Lab: Backpropagation

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Backpropagation performs a backward pass to adjust the neural network's parameters. It's an algorithm that uses gradient descent to update the neural network weights.

Linear Regression via Batch Gradient Descent

Let $\boldsymbol{\theta}^{(t)} = (w^{(t)}, b^{(t)})$ be the parameter estimates of the tth iteration. Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ represents the training batch. Let mean squared error (MSE) be the loss/cost function $\mathcal{L}$.

Finding the Gradients

- **Step 1:** Write down $\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)})$ and $\hat{y}(x_i; \boldsymbol{\theta}^{(t)})$

$$\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}^{(t)}) = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) - y_i\right)^2$$
$$\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) = w^{(t)}x_i + b^{(t)}$$

- **Step 2:** Derive $\frac{\partial \mathcal{L}(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}$ and $\frac{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}{\partial \boldsymbol{\theta}^{(t)}}$

$$\frac{\partial \mathcal{L}(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})} = 2(\hat{y}(x_i; \boldsymbol{\theta}^{(t)}) - y_i)$$
$$\frac{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}{\partial w^{(t)}} = x_i$$
$$\frac{\partial \hat{y}(x_i; \boldsymbol{\theta}^{(t)})}{\partial b^{(t)}} = 1$$

- **Step 3:** Derive $\frac{\partial \mathcal{L}(\mathcal{D},\boldsymbol{\theta}^{(t)})}{\partial \boldsymbol{\theta}^{(t)}}$

$$\frac{\partial \mathcal{L}(\mathcal{D},\boldsymbol{\theta}^{(t)})}{\partial w^{(t)}} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}(\hat{y}(x_i;\boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i;\boldsymbol{\theta}^{(t)})} \frac{\partial \hat{y}(x_i;\boldsymbol{\theta}^{(t)})}{\partial w^{(t)}} = \frac{2}{N}\sum_{i=1}^{N} (\hat{y}(x_i;\boldsymbol{\theta}^{(t)}) - y_i) \cdot x_i \tag{1}$$

$$\frac{\partial \mathcal{L}(\mathcal{D},\boldsymbol{\theta}^{(t)})}{\partial b^{(t)}} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}(\hat{y}(x_i;\boldsymbol{\theta}^{(t)}), y_i)}{\partial \hat{y}(x_i;\boldsymbol{\theta}^{(t)})} \frac{\partial \hat{y}(x_i;\boldsymbol{\theta}^{(t)})}{\partial b^{(t)}} = \frac{2}{N}\sum_{i=1}^{N} (\hat{y}(x_i;\boldsymbol{\theta}^{(t)}) - y_i) \cdot 1 \tag{2}$$

Then, we initialise $\boldsymbol{\theta}^{(0)} = (w^{(0)}, b^{(0)})$ and then apply gradient descent for $t = 1, 2, ...$

$$w^{(t+1)} = w^{(t)} - \eta \cdot \frac{\partial \mathcal{L}(\mathcal{D},\boldsymbol{\theta}^{(t)})}{\partial w}\bigg|_{w^{(t)}} \tag{3}$$

$$b^{(t+1)} = b^{(t)} - \eta \cdot \frac{\partial \mathcal{L}(\mathcal{D},\boldsymbol{\theta}^{(t)})}{\partial b}\bigg|_{b^{(t)}} \tag{4}$$

using the derivatives derived from Equation 1 and Equation 2. η is a chosen learning rate.

Exercise

1. Use backpropagation algorithm to find $\theta^{(3)}$ with $\theta^{(0)} = (w^{(0)} = 1, b^{(0)} = 0)$. The dataset $\mathcal{D}$ is as follows:

i	x_i	y_i
1	2	7
2	3	10
3	5	16

Table 1: Training Dataset for 1.1.

That is, the true model would be $y_i = 3x_i + 1$, i.e., $w = 3, b = 1$. Implement batch gradient descent.

Neural Network

For a neural network with H hidden layers: - L_0 is the input layer (the zeroth hidden layer). L_k represents the kth hidden layer for $k \in \{1, 2, ..., H\}$. L_{H+1} is the output layer (the $H+1$th hidden layer). - $\phi^{(k)}$ represents the activation function for the kth hidden layer, with $k \in \{1, 2, ..., H\}$. $\phi^{(H+1)}$ represents the activation function for the output layer. - $\boldsymbol{w}_j^{(k)}$ represents the weights connecting the activated neurons $\boldsymbol{a}^{(k-1)}$ from the $k-1$th hidden layer to the jth neuron in the kth hidden layer, where $k \in \{1, ..., H+1\}$ and $j \in \{1, ..., q_k\}$, i.e., q_k denotes the number of neurons in the kth hidden layer. $\boldsymbol{a}^{(0)} = \boldsymbol{z}^{(0)} = \boldsymbol{x}$ by definition. - $b_j^{(k)}$ represents the bias for the jth neuron in the kth hidden layer.

Gradients For the Output Layer

The gradient for $\boldsymbol{w}_1^{(H+1)}$, i.e., the weights connecting the neurons in the Hth (last) hidden layer to the first neuron of the output layer, is given by:

$$\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial \boldsymbol{w}_1^{(H+1)}} = \frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial z_1^{(H+1)}} \frac{\partial z_1^{(H+1)}}{\partial \boldsymbol{w}_1^{(H+1)}} \tag{5}$$

where

- $\hat{y}_1 = a_1^{(H+1)} = \phi(z_1^{(H+1)})$
- $z_1^{(H+1)} = \langle \boldsymbol{a}^{(H)}, \boldsymbol{w}_1^{(H+1)} \rangle + b_1^{(H+1)}$.
- $\langle \cdot, \cdot \rangle$ represents the vector product.

Gradients For the Hidden Layers

The gradient for $\boldsymbol{w}_1^{(k)}$, i.e., the weights connecting the activated neurons $\boldsymbol{a}^{(k-1)}$ to the first neuron of the kth hidden layer $a_1^{(k)}$, is given by:

$$\begin{aligned}
\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial \boldsymbol{w}_1^{(k)}} &= \underbrace{\frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial a_1^{(k)}} \frac{\partial a_1^{(k)}}{\partial z_1^{(k)}}}_{\delta_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial \boldsymbol{w}_1^{(k)}} \\
&= \underbrace{\sum_{l \in \{1, \ldots, q_{k+1}\}} \frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial z_l^{(k+1)}} \frac{\partial z_l^{(k+1)}}{\partial a_1^{(k)}}}_{\text{Total Derivative}} \frac{\partial a_1^{(k)}}{\partial z_1^{(k)}} \frac{\partial z_1^{(k)}}{\partial \boldsymbol{w}_1^{(k)}} \\
&= \underbrace{\sum_{l \in \{1, \ldots, q_{k+1}\}} \delta_l^{(k+1)} w_{1,l}^{(k+1)} \frac{\partial a_1^{(k)}}{\partial z_1^{(k)}}}_{\delta_1^{(k)}} \boldsymbol{a}^{(k-1)}
\end{aligned}$$

Based on Equation 6, the derivative of the loss function with respect to the pre-activated value of the ith neuron in the kth hidden layer is given by

$$\delta_i^{(k)} = \frac{\partial \mathcal{L}(\mathcal{D}, \boldsymbol{\theta})}{\partial a_i^{(k)}} \frac{\partial a_i^{(k)}}{\partial z_i^{(k)}} = \sum_{l \in \{1, \ldots, q_{k+1}\}} \delta_l^{(k+1)} w_{i,l}^{(k+1)} \frac{\partial a_i^{(k)}}{\partial z_i^{(k)}}$$

Example 1

- From input layer L_0 to the first hidden layer L_1:

$$a_1^{(1)} = \phi^{(1)}(w_{1,1}^{(1)}x_1 + w_{2,1}^{(1)}x_2 + w_{3,1}^{(1)}x_3 + b_1^{(1)}) = \phi^{(1)}(\langle \boldsymbol{w}_1^{(1)}, \boldsymbol{x} \rangle + b_1^{(1)})$$
$$a_2^{(1)} = \phi^{(1)}(w_{1,2}^{(1)}x_1 + w_{2,2}^{(1)}x_2 + w_{3,2}^{(1)}x_3 + b_2^{(1)}) = \phi^{(1)}(\langle \boldsymbol{w}_2^{(1)}, \boldsymbol{x} \rangle + b_2^{(1)})$$

- From the first hidden layer L_1 to the output layer layer L_2:

$$\hat{y} = \phi^{(2)}(w_{1,1}^{(2)}a_1^{(1)} + w_{2,1}^{(2)}a_2^{(1)} + b_1^{(2)}) = \phi^{(2)}(\langle \boldsymbol{w}_1^{(2)}, \boldsymbol{a}^{(1)} \rangle + b_1^{(2)})$$

- $\phi^{(1)}(z) = S(z)$ (sigmoid function) and $\phi^{(2)}(z) = \exp(z)$ (exponential function).

Let $\boldsymbol{\theta}^{(t)} = (\boldsymbol{w}^{(t)}, \boldsymbol{b}^{(t)}) = \left(\boldsymbol{w}_1^{(t,1)}, \boldsymbol{w}_2^{(t,1)}, \boldsymbol{w}_1^{(t,2)}, b_1^{(t,1)}, b_2^{(t,1)}, b_1^{(t,2)}\right)$ be the parameter estimates of the tth iteration. For illustration, we assume the bias terms $(b_1^{(t,1)}, b_2^{(t,1)}, b_1^{(t,2)})$ are all zeros.

- For $\boldsymbol{w}_1^{(2)}$, apply equation Equation 5
- For $\boldsymbol{w}_1^{(1)}$, apply equation Equation 6
- For $\boldsymbol{w}_2^{(1)}$, apply equation Equation 6

Implementing Backpropagation in Python

See `Week_4_Lab_Notebook.ipynb` for more details. The required packages/functions are as follows:

```python
import os
os.environ["CUDA_VISIBLE_DEVICES"] = ""

import random
import numpy as np
import pandas as pd

from keras.models import Sequential
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.initializers import Constant
```

True weights:

```
w1_1 = np.array([[0.25], [0.5], [0.75]])
w1_2 = np.array([[0.75], [0.5], [0.25]])
w2_1 = np.array([[2.0], [3.0]])
```

Some synthetic data to work with:

```
# Set seed for reproducibility
np.random.seed(0)

# Generate 10000 observations of 3 numerical features
X_data = pd.DataFrame(np.random.randn(10000, 3),
                      columns=['x1', 'x2', 'x3'])
X = np.array(X_data)

# sigmoid activation function
def sigmoid(z):
  return(1/(1+np.exp(-z)))

# Hidden Layer 1
z1_1 = X @ w1_1 #the first neuron before activation
z1_2 = X @ w1_2 #the second neuron before activation
a1_1 = sigmoid(z1_1) #the first neuron after activation
a1_2 = sigmoid(z1_2) #the second neuron after activation

#Output Layer
z2_1 = np.concatenate((a1_1, a1_2), axis = 1) @ w2_1 #pre-activation of the ouput
a2_1 = np.exp(z2_1) #output

#The actual values
y = a2_1
```

From Scratch

```
# Initialised weights
w1_1_hat = np.array([[0.2], [0.6], [1.0]])
w1_2_hat = np.array([[0.4], [0.8], [1.2]])
w2_1_hat = np.array([[1.0], [2.0]])

losses = []
num_iterations = 5000
```

```python
for _ in range(num_iterations):
  ## Compute Forward Passes
  # Hidden Layer 1
  z1_1_hat = X @ w1_1_hat  # the first neuron before activation
  z1_2_hat = X @ w1_2_hat  # the second neuron before activation
  a1_1_hat = sigmoid(z1_1_hat) # the first neuron after activation
  a1_2_hat = sigmoid(z1_2_hat) # the second neuron after activation
  a1_hat = np.concatenate((a1_1_hat, a1_2_hat), axis = 1)

  # Output Layer
  z2_1_hat = a1_hat @ w2_1_hat # the output before activation
  y_hat = np.exp(z2_1_hat).reshape(len(y), 1) # the ouput

  # Track the Losses
  loss = (y_hat - y)**2
  losses.append(np.mean(loss))

  # Compute Deltas
  delta2_1 = 2 * (y_hat - y) * np.exp(z2_1_hat)
  delta1_1 = w2_1_hat[0] * delta2_1 * sigmoid(z1_1_hat) * (1-sigmoid(z1_1_hat))
  delta1_2 = w2_1_hat[1] * delta2_1 * sigmoid(z1_2_hat) * (1-sigmoid(z1_2_hat))

  # Compute Gradients
  d2_1_hat = delta2_1 * a1_hat
  d1_1_hat = delta1_1 * X
  d1_2_hat = delta1_2 * X

  # Learning Rate
  eta = 0.0005
  # Apply Batch Gradient Descent
  w2_1_hat -= eta * np.mean(d2_1_hat, axis = 0).reshape(2, 1)
  w1_1_hat -= eta * np.mean(d1_1_hat, axis = 0).reshape(3, 1)
  w1_2_hat -= eta * np.mean(d1_2_hat, axis = 0).reshape(3, 1)

print(w1_1_hat)
print(w1_2_hat)
print(w2_1_hat)
```

```
[[0.24985576]
 [0.5000211 ]
 [0.75018656]]
[[0.74987578]
```

 [0.49998626]
 [0.25009692]]
[[1.99874327]
 [3.00125615]]

From Keras

```
# An initialiser for the weights in the neural network
init1 = Constant([[0.2, 0.4], [0.6, 0.8], [1.0, 1.2]])
init2 = Constant([[1.0, 2.0]])

# Build a neural network
# `use_bias` (whether to include bias terms for the neurons or not) is True by default
# `kernel_initializer` adjusts the initialisations of the weights
x = Input(shape=X.shape[1:], name="Inputs")
a1 = Dense(2, "sigmoid", use_bias=False,
          kernel_initializer=init1)(x)
y_hat = Dense(1, "exponential", use_bias=False,
            kernel_initializer=init2)(a1)
model = Model(x, y_hat)

# Choosing the optimiser and the loss function
model.compile(optimizer="adam", loss="mse")

# Model Training
# We don't implement early stopping to make the results comparable to the previous section
hist = model.fit(X, y, epochs=5000, verbose=0, batch_size = len(y))

# Print out the weights
print(model.get_weights())
```

```
[array([[0.30257466, 0.80548096],
       [0.49333414, 0.50670713],
       [0.6842522 , 0.2076196 ]], dtype=float32), array([[2.5133712, 2.5152776],
       [2.4867482, 2.4848895]], dtype=float32)]
```