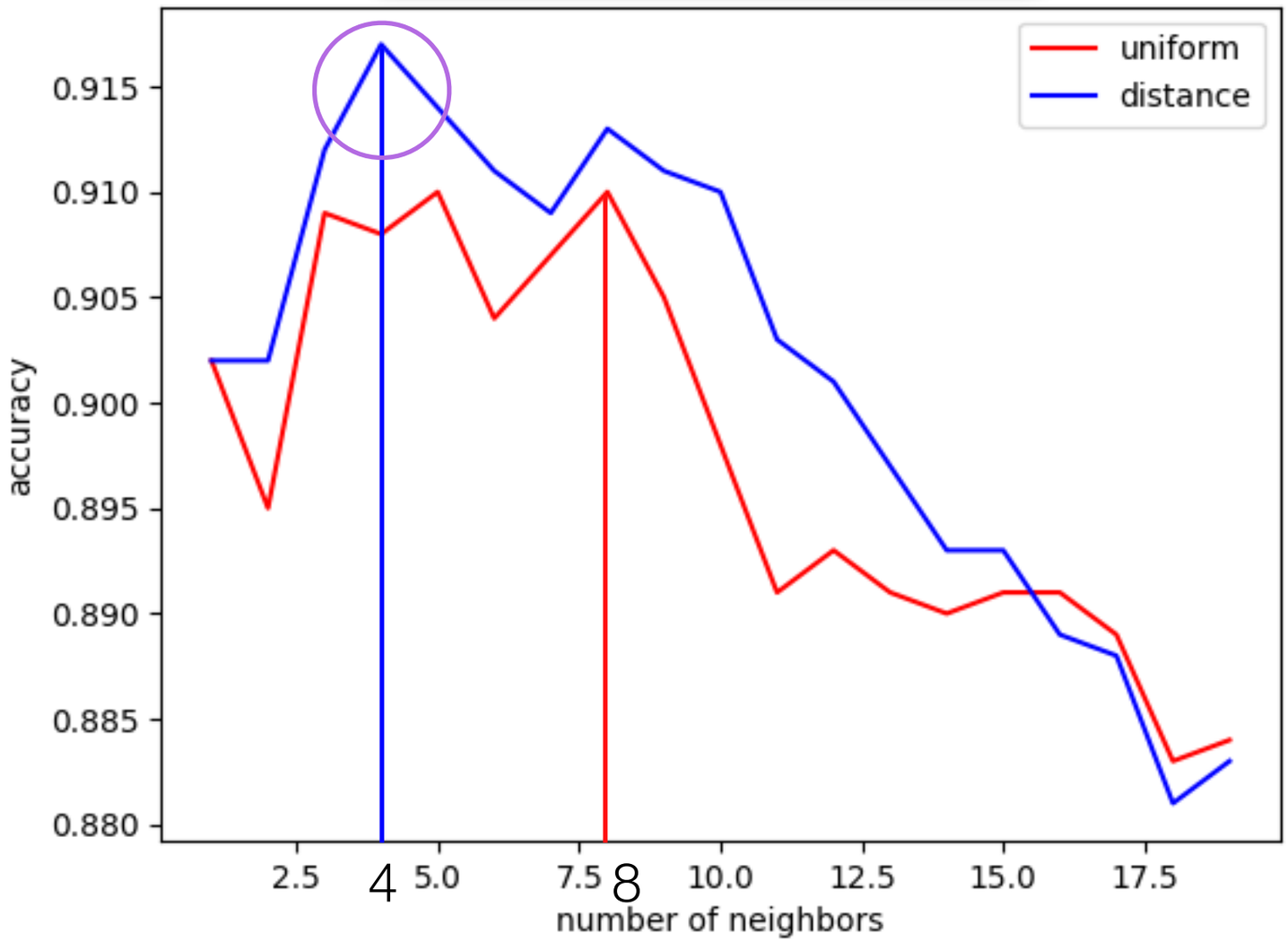


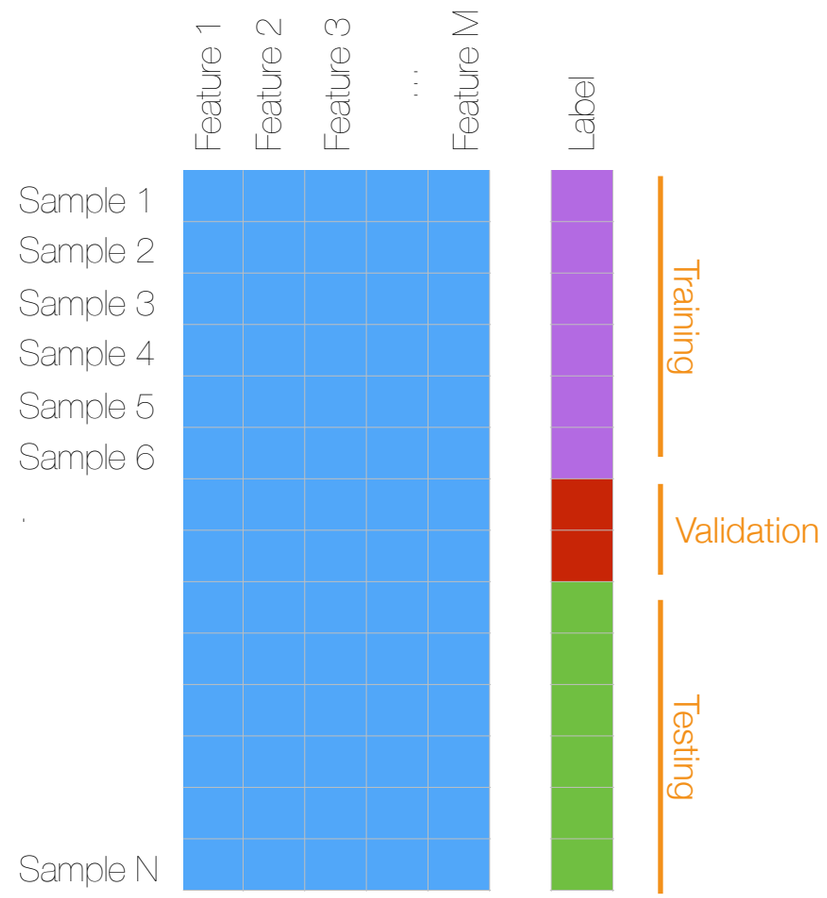


knn.py

K-nearest neighbors

Final Accuracy: 0.708!





knn.py

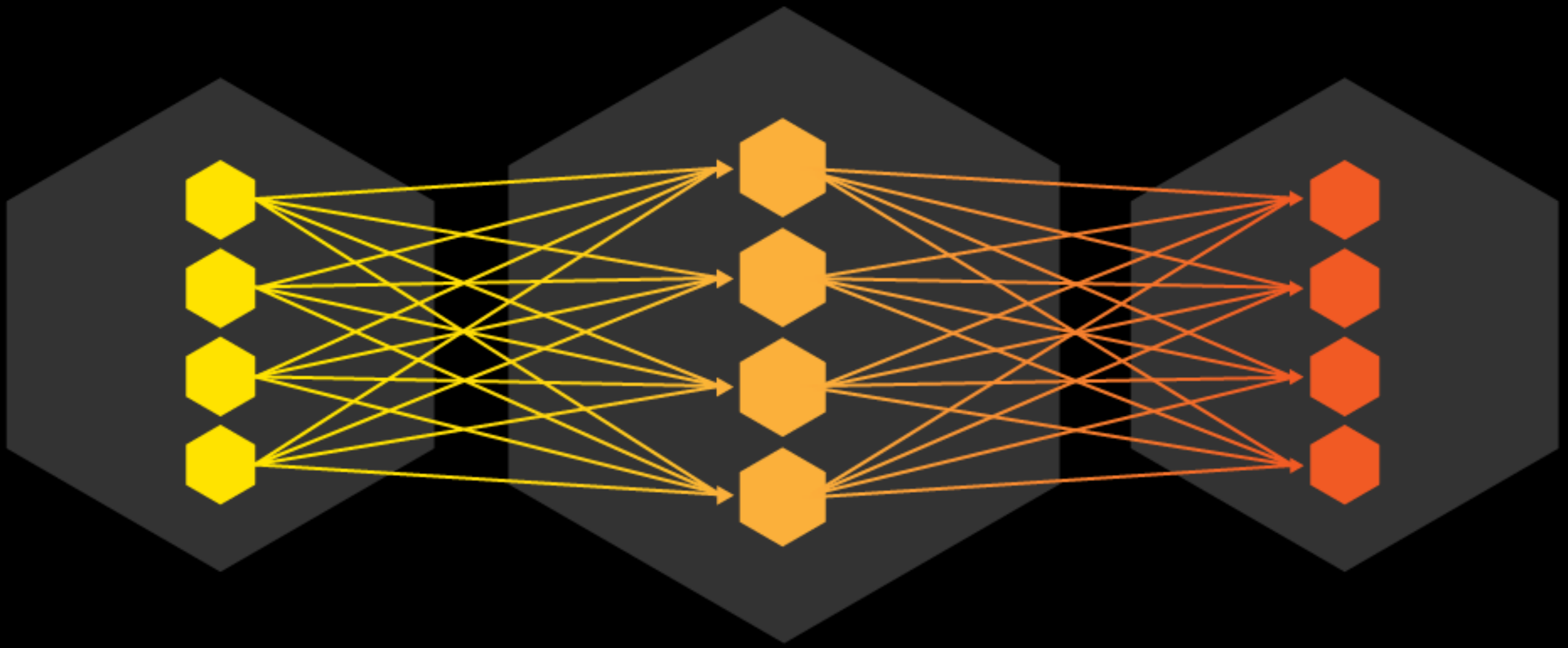
@bgoncalves

K-nea

```python
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

X_train = np.load('input/X_train.npy')
y_train = np.load('input/y_train.npy')

order = list(range(X_train.shape[0]))
np.random.shuffle(order)

X_train = X_train[order]
y_train = y_train[order]

X_test = np.load('input/X_test.npy')
y_test = np.load('input/y_test.npy')

order = list(range(X_test.shape[0]))
np.random.shuffle(order)

X_test = X_test[order]
y_test = y_test[order]

# Use 1000 points from the training set for the validation set
X_validate = X_train[:1000]
X_train = X_train[1000:]

y_validate = y_train[:1000]
y_train = y_train[1000:]

(…)

neigh = KNeighborsClassifier(n_neighbors=4, metric='euclidean', weights='distance')
neigh.fit(X_train, y_train)

y_pred = neigh.predict(X_validate)
acc = accuracy(y_validate, y_pred)

print(acc)
```

Using a Validation Set
prevents
Information Leakage

knn.py



INPUT TERMS

FEATURES
PREDICTIONS
ATTRIBUTES
PREDICTABLE VARIABLES

MACHINE

ALGORITHMS
TECHNIQUES
MODELS

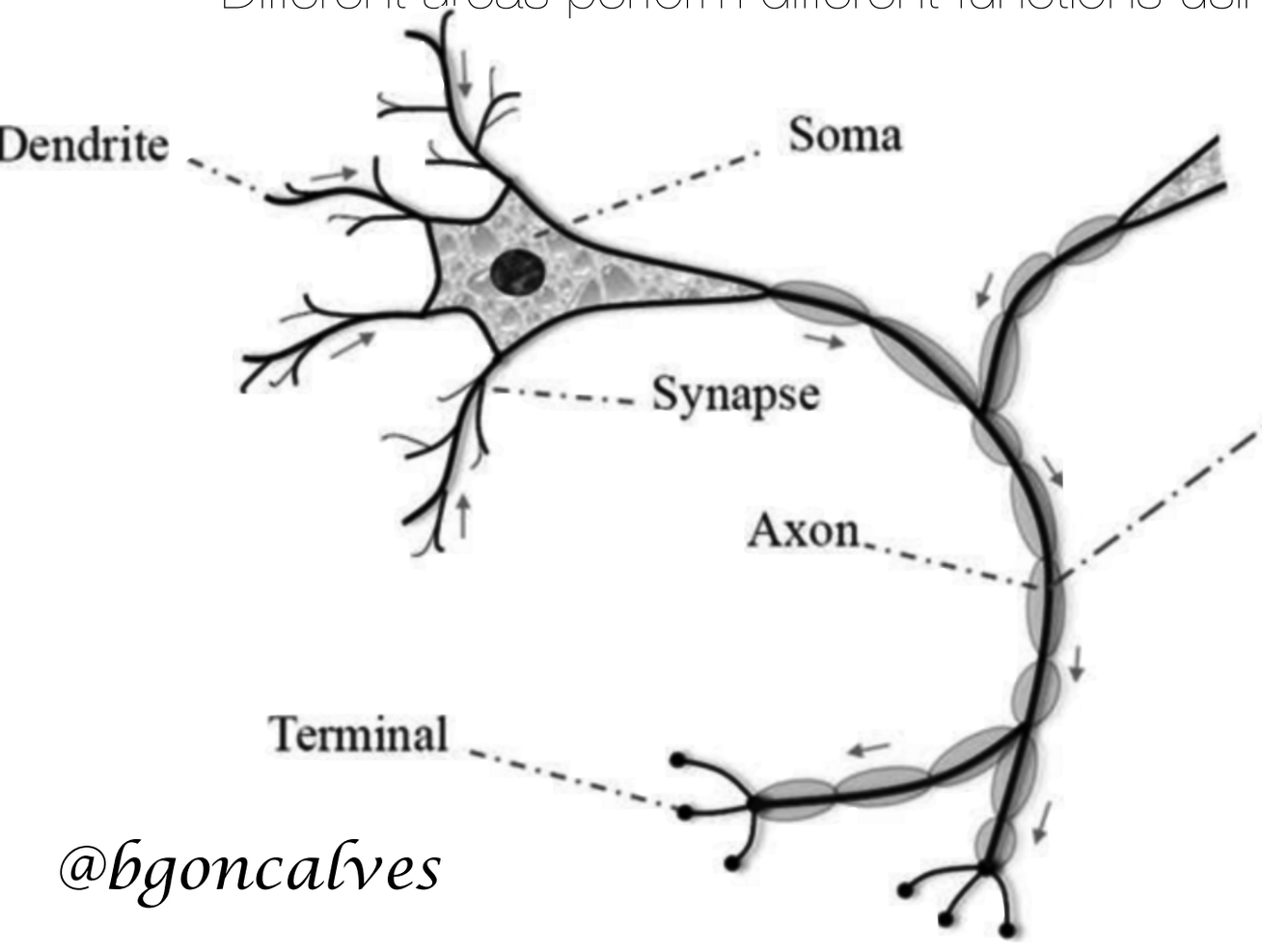
OUTPUT TERMS

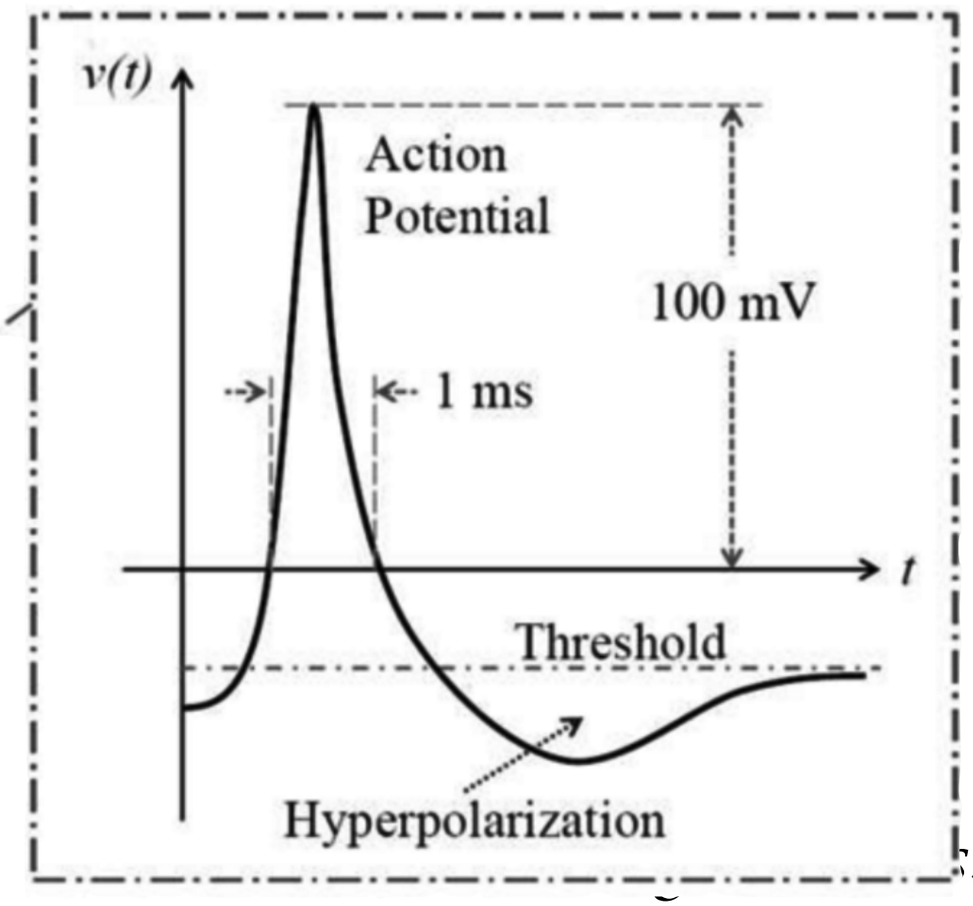
CLASSES
RESPONSES
TARGETS
DEPENDANT VARIABLES

How the Brain "Works" (Cartoon version)

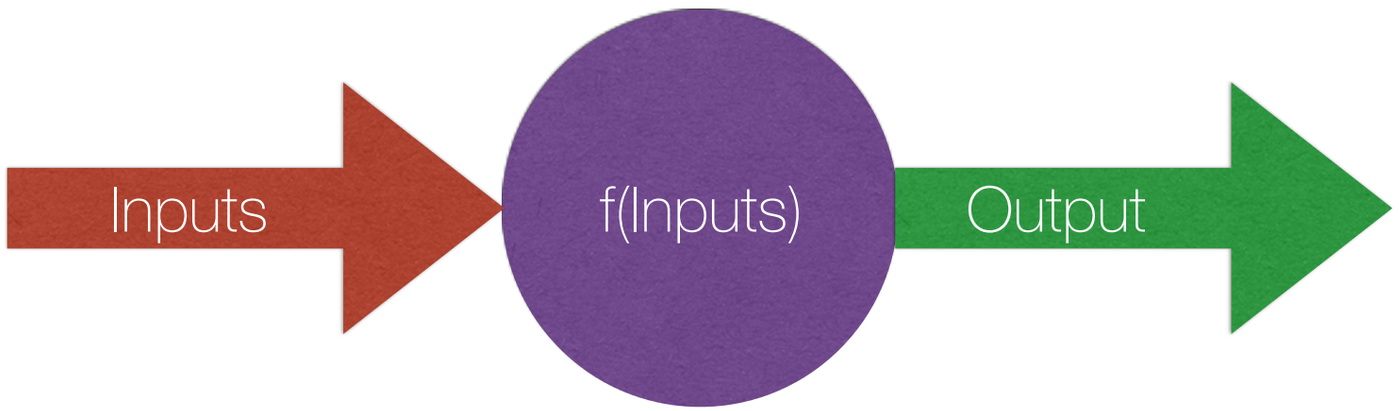
How the Brain "Works" (Cartoon version)

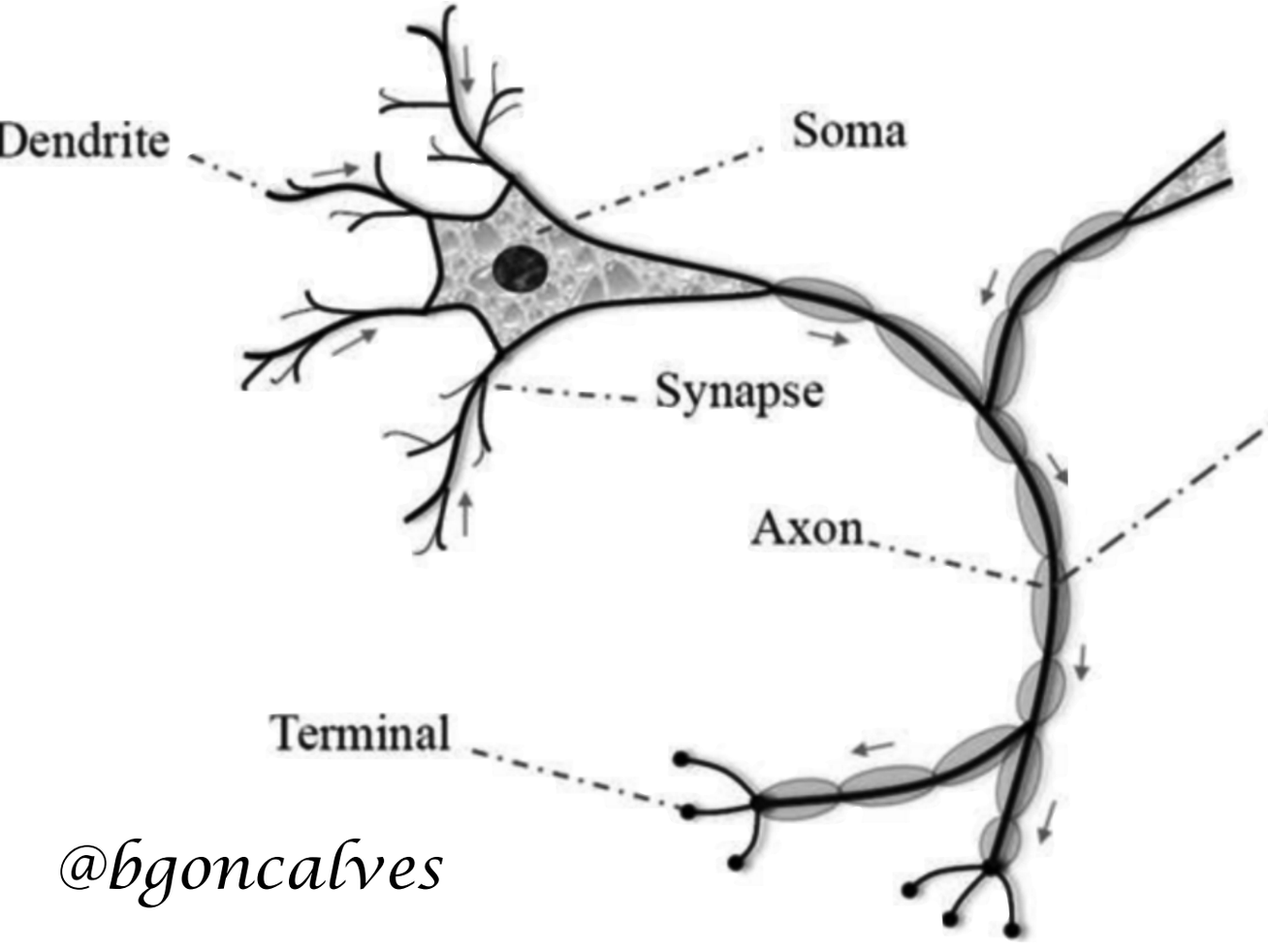
- Each neuron receives input from other neurons
- 10^{11} neurons, each with with 10^{4} weights
- Weights can be positive or negative
- Weights adapt during the learning process
- "neurons that fire together wire together" (Hebb)
- Different areas perform different functions using same structure (Modularity)

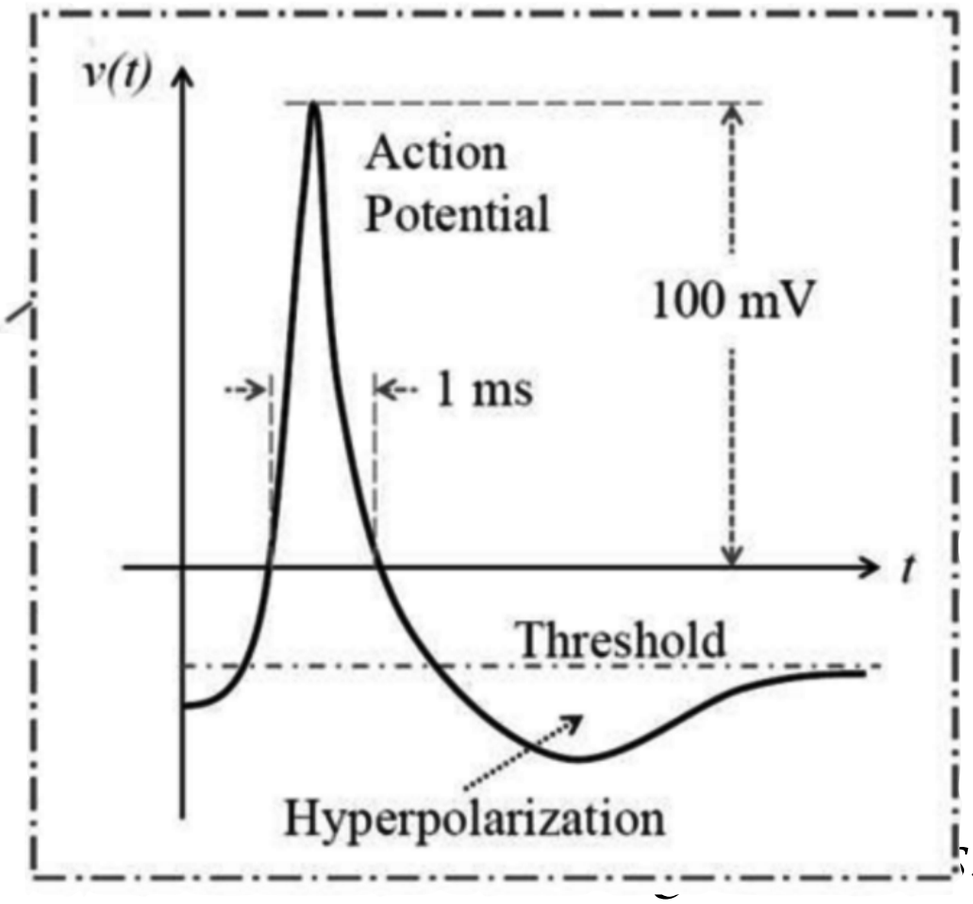




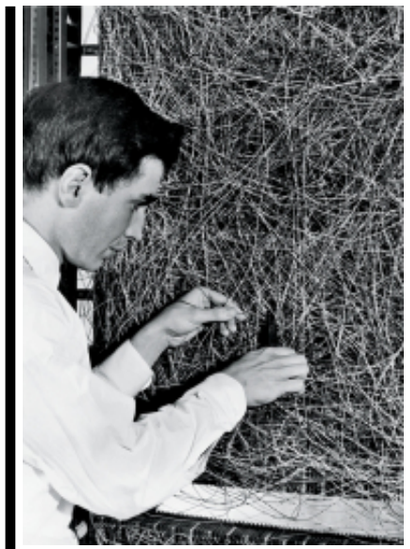
How the Brain "Works" (Cartoon version)







Historical Perspective

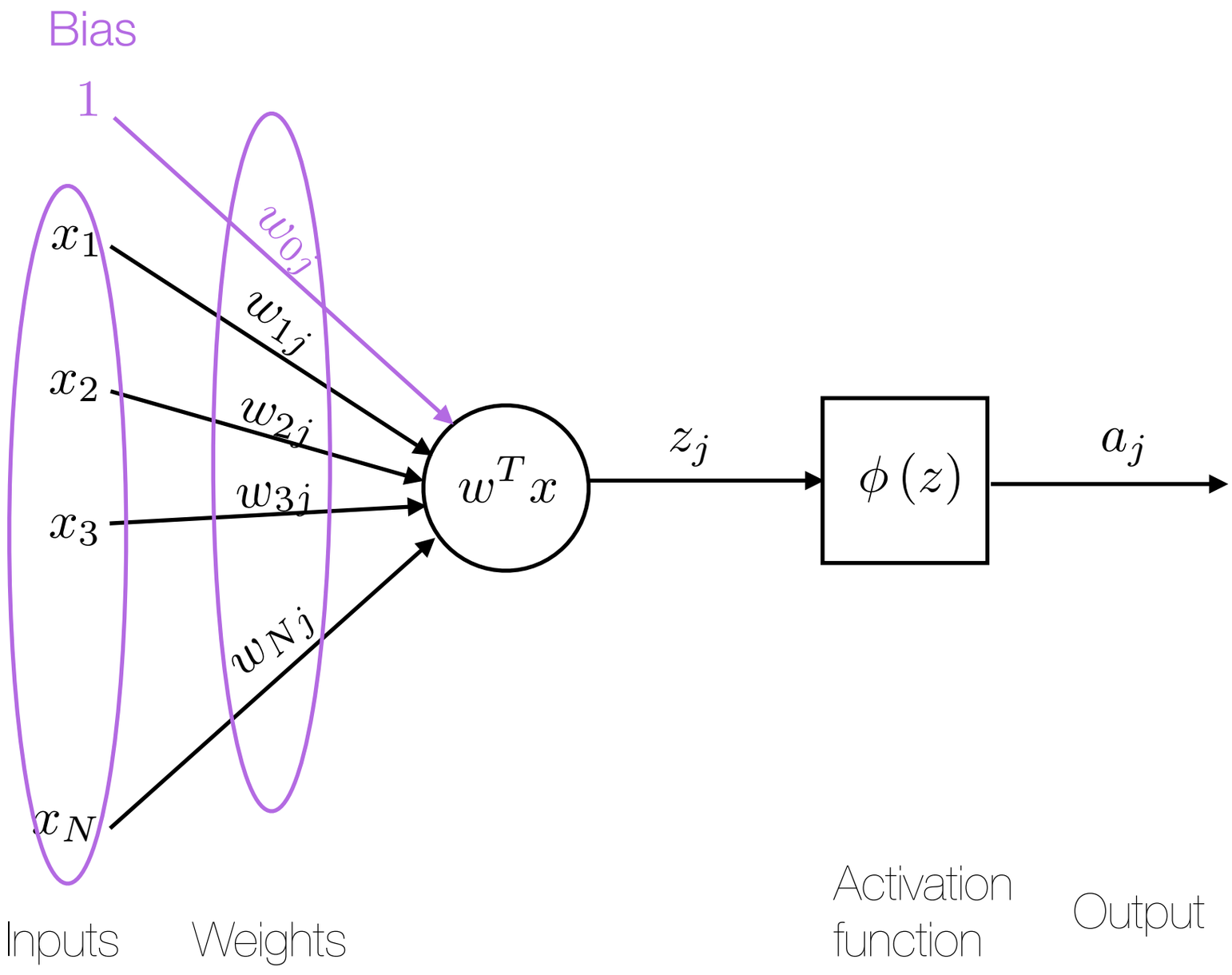


1958

Perceptron

- Popularized by F. Rosenblatt who wrote "Principles of Neurodynamics"
- Still used today
- Simple but limited training procedure
- Single Layer

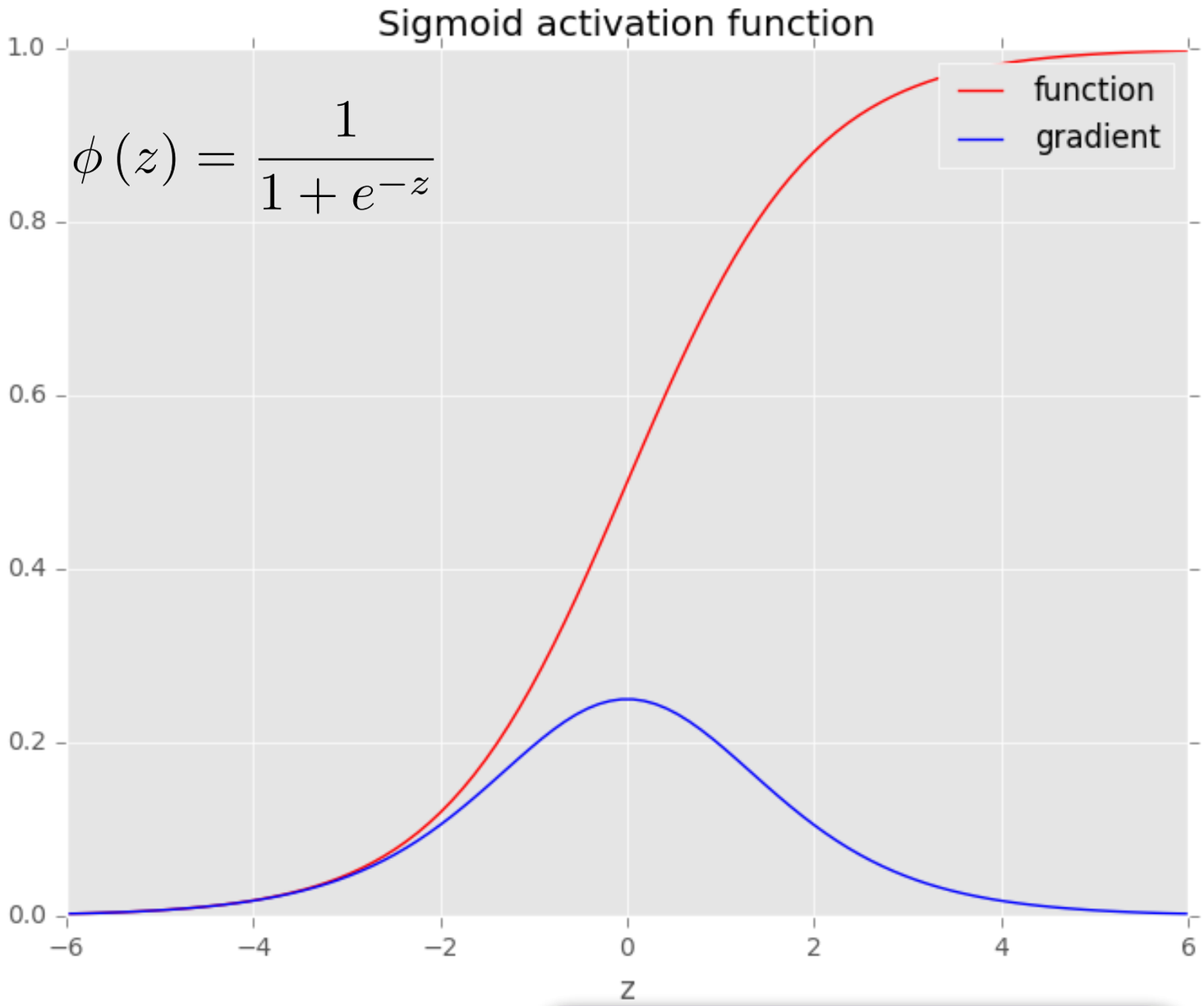
Perceptron


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 - 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

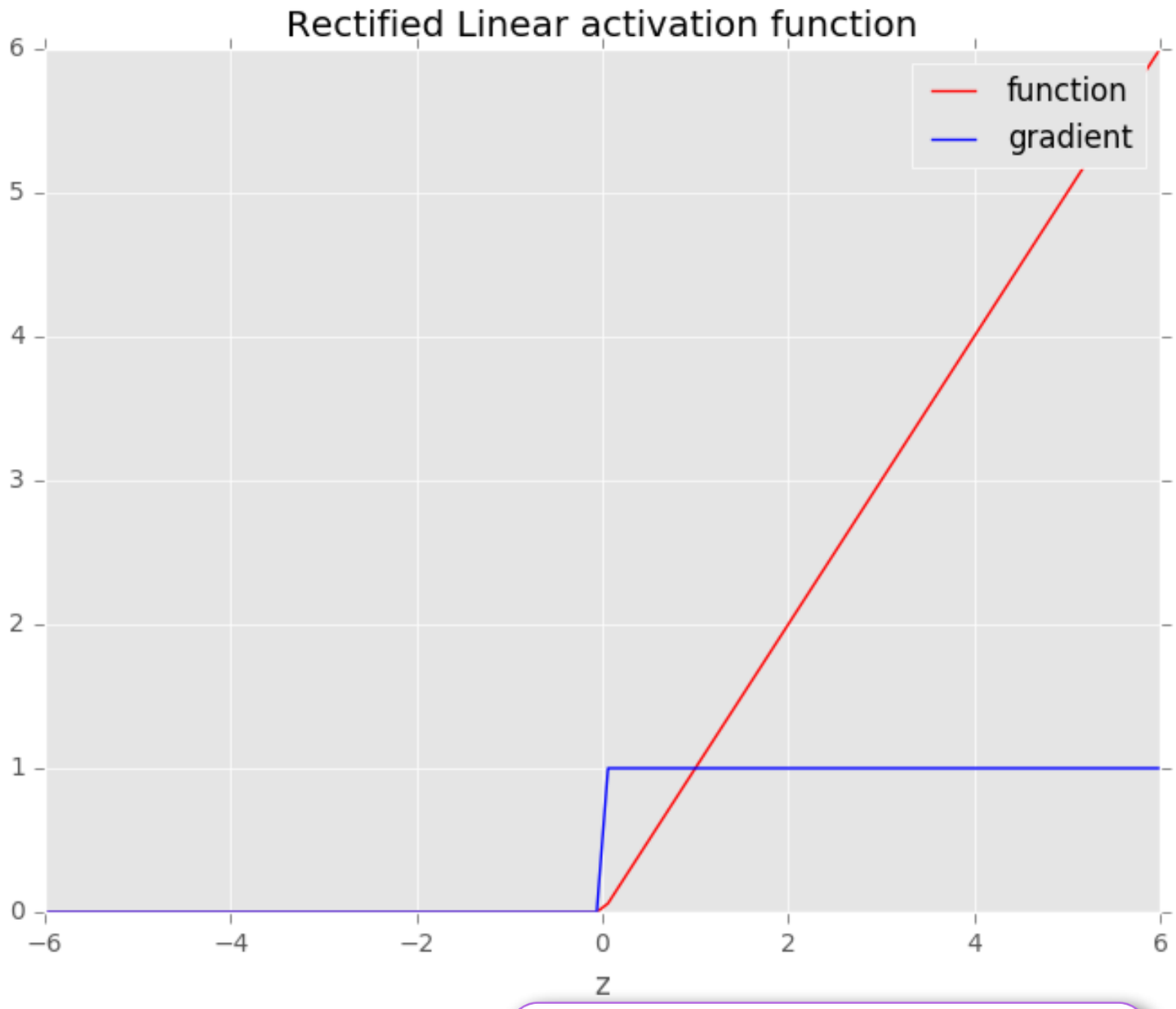


activation.py

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

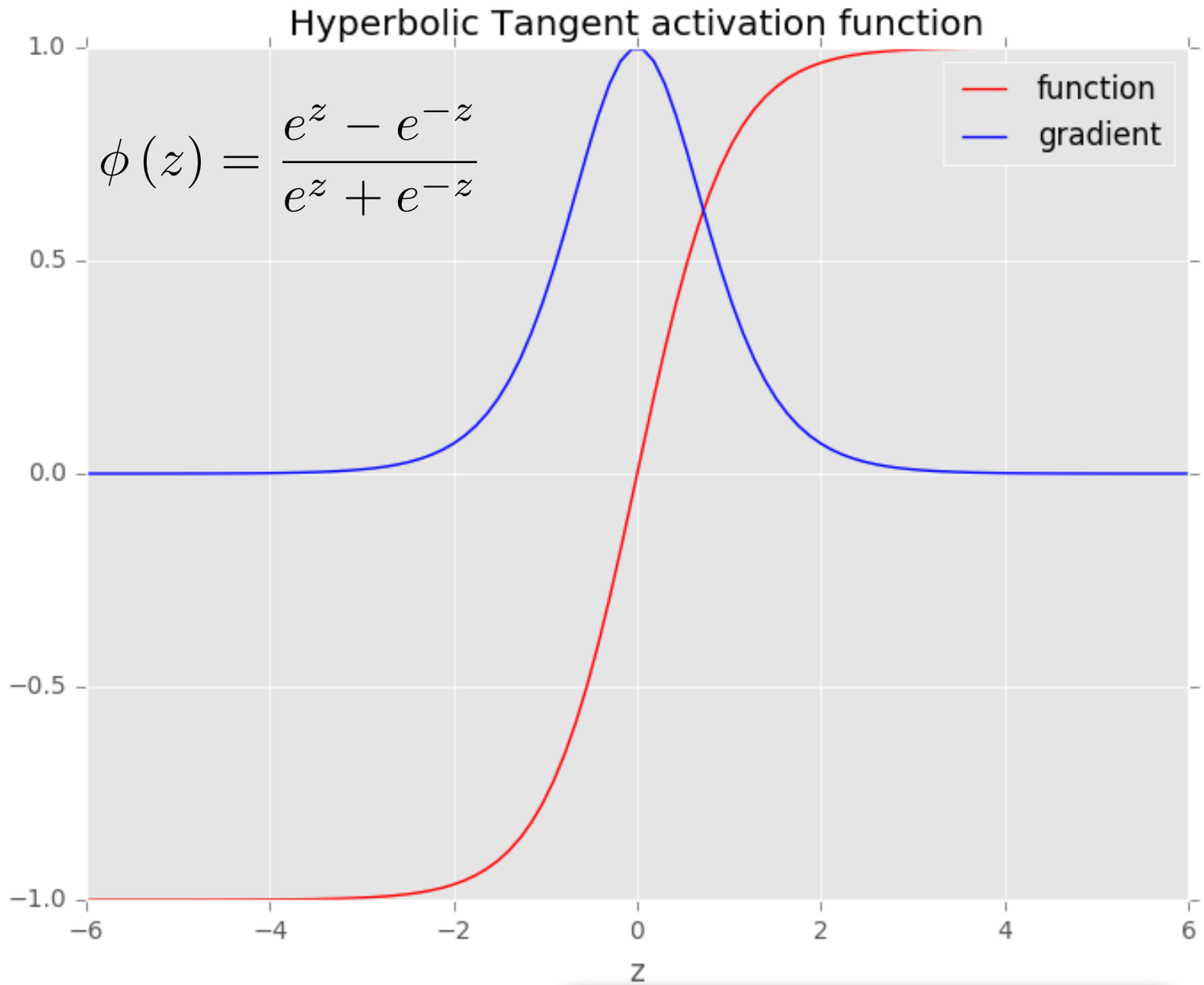


activation.py

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.py

Activation Function

```python
import matplotlib.pyplot as plt
import numpy as np


def linear(z):
    return z


def binary(z):
    return np.where(z > 0, 1, 0)


def relu(z):
    return np.where(z > 0, z, 0)


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


def tanh(z):
    return np.tanh(z)

z = np.linspace(-6, 6, 100)

plt.style.use('ggplot')

plt.plot(z, linear(z), 'r-')
plt.xlabel('z')
plt.title('Linear activation function')
plt.savefig('linear.png')
plt.close()
```

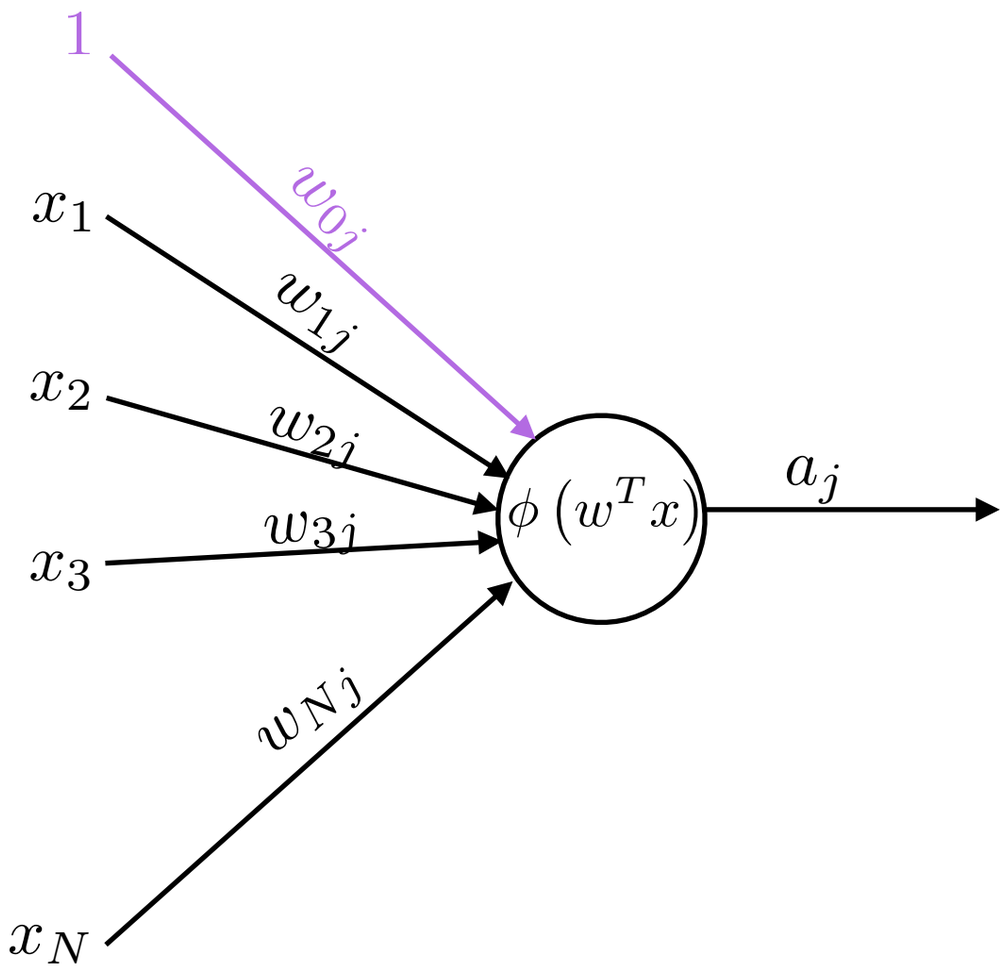
activation.py

Perceptron - Forward Propagation

```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.py

Perceptron

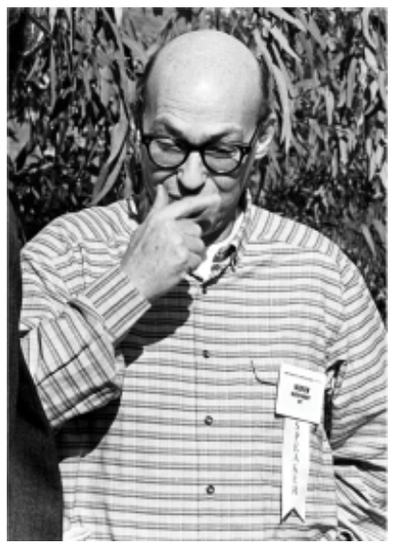


Training Procedure:

- If correct, do nothing
- If output incorrectly outputs 0, add input to weight vector
- if output incorrectly outputs 1, subtract input to weight vector

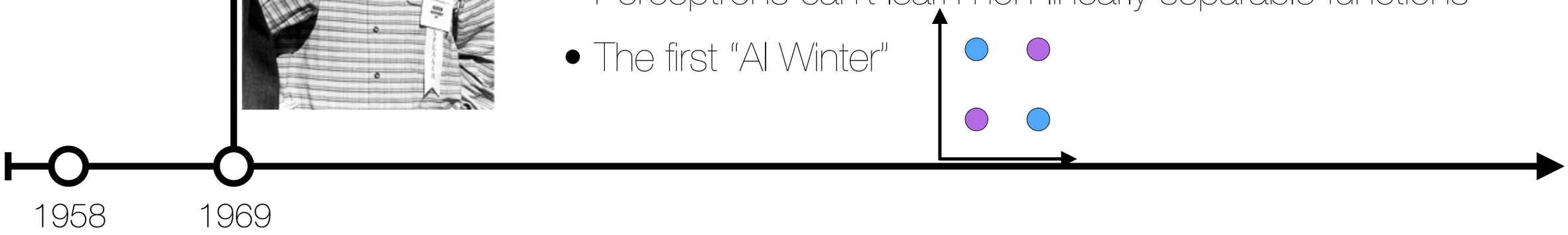
- Guaranteed to converge, if a correct set of weights exists
- Given enough features, perceptrons can learn almost anything
- Specific Features used limit what is possible to learn

Historical Perspective



Marvin Minsky

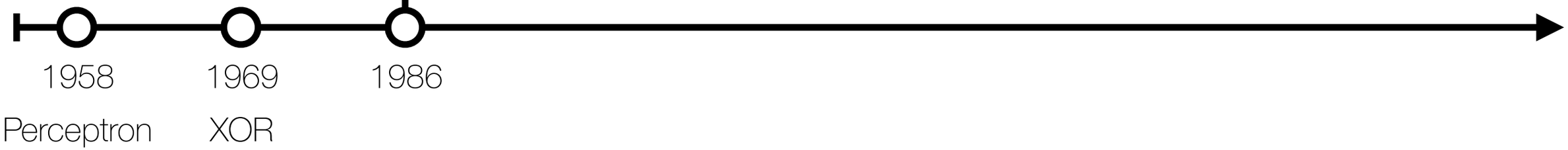
- Co-authors "Perceptrons" with Seymour Papert
- XOR Problem
- Perceptrons can't learn non-linearly separable functions
- The first "AI Winter"



Historical Perspective

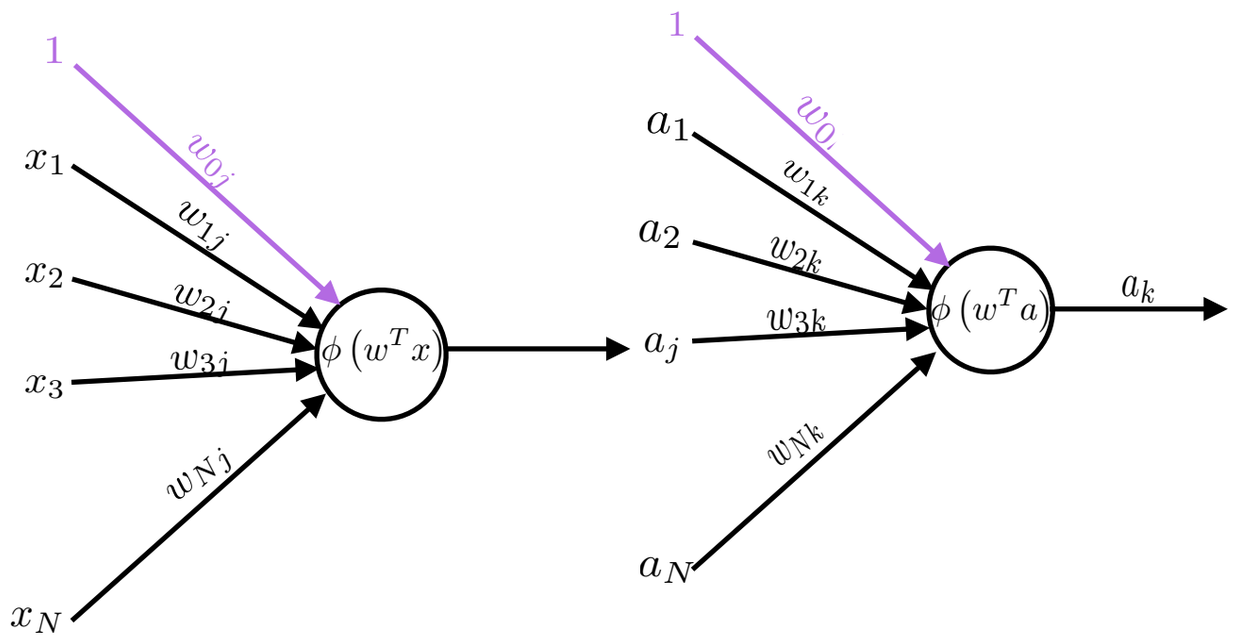
Geoff Hinton

- Discovers "Backpropagation"
- "Multi-layer Perceptron"
- Expensive computation requiring lots of data
- Impractical



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?

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 the squared difference between predicted output and the observed one:

$$E = (t - y)^2$$

- Where t is the real output and y is the predicted one.

- For inner layers there is no "real output"!

Chain-rule

- From the forward propagation described above, we know that the output y_j of a neuron is:

$$y_j = \phi\left(w^T x\right)$$

- But how can we calculate how to modify the weights w_{ij}?

- We take the derivative of the error with respect to the weights!

$$\frac{\partial E}{\partial w_{ij}}$$

- Using the chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}}$$

- And finally we can update each weight in the previous layer:

$$w_{ij} \longleftarrow w_{ij} - \alpha\frac{\partial E}{\partial w_i j}$$

- where α is the learning rate

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:

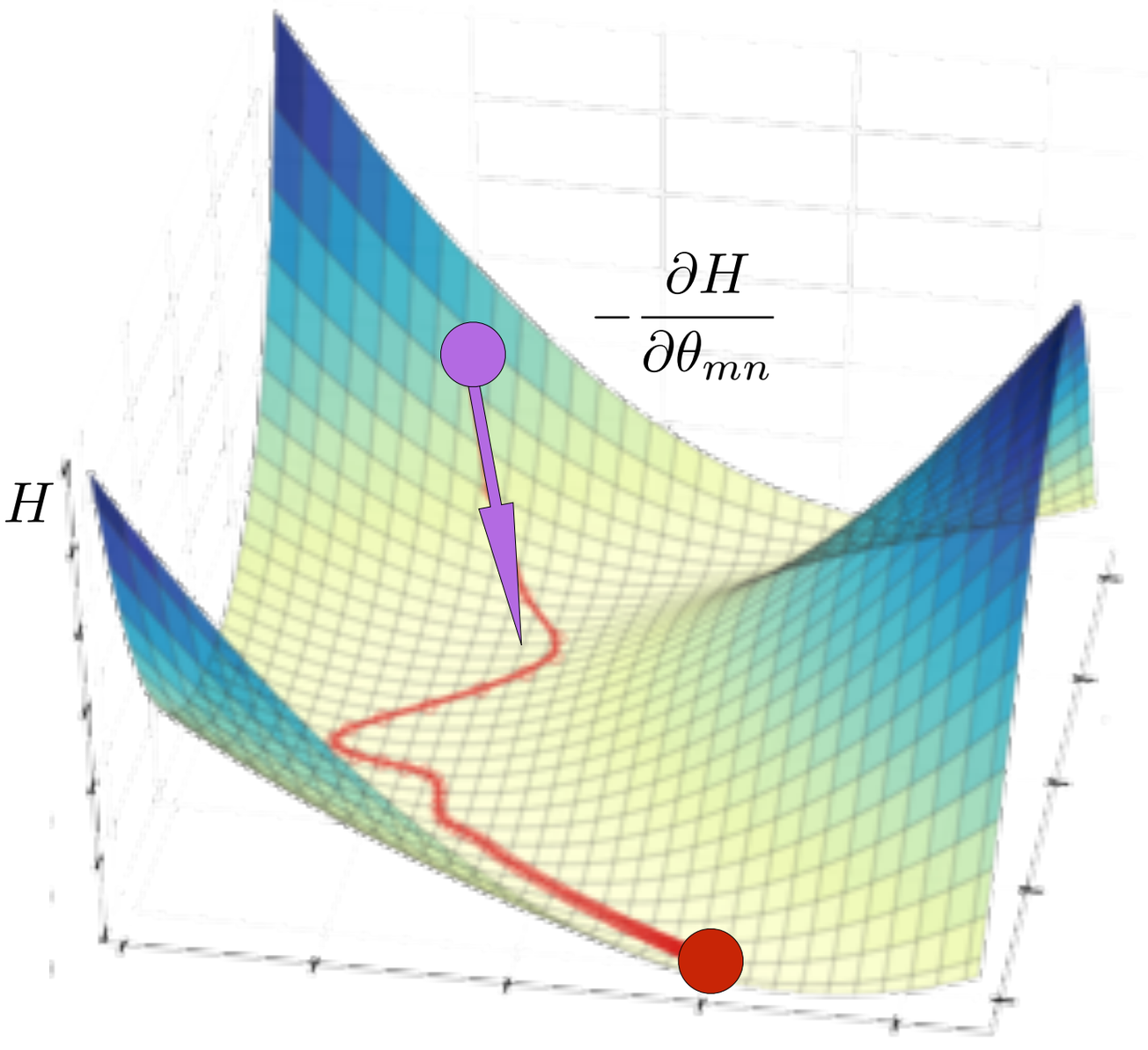
$$E = \frac{1}{N} \sum_n |y_n - a_n|^2$$

  - 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.

Gradient Descent



- Find the gradient for each training batch
- Take a step **downhill** along the direction of the gradient

$$\theta_{mn} \leftarrow \theta_{mn} - \alpha \frac{\partial H}{\partial \theta_{mn}}$$

- where α is the step size.
- Repeat until "convergence".

Historical Perspective

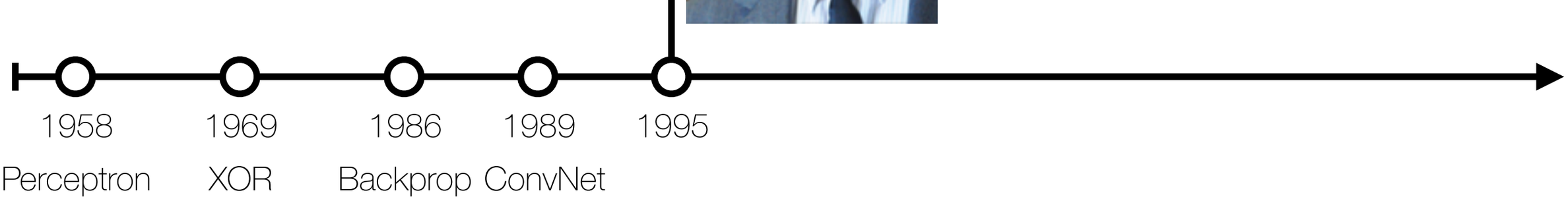
Yann LeCun

- Starts working on Convolutional Neural Networks
- (Eventually) develops the first practical application of image recognition

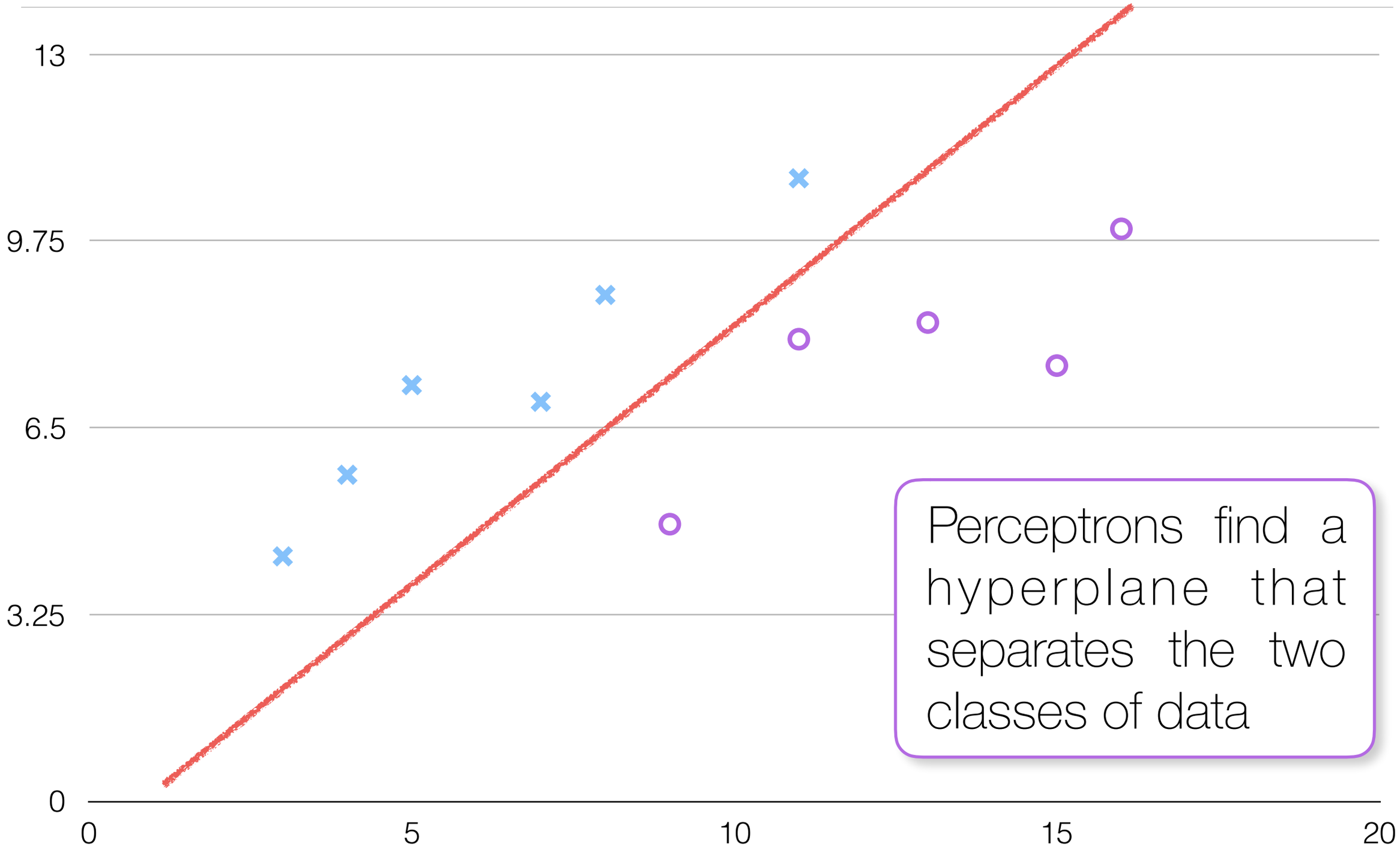
Historical Perspective

Vladimir Vapnik

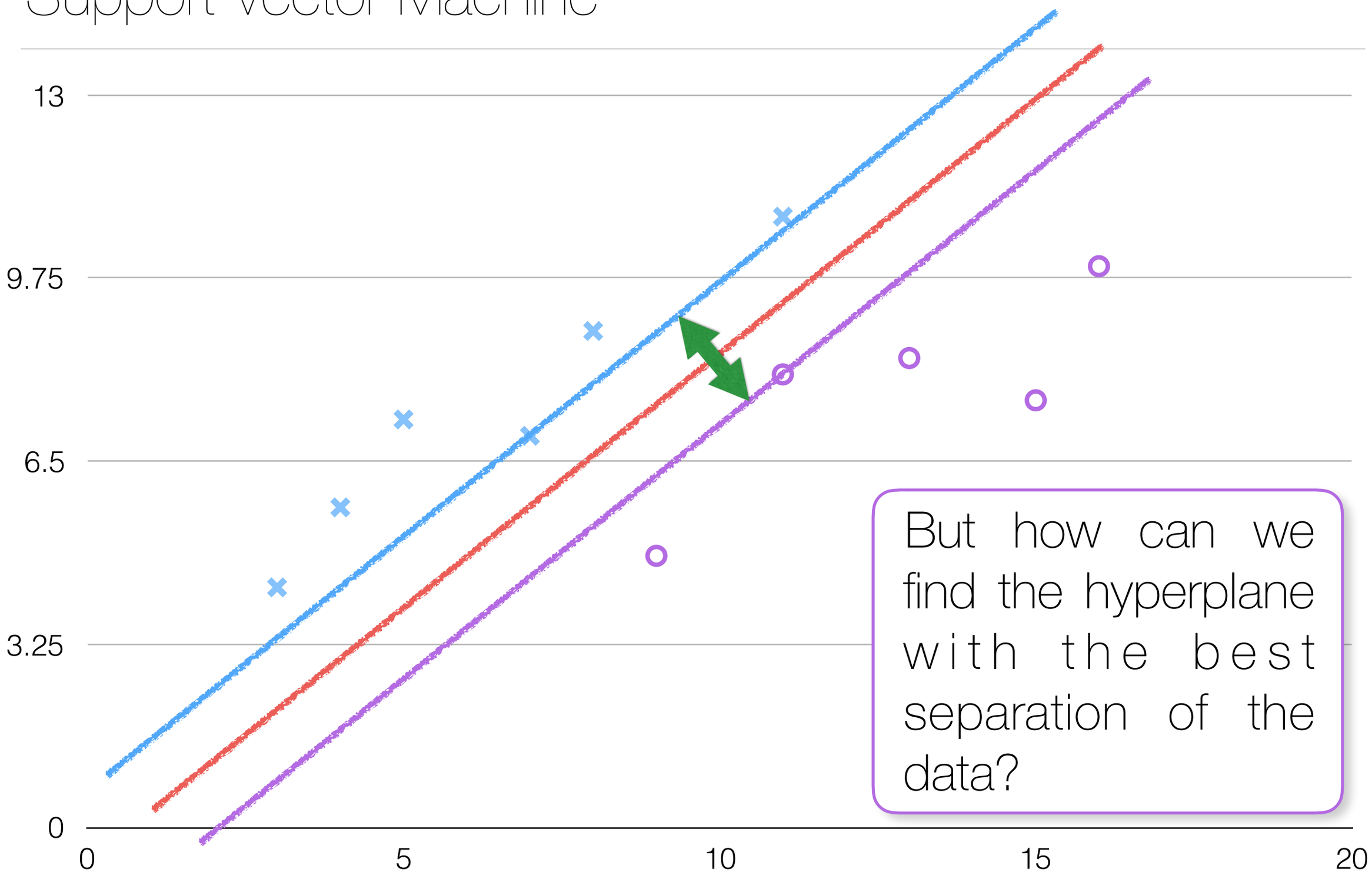
- Support Vector Machines
- A clever variation on Perceptrons
- Kernel Trick



Support Vector Machine



Support Vector Machine



But how can we find the hyperplane with the best separation of the data?

Support Vector Machines

- Decision plane has the form:

$$w^T x = 0$$

- We want $w^T x \geq b$ for points in the "positive" class and $w^T x \leq -b$ for points in the negative class. Where the "margin" $2b$ is as large as possible.

- Normalize such that $b = 1$ and solve the optimization problem:

$$\min_w ||w||^2$$

subject to:

$$y_i \left(w^T x \right) > 1$$

- The margin is:

$$\frac{2}{||w||}$$

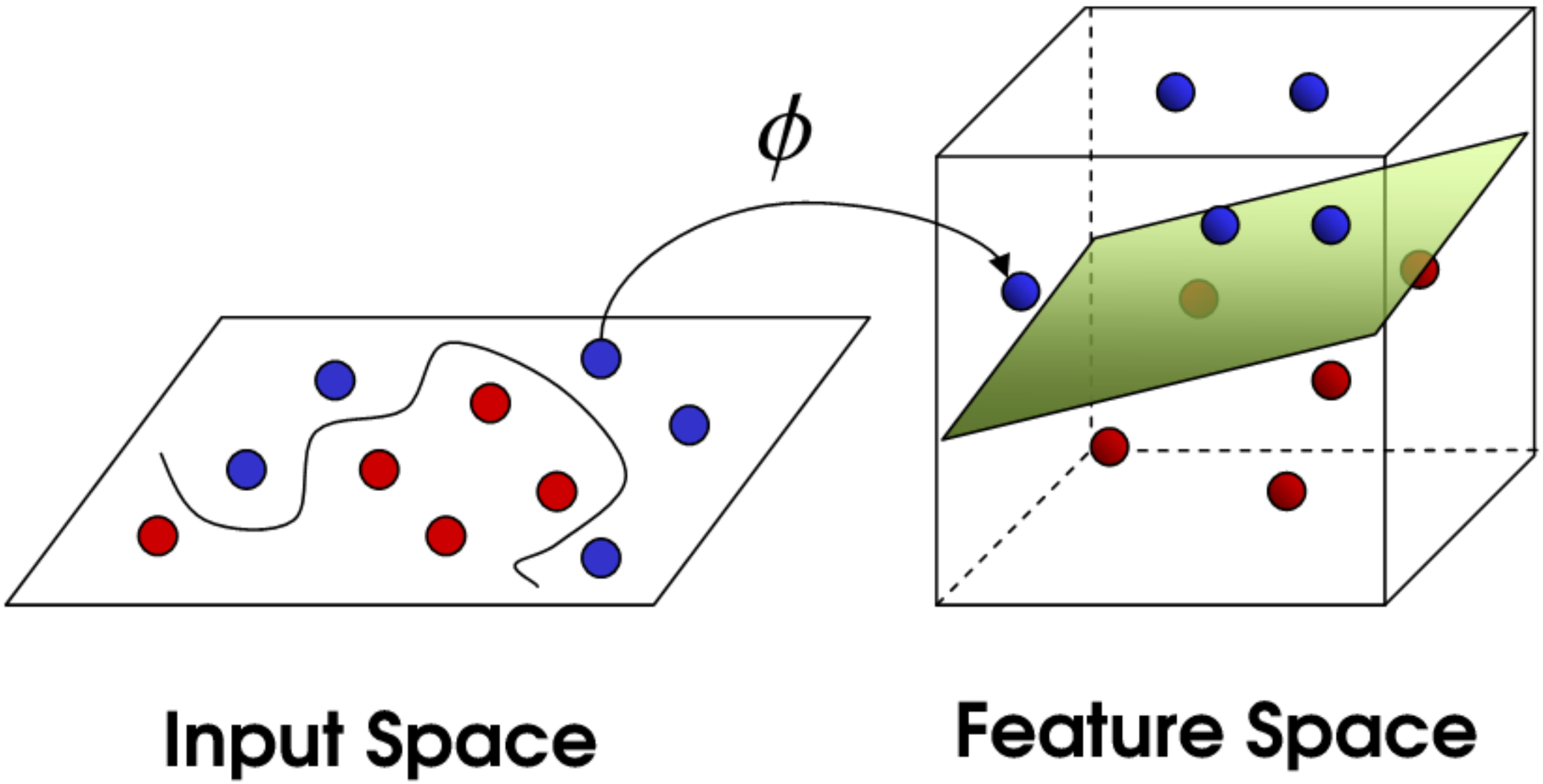
Support Vector Machines

```python
import numpy as np
from sklearn import svm
import matplotlib.pyplot as plt

X_train = np.load('input/X_train.npy')
y_train = np.load('input/y_train.npy')

order = list(range(X_train.shape[0]))
np.random.shuffle(order)
X_train = X_train[order]
y_train = y_train[order]

X_test = np.load('input/X_test.npy')
y_test = np.load('input/y_test.npy')

order = list(range(X_test.shape[0]))
np.random.shuffle(order)
X_test = X_test[order]
y_test = y_test[order]

input_layer_size = X_train.shape[1]
X_train /= 255.
X_test /= 255.

def accuracy(y_test, y_pred):
    return np.sum(y_test==y_pred)/len(y_test)

clf = svm.SVC()
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

print(accuracy(y_test, y_pred))
```

SVM.py

Kernel "trick"

- SVM procedure uses only the dot products of vectors and never the vectors themselves.
- We can redefine the dot product in any way we wish.
- In effect we are mapping from a non-linear input space to a linear feature space



Historical Perspective

Geoff Hinton

- Deep Neural Networks
- Combines backprop with faster machines and larger datasets