Project Report: American Sign Language Prediction using Transfer Learning EfficientNet-B0

1. Introduction

This project report outlines the development of an image classification system designed to recognize American Sign Language (ASL) hand signs. The system takes an ASL input image and determines the corresponding ASL symbol. The project uses a pre-trained `EfficientNet-B0` model. The report highlights the implementation of transfer learning and building a image classification model.

2. Project Objectives

- Develop a multi-class image classification model capable of identifying 29 distinct ASL hand signs (A-Z, nothing, del, space).
- Apply transfer learning to leverage a pre-trained `EfficientNet-B0` model, reducing training time and resource usage.
- Fine-tune the `EfficientNet-B0` model, allowing for adaptation to the specific ASL dataset.
- Implement distributed training with PyTorch for multi-GPU usage for performance and time savings.
- Evaluate the model's performance using appropriate metrics and visualizations.
- Use **MLflow** via **Dagshub** to enable experiment tracking and management.
- Delploy a `Streamlit` app for user interaction and model predictions.

3. Methodology

The project follows these steps:

3.1. Data Acquisition and Preparation (Section 1)

- **Dataset Download:** The project starts by downloading the ASL alphabet dataset from Kaggle using the `kagglehub` library. The library is installed if it is not available, making the code portable to other environments where the library might not be installed.

```python
try:
   import kagglehub
except:
  !pip install kagglehub
import kagglehub

# Download latest version
asl_path = kagglehub.dataset_download('grassknoted/asl-alphabet')
print('Data source import complete. Dataset path: ', asl_path)
```

- **Training & Testing Directory Listing:**
  - The code also lists the directories inside the `asl_alphabet_train` directory to see the different alphabets and symbols categories available.
  - The code also lists out all the test images present in the directory.

```python
!ls /root/.cache/kagglehub/datasets/grassknoted/asl-alphabet/versions/1/asl_alphabet_train/asl_alphabet_train
```

```
!ls /root/.cache/kagglehub/datasets/grassknoted/asl-alphabet/versions/1/asl_alphabet_test
```

3.2. Data Loading and Preprocessing (Section 2)

- `data_loader.py` **Script:** This script handles the loading, preprocessing, and data augmentation for the ASL image dataset.
- **Pre-Trained Weights Loading:** Pre-trained weights for the `EfficientNet-B0` model are loaded.

```python
weights = torchvision.models.EfficientNet_B0_Weights.DEFAULT
model = torchvision.models.efficientnet_b0(weights=weights)
```

- **Image Transformations:** A transformation pipeline is defined for image preprocessing, including resizing, normalization, and conversion to tensors. The transforms are saved to a file so that they can be applied later while doing inference on test data.

```python
auto_transforms = weights.transforms()
torch.save(auto_transforms, "effnetb0_transform.pt")
```

- **Custom Model Classifier:** A custom classifier layer is created and replaced with the original at the end of `EfficientNet-B0` architecture. This layer is used to output the 29 categories of the ASL alphabet.

```python
model.classifier = nn.Sequential(
    nn.Dropout(p=0.2, inplace=True),
    nn.Linear(in_features=1280,
    out_features=len(class_names),
    bias=True)
)
```

- **Data Loaders:** PyTorch's `ImageFolder` and `DataLoader` classes are used to create the training dataset and data loaders with an option for `DistributedSampler` to enable multi-gpu training.

3.3. PyTorch Engine (Section 3)

- `pt_engine.py` **Script:** This script contains the code for the training loop, evaluation loop, early stopping, and data saving.
- `CustomTrainer` **Class:** This class encapsulates the entire training process. It is initialized with all the required elements for the training including model, dataloaders, criterion, optimizer, gpu-id, path, patience, max_epochs, world_size, and a scheduler.
- **Distributed Data Parallel:** If GPUs are used, the model is wrapped in `DistributedDataParallel`, this allows the model training to happen in parallel, utilizing the gpus.
- **Training Loop:** The `run_batch()` function handles the training of each batch of training data, calculates loss, backpropagates the loss and updates the model parameters using the optimizer.
- **Evaluation Loop:** The model is trained by performing an evaluation after each epoch, where the evaluation metrics are calculated using the validation dataset.
- **Early Stopping:** A custom early stopping mechanism is defined to prevent overfitting.
- **Metric Calculation:** `torchmetrics` library provides calculation of evaluation metrics, including accuracy and f1 score.
- **Tensor Gathering:** The results such as training losses, f1 scores and accuracies are gathered from all available GPUs to calculate overall performance of the training.

3.4. Training Script (Section 4)

- `pt_train.py` **Script:** The training script integrates all components and performs the overall model training.
- **Argument Parsing:** The script uses `argparse` to specify training parameters, including epochs, batch size, GPU usage, model save path, training path, learning rate, world size and learning rate scheduler.
- **Distributed Training:** If GPUs are being used, distributed training is implemented using PyTorch's DDP. The script initializes process groups and cleans up after training is finished. It also sets up the device being used for training and evaluation.
- **Free Port:** The script also includes a function `find_free_port()` to find an available port on the system, to be used for the distributed training communication.
- `set_seed` **:** The script uses the `set_seed` function to control the randomness, to allow for reproducible model training and evaluations.
- **Main Training Loop:** The training script uses `mp.spawn` to handle multi-gpu training and executes the main function which utilizes the `CustomTrainer` to train the model.

3.5. Model Training (Section 5)

- **Command-line Execution:** The `pt_train.py` script is run from the notebook. The CLI allows the user to specify training parameters, including the data to use for training, batch size, learning rate, and number of epochs, etc.

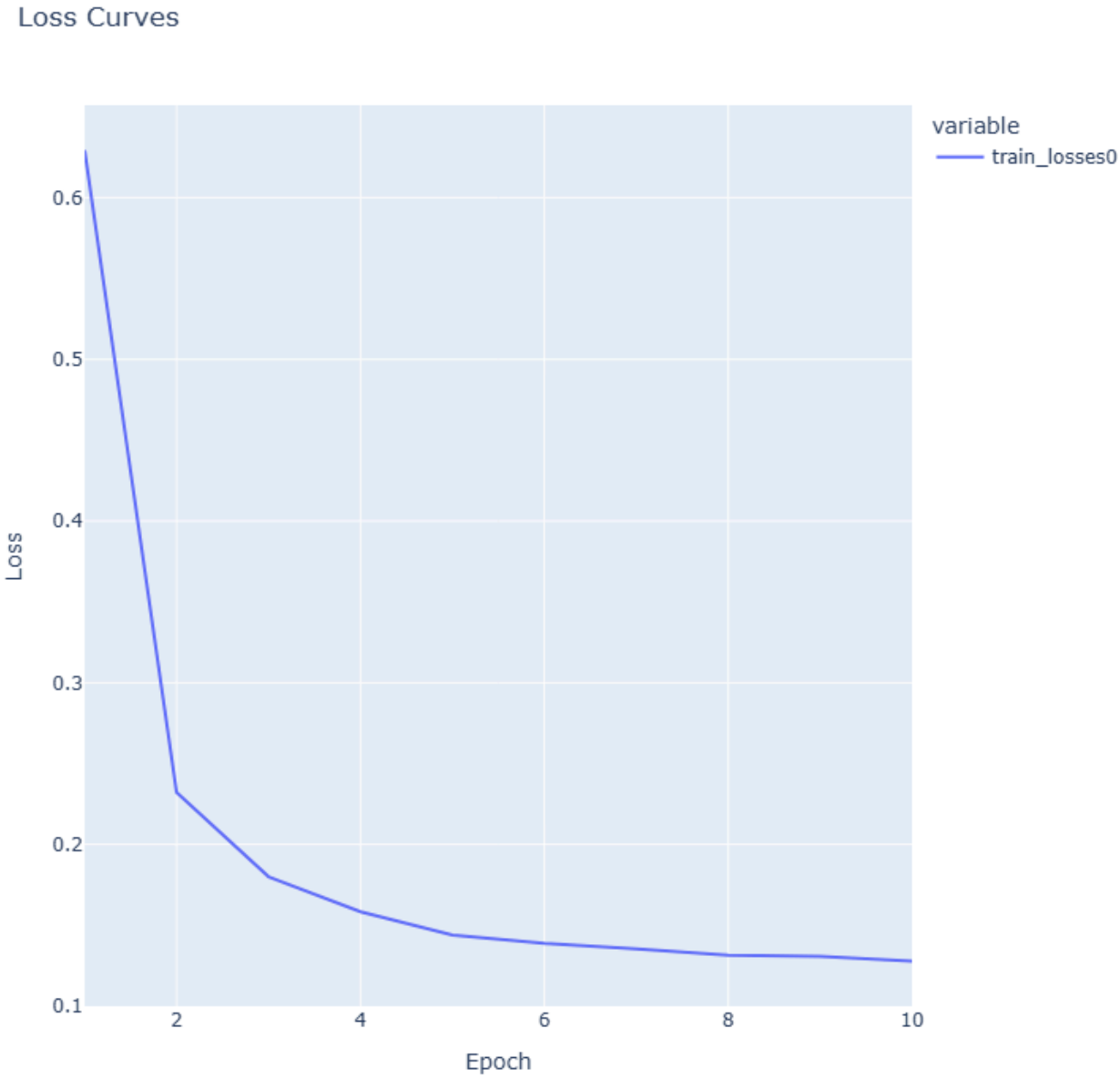
```python
!python pt_train.py --total_epochs 10 --batch_size 64 --gpu --xtrain_path /root/.cache/kagglehub/datasets/grassknoted/asl-alphabet/versions/1/asl_alphabet_train/asl_alphabet_train --learning_rate 0.001 --world_size 1
```

3.6. Model Evaluation (Section 6)

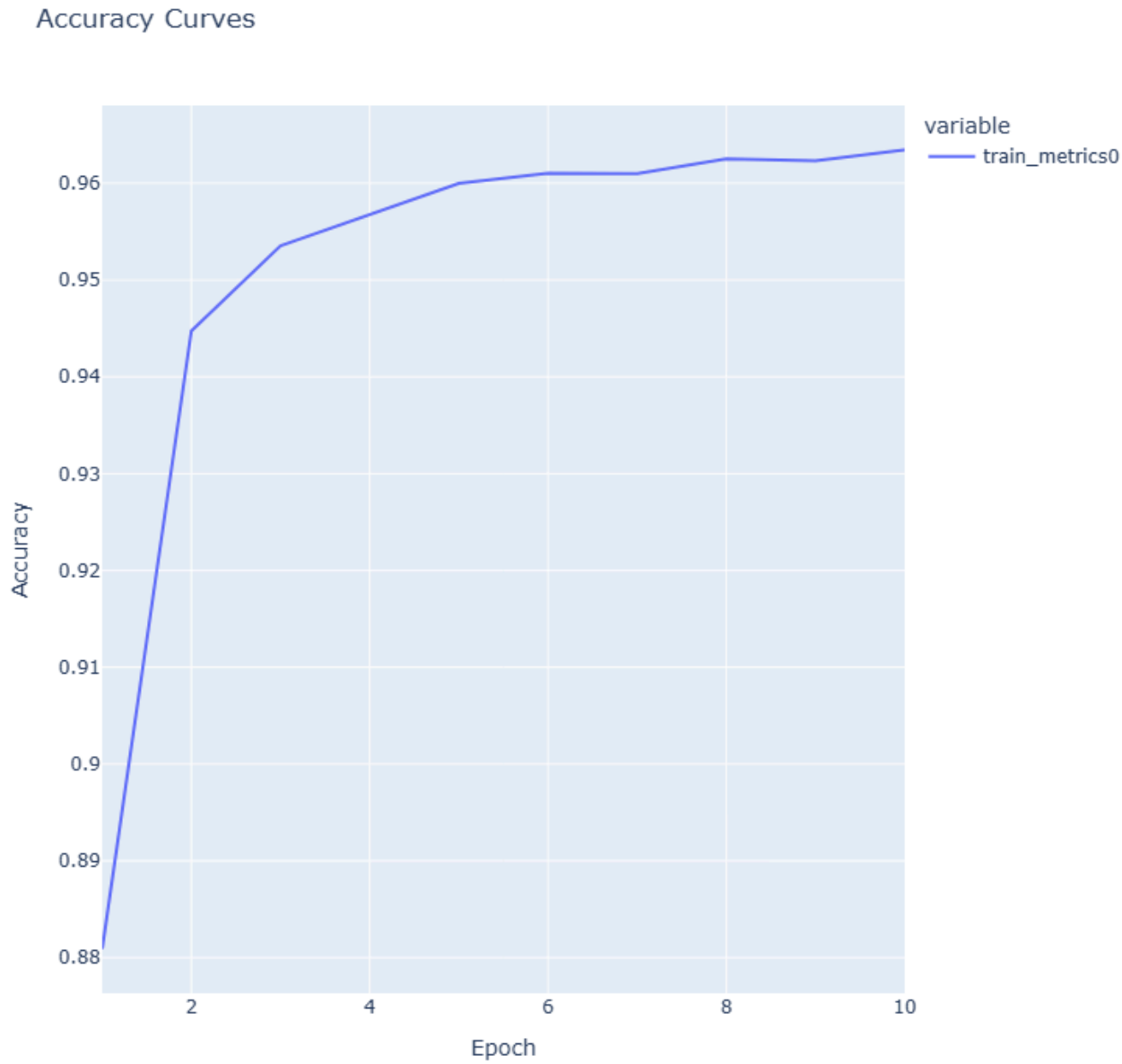
- **Classification Report:** The model performs well on most classes, with high precision, recall, and F1-score values for most classes. This suggests that the model is able to accurately classify those classes in the training dataset. Lower values of precision, recall and F1-score for some classes suggest the need to inspect those classes further, as the model may have some difficulty recognizing patterns related to those classes in the training data. There are a couple of classes for which this is the case in the provided classification report. The results highlight the overall effectiveness of the model in accurately classifying images of sign language letters, with room for potential improvements for certain letter classification using additional training or data augmentation techniques. The high accuracy and relatively balanced scores across classes in the classification report indicate a robust model with good generalization capabilities, making it suitable for detecting different classes of input images.

```
Classification Report:
              precision    recall  f1-score   support

           A       0.97      0.97      0.97      3000
           B       0.98      0.98      0.98      3000
           C       0.99      0.99      0.99      3000
           D       0.99      0.99      0.99      3000
           E       0.97      0.96      0.97      3000
           F       0.99      0.99      0.99      3000
           G       0.98      0.97      0.98      3000
           H       0.98      0.99      0.98      3000
           I       0.97      0.96      0.96      3000
           J       0.97      0.97      0.97      3000
           K       0.97      0.97      0.97      3000
           L       0.98      0.99      0.99      3000
```

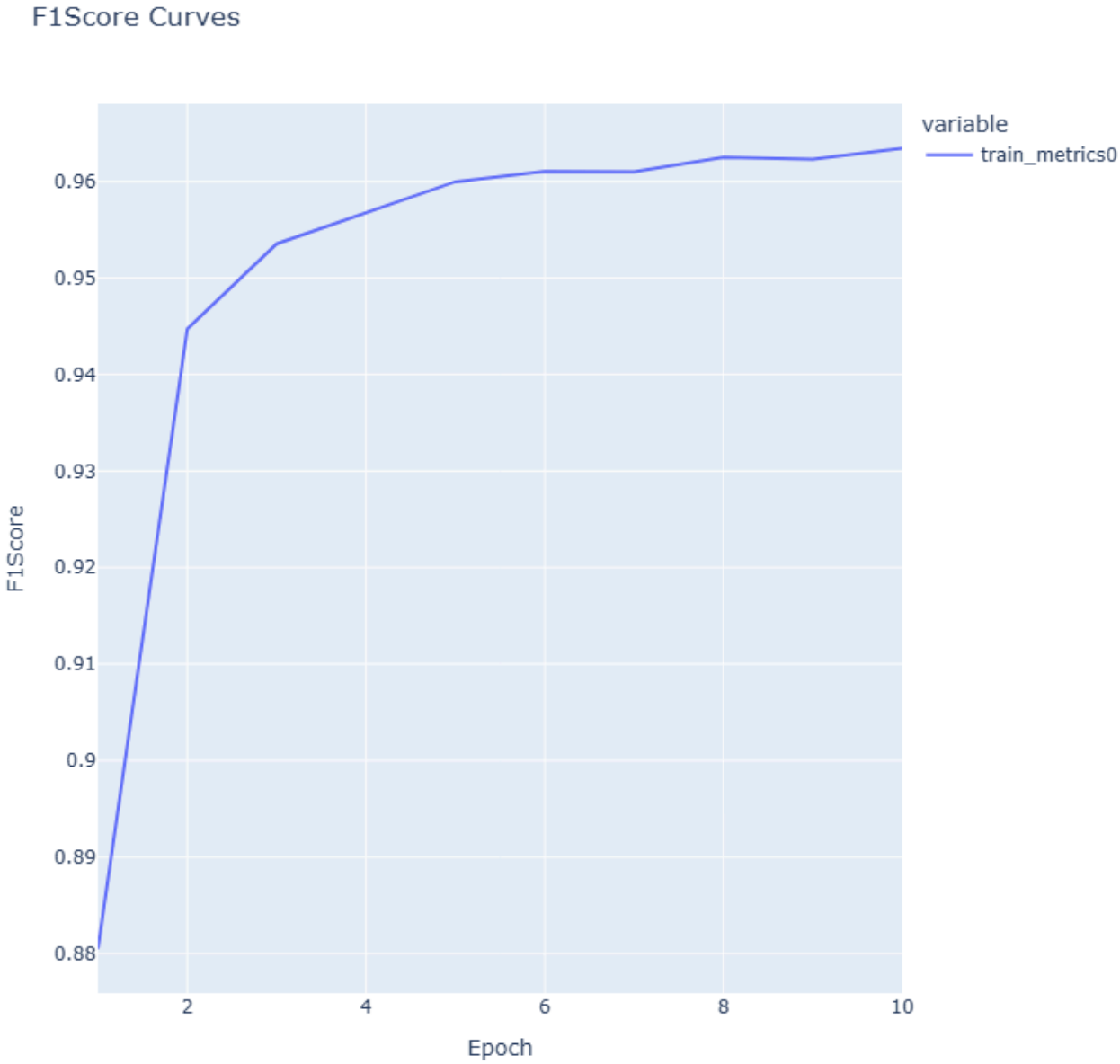
```
              M       0.95      0.95      0.95      3000
              N       0.95      0.96      0.96      3000
              O       0.98      0.98      0.98      3000
              P       0.98      0.97      0.97      3000
              Q       0.97      0.98      0.98      3000
              R       0.92      0.92      0.92      3000
              S       0.93      0.93      0.93      3000
              T       0.94      0.95      0.94      3000
              U       0.90      0.90      0.90      3000
              V       0.93      0.92      0.92      3000
              W       0.95      0.95      0.95      3000
              X       0.92      0.92      0.92      3000
              Y       0.95      0.96      0.95      3000
              Z       0.96      0.96      0.96      3000
            del       0.99      0.98      0.99      3000
        nothing       1.00      1.00      1.00      3000
          space       0.98      0.99      0.99      3000

       accuracy                           0.96     87000
      macro avg       0.96      0.96      0.96     87000
   weighted avg       0.96      0.96      0.96     87000
```


Loss Curves
variable
train_losses0
0.6
0.5
0.4
0.3
0.2
0.1
Loss
2
4
6
8
10
Epoch

Loss Curve

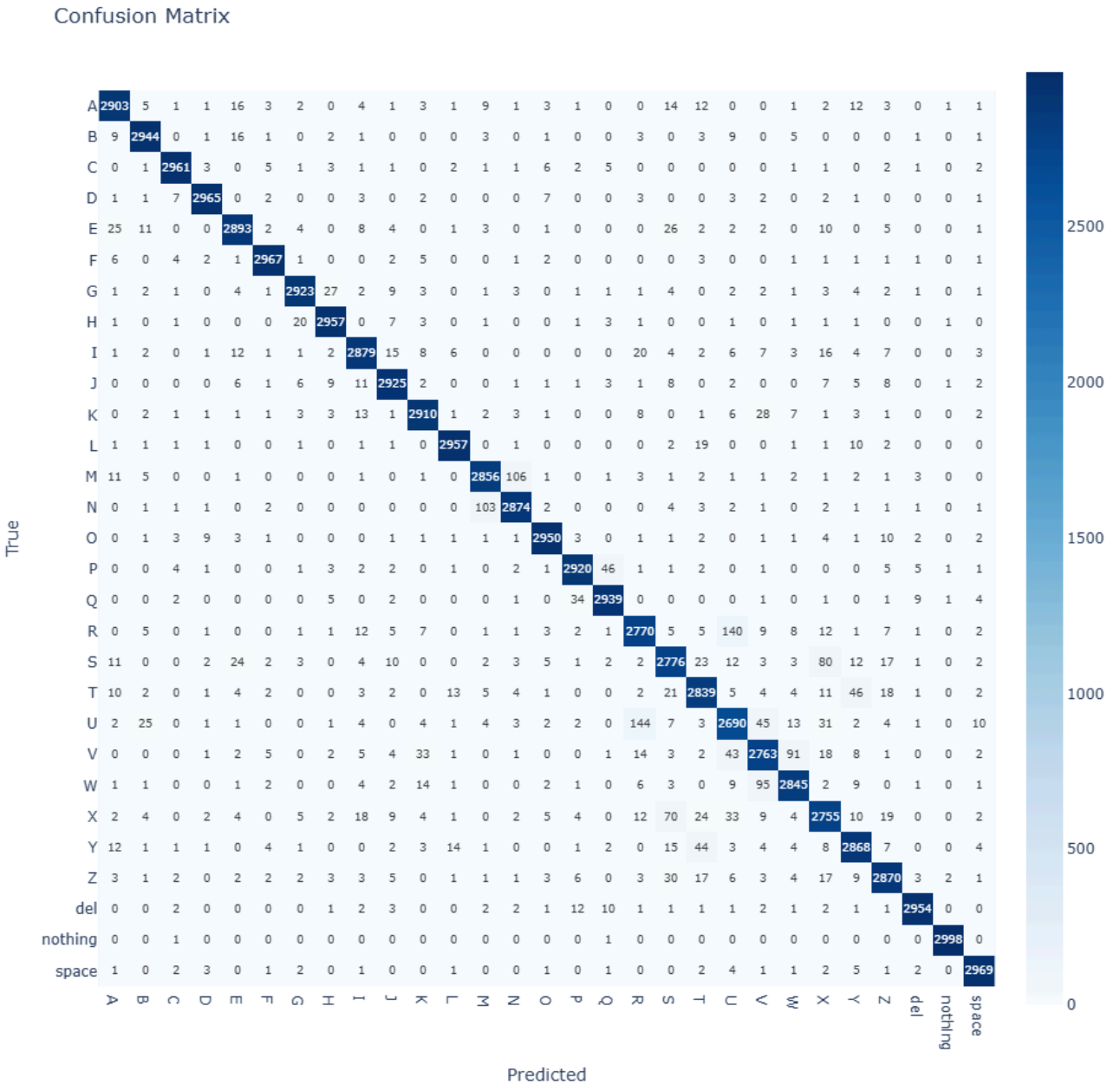


Accuracy Curve



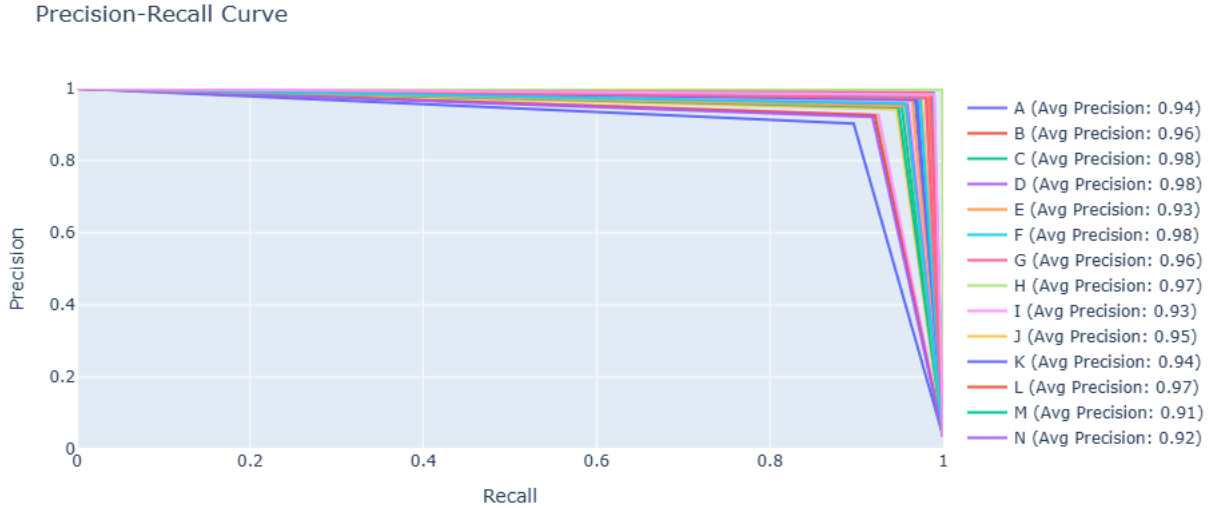
F1-Score Curve

- **confusion Matrices:** The confusion matrices for each class provide a detailed view of the model's performance, highlighting the number of true positives, false positives, true negatives, and false negatives for each class.



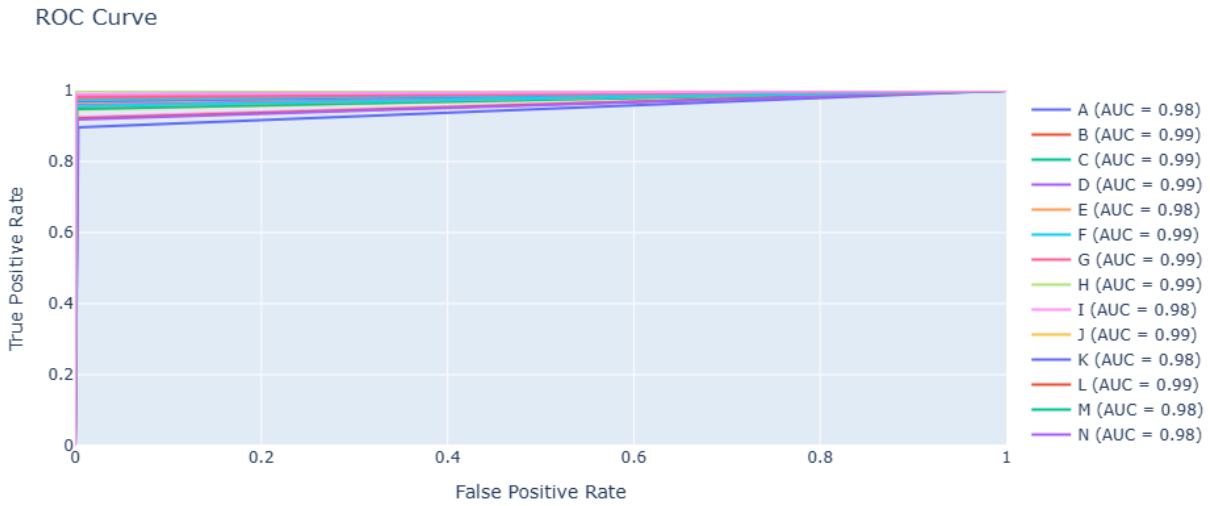
Confusion Matrix

- **Precision Recall Curve:** The precision-recall curve provides a visual representation of the model's precision and recall for each class, helping to identify the optimal threshold for classification.



Precision Recall Curve

- **ROC Curve:** The ROC curve provides a visual representation of the model's true positive rate (sensitivity) against the false positive rate (1 - specificity) for each class, helping to identify the optimal threshold for classification.



ROC Curve

3.7. Model and Artifact Logging (Section 7)

- **Dagshub Integration:** Dagshub is used for experiment tracking, authentication, and logging.
- **MLflow Tracking:** MLflow is utilized for logging experiment parameters, training metrics, and artifacts. The code obtains the tracking url from dagshub interface and set the tracking uri using `mlflow.set_tracking_uri`.
- **Experiment Logging:** After the connection to the MLflow is established, the experiments are logged using a function called `create_experiment`. This function also receives metrics, model, artifacts, run parameters, and tags as part of the log.

3.8. Testing Model on Test Images (Section 8)

- **Prediction Function:** A function `pred_and_plot_image` is defined to demonstrate how to load the trained model and classify the images. The classified image is shown with the ground truth, prediction and the probability score.
- **Image Sampling:** It loads three random test images from the test dataset and makes a prediction.

4. Results and Observations

- The model achieved high training performance as demonstrated in the training output and as verified by the plots of training metrics (accuracy, f1 score and loss) where the model metrics plateaued after a few initial epochs.
- The trained model was able to make predictions on three randomly selected test images with 100% confidence.
- The classification report showed a high overall performance of the model.
- The confusion matrix highlighted a good classification behavior with little to no confusions.
- The precision recall and ROC curves showed high performance scores per class and indicate high performance across categories.

5. Conclusion

The project successfully implemented an ASL image classification system by using transfer learning to re-train a pre-trained `EfficientNet-B0` model on custom data. The model achieved high performance and is able to correctly classify ASL images in all the categories. The notebook shows an implementation of a well-structured training

framework that leverages multi-gpu support and detailed experiment tracking. The next step is to deploy it on streamlit as a user application. Link to the streamlit app is here