Neural network

A neural network is a type of machine learning model inspired by the structure and function of the human brain. It is composed of layers of interconnected "neurons," which process and transmit information. Neural networks are used for a variety of tasks, such as image classification, natural language processing, and prediction.

The basic building block of a neural network is the neuron, which takes input data, performs a mathematical operation on it, and generates an output. The output of one neuron is passed as input to one or more other neurons, forming a network of interconnected neurons. This network can be thought of as a computational graph where edges are the input and output connection of neurons and nodes are the neurons.

Structure of Artificial neurons and their functions

The structure and behavior of a neural network can be adjusted to suit the specific problem being solved. For example, the number of layers, the number of neurons per layer, and the specific activation functions used can be varied. The process of training a neural network involves adjusting the parameters of the network (i.e., the weights and biases of the neurons) so that the output of the network is as close as possible to the desired output for a given set of inputs.

The structure of an artificial neuron, also called a node or perceptron, is designed to mimic the behavior of a biological neuron in the human brain. The basic building block of an artificial neural network is an artificial neuron, which consists of the following components:

- Inputs: The inputs to the neuron are the data that the neuron processes. Each input is assigned a weight, which represents the strength of the connection between that input and the neuron.
- Summation Function: The inputs and their corresponding weights are combined using a summation function, which produces a single scalar value that represents the total input to the neuron.
- Activation Function: The output of the summation function is passed through an activation function, which introduces non-linearity into the output of the neuron and allows the network to learn and represent more complex patterns and relationships in the input data.
- Output: The output of the neuron is the output of the activation function, which represents the final decision or prediction made by the neuron.

The function of an artificial neuron is to take in a set of inputs, perform mathematical operations on those inputs, and generate an output. The inputs are multiplied by their corresponding weights, and these weighted inputs are then summed together. This weighted sum is then passed through an activation function, which produces the final output of the neuron. The output is then passed on to the next layer of neurons, and so on, until the final decision is made.

In summary,

- Inputs to a neuron are data and each data point is assigned a weight which tells about the effect of that input on the output.
- A summation function is used to combine the inputs and the weights to give the weighted sum.
- Activation function is applied on the weighted sum to introduce non-linearity and to obtain the output.
- The output is the final decision or prediction made by that neuron.

Overall, the primary function of artificial neurons is to take input data, perform mathematical operations on it, and produce an output, which can be passed along to other neurons or used as a final decision.

Activation function

In an artificial neural network (ANN), an activation function is a mathematical function that is applied to the output of a neuron to determine its state. The purpose of the activation function is to introduce non-linearity into the output of a neuron, allowing the network to learn and represent more complex patterns and relationships in the input data.

Several types of activation functions can be used in an ANN, including:

- Sigmoid/logidtic: The sigmoid function, which is also known as the logistic function, produces an output in the range of (0, 1), which can be interpreted as a probability.
- ReLU: Rectified Linear Unit (ReLU) activation function returns the input if is positive and returns zero if the input is negative.
- Tanh: The hyperbolic tangent function, which produces an output in the range of (-1, 1)
- Softmax: A special kind of activation function used in the output layer of multi-class classification problems.
- Leaky ReLU, PReLU, ELU etc. also exist which improve over the basic ReLU activation

---

The logistic/ Sigmoid function scales the values between 0 and 1. It is used in the output layer for Binary classification. It may cause a vanishing gradient problem during backpropagation and slows the training time.

$$f(x) = \frac{1}{1+e^{-x}}$$

ReLU also prevents the vanishing gradient problem but introduces an exploding gradient problem during backpropagation. The exploding gradient problem can be prevented by capping gradients.

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Feedforward neural network (FFNN)

The choice of activation function depends on the specific problem you are trying to solve and the characteristics of the data you are working with.

A feedforward neural network is a type of artificial neural network in which the information flows in only one direction, from the input layer to the output layer, without forming any loops. The information flows through the network layer by layer, passing through each neuron in a given layer before moving on to the next layer.

The basic structure of a feedforward neural network consists of an input layer, one or more hidden layers, and an output layer. The input layer receives the input data, and each neuron in this layer represents a feature of the input. The hidden layers process the input data and extract features that are relevant for the task at hand. The output layer produces the final decision or prediction of the network.

The forward propagation, also known as feedforward, is done layer-by-layer. The inputs are passed to the first layer, which processes the data and produces an output. This output is then passed to the next layer, which processes it further and produces another output, and so on. Finally, the output of the last layer is the final decision or prediction of the network.

Feedforward neural networks can be used to solve a wide range of problems, such as image classification, speech recognition, and natural language processing. These networks are typically trained using backpropagation, which is an algorithm that adjusts the weights and biases of the neurons to minimize the error between the predicted output and the desired output.

In summary, feedforward neural networks are a type of artificial neural network where the information flows only in one direction, from input to output, in a sequential manner. It consists of Input, hidden and output layers where hidden layers extract features. The forward propagation is done layer-by-layer and the network is typically trained using backpropagation algorithm.

Forward Propagation: This is the phase where the input data is passed through the network, layer by layer, and the output is produced. During this phase, the input data is passed through each layer of the network, and the neurons in each layer process the data and produce an output. The output of one layer is then passed as input to the next layer, and this process is repeated until the final output of the network is produced. The forward propagation phase is also known as the feedforward phase. the process of forward propagation in an artificial neural network (ANN) is similar to that of linear and logistic regression.

In forward propagation, each neuron in the network receives input from the previous layer and performs a calculation to produce an output. The calculation performed by a neuron can be broken down into three parts:

- Weighted sum: Each input value is multiplied by its corresponding weight and the results are summed together to produce a single scalar value. This step is similar to the linear regression where the dot product of input and weight is taken to generate the weighted sum.
- Bias: A bias value is added to the weighted sum to produce a modified weighted sum. Bias is a scalar value that is used to adjust the output of the neuron, similar to the y-intercept in linear regression.
- Activation function: The output of the neuron is then passed through an activation function, which introduces non-linearity into the output. This step allows the network to learn and represent more complex patterns and relationships in the input data.

The activation function used in the final step of forward propagation is the key difference between linear and logistic regression. While linear regression uses an identity activation function that simply returns the weighted sum, logistic regression uses a sigmoid function which returns the probability of a binary outcome. The sigmoid function squashes the values between 0 and 1 and hence the output is treated as probability.

In summary, forward propagation in an ANN is similar to the calculations performed in linear and logistic regression, but with the addition of an activation function that introduces non-linearity into the output of the neurons. In this phase, input data is passed through each layer, and the neurons in each layer process the data and produce an output. The output of one layer is then passed as input to the next layer, and this process is repeated until the final output of the network is produced.

Multiplying feature values with weights and adding bias to each neuron is basically applying Linear Regression If we apply the Sigmoid function to it, then each neuron is basically performing a Logistic Regression.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

β_0 = *Bias*

β_i = *Weights/ coefficients*

x_i = *Features*

- If we apply **Sigmoid activation function** to $\hat{y}$ then the resultant function would look like,

$$\sigma(\hat{y}) = \frac{1}{1+e^{-\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n}} = \frac{1}{1+e^{-\hat{y}}}$$

Backpropagation in FFNN

Backpropagation: After the forward propagation, the output of the network is compared with the desired output, and the error is calculated. The backpropagation phase is used to adjust the weights and biases of the neurons so that the error is minimized. This is done by propagating the error back through the network, layer by layer, and adjusting the weights and biases of the neurons in each layer. The process is iterative and done multiple times till the network reaches a satisfactory level of accuracy. The backpropagation phase is also known as the learning phase.

The main goal of the backpropagation algorithm is to minimize the error between the predicted output and the desired output by adjusting the weights of the neurons. This is typically done using an optimization algorithm, such as stochastic gradient descent, which iteratively updates the weights of the neurons in the direction that minimizes the error. This process of forward propagation and backpropagation is repeated multiple times until the network reaches a satisfactory level of accuracy on the training data.

In summary, ANN training involves passing the input data forward through the network to produce an output and then comparing the output with the desired output to calculate the error. This error is then backpropagated through the network, and the weights and biases of the neurons are adjusted to minimize the error. This process is repeated until the network reaches a satisfactory level of accuracy on the training data.

An optimization function is applied to perform backpropagation

Gradient descent, Adam optimizer, gradient descent with momentum, and RMS Prop (Root Mean Square Prop) are all optimization algorithms used to adjust the weights of a neural network during the training process.

Gradient Descent: Gradient descent is a basic optimization algorithm that adjusts the weights of a neural network by iteratively updating them in the direction of the negative gradient of the error function. The gradient is calculated with respect to the weights, which tells the steepness and direction of the error function at a given point, so the weights are updated in the opposite direction to reduce the error. This process is repeated until the error reaches a minimum.

Adam Optimizer: Adam is a more advanced optimization algorithm that combines the ideas of gradient descent and momentum. It uses an adaptive learning rate that adjusts the learning rate of each weight individually, which allows it to converge more quickly and accurately than traditional gradient descent. Adam also includes a moving average of the second moments of the gradients, similar to RMS Prop, which can help to avoid oscillations and converge faster.

Gradient Descent with Momentum: Gradient descent with momentum is an optimization algorithm that combines the ideas of gradient descent and momentum. It uses a moving average of the gradients to add a momentum term to the update step of the weights, which can help the algorithm converge more quickly and avoid getting stuck in local minima.

RMS Prop (Root Mean Square Prop): RMS Prop also uses an adaptive learning rate, similar to Adam. It also uses a moving average of the second moments of the gradients, which can help to avoid oscillations and converge faster. The RMS Prop algorithm divides the learning rate for weight by the square root of the moving average of the second moments of the gradients for that weight, which can help to scale the learning rate for each weight individually and improve the stability of the training process.

All of these optimization algorithms are designed to help a neural network converge to a minimum of the error function more quickly and accurately, they all have their own strengths and weaknesses, and the choice of which algorithm to use will depend on the specific problem and dataset.

In summary, Gradient Descent, Adam Optimizer, Gradient Descent with Momentum, and RMS Prop are all optimization algorithms used to adjust the weights of a neural network during the training process. They all use the gradient of the error function to update the weights in order to reduce the error. Adam and RMS Prop also use an adaptive learning rate and a moving average of the second moments of the gradients to improve the stability and convergence of the training process.

Why Neural Networks?

Neural networks, and specifically artificial neural networks (ANNs) are powerful machine learning models that are used for a wide range of tasks, including image classification, speech recognition, natural language processing, and prediction. There are several reasons why neural networks are a popular choice for many applications:

- Flexibility: Neural networks are highly flexible and can be used to solve a wide range of problems. They are not limited to specific types of data or specific types of tasks.
- Non-linearity: Neural networks introduce non-linearity into the output of a neuron, allowing the network to learn and represent more complex patterns and relationships in the input data. This makes them well suited to problems that are not easily solvable using traditional linear models.
- Handling high-dimensional data: Neural networks can handle high-dimensional data, such as images or text, and can extract features from the data that are relevant for the task at hand.
- Handling missing data: Neural networks can handle missing data by imputing the missing values during training and testing.
- Handling noise and outliers: The presence of noise and outliers do not affect the performance of Neural networks much.
- Handling large datasets: Neural networks can be trained on large datasets, which allows them to learn and generalize well on new data.
- Scalability: Neural networks can be easily scaled to larger architectures to handle more complex problems or to process larger amounts of data.
- Good at handling big data: Neural networks are good at handling big data as they are robust to missing data and noisy data.

Overall, the flexibility, non-linearity, and ability to handle high-dimensional data make neural networks an attractive option for many applications, and their ability to handle noise, missing and outlier data, scalability and performance on big data makes them more useful.

The chain rule

The chain rule is a fundamental concept in calculus that allows us to find the derivative of a composite function. A composite function is a function that is made up of two or more other functions, and the chain rule allows us to find the derivative of the composite function in terms of the derivatives of the individual functions that make it up.

The chain rule states that:

if y = f(u) and u = g(x),

$$\frac{\partial L}{\partial w} = \frac{\partial z}{\partial w} \times \frac{\partial a}{\partial z} \times \frac{\partial L}{\partial a}$$

then the derivative of y with respect to x is dy/dx = dy/du * du/dx

The left hand side of this equation, dy/dx, is the derivative of the composite function y with respect to x, and it is often referred to as the "total derivative." The right hand side of this equation, dy/du * du/dx, is the product of the derivative of y with respect to u (dy/du), and the derivative of u with respect to x (du/dx).

Here's an example of how the chain rule can be applied:

Consider the function y = f(x) = (x^2 + 1)^3

To find the derivative of y with respect to x, we could use the chain rule by first noting that u = x^2 + 1, so that:

y = f(u) = u^3

Next, we find the derivative of y with respect to u, which is:

dy/du = 3u^2

We also find the derivative of u with respect to x, which is:

du/dx = 2x

Now we can substitute these into the chain rule:

dy/dx = dy/du * du/dx

= 3u^2 * 2x

= 6x(x^2+

Description of the values in the Chain rule:

- L - Loss function
  - **Example: Cross entropy loss** L = -y log($\hat{y}$) + (1 - y) log(1 - $\hat{y}$)
- w - weights
- z - Linear regression
  - **Example: z =** $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$
- a - activation function used.
  - **Example: a =** $\sigma(z) = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n)$

Terminologies of ANN

Artificial neural networks (ANNs) have several specific terminologies that are used to describe various aspects of the network and its training process. Here are a few of the key terms:

- **Epoch:** An epoch refers to one complete iteration through the entire training dataset during the training process of an ANN. One epoch consists of training the network on all of the training samples once. The number of epochs is a hyperparameter that can be set before training the network, and it determines how many times the network will see the entire training dataset during the training process.
- **Hyperparameters:** Hyperparameters are parameters that are set before training an ANN. They control the overall behavior and architecture of the network and include things like the number of hidden layers, the number of neurons in each layer, the learning rate, the type of activation function, etc. They are different from model parameters, which are learned during the training process.
- **Metrics:** Metrics are used to evaluate the performance of an ANN on a given dataset. Commonly used metrics include accuracy, precision, recall, F1-score, ROC-AUC. The choice of metric will depend on the problem being solved and the dataset being used.
- **Loss function:** Loss function is used to evaluate the error of the network in prediction. It compares the predicted values with the true values. Common loss functions include mean squared error (MSE), cross-entropy loss, binary cross-entropy loss etc. It's an indication of how well the network is doing and helps guide the training process.
- **Overfitting:** Overfitting is a phenomenon that occurs when a model learns the training data too well and as a result, performs poorly on unseen data. It happens when model is too complex, in terms of number of parameters, it will memorize the training data and not generalize well to new unseen data.
- **Underfitting:** Underfitting refers to a model that cannot capture the underlying patterns in the data well enough. This happens when the model is too simple or has too few parameters, and is not able to represent the complexity of the problem at hand. As a result, the model performs poorly on both the training and the test data.
- **Batch:** The dataset is divided into small subsets called batches, where the model's parameters are updated for each batch, during the training process. This technique is called batch gradient descent. Using a batch size of the whole dataset is called Stochastic Gradient Descent (SGD).
- **Dropout:** Dropout is a regularization technique where random neurons are dropped out of the network during training. This helps to prevent overfitting by reducing the complexity of the model and forcing the remaining neurons to learn more robust features.
- **Early Stopping:** It's a technique where training is stopped when the performance of model on the validation dataset starts to degrade after a certain point. This helps to prevent overfitting and select the best model.

Overall, these terminologies play an important role in the designing, training, evaluation and fine-tuning the ANN. Understanding these terms and how they work is important for getting the most out of an artificial neural network, as well as for interpreting the results and communicating with other researchers in the field.

Sample Question:

Understanding of artificial neural networks (ANNs) and their ability to implement them in Python:

You have been given a dataset containing customer information, such as age, income, and credit score. The goal of this task is to train an artificial neural network (ANN) to predict whether a customer is likely to default on their loan or not.

The dataset is provided in the form of a CSV file, and it is divided into two parts: a training set and a test set. You are required to use the training set to train an ANN, and then evaluate the performance of the network on the test set.

You are required to use Python for this task and you should use Keras or Tensorflow library for building the ANN.

You should also report the following:

A brief summary of the data, including the number of samples, number of features, and number of classes.

- A description of the architecture of the network, including the number of layers, the number of neurons in each layer, and the type of activation functions used.
- A plot of the training and validation loss and accuracy over the epochs.
- The final accuracy and confusion matrix of the test set.
- You should also include all of your code and comments, as well as any additional analysis or discussion that you think is necessary to explain your approach and the results fully.

Please make sure to handle the categorical variables and missing values.

Answer:

First, we'll start by importing the necessary libraries:

```
import pandas as pd

import numpy as np

from sklearn.preprocessing import LabelEncoder

from sklearn.model_selection import train_test_split

from keras.models import Sequential

from keras.layers import Dense

from keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt
```

Next, we'll load the dataset from the CSV file and do some preprocessing:

```
# Load the dataset

data = pd.read_csv('customer_data.csv')
```

```
# Handle missing values

data = data.fillna(data.mean())

# Encode the categorical variables

encoder = LabelEncoder()

data['gender'] = encoder.fit_transform(data['gender'])

data['default'] = encoder.fit_transform(data['default'])

# Split the data into training and test sets

train_data, test_data, train_labels, test_labels = train_test_split(data.drop(['default'], axis=1),
data['default'], test_size=0.2)
```

Here we are using pd.read_csv() to load the data from csv file. data.fillna(data.mean()) is used to fill the missing values with the mean of the column. we are also encoding the categorical variables(gender, default) to numeric value. Then we are splitting the data into two parts, training set and test set.

Then, we'll define the architecture of our ANN:

```
# Define the model

model = Sequential()

model.add(Dense(64, input_dim=train_data.shape[1], activation='relu'))

model.add(Dense(32, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

# Compile the model

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Here, we are using the sequential model of keras. In this, we are using 2 hidden layers, first one with 64 neurons and the second one with 32 neurons. the input_dim parameter defines the number of input features. The activation functions used are 'relu' for the hidden layers and 'sigmoid' for the output layer, because our problem is binary classification. we are using binary_crossentropy loss function for binary classification.

Next, we'll train the model and monitor its performance during training:

```
# Train the model

history = model.fit(train_data, train_labels, epochs=100, batch_size=32, validation_data=(test_data,
test_labels), callbacks=[EarlyStopping(monitor='val_loss', patience=3)])
```

```
# Plot the training and validation loss

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Test
```