Efficient AI

ECE 595

Project Part B

Due Date: April 14, 2024

Project Part B – Model Complexity and Neural Network Pruning

Introduction

In this project phase, we're diving neural network optimization, focusing on a technique called pruning. We start by building a neural network to handle a binary classification task. Then, we get into the nitty-gritty of pruning, particularly using the magnitude-based method. We're going to tweak different pruning thresholds and see how they shake up the number of parameters and classification accuracy. Think of it like trimming down excess baggage without losing too much performance. By digging into the results and generating some graphs, we hope to shed light on the fine line between slimming down models and keeping them sharp.

Python Libraries

The code uses some handy Python libraries to work with images and build a neural network for classification:

1. numpy: A Python library for numerical operations and array handling.
2. matplotlib.pyplot: A library for creating visualizations and plots in Python.
3. torch: A powerful framework for deep learning operations and model building.
4. torch.nn: A module in PyTorch providing tools for building neural networks.
5. torch.optim: A module in PyTorch for optimization algorithms in deep learning.
6. torch.nn.utils: Utilities in PyTorch for neural network operations.
7. torch.utils.data: Tools in PyTorch for handling and processing data.
8. sklearn.datasets.make_classification: A function from scikit-learn to generate synthetic classification datasets.
9. sklearn.model_selection.train_test_split: A function from scikit-learn to split datasets into training and testing sets.

Data Structure of Neural Network

The neural network class defined here serves as the foundational component of our project, representing the core architecture upon which our model is built. This class, named Net, encapsulates the structure of a typical feedforward neural network, comprising input, hidden, and output layers. The __init__ method initializes the network's parameters, including the sizes of the input, hidden, and output layers, along with the activation functions applied at each layer. The forward method defines the forward pass computation through the network, where input data is sequentially processed through the layers, applying linear transformations followed by non-linear activation functions such as ReLU and sigmoid. By encapsulating the neural network architecture within this class, we establish a modular and reusable framework for constructing and training various neural network models, facilitating experimentation and analysis throughout our project.

Neural Network Object
Attributes - fc1: nn.Linear - relu: nn.ReLU - fc2: nn.Linear - sigmoid: nn.Sigmoid
Functions + __init__(input_size, hidden_size, output_size) + forward(x) -> output

```
# Define your neural network class
class Net(nn.Module):
def __init__(self, input_size, hidden_size, output_size):
super(Net, self).__init__()
self.fc1 = nn.Linear(input_size, hidden_size) # First fully connected layer
self.relu = nn.ReLU() # ReLU activation function
self.fc2 = nn.Linear(hidden_size, output_size) # Second fully connected layer
self.sigmoid = nn.Sigmoid() # Sigmoid activation function

def forward(self, x):
out = self.fc1(x) # First linear transformation
out = self.relu(out) # Apply ReLU activation
out = self.fc2(out) # Second linear transformation
out = self.sigmoid(out) # Apply sigmoid activation
return out
```

Data

In this part of the code, we're setting up our dataset for a classification task. We're defining the key parameters that determine the dataset's characteristics, like the number of samples, features, classes, and clusters per class.

Step 1)

Using these parameters, we generate a synthetic dataset using the make_classification function from scikit-learn. This function creates a dataset with specified characteristics, including the number of samples, features, classes, and clusters per class. Once the dataset is generated, we split it into training and testing sets using the train_test_split function, where a certain percentage of the data (specified by test_size) is allocated for testing, and the rest is used for training.

```
# Generating and Splitting the Dataset
num_samples = 2500 # Number of samples
num_features = 25 # Number of features per sample
num_classes = 2 # Number of classes
num_clusters_per_class = 2 # Number of clusters per class
```

Step 2)

In this code snippet, the dataset is being split into training and testing sets to evaluate the performance of a machine learning model. The variable test_size is set to 0.3, indicating that 30% of the data will be allocated for testing, leaving the remaining 70% for training. Additionally, random_state is assigned the value 42, which acts as a seed for the random number generator, ensuring reproducibility of the split across different runs. The number 42 is often used as a placeholder for random_state in data science and machine learning tasks for its arbitrary significance and to ensure reproducibility (“Random State (Pseudo-Random Number) in Scikit Learn.”) The train_test_split function from the scikit-learn library is then employed to perform the actual split. It takes in the input features x and their corresponding labels y, along with the test size and random state parameters, and returns four subsets: x_train (training features), x_test (testing features), y_train (training labels), and y_test (testing labels). Importantly, the function is configured to maintain the balance of classes between the training and testing sets, which is crucial for ensuring unbiased model evaluation, particularly in classification tasks where class distribution may be uneven.

```
# Splitting the dataset while maintaining class balance and reproducibility
x_train, x_test, y_train, y_test = train_test_split(
x, y,
test_size=test_size,
random_state=random_state
)
```

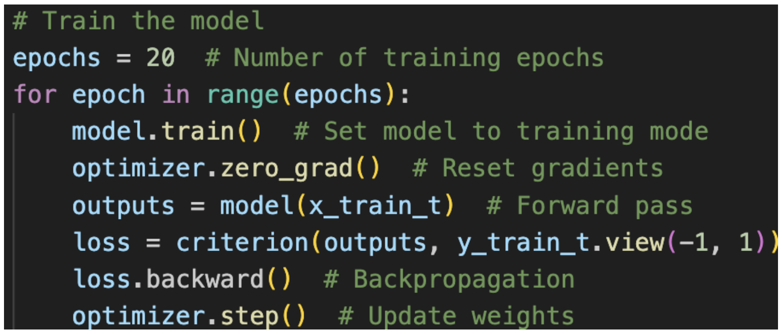
```
# Splitting the dataset into training and testing sets
test_size = 0.3 # Percentage of data for testing
random_state = 42 # Seed for random number generator
```

Step 3) Convert Data to Tensors

Converting data into tensors in PyTorch is essential because PyTorch primarily operates on tensors. Tensors are the fundamental data structure used for computations in PyTorch, enabling seamless integration with PyTorch's deep learning functionalities. By converting data into tensors, it ensures compatibility and efficient computation, facilitating tasks such as model training, evaluation, and inference.

Training the NN Model

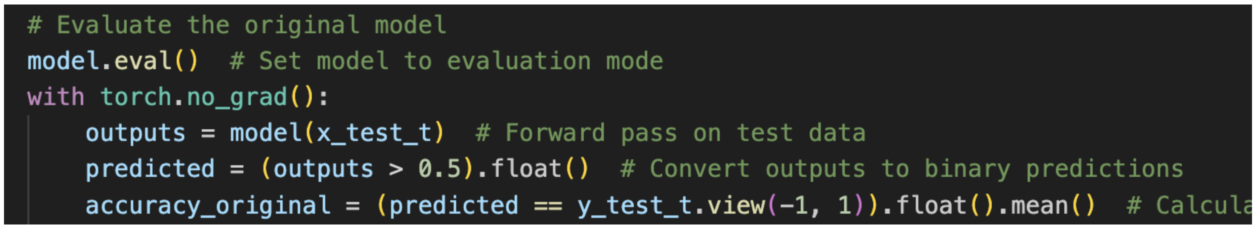
These lines of code are part of a training loop for a neural network model using PyTorch. Firstly, the code sets the number of training epochs to 20, which means the model will be trained over the entire dataset 20 times. Then, there's a loop that iterates over each epoch. Inside the loop, the model is put into training mode using model.train(). This is important because it ensures that certain layers of the model behave differently during training, like dropout layers, which are active only during training to prevent overfitting. Before computing gradients for the current batch of data, the optimizer's gradients need to be reset using optimizer.zero_grad(). This step is necessary because gradients accumulate by default in PyTorch. Next, a forward pass is performed through the model with the training data (x_train_t). This calculates the predicted outputs of the model. Then, the code calculates the loss between the predicted outputs and the actual target values (y_train_t). This is done using a loss function, which measures how far off the model's predictions are from the actual targets. Common loss functions include Mean Squared Error or Cross-Entropy Loss. After calculating the loss, the code performs backpropagation with loss.backward(). This computes the gradients of the loss with respect to all the model parameters. These gradients are then used to update the model's parameters (weights) with optimizer.step(). The optimizer adjusts the weights based on the gradients and the optimization algorithm being used, such as Stochastic Gradient Descent.



```python
# Train the model
epochs = 20  # Number of training epochs
for epoch in range(epochs):
    model.train()  # Set model to training mode
    optimizer.zero_grad()  # Reset gradients
    outputs = model(x_train_t)  # Forward pass
    loss = criterion(outputs, y_train_t.view(-1, 1))
    loss.backward()  # Backpropagation
    optimizer.step()  # Update weights
```

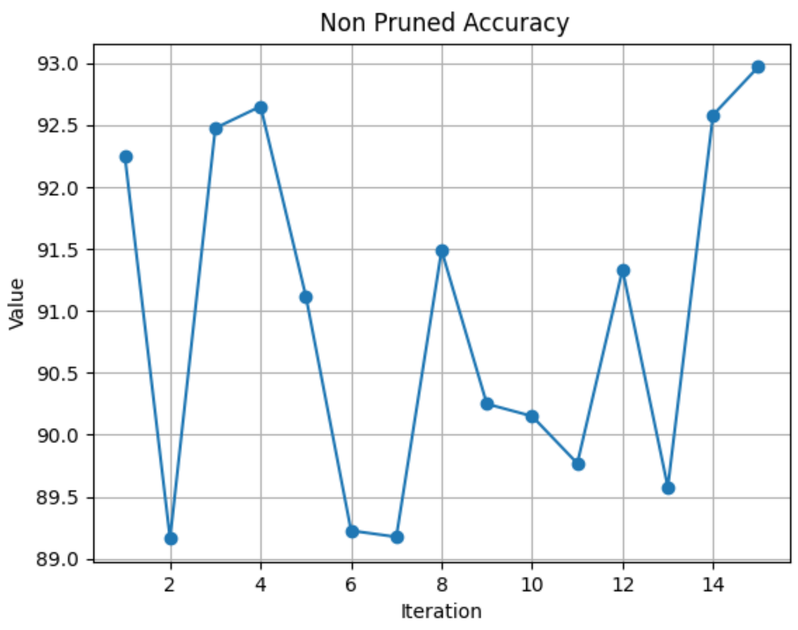
Evaluating the Original Model

First, we set the model to evaluation mode using model.eval(), which ensures that certain operations like dropout are disabled. Then, using torch.no_grad(), we perform a forward pass of the test input data x_test_t through the model to obtain the output predictions outputs. These outputs are then thresholded at 0.5 to convert them into binary predictions, indicating the class labels. Finally, we compare these predicted labels with the true labels y_test_t, calculating the accuracy of the model's predictions by determining the proportion of correct predictions. This accuracy metric, stored in accuracy_original, provides insight into how well the model performs on unseen data and serves as a benchmark for comparison with the pruned models.



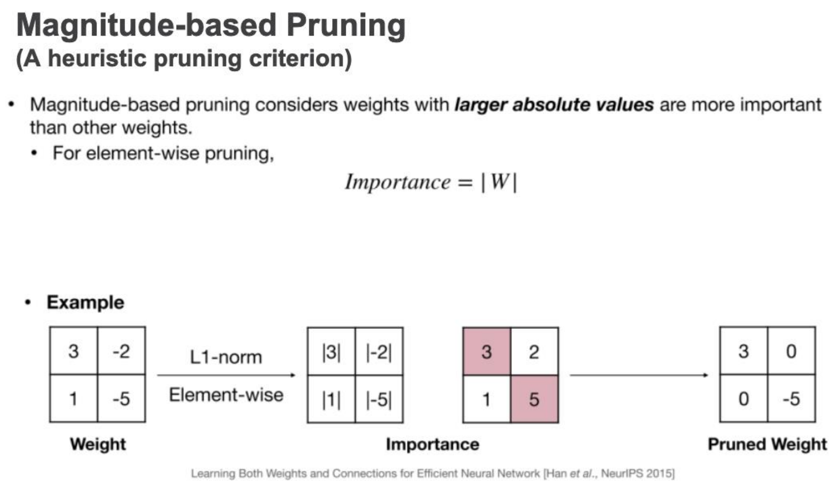
```python
# Evaluate the original model
model.eval()  # Set model to evaluation mode
with torch.no_grad():
    outputs = model(x_test_t)  # Forward pass on test data
    predicted = (outputs > 0.5).float()  # Convert outputs to binary predictions
    accuracy_original = (predicted == y_test_t.view(-1, 1)).float().mean()  # Calcula
```

This is the non-pruned accuracy that was calculated when I ran this code almost 15 times. It fluctuates between .8964 to .9298 accuracy which is not bad without pruning. This is as expected with the various factors considering how we have created the neural network and generated the data.



Magnitude Based Pruning

According to my understanding, magnitude-based pruning is a technique used to reduce the size of neural networks by removing less significant connections. It assesses the magnitude of each weight in the network, considering weights with larger absolute values to be more critical. In essence, the process involves examining the absolute values of all weights, regardless of their signs. The weights with the highest absolute values are retained, while the others are discarded, resulting in a smaller neural network.



Magnitude-based Pruning

(A heuristic pruning criterion)

- Magnitude-based pruning considers weights with ***larger absolute values*** are more important than other weights.
  - For element-wise pruning,

$$Importance = |W|$$

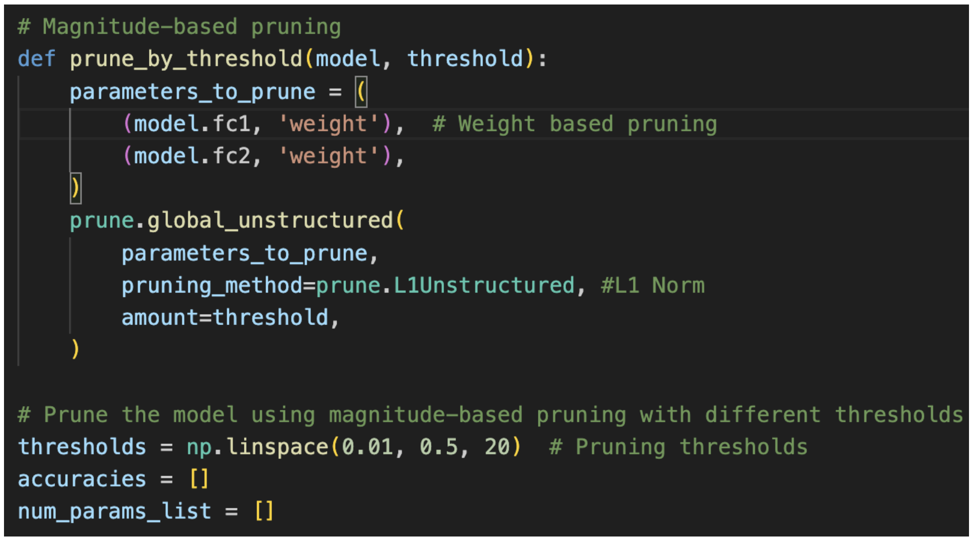
- **Example**

$$\underbrace{\begin{array}{|c|c|}\hline 3 & -2 \\ \hline 1 & -5 \\ \hline\end{array}}_{\textbf{Weight}} \xrightarrow[\text{Element-wise}]{\text{L1-norm}} \underbrace{\begin{array}{|c|c|}\hline |3| & |-2| \\ \hline |1| & |-5| \\ \hline\end{array} \quad \begin{array}{|c|c|}\hline 3 & 2 \\ \hline 1 & 5 \\ \hline\end{array}}_{\textbf{Importance}} \longrightarrow \underbrace{\begin{array}{|c|c|}\hline 3 & 0 \\ \hline 0 & -5 \\ \hline\end{array}}_{\textbf{Pruned Weight}}$$

Learning Both Weights and Connections for Efficient Neural Network [Han *et al.*, NeurIPS 2015]

The code segment is dedicated to implementing magnitude-based pruning, a technique used to reduce the size of neural network models by eliminating less critical connections, thereby improving computational efficiency without significantly sacrificing performance. The prune_by_threshold function takes two arguments: the neural network model and a threshold value. This function prepares the model for pruning by specifying which parameters to prune, in this case, the weights of two fully connected layers (fc1 and fc2).

The threshold determines the magnitude below which weights will be pruned. After defining the pruning function, the code proceeds to prune the model using a range of thresholds. It generates a list of 20 thresholds evenly spaced between 0.01 and 0.5 using NumPy's linspace function. For each threshold, the model is pruned accordingly, and the number of remaining parameters is calculated and stored in num_params_list. Additionally, the accuracy of the pruned model on a test dataset is computed and stored in accuracies. These values will later be used to analyze the trade-off between model size reduction and performance degradation across different pruning thresholds.



```python
# Magnitude-based pruning
def prune_by_threshold(model, threshold):
    parameters_to_prune = (
        (model.fc1, 'weight'),  # Weight based pruning
        (model.fc2, 'weight'),
    )
    prune.global_unstructured(
        parameters_to_prune,
        pruning_method=prune.L1Unstructured, #L1 Norm
        amount=threshold,
    )

# Prune the model using magnitude-based pruning with different thresholds
thresholds = np.linspace(0.01, 0.5, 20)  # Pruning thresholds
accuracies = []
num_params_list = []
```

Analysis and Understanding:

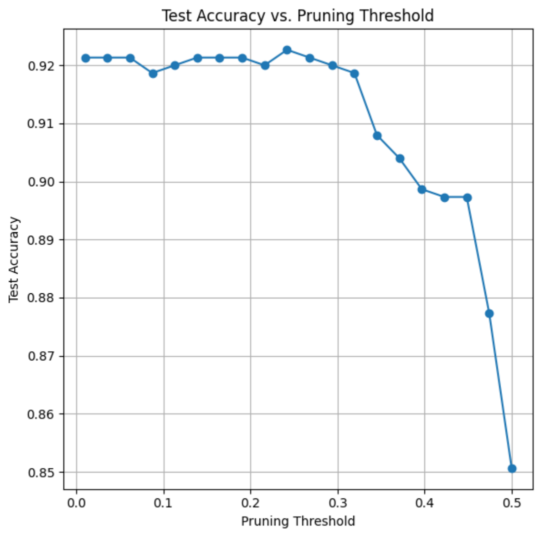
As we adjust the pruning threshold in magnitude-based pruning, we notice changes in both the number of parameters and the classification accuracy. Initially, as we increase the threshold and prune more weights, the number of parameters decreases steadily. This reduction occurs because less important connections are removed from the neural network, making it sparser. However, as the threshold continues to rise, the decrease in the number of parameters may slow down or stabilize, indicating that more critical connections are being pruned.

Regarding classification accuracy, we observe that at lower thresholds, where only a small fraction of weights are pruned, the accuracy remains similar to that of the unpruned model. As we increase the threshold further, accuracy may start to decrease gradually, reflecting the removal of connections important for decision-making. Eventually, beyond a certain threshold, the decrease in accuracy may become more pronounced, indicating that critical connections necessary for capturing important data patterns are being removed, significantly impacting model performance. This suggests a delicate balance between reducing model size and preserving predictive power, requiring careful experimentation and evaluation of pruning thresholds.

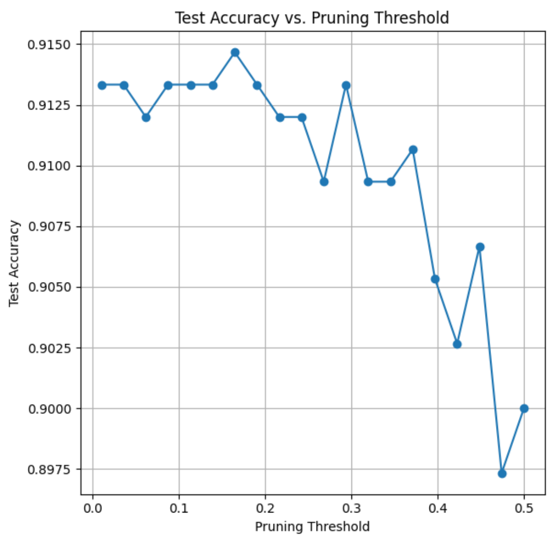
Here I tested out with different number of features and different number of thresholds:

```
thresholds = np.linspace(0.01, 0.5, 20) # Pruning thresholds
```

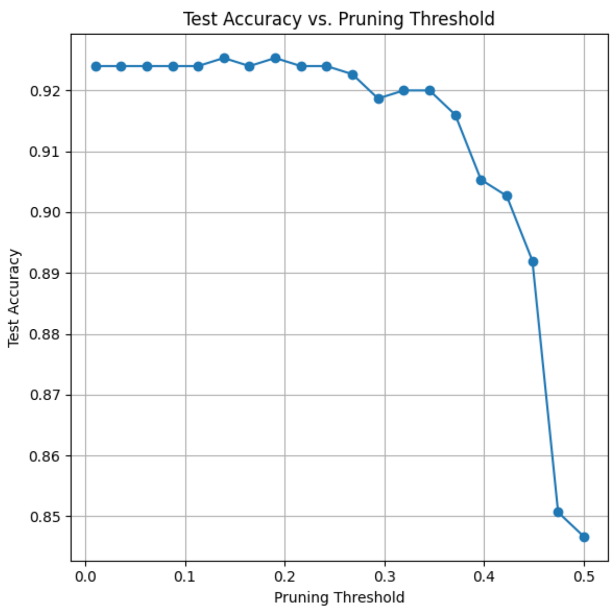
Number of features: 10



Number of features: 20

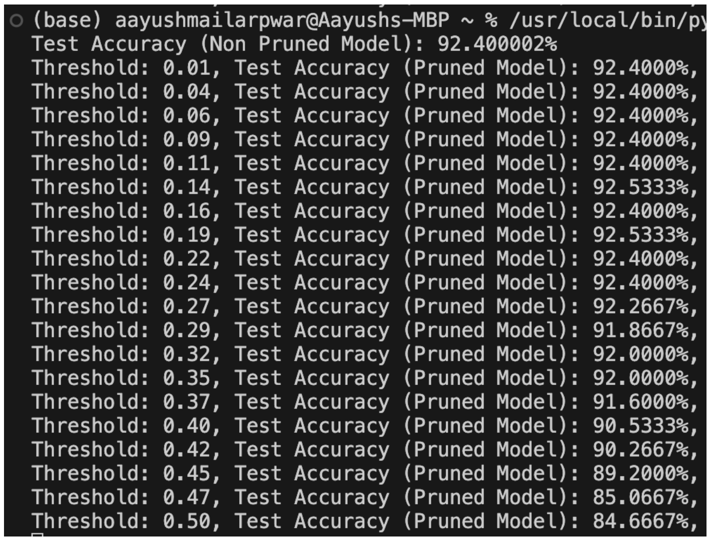


Number of features: 30



These three figures depict comparable information, illustrating the specific weight thresholds at which accuracies start to decline. Given the absence of biases within the model to counterbalance the weights, most weights tend to have modest values, even when considering the entire weight tensor.

Sample Output (Keep running with different model structures):



```
○ (base) aayushmailarpwar@Aayushs-MBP ~ % /usr/local/bin/py
  Test Accuracy (Non Pruned Model): 92.400002%
  Threshold: 0.01, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.04, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.06, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.09, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.11, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.14, Test Accuracy (Pruned Model): 92.5333%,
  Threshold: 0.16, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.19, Test Accuracy (Pruned Model): 92.5333%,
  Threshold: 0.22, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.24, Test Accuracy (Pruned Model): 92.4000%,
  Threshold: 0.27, Test Accuracy (Pruned Model): 92.2667%,
  Threshold: 0.29, Test Accuracy (Pruned Model): 91.8667%,
  Threshold: 0.32, Test Accuracy (Pruned Model): 92.0000%,
  Threshold: 0.35, Test Accuracy (Pruned Model): 92.0000%,
  Threshold: 0.37, Test Accuracy (Pruned Model): 91.6000%,
  Threshold: 0.40, Test Accuracy (Pruned Model): 90.5333%,
  Threshold: 0.42, Test Accuracy (Pruned Model): 90.2667%,
  Threshold: 0.45, Test Accuracy (Pruned Model): 89.2000%,
  Threshold: 0.47, Test Accuracy (Pruned Model): 85.0667%,
  Threshold: 0.50, Test Accuracy (Pruned Model): 84.6667%,
```

Conclusion

In conclusion, this project demonstrated the effectiveness of magnitude-based pruning in reducing the size of neural network models while maintaining reasonable classification accuracy. By iteratively adjusting the pruning threshold, we observed changes in the number of parameters and classification accuracy. The results highlight the trade-off between model size reduction and preservation of predictive power. While higher pruning thresholds led to fewer parameters, they also resulted in a decline in classification accuracy, indicating the importance of selecting an appropriate threshold based on specific application requirements. Overall, this project underscores the potential of pruning techniques in optimizing deep learning models for efficient deployment in resource-constrained environments.

References

“Pytorch-Tutorial/Tutorials/01-Basics/Feedforward_neural_network/Main.py at Master · Yunjey/Pytorch-Tutorial.” *GitHub*, github.com/yunjey/pytorch-tutorial/blob/master/tutorials/01-basics/feedforward_neural_network/main.py.

“Random State (Pseudo-Random Number) in Scikit Learn.” *Stack Overflow*, stackoverflow.com/questions/28064634/random-state-pseudo-random-number-in-scikit-learn Accessed 15 Apr. 2024.

(Adrian Tam, 2023, April 13). Training a PyTorch Model with DataLoader and Dataset. Machine Learning Mastery. https://machinelearningmastery.com/training-a-pytorch-model-with-dataloader-and-dataset/

Codebasics. (2019, March 8). Neural Networks Explained | Part 1: Building a Neural Network [Video]. YouTube. https://www.youtube.com/watch?v=w5WiUcDJosM&list=PL80kAHvQbh-pT4lCkDT53zT8DKmhE0idB&index=5

Appendix Code

```
# For numerical operations and array handling
import numpy as np

# For plotting graphs and visualizations
import matplotlib.pyplot as plt

# For deep learning operations and model building
import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn.utils import prune
from torch.utils.data import DataLoader, TensorDataset

# For generating synthetic dataset for classification
from sklearn.datasets import make_classification

# For splitting dataset into train and test sets
from sklearn.model_selection import train_test_split


# Define your neural network class
class Net(nn.Module):
def __init__(self, input_size, hidden_size, output_size):
super(Net, self).__init__()
self.fc1 = nn.Linear(input_size, hidden_size) # First fully connected layer
self.relu = nn.ReLU() # ReLU activation function
self.fc2 = nn.Linear(hidden_size, output_size) # Second fully connected layer
self.sigmoid = nn.Sigmoid() # Sigmoid activation function

def forward(self, x):
out = self.fc1(x) # First linear transformation
out = self.relu(out) # Apply ReLU activation
out = self.fc2(out) # Second linear transformation
out = self.sigmoid(out) # Apply sigmoid activation
return out

# Generating and Splitting the Dataset
num_samples = 2500 # Number of samples
num_features = 10 # Number of features per sample
num_classes = 2 # Number of classes
num_clusters_per_class = 2 # Number of clusters per class

# Generating synthetic classification dataset
```

```
x, y = make_classification(
n_samples=num_samples,
n_features=num_features,
n_classes=num_classes,
n_clusters_per_class=num_clusters_per_class,
random_state=42
)

# Splitting the dataset into training and testing sets
test_size = 0.3 # Percentage of data for testing
random_state = 42 # Seed for random number generator

# Splitting the dataset while maintaining class balance and reproducibility
x_train, x_test, y_train, y_test = train_test_split(
x, y,
test_size=test_size,
random_state=random_state
)

# Convert data to PyTorch tensors
x_train_t = torch.tensor(x_train, dtype=torch.float32) # Convert training input data to PyTorch tensor
y_train_t = torch.tensor(y_train, dtype=torch.float32) # Convert training target data to PyTorch tensor
x_test_t = torch.tensor(x_test, dtype=torch.float32) # Convert testing input data to PyTorch tensor
y_test_t = torch.tensor(y_test, dtype=torch.float32) # Convert testing target data to PyTorch tensor

# Create the model
input_size = x_train.shape[1] # Input size based on number of features
hidden_size = 256 # Hidden layer size
output_size = 1 # Output layer size
model = Net(input_size, hidden_size, output_size) # Instantiate the neural network model

# Define loss function and optimizer
criterion = nn.BCELoss() # Binary cross-entropy loss function
optimizer = optim.Adam(model.parameters(), lr=0.01) # Adam optimizer with LR 0.01

# Train the model
epochs = 20 # Number of training epochs
for epoch in range(epochs):
model.train() # Set model to training mode
optimizer.zero_grad() # Reset gradients
outputs = model(x_train_t) # Forward pass
loss = criterion(outputs, y_train_t.view(-1, 1)) # Calculate loss
loss.backward() # Backpropagation
optimizer.step() # Update weights

# Evaluate the original model
model.eval() # Set model to evaluation mode
with torch.no_grad():
```

```
outputs = model(x_test_t) # Forward pass on test data
predicted = (outputs > 0.5).float() # Convert outputs to binary predictions
accuracy_original = (predicted == y_test_t.view(-1, 1)).float().mean() # Calculate accuracy
print(f'Test Accuracy (Non Pruned Model): {accuracy_original.item()*100:.6f}%') # Print original model accuracy

# Magnitude-based pruning
def prune_by_threshold(model, threshold):
parameters_to_prune = (
(model.fc1, 'weight'), # Weight based pruning
(model.fc2, 'weight'),
)
prune.global_unstructured(
parameters_to_prune,
pruning_method=prune.L1Unstructured, #L1 Norm
amount=threshold,
)

# Prune the model using magnitude-based pruning with different thresholds
thresholds = np.linspace(0.01, 0.5, 20) # Pruning thresholds
accuracies = []
num_params_list = []

for threshold in thresholds:
# Clone the original model
pruned_model = Net(input_size, hidden_size, output_size) # Instantiate new model
pruned_model.load_state_dict(model.state_dict()) # Load weights from original model

# Prune the cloned model
prune_by_threshold(pruned_model, threshold)

# Count the number of parameters
num_params = sum(p.numel() for p in pruned_model.parameters())
# Evaluate the pruned model
pruned_model.eval() # Set pruned model to evaluation mode
with torch.no_grad():
outputs = pruned_model(x_test_t) # Forward pass on test data
predicted = (outputs > 0.5).float() # Convert outputs to binary predictions
accuracy = (predicted == y_test_t.view(-1, 1)).float().mean() # Calculate accuracy
print(f'Threshold: {threshold:.2f}, Test Accuracy (Pruned Model): {accuracy.item()*100:.4f}%, Number of Parameters:
{num_params}')
accuracies.append(accuracy.item())
num_params_list.append(num_params)

# Plotting
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(thresholds, accuracies, marker='o', linestyle='-')
```

```python
plt.title('Test Accuracy vs. Pruning Threshold')
plt.xlabel('Pruning Threshold')
plt.ylabel('Test Accuracy')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(thresholds, num_params_list, marker='o', linestyle='-')
plt.title('Number of Parameters vs. Pruning Threshold')
plt.xlabel('Pruning Threshold')
plt.ylabel('Number of Parameters')
plt.grid(True)

plt.tight_layout()
plt.show()
```