jesús.rabanal-álvarez

April 9, 2024

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

EXERCISE 1 NEURAL NETWORKS

[2]:
```
class SingleLayerNN:
    def __init__(self, input_dim, seed=42):
        np.random.seed(seed)
        self.input_dim = input_dim
        self.weights = np.random.randn(input_dim)
        self.bias = np.random.randn(1)

    def sigmoid(self, x):
        return 1.0 / (1.0 + np.exp(-x))

    def sigmoid_der(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def f_propagation(self, inputs, validation=False):
        if validation == True:
            z = np.dot(inputs, self.weights) + self.bias
        else:
            z = np.dot(inputs, self.weights) + self.bias
        return self.sigmoid(z)

    def loss(self, predictions, targets):
        eps = 1e-15
        predictions = np.maximum(eps, np.minimum(1-eps, predictions))
        loss = np.linalg.norm(-(targets * np.log(predictions) + (1 - targets) *␣
↪np.log(1 - predictions)))
        return loss

    def b_propagation(self, inputs, predictions, targets, learning_rate):
        delta_w = learning_rate * np.dot(inputs.T, (targets - predictions))
        delta_b = np.mean(learning_rate * (targets - predictions))
        return delta_w, delta_b

    def update_weights_and_biases(self, delta_w, delta_b):
```

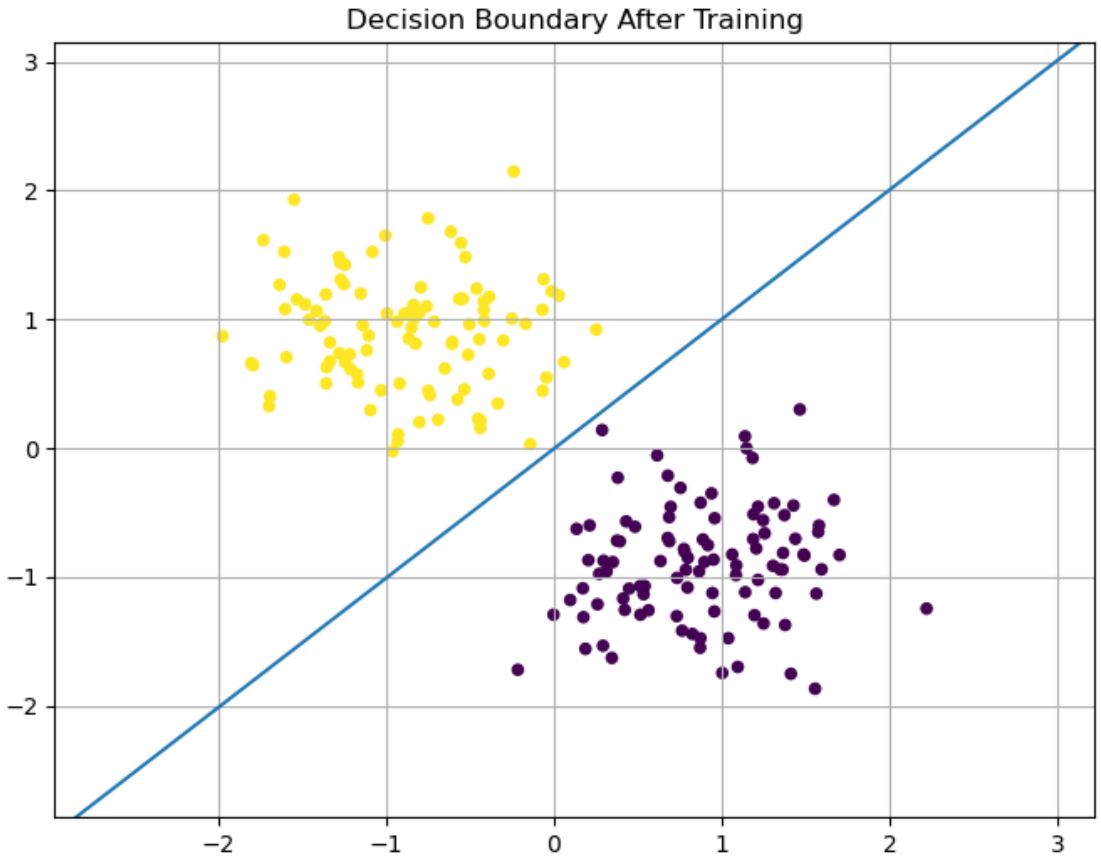
```
        self.weights = self.weights + delta_w
        self.bias = self.bias + delta_b

    def train(self, inputs, targets, validation_data_inputs,␣
↪validation_data_targets, epochs=1000, learning_rate=0.01, showIterations =␣
↪True):
        for i in range(epochs):
            #training
            predictions = self.f_propagation(inputs)

            loss = self.loss(predictions, targets)
            delta_w, delta_b = self.b_propagation(inputs, predictions, targets,␣
↪learning_rate)
            self.update_weights_and_biases(delta_w, delta_b)

            #validation
            val_predictions = self.f_propagation(validation_data_inputs,␣
↪validation=True)
            val_loss = self.loss(val_predictions, validation_data_targets)

            if showIterations == True:
                print('Iteration: '+str(i)+', Loss:'+str(loss)+', Validation␣
↪Loss:'+str(val_loss))

    def predict(self, x):
        predictions = self.f_propagation(x)
        return predictions

    def plot_decision_boundary(self, inputs, targets, title):
        x_min , x_max = inputs[:, 0]. min() - 1, inputs[:, 0]. max() + 1
        x_values = np.linspace(x_min , x_max , 200)

        y_values = ((-self.weights[0] / self.weights[1]) * x_values) - (self.
↪bias / self.weights[1])

        plt.figure(figsize=(8, 6))
        plt.scatter(inputs[:, 0], inputs[:, 1], c=targets, s=20)

        plt.plot(x_values, y_values, label='Decision Boundary')
        plt.xlim(x_min, x_max)
        plt.ylim(inputs[:, 1]. min() - 1, inputs[:, 1]. max() + 1)
        plt.title(title)
        plt.grid(True)
        plt.show()
```

[3]:
```
#create synthetic dataset
n_samples = 100
features_class_0 = np.random.randn(n_samples, 2) + [2, -2] # Class 0
features_class_1 = np.random.randn(n_samples, 2) + [-2, 2] # Class 1
inputs = np.vstack((features_class_0, features_class_1))
targets = np.hstack((np.zeros(n_samples), np.ones(n_samples)))

#standardize the features
inputs = (inputs - np.mean(inputs, axis=0)) / np.std(inputs, axis=0)

#create validation data
n_samples_val = 100
features_class_0_val = np.random.randn(n_samples_val, 2) + [2, -2]  # Class 0
features_class_1_val = np.random.randn(n_samples_val, 2) + [-2, 2]  # Class 1
inputs_val = np.vstack((features_class_0_val, features_class_1_val))
targets_val = np.hstack((np.zeros(n_samples_val), np.ones(n_samples_val)))

# Standardize the validation features
inputs_val = (inputs_val - np.mean(inputs_val, axis=0)) / np.std(inputs_val,␣
 ↪axis=0)
```

[4]:
```
model = SingleLayerNN(input_dim=2)
model.plot_decision_boundary(inputs, targets, title='Decision Boundary Before␣
 ↪Training')
```

[7]:
```
model_trained = SingleLayerNN(input_dim=2)
model_trained.train(inputs, targets, inputs_val, targets_val, epochs=10,␣
↪learning_rate=1)
model_trained.plot_decision_boundary(inputs, targets, title='Decision Boundary␣
↪After Training')
```

```
Iteration: 0, Loss:16.284488875397464, Validation Loss:10.032102003627465
Iteration: 1, Loss:8.573252019991854e-08, Validation Loss:10.032102001708308
Iteration: 2, Loss:8.573251942304992e-08, Validation Loss:10.032101999789152
Iteration: 3, Loss:8.573251853520006e-08, Validation Loss:10.032101997869995
Iteration: 4, Loss:8.573251775833144e-08, Validation Loss:10.032101995950839
Iteration: 5, Loss:8.573251698146282e-08, Validation Loss:10.032101994031683
Iteration: 6, Loss:8.573251620459419e-08, Validation Loss:10.032101992107476
Iteration: 7, Loss:8.573251542772556e-08, Validation Loss:10.03210199018832
Iteration: 8, Loss:8.57325145398757e-08, Validation Loss:10.032101988269163
Iteration: 9, Loss:8.573251376300708e-08, Validation Loss:10.032101986350007
```

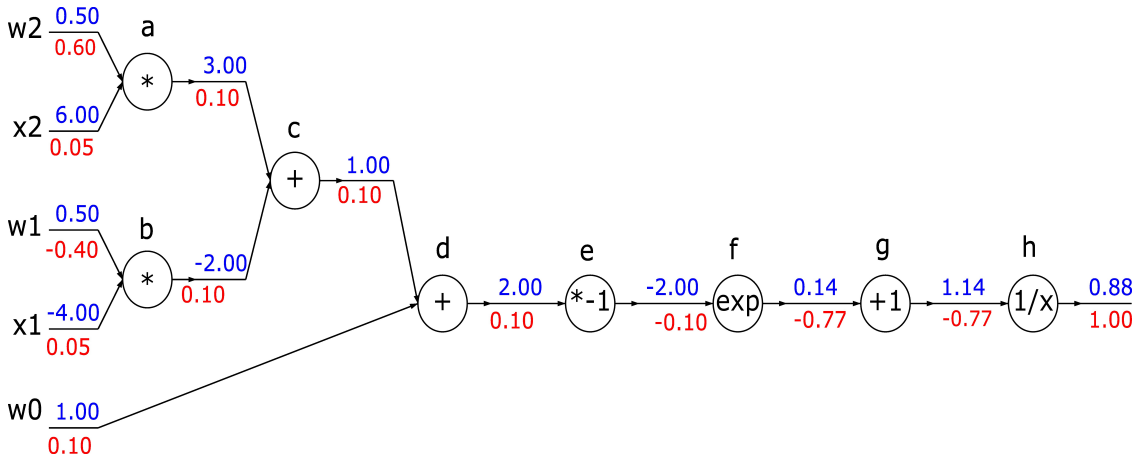


DISCUSSION IMPLEMENTATION AND RESULTS

- Before and after training, a visual examination of decision boundaries reveals significant disparities. The plot titled "Decision Boundary Before Training" illustrates the inability of non-updated weights and biases to delineate a coherent decision boundary. Conversely, the plot labeled "Decision Boundary After Training" demonstrates the model's attainment of a discernible decision boundary subsequent to weight and bias updates. - Furthermore, the selection of an optimal learning rate involved an iterative process, wherein various options were tested, and validation loss calculations were scrutinized. The learning rate plays a pivotal role in determining the speed and stability of convergence during training. Higher learning rates may lead to faster convergence but risk overshooting the optimal solution, while lower learning rates may necessitate longer training times but offer more stable convergence. Through this experimentation, it was determined that iteration 9 at a learning rate of 1, yielded the most satisfactory outcomes in terms of model performance. - The single-layer, single-neuron perceptron is effective for linearly separable datasets but struggles with non-linearly separable ones due to its linear decision boundary. This limitation arises from its inability to capture complex patterns, necessitating alternative models or feature representations for non-linearly separable datasets.

EXERCISE 2 BACKPROPAGATION

1. Calculate and report the local gradients.



For the $h(x) = 1/x$ with constant x, I used the derivative $h(x)' = -1/x^2$
$(-1/1.14^2) \cdot 1 = -0.77$

For the $g(x) = 1x + 1$ with constant x, I used the derivative $g(x)' = 1$
$1 \cdot (-0.77) = -0.77$

For the $f(x) = e^x$ with constant x, I used the derivative $f(x)' = e^x$
$(e^-2) \cdot (-0.77) = -0.10$

For the $e(x) = -1x$ with constant x, I used the derivative $e(x)' = -1$
$-1 \cdot (-0.10) = 0.10$

For the $d(x) = c + w0$ with constant $w0$, I used the derivative $d(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $d(x) = c + w0$ with constant c, I used the derivative $d(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $c(x) = a + b$ with constant a, I used the derivative $c(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $c(x) = a + b$ with constant b, I used the derivative $c(x)' = 1$
$0.10 \cdot 1.00 = 0.10$

For the $a(x) = w_2 \cdot x_2$ with constant w_2, I used the derivative $a(x)' = x_2$
$0.10 \cdot 6.00 = 0.60$

For the $a(x) = w_2 \cdot x_2$ with constant x_2, I used the derivative $a(x)' = w_2$
$0.10 \cdot 0.50 = 0.05$

For the $b(x) = w_1 \cdot x_1$ with constant w_1, I used the derivative $b(x)' = x_1$
$0.10 \cdot (-4.00) = -0.40$

For the $b(x) = w_1 \cdot x_1$ with constant x_1, I used the derivative $b(x)' = w_1$
$0.10 \cdot 0.50 = 0.05$

2. What patterns do you see for backpropagation ?

In the backpropagation that we have calculated we can see several patterns, in addition we get no variations due to derivative of the function given being 1 and in multiplication we can see a pattern

where the derivative of the function is the other term making them intertwined meaning a change in one is gonna affect the other. The gradients are routed through addition and multiplication operations according to the chain rule of calculus. The chain rule states how to compute the derivative of a composite function. For addition operations, the gradients are simply added together. For multiplication operations, the gradients are multiplied by the value of the other input. These rules are applied recursively as gradients are backpropagated through the network during training.

3. Assume we are running gradient descent…

$w_{new} = (1.00, 0.50, 0.50) - 0.1 \times (0.1, -0.4, 0.6) = (1.00, 0.50, 0.50) - (0.01, -0.04, 0.06) = (1.00 - 0.01, 0.50 - (-0.04), 0.50 - 0.06) = (0.99, 0.54, 0.44)$

Then the updated weights for $(w0, w1, w2)$ are $(0.99, 0.54, 0.44)$.

And local gradients for the input features in this specific example are not needed.