SPRINT 3 DOCUMENT

SPACE TRAFFIC DENSITY PREDICTION

Team Members

1.Shravani R S

2.Himaja S

3.Tarunaa A C

4.Karanbir Singh

5.Anushka Dutta

Date: 19th December 2024

Prepared By: Shravani R S

Table of Contents

1.1 Hyperparameter Tuning

Hyperparameter tuning is a crucial step in optimizing the performance of machine learning models. The main techniques used for hyperparameter tuning are:

- **Grid Search**: Exhaustively searches through a manually specified subset of hyperparameters.
- **Random Search**: Randomly selects combinations of hyperparameters to test.

Tuning involves selecting values for parameters like learning rate, batch size, and model architecture to achieve the best model performance.

1.2 Data Loading and Preprocessing

Before a model can be trained, the data must be loaded and preprocessed. Common steps include:

- **Data Loading**: Import data from files (CSV, JSON, images) or databases.
- **Data Preprocessing**:
    - Normalization and Standardization: Scaling the features so they are on the same scale.
    - Handling Missing Data: Techniques like imputation or removal.
    - Data Augmentation: Used mostly for image data (e.g., flipping, rotating, zooming).
    - Encoding Categorical Data: Converting categorical data into numerical format.
    -

1.3 Model Definition and Hyperparameter Tuning

The model is defined using a framework like PyTorch. In PyTorch, we define a model by subclassing nn.Module and implementing the forward method. Hyperparameter tuning can be performed using techniques such as Grid Search to find the best configuration of model parameters.

1.4 Model Evaluation

After training a model, it's evaluated using performance metrics such as accuracy, precision, recall, F1-score, and loss. Common steps include:

- **Train-Validation Split**: The dataset is split into training and validation sets.
- **Loss and Accuracy**: During training, monitor the loss and accuracy metrics to evaluate model performance.

1.5 Visualization

Visualization tools such as **TensorBoard** allow the visualization of metrics such as loss curves, accuracy, and model architecture. This helps in real-time tracking of training progress.

2. PyTorch Model Development

2.1 Set Up PyTorch Model

To create a PyTorch model:

- **Define a Class**: Create a class that inherits from torch.nn.Module.
- **Forward Pass**: Implement the forward method that defines the computation performed at every call.

Example:

python

Copy code

```
import torch
import torch.nn as nn


class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.layer1 = nn.Linear(64, 32)
        self.relu = nn.ReLU()
        self.output = nn.Linear(32, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        return self.output(x)
```

2.2 Define Hyperparameters and Model Training

Important hyperparameters include:

- **Learning Rate**
- **Batch Size**
- **Number of Epochs**
- **Optimizer Type** (e.g., Adam, SGD)

These are defined before training begins. Once set, the model is trained using the training data by passing it through the network, calculating the loss, and updating weights using backpropagation.

2.3 Hyperparameter Tuning with GridSearch

GridSearch is used to systematically explore multiple hyperparameter combinations. This involves defining a grid of hyperparameters and evaluating the model's performance for each combination.

python

Copy code

```
from sklearn.model_selection import GridSearchCV


# Example of parameter grid

param_grid = {'lr': [0.01, 0.001], 'batch_size': [16, 32]}


grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=3)

grid_search.fit(X_train, y_train)
```

2.4 Model Training

Training involves:

1. Loading the data.
2. Passing the data through the model.
3. Computing the loss.
4. Optimizing the model weights using an optimizer.
5. Repeating the process for several epochs.

2.5 Model Evaluation and Metrics

After training, evaluate the model on a validation set. Common metrics include:

- **Accuracy**: The fraction of correct predictions.
- **Precision and Recall**: Measures for classification tasks, especially for imbalanced datasets.
- **Loss Function**: A measure of how well the model performs.

3. Model Optimization

3.1 Model Saving and Loading

Once the model is trained, it is saved using:

python

Copy code

```
torch.save(model.state_dict(), 'model.pth')
```

To load the model for future use, use:

python

Copy code

```
model.load_state_dict(torch.load('model.pth'))
```

3.2 Model Pruning

Model pruning involves reducing the size of the network by eliminating less important weights, leading to improved inference speed and reduced memory usage.

3.3 Transfer Learning

Transfer learning involves using a pre-trained model (e.g., ResNet, VGG) and fine-tuning it on the target task. This is especially useful when you have a limited dataset.

3.4 Distributed Training

Distributed training splits the workload across multiple devices or nodes, reducing training time for large models and datasets.

3.5 TensorBoard for Visualization

Use TensorBoard to track metrics like loss and accuracy during training. Install with:

bash

Copy code

```
pip install tensorboard
```

Integrate it into PyTorch with:

python

Copy code

```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter()
```

3.6 Deployment and Inference

Deploying the model typically involves exposing it through an API. The model can serve predictions by receiving input data, making predictions, and sending results to clients.

3.7 Quantization

Quantization reduces the precision of the model's weights and activations, which leads to smaller and faster models. This is particularly useful for deploying models on edge devices.

3.8 Custom Loss Functions and Optimizers

In some cases, predefined loss functions and optimizers may not be suitable. PyTorch allows for defining custom loss functions and optimizers for specific use cases.

python

Copy code

```
class MyLoss(nn.Module):
    def forward(self, output, target):
        return torch.mean((output - target) ** 2)
```

4. Deployment Phase

4.1 Model Exporting

Once a model is trained and optimized, export it using:

python

Copy code

```
torch.onnx.export(model, dummy_input, "model.onnx")
```

4.2 Creating an API for Model Serving

To serve the model for real-time predictions, you can create a REST API using frameworks like **Flask** or **FastAPI**.

Example with **Flask**:

python

Copy code

```
from flask import Flask, request, jsonify
import torch
```

```
app = Flask(__name__)

model = torch.load('model.pth')

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    # Preprocess data, make prediction
    result = model(data)
    return jsonify(result)

if __name__ == '__main__':
    app.run(debug=True)
```

5. Grid Search in PyTorch

5.1 Imports and Model Definition

GridSearch can be applied by defining the model and specifying the parameter grid.

5.2 Data Generation and Preprocessing

The data should be preprocessed before applying GridSearch, similar to the steps above.

5.3 Hyperparameter Grid Definition

Define the range of hyperparameters to explore, such as learning rate, batch size, and number of layers.

5.4 Grid Search and Model Training

Grid search iterates through the defined hyperparameter combinations, training the model on each and selecting the best-performing combination.

5.5 Model Evaluation on Validation Set

After each grid search iteration, evaluate the model on the validation set to track the performance.

5.6 Tracking Best Hyperparameters

Track and log the best-performing hyperparameter set for future use.