Chapter 2: Training Simple Machine Learning Algorithms for Classification

Created	@August 14, 2025 5:51 PM
Class	ML with Pytorch and Scikit Learn

Artificial neurons

“Instead of trying to produce a program to simulate the adult mind, why not rather try to produce one which simulates the child’s?” -**Alan Turing**

The Perceptron Learning Rule

The principle behind the perceptron is of a simple Artificial Neuron. It’s steps include:

1. Taking inputs (x)

2. Use random small numbers or 0 to initialize the Weights and bias
3. Then compute the y hat value for x(i)

$$\mathbf{w} \cdot \mathbf{x} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

4. Update the bias and weight value only IF the prediction is wrong

Unlike a well known function like sigmoid or gradient descent it strictly predicts either 0 or 1.

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The complete formula for perceptron is as follows

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

- η = learning rate

$$\Delta b = \eta \left(y^{(i)} - \hat{y}^{(i)} \right)$$

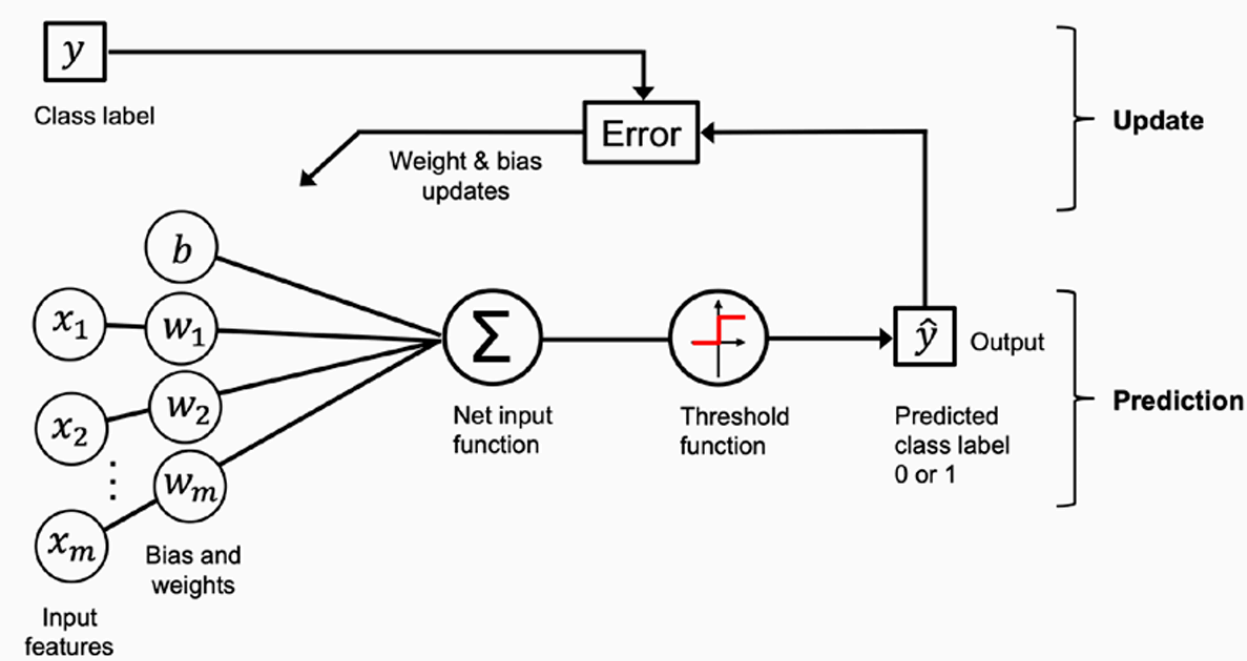

Figure 2.4: Weights and bias of the model are updated based on the error function

*NOTE: Perceptron can only guarantee convergence if the 2 classes and linearly separable ie A straight line can fit through them.

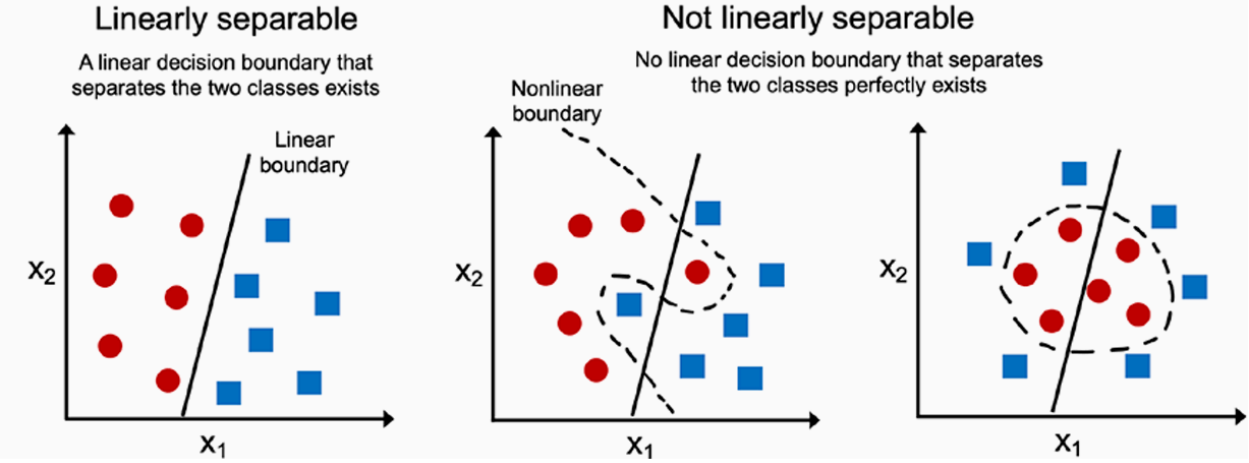

Figure 2.3: Examples of linearly and nonlinearly separable classes

CODE IMPLEMENTATION

Imports:

```
import numpy as np
#yeah that's it
```

Perceptron Code:

```
class Perceptron:
    def __init__(self, l_r=0.01, epochs=200, random_state=1):
        # l_r : learning rate (controls how much weights are adjusted each step)
        # epochs : number of passes over the training dataset
        # random_state : seed for reproducibility of weight initialization
        self.l_r = l_r
        self.epochs = epochs
        self.random_state = random_state

    def fit(self, X, y):
        """
        Fit training data.
        X : {array-like}, shape = [n_samples, n_features]
        y : array-like, shape = [n_samples] (target labels)
        """

        # Initialize a random number generator with a fixed seed
        regen = np.random.RandomState(self.random_state)

        # Initialize weights (small random numbers, mean=0, std=0.01)
        # Number of weights = number of features in X
        self.w_ = regen.normal(loc=0.0, scale=0.01, size=X.shape[1])

        # Initialize bias as 0
        self.b_ = np.float64(0.)

        # To store the number of misclassifications in each epoch
        self.errors = []
```

```
        # Training process over all epochs
        for _ in range(self.epochs):
            errors = 0
            # Loop over each training sample and its label
            for xi, target in zip(X, y):
                # Update rule: learning_rate * (target - prediction)
                update = self.l_r * (target - self.predict(xi))

                # Adjust weights and bias
                self.w_ += update * xi
                self.b_ += update

                # Count if there was a misclassification (non-zero update)
                errors += int(update != 0.0)

            # Store total errors for this epoch
            self.errors.append(errors)

        return self

    def net_input(self, X):
        """Calculate the weighted sum (linear combination of inputs + bias)."""
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        """
        Return class label:
        If net_input >= 0 → predict 1
        Else → predict 0
        (This is the step/threshold function)
        """
        return np.where(self.net_input(X) >= 0.0, 1, 0)
```

Final Testing:

```
X = np.array([
    [2.0, 3.0],
    [1.0, 5.0],
    [2.5, 1.0],
    [0.5, 0.5],
    [3.0, 2.0]
])

# Labels: 0 or 1
y = np.array([1, 1, 0, 0, 1])

ppn = perceptron(l_r=0.01, epochs=50, random_state=1)
ppn.fit(X, y)

ppn.predict(X)
```

Adaptive Linear Neurons and Convergence

We will use the Adaptive Linear Neuron algorithm (ADALINE) published by Bernard Widrow and his doctoral student Tedd Hoff only a few years after Rosenblatt's perceptron algorithm.

Adaline Algorithm

It serves as an improvement on the existing Perceptron model as instead of using the unit step function to make predictions it uses a linear activation function then a threshold function to make the final predictions

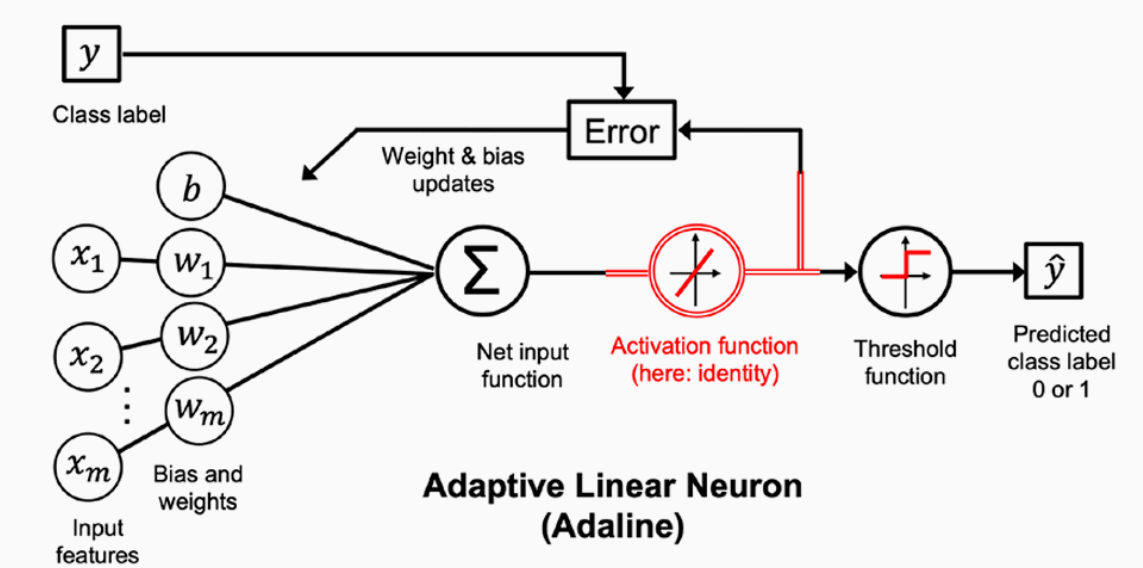

Figure 2.9: A comparison between a perceptron and the Adaline algorithm

In order for our algorithm to learn patterns and adjust it's data to update the parameters (weights, bias) we need to guide it through a goal.

This goal is defined in form of an objective function. For example, if we want to minimize weights and Bias we use a cost/loss function

In case of Adaline, it uses the popular MSE (Mean Squared Error) cost function :-

$$L(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^{m} \left(t^{(i)} - y^{(i)}\right)^2$$

$$y^{(i)} = \mathbf{w} \cdot \mathbf{x}^{(i)} + b$$

It then uses ***Gradient Descent Algorithm*** in order to minimize the loss:

$$w_j := w_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right) x_j^{(i)}$$

$$b := b - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)$$

It then finally does the classification using a threshold function instead of binary classification like in Perceptron

$$\hat{y}_{class}^{(i)} = \begin{cases} 1 & \text{if } \hat{y}^{(i)} \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

CODE IMPLEMENTATION

```
import numpy as np

class adaline:
    def __init__(self, eta=0.01, epochs=100, random_state=1):
        self.eta = eta
        self.epochs = epochs
        self.random_state = random_state

    def fit(self, X,y):

        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=X.shape[1])
        self.b_ = np.float64(0.)
        self.losses_ = []

        for i in range(self.epochs):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)

            self.w_ += self.eta *2.0* X.T.dot(errors) /X.shape[0]
            self.b_ += self.eta *2.0* errors.mean()

            loss = (errors **2).mean()
            self.losses_.append(loss)
```

```
        return self
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
      "Compute linear activation"""
      return X

    def predict(self, X):
        return np.where(self.activation(self.net_input(X))
                        >= 0.5, 1, 0)


X = np.array([
    [2.0, 3.0],
    [1.0, 5.0],
    [2.5, 1.0],
    [0.5, 0.5],
    [3.0, 2.0]
])

# Labels: 0 or 1
y = np.array([1, 1, 0, 0, 1])
```

Improving Gradient descent through Normalization

- Adaline uses **gradient descent**.
- If features have very different scales (e.g., height in cm vs. weight in kg), gradients can become unbalanced.
- Standardization helps the algorithm converge **faster and more reliably**.

$$x' = (x - \mu)/\sigma$$

where:

- μ = mean of the feature
- σ = standard deviation of the feature

CODE EXAMPLE:

```
# --- Training Data Standardization ---
X_s = np.copy(X)
X_s[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_s[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()

# --- Test Data Standardization ---
X_v = np.copy(X_test)
X_v[:, 0] = (X_test[:, 0] - X_test[:, 0].mean()) / X_test[:, 0].std()
X_v[:, 1] = (X_test[:, 1] - X_test[:, 1].mean()) / X_test[:, 1].std()
```

Large-scale Machine Learning and Stochastic Gradient Descent

When we are working with large datasets with millions of features and inputs (common in ML) performing ML on the whole dataset is costly computationally.

Hence, we tend use SGD which simply uses one random training example instead of the whole dataset to compute gradient

A different type of SGD is mini batch gradient which instead of one divide the dataset into different batches (32, 64) and computes them instead

ADVANTAGES:

- Faster convergence
- Can generalize and learn better
- Can escape local minima

Formula:

$$L(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^{m} \left(t^{(i)} - y^{(i)}\right)^2,$$

$$\text{SGD:} \quad \mathbf{w} := \mathbf{w} - \alpha \left(y^{(i)} - t^{(i)}\right) x^{(i)},$$

$$\text{with } y^{(i)} = \mathbf{w}^\top x^{(i)}.$$

CODE IMPLEMENTATION:

```
# ADAptive LInear NEuron (Adaline) with Stochastic Gradient Descent (SGD)

class AdalineSGD:

    def __init__(self, eta=0.01, epochs=100, shuffle=True, random_state=1):
        """
        Parameters:
        -----------
        eta : float
            Learning rate (controls how much weights are updated each step)
        epochs : int
            Number of passes (iterations) over the training dataset
        shuffle : bool
            Whether to shuffle training data every epoch (prevents cycles in learni
ng)
        random_state : int
            Random seed for reproducibility of results
        """
        self.eta = eta
        self.epochs = epochs
        self.shuffle = shuffle
        self.random_state = random_state
        self.w_initialized = False   # To check if weights are initialized

    def fit(self, X, y):
        """
        Fit training data using stochastic gradient descent.
```

```
        Parameters:
        -----------
        X : array-like, shape = [n_samples, n_features]
          Training vectors
        y : array-like, shape = [n_samples]
          Target values

        Returns:
        --------
        self : object
        """

        # Initialize weights with random small numbers
        self._initialize_weights(X.shape[1])

        self.losses_ = []  # Store average loss per epoch

        # Loop over training epochs
        for i in range(self.epochs):

            # Shuffle training data if enabled
            if self.shuffle:
                X, y = self._shuffle(X, y)

            losses = []  # Track losses for each sample in this epoch

            # Loop through each training sample
            for xi, target in zip(X, y):
                losses.append(self._update_weights(xi, target))  # Update weights fo
r each sample

            # Compute average loss for the epoch
            avg_loss = np.mean(losses)
            self.losses_.append(avg_loss)
```

```
        return self

    def partial_fit(self, X, y):
        """
        Update model without reinitializing weights.
        Useful for online learning with streaming data.
        """

        # Initialize weights if not done already
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])

        # If multiple samples, iterate through them
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:  # If only one sample
            self._update_weights(X, y)

        return self

    def _shuffle(self, X, y):
        """Shuffle training data to avoid cycles during learning"""
        r = self.rgen.permutation(len(y))  # Random permutation of indices
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights with small random numbers"""
        self.rgen = np.random.RandomState(self.random_state)  # Random gener
ator
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=m)  # Random weigh
ts
        self.b_ = np.float_(0.)  # Bias initialized to zero
        self.w_initialized = True

    def _update_weights(self, xi, target):
```

```
        """
        Apply Adaline learning rule to update weights.

        Parameters:
        -----------
        xi : array-like, shape = [n_features]
            Training sample
        target : float
            True class label
        """

        # Compute model output
        output = self.activation(self.net_input(xi))

        # Calculate error (difference between true value and prediction)
        error = (target - output)

        # Update weights and bias using Adaline rule (gradient descent step)
        self.w_ += self.eta * 2.0 * xi * error
        self.b_ += self.eta * 2.0 * error

        # Compute squared error loss
        loss = error**2
        return loss
```

End of Chapter 2.