Synthèse Alternance Recherche

Jean-Baptiste Attié

June 2023

Abstract

This document is a synthesis of my research work study at the Institut Mathématique de Marseille (I2M), during my first year at Centrale Marseille. I used to learn the basics understanding of a neural network with the book "Neural Network and Numerical Analysis" [1], then I got familiar with tensorflow and focused on the case of the approximation of high gradient objective function by a neural network.

1 What is a neural network ?

A neural network is a type of machine learning model that is inspired by the structure and function of the human brain. It consists of layers of interconnected nodes (neurons) that can receive and process information, and then produce an output. Each neuron takes in inputs, processes them using an activation function, and passes the result to the next layer of neurons. Neural networks can be trained on large datasets to recognize patterns and relationships in data, and can be used for a wide range of applications, such as image recognition, speech recognition, natural language processing, and predictive modeling.

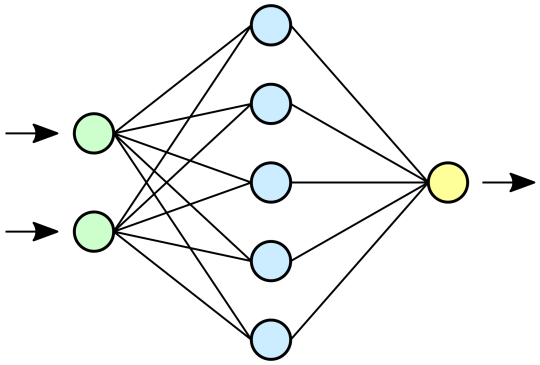


Figure 1: Representation of a Neural-Network

2 Theoric considerations

2.1 Notions

Let be an objective function, which represent the function we want the neural network to approximate,

$$f^{obj} : R^m \to R^n$$

We denote,

$$\forall i \in [|1, N|], y_i = f^{obj}(x_i) \in R^n, N \in N$$

A dataset in a neural network is a collection of input and output pairs used to train and test the network:

$$D = \{(x_i, y_i) \mid i = 1, 2, ..., N\}, D \subset R^m \times R^n$$

2.2 Representation of a NN

Let f be a function representing a neural network:

$$f : R^m \to R^n, \quad x \mapsto Wx + b,$$

where $W \in \mathcal{M}_{n\times m}(R)$ is the matrix of weights and $b \in R^n$ is called the bias.

The representative function of a neural network can be expressed as follows:

$$f = f_p \circ g_{p-1} \circ f_{p-1} \circ g_{p-2} \circ \ldots \circ g_0,$$

where each layer's function, f_i, maps from R^{a_i} to $R^{a_{i+1}}$. The dimensions of the layers are given by $(a_0, a_1, \ldots, a_{p+1}) \in N^{p+2}$.

The bias is $b_i \in R^{a_{i+1}}$, and the overall bias is calculated as $b = b_p + \sum_{i=0}^{p-1}(W_p \cdot \ldots \cdot W_{i+1} \cdot b_i)$.

The weight matrices of the i-layer are denoted as $W_i \in \mathcal{M}_{a_{i+1}\times a_i}(R)$, where the combined weight matrix is represented as $W = W_p \cdot \ldots \cdot W_0$.

The activation functions g_i are typically chosen as $g_i = S$, where S is a sigmoid function with the properties: $S \in C^1(R)$, $S'(x) > 0$, $S(x) \longrightarrow 0$ as $x \longrightarrow -\infty$, and $S(x) \longrightarrow 1$ as $x \longrightarrow +\infty$.

2.3 Loss

We define the cost function that measures the difference between the predicted output of the model and the actual output:

$$J(W, b) = \sum_{(x,y)\in D} L(f_{obj}(x), f(x, W, b)) = \sum_{i=1}^{N} L(y_i - (Wx_i + b))$$

where $L(y, \hat{y})$ measures the error between the expected output and the predicted output for a given input. In the most common case, $L(y, \hat{y})$ is the quadratic loss function $L(y, \hat{y}) = |y - \hat{y}|^2$.

The goal of training a neural network is to minimize the cost function $J(W, b)$ by adjusting the weights and biases of the network.

3 Practical considerations

3.1 Learning a 1 Variable Real Function

This code builds and trains a one-layer neural network that aims to approximate a given objective function:

$$f_{obj}(x) = 50x \cdot e^{-7x^2}$$

.

The network will "learn" this function. A dataset is randomly generated to train the model and also for testing purposes. The program also provides error measurements to evaluate the accuracy of the model.

```
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfd
import matplotlib.pyplot as plt
import random
```

```
import math

% Define the objective function
def fobj(x):
    return 50 * x * math.exp(-7 * x ** 2)

N = 300  # Training dataset size
I = np.linspace(-10, 10, N)  # Training dataset

Ntest = 500
Itest = sorted([random.uniform(-10, 10) for k in range(Ntest)])  # Test dataset

% Convert data to numpy arrays
train_examples, train_labels = np.array([[x] for x in I]), np.array([[fobj(x)] for x in I])
test_examples, test_labels = np.array([[x] for x in Itest]), np.array([[fobj(x)] for x in Itest])

% Define the model
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(50, activation=tf.keras.activations.sigmoid, input_shape=(1,)))
model.add(tf.keras.layers.Dense(1))

% Display the model's architecture
model.summary()

% Compile the model with Adam optimizer and mean squared error loss function
model.compile(optimizer='adam', loss='mean_squared_error')

total_epochs = 100
loss_values = []

% Training the model
for epoch in range(total_epochs):
    print("Epoch", epoch)
    history = model.fit(train_examples, train_labels, epochs=1, batch_size=1, verbose=1)
    loss_values.append(model.evaluate(test_examples, test_labels, verbose=0))

% Evaluate model performance on the test set
loss = model.evaluate(test_examples, test_labels, verbose=0)

% Make predictions and calculate errors
predictions = model.predict(test_examples)
n = len(test_labels)
AbsoluteError = [abs(predictions[i][0] - test_labels[i][0]) for i in range(n)]
MeanAbsoluteError = sum(AbsoluteError) / n

print("Loss on test set: ", loss)
print("Mean Absolute Error: ", MeanAbsoluteError)
```

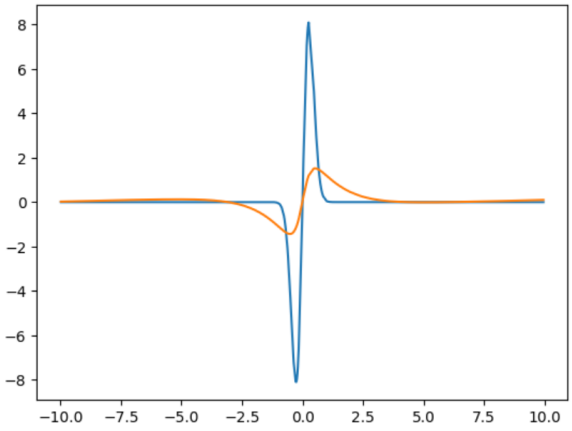


Figure 2: Approximation of f_{obj} (blue) by the Neural Network (orange)

3.2 Case of the function with "high-gradient"

When approximating a function with rapid variations, a neural network may struggle to provide accurate approximations in areas of the function with high gradient changes. The function is well-approximated in regions with relatively slow variations, but not in abrupt change points. One solution could be to increase the size of the dataset, but this comes with additional costs. A more effective approach is to strategically redistribute the data points around the high-gradient regions. This allows the neural network to be specifically trained to perform better in the areas where it is most needed.

This program distributes points within an interval such that the resulting distribution puts more points in regions with strong gradients in the function f.

```
def repartGrad(f, a, b, N):
    # Function to distribute points in the interval to put more points in strong gradients zones

    X = np.linspace(a, b, N)  # Create an array of N evenly spaced points between a and b

    newx = a
    newsub = np.array([newx])  # Initialize newx and newsub arrays to store the new points

    h = 1 / N  # Calculate the step size

    # Calculate the second derivative using the central difference approximation
    ddf = [(f(X[k+1]) - 2 * f(X[k]) + f(X[k-1])) / h**2 for k in range(1, N-1)]

    # Calculate the adjustment factor L for each point based on the second derivative
    L = np.array([1 / (1 + math.log(1 + abs(ddf[k]))) for k in range(0, N-2)])

    Norm = (b - a) / sum(L)  # Calculate the normalization factor

    H = Norm * L  # Adjust the step size based on the normalization factor

    # Iterate over the points in H and increment newx by the corresponding value

    for k in range(1, N-2):
        newx = newx + H[k]
        newsub = np.append(newsub, newx)
```

```
    newsub = np.append(newsub, b)

    return newsub  # Return the resulting array of distributed points
```

By applying this algorithm to the previous objective function, we get a new dataset. The training of the neural network leads to a more desirable approximation:

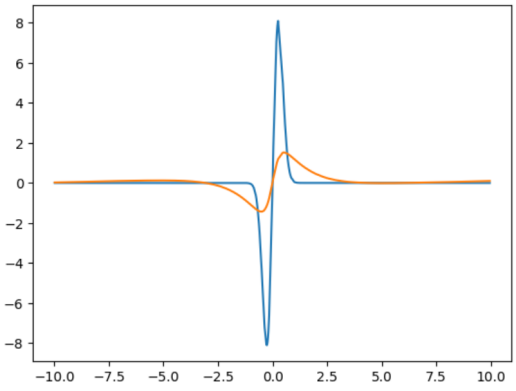


Figure 3: Previous approximation

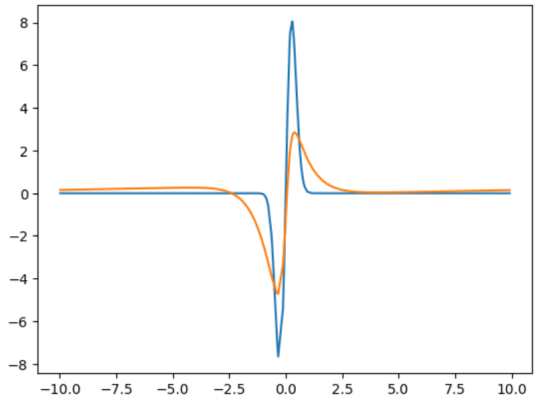


Figure 4: New approximation

3.3 Learning 5 variables real function

This code trains a neural network model to multiply 5 real number (in [-1,1]) using a dataset of 5-dimensional inputs. The dataset is generated by creating a grid of values and generating all possible combinations of 5 variables. The model architecture consists of two hidden layers with 10 neurons each, using the sigmoid activation function. The model is trained using the Adam optimizer and mean squared error loss. The training process is repeated for a fixed number of epochs, and the loss values are stored. Finally, the model is evaluated on a separate test set, and the loss and mean absolute error are printed.

```
import numpy as np
import tensorflow as tf
import random

def fobj(x):
    return np.prod(x)

N = 5  # Size of one dimension of the training dataset
I = np.linspace(-1, 1, N)  # Create a grid of values
# Generate all possible combinations of 5 variables in an array
I5 = np.array(np.meshgrid(*[I]*5)).T.reshape(-1, 5)

N_test = 100  # Size of the test dataset
I_test = np.random.uniform(-1, 1, (N_test, 5))  # Random matrix of 5-tuples

train_examples = I5  # Training examples
train_labels = np.array([fobj(x) for x in I5]).reshape(-1, 1)  # Training labels as column vector

test_examples = I_test  # Test examples
test_labels = np.array([fobj(x) for x in I_test]).reshape(-1, 1)  # Test labels as column vector
```

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(10, activation=tf.keras.activations.sigmoid, input_shape=(5,)))
model.add(tf.keras.layers.Dense(10, activation=tf.keras.activations.sigmoid))
model.add(tf.keras.layers.Dense(1))

# Compile the model with Adam optimizer and mean squared error loss function
model.compile(optimizer='adam', loss='mean_squared_error')
model.summary()

total_epochs = 10
loss_values = []  # List to store the loss values

for epoch in range(total_epochs):
    print("Epoch", epoch)
# Train the model
    history = model.fit(train_examples, train_labels, epochs=1, batch_size=1, verbose=1)
# Evaluate loss on test examples
    loss_values.append(model.evaluate(test_examples, test_labels, verbose=0))

# Final evaluation of loss on test examples
loss = model.evaluate(test_examples, test_labels, verbose=0)
predictions = model.predict(test_examples)  # Predictions on test examples
# Calculate the mean absolute error
mean_absolute_error = np.mean(np.abs(predictions - test_labels))

print("Loss on the test set:", loss)
print("Mean Absolute Error:", mean_absolute_error)
```

The next step is to generalize the algorithm of strong gradients to the case of functions of many dimension.

[1] Bruno Després, "Neural Networks and Numerical Analysis," De Gruyter Series in Applied and Numerical Mathematics.

[2] Wikipedia. "Neural Network."