Hierarchal Softmax: Efficiently Handling A Plethora of Classes

Author: Nicholas Kissoon - 100742790

December 13, 2024

Introduction

Throughout the course we have seen Softmax used in various ways, but had yet to see how Hierarchal Softmax's work that prompted various questions, of which the most prominent being *What is a Hierarchal Softmax and when would you use it?* That question is what this report below sets out to solve as we delve not into what it is, but it's relevancy, it's significance in efficiency, how to build one and how they behave with working examples. To provide some background, before we delve into what exactly a Hierarchal Softmax is, we have to understand what a Softmax is. A Softmax function takes a vector as an input and normalizes them then outputs a probability distribution over the predicted classes. It is quite useful in a variety of situations however there is one where it becomes less favoured; This is where Hierarchal Softmax's come in.

Hierarchal Softmax and word2vec

Given that the Softmax function has a complexity of O(n) of an output of n classes, in turn making it quite computationally expensive to classify cases in which there are many output classes. However, it in such cases like these it should be encouraged to switch over to that of the Hierarchal Softmax to approximate the Softmax function as it possesses a complexity of O(log(n)) which is a substantial difference.

This is particularly interesting in it's application of word2vec as the size of the output vector, the vocabulary of the input in this case, can grow to be quite large. Reaching up to even hundreds of thousands of different words in a given text, it is clear why Softmax will fall short of this task, this is why it must be estimated through Hierarchal Softmax.

Hierarchal Softmax

What Hierarchal Softmax does differently is output this information into the form of a tree, more specifically a multi-layer binary tree, in which it makes training faster/easier for word2vec as only relevant nodes are traversed in order to classify to word. Below will be a worked out and explained example where it can be seen in action. Your end result after a Hierarchal Softmax is a probability distribution across all the vocabulary arranged in binary tree format. Due to it's tree like nature it reduces the original time complexity of Softmax from O(n) to now being a fraction of that at O(log(n)) with n being the number of output classes. This then allows for the computation of large, vast datasets with relatively high accuracy at the cost of a fraction of the time.

Tree Construction and Parameter Based Modeling

A deeper dive on how these trees are constructed. The entire vector, is known as the *vocabulary* consisting up of many *words*. In terms of the tree and how it displays the vocabulary, is that the internal nodes that make up the tree contain a function, sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ for an example, with the edges representing likelihoods (probability) of what the word is until it reaches a leaf node. At each node probabilities are multiplied out depending on whether the left or right child was chosen until you reach the leaf node, in this instance represents the words themselves, in other words the output class. It is also important to note here that the index of the word probability is the same as the index of the word in the output hierarchal softmax. The output of the Hierarchal Softmax has embeddings at every node as opposed for just every word. Hierarchal Softmax learns two parameters at every step, the word embeddings for the input and the node vectors for the nodes that are internal. This can be seen in the Numerical Demonstration of the Update Rule in which both the word embeddings and node vectors are used in the sigmoid function.

Softmax vs Hierarchal Softmax Comparison

Code

```
import numpy as np
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from scipy.special import softmax
from hierarchicalsoftmax import SoftmaxNode, HierarchicalSoftmaxLoss
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import time

# Load the 20 newsgroups dataset
newsgroups = fetch_20newsgroups(subset='all', remove=('headers', 'footers',
'quotes'))
```

```
X, y = newsgroups.data, newsgroups.target

# Preprocess the text data using a bag-of-words model with max 10,000 features
vectorizer = CountVectorizer(max_features=10000)
X = vectorizer.fit_transform(X).toarray()

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define a function to compute softmax probabilities
def regular_softmax(X, W, b):
    scores = np.dot(X, W.T) + b
    return softmax(scores, axis=1)

# Initialize weights and biases for softmax classifier
num_classes = len(np.unique(y))
W = np.random.randn(num_classes, X.shape[1])
b = np.zeros(num_classes)

# Train softmax
learning_rate = 0.01
num_epochs = 10

softmax_start_time = time.time()

for epoch in range(num_epochs):
    probs = regular_softmax(X_train, W, b)
    gradient_W = np.dot((probs - np.eye(num_classes)[y_train]).T, X_train) /
X_train.shape[0]
    gradient_b = np.mean(probs - np.eye(num_classes)[y_train], axis=0)
    gradient_W = gradient_W.T
    W -= learning_rate * gradient_W.T
    b -= learning_rate * gradient_b

# Evaluate softmax on test data
y_pred_softmax = np.argmax(regular_softmax(X_test, W, b), axis=1)
accuracy_softmax = accuracy_score(y_test, y_pred_softmax)

softmax_end_time = time.time()
softmax_execution_time = softmax_end_time - softmax_start_time

# Model for Hierarchical Softmax
class HierarchicalSoftmaxModel(nn.Module):
    def __init__(self, input_size, root):
        super().__init__()
        self.linear = nn.Linear(input_size, root.layer_size)
        self.root = root

    def forward(self, x):
        return self.linear(x)

# Create a hierarchical tree structure for classes
root = SoftmaxNode("root")
for i in range(num_classes):
    SoftmaxNode(str(i), parent=root)

# Initialize hierarchical softmax model and loss function
model = HierarchicalSoftmaxModel(X.shape[1], root)
criterion = HierarchicalSoftmaxLoss(root=root)
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Convert training data to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
y_train_tensor = torch.LongTensor(y_train)

hierarchical_start_time = time.time()

# Train hierarchical softmax model using Adam optimizer
for epoch in range(num_epochs):
    optimizer.zero_grad()
    outputs = model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()

# Evaluate hierarchical softmax on test data
X_test_tensor = torch.FloatTensor(X_test)
outputs = model(X_test_tensor)
_, y_pred_hierarchical = torch.max(outputs, 1)
accuracy_hierarchical = accuracy_score(y_test, y_pred_hierarchical.numpy())

hierarchical_end_time = time.time()
hierarchical_execution_time = hierarchical_end_time - hierarchical_start_time

print(f"Regular Softmax Accuracy: {accuracy_softmax:.4f}")
print(f"Hierarchical Softmax Accuracy: {accuracy_hierarchical:.4f}")
print(f"Regular Softmax Execution Time: {softmax_execution_time:.2f} seconds")
print(f"Hierarchical Softmax Execution Time: {hierarchical_execution_time:.2f}
seconds")

# Plotting the accuracies and execution times
```

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Accuracy plot
labels = ['Softmax', 'Hierarchical Softmax']
accuracies = [accuracy_softmax, accuracy_hierarchical]
ax1.bar(labels, accuracies, color=['red', 'black'])
ax1.set_ylim(0, 1)
ax1.set_ylabel('Accuracy')
ax1.set_title('Comparison of Softmax and Hierarchical Softmax Accuracies')

# Execution time plot
execution_times = [softmax_execution_time, hierarchical_execution_time]
ax2.bar(labels, execution_times, color=['red', 'black'])
ax2.set_ylabel('Execution Time (seconds)')
ax2.set_title('Comparison of Softmax and Hierarchical Softmax Execution
Times')
plt.tight_layout()
plt.show()
```

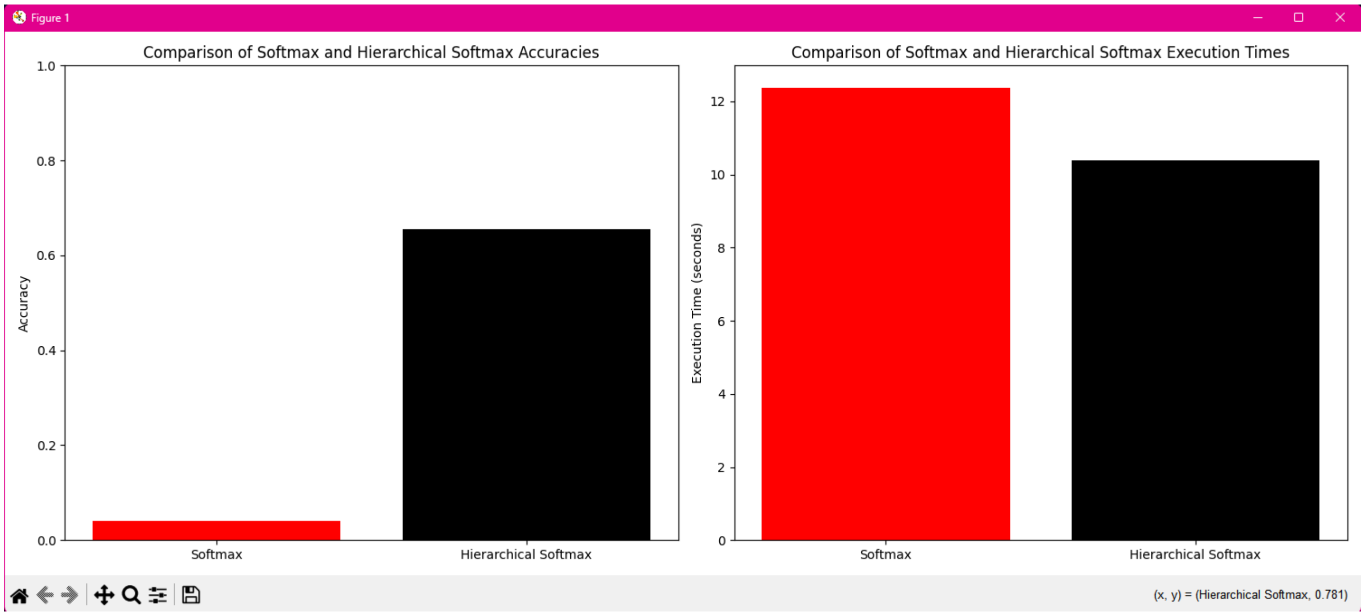
This code is pasted from my file which can be found in this repository. Feel free to play around and experiment with it on different datasets to gain a better understanding.

Code Output

Terminal

```
Regular Softmax Accuracy: 0.0528
Hierarchical Softmax Accuracy: 0.6557
Regular Softmax Execution Time: 12.36 seconds
Hierarchical Softmax Execution Time: 10.39 seconds
```

Plot



Breakdown and Analysis

As mentioned earlier, it is evident here that the Softmax function struggled to classify all the words given a large database whereas the Hierarchal Softmax was able to at least achieve an accuracy of roughly to 66% compared to 5%; a drastic difference. Although it may not be an *A* accuracy of 80% or higher, it is still something to notice as without hierarchical softmax the model wouldn't have been able to return relevant or viable information. As for the time, 2 seconds doesn't seem like a long time but as the dataset grows this will become an increasing issue so it is good that Hierarchical Softmax is more efficient timewise as well. This goes on to further solidify the importance of Hierarchical Softmax.

Numerical Demonstration of the Update Rule

```
           Root
         /      \
      Node 1    Node 2
      /   \     /   \
    Gucci YSL Dior Polo

Let's say the root has the following probabilities:
P(left) = 0.6
P(right) = 0.4

And Node 1:
P(left) = 0.7
P(right) = 0.3
```

```
and Node 2:
P(left) = 0.2
P(right) = 0.8

Then =>
P(Gucci) = 0.6 *  0.7 = 0.42
P(YSL) = 0.6 * 0.3 = 0.18
P(Dior) = 0.4 * 0.2 = 0.08
P(Polo) = 0.4 * 0.8 = 0.32
```

You can see how the probabilities of words are calculated here, it is simply just multiply across the probabilities of the path that you take from Root to Leaf nodes. The $P(node) = \sigma(v_w^T, v_n)$ with v_w as the word embedding vector and v_n is the node vector.

Key Takeaways

Evidently, we have seen how Hierarchical Softmax can achieve tasks that are otherwise difficult to complete Softmax on. This was due to it's ability to handle a vast output across various classes with 66% accuracy as opposed to an abysmal 5% from Softmax, while also achieving it in less time, true to that of their respective time complexities. This organized tree structure to Softmax can help with making the process of classification more efficient, making a seemingly impossible task, possible.