Deep Learning Homework 5

April 21, 2025

Аннотация

В этом отчете представлены результаты обучения нейронной сети для классификации изображений на кастомном датасете в рамках выполнения домашнего задания.

Содержание

1 Цели задания

Построить наиболее точную модель, способную классифицировать изображения.

2 Методология

2.1 Обзор

Для выполнения задания была обучена сверточная нейронная сеть для классификации изображений с использованием библиотеки PyTorch. Модель была обучена на кастомном датасете, было реализовано трансферное обучение с помощью предобученной модели EfficientNet-B4.

2.2 Инструменты

Обучение модели было полностью выполнено на языке Python и библотеки глубокого обучения PyTorch. В качестве среды разработки была выбрана веб-платформа Kaggle и аппаратный акселератор GPU T4 x2.

2.3 Данные

В рамках задания был использован кастомный набор данных, разделенный на тренировочную и тестовую выборки. Каждая состоит примерно из 16500 фотографий высокого качества, представляющих из себя 20 видов предметов моды.

2.4 Архитектура модели

В качестве базовой модели была выбрана предобученная сверточная нейронная сеть EfficientNet-B4, последний выходной слой был модифицирован так, чтобы модель могла предсказывать 20 классов.

2.5 Процесс обучения

Для получения наиболее эффективной модели и равномерного использования тренировочных данных обучение проходило с помощью перекрестной проверки (K-Fold Cross-Validation). В качестве оптимизатора был выбран Adam, а также уменьшение шага обучения с использованием ReduceLROnPlateau scheduler.

2.6 Процесс тестирования

Для получения финального предсказания были усреднены результаты предсказаний всех k моделей, полученных во время обучения с перекрестной валидацией.

3 Обзор данных

3.1 Описание тренировочных данных

Набор тренировочных данных состоит из 16575 фотографий 20 различных видов элементов моды. Каждое изображение находится в формате RGB высокого разрешения с соответствующем ему классу. Почти все изображения имеют разрешение 1800×1440 или 1800×2400, а также незначительное количество иных разрешений. Также данные по классам сильно несбалансированы.

3.2 Сводка метаданных

Атрибут	**Значение**
Общее количество изображений	16,575
Количество классов	20
Разрешение	1800×1440, 1800×2400
Общий размер	2.65 GB
Распределение по классам	Сильно несбалансированное (see Fig 1)

Таблица 1: Метаданные

3.3 Распределение по классам

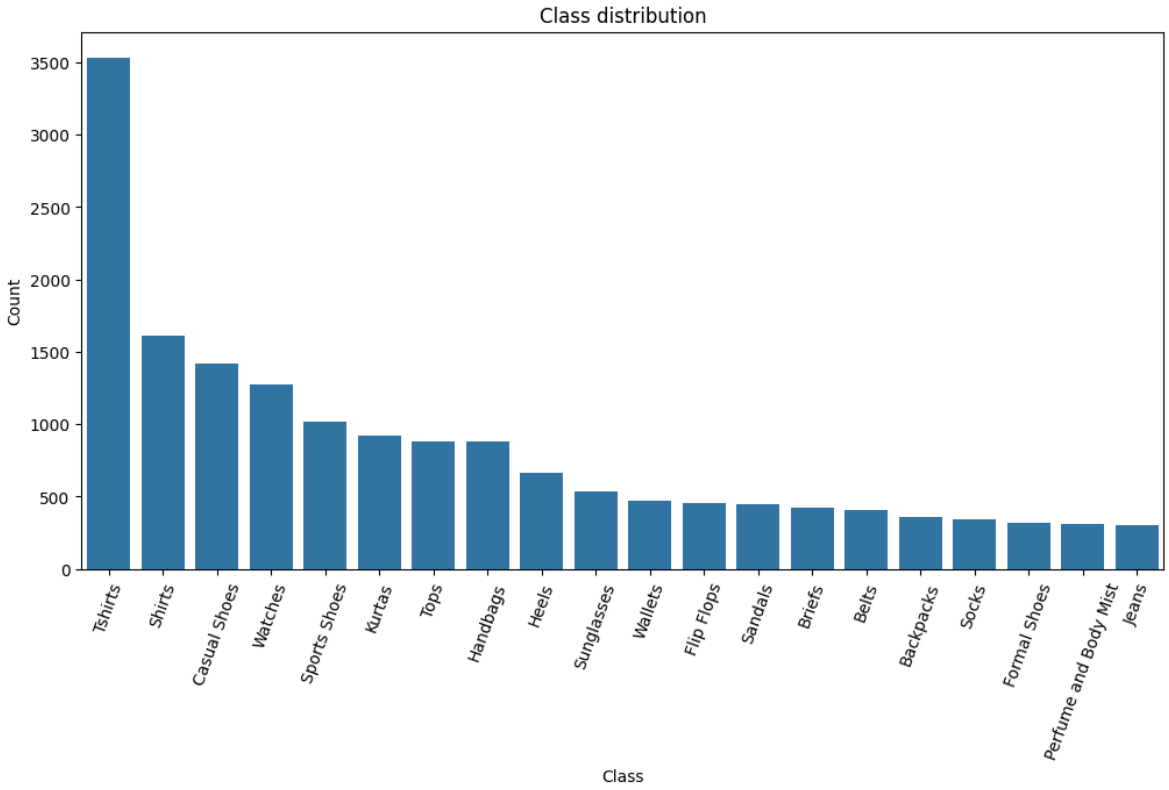

Рис. 1: Class distribution in the dataset.

3.4 Препроцессинг

Особенности тренировочного датасета потребовали применения определенных трансформаций к каждому экземпляру изображения. Также стоит уделить особое внимание распределению по классам, сильная несбалансированность которого может привести к тому, что модель будет учиться классифицировать только классы с наибольшим количеством экземпляров.
Чтобы учесть все особенности данных, во время препроцессинга были применены следующие трансформации к каждому изображению:

- Изменение разрешения до 224×224 пикселей.
- Нормализация mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]
- Приведение в тензор для работы на GPU

Также, для улучшения способности модели обобщать признаки, была применена техника аугментации данных:

- Случайный горизонтальный поворот
- Случайный поворот изображения на 10 градусов
- Случайное изменение параметров изображения (яркость, контрастность, насыщение)

Для решения проблемы несбалансированности классов была применена техника взвешенного сэмплирования (Weighted Sampling), которая увеличивала вероятность того, что экземпляры изображений, принадлежащих наимешьшим классам, попадут в тренировочную выборку.

4 Реализация

4.1 Псевдокод

Здесь представлена общая картина обучения на примере псевдокода

4.1.1 k-Fold Cross Validation

Главный цикл, реализующий перекрестную валидацию. Для каждого k-fold инициализируются модель, наборы тренировочной и валидационной данных, а также фунция потерь. Затем запускается цикл обучения. В реальном коде также инициализируется шаг оптимизатора.

Algorithm 1 k-Fold Cross-Validation Pipeline

Require: Dataset D, number of folds k

Split dataset D into k folds: $D_1, D_2, \ldots, D_k$
for fold $= 1$ to k **do**
    Initialize training subset $D_{\text{train}} = D \setminus D_{\text{fold}}$
    Initialize validation subset $D_{\text{val}} = D_{\text{fold}}$
    Apply transformation to training subset D_{train}
    Apply transformation to training subset D_{val}
    Initialize model M_{fold} with pre-trained weights
    Initialize loss function L (e.g., CrossEntropyLoss)
    Call `Train` function:
        `Train`$(M_{\text{fold}}, D_{\text{train}}, D_{\text{val}}, L)$
end for

4.1.2 Training Loop

Цикл обучения. В каждой эпохе модель обучается на тренировочной выборке, затем используется валидационная выборка для проверки модели. Если значение потери на валидационной выборке не улучшается какое-то заданное количество раз, то цикл останавливается, модель сохраняется и управление передается обратно функции кросс-валидации (Ранняя остановка). Таким образом, в самом конце обучения получится k моделей, каждая из которых обучена на разных наборах тренировочной и валидационных выборках.

Algorithm 2 Training Pipeline (Train function)

Require: Model M, training data D_{train}, validation data D_{val}, loss function L, number of epochs E, learning rate η
  Initialize best validation loss $L_{\text{best}} \leftarrow \infty$
  for epoch $= 1$ to E **do**
    Set model M to training mode
    for each batch $(X_{\text{train}}, y_{\text{train}})$ in D_{train} **do**
      Compute predictions $\hat{y}_{\text{train}} = M(X_{\text{train}})$
      Compute loss $L_{\text{batch}} = L(\hat{y}_{\text{train}}, y_{\text{train}})$
      Compute gradients ∇L_{batch}
      Update model parameters: $M \leftarrow M - \eta \nabla L_{\text{batch}}$
    end for
    Set model M to evaluation mode
    Initialize running validation loss $L_{\text{val}} = 0$
    for each batch $(X_{\text{val}}, y_{\text{val}})$ in D_{val} **do**
      Compute predictions $\hat{y}_{\text{val}} = M(X_{\text{val}})$
      Compute loss $L_{\text{batch}} = L(\hat{y}_{\text{val}}, y_{\text{val}})$
      Accumulate validation loss: $L_{\text{val}} \leftarrow L_{\text{val}} + L_{\text{batch}}$
    end for
    if $L_{\text{val}} < L_{\text{best}}$ **then**
      $L_{\text{best}} \leftarrow L_{\text{loss}}$
    else
      break
    end if
  end for
  Save trained model M to disk

4.1.3 Final Prediction

Вычисление финального предсказания на тестовых данных. Для этого каждая ранее обученная модель вычисляет логиты, которые затем суммируются и их среднее используется для получения предсказания.

Algorithm 3 Final Prediction Using Averaging

Require: Trained models $M_1, M_2, \ldots, M_k$, test dataset D_{test}
Ensure: Final predictions P_{final}
  Initialize empty list $P_{\text{all}} = []$ {To store predictions from all models}
  for each model M_i in $M_1, M_2, \ldots, M_k$ **do**
    Set model M_i to evaluation mode
    Initialize empty list $P_i = []$ {To store predictions from model M_i}
    for each batch (X_{test}) in D_{test} **do**
      Compute predictions $\hat{y}_{\text{test}} = M_i(X_{\text{test}})$
      Append $\hat{y}_{\text{test}}$ to P_i
    end for
    Append P_i to P_{all}
  end for
  Compute final predictions P_{final} by averaging predictions across all models:
  $P_{\text{final}} = \frac{1}{k} \sum_{i=1}^{k} P_{\text{all}}[i]$
  return P_{final}

4.2 Реальный код

В данной секции представлен обзор ключевых компонентов реализации обучения:

- Гиперпараметры
- Загрузка и препроцессинг данных
- Инициализация модели глубокого обучения
- Цикл обучения с кросс-валидацией
- Финальное предсказание

Исходный код, использованный для обучения, доступен по ссылке:
Kaggle
или
GitHub

4.2.1 Гиперпараметры

Данные гиперпараметры инициализируются в самом начале программы и не меняются в течение всего обучения. Шаг оптимизатора корректируется в тренировочном цикле с помощью scheduler.

```
BATCH_SIZE = 64
NUM_EPOCHS = 100
PATIENCE = 10
NUM_CLASSES = 20
LEARNING_RATE = 1e-4
```

Данная переменная не является гиперпараметром, однако необходима на протяжении работы всей программы для аллоцирования тензоров на соответствующем аппаратном обеспечении.

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

4.2.2 Данные

Кастомный класс `Dataset`, необходимый для загрузки данных. На вход принимает мета данные с разметкой `img_labels`, путь к директории где хранятся сырые данные `img_dir`, а также трансформации `transform`, которые необходимо к ним применить.

```
class FashionDataset(Dataset):
    def __init__(self, df_meta: pd.DataFrame, img_dir: str, transform:
    ↪  Callable=None):
        super().__init__()
        self.img_labels = df_meta
        self.img_dir = img_dir
        self.transform = transform

    def __getitem__(self, idx: int) -> (torch.tensor, torch.tensor):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx,
        ↪  0])
        image = Image.open(img_path)
        label = self.img_labels.iloc[idx, 2]

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

        return image, label


    def __len__(self) -> int:
        return len(self.img_labels)
```

Трансформации, которые были применены к тренировочным данным:

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    ),
])
```

К валидационным данным также применялись трансформации, но без аугментации:

```
val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

4.2.3 Инициализация модели

Данная функция использовалась для инициализации предобученной модели. Модель инициализируется в переменную `model`, затем модель модель модифицируется так, чтобы классифицировать 20 классов: `model.classifier[1] = nn.Linear(...)`.
Чтобы эффективно использовать доступные ресурсы для обучения, модель была обернута в класс `nn.DataParallel()`, который позволяет распараллелить вычисления и обучать модель на двух доступных видеокартах.

```
def create_model():
    model = models.efficientnet_b4(
        weights=models.EfficientNet_B4_Weights.IMAGENET1K_V1
    )
    model.classifier[1] = nn.Linear(model.classifier[1].in_features,
    ↪  NUM_CLASSES)

    if torch.cuda.device_count() > 1:
        model = nn.DataParallel(model)
    model.to(device)

    return model
```

4.2.4 Цикл кросс-валидации

Здесь представлен главный цикл, реализующий k-fold cross-validation. Некоторые детали опущены для читаемости. После завершения работы получается k обученных моделей

```
k = 5
kfold = KFold(n_splits=k, shuffle=True, random_state=42)

fold_results = []
for fold, (train_ids, val_ids) in enumerate(kfold.split(dataset)):
    train_subset = Subset(dataset, train_ids)
    val_subset = Subset(dataset, val_ids)

    train_subset.dataset.transform = train_transform
    val_subset.dataset.transform = val_transform

    train_labels = [label for _, label in train_subset]
    class_counts = torch.bincount(torch.tensor(train_labels))
    class_weights = 1. / class_counts
    class_weights = class_weights / class_weights.sum()
    sampler = WeightedRandomSampler(...)

    model = create_model()
    train_dataloader = DataLoader(train_subset, sampler=sampler, ...)
    val_dataloader = DataLoader(val_subset, sampler=sampler, ...)
    criterion = nn.CrossEntropyLoss(weight=class_weights.to(device))
    optimizer = torch.optim.Adam(model.parameters(), ...)
    scheduler = ReduceLROnPlateau(optimizer, ...)

    train(
        model,
        train_dataloader,
        val_dataloader,
        criterion,
        optimizer,
        scheduler,
        fold
    )
```

Ниже находится более подробное описание компонентов данного цикла.

Для начала в цикле инициализируются наборы тренировочных и валидационных данных, затем к ним применяются соответствующие трансформации.

```
train_subset = Subset(dataset, train_ids)
val_subset = Subset(dataset, val_ids)

train_subset.dataset.transform = train_transform
val_subset.dataset.transform = val_transform
```

Далее создается класс `WeightedRandomSampler`, чтобы учесть несбалансированность распределения классов. Для этого подсчитывается количество классов для каждого тренировочного датасета отдельно в каждом k-fold, затем инициализуются веса классов. Данный сэмлер (`sampler`) затем будет передан в класс `DataLoader` тренировочного датасета.

```
train_labels = [label for _, label in train_subset]
class_counts = torch.bincount(torch.tensor(train_labels))
class_weights = 1. / class_counts
class_weights = class_weights / class_weights.sum()
sampler = WeightedRandomSampler(sample_weights,
↪  num_samples=len(sample_weights), replacement=True)
```

В самом конце инициализуются модель, загрузчики данных, функция потерь, оптимизатор и scheduler, которые передаются как аргументы в функцию обучения `train`

```
model = create_model()
train_dataloader = DataLoader(
        train_subset,
        batch_size=BATCH_SIZE,
        sampler=sampler
    )
val_dataloader = DataLoader(
        val_subset,
        batch_size=BATCH_SIZE,
        shuffle=False
    )
criterion = nn.CrossEntropyLoss(weight=class_weights.to(device))
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
scheduler = ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.1,
    patience=5,
    verbose=True
)
train(model, train_dataloader, val_dataloader, criterion, optimizer,
↪  scheduler, fold)
```

4.2.5 Цикл обучения

Данная функция реализует обучение модели. Обучение длится `NUM_EPOCHS` эпох. Если модель не улучшается `patience` раз на валидационных данных, то обучение модели останавливается. Если модель учучшилась на валидационной выборке, то ее состояние сохранятся (`torch.save(...)`). В итоге после выполнения функции получается одна обученная модель. Всего данная функция вызывается k раз.

```
def train(model, train_dataloader, val_dataloader, criterion, optimizer,
↪  scheduler, fold):
    best_val_loss = float("inf")
    patience = 10

    for epoch in range(NUM_EPOCHS):
        model.train()
        running_loss = 0.0
        for images, labels in train_dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for images, labels in val_dataloader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

        train_loss = running_loss / len(train_dataloader)
        val_loss = val_loss / len(val_dataloader)

        scheduler.step(val_loss)

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), ...)
            patience = 10
        else:
            patience -= 1
            if patience == 0:
                print("Early stopping")
                break
```

4.2.6 Финальное предсказание

После обучения все k моделей используются для получения предсказания. Логиты всех моделей суммируются и усредняются.

Для начала инициализируются пустой список `model_to_test`, куда будут будут добавлены k моделей. Для инициализации моделей используется вышеописанная функция `create_model()`, затем в модель загружается `checkpoint`, который был сохранен во время обучения. После, модель переводится в режим инференса с помощью `model.eval()` и добавляется в список.

```
models_to_test = []
for i in range(k):
    model = create_model()
    checkpoint =
    ↪  torch.load(f"/kaggle/working/efficientnet_b4-fold-{i+1}.pth")
    model.load_state_dict(checkpoint)
    model.eval()
    models_to_test.append(model)
```

После, данный список с обученными моделями используется для получения предсказания:

```
predictions = []
with torch.no_grad():
    for images in test_dataloader:
        images = images.to(device)
        outputs0 = models_to_test[0](images)
        outputs1 = models_to_test[1](images)
        outputs2 = models_to_test[2](images)
        outputs3 = models_to_test[3](images)
        outputs4 = models_to_test[4](images)

        avg_outputs = (outputs0 + outputs1 + outputs2 + outputs3 +
        ↪  outputs4) / k
        _, predicted = torch.max(avg_outputs, 1)
        predictions.extend(predicted.cpu().numpy())
```

5 Результаты

5.1 Обзор

В результате обучения было получено 5 моделей, каждая из которых была обучена на разных выборках тренировочных данных благодаря кросс-валидации. Итоговая точность на тестовых данных составила 92.61%, показав таким образом неплохую эффективность в задаче классификации изображений. Функции потерь на тренировочных данных у каждой модели плавно сходились к нулю в течение всех эпох, однако валидационная потеря каждой модели переставала улучшаться примерно после 7-8 эпохи, что говорит о переобучении. Также, по графику 2 можно увидеть, что после шага оптимизатора ни одна модель не начала улучшаться на валидационных данных.

5.2 Кривые функций потерь на тренировочных и валидационных данных

На графике ниже показаны кривые потерь на тренировочных и валидационных данных. Красными вертикальными линиями показаны шаги оптимизаторов для каждой модели.

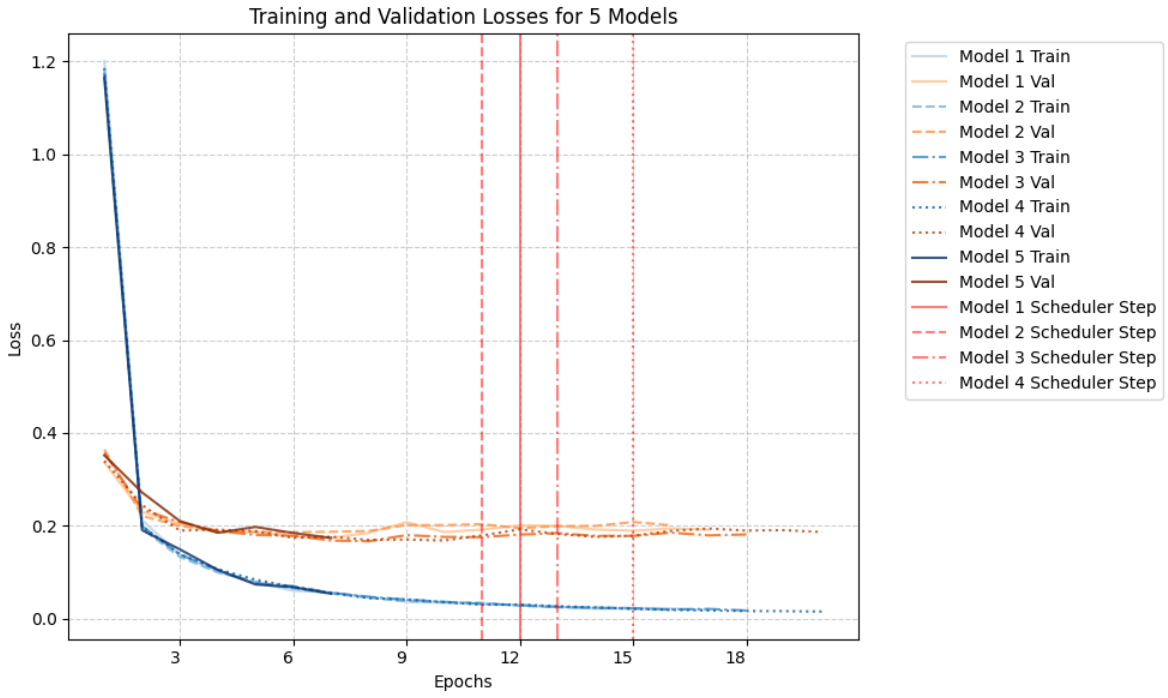

Рис. 2: Значения функций потерь на тренировочных и валидационных данных

5.3 Матрица несоответствий

Чтобы оценить точность модели, была построена матрица несоответствий. Для этого была использована часть данных от всего тренировочного датасета. По матрице видно, что больше всего модель путает топы (Tops) и футболки (Tshirts), а также разные виды обуви, например, повседневную (Casual Shoes) и спортивную (Sports Shoes) обувь.

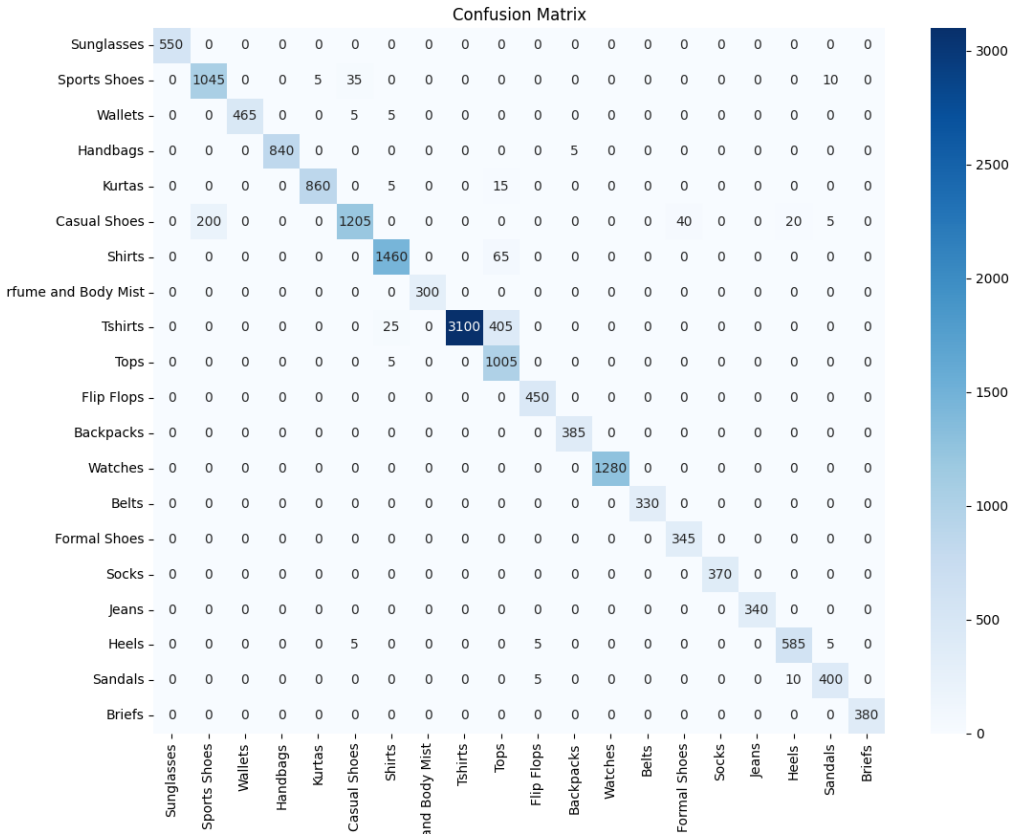

Рис. 3: Матрица несоответствий

6 Предложения по улучшению модели

Первое, на что стоит обратить внимание, это особенность фунции потерть на валидационных данных. Каждая модель переставала улучашаться уже после 7-8 эпохи, даже несмотря на шаг оптмизатора, который не помог ни одной модели улучшиться. Для улучшения ситуации можно предпринять следующие шаги:

- Сменить шаг оптимизатора на другой.
- Подкорректировать batch size и learning rate.
- Усилить аугментацию данных чтобы еще улучшить способности модели к обобщению.
- Увеличить параметр `patience`, чтобы дольше обучаться.

Несмотря на то, что были предприняты меры по аугментации данных и сэмплированию для выравнивания классов, модель все равно испытывает проблемы с классификацией классов, похожих друг на друга, такими как Formal Shoes и Casual Shous или TShirts и Tops. Для этого можно попробовать:

- Использовать более глубокую модель (Например, EfficientNet-B5 или -B6).
- Во время сэмплирования добавить большие веса представителям похожих классов.
- Threshold Tuning. Уделить внимание трешхолдам проблемных классов. Например, увеличить их, чтобы сильнее штрафовать модель за ошибки.