Artificial Neural Networks

Prof. Dr. Sen Cheng

Nov 25, 2019

Problem Set 8: Perceptron

Tutors: Eloy Parra Barrero (eloy.parrabarrero@rub.de), Olya Hakobyan (olya.hakobyan@rub.de)

1. Load the Iris dataset from *sklearn.datasets* (at sci-kit learn's website you can find an example for how to load and display the data). The dataset contains measures of the flowers of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured for each sample: the length and the width of the sepals and petals respectively, in centimeters. Extract the first two features (septal length and width) and make a scatter plot from them. Use two different colors to distinguish Iris setosa samples from the rest. Is the data linearly separable?

2. Implement the perceptron algorithm from scratch. You can follow the pseudocode suggested below. Note that, unlike on the version described in the lecture notes, we are recalculating the overall classification error after every weight update. This is somewhat inefficient, but for the sake of this tutorial, it is interesting to see the evolution of the error at a finer scale. Then, use your perceptron to classify Iris setosa based on septal length and width. Plot the resulting decision boundary together with the data points.

Algorithm 0.1 Perceptron Algorithm - Tutorial variant

Require: X, Y
initialize $w = 0$ or small random values
errors = count total number of misclassified items in training set
for epoch = 0 **to** maximum allowed epochs **do**
  for all data pairs i **do**
    if $y_i \hat{y}_i = -1$ **then**
      update weights using the rule: $w \leftarrow w + 2\eta y_i x_i$
      errors = count total number of misclassified items in training set
    end if
    if errors $\leq$ maximum allowed errors **then**
      return solution found!
    end if
  end for
end for

3. Plot the training error after each iteration of the algorithm. Think about what could happen if your design employed a limited number of iterations.

4. Explore the effect of the learning rate on the speed of convergence when you start with $w = 0$. Draw a conclusion.

5. Extract the first and fourth features (septal length and petal width) and make a scatter plot showing Iris virginica vs the rest. What would happen if a perceptron tried to classify Iris virginica?

6. Implement the pocket algorithm, which returns the best set of weights encountered during training instead of the last solution. For each iteration, plot the training error as well as the error for the best solution seen so far. In what way is this modification of the algorithm helpful?

7. (Bonus Question) In the dataset, the samples are ordered by species. However, convergence can be achieved faster if the samples are shuffled. Try running the algorithm after shuffling the data points once in the beginning, and/or shuffling at the beginning of each epoch. Compare the number of iterations it takes for the basic perceptron algorithm to find a solution in each of these cases, and contrast them to the results you got before. You can also look at the effect of shuffling on the pocket perceptron by plotting the evolution of the best classification error over iterations. In order to get more meaningful results, train the classifiers a number of times and plot the mean and standard deviations of the indicated variables.

Solutions

Core packages and the code for implementing the basic perceptron algorithm and the pocket algorithm:

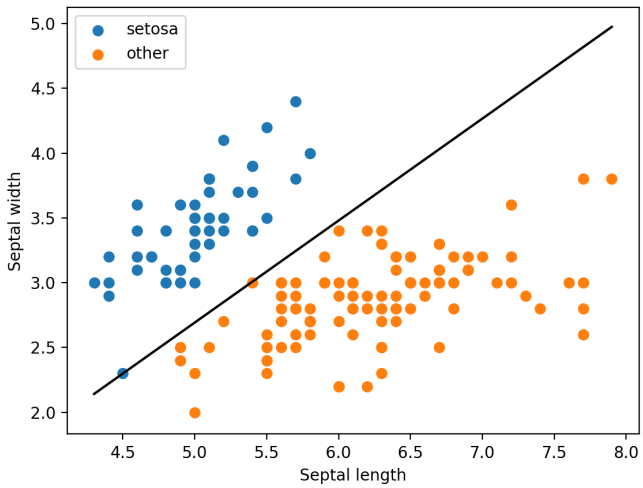
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets


class Perceptron:
    def __init__(self):
        self.weights = None
        self.current_training_error = None
        self.training_errors = None

    def train(self, X, Y, learning_rate, max_epochs, max_error, shuffle=True):
        self.weights = np.zeros(X.shape[1])  # Initialize weights
        self.current_training_error = self.count_errors(X, Y)
        self.training_errors = []

        # Run the perceptron algorithm
        for epoch_num in range(max_epochs):
            if shuffle:
                shuffled_indices = np.random.permutation(len(X))
                X = X[shuffled_indices]
                Y = Y[shuffled_indices]
            for x, y in zip(X, Y):
                self.training_step(X, Y, x, y, learning_rate)
                if self.current_training_error <= max_error:
                    print(f"Solution found at epoch {epoch_num}")
                    return

    def training_step(self, X, Y, x, y, learning_rate):
        y_predicted = 1 if np.dot(self.weights, x) > 0 else -1
        if y * y_predicted == -1:
            self.weights += 2 * learning_rate * y * x
            self.current_training_error = self.count_errors(X, Y)
        self.training_errors.append(self.current_training_error)

    def count_errors(self, X, Y):
        Y_predicted = np.where(np.dot(X, self.weights) > 0, 1, -1)
        return np.count_nonzero(Y*Y_predicted == -1)

    def plot_training_errors(self):
        fig, ax = plt.subplots()
        ax.plot(self.training_errors)
        ax.set_xlabel("Iteration")
        ax.set_ylabel("Error")
        return ax

    def plot_2D_decision_boundary(self, ax, X):
        x1 = np.linspace(X[:, 1].min(), X[:, 1].max(), 2)
        ax.plot(x1, -self.trained_weights()[0] / self.trained_weights()[2]
                - self.trained_weights()[1] / self.trained_weights()[2] * x1, 'k-')

    def trained_weights(self):
        return self.weights


class PocketPerceptron(Perceptron):
    def __init__(self):
```

```
        super().__init__()
        self.weights_in_pocket = None
        self.smallest_training_error = np.inf
        self.smallest_training_errors = []

    def training_step(self, X, Y, x, y, learning_rate):
        super().training_step(X, Y, x, y, learning_rate)
        if self.training_errors[-1] < self.smallest_training_error:
            self.weights_in_pocket = np.copy(self.weights)
            self.smallest_training_error = self.training_errors[-1]
        self.smallest_training_errors.append(self.smallest_training_error)

    def trained_weights(self):
        return self.weights_in_pocket

    def plot_training_errors(self):
        ax = super().plot_training_errors()
        ax.plot(self.smallest_training_errors, 'r', label='Best solution')
        ax.legend()
```

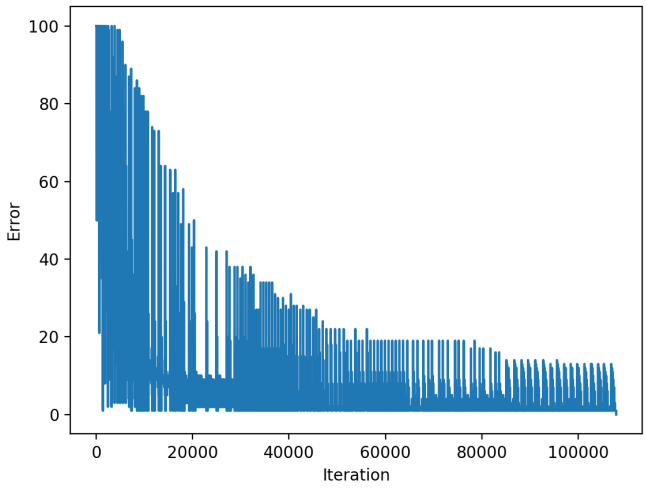
Classify Iris setosa vs the rest:

```
import os
from Perceptron import *


# Create a folder for saving the figures
if not os.path.exists("figures"):
    os.makedirs("figures")

# Load the dataset
iris_dataset = datasets.load_iris()
X = iris_dataset.data[:, :2]  # we only take the first two features
X = np.hstack((np.ones((X.shape[0], 1)), X))  # Add x0 = 1 for bias term
Y = np.where(iris_dataset.target == 0, 1, -1) # reassign labels

# Display the dataset
fig, ax = plt.subplots()
ax.scatter(X[Y == 1, 1], X[Y == 1, 2], label="setosa")
ax.scatter(X[Y == -1, 1], X[Y == -1, 2], label="other")
ax.set_xlabel("Septal length")
ax.set_ylabel("Septal width")
ax.legend()

# Train the basic perceptron
model = Perceptron()
model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=False)
print(f"{model.count_errors(X, Y)} classification errors in training set")
model.plot_2D_decision_boundary(ax, X)
fig.savefig("figures/setosa-scatter.png", dpi=200)
model.plot_training_errors()
plt.savefig("figures/setosa-error.png", dpi=200)
plt.show()
```

1-2. The data is linearly separable, so the perceptron is guaranteed to converge. The scatter plot of septal width vs length, together with the resulting decision boundary should look like this:



3. If you terminate the classic perceptron before convergence, it will return the last solution. However, because the training error oscillates across iterations, you are not guaranteed to have the best solution encountered during training if you do so.



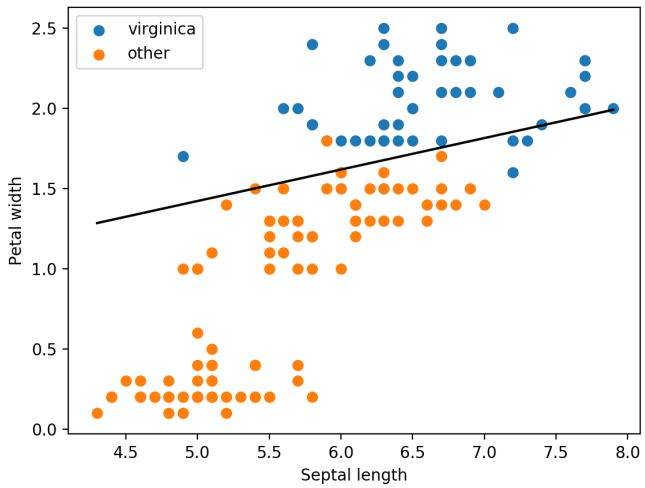
4. If weights are initialized at zero, the learning rate has no effect since only the direction of the weight vector counts.

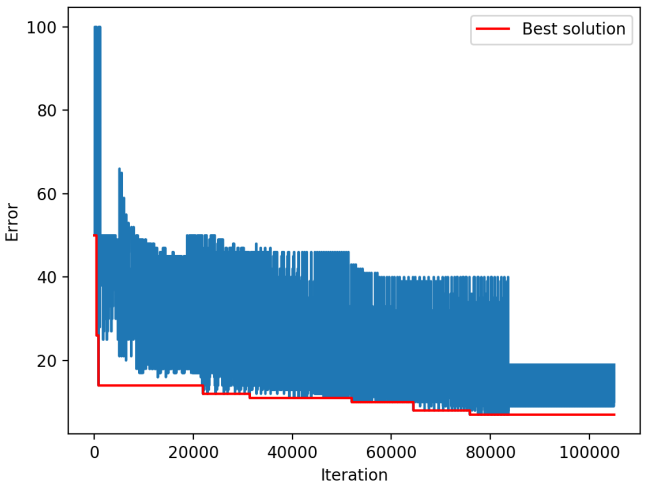
Classify Iris Virginica vs the rest:

```
from Perceptron import *


# Load the dataset
iris_dataset = datasets.load_iris()
X = iris_dataset.data[:, (0, 3)]
X = np.hstack((np.ones((X.shape[0], 1)), X))  # Add x0 = 1 for bias term
Y = np.where(iris_dataset.target == 2, 1, -1)

# Display the dataset
fig, ax = plt.subplots()
ax.scatter(X[Y == 1, 1], X[Y == 1, 2], label="virginica")
ax.scatter(X[Y == -1, 1], X[Y == -1, 2], label="other")
ax.set_xlabel("Septal length")
```

```
ax.set_ylabel("Petal width")
ax.legend()

# Train the pocket perceptron
model = PocketPerceptron()
model.train(X, Y, learning_rate=1, max_epochs=700, max_error=0, shuffle=False)
print(f"{model.smallest_training_errors[-1]} errors in the training set")
model.plot_2D_decision_boundary(ax, X)
fig.savefig("figures/virginica-scatter.png", dpi=200)
model.plot_training_errors()
plt.savefig("figures/virginica-error.png", dpi=200)
plt.show()
```

5-6. Iris virginica cannot be linearly separated from the other two types of Iris when using the septal length and petal width. However, since there are only a few outliers affecting linear separability, we would still like to be able to classify correctly most of the points.
If we used the classic perceptron, the algorithm would never converge, and if we terminated the algorithm after some maximum number of iterations, we could encounter the problem discussed in exercise 3.
The pocket algorithm can help us in this case by returning the best solution encounterd after a fixed number of iterations:



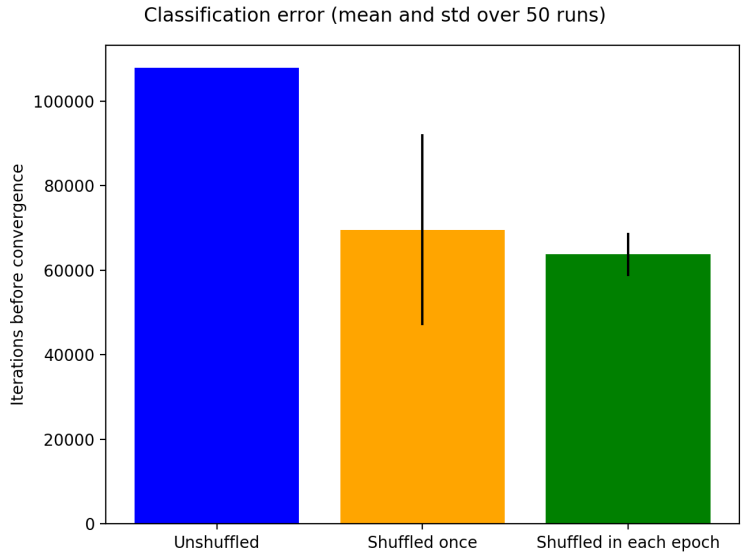


7. With this code we compare the convergence of the basic and the pocket perceptron algorithms without shuffling, shuffling once before training, or shuffling before each training epoch.

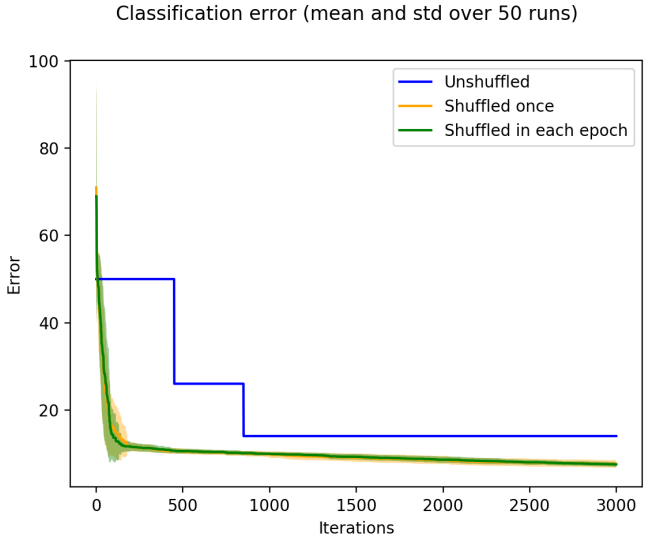
```
from Perceptron import *


# Study the effect of shuffling on the basic perceptron

iris_dataset = datasets.load_iris()
X = iris_dataset.data[:, :2]  # we only take the first two features
X = np.hstack((np.ones((X.shape[0], 1)), X))  # Add x0 = 1 for bias term
Y = np.where(iris_dataset.target == 0, 1, -1)  # reassign labels

model = Perceptron()
model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=False)
conv_unshuffled = (len(model.training_errors))
conv_shuffled = []
conv_shuffled_once = []
N = 50

for i in range(N):
    print(f"Run number {i}")
    model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=True)
    conv_shuffled.append(len(model.training_errors))
    shuffled_indices = np.random.permutation(len(X))
    X = X[shuffled_indices]
    Y = Y[shuffled_indices]
    model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=False)
    conv_shuffled_once.append(len(model.training_errors))

fig, ax = plt.subplots(constrained_layout=True)
ax.bar(1, conv_unshuffled, color='b')
ax.bar(2, np.mean(conv_shuffled_once), yerr=np.std(conv_shuffled_once), color='orange')
ax.bar(3, np.mean(conv_shuffled), yerr=np.std(conv_shuffled), color='g')
ax.set_xticks(np.arange(1, 4))
ax.set_xticklabels(['Unshuffled', 'Shuffled once', 'Shuffled in each epoch'])
ax.set_ylabel('Iterations before convergence')
fig.suptitle(f"Classification error (mean and std over {N} runs)")
fig.savefig("figures/shuffled-basic.png", dpi=200)


# Study the effect of shuffling on pocket perceptron

iris_dataset = datasets.load_iris()
X = iris_dataset.data[:, (0, 3)]
X = np.hstack((np.ones((X.shape[0], 1)), X))  # Add x0 = 1 for bias term
Y = np.where(iris_dataset.target == 2, 1, -1)

N = 50
num_epochs = 20

model_unshuffled = PocketPerceptron()
model_unshuffled.train(X, Y, learning_rate=1, max_epochs=num_epochs,
                       max_error=0, shuffle=False)
unshuffled = model_unshuffled.smallest_training_errors

shuffled = []
shuffled_once = []

for i in range(N):
```

```
        print(f"Run number {i}")
        model_shuffled = PocketPerceptron()
        model_shuffled_once = PocketPerceptron()
        model_shuffled.train(X, Y, learning_rate=1, max_epochs=num_epochs, max_error=0)
        shuffled.append(model_shuffled.smallest_training_errors)
        shuffled_indices = np.random.permutation(len(X))
        X = X[shuffled_indices]
        Y = Y[shuffled_indices]
        model_shuffled_once.train(X, Y, learning_rate=1, max_epochs=num_epochs,
                                  max_error=0, shuffle=False)
        shuffled_once.append(model_shuffled_once.smallest_training_errors)

shuffled_mean = np.mean(shuffled, axis=0)
shuffled_std = np.std(shuffled, axis=0)
shuffled_once_mean = np.mean(shuffled_once, axis=0)
shuffled_once_std = np.std(shuffled_once, axis=0)
fig, ax = plt.subplots()
ax.plot(unshuffled, color='b', label="Unshuffled")
ax.fill_between(np.arange(shuffled_once_mean.size),
                shuffled_once_mean - shuffled_once_std,
                shuffled_once_mean + shuffled_once_std,
                facecolor='orange', alpha=0.4)
ax.plot(shuffled_once_mean, color='orange', label="Shuffled once")
ax.fill_between(np.arange(shuffled_mean.size), shuffled_mean - shuffled_std,
                shuffled_mean + shuffled_std,
                facecolor='g', alpha=0.4)
ax.plot(shuffled_mean, color='g', label="Shuffled in each epoch")
ax.set_xlabel("Iterations")
ax.set_ylabel("Error")
ax.legend()
fig.suptitle(f"Classification error (mean and std over {N} runs)")
fig.savefig("figures/shuffled-pocket.png", dpi=200)

plt.show()
```

For the basic perceptron algorithm, we plot the number of iterations before a solution is found for the unshuffled case, and the mean and standard deviation of 50 runs for the cases where the training set is shuffled once at the beginning, or before each training epoch. As you can see, shuffling the data leads to faster convergence.



For the pocket perceptron, we plot the evolution of the best classification error found over iterations. Again, we compare the unshuffled case with the mean and standard deviation of 50 runs for the two shuffling scenarios.


Classification error (mean and std over 50 runs)
Unshuffled
Shuffled once
Shuffled in each epoch
Error
Iterations
100
80
60
40
20
0
500
1000
1500
2000
2500
3000