Batch Data Analytics

201374125

Part 1: Perceptron

A perceptron is a classification algorithm that derives a hyperplane of p-dimensional space to separate a set of external stimuli into two classes C_1 or C_2. This hyperplane is the decision boundary and as such correct classification relies on linearly separable data.

Error-correction learning is used for training, adjusting weights to position the decision boundary. For example, for a training dataset $D_{train} = \{(\mathbf{x}_1, d_1), \ldots, (\mathbf{x}_s, d_s)\}$ with s samples, an n-dimensional input vector $\mathbf{x}_j$, and the desired output $\mathbf{d}_j$ where $\mathbf{d}_j \in \{C_1, C_2\}$. The perceptron attempts to find a vector $\mathbf{w}$ which satisfies the inequalities,

$$\left\{ \begin{array}{ll} \mathbf{w}^\top \mathbf{x} > 0 & \forall \mathbf{x} \in \mathrm{D}_1 \\ \mathbf{w}^\top \mathbf{x} \leq 0 & \forall \mathbf{x} \in \mathrm{D}_2 \end{array} \right\} \text{ where: } \mathrm{D}_1 \cap \mathrm{D}_2 = \emptyset, \mathrm{D}_1 \cup \mathrm{D}_2 = \mathrm{D}_{\text{train}}$$

Steps

1. **Initialisation:** Set $t = 1$, $w_1 = 0$ (or small random numbers), learning rate $\eta \in (0, 1]$
2. **Activation:** Apply the sample input $\mathbf{x}_t$ to the neuron
3. **Response:** Compute its response:

$$\mathrm{y(t)} = \text{signum}\left[\mathbf{w}(\mathrm{t})^{\mathrm{T}} \mathbf{x}(\mathrm{t})\right]$$

4. **Adaptation:** Update current weight vector via:

$$\mathbf{w}(\mathrm{t}+1) = \mathbf{w}(\mathrm{t}) + \eta \underbrace{[\mathrm{d(t)} - \mathrm{y(t)}}_{\text{error signal} \in \{0,+2\}} \mathbf{x}(\mathrm{t})$$

$$\mathrm{d(t)} = \left\{ \begin{array}{ll} +1 & \text{if } \mathbf{x}(\mathrm{t}) \text{ belongs to } \mathrm{C}_1 \\ -1 & \text{if } \mathbf{x}(\mathrm{t}) \text{ belongs to } \mathrm{C}_2 \end{array} \right.$$

5. **Continuation:** Until all samples classified correctly, or maximum epochs reaches set $t = t + 1$ and go to step 2.

The perceptron architecture was implemented from scratch using the `Python` programming language with the `numpy` library to assist with working with numerical arrays. This code is shown on the following page:

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


class Perceptron(object):
    def __init__(self, num_epochs: int, lr: float) -> None:
        self.num_epochs = num_epochs
        self.lr = lr

    def predict(self, input: np.ndarray) -> int:
        activation = np.dot(input, self.weights[1:]) + self.weights[0]
        return 1 if activation > 0 else 0

    def train(self, data: np.ndarray, labels: np.ndarray) -> None:
        self.data = data
        self.labels = labels
        self.weights = np.random.rand(self.data.shape[1] + 1)

        for epoch in range(self.num_epochs):
            accuracy = 0
            assert len(self.data) == len(self.labels)
            for idx, _ in enumerate(self.data):
                prediction = self.predict(self.data[idx])

                self.weights[1:] += (
                    self.lr * (self.labels[idx] - prediction) * self.data[idx]
                )
                self.weights[0] += self.lr * (self.labels[idx] - prediction)

                accuracy += 1 if prediction == self.labels[idx] else 0
            print(
                f"Epoch {epoch + 1}\n"
                f"Weights updated: {self.weights[1:3]}...\n"
                f"Bias updated: {self.weights[0]}\n"
                f"Accuracy: {(accuracy / len(self.data)) * 100}%\n"
            )

    def plot(self) -> None:
        y_intercept: tuple[int, int] = (-self.weights[0] / self.weights[2], 0)
        m: int = -(self.weights[0] / self.weights[2]) / (
            self.weights[0] / self.weights[1]
        )
        axes = plt.axes(
            xlim=(min(self.data[:, 0]), max(self.data[:, 0])),
            ylim=(min(self.data[:, 1]), max(self.data[:, 1])),
        )
        x_vals = np.array(axes.get_xlim())
        y_vals = y_intercept + m * x_vals
        plt.plot(x_vals, y_vals)
        plt.scatter(self.data[:, 0], self.data[:, 1], c=self.labels)
        plt.show()
```

The `Perceptron` class is initialised with the user specified max number of epochs and the learning rate. In this implementation, the number of desired inputs and outputs are determined automatically based on the shape of the input data and labels. The `predict` method implements the activation function

$$\mathrm{g(z)} = \begin{cases} 1, & \text{if } z \geq 0 \\ -1, & \text{otherwise} \end{cases},$$

and

$$z = \sum_{j=0}^{m} x_j w_j + b = w^T x + b,$$

here, w is the weight vector and x is a sample vector from the training dataset, and b is a bias value:

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

In this implementation the bias is incorporated as w_0.

Weights are updated in the `train` method over a user-specified number of epochs. For each sample x^i in the training vector, weights are updated $w_j = w_j + \Delta w_j$ where

$$\Delta w_j = \eta \left(\text{ target }^i - \text{ output }^i \right) x_j^i$$

here, η is the learning rate.

The `plot` method takes the first two features of the input data, the hyperplane derived from the learned weights, and plots them in relation to each other. This allows the decision boundary to be visualised.

The following code demonstrates the Perceptron, initialised with a maximum of 10 epochs, and a learning rate of 0.01.

```
import numpy as np

from perceptron import Perceptron
from sklearn.datasets import make_blobs

perceptron = Perceptron(num_epochs=10, lr=0.01)
```

First the perceptron is trained on randomly generated linearly separable data with 1,000 samples, and two output features:

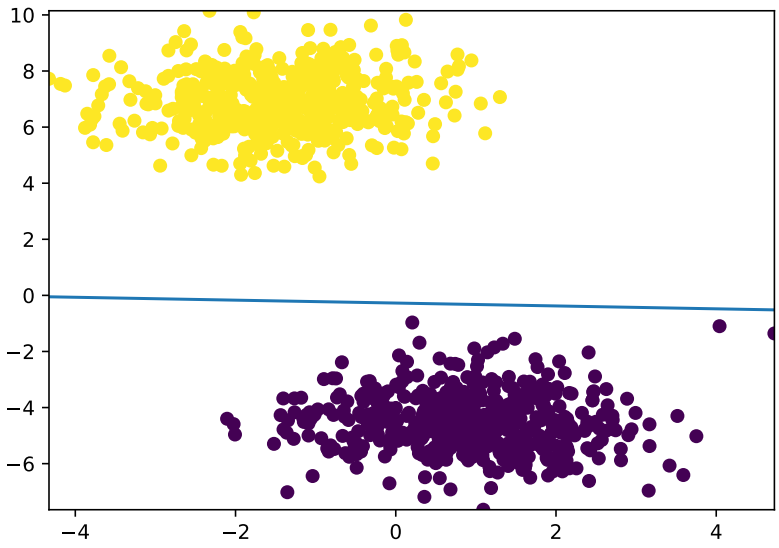
```
X, y = make_blobs(n_samples=1000, centers=2, n_features=2, random_state=100)
perceptron.train(X, y)
```

```
Epoch 1
Weights updated: [0.53292939 0.94687167]...
Bias updated: 0.6641993769678701
Accuracy: 99.5%

Epoch 2
Weights updated: [0.42118558 0.99178154]...
Bias updated: 0.6341993769678701
```

```
Accuracy: 99.7%

Epoch 3
Weights updated: [0.3335103  1.01633601]...
Bias updated: 0.6141993769678701
Accuracy: 99.8%

Epoch 4
Weights updated: [0.24583503 1.04089048]...
Bias updated: 0.5941993769678701
Accuracy: 99.8%

Epoch 5
Weights updated: [0.15815975 1.06544495]...
Bias updated: 0.57419937696787
Accuracy: 99.8%

Epoch 6
Weights updated: [0.11774485 1.07642904]...
Bias updated: 0.56419937696787
Accuracy: 99.9%

Epoch 7
Weights updated: [0.11774485 1.07642904]...
Bias updated: 0.56419937696787
Accuracy: 100.0%

Epoch 8
Weights updated: [0.11774485 1.07642904]...
Bias updated: 0.56419937696787
Accuracy: 100.0%

Epoch 9
Weights updated: [0.11774485 1.07642904]...
Bias updated: 0.56419937696787
Accuracy: 100.0%

Epoch 10
Weights updated: [0.11774485 1.07642904]...
Bias updated: 0.56419937696787
Accuracy: 100.0%
```

The following plot visualises the decision boundary for this trained perceptron.

```
perceptron.plot()
```



This model may now be used to make a classification prediction.

```
perceptron.predict(np.array([8, -2]))
```

```
0
```

Here is an example with a much large dataset, and demonstrates the ability for the Perceptron to automatically scale with the number of features.

```
X, y = make_blobs(n_samples=10_000, centers=2, n_features=100, random_state=100)
perceptron.train(X, y)
```

```
Epoch 1
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 99.99%

Epoch 2
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 3
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 4
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 5
```

```
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 6
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 7
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 8
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 9
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%

Epoch 10
Weights updated: [0.39644207 0.73694285]...
Bias updated: 0.7835567171912546
Accuracy: 100.0%
```

Finally, here is a dataset that isn't linearly separable, so accuracy is poor.

```
X, y = make_blobs(n_samples=10_000, centers=4, n_features=2, random_state=100)
perceptron.train(X, y)
```

```
Epoch 1
Weights updated: [-98.30677353 114.98321516]...
Bias updated: 77.67988589516935
Accuracy: 49.89%

Epoch 2
Weights updated: [-198.13250443  227.53225995]...
Bias updated: 155.31988589517132
Accuracy: 49.99%

Epoch 3
Weights updated: [-297.1464604   340.78511322]...
Bias updated: 232.85988589517908
Accuracy: 49.99%

Epoch 4
Weights updated: [-396.06436472  454.10684023]...
Bias updated: 310.38988589513207
Accuracy: 49.99%
```

```
Epoch 5
Weights updated: [-495.10338995  567.3132963 ]...
Bias updated: 387.91988589506155
Accuracy: 49.980000000000004%

Epoch 6
Weights updated: [-593.78058733  680.85671618]...
Bias updated: 465.4098858949911
Accuracy: 49.980000000000004%

Epoch 7
Weights updated: [-692.2564885   794.58482457]...
Bias updated: 542.8798858949207
Accuracy: 49.980000000000004%

Epoch 8
Weights updated: [-790.89358072  908.15556626]...
Bias updated: 620.3698858948502
Accuracy: 49.980000000000004%

Epoch 9
Weights updated: [-889.60577417 1021.65127206]...
Bias updated: 697.8698858947797
Accuracy: 49.980000000000004%

Epoch 10
Weights updated: [-988.30910266 1135.15154787]...
Bias updated: 775.3698858947092
Accuracy: 49.980000000000004%
```

Part 2: Multi-layer Perceptron

Multi-layer perceptrons (MLP) have non-linear activation functions which are differentiable. While smooth activation functions like the sigmoid ($S(x) = \frac{1}{1+e^{-x}}$) were once popular, it is now more common to use the simpler ReLU ($\max(0, x)$) which is easier to compute and prevents issues like exploding or vanishing gradients. Unlike single-layer perceptrons (SLP), MLPs have one or more 'hidden' layers of neurons, which allows for more complex, non-linear relationships to be learnt, meaning MLPs can solve problems like *XOR* that are not solvable by SLPs. Each node can receive input from any other node in the previous layer, meaning they have high degrees of connectivity.

Backpropagation (BP) is used for training MLPs which computes the gradients of a cost function with respect to the weights in each neuron, which are then updated to minimise the loss. The chain rule is used to calculate gradients using derivatives, starting with the final layer and moving towards the first one.

Backpropagation may be split into a forward and backward pass:

- **Forward Pass**
  - Weights fixed
  - Start at first hidden layer, for all neurons calculate outputs and ILFs: $\mathrm{y_j(n)} = \phi_\mathrm{j}\left[\mathrm{v_j(n)}\right] \quad \mathrm{v_j(n)} = \sum_{\mathrm{i}-0}^{\mathrm{p}} \mathrm{w_{ji}(n)y_i(n)}$
  - Move to each layer until reaching output layer

- **Backward pass**
  - Start from output layer, working backwards to first hidden layer
  - All input signals to nodes are fixed
  - For each node in the current layer, calculate local gradient:
    - $\delta_j(n) = e_j(n)\phi'_j\left[v_j(n)\right]$ for output layer
    - $\delta_j(n) = \phi'_j\left[v_j(n)\right]\sum_k \delta_k(n)w_{kj}(n)$ for hidden layers
  - All local gradients are used to update weights using $\Delta \mathrm{w_{ji}(n)} = -\eta\frac{\partial \mathrm{E(n)}}{\partial \mathrm{w_{ji}(n)}} = \eta\delta_{\mathrm{j}}(\mathrm{n})\mathrm{y_i(n)}$

The following code snippet is an implementation of a MLP using python:

```
import numpy as np
from sklearn.datasets import make_blobs


def relu(x: np.ndarray) -> np.ndarray:
    return np.maximum(x, 0)


def error(A, y):
    return (np.mean(np.power(A - y, 2))) / 2


def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights) - 1):
        activation += weights[i] * inputs[i]
    return activation


def transfer_derivative(output):
    return output * (1.0 - output)


class MLP(object):
    def __init__(
        self,
        num_inputs,
        num_epochs: int,
        lr: float,
        hidden_layers=[1],
        num_outputs=2,
    ) -> None:
        self.num_inputs = num_inputs
        self.num_outputs = num_outputs
        self.hidden_layers = hidden_layers
        self.num_epochs = num_epochs
        self.lr = lr

        self.network = []
        self.input_weights = [
            {"weights": np.append(np.random.rand(self.num_inputs), 1)}
        ]
```

```
        self.hidden_weights = [
            {"weights": np.append(np.random.rand(i), 1)} for i in self.hidden_layers
        ]
        self.output_weights = [
            {"weights": np.append(np.random.rand(self.hidden_layers[-1]), 1)}
            for _ in range(self.num_outputs)
        ]
        self.network.append(self.input_weights)
        self.network.append(self.hidden_weights)
        self.network.append(self.output_weights)

    def forward(self, input):
        for layer in self.network:
            new_inputs = []
            for neuron in layer:
                activation = (
                    np.dot(neuron["weights"][:-1], input) + neuron["weights"][-1]
                )
                neuron["output"] = relu(activation)
                new_inputs.append(neuron["output"])
            input = new_inputs
        return input

    def backward(self, label):
        for i in reversed(range(len(self.network))):
            layer = self.network[i]
            errors = []
            if i != len(self.network) - 1:
                for j in range(len(layer)):
                    error = 0.0
                    for neuron in self.network[i + 1]:
                        error += neuron["weights"][j] * neuron["delta"]
                        errors.append(error)
            else:
                for j in range(len(layer)):
                    neuron = layer[j]
                    errors.append(label[j] - neuron["output"])
            for j in range(len(layer)):
                neuron = layer[j]
                neuron["delta"] = errors[j] * transfer_derivative(neuron["output"])

    def update_weights(self, point):
        for i in range(len(self.network)):
            inputs = point[:-1]
            if i != 0:
                inputs = [neuron["output"] for neuron in self.network[i - 1]]
            for neuron in self.network[i]:
                for j in range(len(inputs)):
                    neuron["weights"][j] += self.lr * neuron["delta"] * inputs[j]
                neuron["weights"][-1] += self.lr * neuron["delta"]
```

```
    def train(self, train, labels):
        for epoch in range(self.num_epochs):
            sum_error = 0
            for point, label in zip(train, labels):
                if type(label) != list():
                    label = [label]
                outputs = self.forward(point)
                sum_error += sum(
                    [(label[i] - outputs[i]) ** 2 for i in range(self.num_outputs)]
                )
                self.backward(label)
                self.update_weights(point)
            print(f">epoch={epoch}, error={sum_error}")


if __name__ == "__main__":
    X, y = make_blobs(n_samples=10, centers=2, n_features=2, random_state=100)  # type: ignore

    mlp = MLP(num_inputs=2, num_epochs=10, lr=0.001, num_outputs=1)
    mlp.train(train=X, labels=y)
```

There is a small bug that prevents this network from working unfortunately, so for use in part 3 a MLP is creating using pytorch:

```
import pytorch_lightning as pl
import torch
from torch import nn
from torch.utils.data.dataloader import DataLoader
from torchvision import transforms
from torchvision.datasets import CIFAR10


class MLP(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(32 * 32 * 3, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 10),
        )
        self.ce = nn.CrossEntropyLoss()

    def forward(self, x):
        return self.layers(x)

    def training_step(self, batch, batch_idx):
        x, y = batch
        x = x.view(x.size(0), -1)
        y_hat = self.forward(x)
        loss = self.ce(y_hat, y)
```

```
        self.log("train_loss", loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=1e-4)


if __name__ == "__main__":
    pl.seed_everything(42)
    dataset = CIFAR10(root="data/", transform=transforms.ToTensor(), download=True)
    mlp = MLP()
    trainer = pl.Trainer(
        logger=None, gpus=-1, auto_select_gpus=True, deterministic=False, max_epochs=5
    )
    trainer.fit(mlp, DataLoader(dataset, batch_size=128, num_workers=16))
```

Part 3: Genetic Algorithm

A genetic algorithm is a type of search or optimisation techniques to mimic elements from natural genetics. They use natural selection to select members of a population that perform well at a specific task.

1. Initialise a random starting population
2. Repeat until convergence is satisfied
    a. Evaluate the objective function F for all members
    b. Calculate the fitness values f of all members
    c. Perform parent selection based on their fitness probabilities
    d. Apply crossover to selected parents to generate offspring
    e. Apply mutation to offspring
    f. Replace some parents with the offspring and create a new population
3. Take the best individual and pass it as the final answer

The following code shows the ability to create a simple swarm genetic algorithm using the `deap` python library. `deap` is a modern evolutionary algorithm library which focusses on prototyping and explicit structures.

```
import math
import operator
import random

from deap import base
from deap import benchmarks
from deap import creator
from deap import tools
import numpy

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create(
    "Particle",
    list,
    fitness=creator.FitnessMax,
    speed=list,
    smin=None,
    smax=None,
```

```
    best=None,
)


def generate(size, pmin, pmax, smin, smax):
    part = creator.Particle(random.uniform(pmin, pmax) for _ in range(size))
    part.speed = [random.uniform(smin, smax) for _ in range(size)]
    part.smin = smin
    part.smax = smax
    return part


def updateParticle(part, best, phi1, phi2):
    u1 = (random.uniform(0, phi1) for _ in range(len(part)))
    u2 = (random.uniform(0, phi2) for _ in range(len(part)))
    v_u1 = map(operator.mul, u1, map(operator.sub, part.best, part))
    v_u2 = map(operator.mul, u2, map(operator.sub, best, part))
    part.speed = list(map(operator.add, part.speed, map(operator.add, v_u1, v_u2)))
    for i, speed in enumerate(part.speed):
        if abs(speed) < part.smin:
            part.speed[i] = math.copysign(part.smin, speed)
        elif abs(speed) > part.smax:
            part.speed[i] = math.copysign(part.smax, speed)
    part[:] = list(map(operator.add, part, part.speed))


toolbox = base.Toolbox()
toolbox.register("particle", generate, size=2, pmin=-6, pmax=6, smin=-3, smax=3)
toolbox.register("population", tools.initRepeat, list, toolbox.particle)
toolbox.register("update", updateParticle, phi1=2.0, phi2=2.0)
toolbox.register("evaluate", benchmarks.h1)


def main():
    pop = toolbox.population(n=5)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean)
    stats.register("std", numpy.std)
    stats.register("min", numpy.min)
    stats.register("max", numpy.max)

    logbook = tools.Logbook()
    logbook.header = ["gen", "evals"] + stats.fields

    GEN = 1000
    best = None

    for g in range(GEN):
        for part in pop:
            part.fitness.values = toolbox.evaluate(part)
            if not part.best or part.best.fitness < part.fitness:
```

```
                part.best = creator.Particle(part)
                part.best.fitness.values = part.fitness.values
            if not best or best.fitness < part.fitness:
                best = creator.Particle(part)
                best.fitness.values = part.fitness.values
        for part in pop:
            toolbox.update(part, best)

        # Gather all the fitnesses in one list and print the stats
        logbook.record(gen=g, evals=len(pop), **stats.compile(pop))
        print(logbook.stream)

    return pop, logbook, best


if __name__ == "__main__":
    main()
```

The `generate` function first creates the initial population based on the parameters used in `creator.Create("Particle", ...)` (1), this population has the ability to learn one parameter 'Speed' which updates based on `FitnessMax`. Particles are updated to optimise the speed.

`main` simulates population movement and calculates effectiveness using `tools.Statistics(...)`. Unfortunately I do not know how to integrate this code with MLPs.