

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ВТ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Основы нейронных сетей»
Тема: «Свёрточные нейронные сети»

Студент гр. 2308

Кислицин К. Р.

Преподаватель

Турсуков Н. О.

Санкт-Петербург

2025

Содержание

Цель работы	3
Задачи работы	3
Теоретическая информация	3
Ход работы.....	6
Выводы	24
Приложения	25

Цель работы

Целью работы является получение и закрепление навыков предобработки данных, разработки архитектур и обучения свёрточных нейронных сетей.

Задачи работы

1. Подготовка данных

- Предобработка
- Расширение
- Корреляция данных (матрица корреляций)

2. Разработка архитектур:

- Разработка 3-5 архитектур
- Подготовка предобученной модели

3. Обучение моделей и подбор параметров:

- Использование функций потерь, оптимизаторов и методов регуляризации

3. Оценка моделей

- Визуализация предсказанных значений
- Оценка качества прогноза
- Сравнение моделей

Теоретическая информация

Сверточные нейронные сети (Convolutional Neural Networks, CNN) представляют собой класс глубоких нейронных сетей, которые значительно улучшили результаты в задачах компьютерного зрения, таких как классификация изображений, детекция объектов, сегментация и другие. Основной особенностью CNN является использование сверточных слоев, которые автоматически извлекают важные признаки из изображений.

1. История и развитие

Идея сверточных нейронных сетей возникла в 1980-х годах, когда японский исследователь Йошуа Лекун предложил использовать сверточные слои для распознавания рукописных цифр в проекте LeNet. Однако важное развитие CNN получили с появлением более мощных вычислительных ресурсов и больших наборов данных. Сетевые архитектуры, такие как AlexNet (2012), VGG (2014), ResNet (2015) и другие, позволили значительно улучшить точность распознавания на стандартизированных наборах данных (например, ImageNet).

2. Основные компоненты CNN

CNN состоит из нескольких ключевых компонентов, каждый из которых играет важную роль в процессе извлечения признаков из входных данных:

Сверточный слой (Convolutional Layer): Сверточный слой выполняет операцию свертки между изображением и набором фильтров (или ядер). Этот процесс позволяет выявить локальные особенности изображения, такие как края, текстуры, углы и другие характерные элементы. Каждый фильтр представляет собой небольшую матрицу, которая скользит по изображению, вычисляя скалярное произведение между фильтром и частью изображения. Результатом свертки является карта признаков (feature map), которая содержит активированные признаки.

Функция активации (Activation Function): После свертки выходные значения проходят через функцию активации, такую как ReLU (Rectified Linear Unit), которая добавляет нелинейность в модель. Это необходимо для того, чтобы сеть могла моделировать сложные зависимости между признаками.

Пуллинг (Pooling): Пуллинг — это операция, которая используется для уменьшения размерности карты признаков и уменьшения вычислительных затрат. Наиболее распространены два типа пуллинга: максимальный пуллинг (max pooling) и средний пуллинг (average pooling). В случае максимального пуллинга выбирается максимальное значение в каждом сегменте карты признаков.

Полносвязные слои (Fully Connected Layer): После извлечения признаков сверточными и пуллинговыми слоями, эти признаки передаются в полносвязные слои. В этих слоях каждый нейрон связан с каждым нейроном предыдущего слоя. Полносвязные слои часто используются для классификации или регрессии, где на выходе сети предоставляется конечный результат.

Выходной слой (Output Layer): На выходе CNN обычно применяется softmax или sigmoid функция для классификации, которая нормализует выходные значения в вероятности принадлежности к различным классам.

3. Особенности и преимущества CNN

Локальность признаков: Сверточные слои используют локальные рецептивные поля, что позволяет извлекать информацию о локальных признаках, таких как края или текстуры, и помогает лучше понять структуру изображения.

Параметризация: В отличие от традиционных полносвязных сетей, где каждый нейрон связан с каждым в предыдущем слое, сверточные сети используют общие веса для всех позиций в изображении. Это сокращает количество параметров, делая модель более эффективной и менее подверженной переобучению.

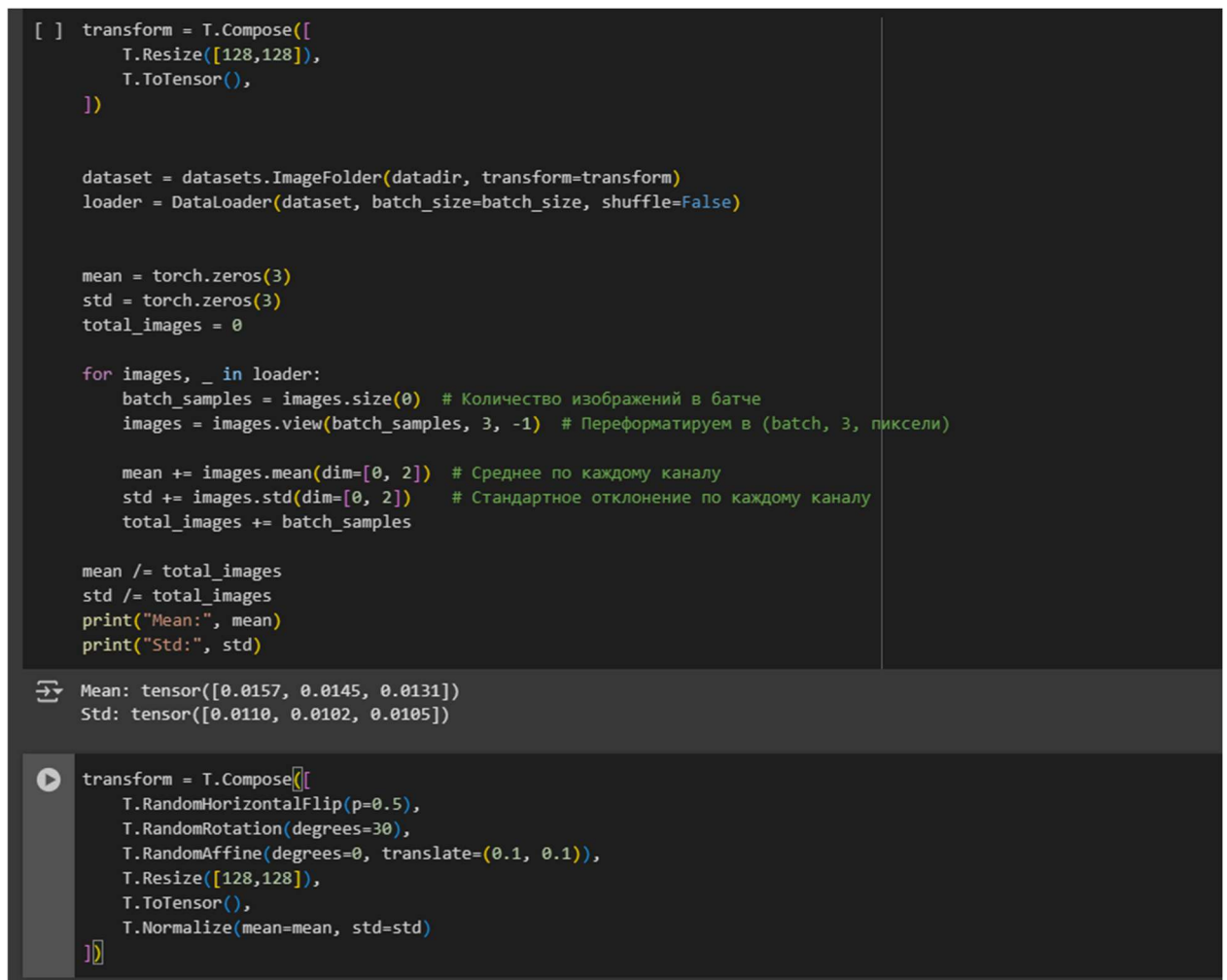
Переводимость признаков: Признаки, извлеченные сверточными слоями, являются инвариантными к переворотам, сдвигам и

масштабированию. Это означает, что CNN может распознавать объекты на изображениях, независимо от того, где и как они находятся в кадре.

Ход работы

В рамках данной лабораторной работы был рассмотрен датасет, состоящий из изображений главных героев мультсериала Скуби-Ду.

Создал объект для предобработки данных. Mean и std для нормализации подобраны в соответствии с датасетом. Настроено расширение с помощью вращения, сдвига и отражения (Рисунок 1).



```
[ ] transform = T.Compose([
    T.Resize([128,128]),
    T.ToTensor(),
])

dataset = datasets.ImageFolder(datadir, transform=transform)
loader = DataLoader(dataset, batch_size=batch_size, shuffle=False)

mean = torch.zeros(3)
std = torch.zeros(3)
total_images = 0

for images, _ in loader:
    batch_samples = images.size(0) # Количество изображений в батче
    images = images.view(batch_samples, 3, -1) # Переформатируем в (batch, 3, пиксели)

    mean += images.mean(dim=[0, 2]) # Среднее по каждому каналу
    std += images.std(dim=[0, 2]) # Стандартное отклонение по каждому каналу
    total_images += batch_samples

mean /= total_images
std /= total_images
print("Mean:", mean)
print("Std:", std)

↵ Mean: tensor([0.0157, 0.0145, 0.0131])
Std: tensor([0.0110, 0.0102, 0.0105])

▶ transform = T.Compose([
    T.RandomHorizontalFlip(p=0.5),
    T.RandomRotation(degrees=30),
    T.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    T.Resize([128,128]),
    T.ToTensor(),
    T.Normalize(mean=mean, std=std)
])
```

Рисунок 1 – Предобработка данных

Создал датасет, с помощью названий папок классов, разделил его на обучающую и тестовую выборки, создал загрузчики для выборок (Рисунок 2)

```
dataset=datasets.ImageFolder(datadir,transform=transform)
len(dataset)

221

[ ] train_dataset, test_dataset = torch.utils.data.random_split(dataset, [191, 30])

[ ] train_loader = DataLoader(train_dataset, batch_size=batch_size,shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,shuffle=True)
```

Рисунок 2 – Создание датасета

Вывел примеры входных данных до и после нормализации (Рисунок 3).

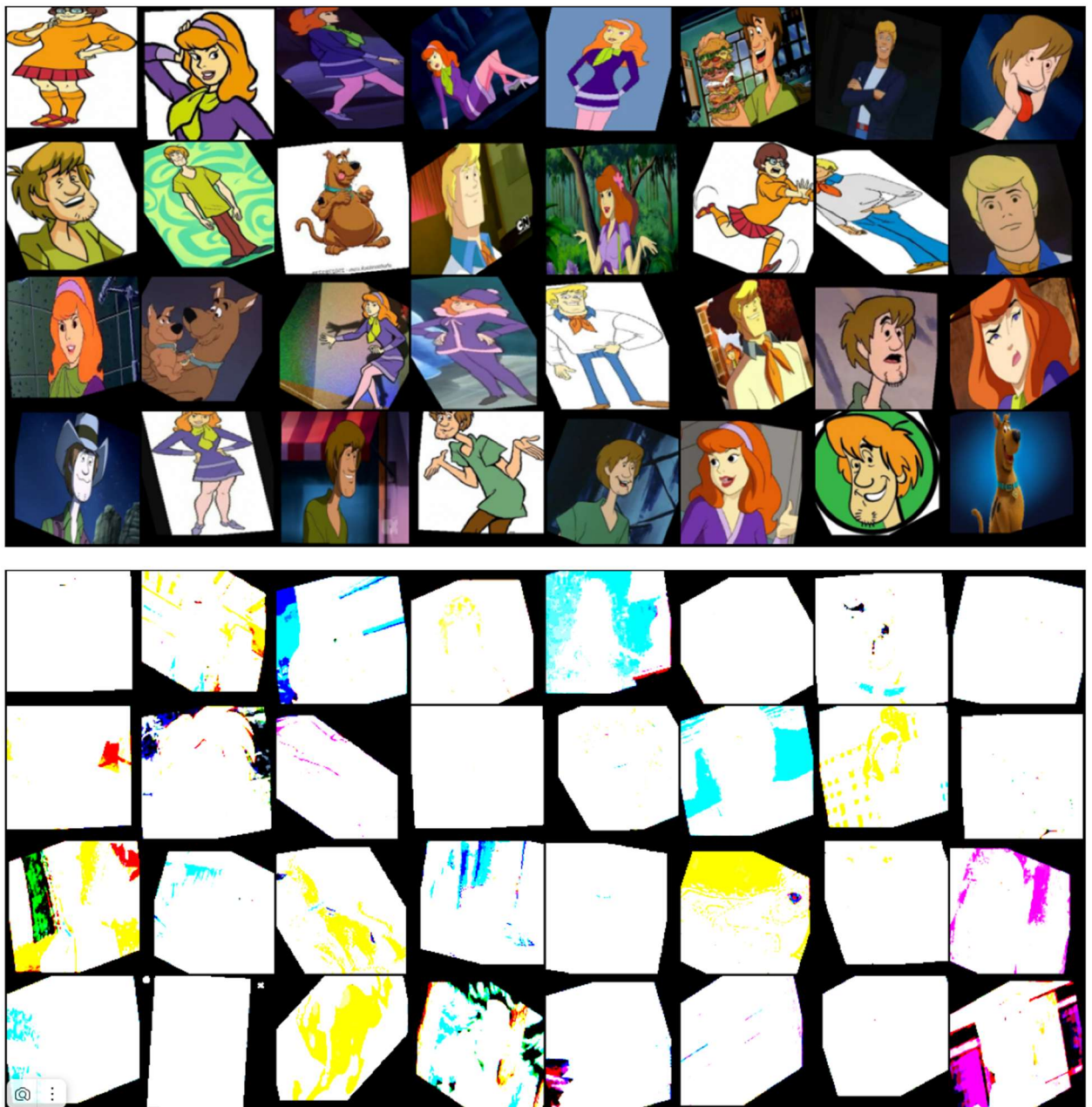


Рисунок 3 – Вывод примеров

Создал оптимизатор и функцию ошибок, загрузил данные на gpu (Рисунок 4).

```
[ ] optimizer = optim.Adam(model.parameters(), weight_decay=1e-4, lr=0.001)
    criterion = nn.CrossEntropyLoss()

[ ] for images, labels in train_loader:
    images, labels = images.to(device), labels.to(device)
    output = model(images)
```

Рисунок 4 – Оптимизатор и функция ошибки

С помощью библиотеки ignite создал объекты для обучения и валидации (Рисунок 5)

```
▶ trainer = create_supervised_trainer(model, optimizer, criterion, device=device)

[ ] metrics = { "loss" : Loss(criterion),
               "accuracy" : Accuracy(),
               "cm": ConfusionMatrix(num_classes=5)}

[ ] train_evaluator = create_supervised_evaluator(model, metrics = metrics, device = device)
    test_evaluator = create_supervised_evaluator(model, metrics = metrics, device = device)

[ ] training_history = {'accuracy':[], 'loss':[]}
    test_history = {'accuracy':[], 'loss':[]}
    last_epoch = []
```

Рисунок 5 – Объекты для обучения и валидации

Создал механизм, выводящий информацию в процессе обучения и сохраняющий наиболее удачную модель (Рисунок 6).

```

@trainer.on(Events.EPOCH_COMPLETED)
def log_training_results(trainer):
    train_evaluator.run(train_loader)
    metrics = train_evaluator.state.metrics
    accuracy = metrics['accuracy']*100
    loss = metrics['loss']
    last_epoch.append(0)
    training_history['accuracy'].append(accuracy)
    training_history['loss'].append(loss)
    print("Результат обучения - Эпоха: {} Сред. точность: {:.2f} Сред. ошибка: {:.5f}"
          .format(trainer.state.epoch, accuracy, loss))

@trainer.on(Events.EPOCH_COMPLETED)
def log_test_results(trainer):
    test_evaluator.run(test_loader)
    metrics = test_evaluator.state.metrics
    accuracy = metrics['accuracy']*100
    loss = metrics['loss']
    test_history['accuracy'].append(accuracy)
    test_history['loss'].append(loss)
    print("Результат валидации - Эпоха: {} Сред. точность: {:.2f} Сред. ошибка: {:.5f}"
          .format(trainer.state.epoch, accuracy, loss))

[ ] breaker = EarlyStopping(patience = 10, score_function = lambda engine : engine.state.metrics['accuracy'],
                             trainer = trainer)
test_evaluator.add_event_handler(Events.COMPLETED, breaker);

[ ] to_save = {'model': model}
saver = Checkpoint(
    to_save, './models',
    n_saved=1, filename_prefix='best',
    score_name="accuracy",
    global_step_transform=global_step_from_engine(trainer),
)
test_evaluator.add_event_handler(Events.COMPLETED, saver);

@trainer.on(Events.COMPLETED)
def log_confusion_matrix(trainer):
    test_evaluator.run(test_loader)
    metrics = test_evaluator.state.metrics
    cm = metrics['cm']
    cm = cm.numpy()
    cm = cm.astype(int)
    classes = dataset.classes
    fig, ax = plt.subplots(figsize=(10,10))
    ax= plt.subplot()
    sns.heatmap(cm, annot=True, ax = ax,fmt="d")
    # labels, title and ticks
    ax.set_xlabel('Predicted labels')
    ax.set_ylabel('True labels')
    ax.set_title('Confusion Matrix')
    ax.xaxis.set_ticklabels(classes,rotation=90)
    ax.yaxis.set_ticklabels(classes,rotation=0)

```

Рисунок 6 – Механизм обучения

Модель 1

Создал архитектуру модели 1 (Рисунок 7).

```

00 00 00
Модель 1
00 00 00

class M1(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        self.fc1 = nn.Linear(64 * 32 * 32, 128)
        self.fc2 = nn.Linear(128, 5)

    def forward(self, x):
        x = self.pool(torch.tanh(self.conv1(x)))
        x = self.pool(torch.tanh(self.conv2(x)))

        x = x.view(-1, 64 * 32 * 32)
        x = torch.tanh(self.fc1(x))
        x = self.fc2(x)

        return x

model = M1()
model.to(device)

```

Рисунок 7 – Модель 1

Модель имеет минимальное количество слоёв, в связи с чем настройка гиперпараметров для неё не проводилась. Результаты можно видеть на Рисунке 8.

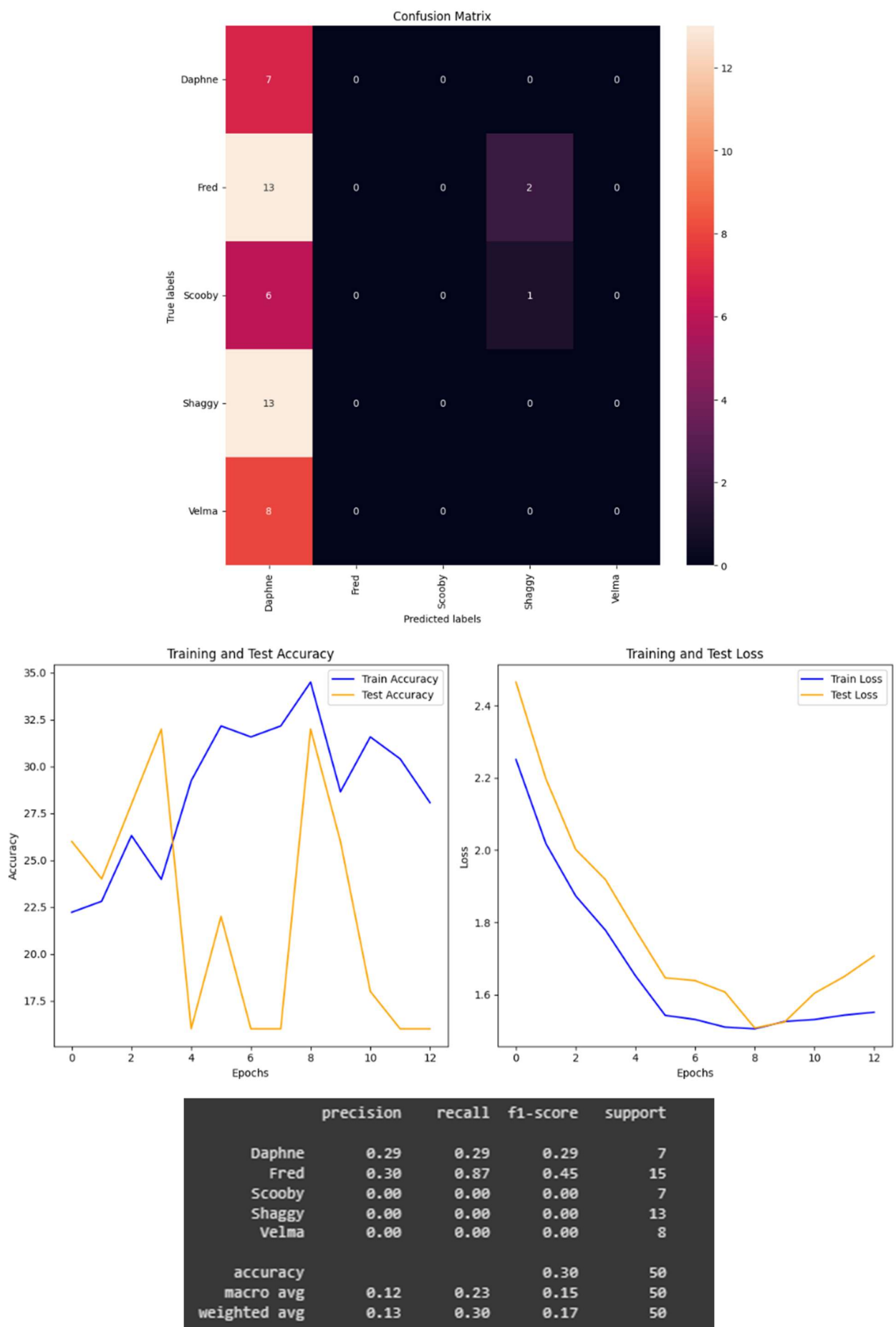


Рисунок 8 – Результаты модели 1

Модель 2

Создал архитектуру модели 2 (Рисунок 9).

```
"""
Модель 2
"""

class M2(nn.Module):
    def __init__(self):
        super(M2, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=2, padding=1)
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=2, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=2, padding=1)
        self.conv6 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=2, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv7 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=2, padding=1)
        self.conv8 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=2, padding=1)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(512*9*9, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 5)

    def forward(self, x):

        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool1(x)

        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool2(x)

        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = self.pool3(x)

        x = F.relu(self.conv7(x))
        x = F.relu(self.conv8(x))
        x = self.pool4(x)
        print(x.shape)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)

        return x

model = M2()
model.to(device)
```

Рисунок 9 – Модель 2

Для данной модели производилась настройка различных параметров: скорость обучения, штрафы и т.д. На версии 2 было отключено расширение датасета, что сильно улучшило результаты. Данные по 4 версиям преведины на Рисунках 10-12.

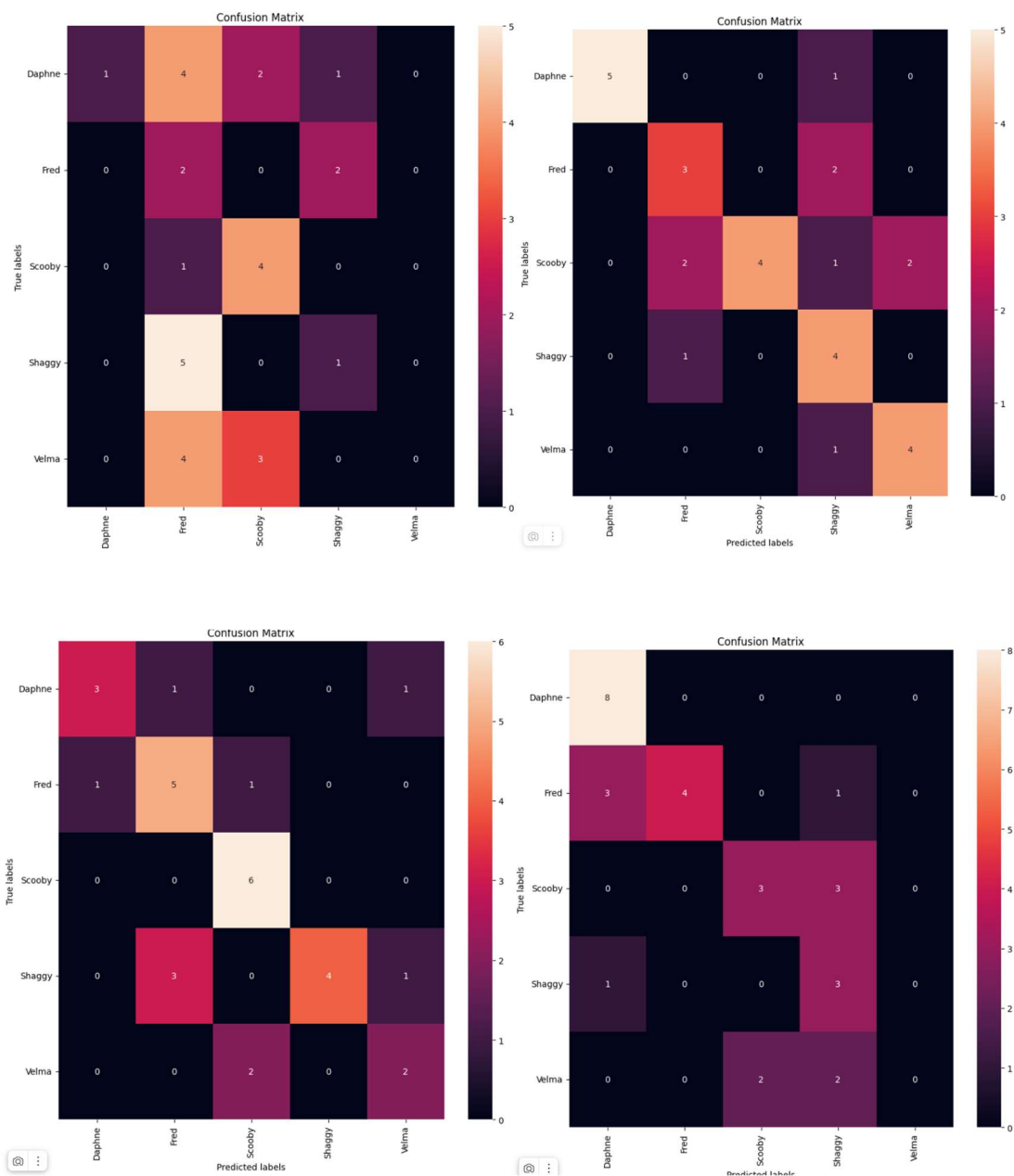


Рисунок 10 – Матрицы для 4 версий модели 2

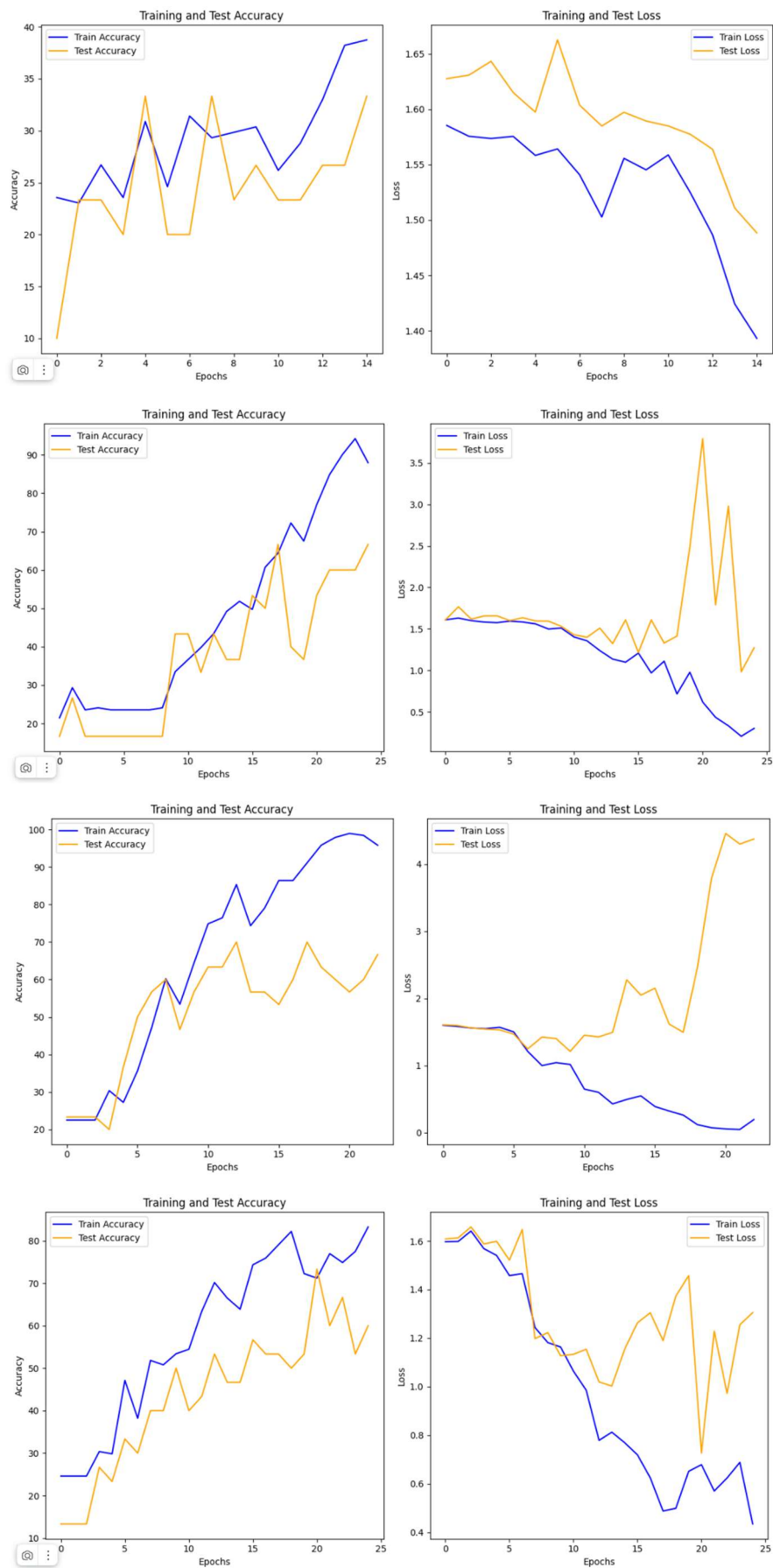


Рисунок 11 – Графики для 4 версий модели 2

	precision	recall	f1-score	support
Daphne	0.00	0.00	0.00	8
Fred	0.00	0.00	0.00	4
Scooby	0.31	0.80	0.44	5
Shaggy	0.29	0.83	0.43	6
Velma	0.00	0.00	0.00	7
accuracy			0.30	30
macro avg	0.12	0.33	0.18	30
weighted avg	0.11	0.30	0.16	30

	precision	recall	f1-score	support
Daphne	0.75	0.50	0.60	6
Fred	0.80	0.80	0.80	5
Scooby	0.56	1.00	0.72	9
Shaggy	0.80	0.80	0.80	5
Velma	0.00	0.00	0.00	5
accuracy			0.67	30
macro avg	0.58	0.62	0.58	30
weighted avg	0.59	0.67	0.60	30

	precision	recall	f1-score	support
Daphne	0.89	1.00	0.94	8
Fred	1.00	1.00	1.00	8
Scooby	0.60	0.50	0.55	6
Shaggy	0.29	0.50	0.36	4
Velma	1.00	0.25	0.40	4
accuracy			0.73	30
macro avg	0.75	0.65	0.65	30
weighted avg	0.80	0.73	0.73	30

Рисунок 12 – Метрики для 3 версий модели 2

Модель 3

Поскольку на графиках модели 2 наблюдается переобучение модель была упрощена. Также были попытки добавления других методов регуляризации, но они стабильно ухудшали результат (Рисунок 13).


```
Модель 3
class M3(nn.Module):
    def __init__(self):
        super(M3, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=2, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=2, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv7 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=2, padding=1)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(512*8*8, 256)
        # self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(256, 64)
        self.fc4 = nn.Linear(64, 5)

    def forward(self, x):

        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool1(x)

        x = F.relu(self.conv3(x))
        x = self.pool2(x)

        x = F.relu(self.conv5(x))
        x = self.pool3(x)

        x = F.relu(self.conv7(x))
        x = self.pool4(x)
        print(x.shape)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # x = F.relu(self.fc3(x))
        x = self.fc4(x)

        return x

model = M3()
model.to(device)
```

Рисунок 13 – Модель 3

Результаты 2 версий для этой архитектуры приведены на Рисунках 14-15.

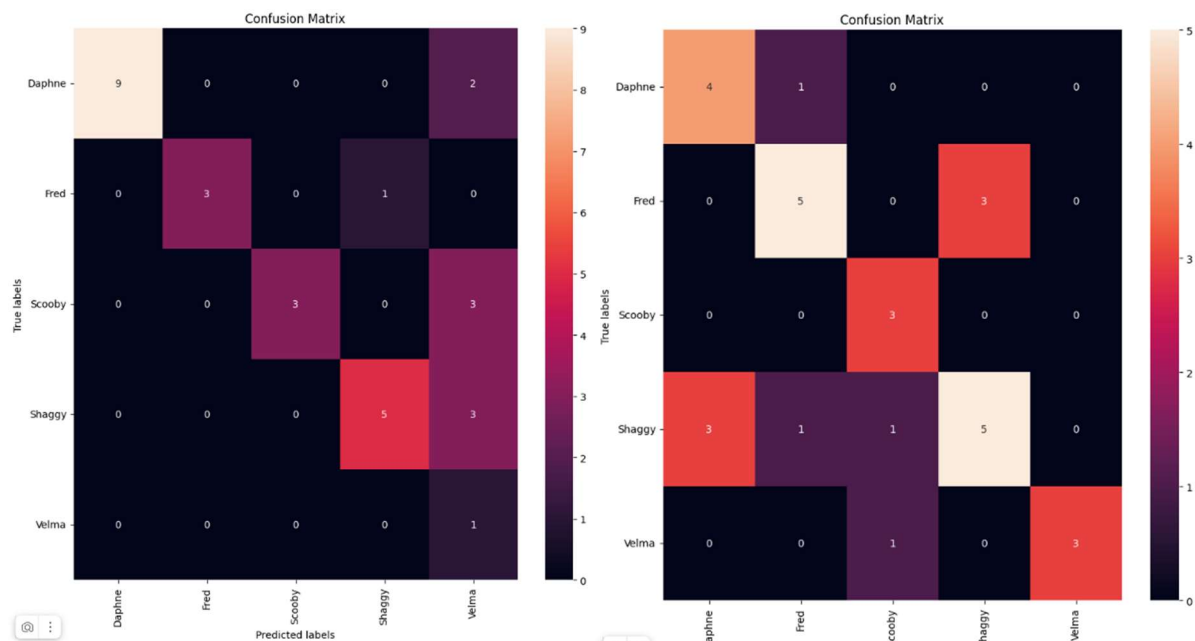


Рисунок 14 – Матрицы для 2 версий модели 3

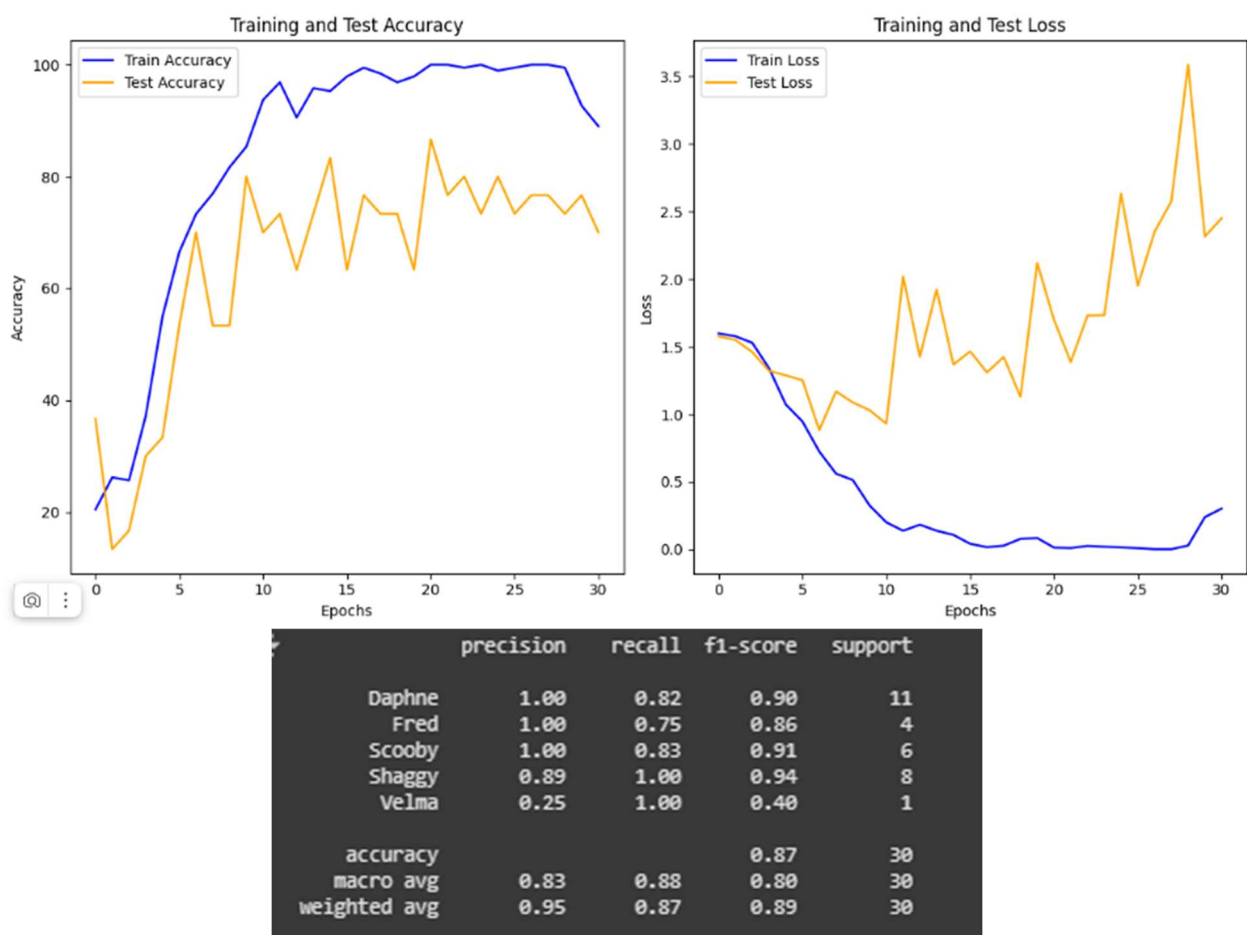


Рисунок 15 – Графики и результаты для версии 1 модели 3

Модель 4

Создал архитектуру модели 4 с немного изменённой конфигурацией слоёв (Рисунок 16).

```
Модель 4

class M4(nn.Module):
    def __init__(self):
        super(M4, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=2, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=2, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv7 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=2, padding=1)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(512*8*8, 1024)
        self.fc2 = nn.Linear(1024, 256)
        self.fc3 = nn.Linear(256, 64)
        self.fc4 = nn.Linear(64, 5)

    def forward(self, x):

        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool1(x)

        x = F.relu(self.conv3(x))
        x = self.pool2(x)

        x = F.relu(self.conv5(x))
        x = self.pool3(x)

        x = F.relu(self.conv7(x))
        x = self.pool4(x)
        print(x.shape)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)

        return x

model = M4()
model.to(device)
```

Рисунок 16 – Модель 4

Обучил модель SVM и предсказал значения на тестовой выборке (Рисунок 17).

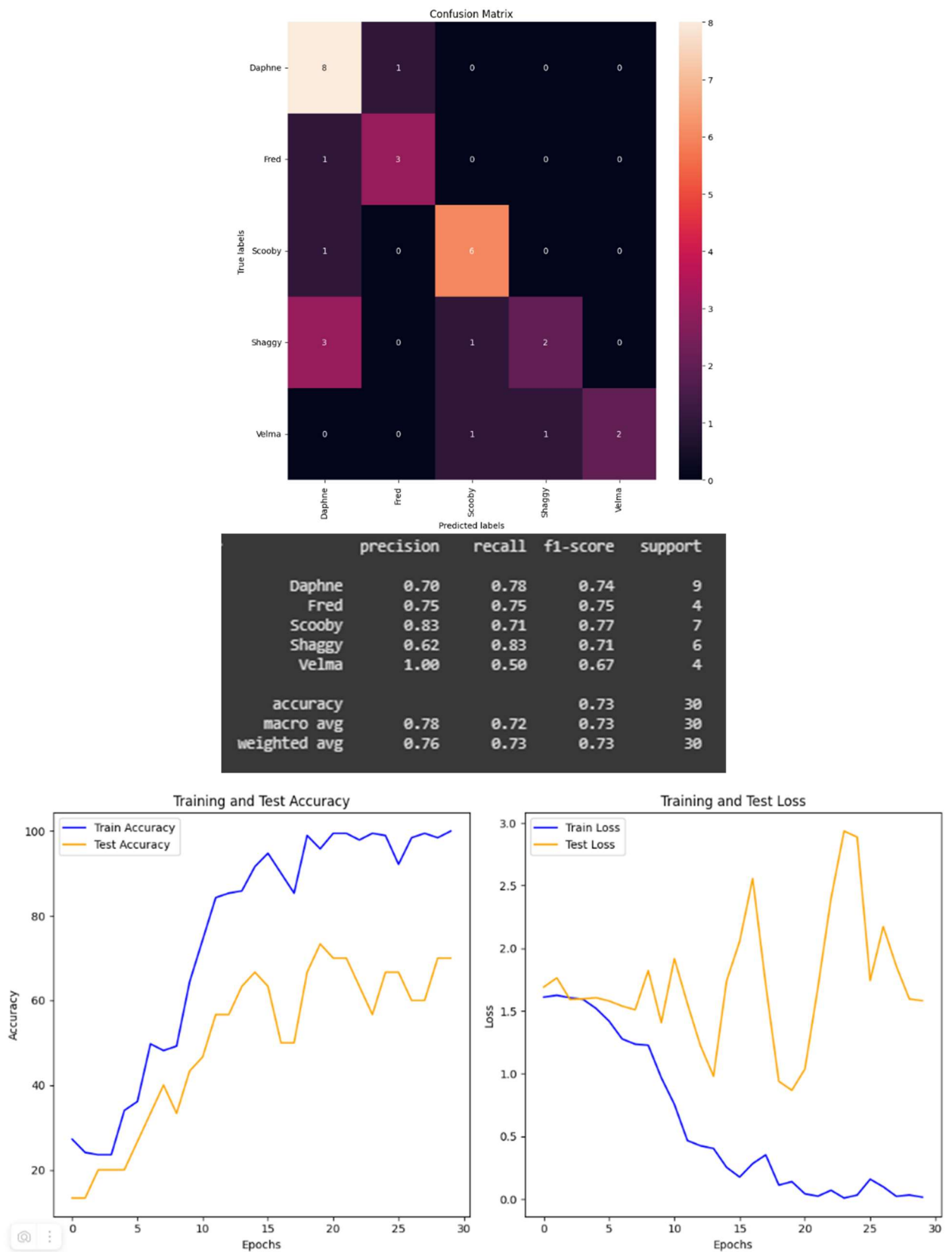


Рисунок 17 – Результаты модели 4

Модель ResNet

Настроил датасет под модель и заменил последний слой (Рисунок 18).

```
transform = T.Compose([ # для тр
    T.Resize((224, 224)),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

from torchvision import models

model = models.resnet18(pretrained=True)

model.fc = nn.Linear(model.fc.in_features, 5)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

Рисунок 18 – Настройка ResNet

Результаты дообученной модели можно видеть на Рисунке 19.

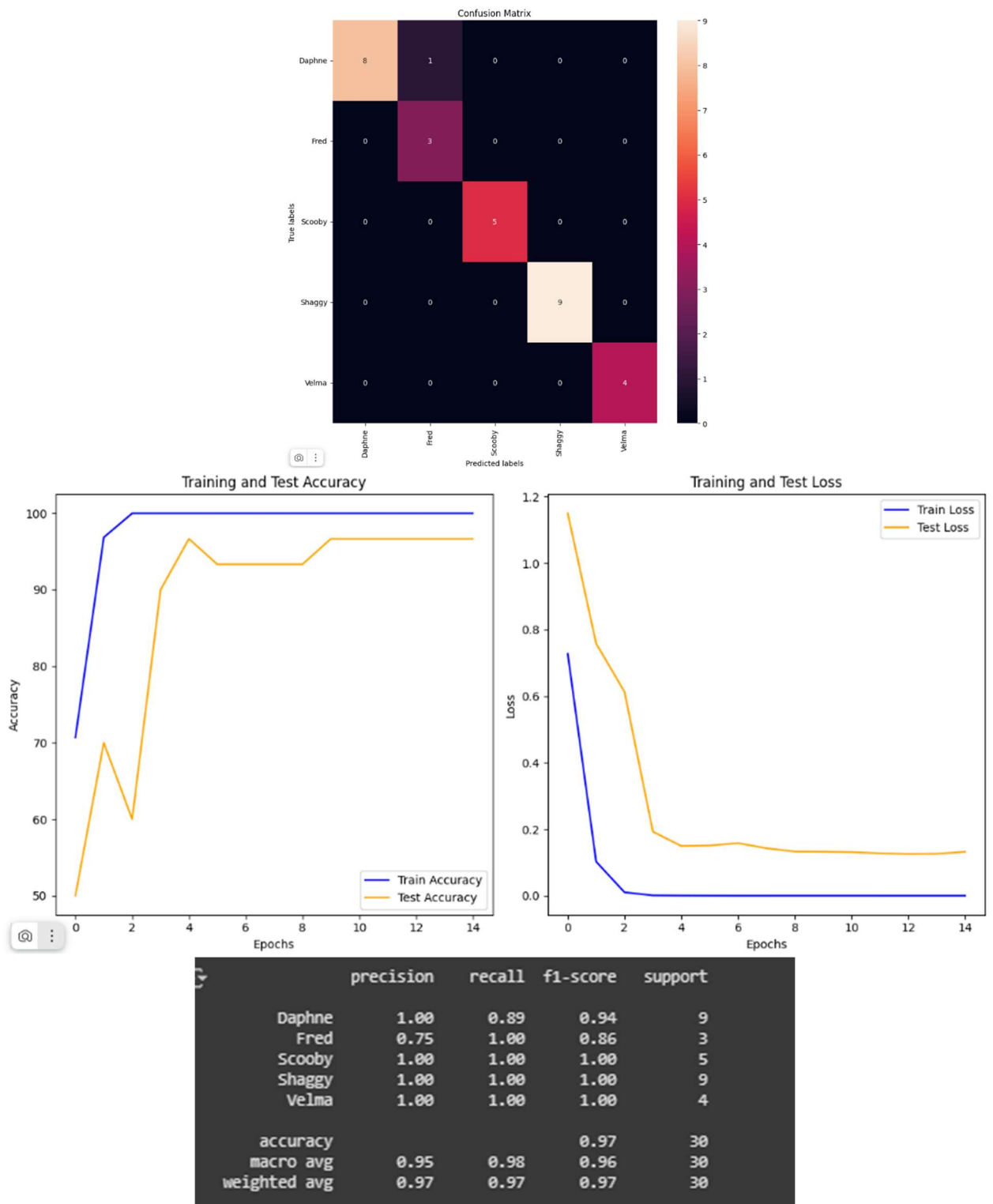


Рисунок 19 – Результаты модели ResNet

Создал диаграмму сравнения всех исследованных моделей (Рисунок 20).



Рисунок 20 – Сравнение всех моделей

Выводы

В рамках данной лабораторной работы была проведена предварительная обработка данных. Этот процесс включал расширение датасета, масштабирование, а также нормализацию, что способствовало повышению качества обучения моделей. В процессе было выявлено сильное негативное влияние расширения на результат обучения, что предполагает слабую устойчивость рассмотренных моделей к помехам.

На этапе создания архитектур были опробованы разные конфигурации слоёв. Была также предпринята попытка использования различных методов регуляризации, но положительного эффекта она не дала.

Оценка качества моделей проводилась с использованием метрик precision, recall и F1-score, что позволило оценить их точность и полноту в решении задачи классификации. Также для визуализации процесса и результата обучения использовались матрицы и графики точности и ошибок.

Наихудшие результаты показала модель 1, что очевидно т.к. она слишком простая для данной задачи.

Наилучшие результаты показала модель ResNet, что также очевидно. Из самописных архитектур наилучшие результаты показала модель 3.1 с точностью 0.87.

Для улучшения метрик всех моделей, можно увеличить количество данных (в использованном датасете было всего 221 изображение), провести тонкую настройку параметров для каждой из моделей. Однако это требует много времени и вычислительных ресурсов.

Приложения

Ссылка на блокнот в google.colab:

<https://colab.research.google.com/drive/1hF51fWuG2o38YQoTcVFEY70nmwtcUQaO?usp=sharing>