Final Project - Code

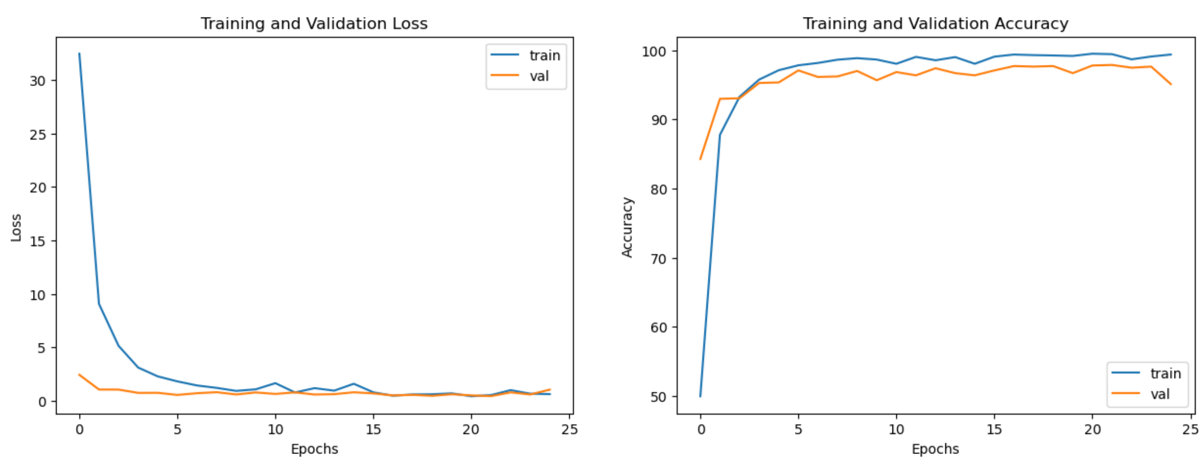
This notebook serves as the final code utilized for the ASL classification project for EEL5840.
Team - FML_Party
Members - Darian Jennings and Ashley Hart

Model used - VGG19

- You should expect the test dataset to have the same format as the training data: $270,000 \times M$ `numpy` array, where M is the number of test samples.
- This means that *any* pre-processing applied in the training data should also be applied in the test data.

In [2]:
```
%run training.py
```

```
---TRAINING COMPLETE---
Unknown indices:  []
Best model info:  {'epoch': 21, 'train_loss': 0.021766129745325696, 'train_accuracy': 99.42470389170897, 'val_loss': 0.08525952994823456, 'val_accuracy': 97.86729857819905}
```



In [7]:
```
import os
import gc
import numpy as np
import matplotlib.pyplot as plt
from joblib import dump
from PIL import Image
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset, random_split
from torchvision.transforms import AutoAugment, AutoAugmentPolicy
from torchvision.transforms import transforms as T
import torch.optim as optim
from torchvision.models import vgg19
from torchvision.models import VGG19_Weights
from sklearn.metrics import recall_score, f1_score, precision_score
from tqdm import tqdm

data_train = np.load('data_train.npy')
labels_train = np.load('labels_train.npy')

data_train.shape, labels_train.shape
```

Out[7]:
```
((270000, 8443), (8443,))
```

In [8]:
```
# Standard IMAGENET vals sourced from google
imagenet_means = [0.485, 0.456, 0.406]
imagenet_stds = [0.229, 0.224, 0.225]

# Define AutoAugment transform
# - automatically augments data based on a given auto-augmentation policy
augmenter = AutoAugment(policy=AutoAugmentPolicy.IMAGENET)

# Resize, augment,...
# NOTE - In PyTorch, T.ToTensor() is a transformation that converts a PIL
# tensor and scales the values to the range [0.0, 1.0]
# NOTE - Normalize expects Tensor - so convert to tensor then normalize

# create a Tensor dataset (performs transformations AND augmentation)
class TensorDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return self.data.shape[1]

    def __getitem__(self, idx):
        image = self.data[:, idx].reshape(300, 300, 3)
        image = Image.fromarray(image)
        if self.transform:
            image = self.transform(image)
        label = self.labels[idx]
        return image, label

preprocess = T.Compose([
    augmenter,
    T.Resize((224, 224)),
    T.ToTensor(),
    T.Normalize(imagenet_means, imagenet_stds)
])

dataset = TensorDataset(data_train, labels_train, transform=preprocess)
print("Created Tensor Dataset")
```

```
Created Tensor Dataset
```

In [10]:
```
# Create split sizes for training-validation-test, ---- use 70-15-15 rule
train_size = int(0.7 * len(dataset))
temp_size = len(dataset) - train_size
val_size = int(0.5 * temp_size)
test_size = temp_size - val_size

# Split dataset into training-validation-test using sizes (random_split)
train_dataset, temp_dataset = random_split(dataset, [train_size, temp_si
val_dataset, test_dataset = random_split(temp_dataset, [val_size, test_s

# Use DataLoader - load data into model - flexible for memory constraint
train_dataflow = DataLoader(train_dataset, batch_size=256, shuffle=True)
val_dataflow = DataLoader(val_dataset, batch_size=256)
test_dataflow = DataLoader(test_dataset, batch_size=256)
print("Created Dataflows")
```

```
Created Dataflows
```

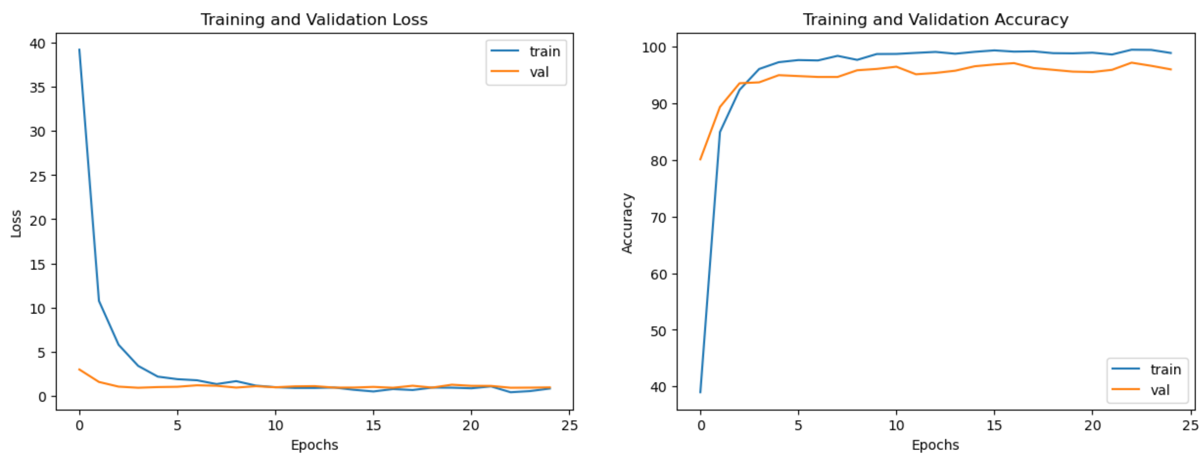
In [11]:
```
# Create instance of pre-trained VGG19 model
def pretrainedVGG19(num_classes):
    model = vgg19(weights=VGG19_Weights.DEFAULT)          # pre-trained
    num_ftrs = model.classifier[6].in_features            # extract n_f
    model.classifier[6] = nn.Linear(num_ftrs, num_classes) # n_output_fea
    return model

# Define device to utilize and num_of_classes for model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Call model with give number of classes
nclasses = 9
model = pretrainedVGG19(nclasses)
model.to(device)

# Define loss, add weight decay to optimizer (standard is 1e_05) - L2 pe
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001, weight_decay=1e-05

# Setup save path
save_path = '/blue/eel5840/darian.jennings/final_proj/'
os.makedirs(save_path, exist_ok=True)

# Track epoch with highest accuracy for epochs 1-N
highest_val_accuracy = 0

# Track which indices were marked as the unknown class - should be 0 at 
# Store vals for train, loss, and accuracies respectively
unknown_indices = []
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

# Initialize dictionary to store information about best model
best_model_info = {
    'epoch': None,
    'train_loss': None,
    'train_accuracy': None,
    'val_loss': None,
    'val_accuracy': None,
}

# Collect garbage
collected = gc.collect()

num_epochs = 25
for epoch in range(num_epochs):
    model.train()
    # reset for each epoch
    train_loss, train_correct, train_total = 0.0, 0, 0
    for images, labels in tqdm(train_dataflow, desc=f"Epoch {epoch+1}/{nu
        images, labels = images.to(device, dtype=torch.float), labels.to

        # raw output scores
        outputs = model(images)
        # calculate the probabilities
```

```
            probabilities = torch.nn.functional.softmax(outputs, dim=1)
            #max_prob, _ = torch.max(probabilities, dim=1)
            #print(max_prob)

            # check if all probabilities are below 0.1
            unknown = (probabilities < 0.1).all(dim=1)
            # get the predicted classes
            _, predicted = torch.max(outputs.data, 1)
            # assign -1 to the unknown class
            predicted[unknown] = -1
            unknown_indices.extend(i for i, x in enumerate(predicted) if x =

            # calculate loss petween predicted outputs and labels
            loss = criterion(outputs, labels)
            # set grads to zero, make suere we don't accumulate gradients fr
            optimizer.zero_grad()
            # computes the gradients of the loss function
            loss.backward()
            # updates the parameters of the neural network
            optimizer.step()
            train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            train_total += labels.size(0)
            train_correct += (predicted == labels).sum().item()
            train_accuracy = 100 * train_correct / train_total

        # evaluate
        model.eval()
        # calculate validation loss and accuracy
        val_loss, val_correct, val_total = 0.0, 0, 0
        # deactivate autograd engine - prevent updates & data leakage during
        with torch.no_grad():
            for inputs, labels in val_dataflow:
                inputs, labels = inputs.to(device, dtype=torch.float), label
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item()
                val_accuracy = 100 * val_correct / val_total

        # check if current accuracy is higher than the highest accuracy
        if val_accuracy > highest_val_accuracy:
            highest_val_accuracy = val_accuracy

            # Send info to dictionary
            best_model_info['epoch'] = epoch
            best_model_info['train_loss'] = train_loss / len(train_dataflow)
            best_model_info['train_accuracy'] = train_accuracy
            best_model_info['val_loss'] = val_loss / len(val_dataflow)
            best_model_info['val_accuracy'] = val_accuracy
            best_model_info['model_state_dict'] = model.state_dict()

        train_losses.append(train_loss)
        train_accuracies.append(train_accuracy)
        val_losses.append(val_loss)
```

```
    val_accuracies.append(val_accuracy)
    print(f"Train Acc: {train_accuracy:.2f}%, Train Loss: {train_loss/le

# Save the best model --- information is stored in dictionary (best_mode
torch.save(best_model_info['model_state_dict'], os.path.join(save_path,
# Pop off model.dict -- for printing purposes (clean)
best_model_info.popitem()
print("---TRAINING COMPLETE---")
print("Unknown indices: ", unknown_indices)
print("Best model info: ", best_model_info)
collected = gc.collect()
```

```
Train Acc: 98.60%, Train Loss: 0.04456506217441832,Val Acc: 95.89%, Val
Loss: 0.22468238174915314

Epoch 23/25 [Training]: 100%|██████████| 24/24 [00:23<00:00,  1.01it/s]

Train Acc: 99.44%, Train Loss: 0.01685492018683969,Val Acc: 97.16%, Val
Loss: 0.18313270211219787

Epoch 24/25 [Training]: 100%|██████████| 24/24 [00:23<00:00,  1.01it/s]

Train Acc: 99.41%, Train Loss: 0.022397278575226665,Val Acc: 96.60%, Va
l Loss: 0.1855709046125412

Epoch 25/25 [Training]: 100%|██████████| 24/24 [00:23<00:00,  1.01it/s]

Train Acc: 98.87%, Train Loss: 0.034431916335355105,Val Acc: 95.97%, Va
l Loss: 0.19136399924755096
---TRAINING COMPLETE---
Unknown indices:  []
Best model info:  {'epoch': 22, 'train_loss': 0.01685492018683969, 'tra
in_accuracy': 99.44162436548223, 'val_loss': 0.18313270211219787, 'val_
accuracy': 97.1563981042654}
```

In [12]:
```
# Plot learning curves - training vs validation - for loss & accuracy
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

axs[0].set_title("Training and Validation Loss")
axs[0].plot(train_losses,label="train")
axs[0].plot(val_losses,label="val")
axs[0].set_xlabel("Epochs")
axs[0].set_ylabel("Loss")
axs[0].legend()

axs[1].set_title("Training and Validation Accuracy")
axs[1].plot(train_accuracies,label="train")
axs[1].plot(val_accuracies,label="val")
axs[1].set_xlabel("Epochs")
axs[1].set_ylabel("Accuracy")
axs[1].legend()

plt.show()
```



In [13]:
```
# Call model instance (if not previously called) and load the saved mode
model = pretrainedVGG19(nclasses)
model.load_state_dict(torch.load('/blue/eel5840/darian.jennings/final_pr

# Move the model on the same device as the data, either CPU or GPU, for 
model = model.to(device)
model.eval()
```

Out[13]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
l_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
l_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
```

```
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
l_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=9, bias=True)
  )
)
```

In [14]:
```
test_accuracy, test_loss = 0, 0
correct, total = 0, 0
unknown_indices = []
test_losses = []
test_accuracies = []

with torch.no_grad():
    for data, target in test_dataflow:
        data, target = data.to(device, dtype=torch.float), target.to(dev
        output = model(data)

        # calculate the probabilities
        probabilities = torch.nn.functional.softmax(output, dim=1)
        #max_prob, _ = torch.max(probabilities, dim=1)
        #print(max_prob)
        # creates a boolean tensor unknown where each element is True if
        # probabilities in the corresponding row of the probabilities te
        # and False otherwise
        unknown = (probabilities < 0.25).all(dim=1)
        _, predicted = torch.max(output.data, 1)
        #print(predicted)
        # classify unknown images to the unknown class (-1)
        predicted[unknown] = -1
        unknown_indices.extend(i for i, x in enumerate(predicted) if x =

        test_loss += criterion(output, target).item()
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

        test_losses.append(test_loss)
        test_accuracies.append(test_accuracy)

# Calculate test metrics
test_loss /= len(test_dataflow.dataset)
test_accuracy = 100 * correct / total
y_true = target.cpu().numpy()
y_pred = predicted.cpu().numpy()
f1 = f1_score(y_true, y_pred, average='weighted')
recall = recall_score(y_true, y_pred, average='weighted')
precision = precision_score(y_true, y_pred, average='weighted')

print(f'Test Accuracy: {test_accuracy:.2f}%, Test Loss: {test_loss:.4f},
print("---TEST COMPLETE---")
print("Unknown indices: ", set(unknown_indices))
```

```
Test Accuracy: 96.29%, Test Loss: 0.0006, F1-score: 0.9710, Recall: 0.9
712, Precision: 0.9732
---TEST COMPLETE---
Unknown indices:  set()
```