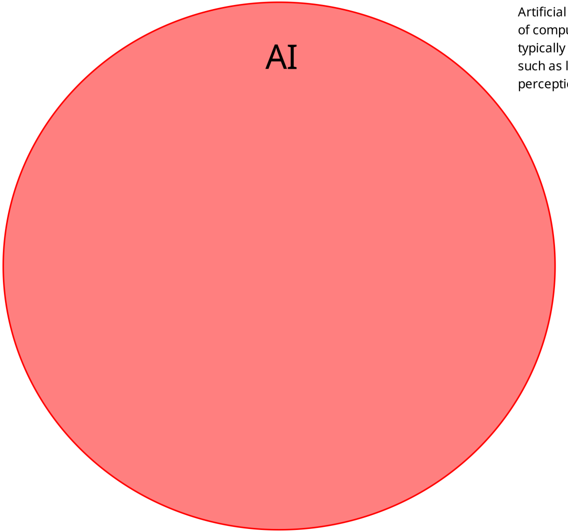
Deep Learning Course

Lesson 1 — Introduction & The Perceptron

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

October 10, 2025

What is Artificial Intelligence?

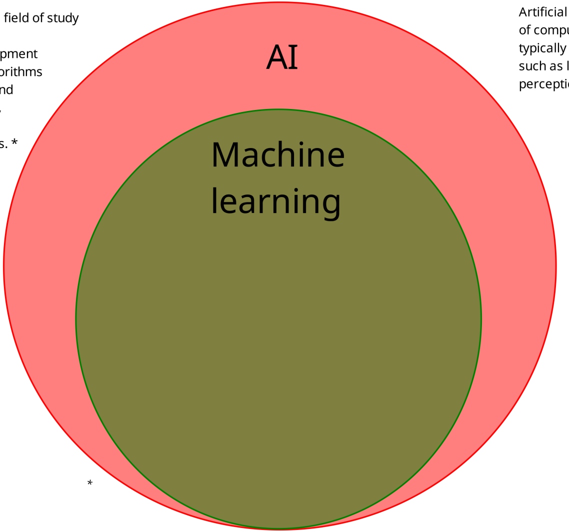


Artificial intelligence (AI) is the capability of computational systems to perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making.*

* *Source: Wikipedia*

What is Machine Learning?

Machine learning (ML) is a field of study
in artificial intelligence
concerned with the development
and study of statistical algorithms
that can learn from data and
generalise to unseen data,
and thus perform tasks
without explicit instructions. *



*

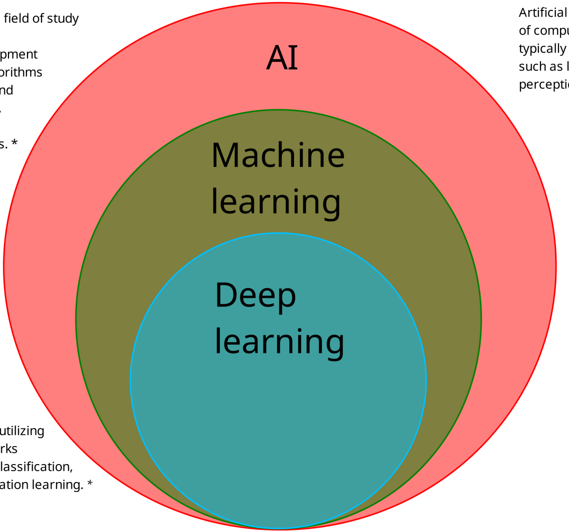
Artificial intelligence (AI) is the capability
of computational systems to perform tasks
typically associated with human intelligence,
such as learning, reasoning, problem-solving,
perception, and decision-making.*

** Source: Wikipedia*

What is Deep Learning?

Machine learning (ML) is a field of study in artificial intelligence concerned with the development and study of statistical algorithms that can learn from data and generalise to unseen data, and thus perform tasks without explicit instructions. *

Artificial intelligence (AI) is the capability of computational systems to perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making.*



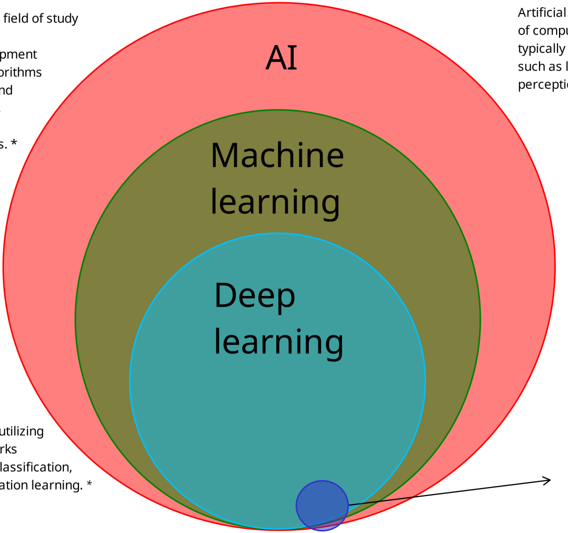
** Source: Wikipedia*

Deep learning focuses on utilizing multilayered neural networks to perform tasks such as classification, regression, and representation learning. *

Our Course

Machine learning (ML) is a field of study in artificial intelligence concerned with the development and study of statistical algorithms that can learn from data and generalise to unseen data, and thus perform tasks without explicit instructions. *

Artificial intelligence (AI) is the capability of computational systems to perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making.*



** Source: Wikipedia*

Deep learning focuses on utilizing multilayered neural networks to perform tasks such as classification, regression, and representation learning. *

A deep-learning from scratch course on MLPs, CNNs and GANs with hands-on projects in computer vision

Artificial Intelligence

- Broad field: perception, reasoning, planning, language.
- Symbolic AI vs. Data-driven AI.
- **Goal:** systems that choose actions to achieve objectives.

- Subfield of AI: algorithms that improve with data.
- Supervised, Unsupervised, Reinforcement Learning.
- **Core idea:** learn a function $f : \mathcal{X} \to \mathcal{Y}$.

Deep Learning

- Subset of ML using deep neural networks.
- Learns hierarchical representations (features).
- Major successes: vision, speech, NLP, generative models.

Mental model for "Is this house worth it?"

$x_1 :=$ does it cost less than 200K?

Mental model for “Is this house worth it?”

x_1 := does it
cost less than 200K?

x_2 := does it
have 3 or more rooms?

Mental model for “Is this house worth it?”

$x_1 :=$ does it cost less than 200K?

$x_2 :=$ does it have 3 or more rooms?

$x_3 :=$ does it have a private garden?

Mental model for “Is this house worth it?”

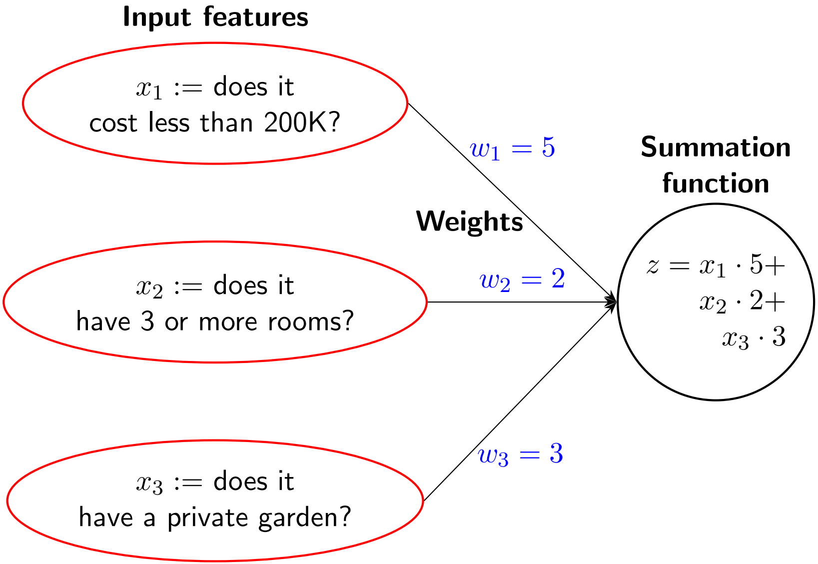
Input features

$x_1 :=$ does it cost less than 200K?

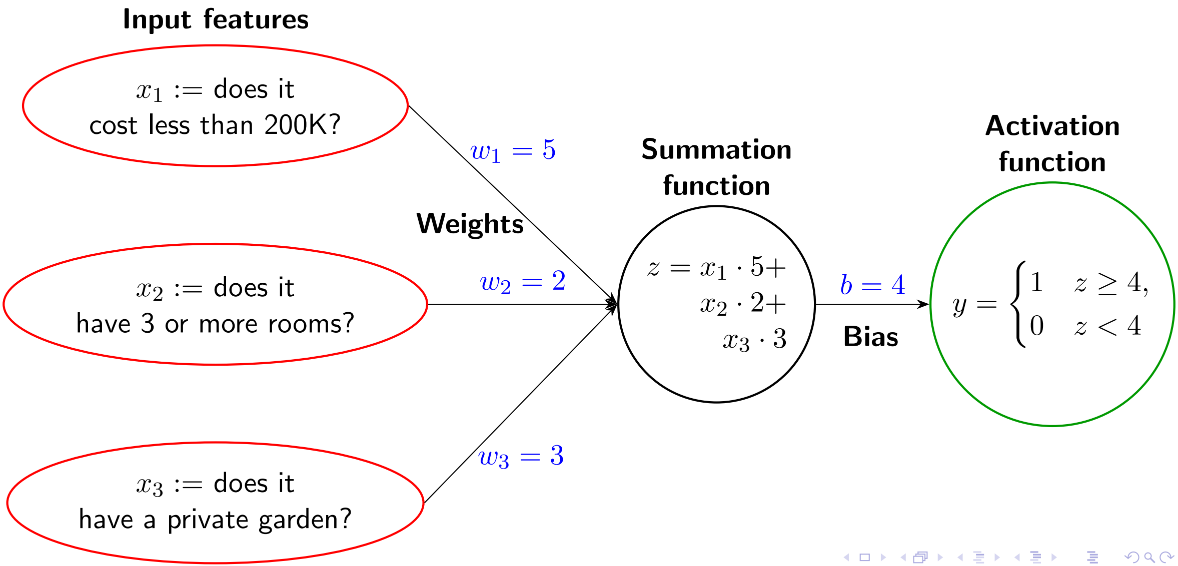
$x_2 :=$ does it have 3 or more rooms?

$x_3 :=$ does it have a private garden?

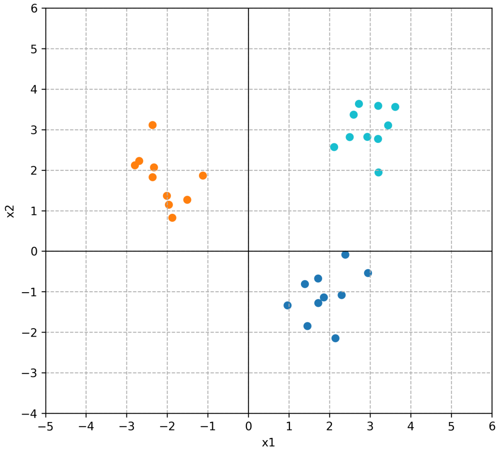
Mental model for "Is this house worth it?"



Mental model for “Is this house worth it?”



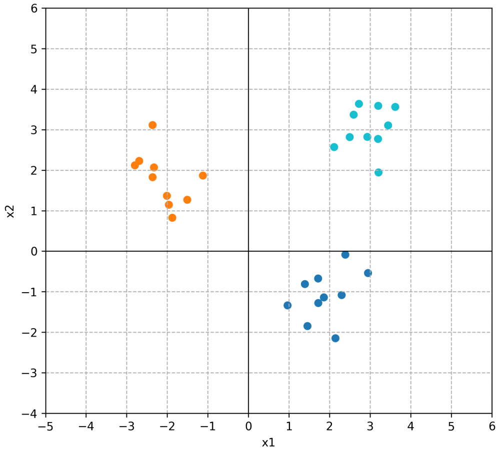
The quiz moment



Question

How many neurons in the input layer are required to separate "good" points (orange) and "bad" points (no-orange)?

The quiz moment



Question

How many neurons in the input layer are required to separate "good" points (orange) and "bad" points (no-orange)?

Answer

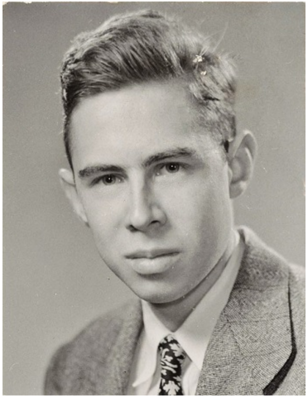
Only 2. x_1 and x_2 are the only input features. The color is not an input feature. It's the target.

The Perceptron

The classical perceptron has been designed by the American psychologist Frank Rosenblatt in 1958

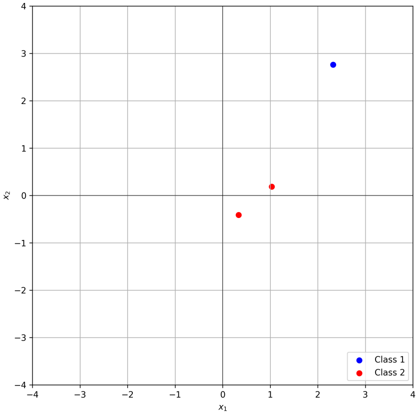
The Perceptron

The classical perceptron has been designed by the American psychologist Frank Rosenblatt in 1958



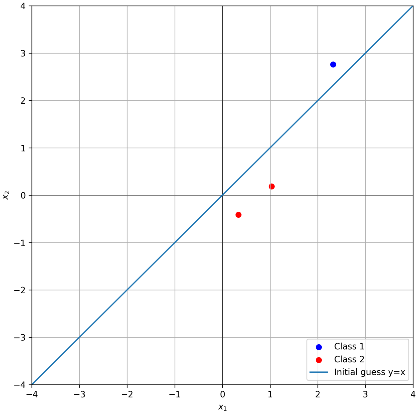
The perceptron

- The perceptron only works if the two classes are linearly separable.
- That means there exists a straight line (in 2D), a plane (in 3D), or, more generally, a hyperplane (in higher dimensions) that perfectly separates the positive and negative examples.



The perceptron

- The perceptron only works if the two classes are linearly separable.
- That means there exists a straight line (in 2D), a plane (in 3D), or, more generally, a hyperplane (in higher dimensions) that perfectly separates the positive and negative examples.



Back to school

What's the equation of a straight line?

Back to school

What's the equation of a straight line?
The equation of a straight line is usually written this way:

$$y = mx + C$$

Or using our "updated" coordinate system:

$$x_2 = mx_1 + C$$

Back to school

However, with the formulation $y = mx + C$ we are not able to represent vertical lines.

Back to school

However, with the formulation $y = mx + C$ we are not able to represent vertical lines.
To express any straight line on a Cartesian plane, we can use the general formula:

$$Ax + By + C = 0$$

Back to school

However, with the formulation $y = mx + C$ we are not able to represent vertical lines.
To express any straight line on a Cartesian plane, we can use the general formula:

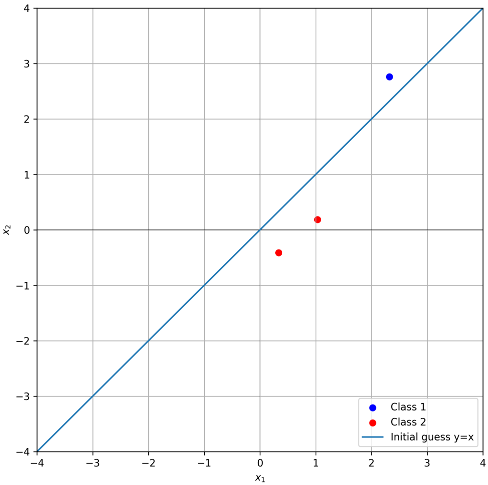
$$Ax + By + C = 0$$

Or using our "updated" coordinate system:

$$w_1 x_1 + w_2 x_2 + b = 0$$

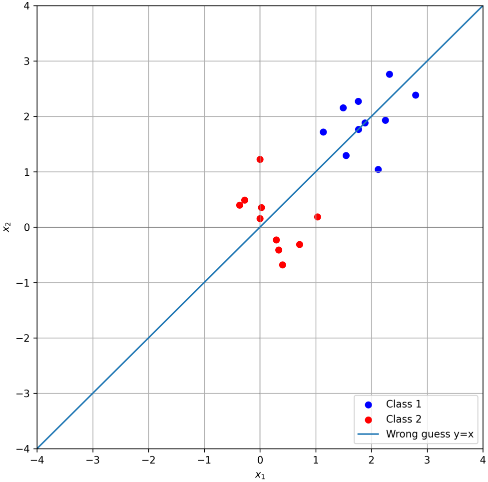
The perceptron

- We can try to guess a linear separator



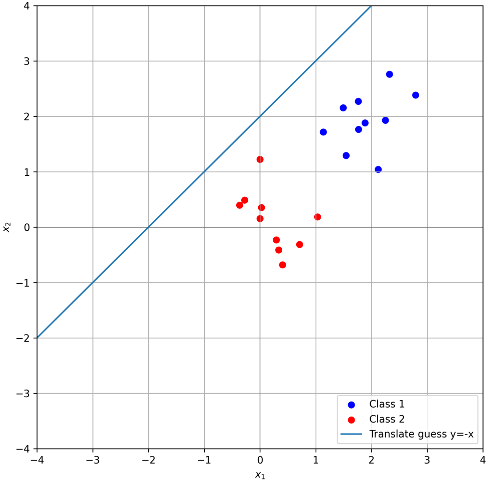
The perceptron

- We can try to guess a linear separator
- Probably a wrong one.



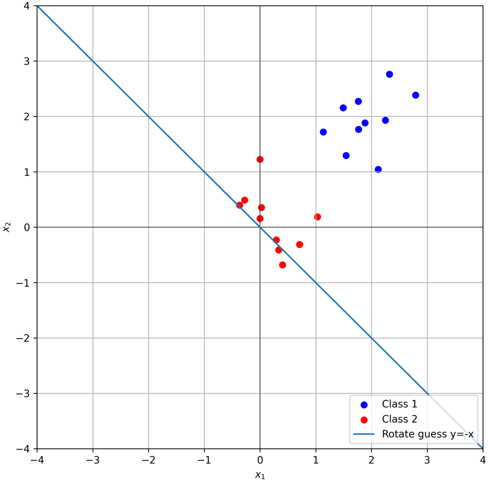
The perceptron

- We can try to guess a linear separator
- Probably a wrong one.
- Translate the separator does not work.



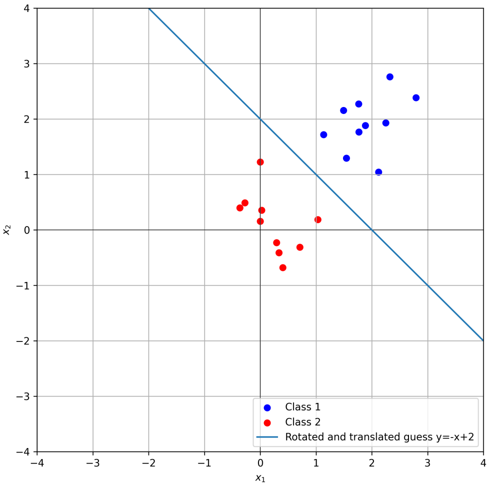
The perceptron

- We can try to guess a linear separator
- Probably a wrong one.
- Translate the separator does not work.
- Neither rotating it does.



The perceptron

- We can try to guess a linear separator
- Probably a wrong one.
- Translate the separator does not work.
- Neither rotating it does.
- We have to rotate and translate the separator, but how?



Perceptron Learning Algorithm: update on a single misclassified point

Input: weights w_1, w_2, bias b, learning rate $\eta > 0$
Data: point (x_1, x_2) with label $y \in \{0, 1\}$
Compute score: $s \leftarrow w_1 \cdot x_1 + w_2 \cdot x_2 + b$
Prediction (STEP):

$$\hat{y} \leftarrow \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

if $\hat{y} \neq y$ **then**
  Error: $e \leftarrow y - \hat{y}$ // $e \in \{-1, +1\}$
  $w_1 \leftarrow w_1 + \eta \cdot e \cdot x_1$
  $w_2 \leftarrow w_2 + \eta \cdot e \cdot x_2$
  $b \leftarrow b + \eta \cdot e$
else
  No update

Output: updated weights (w_1, w_2) and bias b

Why does it work?

When we update

$$b \leftarrow b + \eta \cdot e$$

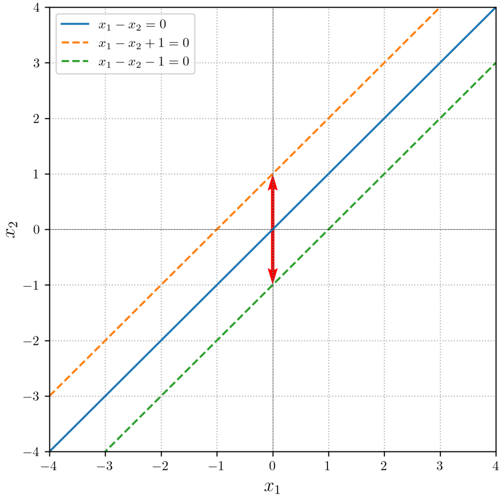
we are simply shifting the line proportionally to η, but what happens when we update the weights with

$$w_1 \leftarrow w_1 + \eta \cdot e \cdot x_1$$

and

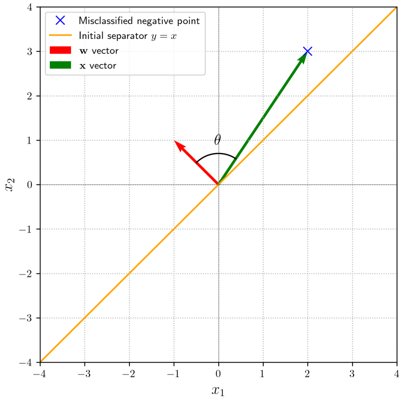
$$w_2 \leftarrow w_2 + \eta \cdot e \cdot x_2$$

?



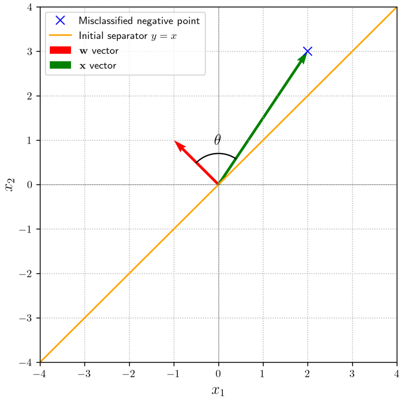
A geometric intuition, with a misclassified negative point

- w is the normal vector to the straight line separator



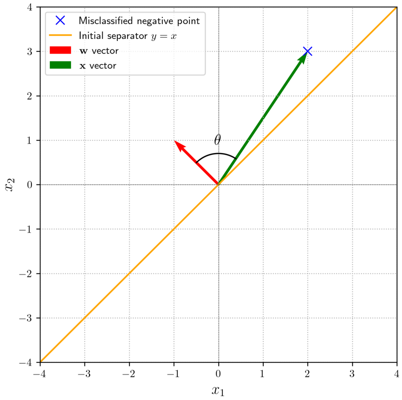
A geometric intuition, with a misclassified negative point

- w is the normal vector to the straight line separator
- $w_1 x_1 + w_2 x_2 = \langle x, w \rangle = \|x\| \|w\| \cos\theta$



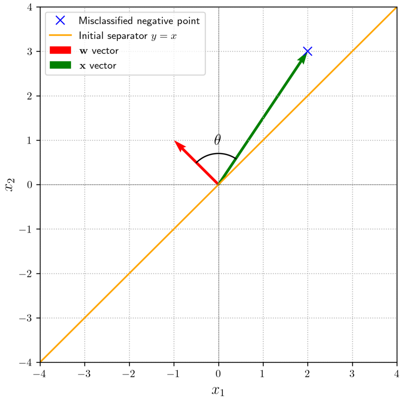
A geometric intuition, with a misclassified negative point

- w is the normal vector to the straight line separator
- $w_1 x_1 + w_2 x_2 = \langle x, w \rangle = \|x\| \|w\| \cos\theta$
- $\cos(x) > 0$ when θ is between 0° and 90°



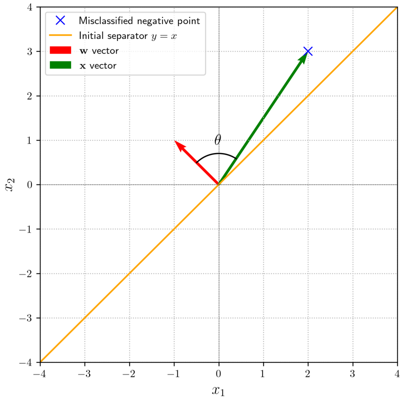
A geometric intuition, with a misclassified negative point

- w is the normal vector to the straight line separator
- $w_1 x_1 + w_2 x_2 = \langle x, w \rangle = \|x\| \|w\| \cos\theta$
- $\cos(x) > 0$ when θ is between $0°$ and $90°$
- $\cos(x) < 0$ when θ is between $90°$ and $180°$



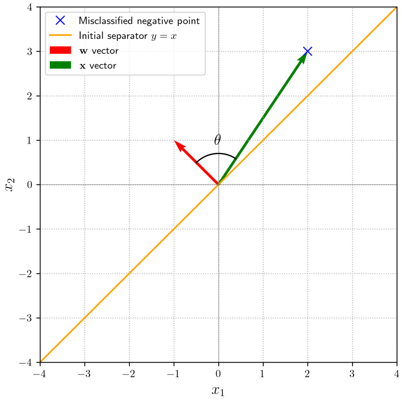
A geometric intuition, with a misclassified negative point

- w is the normal vector to the straight line separator
- $w_1x_1 + w_2x_2 = \langle x, w\rangle = \|x\|\|w\| \cos\theta$
- $\cos(x) > 0$ when θ is between 0° and 90°
- $\cos(x) < 0$ when θ is between 90° and 180°
- We have to make θ narrower if y is positive, i.e. vectors pointing roughly in in the same direction



A geometric intuition, with a misclassified negative point

- w is the normal vector to the straight line separator
- $w_1x_1 + w_2x_2 = \langle x, w \rangle = \|x\|\|w\| \cos\theta$
- $\cos(x) > 0$ when θ is between $0°$ and $90°$
- $\cos(x) < 0$ when θ is between $90°$ and $180°$
- We have to make θ narrower if y is positive, i.e. vectors pointing roughly in in the same direction
- We have to make θ wider if y is negative, i.e. vectors pointing roughly in in opposite directions



Intuition of PLA Convergence

- The Perceptron Learning Algorithm corrects mistakes one by one.

Intuition of PLA Convergence

- The Perceptron Learning Algorithm corrects mistakes one by one.
- Each time we find a misclassified point, we adjust the separator to move it closer to the correct side.

Intuition of PLA Convergence

- The Perceptron Learning Algorithm corrects mistakes one by one.
- Each time we find a misclassified point, we adjust the separator to move it closer to the correct side.
- Even if a correction may misclassify some earlier points, repeating the process over all data gradually improves the alignment.

Intuition of PLA Convergence

- The Perceptron Learning Algorithm corrects mistakes one by one.
- Each time we find a misclassified point, we adjust the separator to move it closer to the correct side.
- Even if a correction may misclassify some earlier points, repeating the process over all data gradually improves the alignment.
- By cycling through the data and applying updates, the separator cannot keep making mistakes forever if the data are linearly separable.

Intuition of PLA Convergence

- The Perceptron Learning Algorithm corrects mistakes one by one.
- Each time we find a misclassified point, we adjust the separator to move it closer to the correct side.
- Even if a correction may misclassify some earlier points, repeating the process over all data gradually improves the alignment.
- By cycling through the data and applying updates, the separator cannot keep making mistakes forever if the data are linearly separable.
- Therefore, after a finite number of corrections, the algorithm stops making errors: it **converges**.

Lab Time

We have two positives points (class 1): (2,2) and (4,0) and two negative points (class 0): (2,0) and (0,2). We want to find a linear separator.
Start from the initial linear separator $y = 1$, with a learning rate $\eta = 1$, and find a proper linear separator. Loop over the points in the given order.

Lab Time

We have two positives points (class 1): (2,2) and (4,0) and two negative points (class 0): (2,0) and (0,2). We want to find a linear separator.
Start from the initial linear separator $y = 1$, with a learning rate $\eta = 1$, and find a proper linear separator. Loop over the points in the given order.
You have 30 minutes.

Lab Time

We have two positives points (class 1): (2,2) and (4,0) and two negative points (class 0): (2,0) and (0,2). We want to find a linear separator.
Start from the initial linear separator $y = 1$, with a learning rate $\eta = 1$, and find a proper linear separator. Loop over the points in the given order.
You have 30 minutes.
Good luck!

My trivial implementation

```
x =  [[2, 2],
      [4, 0],
      [2, 0],
      [0, 2]]

y = [1,1,0,0]


#Parameters Initialization
w0 = 0
w1 = 1
b = -1

learning_rate = 1

while True:
  errors = 0
  for i in range(4):
    x_i = x[i]
    y_i = y[i]
```

My trivial implementation

```
    if w0 *x_i[0] + w1*x_i[1] + b > 0:
      y_hat = 1
    else:
      y_hat = 0

    if y_i != y_hat:
      w0 = w0 + (y_i - y_hat)*x_i[0]
      w1 = w1 + (y_i - y_hat)*x_i[1]
      b = b + (y_i - y_hat)

      print(w0,w1,b)
      errors += 1

  if errors == 0:
    break


print(w0,w1,b)
```

There is room from improvements: NumPy

- **NumPy** is the fundamental Python library for numerical computing.

There is room from improvements: NumPy

- **NumPy** is the fundamental Python library for numerical computing.
- It provides the **ndarray** object: a fast, memory-efficient multidimensional array.

There is room from improvements: NumPy

- **NumPy** is the fundamental Python library for numerical computing.
- It provides the **ndarray** object: a fast, memory-efficient multidimensional array.
- Includes a wide range of functions for **linear algebra, statistics, and mathematical operations**.

There is room from improvements: NumPy

- **NumPy** is the fundamental Python library for numerical computing.
- It provides the **ndarray** object: a fast, memory-efficient multidimensional array.
- Includes a wide range of functions for **linear algebra, statistics, and mathematical operations**.
- Designed to work efficiently with large datasets and to integrate with other scientific libraries.

Numpy example

Creating and using an array

```
import numpy as np
a = np.array([1, 2, 3])
print(a * 2)   # Output: [2 4 6]
```

My trivial implementation, v2

```
import numpy as np

x =  np.array([[2, 2],
               [4, 0],
               [2, 0],
               [0, 2]])

y = np.array([1,1,0,0])


#Initialization
w = np.array([0, 1])
b = -1
learning_rate = 1

#Activation function
def step_function(z):
        if z > 0:
                return 1
        else:
```

My trivial implementation, v2

```
            return 0

while True:
        errors = 0
        for i in range(4):
                x_i = x[i]
                y_i = y[i]

                z = w[0]*x_i[0] + w[1]*x_i[1] + b
                y_hat = step_function(z)

                if y_i != y_hat:
                        w = w + (y_i - y_hat)*x_i
                        b = b + (y_i - y_hat)
                        errors += 1

        if errors == 0:
                break


print(w,b)
```

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:
  - The prediction is directly linked to the sign of $\mathbf{w} \cdot \mathbf{x}$.

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:
  - The prediction is directly linked to the sign of $\mathbf{w} \cdot \mathbf{x}$.
  - The update rule becomes very simple: $\mathbf{w} \leftarrow \mathbf{w} + y\,\mathbf{x}$ when there is a mistake.

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:
  - The prediction is directly linked to the sign of $\mathbf{w} \cdot \mathbf{x}$.
  - The update rule becomes very simple: $\mathbf{w} \leftarrow \mathbf{w} + y\,\mathbf{x}$ when there is a mistake.
  - Algebraic manipulations in the proof of convergence are much easier (inner product inequalities).

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:
  - The prediction is directly linked to the sign of $\mathbf{w} \cdot \mathbf{x}$.
  - The update rule becomes very simple: $\mathbf{w} \leftarrow \mathbf{w} + y\,\mathbf{x}$ when there is a mistake.
  - Algebraic manipulations in the proof of convergence are much easier (inner product inequalities).
- If we use outputs $\{0, 1\}$ instead:

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:
  - The prediction is directly linked to the sign of $\mathbf{w} \cdot \mathbf{x}$.
  - The update rule becomes very simple: $\mathbf{w} \leftarrow \mathbf{w} + y\,\mathbf{x}$ when there is a mistake.
  - Algebraic manipulations in the proof of convergence are much easier (inner product inequalities).
- If we use outputs $\{0, 1\}$ instead:
  - We must translate labels back and forth to apply the update rule.

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:
  - The prediction is directly linked to the sign of $\mathbf{w} \cdot \mathbf{x}$.
  - The update rule becomes very simple: $\mathbf{w} \leftarrow \mathbf{w} + y\,\mathbf{x}$ when there is a mistake.
  - Algebraic manipulations in the proof of convergence are much easier (inner product inequalities).
- If we use outputs $\{0, 1\}$ instead:
  - We must translate labels back and forth to apply the update rule.
  - Expressions like $y(\mathbf{w} \cdot \mathbf{x})$ would not work directly.

Why Use the Sign Function in the Perceptron

- The perceptron must decide whether a point is on one side of the separator or the other.
- If we use outputs $\{+1, -1\}$ via the **sign function**, then:
  - The prediction is directly linked to the sign of $\mathbf{w} \cdot \mathbf{x}$.
  - The update rule becomes very simple: $\mathbf{w} \leftarrow \mathbf{w} + y\,\mathbf{x}$ when there is a mistake.
  - Algebraic manipulations in the proof of convergence are much easier (inner product inequalities).
- If we use outputs $\{0, 1\}$ instead:
  - We must translate labels back and forth to apply the update rule.
  - Expressions like $y(\mathbf{w} \cdot \mathbf{x})$ would not work directly.
- **Conclusion:** using the sign function and $\{\pm 1\}$ labels simplifies both the algorithm and its theoretical analysis.

From Sums to Vector Notation in NumPy

- When we write the perceptron in code, we often expand the sum by hand. Example with two features:

```
z = w[0]*x_i[0] + w[1]*x_i[1] + b
```

From Sums to Vector Notation in NumPy

- When we write the perceptron in code, we often expand the sum by hand. Example with two features:

```
z = w[0]*x_i[0] + w[1]*x_i[1] + b
```

- In vector notation, this is simply:

$$z = \mathbf{w} \cdot \mathbf{x}_i + b$$

where $\mathbf{w}$ and $\mathbf{x}_i$ are vectors.

From Sums to Vector Notation in NumPy

- When we write the perceptron in code, we often expand the sum by hand. Example with two features:

```
z = w[0]*x_i[0] + w[1]*x_i[1] + b
```

- In vector notation, this is simply:

$$z = \mathbf{w} \cdot \mathbf{x}_i + b$$

  where $\mathbf{w}$ and $\mathbf{x}_i$ are vectors.

- NumPy makes this compact with the `@` operator for dot products:

```
z = x_i @ w + b
```

From Sums to Vector Notation in NumPy

- When we write the perceptron in code, we often expand the sum by hand. Example with two features:

```
z = w[0]*x_i[0] + w[1]*x_i[1] + b
```

- In vector notation, this is simply:

$$z = \mathbf{w} \cdot \mathbf{x}_i + b$$

  where $\mathbf{w}$ and $\mathbf{x}_i$ are vectors.
- NumPy makes this compact with the `@` operator for dot products:

```
z = x_i @ w + b
```

- This works for any number of features, not just 2. Instead of writing out all multiplications and additions, we let NumPy handle the vector dot product.

Matrix Form with Transpose

- For a single data point $\mathbf{x}_i$ (a column vector) and weights $\mathbf{w}$:

$$z = \mathbf{w}^\top \mathbf{x}_i + b$$

Matrix Form with Transpose

- For a single data point $\mathbf{x}_i$ (a column vector) and weights $\mathbf{w}$:

$$z = \mathbf{w}^\top \mathbf{x}_i + b$$

- Here, $\mathbf{w}^\top$ is a row vector and $\mathbf{x}_i$ is a column vector. Their multiplication gives a scalar.

Matrix Form with Transpose

- For a single data point $\mathbf{x}_i$ (a column vector) and weights $\mathbf{w}$:

$$z = \mathbf{w}^\top \mathbf{x}_i + b$$

- Here, $\mathbf{w}^\top$ is a row vector and $\mathbf{x}_i$ is a column vector. Their multiplication gives a scalar.
- In NumPy, we usually keep $\mathbf{x}_i$ as a 1D array, but we can also write it explicitly with shapes:

```
import numpy as np

x_i = np.array([[2.0],
                [3.0]])    # shape (2,1) column vector
w   = np.array([[0.5],
                [-1.0]])   # shape (2,1) column vector

z = w.T @ x_i + b   # matrix multiplication with transpose
```

Matrix Form with Transpose

- For a single data point $\mathbf{x}_i$ (a column vector) and weights $\mathbf{w}$:

$$z = \mathbf{w}^\top \mathbf{x}_i + b$$

- Here, $\mathbf{w}^\top$ is a row vector and $\mathbf{x}_i$ is a column vector. Their multiplication gives a scalar.
- In NumPy, we usually keep $\mathbf{x}_i$ as a 1D array, but we can also write it explicitly with shapes:

```
import numpy as np

x_i = np.array([[2.0],
                [3.0]])    # shape (2,1) column vector
w   = np.array([[0.5],
                [-1.0]])   # shape (2,1) column vector

z = w.T @ x_i + b   # matrix multiplication with transpose
```

- This is mathematically the same as the dot product, just written in matrix notation.

Perceptron Learning Algorithm (pseudocode)

Algorithm 1: Perceptron Learning

Input: Training set $\{(x_i, y_i)\}_{i=1}^{n}$, $y_i \in \{-1, +1\}$; learning rate $\eta > 0$

Output: Weights w, bias b

Initialize $w \leftarrow 0$, $b \leftarrow 0$

for *epoch* $= 1, 2, \ldots$ **do**

    $errors \leftarrow 0$

    for $i = 1$ **to** n **do**

        $z \leftarrow w^\top x_i + b$

        $\hat{y} \leftarrow \mathrm{sign}(z)$

        if $\hat{y} \neq y_i$ **then**

            $w \leftarrow w + \eta\, y_i\, x_i$

            $b \leftarrow b + \eta\, y_i$

            $errors \leftarrow errors + 1$

    if $errors = 0$ **then**

        break

My trivial implementation, v3

```python
import numpy as np

# Dataset
X = np.array([[2, 2],
              [4, 0],
              [2, 0],
              [0, 2]])

# Labels in {-1, +1} instead of {0,1}
y = np.array([+1, +1, -1, -1])

# Hyperparameters
epochs = 20
learning_rate = 1

# Initialization
w = np.array([0.0, 1.0])
b = -1.0
```

My trivial implementation, v3

```python
# Activation function: sign
def activation(z):
    return np.where(z > 0, 1, -1)


for epoch in range(epochs):
    errors = 0
    for i in range(len(X)):
        x_i = X[i]
        y_i = y[i]

        # Vector notation
        z = x_i @ w + b
        y_hat = activation(z)

        # Update if misclassified
        if y_i != y_hat:
            w = w + learning_rate * y_i * x_i
            b = b + learning_rate * y_i
            errors += 1
```

My trivial implementation, v3

```
    if errors == 0:
        print(f"Converged after {epoch+1} epochs")
        break

print("Final weights:", w)
print("Final bias:", b)
```

Lab Time: implement a Perceptron Class in NumPy

- **Init:** store weights w, bias b, learning rate, and activation function.

Lab Time: implement a Perceptron Class in NumPy

- **Init:** store weights w, bias b, learning rate, and activation function.
- **Modes:** `train()` and `eval()` switch training/eval mode.

Lab Time: implement a Perceptron Class in NumPy

- **Init:** store weights w, bias b, learning rate, and activation function.
- **Modes:** `train()` and `eval()` switch training/eval mode.
- **Forward:** compute $z = x@w + b$.

Lab Time: implement a Perceptron Class in NumPy

- **Init:** store weights w, bias b, learning rate, and activation function.
- **Modes:** `train()` and `eval()` switch training/eval mode.
- **Forward:** compute $z = x@w + b$.
- **Predict:** apply activation to z.

Lab Time: implement a Perceptron Class in NumPy

- **Init:** store weights w, bias b, learning rate, and activation function.
- **Modes:** `train()` and `eval()` switch training/eval mode.
- **Forward:** compute $z = x@w + b$.
- **Predict:** apply activation to z.
- **Fit step:** loop over data, update $w \leftarrow w + lr \cdot y_i x_i$, $b \leftarrow b + lr \cdot y_i$ if misclassified.

Lab Time: implement a Perceptron Class in NumPy

- **Init:** store weights w, bias b, learning rate, and activation function.
- **Modes:** `train()` and `eval()` switch training/eval mode.
- **Forward:** compute $z = x@w + b$.
- **Predict:** apply activation to z.
- **Fit step:** loop over data, update $w \leftarrow w + lr \cdot y_i x_i$, $b \leftarrow b + lr \cdot y_i$ if misclassified.
- **Usage:** create model, train for epochs with `fit_step`, then use `predict` on new points.

My trivial implementation, v4

```
import numpy as np

class Perceptron:
    def __init__(self, w, b, lr, activation):
        self.w = w
        self.b = b
        self.lr = lr
        self.training = True
        self.activation = activation

    def set_activation(self, activation):
        self.activation = activation

    def train(self):
        self.training = True

    def eval(self):
        self.training = False

    def forward(self, x):
```

My trivial implementation, v4

```
        return x @ self.w + self.b

    def predict(self, x):
        return self.activation(self.forward(x))

    def fit_step(self, X, y):
        errors = 0
        for i in range(len(X)):
            x_i, y_i = X[i], y[i]
            y_hat = self.predict(x_i)
            if y_i != y_hat:
                self.w += self.lr * y_i * x_i
                self.b += self.lr * y_i
                errors += 1
        return errors

def activation(z):
    s = np.sign(z)
    return -1 if s == 0 else s
```

My trivial implementation, v4

```
X = np.array([[2,2],[4,0],[2,0],[0,2]])
y = np.array([+1,+1,-1,-1])

w = np.array([0, 1])
b = -1
learning_rate = 1

model = Perceptron(w, b, lr=learning_rate, activation=activation)

model.train()
for epoch in range(20):
    if model.fit_step(X, y) == 0:
        break

model.eval()
X_new = np.array([[1,1],[3,1],[0,0]])
preds = [model.predict(x) for x in X_new]
print("Preds:", preds)
print("Weights:", model.w, "Bias:", model.b)
```

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.
- **Steps:**

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.
- **Steps:**
  1. Generate 1000 points from a linear function (e.g. sample x from a normal distribution, compute $z = x@w^\star + b^\star$, assign $y = \mathrm{sign}(z)$).

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.
- **Steps:**
  1. Generate 1000 points from a linear function (e.g. sample x from a normal distribution, compute $z = x@w^\star + b^\star$, assign $y = \mathrm{sign}(z)$).
  2. Use 800 points for training and 200 for testing.

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.
- **Steps:**
  1. Generate 1000 points from a linear function (e.g. sample x from a normal distribution, compute $z = x@w^\star + b^\star$, assign $y = \text{sign}(z)$).
  2. Use 800 points for training and 200 for testing.
  3. Train your perceptron model on the training set.

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.
- **Steps:**
  1. Generate 1000 points from a linear function (e.g. sample x from a normal distribution, compute $z = x@w^{\star} + b^{\star}$, assign $y = \mathrm{sign}(z)$).
  2. Use 800 points for training and 200 for testing.
  3. Train your perceptron model on the training set.
  4. Evaluate on the test set and count the number of errors.

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.
- **Steps:**
  1. Generate 1000 points from a linear function (e.g. sample x from a normal distribution, compute $z = x@w^\star + b^\star$, assign $y = \mathrm{sign}(z)$).
  2. Use 800 points for training and 200 for testing.
  3. Train your perceptron model on the training set.
  4. Evaluate on the test set and count the number of errors.
- **Hint:** Use `np.random.normal` for sampling points.

Lab Time: Train and Evaluate a Perceptron

- **Task:** Implement an experiment with the Perceptron class.
- **Steps:**
  1. Generate 1000 points from a linear function (e.g. sample x from a normal distribution, compute $z = x@w^\star + b^\star$, assign $y = \text{sign}(z)$).
  2. Use 800 points for training and 200 for testing.
  3. Train your perceptron model on the training set.
  4. Evaluate on the test set and count the number of errors.
- **Hint:** Use `np.random.normal` for sampling points.
- **Explore:** Try changing the train/test split (e.g. 100/900, 200/800) and see how performance varies.

My trivial implementation, v5

```
# ----- 1) Generate 1000 points from a linear function -----
rng = np.random.default_rng(42)
n = 1000
d = 2

# True separating function: y = sign(w* x + b)
w_true = np.array([1.5, -0.8])
b_true = 0.2

X = rng.normal(0, 1.0, size=(n, d))
z = X @ w_true + b_true
y = np.where(z > 0, 1, -1)

# ----- 2) Split: 800 train / 200 test -----
idx = rng.permutation(n)
train_idx, test_idx = idx[:800], idx[800:]
X_train, y_train = X[train_idx], y[train_idx]
X_test,  y_test  = X[test_idx],  y[test_idx]

# ----- 3) Train the model -----
w0 = np.zeros(d)
```

My trivial implementation, v5

```python
b0 = 0.0
lr = 1.0
model = Perceptron(w=w0, b=b0, lr=lr, activation=activation)

model.train()
max_epochs = 50
for epoch in range(max_epochs):
    errs = model.fit_step(X_train, y_train)
    if errs == 0:
        print(f"Converged in {epoch+1} epochs")
        break

# ----- 4) Evaluate on the 200 test points -----
model.eval()
y_pred = np.array([model.predict(x) for x in X_test])
errors = int(np.sum(y_pred != y_test))
acc = 1 - errors / len(y_test)

print("Test errors:", errors, f"/ {len(y_test)}")
print("Test accuracy:", acc)
print("Weights:", model.w, "Bias:", model.b)
```

Key Takeaways

- Perceptron = linear classifier with simple mistake-driven updates.
- Intuitive geometric effect: rotate/shift boundary to fix errors.
- Converges in finite steps if data are linearly separable with margin.

Next Time

- Multilayer Networks (MLP): forward pass and activations.
- Loss functions.
- From linear to non-linear decision boundaries.

Appendix: PLA convergence proof

- Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$ with $y_i \in \{+1, -1\}$.

Appendix: PLA convergence proof

- Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$ with $y_i \in \{+1, -1\}$.
- Model prediction: $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

Appendix: PLA convergence proof

- Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ with $y_i \in \{+1, -1\}$.
- Model prediction: $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.
- Absorb the bias: augment $\mathbf{x}_i \leftarrow (\mathbf{x}_i, 1)$ and $\mathbf{w} \leftarrow (\mathbf{w}, b)$.

Appendix: PLA convergence proof

- Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ with $y_i \in \{+1, -1\}$.
- Model prediction: $\hat{y} = \mathrm{sign}(\mathbf{w} \cdot \mathbf{x})$.
- Absorb the bias: augment $\mathbf{x}_i \leftarrow (\mathbf{x}_i, 1)$ and $\mathbf{w} \leftarrow (\mathbf{w}, b)$.
- Linear separability with margin: there exists a *unit* $\mathbf{w}^\star$ and $\gamma > 0$ such that

$$y_i \left(\mathbf{w}^\star \cdot \mathbf{x}_i\right) \geq \gamma \quad \forall i, \qquad \text{and} \qquad \|\mathbf{x}_i\| \leq R.$$

Appendix: PLA convergence proof

- Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$ with $y_i \in \{+1, -1\}$.
- Model prediction: $\hat{y} = \operatorname{sign}(\mathbf{w} \cdot \mathbf{x})$.
- Absorb the bias: augment $\mathbf{x}_i \leftarrow (\mathbf{x}_i, 1)$ and $\mathbf{w} \leftarrow (\mathbf{w}, b)$.
- Linear separability with margin: there exists a *unit* $\mathbf{w}^\star$ and $\gamma > 0$ such that

$$y_i \left(\mathbf{w}^\star \cdot \mathbf{x}_i\right) \geq \gamma \quad \forall i, \qquad \text{and} \qquad \|\mathbf{x}_i\| \leq R.$$

- We assume already known: each update moves the separator in the correct direction for the mistaken point.

Normalize First: Put Data Inside the Unit Ball

- Let $R = \max_i \|\mathbf{x}_i\| > 0$. Define rescaled points

$$\tilde{\mathbf{x}}_i = \frac{\mathbf{x}_i}{R} \quad \Rightarrow \quad \|\tilde{\mathbf{x}}_i\| \leq 1.$$

Normalize First: Put Data Inside the Unit Ball

- Let $R = \max_i \|\mathbf{x}_i\| > 0$. Define rescaled points

$$\tilde{\mathbf{x}}_i = \frac{\mathbf{x}_i}{R} \quad \Rightarrow \quad \|\tilde{\mathbf{x}}_i\| \le 1.$$

- **Multiplying both sides by a positive constant preserves inequalities.** Since $y_i(\mathbf{w}^\star \cdot \mathbf{x}_i) \ge \gamma$ and $\frac{1}{R} > 0$, if we multiply both sides by $\frac{1}{R}$, then

$$y_i\left(\mathbf{w}^\star \cdot \tilde{\mathbf{x}}_i\right) \ge \frac{\gamma}{R} =: \tilde{\gamma}.$$

Normalize First: Put Data Inside the Unit Ball

- Let $R = \max_i \|\mathbf{x}_i\| > 0$. Define rescaled points

$$\tilde{\mathbf{x}}_i = \frac{\mathbf{x}_i}{R} \quad \Rightarrow \quad \|\tilde{\mathbf{x}}_i\| \le 1.$$

- **Multiplying both sides by a positive constant preserves inequalities.** Since $y_i(\mathbf{w}^\star \cdot \mathbf{x}_i) \ge \gamma$ and $\frac{1}{R} > 0$, if we multiply both sides by $\frac{1}{R}$, then

$$y_i\left(\mathbf{w}^\star \cdot \tilde{\mathbf{x}}_i\right) \ge \frac{\gamma}{R} =: \tilde{\gamma}.$$

- Thus, after a harmless rescaling, we may **assume** $\|\mathbf{x}_i\| \le 1$ (unit ball) and margin γ possibly replaced by $\tilde{\gamma}$. For clarity below we work with $\|\mathbf{x}_i\| \le 1$.

Mistake Condition & Update

- A mistake at $(\mathbf{x}_i, y_i)$ means $\hat{y} \neq y_i$.

Mistake Condition & Update

- A mistake at $(\mathbf{x}_i, y_i)$ means $\hat{y} \neq y_i$.
- Equivalently, $y_i(\mathbf{w}_t \cdot \mathbf{x}_i) \leq 0$.
  Why? Because $\hat{y} = \text{sign}(\mathbf{w}_t \cdot \mathbf{x}_i)$ and $y_i \in \{\pm 1\}$, so **if we multiply both sides** of $(\mathbf{w}_t \cdot \mathbf{x}_i)$'s sign by y_i (which is ± 1), the inequality direction is preserved and misclassification becomes $y_i(\mathbf{w}_t \cdot \mathbf{x}_i) \leq 0$.

Mistake Condition & Update

- A mistake at $(\mathbf{x}_i, y_i)$ means $\hat{y} \neq y_i$.
- Equivalently, $y_i(\mathbf{w}_t \cdot \mathbf{x}_i) \leq 0$.
  Why? Because $\hat{y} = \mathrm{sign}(\mathbf{w}_t \cdot \mathbf{x}_i)$ and $y_i \in \{\pm 1\}$, so **if we multiply both sides** of $(\mathbf{w}_t \cdot \mathbf{x}_i)$'s sign by y_i (which is ± 1), the inequality direction is preserved and misclassification becomes $y_i(\mathbf{w}_t \cdot \mathbf{x}_i) \leq 0$.
- PLA update on a mistake:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \, \mathbf{x}_i.$$

Progress Lemma: Alignment Increases by at least γ

- If $(\mathbf{x}_i, y_i)$ is misclassified at time t:

$$\mathbf{w}_{t+1} \cdot \mathbf{w}^\star \;=\; \big(\mathbf{w}_t + y_i \mathbf{x}_i\big) \cdot \mathbf{w}^\star \;=\; \mathbf{w}_t \cdot \mathbf{w}^\star \;+\; y_i \big(\mathbf{x}_i \cdot \mathbf{w}^\star\big).$$

Progress Lemma: Alignment Increases by at least γ

- If $(\mathbf{x}_i, y_i)$ is misclassified at time t:

$$\mathbf{w}_{t+1} \cdot \mathbf{w}^\star \;=\; \big(\mathbf{w}_t + y_i \mathbf{x}_i\big) \cdot \mathbf{w}^\star \;=\; \mathbf{w}_t \cdot \mathbf{w}^\star \;+\; y_i\big(\mathbf{x}_i \cdot \mathbf{w}^\star\big).$$

- By the margin assumption, $y_i(\mathbf{x}_i \cdot \mathbf{w}^\star) \geq \gamma$.

Progress Lemma: Alignment Increases by at least γ

- If $(\mathbf{x}_i, y_i)$ is misclassified at time t:

$$\mathbf{w}_{t+1} \cdot \mathbf{w}^\star \;=\; \big(\mathbf{w}_t + y_i \mathbf{x}_i\big) \cdot \mathbf{w}^\star \;=\; \mathbf{w}_t \cdot \mathbf{w}^\star \;+\; y_i \big(\mathbf{x}_i \cdot \mathbf{w}^\star\big).$$

- By the margin assumption, $y_i(\mathbf{x}_i \cdot \mathbf{w}^\star) \geq \gamma$.
- **Adding** the nonnegative quantity $y_i(\mathbf{x}_i \cdot \mathbf{w}^\star)$ $(\geq \gamma)$ to $\mathbf{w}_t \cdot \mathbf{w}^\star$ gives

$$\mathbf{w}_{t+1} \cdot \mathbf{w}^\star \;\geq\; \mathbf{w}_t \cdot \mathbf{w}^\star \;+\; \gamma.$$

Progress Lemma: Alignment Increases by at least γ

- If $(\mathbf{x}_i, y_i)$ is misclassified at time t:

$$\mathbf{w}_{t+1}\cdot\mathbf{w}^\star \;=\; \big(\mathbf{w}_t + y_i\mathbf{x}_i\big)\cdot\mathbf{w}^\star \;=\; \mathbf{w}_t\cdot\mathbf{w}^\star \;+\; y_i\big(\mathbf{x}_i\cdot\mathbf{w}^\star\big).$$

- By the margin assumption, $y_i(\mathbf{x}_i\cdot\mathbf{w}^\star) \geq \gamma$.
- **Adding** the nonnegative quantity $y_i(\mathbf{x}_i\cdot\mathbf{w}^\star)$ $(\geq \gamma)$ to $\mathbf{w}_t\cdot\mathbf{w}^\star$ gives

$$\mathbf{w}_{t+1}\cdot\mathbf{w}^\star \;\geq\; \mathbf{w}_t\cdot\mathbf{w}^\star \;+\; \gamma.$$

- After T mistakes (updates), by iterating the inequality:

$$\mathbf{w}_T\cdot\mathbf{w}^\star \;\geq\; \mathbf{w}_0\cdot\mathbf{w}^\star \;+\; T\gamma. \quad \text{With } \mathbf{w}_0 = \mathbf{0}: \;\; \mathbf{w}_T\cdot\mathbf{w}^\star \geq T\gamma.$$

Norm Growth Lemma

- On a mistake,

$$\|\mathbf{w}_{t+1}\|^2 = \|\mathbf{w}_t\|^2 + 2y_i\left(\mathbf{w}_t \cdot \mathbf{x}_i\right) + \|\mathbf{x}_i\|^2.$$

Norm Growth Lemma

- On a mistake,
$$\|\mathbf{w}_{t+1}\|^2 = \|\mathbf{w}_t\|^2 + 2y_i\left(\mathbf{w}_t \cdot \mathbf{x}_i\right) + \|\mathbf{x}_i\|^2.$$
- Since it is a mistake, $y_i(\mathbf{w}_t \cdot \mathbf{x}_i) \leq 0$, hence
$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + \|\mathbf{x}_i\|^2 \leq \|\mathbf{w}_t\|^2 + 1 \quad (\text{because } \|\mathbf{x}_i\| \leq 1).$$

Norm Growth Lemma

- On a mistake,

$$\|\mathbf{w}_{t+1}\|^2 = \|\mathbf{w}_t\|^2 + 2y_i\,(\mathbf{w}_t\cdot\mathbf{x}_i) + \|\mathbf{x}_i\|^2.$$

- Since it is a mistake, $y_i(\mathbf{w}_t\cdot\mathbf{x}_i) \leq 0$, hence

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + \|\mathbf{x}_i\|^2 \leq \|\mathbf{w}_t\|^2 + 1 \quad (\text{because } \|\mathbf{x}_i\| \leq 1).$$

- **Now sum both sides over the T mistakes:**

$$\sum_{t=0}^{T-1}\left(\|\mathbf{w}_{t+1}\|^2 - \|\mathbf{w}_t\|^2\right) \leq \sum_{t=0}^{T-1} 1 = T.$$

Norm Growth Lemma

- On a mistake,

$$\|\mathbf{w}_{t+1}\|^2 = \|\mathbf{w}_t\|^2 + 2y_i\,(\mathbf{w}_t \cdot \mathbf{x}_i) + \|\mathbf{x}_i\|^2.$$

- Since it is a mistake, $y_i(\mathbf{w}_t \cdot \mathbf{x}_i) \leq 0$, hence

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + \|\mathbf{x}_i\|^2 \leq \|\mathbf{w}_t\|^2 + 1 \quad (\text{because } \|\mathbf{x}_i\| \leq 1).$$

- **Now sum both sides over the T mistakes:**

$$\sum_{t=0}^{T-1} \left(\|\mathbf{w}_{t+1}\|^2 - \|\mathbf{w}_t\|^2\right) \leq \sum_{t=0}^{T-1} 1 = T.$$

- The left-hand side is a **telescoping sum**:

$$\|\mathbf{w}_T\|^2 - \|\mathbf{w}_0\|^2 \leq T. \quad \Rightarrow \quad \|\mathbf{w}_T\| \leq \sqrt{T} \quad (\text{with } \mathbf{w}_0 = \mathbf{0}).$$

Combine & Conclude (Normalized and General Forms)

- By Cauchy–Schwarz with $\|\mathbf{w}^\star\| = 1$:

$$T\gamma \;\le\; \mathbf{w}_T\cdot\mathbf{w}^\star \;\le\; \|\mathbf{w}_T\| \;\le\; \sqrt{T}.$$

Combine & Conclude (Normalized and General Forms)

- By Cauchy–Schwarz with $\|\mathbf{w}^\star\| = 1$:

$$T\gamma \;\le\; \mathbf{w}_T \cdot \mathbf{w}^\star \;\le\; \|\mathbf{w}_T\| \;\le\; \sqrt{T}.$$

- Therefore (under $\|\mathbf{x}_i\| \le 1$): $\quad T \;\le\; \dfrac{1}{\gamma^2}.$

Combine & Conclude (Normalized and General Forms)

- By Cauchy–Schwarz with $\|\mathbf{w}^\star\| = 1$:

$$T\gamma \;\leq\; \mathbf{w}_T \cdot \mathbf{w}^\star \;\leq\; \|\mathbf{w}_T\| \;\leq\; \sqrt{T}.$$

- Therefore (under $\|\mathbf{x}_i\| \leq 1$): $\quad T \;\leq\; \dfrac{1}{\gamma^2}.$
- **Undoing the rescaling (general case)**: if the original data satisfy $\|\mathbf{x}_i\| \leq R$, the same argument yields

$$T \;\leq\; \left(\frac{R}{\gamma}\right)^2.$$

Combine & Conclude (Normalized and General Forms)

- By Cauchy–Schwarz with $\|\mathbf{w}^\star\| = 1$:

$$T\gamma \;\le\; \mathbf{w}_T\!\cdot\!\mathbf{w}^\star \;\le\; \|\mathbf{w}_T\| \;\le\; \sqrt{T}.$$

- Therefore (under $\|\mathbf{x}_i\| \le 1$): $\quad T \;\le\; \dfrac{1}{\gamma^2}.$
- **Undoing the rescaling (general case)**: if the original data satisfy $\|\mathbf{x}_i\| \le R$, the same argument yields

$$T \;\le\; \left(\frac{R}{\gamma}\right)^2.$$

- Only finitely many mistakes $\Rightarrow$ only finitely many updates.
  Conclusion: on linearly separable data, the PLA *converges*.

Thanks!