Back propogation in neural networks

Lecture Notes

Generated · February 11, 2026

LECTURE OUTLINE

Back Propagation in Neural Networks

What This Lecture Covers

In this lecture, we will explore the concept of back propagation, a fundamental algorithm used for training and optimization in neural networks. We will delve into the history, mechanics, and applications of back propagation, as well as its practical implementation using Python. By the end of this lecture, you should have a thorough understanding of how to apply back propagation in neural networks.

I. Introduction

Back propagation is a crucial algorithm used for training artificial neural networks. It was first introduced by Rumelhart et al. (1986) and has since become a cornerstone of deep learning. In this section, we will discuss the definition, history, and importance of back propagation.

Back propagation is an optimization technique that allows us to train neural networks by iteratively adjusting the model's parameters to minimize the error between predicted outputs and actual outputs. The algorithm consists of two main passes: forward pass and backward pass.

The forward pass involves calculating the output of each layer in the network, given the input data. This is done by applying the activation functions to the weighted sum of inputs. In contrast, the backward pass involves computing the error between predicted outputs and actual outputs, as well as calculating the gradients of the cost function with respect to the model parameters.

The evolution of back propagation has been influenced by advances in computing power, data storage, and artificial intelligence. The development of deep learning algorithms like convolutional neural networks (CNNs) and recurrent neural networks (RNNs) relies heavily on efficient back propagation implementations.

II. Back Propagation Algorithm

In this section, we will break down the back propagation algorithm into its two main components: forward pass and backward pass.

Forward Pass

During the forward pass, we calculate the output of each layer in the network, given the input data. This is done by applying the activation functions to the weighted sum of inputs.

Let's consider a simple neural network with one input layer, one hidden layer, and one output layer. The input data is represented as x, the weights are represented as w, and the bias terms are represented as b. We apply the sigmoid activation function to the weighted sum of inputs, resulting in the following equation:

a = σ(w^T x + b)

where σ is the sigmoid function.

The output of each layer is calculated by applying the sigmoid function to the weighted sum of inputs. The result is passed through another layer, until we reach the final output.

Backward Pass

During the backward pass, we calculate the error between predicted outputs and actual outputs. We also compute the gradients of the cost function with respect to the model parameters.

The calculation involves the following steps:

1. Calculate the error between predicted outputs and actual outputs.
2. Compute the gradients of the cost function with respect to each layer's output.
3. Apply the chain rule to compute the gradients of the cost function with respect to each weight and bias term.
4. Update the model parameters using stochastic gradient descent (SGD) or batch gradient descent (BGD).

III. Training a Neural Network Using Back Propagation

In this section, we will discuss how to train a neural network using back propagation.

There are two main optimization algorithms used in back propagation: stochastic gradient descent (SGD) and batch gradient descent (BGD). SGD is an online optimization algorithm that updates the model parameters based on the gradients computed from a single example. BGD, on the other hand, updates the model parameters based on the gradients computed from the entire training dataset.

We will also discuss the Adam and RMSProp optimization algorithms, which are variants of SGD that adapt the learning rate for each parameter individually.

Regularization techniques, such as L1 and L2 regularization, can be used to prevent overfitting by adding a penalty term to the cost function. We will explore how to implement regularization in back propagation.

### Example: Training a Neural Network using Keras or PyTorch

We will use Python's Keras library to train a simple neural network. The code will involve defining the model architecture, compiling the model, and training it on a sample dataset. python from keras.models import Sequential from keras.layers import Dense import numpy as np

# Define the model architecture model = Sequential() model.add(Dense(64, activation='relu', input_shape=(784,))) model.add(Dense(10, activation='softmax'))

# Compile the model model.compile(loss='categorical_crossentropy', optimizer='adam')

# Train the model on a sample dataset Xtrain, ytrain = ... # Load the dataset Xtrain = Xtrain.reshape((Xtrain.shape[0], 784)) ytrain = np.argmax(ytrain, axis=1) model.fit(Xtrain, ytrain, epochs=10, batchsize=128)

IV. Applications of Back Propagation

Back propagation has numerous applications in various fields, including:

- Image classification and object detection
- Natural Language Processing and sentiment analysis
- Time series prediction and regression

We will explore some of these applications in detail.

### Example: Image Classification using CNNs

Convolutional neural networks (CNNs) are a type of neural network that can be trained using back propagation. We will use the MNIST dataset to train a CNN for image classification. python from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Define the model architecture model = Sequential() model.add(Conv2D(32, kernelsize=(3, 3), activation='relu', inputshape=(28, 28, 1))) model.add(MaxPooling2D(pool_size=(2, 2))) model.add(Flatten()) model.add(Dense(128, activation='relu')) model.add(Dense(10, activation='softmax'))

# Compile the model model.compile(loss='categorical_crossentropy', optimizer='adam')

# Train the model on the MNIST dataset Xtrain, ytrain = ... # Load the dataset Xtrain = Xtrain.reshape((Xtrain.shape[0], 28, 28, 1)) ytrain = np.argmax(ytrain, axis=1) model.fit(Xtrain, ytrain, epochs=10, batchsize=128)

V. Practical Implementation of Back Propagation

In this section, we will explore the practical implementation of back propagation using Python.

We will use PyTorch to train a simple neural network. The code will involve defining the model architecture, compiling the model, and training it on a sample dataset. python import torch import torch.nn as nn

# Define the model architecture class Net(nn.Module): def init(self): super(Net, self).init() self.fc1 = nn.Linear(784, 128) self.fc2 = nn.Linear(128, 10)

def forward(self, x): x = torch.relu(self.fc1(x)) x = self.fc2(x) return x

# Compile the model model = Net() criterion = nn.CrossEntropyLoss() optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Train the model on a sample dataset Xtrain, ytrain = ... # Load the dataset for epoch in range(10): optimizer.zerograd() outputs = model(Xtrain) loss = criterion(outputs, y_train) loss.backward() optimizer.step()

VI. Conclusion/Q&A

In this lecture, we have explored the concept of back propagation, a fundamental algorithm used for training and optimization in neural networks. We have discussed its history, mechanics, and applications, as well as its practical implementation using Python.

We will leave time for student questions and answers, and encourage further exploration and research on back propagation.

★ KEY TAKEAWAYS

1. Back propagation is an optimization technique that allows us to train neural networks by iteratively adjusting the model's parameters to minimize the error between predicted outputs and actual outputs.

2. The algorithm consists of two main passes: forward pass and backward pass.

3. The forward pass involves calculating the output of each layer in the network, given the input data. The backward pass involves computing the error between predicted outputs and actual outputs, as well as calculating the gradients of the cost function with respect to the model parameters.

4. Stochastic gradient descent (SGD) and batch gradient descent (BGD) are two main optimization algorithms used in back propagation.

5. Regularization techniques, such as L1 and L2 regularization, can be used to prevent overfitting by adding a penalty term to the cost function.

6. By applying these concepts, you should be able to implement back propagation in neural networks using Python.