Lab 8

Transfer Learning & Multitask Learning

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.models as models
import torchvision.transforms as transforms
from torch.utils.tensorboard import SummaryWriter
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader,Subset,Dataset,random_split
from PIL import Image
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
from collections import Counter
import numpy as np
import seaborn as sns
from tqdm import tqdm
import matplotlib.pyplot as plt
import os
from sklearn.metrics import classification_report, confusion_matrix

seed = 4912
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

[1] import library ที่จำเป็นใน lab นี้ โดย fix ค่า seed ไว้

```python
class MultiLanguageHandwrittenDataset(Dataset):
    def __init__(self, root_dirs, languages, transforms=None):
        self.root_dirs = root_dirs
        self.languages = languages
        self.transform = transforms
        self.samples = []

        for root_dir, language in zip(self.root_dirs, self.languages):
            for subdir in os.scandir(root_dir):
                if subdir.is_dir():
                    digit_label = int(subdir.name)  # digit label
                    for file in os.scandir(subdir.path):
                        if file.is_file():
                            self.samples.append((file.path, digit_label, self.languages.index(language)))

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        image_path, digit_label, language_label = self.samples[idx]
        image = Image.open(image_path).convert('L')  # Grayscale

        if self.transform:
            image = self.transform(image)

        return image, digit_label, language_label
```

[2] class MultiLanguageHandwrittenDatasetซ ทำหน้าที่โหลดและจัดการชุดข้อมูลของตัวลขที่เขียนด้วยมือ โดยประกอบด้วยตัวเลข (0-9) ในหลายภาษา (อังกฤษและไทย)

ฟังก์ชันภายในคลาส:

- **init(self, root_dirs, languages, transforms=None):**
  - root_dirs: list ของ path ของ folder แต่ละภาษา
  - languages: list ของภาษาที่อยู่ในแต่ละโฟลเดอร์หลัก (เป็นรายการของสตริง เช่น "Thai", "English")
  - transforms: ฟังก์ชันสำหรับปรับแต่งภาพ
  - ฟังก์ชันนี้จะทำการเก็บข้อมูลเส้นทางของภาพทั้งหมดไว้ในตัวแปร self.samples โดยวนลูปผ่านโฟลเดอร์หลัก และโฟลเดอร์ย่อย
    - โฟลเดอร์หลัก: สอดคล้องกับแต่ละภาษา
    - โฟลเดอร์ย่อย: สื่อถึงตัวเลข (ชื่อโฟลเดอร์ย่อยจะเป็นตัวเลข 0-9)
    - ไฟล์ภาพ: ไฟล์ภาพตัวเลขที่เขียนด้วยมือที่อยู่ในโฟลเดอร์ย่อย
  - ในการเก็บข้อมูล self.samples จะเป็น tuple ประกอบด้วย 3 ค่า
    - Path ไปยังไฟล์ภาพ (sttring)
    - เลขประจำตัวเลข (int)
    - ดัชนีของภาษาในรายการ languages (int)
- **len(self):**
  - ฟังก์ชันนี้จะ return จำนวนภาพทั้งหมดในชุดข้อมูล โดยการนับความยาวของ self.samples
- **getitem(self, idx):**
  - ฟังก์ชันนี้ดึงข้อมูลภาพ เลขประจำตัวเลข และภาษา จากรายการ self.samples ที่ตำแหน่ง idx
  - เปิดไฟล์ภาพด้วย Image.open และแปลงเป็นภาพขาวดำด้วย convert('L') **(Grayscale)**
  - ใช้ฟังก์ชันปรับแต่งภาพ self.transform
  - ส่งกลับค่าเป็น tuple ประกอบด้วย 3 ค่า
    - ภาพ (Tensor หรือ NumPy array)
    - เลขประจำตัวเลข (int)
    - ภาษา (int) - สอดคล้องกับดัชนีในรายการ languages

```python
# Function to visualize the batch of images
def show_batch(images, labels, languages):
    plt.figure(figsize=(10, 10))
    for i in range(len(images)):
        plt.subplot(4, 4, i + 1)  # Adjust grid size as necessary
        # Permute dimensions to (height, width, channels)
        img = images[i].permute(1, 2, 0).numpy()  # Convert to numpy array for plotting
        plt.imshow(img)  # Use the permuted image
        plt.title(f'Label: {labels[i]}, Language: {languages[i]}')
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

```
### START CODE HERE ###
# Define the relative paths
eng_dataset_reletive_path = 'data/eng-handwritten-dataset'
thai_dataset_reletive_path = 'data/thai-handwritten-dataset'

# Get the absolute paths
eng_dataset_absolute_path = os.path.abspath(eng_dataset_reletive_path)
thai_dataset_absolute_path = os.path.abspath(thai_dataset_reletive_path)

# Update root_dirs with the absolute paths
root_dirs = [eng_dataset_absolute_path, thai_dataset_absolute_path]

print(root_dirs)

languages = ['English', 'Thai']

# Define transformations
transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=3),  # Convert grayscale to 3 channels; VGG16 ซึ่งถูกออกแบบมาให้รับข้อมูลภาพที่มี 3 ช่อง (RGB) เป็นอินพุต
    transforms.Resize((224, 224)),                # Resize to 224x224 (required for VGG16)
    transforms.ToTensor(),        # Convert to tensor
])

# Initialize the dataset
dataset = MultiLanguageHandwrittenDataset(root_dirs, languages, transforms=transform)

# Split the dataset into training, validation, and test sets
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size])

# Create DataLoaders for each split
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

# Display the first batch
images, labels, languages = next(iter(train_loader))
show_batch(images, labels, languages)
### END CODE HERE ###
```

[4] **Path Definitions**:

- Defines the relative paths to two datasets: one for English and one for Thai handwritten digits.
- Converts these relative paths into absolute paths using os.path.abspath() and stores them in root_dirs.

Languages:

- Languages: เก็บชื่อภาษาอังกฤษและภาษาไทย

Transformations:

- **Transforms** the images to fit the VGG16 model requirements:
  - Converts the grayscale images to **3 channels** (RGB) because **VGG16 requires input with 3 channels**.
  - Resizes the images to **224x224 pixels**, which is the required input size for VGG16.
  - Converts the images to tensors.

Dataset Initialization:

- Initializes the MultiLanguageHandwrittenDataset using the absolute paths of the datasets and the defined transformations.

Dataset Splitting:

- Splits the dataset into **training (70%)**, **validation (15%)**, and **test (15%)** sets using random_split().

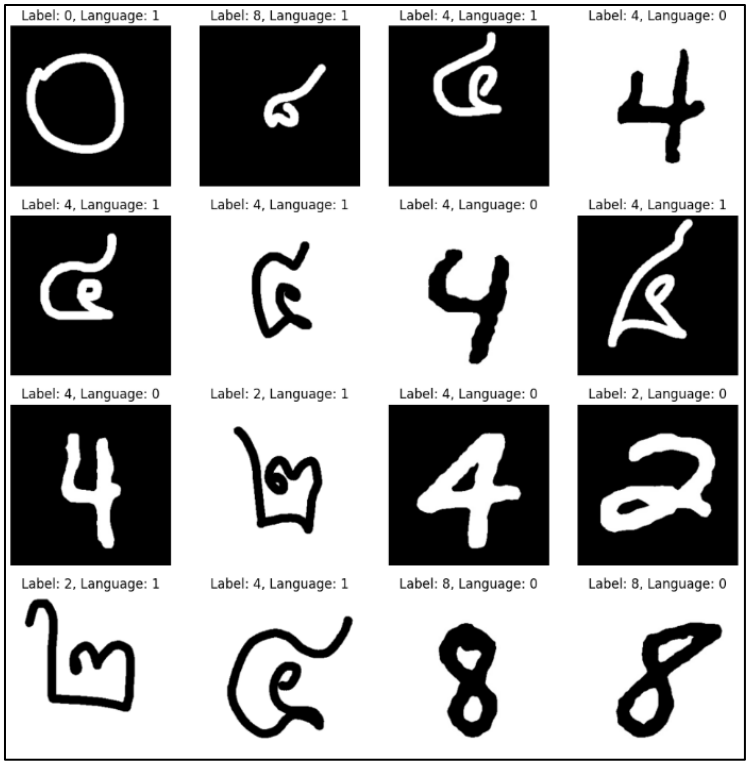
DataLoaders:

- Creates **DataLoaders** for training, validation, and test sets, with a batch size of 16. Shuffling is enabled for the training set but disabled for validation and test sets.

Visualizing a Batch:

- Retrieves the first batch of images, labels, and languages from the train_loader and uses show_batch() to display the batch.

ผลลัพธ์:



จะเห็นได้ว่า:

- **Label:** แทนตัวเลขแทน digit 0-9
- **Language:** แทนภาษาของตัวเลขนั้นๆ โดยที่ 0 แทน เลขภาษาอังกฤษ และ 1 แทนเลขภาษาไทย

```python
class customVGG16(nn.Module):
    def __init__(self,add_feat_dims=None,h_dims=None,num_classes=10,input_size=(1,28,28),trainable_layers_idx=None):
        super(customVGG16, self).__init__()

        # Load the pretrained VGG16 model
        self.vgg16 = models.vgg16(weights=models.VGG16_Weights.IMAGENET1K_V1)

        # Change the first convolutional layer to accept the specified number of input channels
        self.vgg16.features[0] = nn.Conv2d(3, 64, kernel_size=3, padding=1)

        # Freeze all convolutional layers by default
        for param in self.vgg16.features.parameters():
            param.requires_grad = False

        # Unfreeze specific layers if provided
        if trainable_layers_idx is not None:
            for idx in trainable_layers_idx:
                if idx < len(self.vgg16.features):  # Ensure the index is within bounds
                    for param in self.vgg16.features[idx].parameters():
                        param.requires_grad = True

        # Modify the VGG layers as needed for your input size
        # You can customize how many layers you keep and any additional pooling layers
        layers = list(self.vgg16.features.children())
        # Keep the first few layers and modify the pooling if needed
        self.vgg16.features = nn.Sequential(*layers[:10])  # Adjust as needed

        # Add an adaptive average pooling layer
        self.vgg16.features.add_module('adaptive_pool', nn.AdaptiveAvgPool2d((7, 7)))

        # Compute the input size for the fully connected layers
        in_features_fc = self._get_input_size_fc(input_size)
```

```python
        # Build the classifier
        if add_feat_dims is not None:
            fc_layers = []
            for dim in add_feat_dims:
                fc_layers.append(nn.Linear(in_features_fc, dim))
                fc_layers.append(nn.ReLU())
                fc_layers.append(nn.Dropout(0.5))
                in_features_fc = dim
            self.classifier = nn.Sequential(*fc_layers, nn.Linear(in_features_fc, num_classes))
        elif h_dims is not None:
            fc_layers = []
            for i, hdim in enumerate(h_dims):
                if i == 0:
                    fc_layers.append(nn.Linear(in_features_fc, hdim))
                else:
                    fc_layers.append(nn.Linear(h_dims[i - 1], hdim))
                fc_layers.append(nn.Dropout(0.4))
                fc_layers.append(nn.ReLU(inplace=True))
            fc_layers.append(nn.Linear(h_dims[-1], num_classes))
            self.classifier = nn.Sequential(*fc_layers)
        else:
            self.classifier = nn.Linear(in_features_fc, num_classes)

    def _get_input_size_fc(self, input_shape):
        """Compute the input size for the fully connected layer based on the input size."""
        with torch.no_grad():
            x = torch.zeros(1, *input_shape)
            x = self.vgg16.features(x)
            x = torch.flatten(x, 1)
            return x.size(1)

    def forward(self, x):
        """Define the forward pass."""
        # Forward pass through VGG16 features
        x = self.vgg16.features(x)
        x = torch.flatten(x, 1)  # Flatten starting from dimension 1

        # Forward pass through classifier
        x = self.classifier(x)
        return x
```

[5] This code defines a custom VGG16 model class customVGG16, which is based on a pretrained VGG16 model but allows for customization such as changing the input layer, freezing/unfreezing specific layers, and adding custom fully connected layers.

- **Initialization (__init__ method)**:
  - **VGG16 Pretrained Model**: Loads the pretrained VGG16 model using models.vgg16(weights=models.VGG16_Weights.IMAGENET1K_V1).
  - **Input Layer Modification**: Changes the first convolutional layer to accept 3 channels (RGB), which is needed to process the transformed grayscale images (now with 3 channels).
- **Freezing and Unfreezing Layers**:
  - **Default Freezing**: By default, all convolutional **layers are frozen by setting requires_grad = False**, preventing their weights from being updated during training. (แช่แข็งเลเยอร์ทั้งหมดของ VGG16 เพื่อไม่ให้ปรับน้ำหนักระหว่างฝึก)
  - **Conditional Unfreezing**: If **trainable_layers_idx is provided** (a list of layer indices), the specified **layers are unfrozen by setting requires_grad = True** for those layers.
    - **Explanation**: This allows fine-tuning specific layers of the VGG16 model while keeping others frozen, helping balance training efficiency and performance.
- **Layer Modification**:
  - Retains the first 10 layers of the VGG16 feature extractor, though this can be adjusted (self.vgg16.features = nn.Sequential(*layers[:10])).
    - self.vgg16.features = nn.Sequential(*layers[:10]) จะทำการเลือกและเก็บเฉพาะ layers ที่ต้องการจากโมเดล VGG16 โดยการใช้ nn.Sequential เพื่อสร้างโมดูลใหม่ที่ประกอบด้วย layers แรก 10 ชั้นจากโมเดล VGG16 ที่โหลดมา โดยที่:

> **VGG16 เป็นโมเดลโครงข่ายประสาทเทียมที่ถูกออกแบบมาสำหรับการจำแนกประเภทภาพ มีทั้งหมด 16 ชั้น ซึ่งประกอบด้วย:**
>
> 1. Convolutional Layers: ทำการตรวจจับฟีเจอร์ในภาพ
> 2. Pooling Layers: ลดขนาดของภาพเพื่อให้ประมวลผลได้เร็วขึ้น
> 3. Fully Connected Layers: ทำการจำแนกประเภทสุดท้าย

- **เลือกเก็บเฉพาะ 10 layer จากโมเดล 10 ชั้นแรก นั้นคือ**
  1. **Conv2d (3, 64)**
     - **การใช้งาน:** ใช้เพื่อทำการคอนโวลูชันกับภาพที่มี 3 ช่อง (RGB) เพื่อดึงฟีเจอร์เริ่มต้น เช่น ขอบและพื้นผิว
  2. **Conv2d (64, 64)**

- **การใช้งาน:** ทำการคอนโวลูชันเพื่อเรียนรู้ฟีเจอร์ที่ซับซ้อนยิ่งขึ้นจากฟีเจอร์ที่ได้จากเลเยอร์ก่อนหน้า

3. **MaxPool (2, 2)**
   - **การใช้งาน:** ทำการลดขนาดของภาพเพื่อให้ได้ฟีเจอร์ที่มีขนาดเล็กลงและลดจำนวนพารามิเตอร์ในโมเดล
4. **Conv2d (64, 128)**
   - **การใช้งาน:** เริ่มเรียนรู้ฟีเจอร์ที่ซับซ้อนมากขึ้นจากฟีเจอร์ที่ถูกเรียนรู้จากบล็อกก่อนหน้า
5. **Conv2d (128, 128)**
   - **การใช้งาน:** คอนโวลูชันเพิ่มเติมเพื่อเรียนรู้ฟีเจอร์ที่ลึกยิ่งขึ้น
6. **MaxPool (2, 2)**
   - **การใช้งาน:** ลดขนาดภาพอีกครั้งเพื่อให้การประมวลผลเป็นไปอย่างมีประสิทธิภาพ
7. **Conv2d (128, 256)**
   - **การใช้งาน:** เริ่มเข้าสู่ฟีเจอร์ที่มีความซับซ้อนมากขึ้นและมีการใช้จำนวนฟีเจอร์ที่มากขึ้น
8. **Conv2d (256, 256)**
   - **การใช้งาน:** เพิ่มจำนวนฟีเจอร์เพื่อให้การเรียนรู้เป็นไปอย่างมีประสิทธิภาพ
9. **Conv2d (256, 256)**
   - **การใช้งาน:** ทำให้ฟีเจอร์ที่ได้มีความละเอียดและซับซ้อนยิ่งขึ้น
10. **MaxPool (2, 2)**
    - **การใช้งาน:** การลดขนาดเพื่อสกัดฟีเจอร์สุดท้ายที่มีความสำคัญ

การเก็บเฉพาะ 10 เลเยอร์แรกจาก VGG16 จะช่วยให้สามารถทำการดึงฟีเจอร์พื้นฐานที่สำคัญจากภาพได้ โดยเฉพาะสำหรับภาพที่มีขนาดเล็ก เช่น 28x28 ซึ่งอาจไม่ต้องการฟีเจอร์ที่ซับซ้อนมากนักจากเลเยอร์ที่อยู่ลึกกว่านั้น นอกจากนี้ยังช่วยลดความซับซ้อนของโมเดลและเพิ่มความเร็วในการประมวลผลอีกด้วย

- Adds an **Adaptive Average Pooling layer** to reshape the output for the fully connected layers.

- **Fully Connected Layers (Classifier)**:
  - Computes the number of input features required for the fully connected layers based on the input image size (input_size).

- Builds the classifier with either additional feature dimensions (add_feat_dims) or hidden dimensions (h_dims) provided during initialization, with optional dropout and ReLU activation layers.

- **Helper Method (_get_input_size_fc)**:
  - คำนวณขนาดอินพุตสำหรับ layer: Classifier โดยส่งผ่านอินพุตผ่านเลเยอร์ VGG16 และแบนผลลัพธ์
  - This method calculates the input size of the fully connected layers based on the size of the output from the feature extractor part of VGG16.
- **Forward Method**:
  - Defines the forward pass of the model, first passing input through the modified VGG16 feature extractor, flattening the output, and then passing it through the custom classifier.

Detailed Explanation of Layer Freezing/Unfreezing

- **Freezing Layers**: By default, the pretrained VGG16 layers are frozen to retain their weights learned from the ImageNet dataset. This prevents the convolutional layers from updating their weights, reducing training time and avoiding overfitting when fine-tuning on a smaller dataset.
- **Unfreezing Layers**: Specific layers can be unfrozen based on the trainable_layers_idx. This allows those layers to be trained, while the rest of the VGG16 model remains frozen. Unfreezing certain layers can help the model adapt better to new datasets without retraining the entire network. For example, unfreezing the later layers of VGG16 might be useful when fine-tuning for complex tasks, as they contain higher-level features relevant to the task at hand.

```python
def train(model, opt, loss_fn, train_loader, val_loader, epochs=10, writer=None, checkpoint_path=None, device='cpu', task='digit'):
    print("🚀 Training on", device)
    model = model.to(device)

    for epoch in range(epochs):
        model.train()
        train_loss, train_correct = 0.0, 0
        total_train = 0

        # Training Loop
        train_bar = tqdm(train_loader, desc=f'🚀 Training Epoch [{epoch + 1}/{epochs}]', unit='batch')
        for images, labels, languages in train_bar:
            images = images.to(device)

            if task == 'digit':
                labels = labels.to(device)
                targets = labels
            elif task == 'language':
                languages = languages.to(device)
                targets = languages

            # Forward pass
            outputs = model(images)
            loss = loss_fn(outputs, targets)

            # Backward pass and optimization
            opt.zero_grad()
            loss.backward()
            opt.step()

            # Calculate training loss and accuracy
            train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            train_correct += (predicted == targets).sum().item()
            total_train += targets.size(0)

            # Update progress bar
            train_bar.set_postfix(loss=train_loss / (len(train_bar)))
        # Log training metrics to TensorBoard
        train_loss /= len(train_loader)
        train_accuracy = train_correct / total_train
        if writer:
            writer.add_scalar('Loss/train', train_loss, epoch)
            writer.add_scalar('Accuracy/train', train_accuracy, epoch)

        # Validation Loop
        model.eval()
        val_loss, val_correct = 0.0, 0
        total_val = 0
        y_true, y_pred = [], []

        val_bar = tqdm(val_loader, desc='📄 Validation', unit='batch')
        with torch.no_grad():
            for images, labels, languages in val_bar:
                images = images.to(device)

                if task == 'digit':
                    labels = labels.to(device)
                    targets = labels
                elif task == 'language':
                    languages = languages.to(device)
                    targets = languages

                # Forward pass
                outputs = model(images)
                loss = loss_fn(outputs, targets)

                # Calculate validation loss and accuracy
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_correct += (predicted == targets).sum().item()
                total_val += targets.size(0)

                # Collect true and predicted labels for evaluation
                y_true.extend(targets.cpu().numpy())
                y_pred.extend(predicted.cpu().numpy())
```

```
            # Update validation progress bar
            val_bar.set_postfix(loss=val_loss / (len(val_bar)))

    # Log validation metrics to TensorBoard
    val_loss /= len(val_loader)
    val_accuracy = val_correct / total_val
    if writer:
        writer.add_scalar('Loss/val', val_loss, epoch)
        writer.add_scalar('Accuracy/val', val_accuracy, epoch)

    print(f'Epoch [{epoch + 1}/{epochs}]\n'
        f'Loss:\t\t{train_loss:.4f},\tAccuracy:\t{train_accuracy:.4f}\n'
        f'Val Loss:\t{val_loss:.4f},\tVal Accuracy:\t{val_accuracy:.4f}')

    # Save checkpoint if path is provided
    if checkpoint_path:
        torch.save(model.state_dict(), checkpoint_path)
```

Key Details:

- **Task-Specific Training**: The function uses the task argument to decide whether the model is training for **digit classification (targets = labels) or language classification (targets = languages).**
- **Logging with TensorBoard**: If writer is provided, training and validation metrics are logged for visualization.
- **Checkpointing**: The model's state is saved at the end of each epoch, which allows resuming training from that point.

```
def evaluate_task(y_true, y_pred, task_name="Classification Task"):
    # Print the classification report
    print(f"{task_name} - Classification Report:")
    print(classification_report(y_true, y_pred, zero_division=1))

    # Compute the confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    # Plot confusion matrix using seaborn
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
    plt.title(f"{task_name} - Confusion Matrix")
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()
```

[7] This function evaluate_task evaluates the performance of a classification model by generating a classification report and confusion matrix.

```python
# Evaluate on the test set
def evaluate_on_test_set(model, test_loader, device, task="digit"):
    model.eval()
    test_correct = 0
    total_test = 0
    y_true, y_pred = [], []

    with torch.no_grad():
        for images, labels, languages in test_loader:
            images = images.to(device)

            if task == 'digit':
                labels = labels.to(device)
                targets = labels
            elif task == 'language':
                languages = languages.to(device)
                targets = languages

            # Get the model's outputs; handle the output being a tuple
            outputs = model(images)
            if isinstance(outputs, tuple):
                outputs = outputs[0]  # Assuming the first element is the relevant output

            _, predicted = torch.max(outputs, 1)

            test_correct += (predicted == targets).sum().item()
            total_test += targets.size(0)

            # Collect true and predicted labels for evaluation
            y_true.extend(targets.cpu().numpy())
            y_pred.extend(predicted.cpu().numpy())

    test_accuracy = test_correct / total_test
    print(f'Test Accuracy: {test_accuracy:.4f}')
    evaluate_task(y_true, y_pred, task_name=task)
```

Transfer learning for Digit classification task

Declare the `customVGG16` model with custom layers of your choice. Then, split the dataset into training, validation, and test sets, and proceed to train the model.

```python
# Define the parameters
# Initialize the model with specified configurations
trainable_layers_idx = [-1, -2, -3, -4, -5]  # Layers to unfreeze
model1 = customVGG16(add_feat_dims=[512], h_dims=[512, 256, 256], input_size=(3, 224, 224), trainable_layers_idx=trainable_layers_idx)

# Setup TensorBoard for logging
writer = SummaryWriter(log_dir='runs/custom_vgg16_digit_experiment')

# Define an optimizer and loss function
opt = optim.Adam(model1.parameters(), lr=0.001)  # Adam optimizer
loss_fn = nn.CrossEntropyLoss()  # Common loss for classification tasks

num_epochs = 100  # Set number of epochs
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train(model1, opt, loss_fn, train_loader, val_loader, epochs=num_epochs, writer=writer, device=device, task='digit')

writer.close()
```

[9] This code defines and initiates the training process for a custom VGG16 model (model1) for a digit classification task.

1. **Model Initialization**:

- **customVGG16 Model**:
  - **model1 = customVGG16(add_feat_dims=[512], h_dims=[512, 256, 256], input_size=(3, 224, 224), trainable_layers_idx=trainable_layers_idx)**
    - Creates an instance of the customVGG16 model with specific configurations:
      - **Trainable Layers**: The last 5 layers of the VGG16 model are unfrozen (trainable_layers_idx = [-1, -2, -3, -4, -5]), รายการของ index ที่แสดงถึงเลเยอร์ที่ต้องการให้โมเดลสามารถปรับเรียนรู้ได้ (unfreeze) โดยการใช้ค่าลบหมายถึงการนับจากด้านหลัง เช่น -1 หมายถึงเลเยอร์สุดท้าย
        - meaning these layers' weights will be updated during training. This allows the model to fine-tune the deeper convolutional layers.
      - **Additional Feature Dimensions**: Adds a fully connected layer with 512 units.
      - **Hidden Layers**: Includes hidden layers with dimensions [512, 256, 256], which adds complexity to the classification head.
      - **Input Size**: Specifies that the input images are of size (3, 224, 224), which aligns with VGG16's expected input dimensions.

2. **TensorBoard Setup**:
   - **writer = SummaryWriter(log_dir='runs/custom_vgg16_digit_experiment')**
     - Initializes a TensorBoard SummaryWriter for logging training metrics such as loss and accuracy under the directory runs/custom_vgg16_digit_experiment. These logs will be available for visualization in TensorBoard.
3. **Optimizer and Loss Function**:
   - **opt = optim.Adam(model1.parameters(), lr=0.001) # Adam optimizer**
   - **loss_fn = nn.CrossEntropyLoss() # Common loss for classification tasks**
     - **Optimizer**: Uses the **Adam** optimizer with a learning rate of 0.001. Adam is widely used due to its adaptive learning rates and ability to handle sparse gradients.
     - **Loss Function**: Defines the **CrossEntropyLoss**, which is the most common loss function for multi-class classification tasks.
4. **Training Configuration**:
   - **num_epochs = 100 # Set number of epochs**
   - **device = torch.device("cuda" if torch.cuda.is_available() else "cpu")**

- **Number of Epochs**: Sets the number of training epochs to 100.
- **Device Selection**: Automatically chooses to train on GPU if available, or falls back to CPU otherwise (cuda if available, else cpu).

5. **Training Execution**:
   - **train(model1, opt, loss_fn, train_loader, val_loader, epochs=num_epochs, writer=writer, device=device, task='digit')**
     - Calls the train function, which handles the entire training loop (described in a previous cell breakdown). The training will be **conducted on the digit classification task (task='digit')**.
6. **Closing TensorBoard Writer**:
   - **writer.close()**
     - After training is complete, the writer.close() method ensures that all logs are saved and the TensorBoard writer is properly closed.

Detailed Explanation of Layer Freezing/Unfreezing:

trainable_layers_idx = [-1, -2, -3, -4, -5]

- **these layers are and their function in VGG16:**
  - **-1:** This refers to the last layer in the feature extraction part, which is a max pooling layer. Max pooling reduces the spatial dimensions of the feature maps, retaining important features while discarding less significant ones.
  - **-2:** The second-to-last layer is a convolutional layer with 512 filters, each of size 3x3. This layer detects complex patterns from the input data.
  - **-3:** This is another convolutional layer with 512 filters, similar to -2. It extracts more high-level abstract features from the input.
  - **-4:** Another convolutional layer with 512 filters, continuing the pattern of detecting features from the previous layers.
  - **-5:** The fifth-to-last layer is also a convolutional layer with 512 filters, contributing to extracting deep features from the input image.

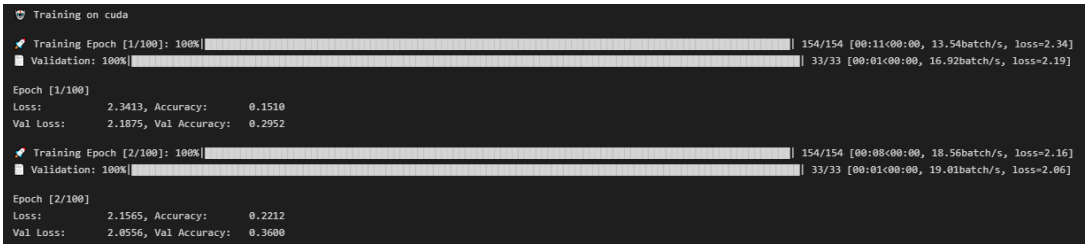
- **Unfreezing the Last 5 Layers**: The model has unfrozen the last 5 convolutional layers of the VGG16 feature extractor. These layers will now be trainable, meaning their weights can be adjusted during training to better

adapt to the new digit classification task. This is useful because these deeper layers learn more abstract features, and fine-tuning them helps the model specialize in the current task while leveraging the pretrained weights of the earlier layers for more general feature extraction.

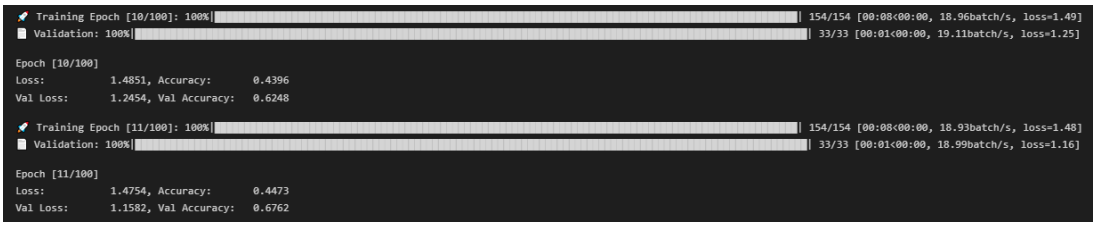
- **Freezing the Other Layers**: All other layers remain frozen, keeping their pretrained weights from the ImageNet dataset. This reduces the computational cost and prevents overfitting, as the frozen layers provide a solid base of general features.

ผลลัพธ์การ train:

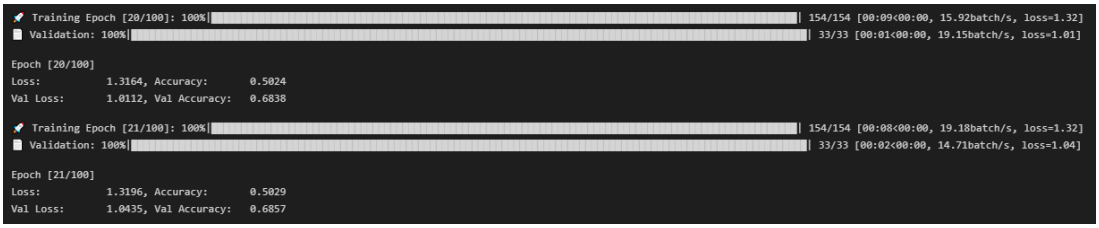
- รอบที่ 1-2



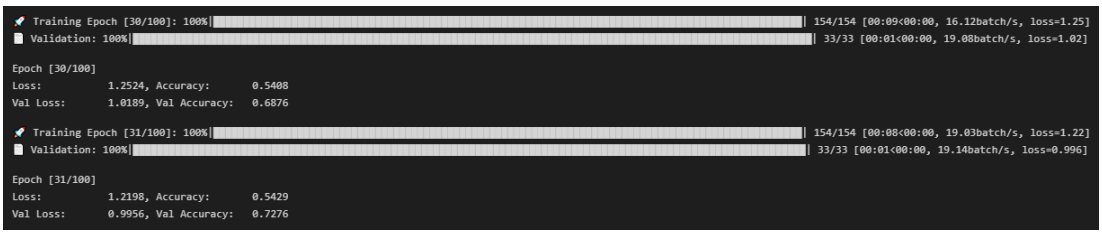
- รอบที่ 10-11



- รอบที่ 20-21



- รอบที่ 30-31



- รอบที่ 40-41

```
🚀 Training Epoch [40/100]: 100%| 154/154 [00:08<00:00, 19.11batch/s, loss=1.16]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.20batch/s, loss=0.858]

Epoch [40/100]
Loss:          1.1557, Accuracy:      0.5584
Val Loss:      0.8579, Val Accuracy:  0.7486

🚀 Training Epoch [41/100]: 100%| 154/154 [00:08<00:00, 19.06batch/s, loss=1.16]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.10batch/s, loss=1.03]

Epoch [41/100]
Loss:          1.1636, Accuracy:      0.5759
Val Loss:      1.0338, Val Accuracy:  0.7048
```

- รอบที่ 50-51

```
🚀 Training Epoch [50/100]: 100%| 154/154 [00:08<00:00, 19.06batch/s, loss=1.11]
📄 Validation: 100%| 33/33 [00:01<00:00, 18.90batch/s, loss=0.84]

Epoch [50/100]
Loss:          1.1078, Accuracy:      0.5849
Val Loss:      0.8400, Val Accuracy:  0.7562

🚀 Training Epoch [51/100]: 100%| 154/154 [00:10<00:00, 14.41batch/s, loss=1.1]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.08batch/s, loss=0.903]

Epoch [51/100]
Loss:          1.1015, Accuracy:      0.5792
Val Loss:      0.9026, Val Accuracy:  0.7448
```

- รอบที่ 60-61

```
🚀 Training Epoch [60/100]: 100%| 154/154 [00:08<00:00, 18.89batch/s, loss=1.07]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.09batch/s, loss=0.858]

Epoch [60/100]
Loss:          1.0664, Accuracy:      0.5935
Val Loss:      0.8576, Val Accuracy:  0.7276

🚀 Training Epoch [61/100]: 100%| 154/154 [00:08<00:00, 18.94batch/s, loss=1.03]
📄 Validation: 100%| 33/33 [00:01<00:00, 18.90batch/s, loss=0.888]

Epoch [61/100]
Loss:          1.0318, Accuracy:      0.6147
Val Loss:      0.8883, Val Accuracy:  0.7543
```

- รอบที่ 70-71

```
🚀 Training Epoch [70/100]: 100%| 154/154 [00:14<00:00, 10.56batch/s, loss=1.04]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.11batch/s, loss=0.828]

Epoch [70/100]
Loss:          1.0352, Accuracy:      0.5996
Val Loss:      0.8278, Val Accuracy:  0.7562

🚀 Training Epoch [71/100]: 100%| 154/154 [00:08<00:00, 19.11batch/s, loss=1.05]
📄 Validation: 100%| 33/33 [00:01<00:00, 18.50batch/s, loss=0.853]

Epoch [71/100]
Loss:          1.0526, Accuracy:      0.5894
Val Loss:      0.8531, Val Accuracy:  0.7752
```

- รอบที่ 80-81

```
🚀 Training Epoch [80/100]: 100%| 154/154 [00:08<00:00, 17.53batch/s, loss=1.03]
📄 Validation: 100%| 33/33 [00:01<00:00, 18.95batch/s, loss=0.859]

Epoch [80/100]
Loss:          1.0329, Accuracy:      0.5996
Val Loss:      0.8585, Val Accuracy:  0.7181

🚀 Training Epoch [81/100]: 100%| 154/154 [00:08<00:00, 18.90batch/s, loss=0.995]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.06batch/s, loss=0.812]

Epoch [81/100]
Loss:          0.9955, Accuracy:      0.6216
Val Loss:      0.8118, Val Accuracy:  0.7505
```

- รอบที่ 90-91

```
🚀 Training Epoch [90/100]: 100%| 154/154 [00:08<00:00, 18.98batch/s, loss=0.995]
📄 Validation: 100%| 33/33 [00:01<00:00, 18.92batch/s, loss=0.828]

Epoch [90/100]
Loss:          0.9953, Accuracy:      0.6237
Val Loss:      0.8278, Val Accuracy:  0.7467

🚀 Training Epoch [91/100]: 100%| 154/154 [00:09<00:00, 16.27batch/s, loss=0.964]
📄 Validation: 100%| 33/33 [00:03<00:00, 10.22batch/s, loss=0.823]

Epoch [91/100]
Loss:          0.9642, Accuracy:      0.6347
Val Loss:      0.8235, Val Accuracy:  0.7505
```

- รอบที่ 99-100

```
Training Epoch [99/100]: 100%| 154/154 [00:08<00:00, 18.95batch/s, loss=0.934]
Validation: 100%|          33/33 [00:01<00:00, 19.12batch/s, loss=0.822]

Epoch [99/100]
Loss:          0.9336, Accuracy:      0.6351
Val Loss:      0.8219, Val Accuracy:  0.7505

Training Epoch [100/100]: 100%| 154/154 [00:08<00:00, 19.08batch/s, loss=0.955]
Validation: 100%|          33/33 [00:01<00:00, 18.92batch/s, loss=0.828]

Epoch [100/100]
Loss:          0.9546, Accuracy:      0.6294
Val Loss:      0.8284, Val Accuracy:  0.7638
```

โดยพบว่า:

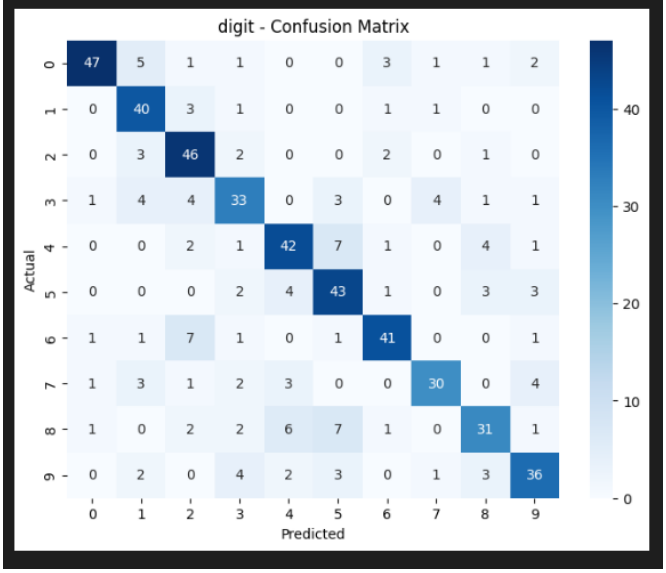
พบว่า **loss** มีการลดลงเรื่อย ๆ จนถึงรอบที่ **80** และหลังจากนั้นเริ่มแกว่งไปมา (บางครั้งเพิ่ม บางครั้งลด) ซึ่งอาจจะเกิดจากการที่ **training loss** เริ่มแกว่งหลังจากลดลงอย่างต่อเนื่องอาจเป็นสัญญาณว่าโมเดลเริ่มถึงจุดที่ไม่สามารถเรียนรู้ได้มากขึ้น หรือมีปัญหาเช่น **overfitting** หรือการตั้งค่า **learning rate** ที่สูงเกินไป **(เนื่องจากตอนแรกกำหนด รอบ=10 แต่สังเกตเห็นว่า** loss **ลดลงอย่างต่อเนื่องแต่ไม่ลดลงน้อยกว่า 1 สักที จึงเพิ่มรอบเรื่อยๆ จนเป็น 100 รอบครับ)**

```
### START CODE HERE ###
evaluate_on_test_set(model1, test_loader, device, task = 'digit')
### END CODE HERE ###
```

[10] This code evaluates the trained model1 on the test dataset for the digit classification task.

ผลลัพธ์:

```
Test Accuracy: 0.7410
digit - Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.77      0.84        61
           1       0.69      0.87      0.77        46
           2       0.70      0.85      0.77        54
           3       0.67      0.65      0.66        51
           4       0.74      0.72      0.73        58
           5       0.67      0.77      0.72        56
           6       0.82      0.77      0.80        53
           7       0.81      0.68      0.74        44
           8       0.70      0.61      0.65        51
           9       0.73      0.71      0.72        51

    accuracy                           0.74       525
   macro avg       0.75      0.74      0.74       525
weighted avg       0.75      0.74      0.74       525
```



- **Class ที่ model ทายผลผิดพลาดมากที่สุด:**
  - ข้อผิดพลาดที่เกิดขึ้นบ่อยที่สุดคือการทายผิดจาก class 3 เป็น class 1/2/7/5 ซึ่งมี precision และ recall ที่ต่ำกว่า class อื่นๆ การทายคลาส 3 มีค่า f1-score เพียง 0.66 ทำให้แสดงให้เห็นว่ามีความสับสน และ class 8 แม้จะมี precision และ recall ที่มากกว่า class 3 แต่พบว่ามี f1-score ที่น้อยที่สุดคือ 0.65 นั้นคือ model มักจะสับสนและทายจาก class 8 เป็น class 4/5
- **Class ที่ model ทายได้ดีที่สุด:**
  - class 0 มีค่าความแม่นยำสูงสุด (precision) อยู่ที่ 0.92 และค่า recall ที่ 0.77 ทำให้มีค่า f1-score สูงถึง 0.84 ซึ่งแสดงว่าโมเดลสามารถจำแนก class 0 ได้ดีที่สุดเมื่อเปรียบเทียบกับ class อื่นๆ

Transfer learning for Language classification task

Declare a NEW `customVGG16` model with custom layers of your choice.

```
# Define the parameters for the language classification model
trainable_layers_idx = [-1, -2, -3, -4, -5]  # Specify which layers to unfreeze
model2 = customVGG16(add_feat_dims=[512], h_dims=[256, 128], input_size=(3, 224, 224), trainable_layers_idx=trainable_layers_idx)

# Setup TensorBoard for logging
writer = SummaryWriter(log_dir='runs/custom_vgg16_language_experiment')

# Define an optimizer and loss function
opt = optim.Adam(model2.parameters(), lr=0.001)  # Adam optimizer
loss_fn = nn.CrossEntropyLoss()  # Common loss for classification tasks

# Number of epochs to train
num_epochs = 10  # Adjust as needed
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Training the model for language classification
train(model2, opt, loss_fn, train_loader, val_loader, epochs=num_epochs, writer=writer, device=device, task='language')

# Close the TensorBoard writer
writer.close()
```

[11] This code initializes and trains a second model (model2) for the **language classification task**, using a similar structure to the digit classification model (model1), but with specific modifications for the task.

1. **Model Initialization**:
   - **customVGG16 Model**:
     - Creates an instance of the customVGG16 model with specific configurations for language classification:
       - **Trainable Layers**: The last 5 layers of the VGG16 model are unfrozen (trainable_layers_idx = [-1, -2, -3, -4, -5]), similar to the previous model.

- **Additional Feature Dimensions**: Adds a fully connected layer with 512 units, followed by hidden layers with dimensions [256, 128].
- **Input Size**: Specifies the input image size as (3, 224, 224), which fits VGG16's input requirements.

2. **TensorBoard Setup**:
    - Initializes a **TensorBoard** SummaryWriter for logging training metrics under the directory runs/custom_vgg16_language_experiment.
3. **Optimizer and Loss Function**:
    - **Optimizer**: Uses the **Adam** optimizer with a learning rate of 0.001.
    - **Loss Function**: Uses **CrossEntropyLoss** for the language classification task, which is suitable for multi-class classification problems.
4. **Training Configuration**:
    - **Number of Epochs**: Trains the model for 10 epochs.
    - **Device Selection**: Chooses the training device (GPU if available, otherwise CPU).
5. **Training Execution**:
    - Calls the train function to train the model for the **language classification** task (task='language'), similar to how the digit classification was trained but now focused on distinguishing between languages.
6. **Closing TensorBoard Writer**:
    - Once training is complete, the **TensorBoard writer** is closed, ensuring that all logs are saved properly.

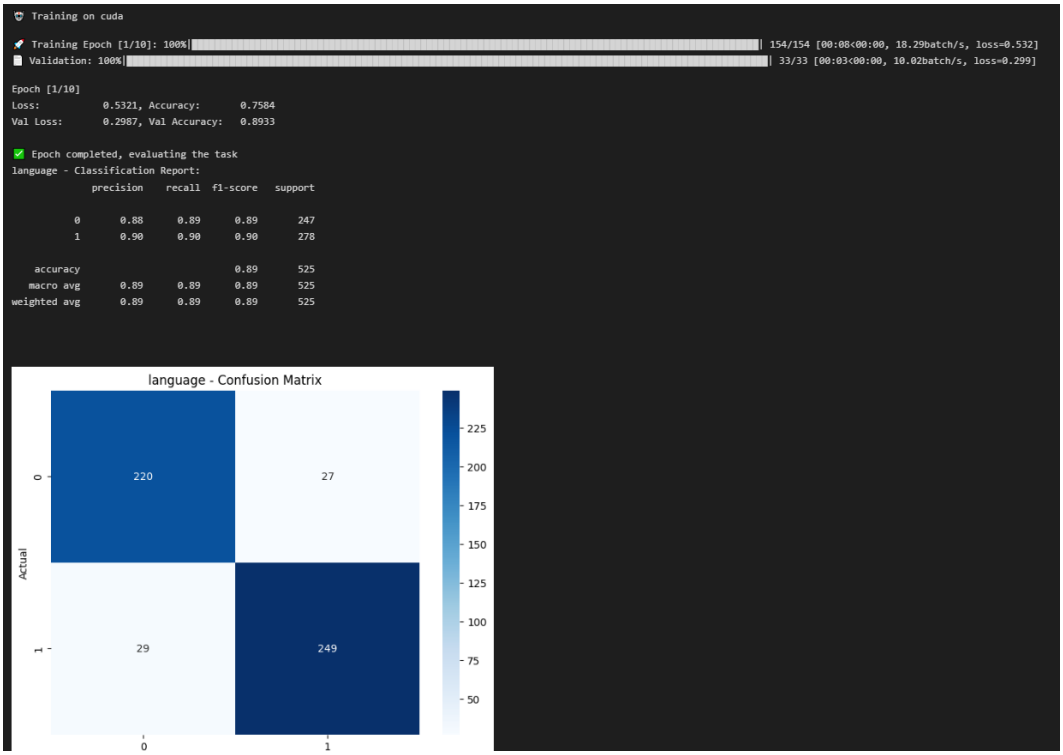
Detailed Explanation of Layer Freezing/Unfreezing for Language Classification:

- **Unfreezing the Last 5 Layers**: The last 5 layers of the VGG16 model are unfrozen, allowing their weights to be updated during training. These layers help the model adapt to the task of language classification by learning more specific features related to the language's visual characteristics.
- **Fully Connected Layers**: The model's classifier head for language classification includes additional feature dimensions and hidden layers with reduced complexity (512, 256, 128) compared to the digit classification

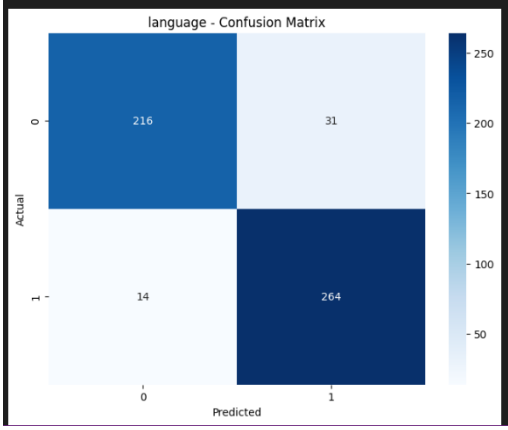
model. This helps tailor the model's capacity to the task of distinguishing between different languages, which might require different feature representations compared to digit classification.

ผลลัพธ์การ train:

- รอบที่ 1-2

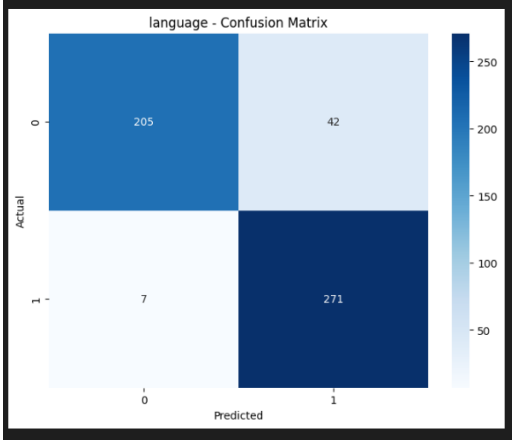


```
🚀 Training Epoch [2/10]: 100%| 154/154 [00:12<00:00, 12.24batch/s, loss=0.332]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.31batch/s, loss=0.24]

Epoch [2/10]
Loss:          0.3319, Accuracy:      0.8706
Val Loss:      0.2396, Val Accuracy:  0.9143

✅ Epoch completed, evaluating the task
language - Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.87      0.91       247
           1       0.89      0.95      0.92       278

    accuracy                           0.91       525
   macro avg       0.92      0.91      0.91       525
weighted avg       0.92      0.91      0.91       525
```

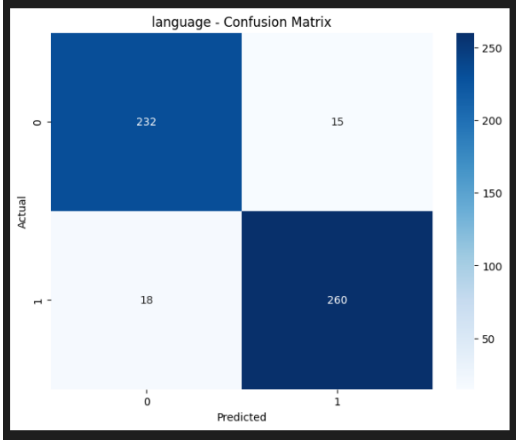


- รอบที่ 4-5-6

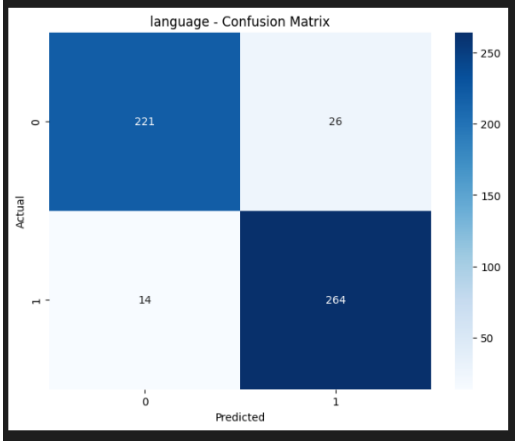
```
🚀 Training Epoch [4/10]: 100%| 154/154 [00:08<00:00, 19.23batch/s, loss=0.258]
📄 Validation: 100%| 33/33 [00:01<00:00, 19.32batch/s, loss=0.264]

Epoch [4/10]
Loss:          0.2577, Accuracy:      0.8955
Val Loss:      0.2643, Val Accuracy:  0.9067

✅ Epoch completed, evaluating the task
language - Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.83      0.89       247
           1       0.87      0.97      0.92       278

    accuracy                           0.91       525
   macro avg       0.92      0.90      0.91       525
weighted avg       0.91      0.91      0.91       525
```



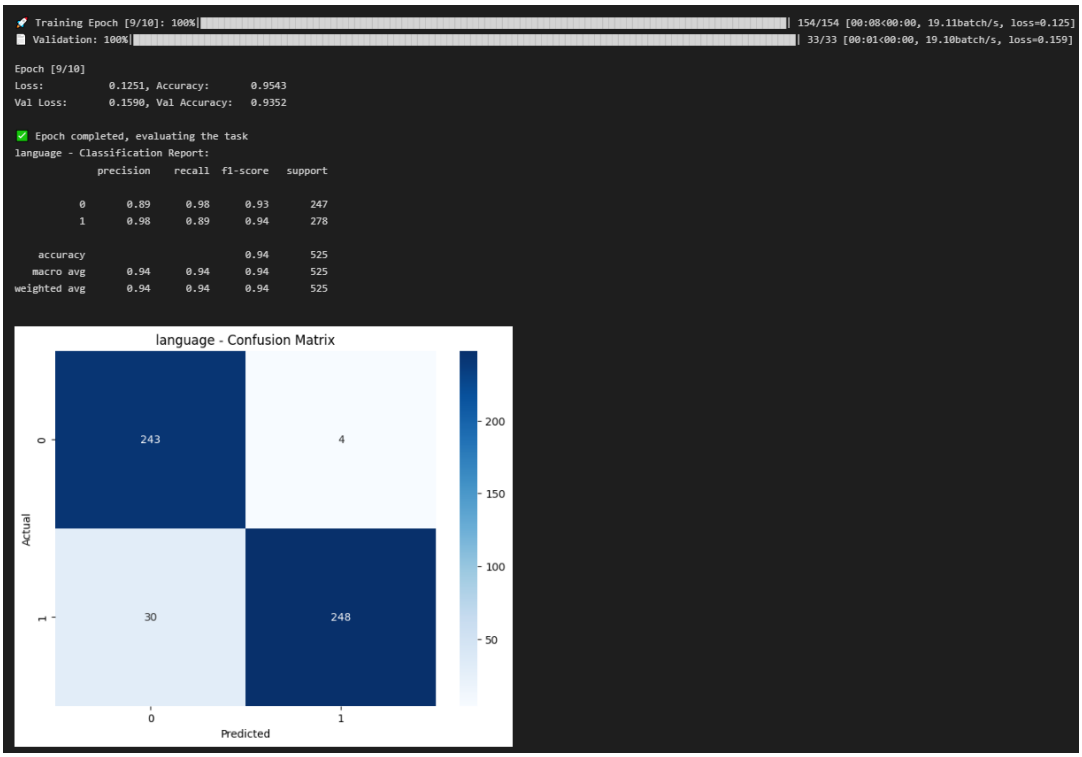
```
Training Epoch [5/10]: 100%| 154/154 [00:07<00:00, 19.34batch/s, loss=0.21]
Validation: 100%| 33/33 [00:01<00:00, 19.16batch/s, loss=0.145]

Epoch [5/10]
Loss:          0.2101, Accuracy:      0.9159
Val Loss:      0.1449, Val Accuracy:  0.9371

✅ Epoch completed, evaluating the task
language - Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.94      0.93       247
           1       0.95      0.94      0.94       278

    accuracy                           0.94       525
   macro avg       0.94      0.94      0.94       525
weighted avg       0.94      0.94      0.94       525
```

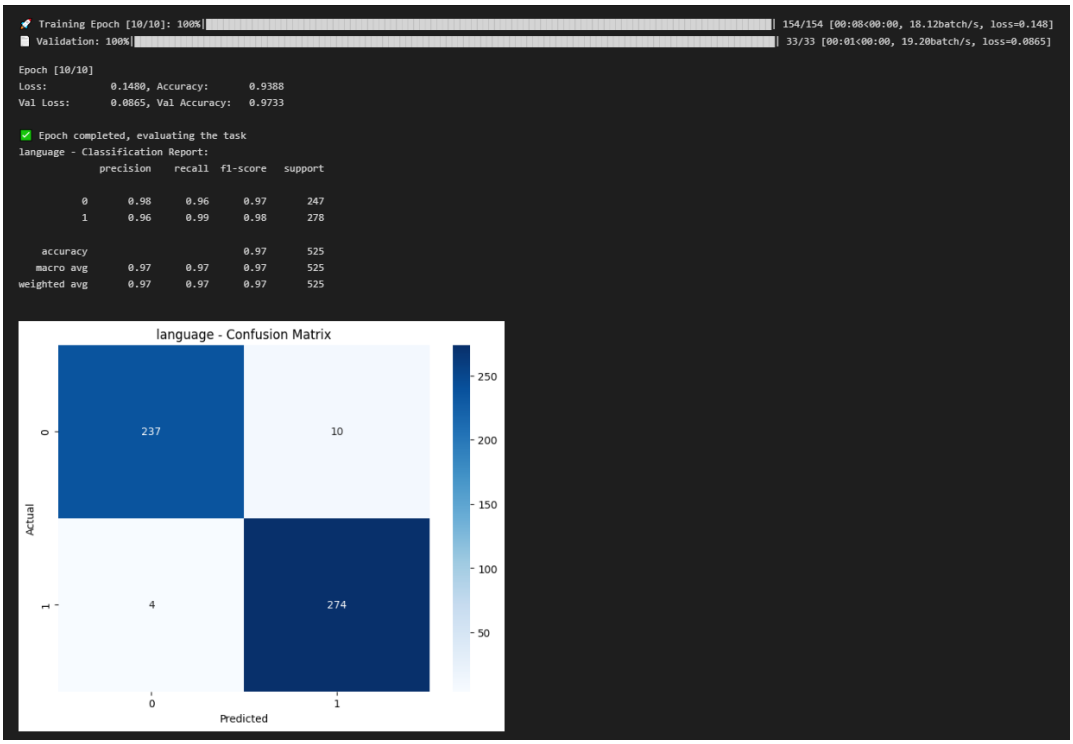



```
Training Epoch [6/10]: 100%| 154/154 [00:07<00:00, 19.35batch/s, loss=0.17]
Validation: 100%| 33/33 [00:01<00:00, 18.17batch/s, loss=0.17]

Epoch [6/10]
Loss:          0.1696, Accuracy:      0.9327
Val Loss:      0.1697, Val Accuracy:  0.9238

✅ Epoch completed, evaluating the task
language - Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.89      0.92       247
           1       0.91      0.95      0.93       278

    accuracy                           0.92       525
   macro avg       0.93      0.92      0.92       525
weighted avg       0.92      0.92      0.92       525
```



- รอบที่ 9-10





พบว่า:

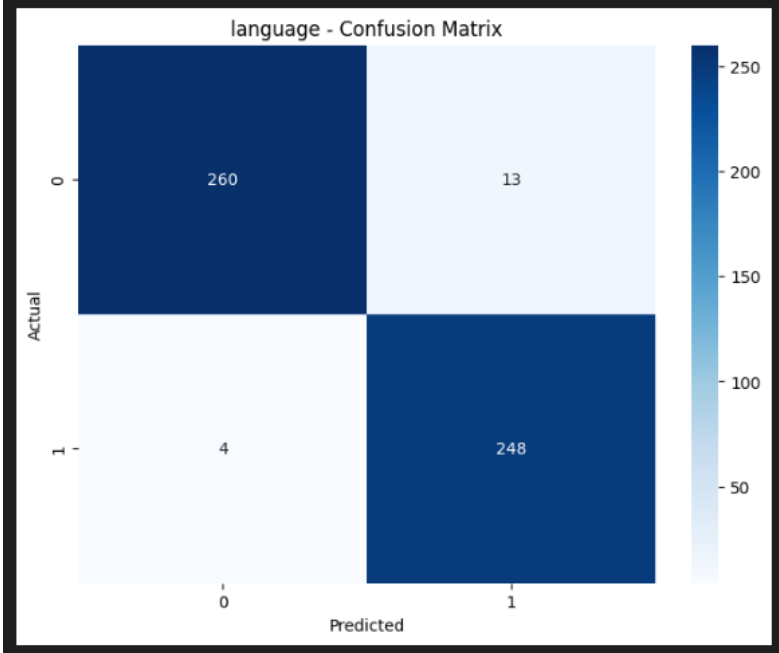
- การที่ train loss และ validate loss ลดลงอย่างต่อเนื่องจนหมดรอบเป็นสัญญาณที่ดีที่แสดงว่าโมเดลกำลังเรียนรู้ได้ดีและมีความสามารถในการทั่วไป ซึ่งหมายความว่าโมเดลมีแนวโน้มที่จะทำงานได้ดีในข้อมูลใหม่ที่ไม่เคยเห็นมาก่อน **(จึงฝึกเพียง 10 รอบ)**

```
### START CODE HERE ###
evaluate_on_test_set(model2, test_loader, device, task = 'language')
### END CODE HERE ###
```

[12] This code evaluates the trained model2 on the test dataset for the **language classification task**.

ผลลัพธ์:

```
Test Accuracy: 0.9676
language - Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.95      0.97       273
           1       0.95      0.98      0.97       252

    accuracy                           0.97       525
   macro avg       0.97      0.97      0.97       525
weighted avg       0.97      0.97      0.97       525
```



พบว่า:

- โมเดลมีความแม่นยำโดยรวมที่ 0.9676 และมีการทำงานที่ดีในการจำแนกทั้ง class 0 และ class 1

Multitask learning

```python
class customVGG16_multitask(nn.Module):
    def __init__(self, add_feat_dims=None, h_dims=None, input_size=(3, 224, 224), trainable_layers_idx=None):
        super(customVGG16_multitask, self).__init__()

        # Load pretrained VGG16 model
        self.vgg16 = models.vgg16(weights='DEFAULT')

        # The first layer already accepts 3 channels (RGB), no need to modify
        # Get the input size for the fully connected layer
        self.input_fc_size = self._get_input_size_fc(input_size)

        # Adding additional feature dimensions
        self.fc_layers = []
        if h_dims:
            in_features = self.input_fc_size
            for h_dim in h_dims:
                self.fc_layers.append(nn.Linear(in_features, h_dim))
                self.fc_layers.append(nn.ReLU())
                in_features = h_dim

        # Append the additional layers only if add_feat_dims is specified
        if add_feat_dims is not None:
            self.fc_layers.append(nn.Linear(in_features, add_feat_dims))
            self.fc_layers.append(nn.ReLU())
            in_features = add_feat_dims

        # Final output layers for multitask learning
        self.digit_fc = nn.Linear(in_features, 10)  # Assuming 10 digits (0-9)
        self.lang_fc = nn.Linear(in_features, len(languages))  # Number of languages based on input

        self.fc_layers = nn.Sequential(*self.fc_layers)

        # Freeze layers if specified
        if trainable_layers_idx is not None:
            for i, layer in enumerate(self.vgg16.features):
                if i not in trainable_layers_idx:
                    for param in layer.parameters():

    def _get_input_size_fc(self, input_shape):
        with torch.no_grad():
            x = torch.zeros(1, *input_shape)
            x = self.vgg16.features(x)  # Pass through feature extractor
            x = self.vgg16.avgpool(x)   # Average pooling
            x = torch.flatten(x, 1)      # Flatten for the fully connected layer
            return x.size(1)             # Return the size of the flattened output

    def forward(self, x):
        x = self.vgg16.features(x)  # Extract features
        x = self.vgg16.avgpool(x)   # Average pooling
        x = torch.flatten(x, 1)      # Flatten for the fully connected layer

        x = self.fc_layers(x)        # Pass through custom FC layers

        # Outputs for both tasks
        out1 = self.digit_fc(x)      # Digit classification
        out2 = self.lang_fc(x)       # Language classification

        return out1, out2
```

[13] This code defines a custom VGG16-based model, customVGG16_multitask, for **multitask learning**. The model is designed to handle both digit and language classification tasks simultaneously, sharing a common feature extractor (VGG16) but having separate fully connected (FC) output layers for each task.

1. **Model Initialization (__init__)**:
    - **Pretrained VGG16**: Loads a pretrained VGG16 model (self.vgg16 = models.vgg16(weights='DEFAULT')), which serves as the feature extractor for the multitask model.
    - **Input Size for FC Layers**: The function _get_input_size_fc() is used to compute the number of features output by the VGG16 layers after pooling, which will be passed to the custom fully connected layers.

- **Custom FC Layers**:
  - If h_dims (hidden layer dimensions) are provided, the model adds these fully connected layers with ReLU activation.
  - If add_feat_dims is specified, an additional fully connected layer is added before the final output layers.
  - The FC layers are stored in self.fc_layers, which is a sequential model of the added layers.
- **Multitask Output Layers**:
  - **digit_fc**: A final linear layer for digit classification with 10 output units (for digits 0-9).
  - **lang_fc**: A final linear layer for language classification, with the number of output units equal to the number of languages (based on the input).
- **Layer Freezing**:
  - If trainable_layers_idx is provided, the layers not in this list are frozen (i.e., their weights are not updated during training). This allows selective fine-tuning of the VGG16 layers.

2. **Helper Method (_get_input_size_fc)**:
   - This method calculates the input size for the fully connected layers by passing a dummy tensor through the VGG16 feature extractor and pooling layers to determine the output shape.
3. **Forward Pass (forward)**:
   - **Feature Extraction**: The input is passed through the VGG16 feature extraction layers and average pooling.
   - **Fully Connected Layers**: The output is flattened and passed through the custom fully connected layers.
   - **Multitask Outputs**: Two outputs are generated:
     - out1: Output from digit_fc, which is responsible for digit classification.
     - out2: Output from lang_fc, which is responsible for language classification.
   - These two outputs are returned together, enabling the model to perform multitask learning.

Detailed Explanation of Layer Freezing/Unfreezing:

- **Freezing Layers**: The code allows you to freeze specific VGG16 layers based on the trainable_layers_idx parameter. Layers not included in this index list will be frozen, meaning their weights will not be updated

during training. This helps retain the general features learned by those layers from the pretrained VGG16 on the ImageNet dataset.

- **Unfreezing Layers**: Layers that are included in the trainable_layers_idx list will remain trainable, allowing their weights to be updated during training. This fine-tuning can be useful for allowing the model to adapt to the specifics of the multitask problem (digit and language classification).

```python
def train_multi(model, opt, loss_fn, train_loader, val_loader, epochs=10, writer=None, checkpoint_path=None, device='cpu'):
    print("🚀 Training on", device)
    model = model.to(device)  # Move model to device

    for epoch in range(epochs):
        model.train()  # Set the model to training mode
        train_loss, train_correct_digit, train_correct_lang = 0.0, 0, 0
        total_train = 0

        # Training Loop with progress bar
        train_bar = tqdm(train_loader, desc=f'🚀 Training Epoch [{epoch + 1}/{epochs}]', unit='batch')
        for images, labels, languages in train_bar:
            images = images.to(device)
            labels = labels.to(device)
            languages = languages.to(device)

            # Forward pass
            outputs_digit, outputs_lang = model(images)
            loss_digit = loss_fn(outputs_digit, labels)
            loss_lang = loss_fn(outputs_lang, languages)
            loss = loss_digit + loss_lang

            # Backward pass and optimization
            opt.zero_grad()
            loss.backward()
            opt.step()

            # Calculate training loss and accuracy
            train_loss += loss.item()
            _, predicted_digit = torch.max(outputs_digit.data, 1)
            _, predicted_lang = torch.max(outputs_lang.data, 1)
            train_correct_digit += (predicted_digit == labels).sum().item()
            train_correct_lang += (predicted_lang == languages).sum().item()
            total_train += labels.size(0)

            # Update progress bar
            train_bar.set_postfix(loss=train_loss / (len(train_bar)))

        # Log training metrics to TensorBoard
        train_loss /= len(train_loader)
        train_accuracy_digit = train_correct_digit / total_train
        train_accuracy_lang = train_correct_lang / total_train
        if writer:
            writer.add_scalar('Loss/train', train_loss, epoch)
            writer.add_scalar('Accuracy/train_digit', train_accuracy_digit, epoch)
            writer.add_scalar('Accuracy/train_lang', train_accuracy_lang, epoch)
```

```python
    # Validation loop with progress bar
    model.eval()  # Set the model to evaluation mode
    val_loss, val_correct_digit, val_correct_lang = 0.0, 0, 0
    total_val = 0
    y_true_digit, y_pred_digit = [], []
    y_true_lang, y_pred_lang = [], []

    val_bar = tqdm(val_loader, desc='📄 Validation', unit='batch')
    with torch.no_grad():
        for images, labels, languages in val_bar:
            images = images.to(device)
            labels = labels.to(device)
            languages = languages.to(device)

            # Forward pass
            outputs_digit, outputs_lang = model(images)
            loss_digit = loss_fn(outputs_digit, labels)
            loss_lang = loss_fn(outputs_lang, languages)
            loss = loss_digit + loss_lang

            # Calculate validation loss and accuracy
            val_loss += loss.item()
            _, predicted_digit = torch.max(outputs_digit.data, 1)
            _, predicted_lang = torch.max(outputs_lang.data, 1)
            val_correct_digit += (predicted_digit == labels).sum().item()
            val_correct_lang += (predicted_lang == languages).sum().item()
            total_val += labels.size(0)

            # Collect true and predicted labels for evaluation
            y_true_digit.extend(labels.cpu().numpy())
            y_pred_digit.extend(predicted_digit.cpu().numpy())
            y_true_lang.extend(languages.cpu().numpy())
            y_pred_lang.extend(predicted_lang.cpu().numpy())

            # Update validation progress bar
            val_bar.set_postfix(loss=val_loss / (len(val_bar)))
```

```python
    # Log validation metrics to TensorBoard
    val_loss /= len(val_loader)
    val_accuracy_digit = val_correct_digit / total_val
    val_accuracy_lang = val_correct_lang / total_val
    if writer:
        writer.add_scalar('Loss/val', val_loss, epoch)
        writer.add_scalar('Accuracy/val_digit', val_accuracy_digit, epoch)
        writer.add_scalar('Accuracy/val_lang', val_accuracy_lang, epoch)

    print(f'Epoch [{epoch + 1}/{epochs}]\n'
          f'Train Loss:\t{train_loss:.4f},\t'
          f'Train Accuracy Digit:\t{train_accuracy_digit:.4f},\t'
          f'Train Accuracy Lang:\t{train_accuracy_lang:.4f},\n'
          f'Val Loss:\t{val_loss:.4f},\t'
          f'Val Accuracy Digit:\t{val_accuracy_digit:.4f},\t'
          f'Val Accuracy Lang:\t{val_accuracy_lang:.4f}')

    # Save checkpoint if path is provided
    if checkpoint_path:
        torch.save(model.state_dict(), checkpoint_path)

    print("\n✅ Epoch completed, evaluating the task")
    # Evaluate tasks if specified
    evaluate_task(y_true_digit, y_pred_digit, task_name="digit")
    evaluate_task(y_true_lang, y_pred_lang, task_name="language")
```

Key Points:

- **Multitask Learning**: The function handles two tasks in parallel: digit classification and language classification, with shared features but separate outputs.
- **Loss and Accuracy**: Both tasks are optimized simultaneously by computing and summing their respective losses (loss_digit + loss_lang). Accuracy for both tasks is tracked separately.

- **TensorBoard Logging**: If a TensorBoard writer is provided, the function logs loss and accuracy metrics for both training and validation, which can be visualized later.
- **Evaluation**: At the end of each epoch, classification reports and confusion matrices for both tasks are generated to evaluate the model's performance on the digit and language tasks.

```
### START CODE HERE ###
# Initialize the model for multitask learning
model = customVGG16_multitask(add_feat_dims=128, h_dims=[256, 128], trainable_layers_idx=[i for i in range(10)])


# Setup TensorBoard for logging
writer = SummaryWriter(log_dir='runs/custom_vgg16_multitask_experiment')

# Define an optimizer and loss function
opt = optim.Adam(model.parameters(), lr=0.001)  # Adam optimizer
loss_fn = nn.CrossEntropyLoss()  # Common loss for classification tasks

num_epochs = 10  # Set number of epochs
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Train the multitask model
train_multi(model, opt, loss_fn, train_loader, val_loader, epochs=num_epochs, writer=writer, device=device)
### END CODE HERE ###
```

1. **Model Initialization**:
    - **customVGG16_multitask Model**:
        - Creates an instance of the customVGG16_multitask model with the following configurations:
            - **Additional Feature Dimensions**: Adds a fully connected layer with 128 units before the multitask outputs.
            - **Hidden Layers**: Includes hidden layers with dimensions [256, 128] to process the features extracted from VGG16.
            - **Trainable Layers**: The first 10 layers of the VGG16 feature extractor are trainable (trainable_layers_idx=[i for i in range(10)]), meaning the weights of these layers will be updated during training. This helps the model adapt to the specific multitask problem.
2. **TensorBoard Setup**:
    - Initializes a **TensorBoard** SummaryWriter for logging training and validation metrics under the directory runs/custom_vgg16_multitask_experiment.
3. **Optimizer and Loss Function**:

- **Optimizer**: Uses the **Adam** optimizer with a learning rate of 0.001. This optimizer is commonly used because of its adaptive learning rate and efficiency in handling sparse gradients.
- **Loss Function**: Uses **CrossEntropyLoss**, which is appropriate for both digit and language classification tasks, as they are multi-class classification problems.

4. **Training Configuration**:
   - **Number of Epochs**: The model will be trained for 10 epochs.
   - **Device Selection**: The model is trained on a GPU ('cuda') if available; otherwise, it will use the CPU.
5. **Training Execution**:
   - The train_multi function is called to train the model on both tasks simultaneously (digit and language classification) over the specified number of epochs. This function:
     - Trains the model using the **training dataset** and computes the loss and accuracy for both tasks.
     - Evaluates the model on the **validation dataset** after each epoch, providing feedback on how well the model is performing on unseen data.
     - Logs all relevant metrics (loss, accuracy) for both tasks (digit and language) to **TensorBoard** for visualization.
     - Optionally saves the model's state if a checkpoint path is provided.

Detailed Explanation of Layer Freezing/Unfreezing for Multitask Learning:

- **Unfreezing Layers**: In this configuration, the first 10 layers of the VGG16 feature extractor are trainable (trainable_layers_idx=[i for i in range(10)]). This allows the model to fine-tune these layers to better adapt to the multitask nature of the problem (digit and language classification).
- **Freezing Layers**: The remaining layers of the VGG16 feature extractor remain frozen, which helps preserve the general features learned from the pretrained model on ImageNet while reducing computational cost and preventing overfitting.

นั้นคือ:

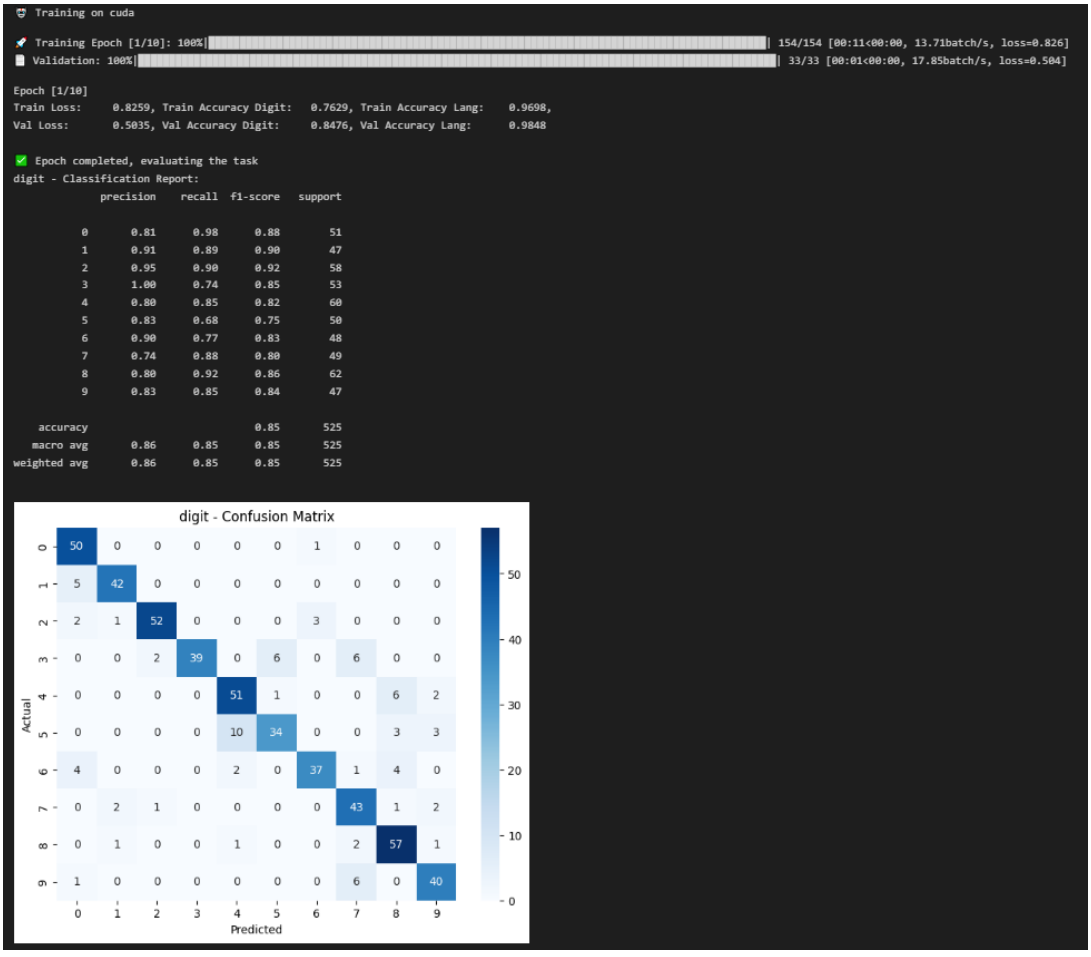
- **10 Layers แรกของ VGG16 (Shared Layers)**

- โมเดลเริ่มต้นด้วยการใช้ 10 ชั้นแรก ของ VGG16 ซึ่งรวมถึง convolutional layers และ max pooling layers
- ชั้นเหล่านี้ทำหน้าที่จับลักษณะพื้นฐาน (low-level features) และลักษณะระดับกลาง (mid-level features) เช่น ขอบ, มุม และรูปร่าง

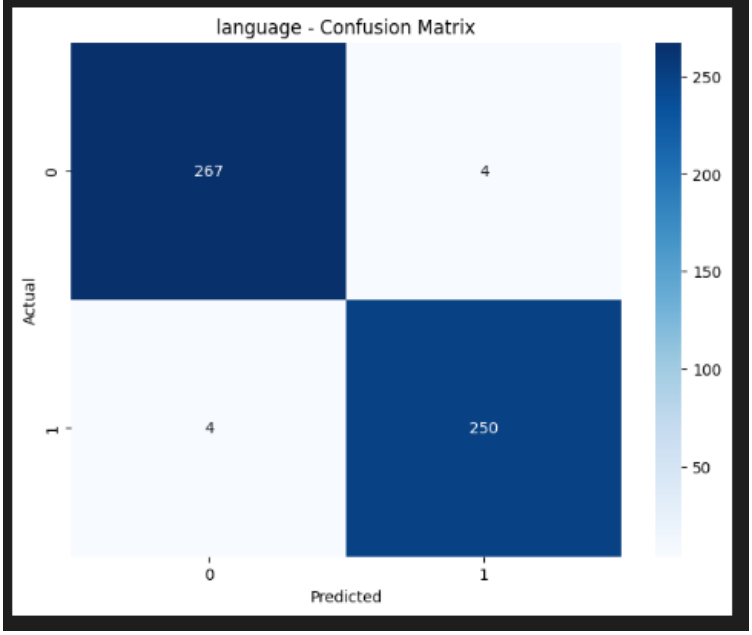
- Multi-Task Learning Layers **(Task-Specific Layers)**
  - หลังจากนั้นจะมี ชั้น fully connected (FC) ที่ทำหน้าที่เป็น multi-task learning layers ซึ่งมีการจัดการเพื่อให้สามารถทำการจำแนกประเภทได้ทั้ง 2 งานคือ:
    - Digit Classification: ชั้น output ที่ทำการจำแนกประเภทตัวเลข (0-9)
      - self.digit_fc = nn.Linear(in_features, 10)
    - Language Classification: ชั้น output ที่ทำการจำแนกประเภทภาษา
      - self.lang_fc = nn.Linear(in_features, len(languages))

ผลลัพธ์การ train:

- รอบที่ 1

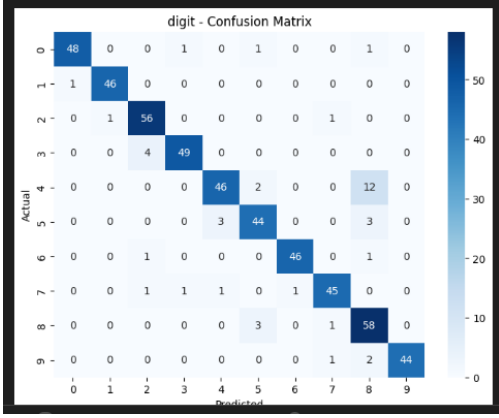


```
language - Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       271
           1       0.98      0.98      0.98       254

    accuracy                           0.98       525
   macro avg       0.98      0.98      0.98       525
weighted avg       0.98      0.98      0.98       525
```

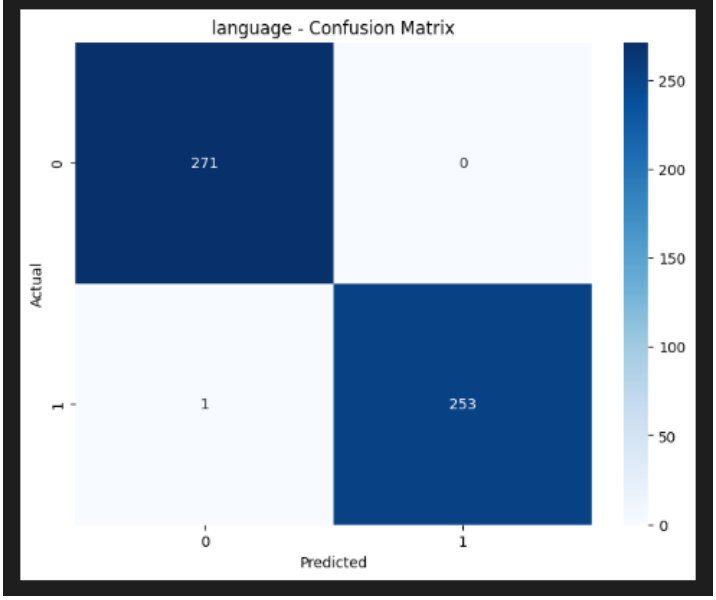


- รอบที่ 5

```
Training Epoch [5/10]: 100%| 154/154 [00:11<00:00, 13.43batch/s, loss=0.0678]
Validation: 100%| 33/33 [00:01<00:00, 17.36batch/s, loss=0.351]

Epoch [5/10]
Train Loss:     0.0678, Train Accuracy Digit:   0.9816, Train Accuracy Lang:    0.9996,
Val Loss:       0.3515, Val Accuracy Digit:     0.9181, Val Accuracy Lang:      0.9981

✅ Epoch completed, evaluating the task
digit - Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.94      0.96        51
           1       0.98      0.98      0.98        47
           2       0.90      0.97      0.93        58
           3       0.96      0.92      0.94        53
           4       0.92      0.77      0.84        60
           5       0.88      0.88      0.88        50
           6       0.98      0.96      0.97        48
           7       0.94      0.92      0.93        49
           8       0.75      0.94      0.83        62
           9       1.00      0.94      0.97        47

    accuracy                           0.92       525
   macro avg       0.93      0.92      0.92       525
weighted avg       0.92      0.92      0.92       525
```

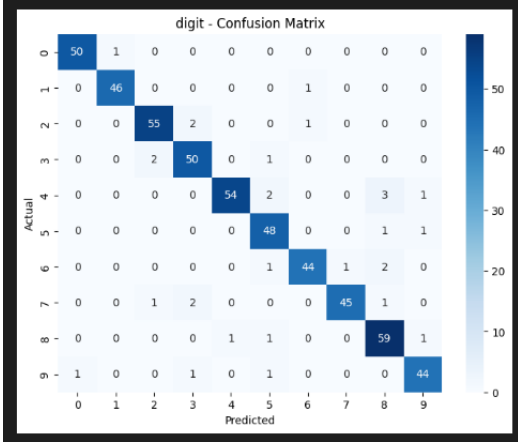


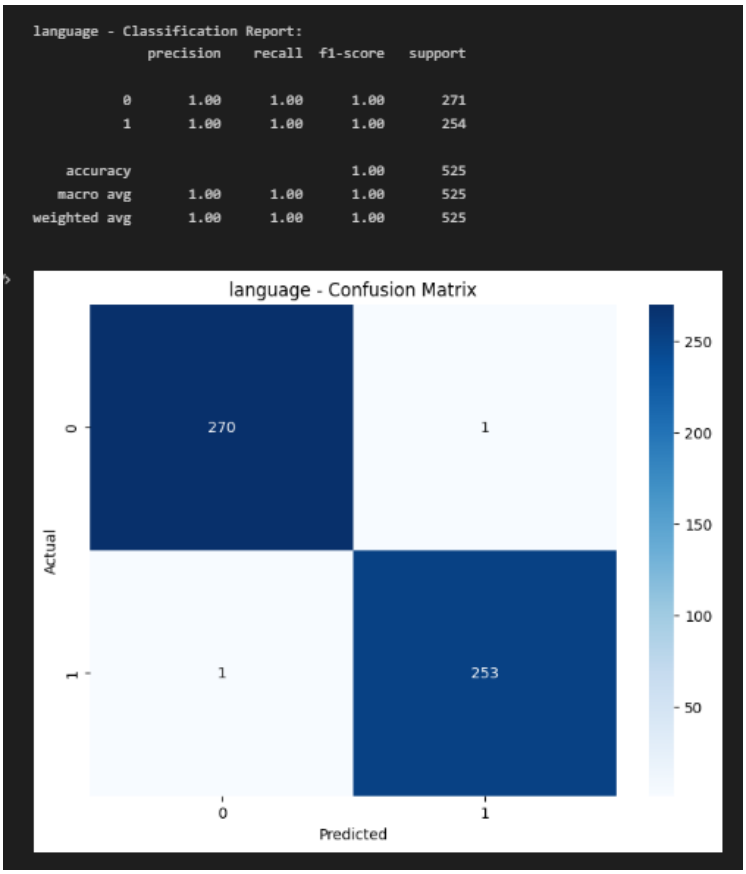
```
language - Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       271
           1       1.00      1.00      1.00       254

    accuracy                           1.00       525
   macro avg       1.00      1.00      1.00       525
weighted avg       1.00      1.00      1.00       525
```



- รอบที่ 10

```
🚀 Training Epoch [10/10]: 100%| 154/154 [00:11<00:00, 13.60batch/s, loss=0.000301]
📄 Validation: 100%| 33/33 [00:01<00:00, 18.08batch/s, loss=0.301]

Epoch [10/10]
Train Loss:     0.0003, Train Accuracy Digit:   1.0000, Train Accuracy Lang:   1.0000,
Val Loss:       0.3013, Val Accuracy Digit:     0.9429, Val Accuracy Lang:     0.9962

✅ Epoch completed, evaluating the task
digit - Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.98      0.98        51
           1       0.98      0.98      0.98        47
           2       0.95      0.95      0.95        58
           3       0.91      0.94      0.93        53
           4       0.98      0.90      0.94        60
           5       0.89      0.96      0.92        50
           6       0.96      0.92      0.94        48
           7       0.98      0.92      0.95        49
           8       0.89      0.95      0.92        62
           9       0.94      0.94      0.94        47

    accuracy                           0.94       525
   macro avg       0.95      0.94      0.94       525
weighted avg       0.94      0.94      0.94       525
```





พบว่า:

- กลุ่มผมเลือกฝึกโมเดล 10 รอบและพบว่า train loss มีค่าลดลงเรื่อยๆ อย่างมากในขณะที่ validate loss ก็ลดลงเช่นกันแต่ลดลงช้ากว่า train loss อย่างมาก คาดว่า:
  - **โมเดลเรียนรู้ได้ดี:** การที่ train loss ลดลงอย่างรวดเร็วแสดงว่าโมเดลกำลังเรียนรู้และปรับตัวได้ดีในชุดข้อมูลฝึกฝน **(จึงฝึกเพียง 10 รอบ)**
  - **ความเสี่ยงของการ Overfitting:** การที่ validate loss ลดลงช้ากว่า train loss อาจบ่งบอกว่าโมเดลกำลังเริ่มมีแนวโน้มที่จะ overfit ต่อข้อมูลฝึก ฝึกเรียนรู้รายละเอียดของข้อมูลฝึกมากเกินไป และอาจจะไม่สามารถทั่วไปกับข้อมูลใหม่ (validation set) ได้ดีเท่าที่ควร
  - หาก validate loss เริ่มไม่ลดลงหรือมีแนวโน้มที่จะแย่ลง อาจต้องพิจารณาปรับแต่ง hyperparameters เช่น ลด learning rate, หรือใช้ techniques อื่นๆ เช่น dropout เพื่อช่วยให้โมเดลมีความสามารถในการทั่วไปมากขึ้น

```python
def evaluate_multi_task_on_test_set(model, test_loader, device):
    model.eval()
    test_loss, test_correct_digit, test_correct_lang = 0.0, 0, 0
    total_test_digit, total_test_lang = 0, 0
    y_true_digit, y_pred_digit = [], []
    y_true_lang, y_pred_lang = [], []

    with torch.no_grad():
        for images, labels, languages in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            languages = languages.to(device)

            # Forward pass
            outputs_digit, outputs_lang = model(images)
            loss_digit = loss_fn(outputs_digit, labels)
            loss_lang = loss_fn(outputs_lang, languages)
            loss = loss_digit + loss_lang
            test_loss += loss.item()

            # Get predictions
            _, predicted_digit = torch.max(outputs_digit, 1)
            _, predicted_lang = torch.max(outputs_lang, 1)

            # Calculate correct predictions
            test_correct_digit += (predicted_digit == labels).sum().item()
            test_correct_lang += (predicted_lang == languages).sum().item()
            total_test_digit += labels.size(0)
            total_test_lang += languages.size(0)

            # Collect true and predicted labels for evaluation
            y_true_digit.extend(labels.cpu().numpy())
            y_pred_digit.extend(predicted_digit.cpu().numpy())
            y_true_lang.extend(languages.cpu().numpy())
            y_pred_lang.extend(predicted_lang.cpu().numpy())

    # Calculate average loss and accuracy
    test_loss /= len(test_loader)
    test_accuracy_digit = test_correct_digit / total_test_digit
    test_accuracy_lang = test_correct_lang / total_test_lang

    print(f'Test Loss: {test_loss:.4f}')
    print(f'Test Accuracy Digit: {test_accuracy_digit:.4f}')
    print(f'Test Accuracy Language: {test_accuracy_lang:.4f}')

    # Evaluate tasks for metrics
    evaluate_task(y_true_digit, y_pred_digit, task_name="digit")
    evaluate_task(y_true_lang, y_pred_lang, task_name="language")
```

[16] function ที่ใช้ในการประเมินโมเดลที่ทำงานหลายงาน (multi-task) โดยจะทำการทดสอบโมเดลบนชุดข้อมูลทดสอบ (test set) และคำนวณค่าความสูญเสีย (loss) และความถูกต้อง (accuracy) สำหรับทั้งงานที่เกี่ยวกับตัวเลข (digit) และภาษา (language)

1. การเตรียมการ:

- ตั้งค่า model.eval() เพื่อสลับโมเดลไปยังโหมดประเมิน
- กำหนดตัวแปรสำหรับเก็บผลลัพธ์:
    - test_loss: เก็บค่า loss เฉลี่ย
    - test_correct_digit, test_correct_lang: เก็บจำนวนการทำนายถูกสำหรับงานตัวเลขและภาษา
    - total_test_digit, total_test_lang: เก็บขนาดทั้งหมดของชุดข้อมูลทดสอบสำหรับงานตัวเลขและภาษา
    - y_true_digit, y_pred_digit: เก็บค่า true label และ predicted label สำหรับงานตัวเลข
    - y_true_lang, y_pred_lang: เก็บค่า true label และ predicted label สำหรับงานภาษา

2. Looping ผ่านข้อมูลทดสอบ:

- วนลูปผ่านข้อมูลทดสอบ (test_loader) โดยแยกข้อมูลภาพ (images), label งานตัวเลข (labels), และ label งานภาษา (languages)
- ย้ายข้อมูลไปยังอุปกรณ์คำนวณ (device)

3. ประเมินผลลัพธ์:

- ทำการ forward pass ผ่านโมเดล (outputs_digit, outputs_lang)
- คำนวณ loss สำหรับแต่ละงาน (loss_digit, loss_lang) โดยใช้ฟังก์ชัน loss_fn
- คำนวณ loss รวม (loss)
- เก็บผลรวมของ loss
- หาค่า predicted label สำหรับแต่ละงาน (predicted_digit, predicted_lang)
- คำนวณจำนวนการทำนายถูกสำหรับแต่ละงาน (test_correct_digit, test_correct_lang)
- เก็บจำนวนข้อมูลทั้งหมดสำหรับแต่ละงาน (total_test_digit, total_test_lang)
- เก็บค่า true label และ predicted label สำหรับการประเมินเพิ่มเติม (y_true_digit, y_pred_digit, y_true_lang, y_pred_lang)

4. คำนวณค่าเฉลี่ยและแสดงผล:

- คำนวณค่าเฉลี่ยของ loss (test_loss)
- คำนวณความแม่นยำสำหรับแต่ละงาน (test_accuracy_digit, test_accuracy_lang)
- แสดงผลลัพธ์:
    - test loss
    - test accuracy สำหรับงานตัวเลข
    - test accuracy สำหรับงานภาษา

5. ประเมินผลลัพธ์เพิ่มเติม:

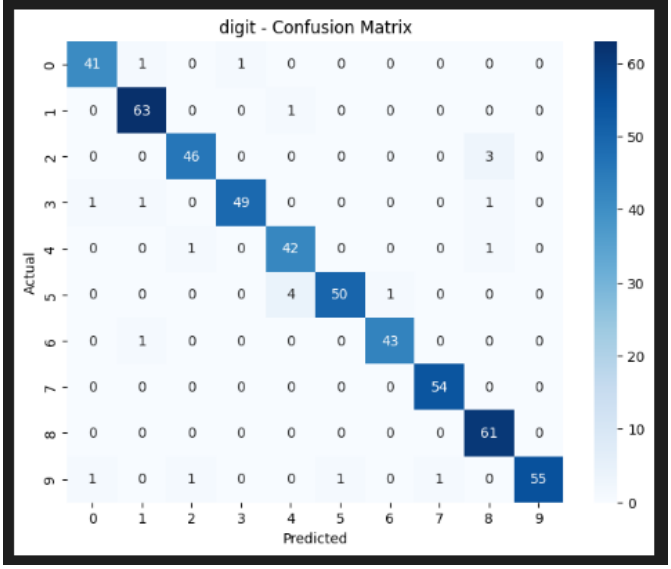
- ใช้ฟังก์ชัน evaluate_task ที่กล่าวไปใน block ที่ [7]

```
### START CODE HERE ###
evaluate_multi_task_on_test_set(model, test_loader, device)
### END CODE HERE ###
```

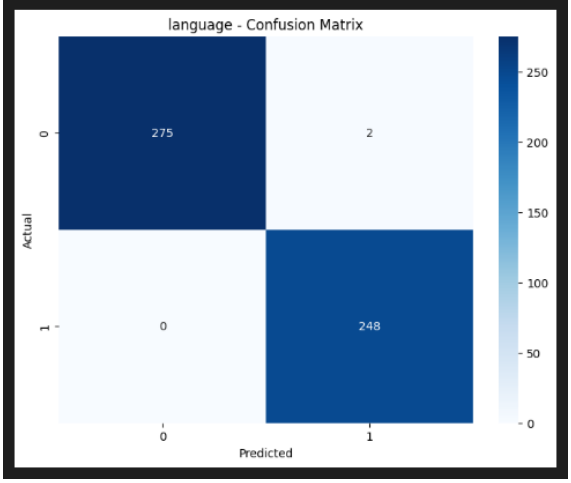
[17] This code evaluates the trained multitask model on the test dataset for both the **digit classification task** and the **language classification task**.

ผลลัพธ์:

```
Test Loss: 0.2228
Test Accuracy Digit: 0.9600
Test Accuracy Language: 0.9962
digit - Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.95      0.95        43
           1       0.95      0.98      0.97        64
           2       0.96      0.94      0.95        49
           3       0.98      0.94      0.96        52
           4       0.89      0.95      0.92        44
           5       0.98      0.91      0.94        55
           6       0.98      0.98      0.98        44
           7       0.98      1.00      0.99        54
           8       0.92      1.00      0.96        61
           9       1.00      0.93      0.96        59

    accuracy                           0.96       525
   macro avg       0.96      0.96      0.96       525
weighted avg       0.96      0.96      0.96       525
```




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

           0       1.00      0.99      1.00       277
           1       0.99      1.00      1.00       248

    accuracy                           1.00       525
   macro avg       1.00      1.00      1.00       525
weighted avg       1.00      1.00      1.00       525
```



พบว่า:

- โมเดลมีประสิทธิภาพสูงในการจำแนกประเภทตัวเลข โดยเฉพาะสำหรับคลาสต่าง ๆ โดยมีความแม่นยำและ F1-Score สูงสำหรับส่วนใหญ่
    - ค่าที่ต่ำสุดใน Precision คือ 0.89 และมี F1-Score = 092 สำหรับตัวเลข 4 ซึ่งอาจแสดงให้เห็นว่ามีการจำแนกประเภทที่อาจจะต้องปรับปรุงในกรณีนี้
- โมเดลทำงานได้อย่างยอดเยี่ยมในการจำแนกประเภทภาษา โดยมีความแม่นยำและ F1-Score สูงสำหรับทั้งสองคลาส (0(ภาษาอังกฤษ) และ 1(ภาษาไทย))
    - ค่าต่าง ๆ อยู่ที่ 1.00 หรือใกล้เคียงกับ 1 หมายถึงว่าโมเดลมีความสามารถในการจำแนกประเภทได้อย่างแม่นยำมาก และไม่มีปัญหาในการจำแนกประเภทภาษาในชุดทดสอบนี้