Galaxy Imagery Classification

Introduction

Understanding the origins and evolution of galaxies is a fundamental aspect of astronomy. As galaxies come in various shapes, sizes, and may contain differing features, classifying them based on their morphology is an important task for astronomical research in this domain. To utilize galaxy imagery data effectively, such classification must be reliable and scalable. The Galaxy Zoo project, a pioneering citizen science initiative, has made significant strides in this domain by allowing volunteers to visually classify galaxy images from the Sloan Digital Sky Survey. However, with the rise of increasingly sophisticated and larger astronomical datasets, particularly of image data, there is a clear need for automation of this classification process.

The Galaxy Zoo Challenge as hosted on Kaggle, aims to address this need by opening a competition around suitable machine learning techniques to classify galaxies for a large dataset of images. Participants are tasked with developing algorithms that can replicate the probability distributions of galaxy classifications derived from the data obtained by human volunteers.

In this report, we explore different artificial neural network (ANN) architectures, predominately convolutional neural networks (CNN) and the application of transfer learning to the Galaxy Zoo dataset. CNNs have revolutionized image processing tasks due to their ability to automatically learn hierarchical features from raw image data and with the rise of deep learning, large models enable classification of large and complex patterns in data. Transfer learning, a technique that utilizes pre-trained models on new, but related tasks, offers a promising approach to improving classification performance, especially when dealing with limited labeled data.

The objective of our report is to evaluate the performance of various ANN architectures and assess the potential ways to use transfer learning on the accuracy of galaxy classification. By doing so, we aim to explore potential avenues for future research of robust classification systems on astronomical imagery data.

Data Description

The data utilized in this study is derived from the Galaxy Zoo challenge which can be found using the following link directing to Kaggle. This dataset includes 61.579 training and 79.976 test images of galaxies, the training data accompanied with a csv file containing labels for the target vector of 37 values representing the possible answers to 11 feature questions from the survey. As the values are probability distributions of various morphological galaxy features, the probability for each class would sum to 1, however, as some questions are conditional on the answers of previous questions, this is not the case. The final probabilistic values are based on a detailed decision tree design as can be seen below and are post-processed and weighted averages of the classifications made by the citizen scientists.

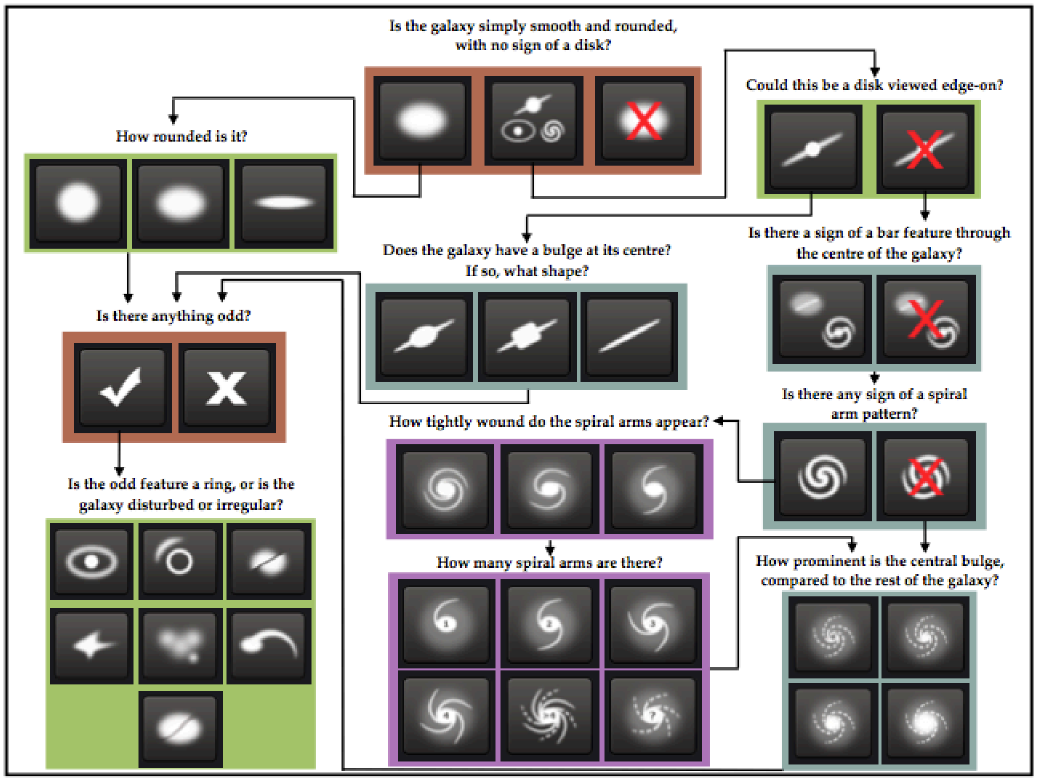
Is the galaxy simply smooth and rounded, with no sign of a disk?
Could this be a disk viewed edge-on?
How rounded is it?
Does the galaxy have a bulge at its centre? If so, what shape?
Is there a sign of a bar feature through the centre of the galaxy?
Is there anything odd?
Is there any sign of a spiral arm pattern?
How tightly wound do the spiral arms appear?
Is the odd feature a ring, or is the galaxy disturbed or irregular?
How many spiral arms are there?
How prominent is the central bulge, compared to the rest of the galaxy?

Figure 1. Flowchart of the classification tasks for GZ2, beginning at the top centre. Tasks are colour-coded by their relative depths in the decision tree. Tasks outlined in brown are asked of every galaxy. Tasks outlined in green, blue, and purple are (respectively) one, two or three steps below branching points in the decision tree. Table 2 describes the responses that correspond to the icons in this diagram.

The dataset is divided into two datasets, one for training and one for testing on Kaggle. The training dataset consists of JPG images of galaxies with the center of each galaxy positioned in the center of each image. The test dataset includes similar images of galaxies, which are used to evaluate the performance of the models. Predictions for the test dataset should be formatted according to Kaggle's submission requirements, listing the GalaxyID and the predicted probabilities for each class.

Fill in your Kaggle API credentials below. Do note to keep these private if you intend on publishing a derived version of this notebook!

In [ ]:
```
import os
os.environ['KAGGLE_USERNAME'] = 'joneron'
os.environ['KAGGLE_KEY'] = 'd495db6490b0e9b3ac98e008034829c9'
import kaggle
```

Import packages

In [ ]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as transforms
from torchvision.models import densenet121, resnet18
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from tqdm.notebook import tqdm
from zipfile import ZipFile
```

Setup

In [ ]:
```
SEED = 42
DATA_PATH = 'data'
MEDIA_PATH = 'media'
MODEL_PATH = 'models'
[os.makedirs(path, exist_ok=True) for path in [MEDIA_PATH, MODEL_PATH]]
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Check which accelerator is available. Change runtime within Google Colab or adjust based on your available hardware.

In [ ]:
```
if torch.backends.mps.is_available():
    device = torch.device('mps')
elif torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
print(f'Training on {device}.')
```

Preprocessing

The data from kaggle was downloaded and extracted from its zip archives to obtain images and CSV label file. Next, the file links to assign labels to the corresponding dtraining data in the CSV label file were parsed and updated to correspond to our environment setup. To ensure robust model training and no additional image order biases, the data was shuffled and split into a training and validation sets using a 80/20 ratio. This step helps to prevent overfitting and provides a means to evaluate model performance on a separate validation set. A custom dataset class was defined to facilitate efficient loading of images and labels during training. Various image preprocessing transformations were considered, such as converting images to grayscale, cropping, resizing, and normalizing pixel values. Example transformations were visualized to ensure the correctness of the preprocessing steps and to select the most effective transformation pipeline for the training process.

Downloading and unpacking dataset files.

In [ ]:
```
challenge_name = 'galaxy-zoo-the-galaxy-challenge'

if not os.path.exists(DATA_PATH):
    os.makedirs(DATA_PATH)

    kaggle.api.authenticate()
    kaggle.api.competition_download_files(challenge_name, path=DATA_PATH)

    # Extract competition zip file
    with ZipFile(f'{DATA_PATH}/{challenge_name}.zip', 'r') as zipf:
            zipf.extractall(DATA_PATH)

    # Extract contained train, test zip files within
    for file in os.listdir(DATA_PATH):
        if file.endswith('.zip'):
            with ZipFile(f'{DATA_PATH}/{file}', 'r') as zipf:
                zipf.extractall(DATA_PATH)

    # Only keep data and clear zips
    for file in os.listdir(DATA_PATH):
        if file.endswith('.zip'):
            os.remove(f'{DATA_PATH}/{file}')
    print('Setup complete.')
else:
    print('Setup skipped, files detected.')
```

Load training targets and reformat URL strings.

In [ ]:
```
# Parse and reformat
train_img_dir = f'{DATA_PATH}/images_training_rev1/'
train_full = pd.read_csv(f'{DATA_PATH}/training_solutions_rev1.csv')
train_full['GalaxyID'] = train_full['GalaxyID'].astype(int).astype(str) + '.jpg'
train_full['GalaxyID'] = train_full['GalaxyID'].str.zfill(8)
train_full['GalaxyID'] = train_img_dir + train_full['GalaxyID']

# Shuffle data and split into training and validation dataset based on a 80/20 ratio split.
train_full = train_full.sample(frac=1, random_state=SEED).reset_index(drop=True)
train, validation = train_test_split(train_full, test_size=0.2, random_state=SEED)

# Visually inspect data targets
train.head()
```

Dataset class definition

In [ ]:
```
classes = ['Class1.1', 'Class1.2', 'Class1.3', 'Class2.1', 'Class2.2', 'Class3.1', 'Class3.2', 'Class4.1', 'Cla
n_sub_classes = [3, 2, 2, 2, 4, 2, 3, 7, 3, 3, 6]
n_main_classes = 11
n_classes = len(classes)

class GalaxyDataset(Dataset):
    def __init__(self, df, transform=None):
        self.df = df
        self.transform = transform

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

    def __getitem__(self, idx):
        img_name = self.df.iloc[idx, 0]
        img = plt.imread(img_name)
        label = self.df.iloc[idx, 1:].values.astype(float)

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

        return img, label
```

Preprocessing configuration

In [ ]:
```
def get_transform_config(grayscale=False, crop=None, resize=None):
    """
    Returns a composition of transformations for preprocessing images
    """
    transform = []
    transform.append(transforms.ToPILImage())
    if grayscale:
        transform.append(transforms.Grayscale(num_output_channels=1))
    if crop:
        transform.append(transforms.CenterCrop(crop))
    if resize:
        transform.append(transforms.Resize(resize))
    transform.append(transforms.ToTensor())
    return transforms.Compose(transform)

def view_example_transforms(transform_configs, config_names, n_examples=5):
    """
    Displays original and transformed images
    """
    n_configs = len(transform_configs)
    fig, axes = plt.subplots(n_configs, n_examples, figsize=(3*n_examples, 3.5*n_configs))
```

```python
    config_names = list(config_names)  # Convert dict_keys to list

    for i, transform in enumerate(transform_configs):
        dataset = GalaxyDataset(train, transform)

        for j in range(n_examples):
            img, label = dataset[j]
            if img.shape[0] == 1:
                img = img.squeeze(0)
                axes[i, j].imshow(img, cmap='gray')
            else:
                img = img.permute(1, 2, 0)
                axes[i, j].imshow(img)
            axes[i, j].axis('off')

        axes[i, int(np.floor(n_examples/2))].set_title(config_names[i], fontweight='bold')

    plt.tight_layout()
    plt.show()

transform_configs = {
    'Original': get_transform_config(),
    'Grayscale': get_transform_config(grayscale=True),
    'Grayscale + Crop 256': get_transform_config(grayscale=True, crop=256),
    'Grayscale + Crop 256 + Resize 128': get_transform_config(grayscale=True, crop=256, resize=128),
    'Grayscale + Crop 256 + Resize 64': get_transform_config(grayscale=True, crop=256, resize=64)
}

view_example_transforms(transform_configs.values(), transform_configs.keys())
```

Methodology

Custom network architectures

We designe two neural network architectures: a convolutional network (`ConvNet`) and a fully linear network (`LinNet`). The `ConvNet` consists of three convolutional layers followed by ReLU activation and max-pooling layers. It also includes fully connected layers to produce the final classification output. In contrast, `LinNet` is a simpler baseline model with fully connected layers but no convolutional layers, used to compare performance benefits.

Transfer Learning

As an alternative approach, we utilize the `DenseNet121` and `ResNet18` architecture, pre-trained on the ImageNet dataset, for transfer learning. By replacing and training the final classification layer on our galaxy dataset while freezing the initial layers, we attempt to leverage the pre-trained model feature extraction capabilities to ideally improve performance and reducing training time.

Training process

The models were trained using the Root Mean Squared Error (RMSE) loss function as is employed by the challenge assessment. For optimization, the Adam optimizer is utilized as it is generally accepted as a well-performing optimizer. After setting up datasets for training and validation, we selected the best initial weights through multiple training trials and employ early stopping in the longer training procedure that follows to prevent overfitting.

Parameter selection

Hyperparameters such as learning rate, batch size, and the number of epochs were chosen within commonly used ranges. The training and validation losses were plotted to monitor the learning process. A thorough analysis was not performed.

Metrics

The accuracy of our model is determined using its likelihood to predict the most probable outcome of a class. This is a simplification as the accuracy ideally reflects the entire probability distribution as is expressed by RMSE.

In [ ]:
```python
class ConvNet(nn.Module):
    """
    Convolutional network
    """
    def __init__(self, dim=424, in_channels=3, intermed_size=300):
        super(ConvNet, self).__init__()
        self.dim = dim
        self.conv1 = nn.Conv2d(in_channels, 10, kernel_size=4)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=4)
        self.conv3 = nn.Conv2d(20, 40, kernel_size=4)

        self.fc1 = nn.Linear(1000, intermed_size)
        self.fc2_qs = nn.ModuleList([nn.Linear(intermed_size, n) for n in n_sub_classes])

    def forward(self, x):
        n_batch = x.size(0)
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = F.relu(F.max_pool2d(self.conv3(x), 2))
        x = x.view(n_batch, -1)
        x = F.relu(self.fc1(x))
        x = torch.cat([F.sigmoid(fc(x)) for fc in self.fc2_qs], dim=1)
        return x


class LinNet(nn.Module):
    """
    Fully linear network
    """
    def __init__(self, dim=424, in_channels=3):
        super(LinNet, self).__init__()
        self.dim = dim
        self.fc1 = nn.Linear(dim*dim*in_channels, 300)
        self.fc2_qs = nn.ModuleList([nn.Linear(300, n) for n in n_sub_classes])

    def forward(self, x):
        n_batch = x.size(0)
        x = x.view(n_batch, -1)
        x = F.relu(self.fc1(x))
        x = torch.cat([F.sigmoid(fc(x)) for fc in self.fc2_qs], dim=1)
        return x


def DenseNet121(n_classes):
    model = densenet121(pretrained=True, memory_efficient=True) # Load pretrained model
    model.classifier = nn.Linear(1024, n_classes)
    for param in model.parameters():
        param.requires_grad = False
    for param in model.classifier.parameters():
        param.requires_grad = True
    return model


def ResNet18(n_classes):
    model = resnet18(pretrained=True) # Load pretrained model
    model.fc = nn.Linear(512, n_classes)
    for param in model.parameters():
        param.requires_grad = False
    for param in model.fc.parameters():
        param.requires_grad = True
    return model
```

Training and evaluation code for networks

In [ ]:
```
def rmse_loss(y_pred, y_true):
    return torch.sqrt(F.mse_loss(y_pred, y_true))

def hierarchical_loss(outputs, targets, n_sub_classes=n_sub_classes):
    """
    Custom loss function that respects the hierarchical structure of the decision tree.
    """
    loss = 0
    start = 0
    for n in n_sub_classes:
        end = start + n
        loss += F.binary_cross_entropy(outputs[:, start:end], targets[:, start:end])
        start = end
    return loss

def train_model(model, criterion, optimizer, train_loader, val_loader, n_epochs=10, show_avg_loss=100, exclude_
    """
    Trains the model using the given criterion, optimizer and data loader for a given number of epochs or unti
    """
    model = model.to(device)
    model.train()

    metrics = {
        'train_loss_by_batch': [],
        'val_loss_by_epoch': []
    }
```

```python
    train_loss = metrics['train_loss_by_batch']
    val_loss = metrics['val_loss_by_epoch']

    pbar = tqdm(total=len(train_loader) * n_epochs, leave=False, desc='Loss: ')

    if not exclude_init_val:
        # Validate once for initial loss
        mean_loss, _ = validate_model(model, criterion, val_loader)
        val_loss.append(mean_loss)

    for n in range(n_epochs):
        # Train
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            inputs = inputs.float().to(device)
            labels = labels.float().to(device)

            optimizer.zero_grad()

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_loss.append(loss.item())
            pbar.update(1)
            if i % show_avg_loss == show_avg_loss-1:
                pbar.set_description(f'Loss: {sum(train_loss[-show_avg_loss:])/show_avg_loss:.4f}')

        # Validate to test for over-fitting
        mean_loss, _ = validate_model(model, criterion, val_loader)
        val_loss.append(mean_loss)

        if n > 2:
            if val_loss[n-2] < val_loss[n-1] < val_loss[n]:
                pbar.close()
                # Loss increased twice in a row on validation set so we seem to have converged
                break
    pbar.close()
    return model, metrics

def reset_init_weights(model):
    """
    Resets the initial weights of the model using Xavier initialization
    """
    if isinstance(model, nn.Conv2d) or isinstance(model, nn.Linear):
        nn.init.xavier_uniform_(model.weight)
        model.bias.data.fill_(0.01)

def select_best_initial_weights(model, criterion, lr, train_loader, val_loader, n_epochs=1, n_trials=10):
    """
    Selects the best initial weights for the model by running multiple short training cycles and selecting the
    """
    model = model.to(device)
    best_loss = np.inf
    best_weights = None
    pbar = tqdm(range(n_trials), desc='Trial', leave=False)
    for _ in pbar:
        model.apply(reset_init_weights) # Reset initial random weights
        optimizer = optim.Adam(model.parameters(), lr=lr)
        model, metrics = train_model(model, criterion, optimizer, train_loader, val_loader, n_epochs)
        test_loss, test_accuracy = validate_model(model, criterion, val_loader)
        cross_accuracy = torch.mean(test_accuracy)
        pbar.set_description(f'Test accuracy: {cross_accuracy}')
        if test_loss < best_loss:
            best_loss = test_loss
            best_weights = model.state_dict()
            best_weights_metrics = metrics
    return best_weights, best_weights_metrics

def train_best_model(model, criterion, optimizer, lr, train_loader, val_loader, n_train_epochs, n_trial_epochs=
    """
    If in doubt, the most comprehensive training.
    Trains the model by selecting the best initial weights and then performing training for given number of epo
    """
    best_weights, metrics = select_best_initial_weights(model, criterion, lr, train_loader, val_loader, n_epoch
    model.load_state_dict(best_weights)
    model = model.to(device)
    if n_trial_epochs < n_train_epochs:
        model, further_metrics = train_model(model, criterion, optimizer, train_loader, val_loader, n_train_epo
        metrics = merge_metrics(metrics, further_metrics)
    if save_model:
        torch.save(model.state_dict(), f'{MODEL_PATH}/{model_name}.pt')
    # Merge metrics
    return model, metrics

def validate_model(model, criterion, val_loader):
    """
    Validates the model using the given criterion and data loader
    """
    model = model.to(device)
    model.eval()
    total_loss = 0
    multiclass_acc = torch.zeros((11))
    with torch.no_grad():
        for i, data in enumerate(val_loader, 0):
            inputs, labels = data
            inputs = inputs.float().to(device)
            labels = labels.float().to(device)

            outputs = model(inputs)

            loss = criterion(outputs, labels)
            total_loss += loss.item()

            multiclass_acc += get_accuracy(outputs, labels)

    mean_loss = total_loss / len(val_loader)
    mean_multiclass_acc = multiclass_acc / len(val_loader)
    return mean_loss, mean_multiclass_acc

def merge_metrics(metrics1, metrics2):
    """
    Merges two metrics dictionaries
    """
    merged = {}
    for key in metrics1.keys():
        if type(metrics1[key]) == list:
            merged[key] = metrics1[key] + metrics2[key]
        else:
            raise ValueError('Only lists are supported for merging')
    return merged

def get_accuracy(outputs, labels):
    """
    Returns the accuracy of the model for each class
    """
    predictions = map(torch.argmax, [outputs[:, i:i+n] for i, n in enumerate(n_sub_classes)])
    true_targets = map(torch.argmax, [labels[:, i:i+n] for i, n in enumerate(n_sub_classes)])
    return torch.tensor([torch.sum(pred == true).item() for pred, true in zip(predictions, true_targets)])

def view_train_loss_vs_val_loss(train_loss, val_loss, model_name, batches_per_epoch):
    train_batches = range(len(train_loss))
    val_epochs = [i*batches_per_epoch for i in range(len(val_loss))]
    plt.figure(figsize=(10, 5))
    plt.plot(train_batches, train_loss, label='Train loss')
    plt.plot(val_epochs, val_loss, label='Validation loss')
    plt.xlabel('Mini-batches')
    plt.ylabel('Loss')
    plt.title(f'{model_name} loss after training {len(train_loss)} mini-batches')
    plt.legend()
    plt.savefig(f'{MEDIA_PATH}/train_vs_val_loss_{model_name}.png')
    plt.show()

def view_accuracy(val_accuracy, train_accuracy, model_name):
    classes = range(len(val_accuracy))
    width = 0.35
    fig, ax = plt.subplots(figsize=(10, 5), layout='tight')
    ax.bar([e - width/2 for e in classes], val_accuracy, width, label='Validation accuracy')
    ax.bar([e + width/2 for e in classes], train_accuracy, width, label='Train accuracy')
    ax.set_xlabel('Classes')
    ax.set_xticks([i for i in range(val_accuracy.shape[0])])
    ax.set_ylabel('Accuracy')
    ax.set_title(f'{model_name} accuracy')
    ax.legend()
    plt.savefig(f'{MEDIA_PATH}/accuracy_{model_name}.png')
    plt.show()
```

Results

We find that all architectures utilised quickly converge during the first few epochs. Following training progress is slowed down and appears to converge at a suboptimal rate. Through a few trial attempts, we find that the initial randomization of weights does impact the performance significantly. However, distinictions between `ConvNet` and `LinNet` in terms of their predictability are hard to make as accuracy scores appear very low. This likely implies that our neural network pipeline or pre-processing of images is largely unsuccessful in capturing the detail of this dataset sufficiently.

Due to the size of the transfer learning models, elaborate training analysis beyond the loss visualizations were not feasible.

Parameter definition, model setup and training process

In [ ]:
```
lr = 1e-5
n_train_epochs = 10
n_trial_epochs = 1
n_trials = 10
batch_size = 64
dim = 64
in_channels_preprocessed = 1

criterion = rmse_loss

transform_config = get_transform_config()
train_loader = DataLoader(GalaxyDataset(train, transform_config), batch_size=batch_size, shuffle=True)
val_loader = DataLoader(GalaxyDataset(validation, transform_config), batch_size=batch_size, shuffle=False)

transform_config_preprocessed = get_transform_config(grayscale=True, crop=256, resize=dim)
train_loader_preprocessed = DataLoader(GalaxyDataset(train, transform_config_preprocessed), batch_size=batch_si
val_loader_preprocessed = DataLoader(GalaxyDataset(validation, transform_config_preprocessed), batch_size=batch

# Train model configurations
models = [
    ConvNet(dim=dim, in_channels=in_channels_preprocessed),
    LinNet(dim=dim, in_channels=in_channels_preprocessed),
    DenseNet121(n_classes),
    ResNet18(n_classes)
    ]
model_names = [
    'ConvNet',
    'LinNet',
    'DenseNet121',
    'ResNet18'
    ]
train_loaders = [
    train_loader_preprocessed,
    train_loader_preprocessed,
    train_loader,
    train_loader
    ]
val_loaders = [
    val_loader_preprocessed,
    val_loader_preprocessed,
    val_loader,
    val_loader
    ]

for model, model_name, train, val in zip(models, model_names, train_loaders, val_loaders):
    optimizer = optim.Adam(model.parameters(), lr)
    if model_name in ['ConvNet', 'LinNet']:
        model, metrics = train_best_model(model, criterion, optimizer, lr, train, val, n_train_epochs, n_trial_
    else:
        model, metrics = train_model(model, criterion, optimizer, train, val, n_train_epochs)
    view_train_loss_vs_val_loss(metrics['train_loss_by_batch'], metrics['val_loss_by_epoch'], model_name, len(t
    val_loss, val_accuracy = validate_model(model, criterion, val)
    train_loss, train_accuracy = validate_model(model, criterion, train)
    view_accuracy(val_accuracy, train_accuracy, model_name)
```

Discussion

There are a number of flaws in methodology and model selection that are cause for concern regarding the reliability of results. Due to the selection of large networks, training takes a large amount of time and the exploration of the parameter space in many configurations becomes infeasible. For the inital exploration, smaller models are recommended and should suffice in capturing the core of this challenge and upscaling of models be delayed unless access to appropriate hardware exists.

We note that we observe the networks learn and converge to a rather suboptimal solution. This is likely due to parameterization of learning rate as well as due to the lack of further pre-processing transformations. Introduction of rotations, translations amongst others could have improved the results further, as also elaborated on by Ethiraj et. al..

The choice to not incorporate the probabilistic decision tree information was made for simplicity and according to Ethiraj et. al. this does not majorly impact results. However, particularly in our poor performing scenario, such incorporation could have aided in improving our results.

Conclusion

Overall, we believe that our project has not been successful in showcasing new avenues to utilize neural networks on this dataset. However, we hope it serves as a reminder to follow a rigorous procedure starting from base principles and exploring parameter spaces at a smaller scale to enable well-designed machine learning practices target at a particular problem set.

Note on Usage of Large Language Models (LLMs)

For the completion of this project, GPT-4, GPT-4o-64k and GPT-4o-128k were used through the intermediary provider Poe. Their main contributions were the generation of a draft code elements for training and validation functions as well as the creation of an outline for the report, compilation of information for the introduction and method sections, which were used as a starting reference. All content generated by the model was inspected manually and in most cases modified again before arriving at this final format.