Artificial Vision and Pattern Recognition

Assignment #2: Action Recognition From Still Images Using Deep Learning Networks

Mark Safrónov

MESIIA 2024-2025

The task

Given the dataset in [Stanford40], we need to build two image classifiers for the image classes mentioned in this dataset.

One classifier should be a custom-build CNN network, second classifier should be based on a pretrained model like ResNet, Googlenet, VGG, MobileNet.

The solution will be performed in Python 3.12 using the PyTorch libraries [PyTorch].

Data preparation

We transform the original dataset into the format readable by the built-in methods from Pytorch, namely, the torchvision.datasets.ImageFolder[1] class.

Train-test split

The original dataset already includes the suggested train/test split. However, it's a bit unconventional, as there's exactly 100 train images for each image class, and arbitrary amount of test ones, sometimes less than that, sometimes a lot more. So, **we prepare two datasets out of the original one**.

The first dataset will be called "given splits", and it will be split to train/test data according to the suggestions from the Stanford paper.

The second dataset will contain images split to train and test datasets at random with the classic 80:20 split.

A separate Python script has been used to transform the original flat dataset into two distinct datasets, physically reorganizing files into folders according to the convention assumed by the ImageFolder dataset loader from PyTorch.

Images normalization

Preliminary analysis of the images in the dataset uncovered the following statistics:

1. Max Width: 997

1 https://pytorch.org/vision/main/generated/torchvision.datasets.ImageFolder.html

2. Max Height: 965
3. Min Width: 200
4. Min Height: 200
5. Average Width: 424.82364666386906,
6. Average Height: 396.36372219890893
7. Median Width: 400.0
8. Median Height: 400.0
9. Mean RGB values: [119.40268332, 112.331025, 102.14789407],
10. Standard deviation of RGB values: [63.13785612, 61.05932064, 61.90802032]

We have surprisingly consistent median for the sizes so this will be the basis for resizing the images. We will resize/crop the images to 400×400 pixels in size.

The mean and the standard deviation will be used for normalizing the color channels. The data above is the raw value, on the scale of 0..255, we'll need to convert it to the range 0..1, which gives us the following normalization coefficients:

1. Mean: [0.46824582, 0.44051382, 0.40057998]
2. Std: [0.24759944, 0.23944832, 0.24277655]

It is very close to the standard ImageNet normalization coefficients from [ImageNet], but is different enough (around 10% difference) to use it instead of ImageNet ones.

Final exploration variants space

The goal of this exercise is to try the full process of building, training and evaluating the image classifier, so I will concentrate on exploring different variants of data flows and not on fine-tuning one of them in searching of the best classification accuracy.

We'll, thus, check the following options in configuration:

1. Different splits:
    1. Given split from Stanford
    2. Random split 80:20
2. Model type:
    1. Custom CNN
    2. Pre-trained model modified for our amount of classes (40 actions)
3. Data augmentation:
    1. Testing only on the given train images
    2. On each training epoch, perform random modifications on the training dataset images

CNN model definition

Construction of the CNN was tool-assisted. The OpenAI's ChatGPT 4o has been used ([ChatGPT]).

The overall flowchart of the CNN is presented on the image to the right.

As the median size of the images in the given dataset is 400×400, it has been decided that our CNN will be tailored for the input of this size. This is almost four times larger than the classic 224×224 of the [ImageNet] so there's hope that it will influence the accuracy positively.

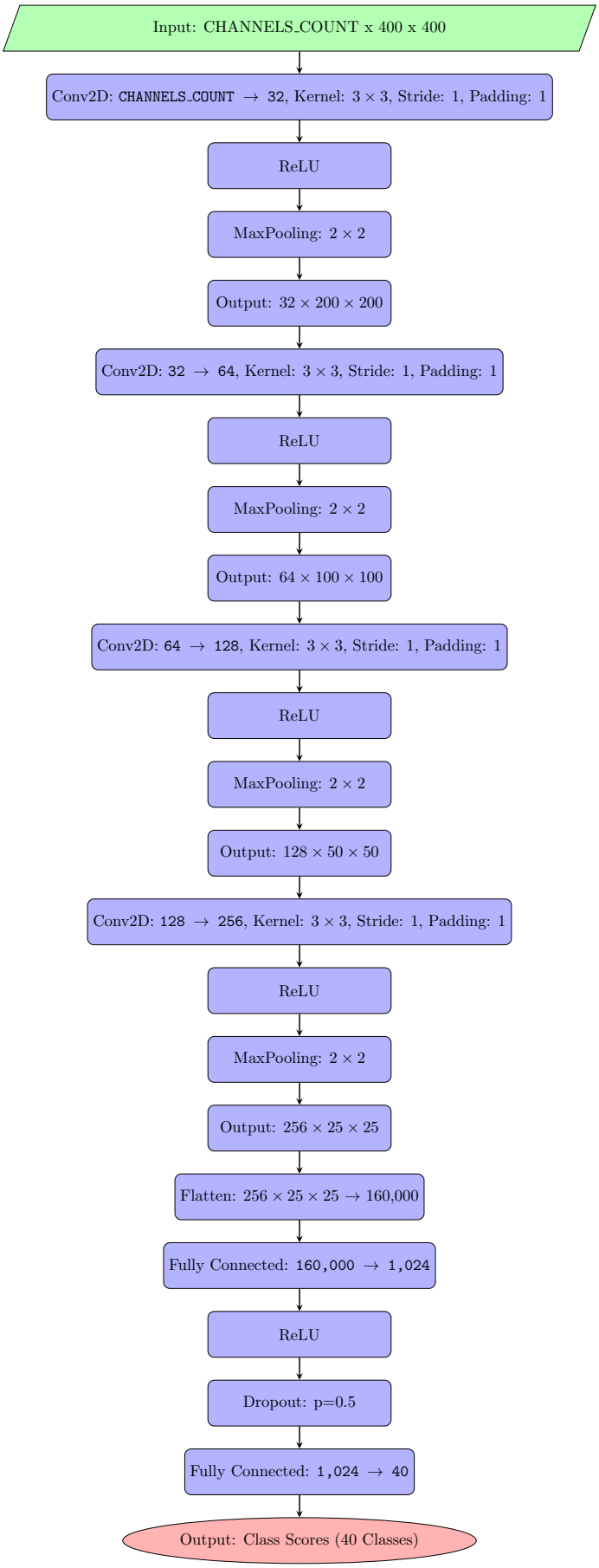
Thus, the network we will use is constructed like that:

1. We repeat four convolutions with kernel size 3, stride 1 and padding 1, using ReLU as activation.
2. On each convolution we maxpool 2×2 reducing the size of the input fourfold.

At the end of convolutions, we end up with 256 feature maps of size 25×25. We flatten them in a feature vector of 160 000 elements and pass it into the fully connected hidden layer with 1024 outputs. Activation function here is ReLU again, and in addition to that we perform a random activation dropout of 50% to prevent overfitting.

Resulting activations are passed to the output layer which transforms 1024 features into 40 classes we are looking for.

The following are the summary given by 4o about its design decisions:



Why This Architecture?

- **Stacked Convolutions (3x3 Kernels):** Borrowed from VGGNet, this allows hierarchical feature extraction.
- **Pooling for Downsampling:** Reduces spatial dimensions to make the model computationally efficient while retaining essential information.

- **Increasing Filters:** Inspired by both VGGNet and ResNet, progressively increasing the number of filters allows the model to capture more complex features at deeper layers.
- **Fully Connected Layers:** Common in traditional architectures for final classification, ensuring global feature aggregation.
- **Dropout:** Reduces overfitting, as discussed in AlexNet and other research.

End quote.

Indeed, if we look at the original [VGGNet] and [ResNet] papers we see that our custom CNN is essentially the scaled down variant of VGGNet. VGGNet uses five convolution-maxpooling stages, while we use four. The most glaring difference is that at the end VGGNet has three fully connected layers while we have two and they don't have a 50% dropout step.

Dropout was clearly taken directly from [AlexNet] section 5.2.

It should be noted also that both [AlexNet] and [VGGNet] use two fully connected hidden layers behind the output fully connected layer, while ChatGPT suggested using only one. For reference, [ResNet] don't have any. Also we don't have a "shortcuts" between the convolution layers which are the staple of [ResNet] and its core points.

The custom CNN definition in PyTorch is presented in the Appendix A. Custom CNN.

Pretrained model definition

ResNet-18 (18-layer ResNet outlined in Table 1 of [ResNet]) has been chosen as a tradeoff between the computational requirements and the accuracy of predictions. We'll implement transfer learning on this model [TransL]. The only changes to this model would be changes in the output layer – as the original ResNet was ImageNet-style, it has 1000 outputs, while we need only 40. So, we replace the 1000-element fully connected layer at the end with the 40-element one.

As ResNet is an ImageNet-based architecture, we will resize the input images to 224×224 px size.

Data augmentation

We will implement the data augmentation techniques to check whether they influence the accuracy of the classification positively. Theoretically, according to [AlexNet], for example, they definitely improve the training process as data augmentation essentially provides new training examples "for free".

The following augmentation steps were used:

1. Random horizontal flip[2] (with 0.5 probability)
2. Random rotation[3] (from -10 to +10 degrees)
3. Color jitter[4] (up to 20% changes in both directions for brightness, contrast, saturation and hue).

2 https://pytorch.org/vision/stable/generated/torchvision.transforms.RandomHorizontalFlip.html
3 https://pytorch.org/vision/stable/generated/torchvision.transforms.RandomRotation.html
4 https://pytorch.org/vision/stable/generated/torchvision.transforms.ColorJitter.htm

Model validation protocol

The following protocol has been used for testing and validating all the configurations of data and neural networks.

1. Determine whether we work on CUDA or on CPU.
2. Make the CNN under test (either our custom architecture or ResNet-18)
3. If there is the saved checkpoint with weights for this model, load it and skip the training by going to the pt. 6.
4. Load the training dataset using the standard PyTorch machinery.
5. Train using the Adam optimizer[5] (as per [Adam]) using the cross entropy loss function[6]. Use weight decay of 0.0005 and learning rate of 0.001
6. Load the testing dataset using the standard PyTorch machinery.
7. If the data augmentation has been used for training, disable it for testing.
8. Predict the labels for the test dataset.
9. Calculate the accuracy by comparing the amount of matches (correct predictions) with the total size of the test dataset.
10. Calculate the confusion matrix for all 40 labels.

Exploration of the parameters hyperspace was not the goal of this exercise (comparison of CNN architectures was) so the same learning rate was used for all the cases.

All computations has been performed on CUDA as it was available and it has no influence on the accuracy of the models. The GPU used was NVIDIA RTX 2060. This was the only limiting factor why the ResNet-18 was chosen instead of ResNet-101 which has 22% better results according to [ResNet] (21.75% error rate compared to 27.88% for ResNet-18).

5 https://pytorch.org/docs/stable/generated/torch.optim.Adam.html
6 https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

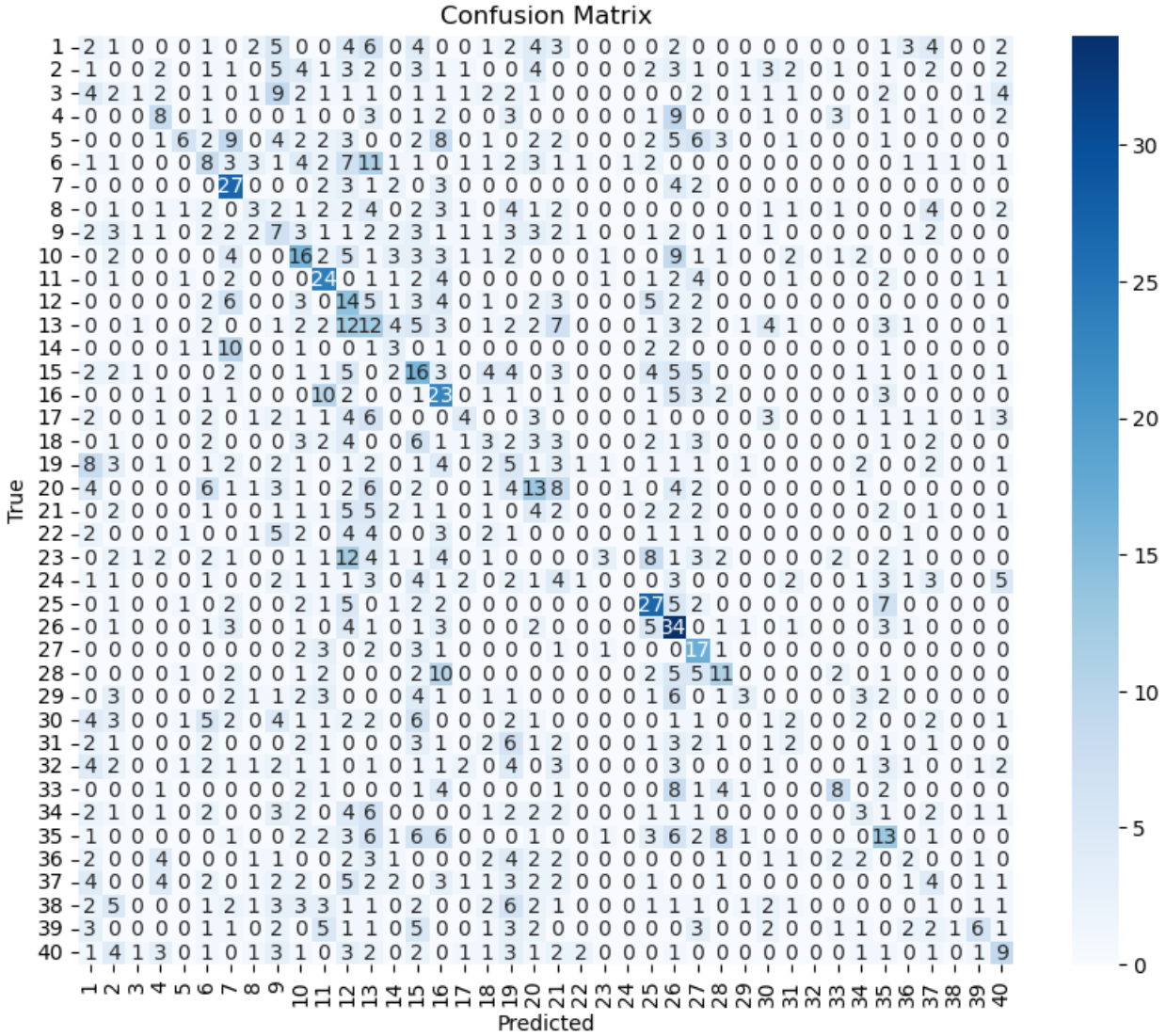
Results analysis

Below are the results of the running the various configurations of the system on the given dataset using PyTorch.

Custom CNN raw augmented images random split

The “best case” for the custom CNN of our architecture:

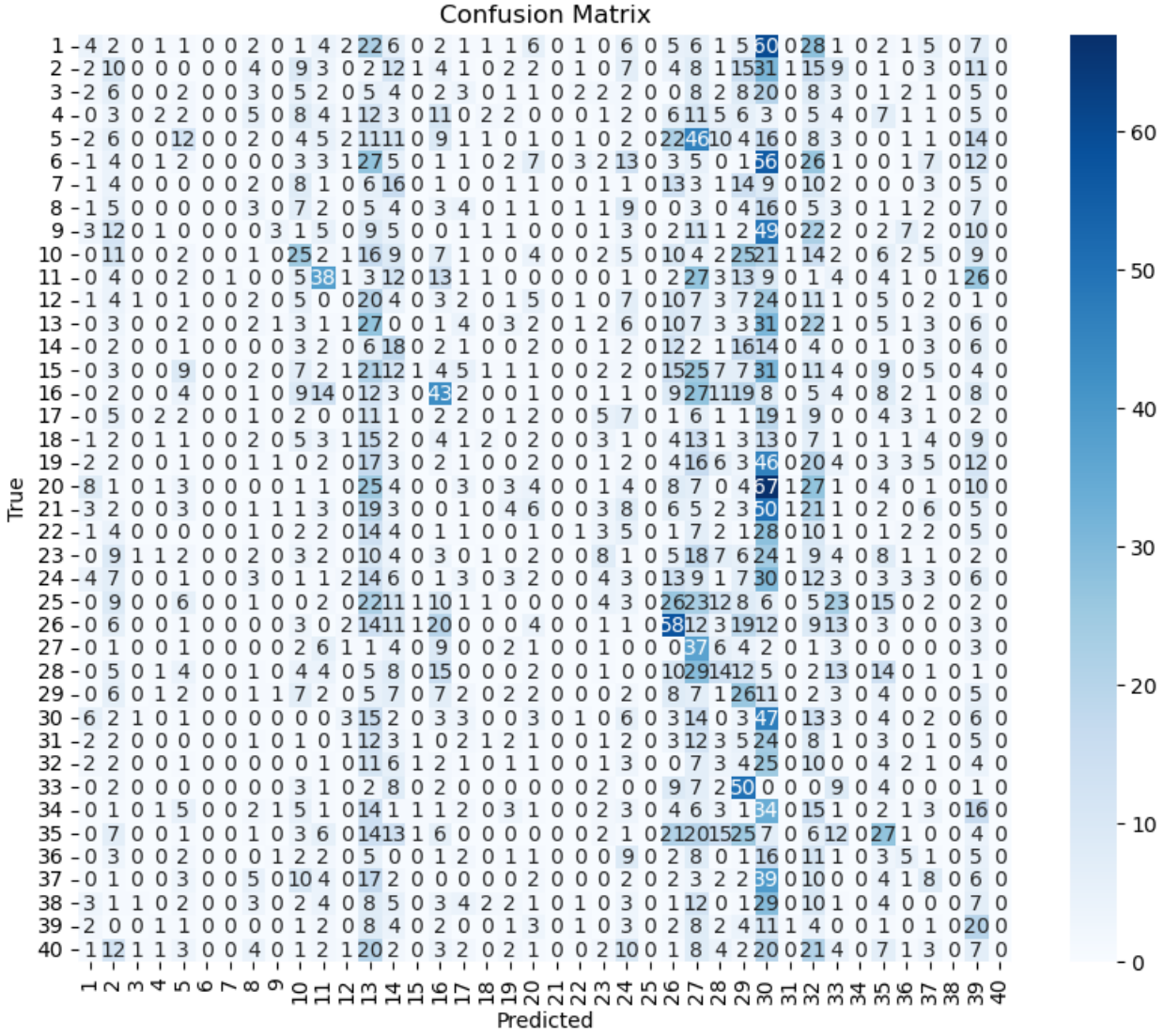
1. data augmentation enabled
2. used random 80:20 train-test split



Accuracy of the model on the test images: 18.24%

Custom CNN raw images given split

The baseline comparison with the split explicitly suggested by [Stanford40]. In this variant, we don’t use data augmentation, essentially training *only* on the images suggested by that paper.

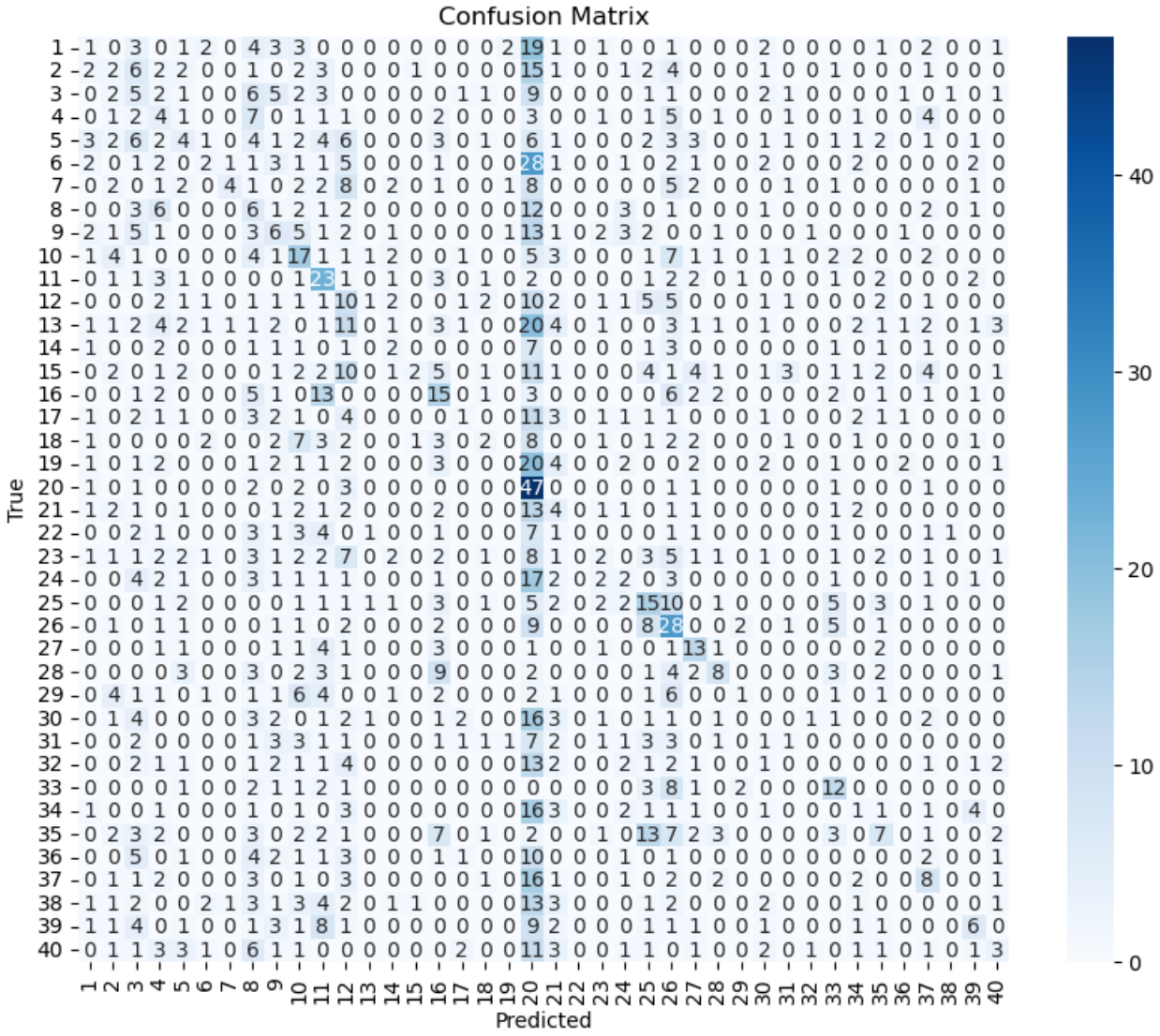


Accuracy of the model on the test images: 8.44%

As can be seen, given splits result in a significant bias, which is possibly results from the relatively small amount of training images (100 per each class).

Custom CNN raw images random split

For reference, we train with data augmentation disabled but using the random 80:20 train-test split instead of the suggestions from [Stanford40].



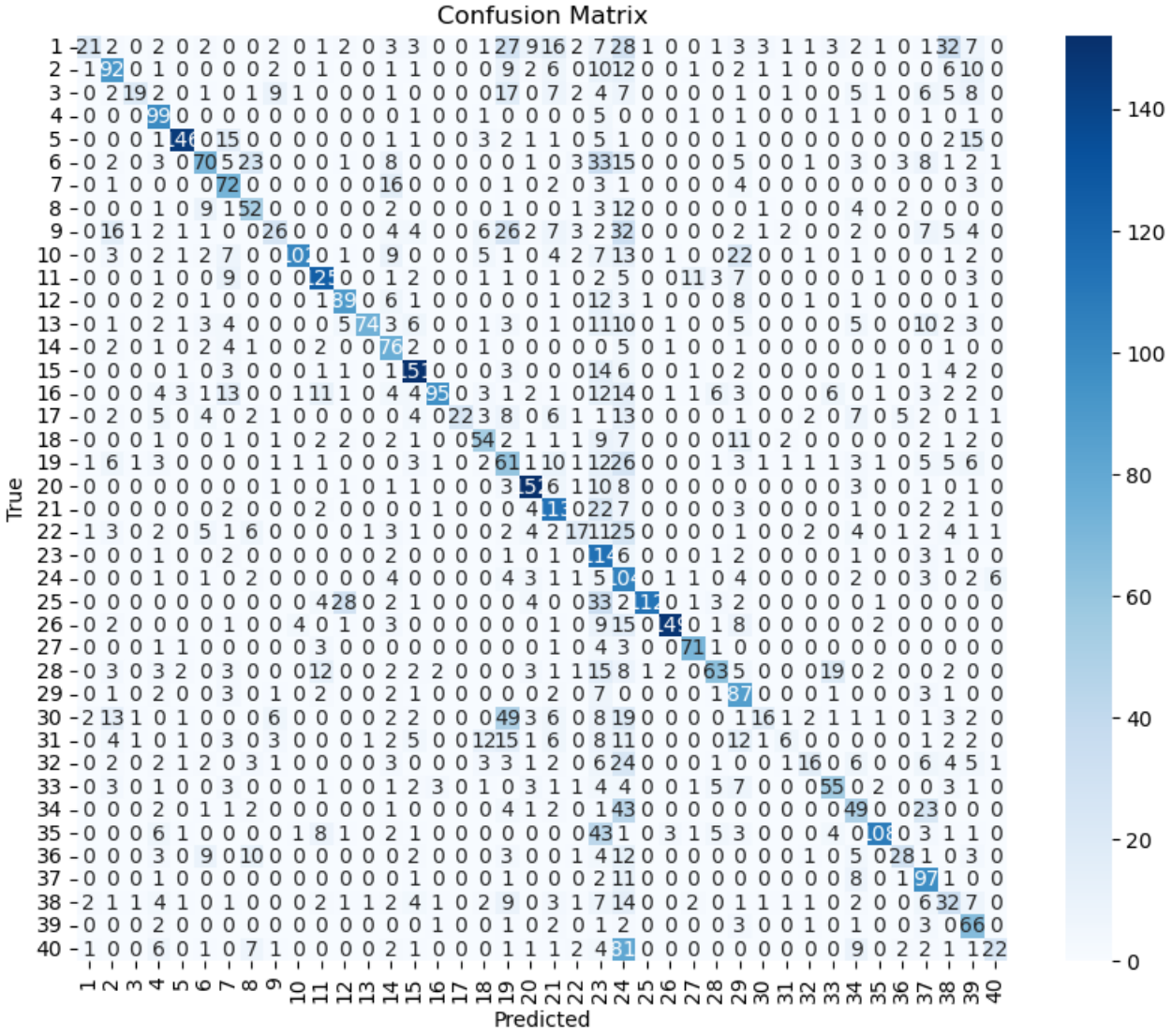
Accuracy of the model on the test images: 14.16%

Resnet18 raw augmented images given split

In this and in the following sections, we'll use transfer learning on ResNet-18.

The confusion matrix below is for the following configuration:

1. Train-test split suggested by [Stanford40]
2. Data augmentation enabled



Accuracy of the model on the test images: 52.84%

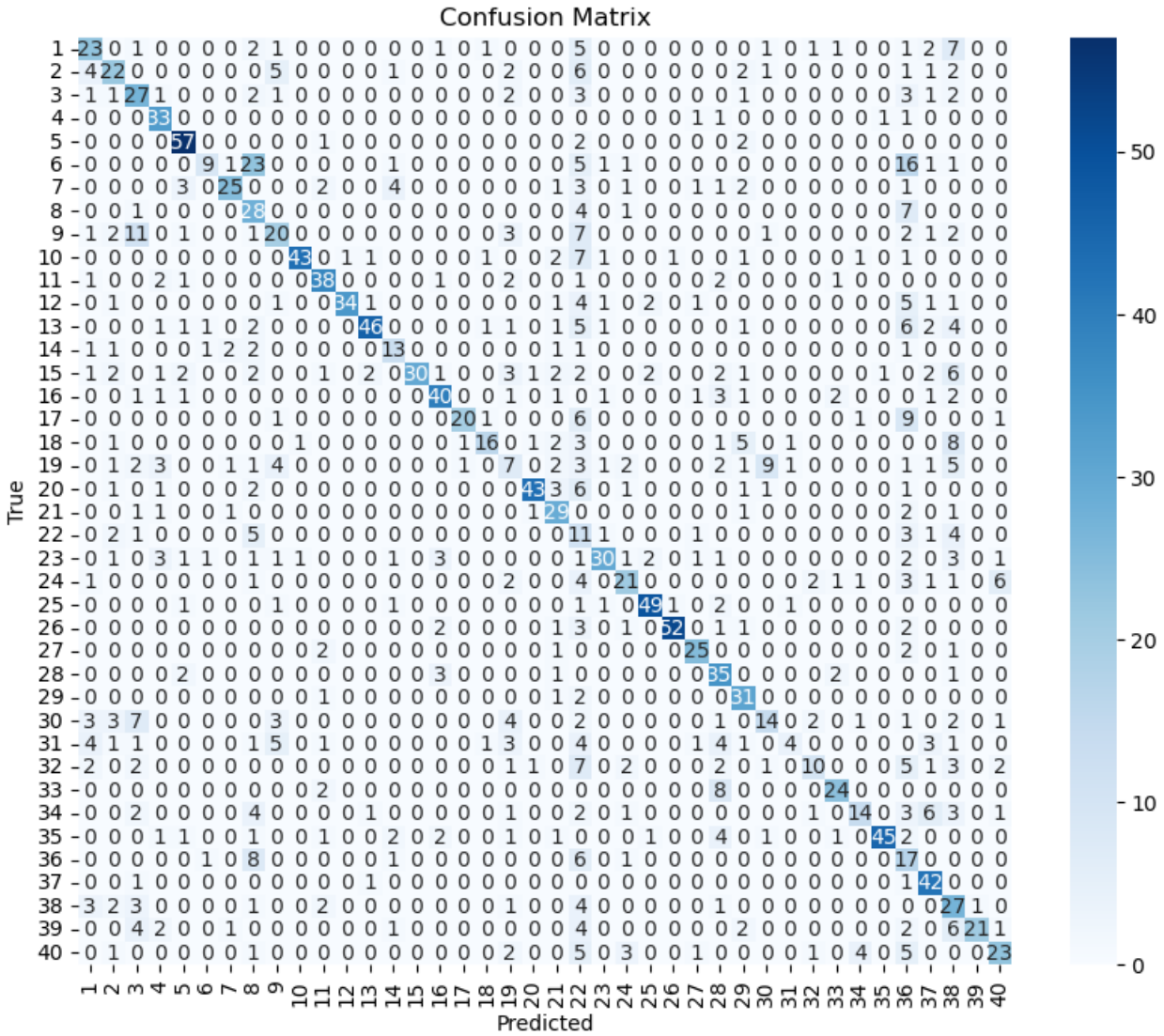
Although we see a minor bias for classes 23-24, this is significantly better accuracy compared to our custom CNN architecture.

Resnet18 raw augmented images random split

This is the direct comparison with the “best case” for our custom CNN architecture:

1. Random 80:20 train-test split
2. Data augmentation enabled

We got the following confusion matrix:

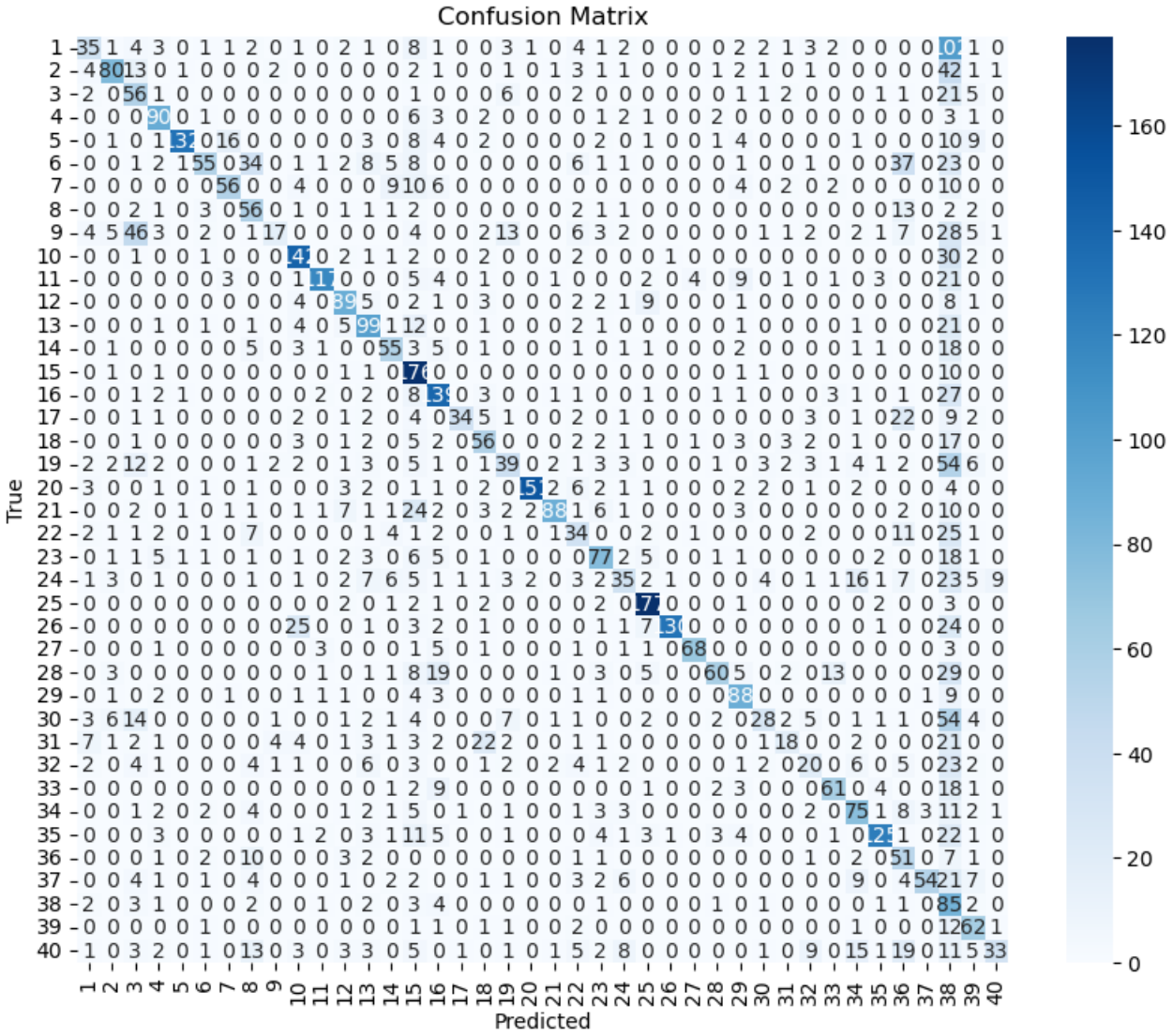


Accuracy of the model on the test images: 58.91%

This is the best result overall achieved in this exercise. It can be seen that the bias on classes 23-24 from the previous case was eliminated. It must be noted that the training time was overwhelmingly smaller compared to our custom CNN architecture: just 7 minutes 32 seconds versus more than 140 minutes for the custom CNN.

Resnet18 raw images given split

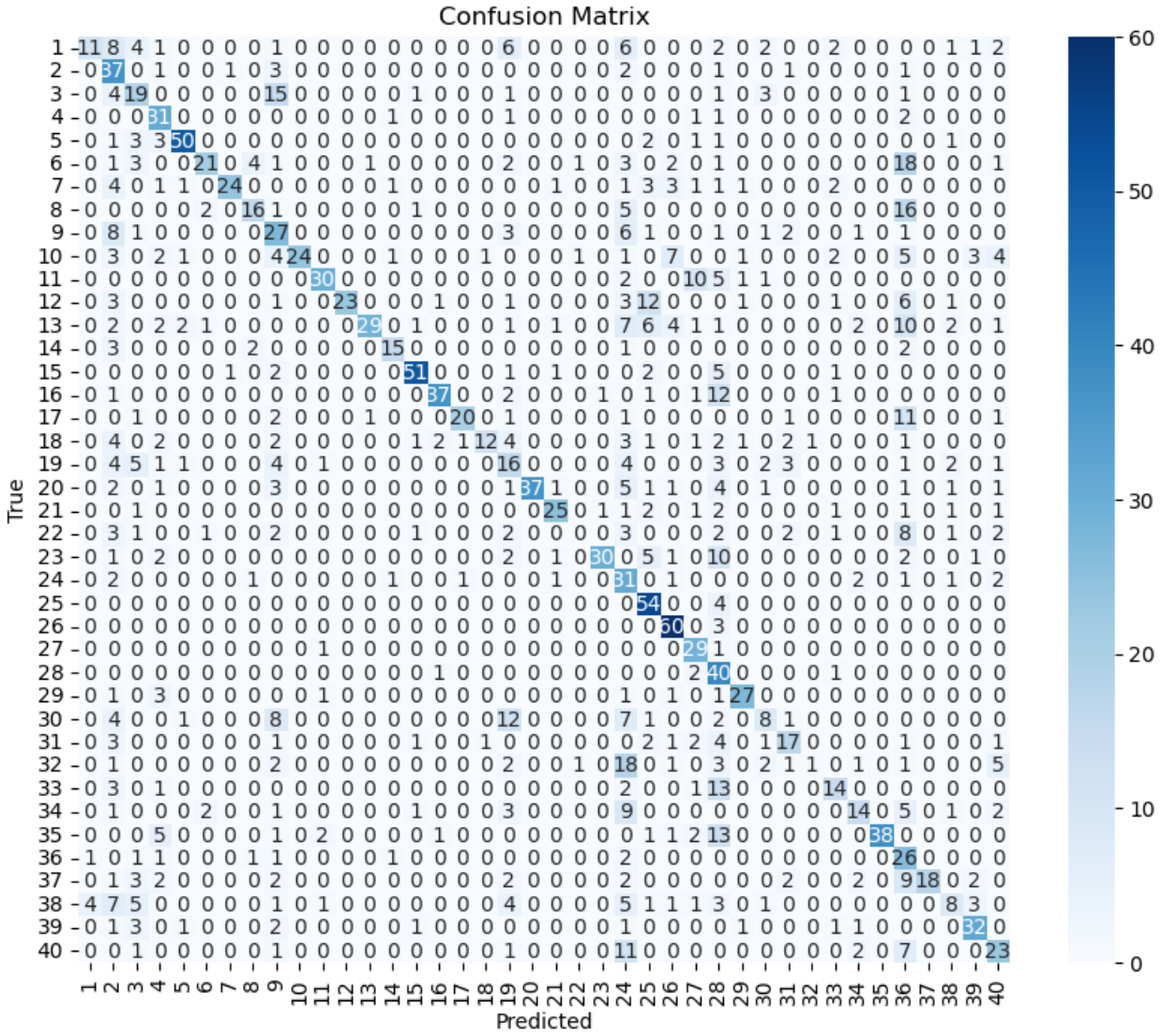
If we remove the data augmentation, we see the bias appearing again:



Accuracy of the model on the test images: 55.01%

Resnet18 raw images random split

And if we replace the given split with the random split we see that the bias mostly disappears:



Accuracy of the model on the test images: 54.99%

Despite the marginally lower accuracy compared to the results with the given splits, it can be argued that eliminating bias is more important than getting better average accuracy.

Tests using the bounding boxes

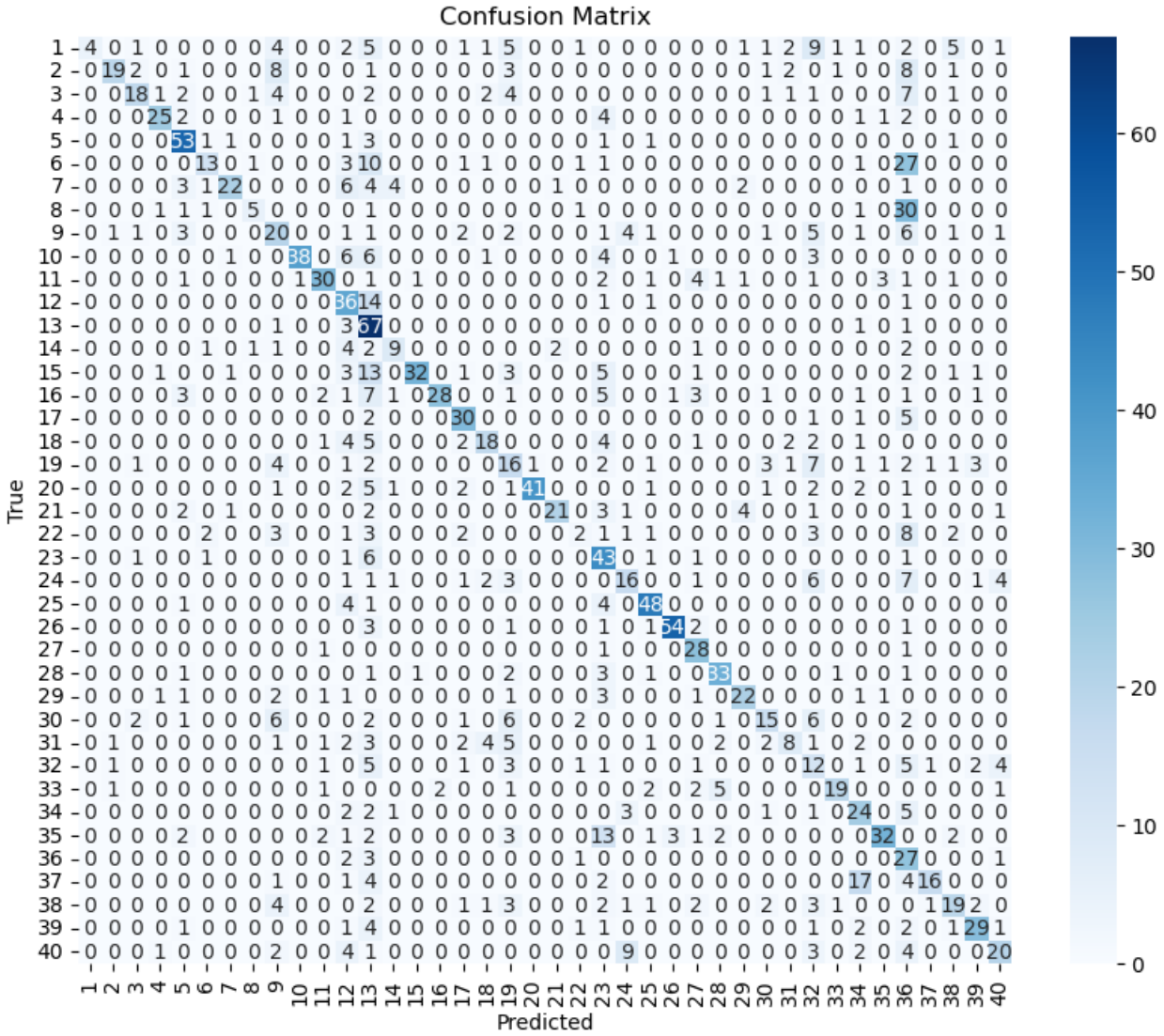
Provided dataset [Stanford40] includes not only the images, but also the metadata about the bounding boxes for the locations of people performing the actions on the images. It is assumed that using these bounding boxes will help increase accuracy of classification.

As an additional test, the provided bounding boxes were used as bitmap masks built-in to the images themselves. To implement this, a custom data source class has been implemented, presented in Appendix B. ImageFolderWithBBox.

We will use the best performing setup we have: resnet18 with data augmentation on random data split, and enhance every image with the bounding box. The bounding box will be included in the

image data itself as a fourth “color” channel. This will require us to modify the resnet18 a bit further, changing its first layer from 3-input to 4-input one. The code for that is presented in the Appendix C. 4-channel ResNet-18 with 40 outputs.

The confusion matrix at the end of 20 training epochs is as follows:



Accuracy of the model on the test images: 54.29%

Curiously enough, the accuracy dropped by 4% when using the bounding boxes. It’s possible that the strategy of using the bounding boxes directly as an additional input is not helpful.

The other possible option to explore in the future work is to use the provided bounding boxes to crop the training images, effectively leaving only the images of people performing the actions. This potentially has a downside of losing the context, as we remove the majority of the image by doing such a cropping operation.

Conclusions

In general, we see that usage of the random split eliminates the bias visible on the given split. In addition to that, adding data augmentation further improves accuracy. Both of this tricks increase the amount of training examples, effectively prolonging the training of the model.

This, however, can be interpreted in the other way. As in the given split the amount of test images is larger, it's possible that testing on that dataset actually *uncovers* the biases, which are not visible when we testing with the random splits, as the amount of testing images is smaller there. However, it's not possible to confirm this assumption as I don't have enough spare images to meaningfully setup a proper comparison.

Transfer training was overwhelmingly successful, as we were able to significantly reduce the training time while increasing the accuracy from the get-go. It is possible that our custom CNN would be able to achieve the comparable results given enough resources. But [ResNet] paper mentions using more than a million of images as a training dataset compared to our 9500 and [AlexNet] mentions using double GPU setup running for 6 days. All of this is unrealistic for me.

The code of the implementation is published in GitHub at https://github.com/hijarian/avpr-assignment-2. See the README.md there for organization details.

Bibliography

[Stanford40] **B. Yao, X. Jiang, A. Khosla, A.L. Lin, L.J. Guibas, and L. Fei-Fei**. (2011). *Human Action Recognition by Learning Bases of Action Attributes and Parts.* Internation Conference on Computer Vision (ICCV), Barcelona, Spain. November 6-13, 2011. http://vision.stanford.edu/Datasets/40actions.html

[ImageNet] **Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L.** (2015). *ImageNet Large Scale Visual Recognition Challenge*. International Journal of Computer Vision, 115(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[AlexNet] **Krizhevsky, A., Sutskever, I., & Hinton, G. E.** (2012). *ImageNet Classification with Deep Convolutional Neural Networks*. Advances in Neural Information Processing Systems, 25, 1097–1105.

[VGGNet] **Simonyan, K., & Zisserman, A.** (2014). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv preprint arXiv:1409.1556.

[ResNet] **He, K., Zhang, X., Ren, S., & Sun, J.** (2016). *Deep Residual Learning for Image Recognition*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–778.

[ChatGPT] **OpenAI.** (2025). *ChatGPT (January 2, 2025 version)*. AI assistant. https://openai.com/chatgpt.

[PyTorch] **Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al.** (2019): *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* Advances in Neural Information Processing Systems 32: 8024–8035. https://pytorch.org.

[TransL] **Pan, Sinno Jialin, and Qiang Yang**. (2010). *A Survey on Transfer Learning.* IEEE Transactions on Knowledge and Data Engineering 22, no. 10 (2010): 1345–1359. https://doi.org/10.1109/TKDE.2009.191

[Adam] **Kingma, Diederik P., and Jimmy Ba**. (2017). *Adam: A Method for Stochastic Optimization.* arXiv preprint arXiv:1412.6980. https://arxiv.org/abs/1412.6980.

Appendix A. Custom CNN

```
import torch.nn as nn

CHANNELS_COUNT=3

class CustomCNN(nn.Module):
    def __init__(self, num_classes=40):
        super(CustomCNN, self).__init__()
        self.features = nn.Sequential(
              # Convolutional layers

            nn.Conv2d(CHANNELS_COUNT, 32, kernel_size=3, stride=1, padding=1),
              # 3x400x400 -> 32x400x400
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
              # 32x400x400 -> 32x200x200

            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
              # 32x200x200 -> 64x200x200
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
              # 64x200x200 -> 64x100x100

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
              # 64x100x100 -> 128x100x100
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
              # 128x100x100 -> 128x50x50

            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
              # 128x50x50 -> 256x50x50
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
              # 256x50x50 -> 256x25x25
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
              # 256x25x25 -> 256*25*25
            nn.Linear(256 * 25 * 25, 1024),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, num_classes)
              # 1024 -> 40
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

def make_model(device):
    """
    Make the custom CNN model compatible with resnet or other pretrained models.
    Configured for images with 3 input channels and 40 output channels.
    """
    return CustomCNN(num_classes=40).to(device)
```

Appendix B. ImageFolderWithBBox

```
import os
import torch
from torchvision.datasets import ImageFolder
from torchvision import transforms
from PIL import Image, ImageDraw
import numpy as np
from my_bbox import get_bounding_box
from my_bitmask import generate_bitmask

class ImageFolderWithBBox(ImageFolder):
    def __init__(self, root, augment=False):
        super(ImageFolderWithBBox, self).__init__(root)
        self.augment = augment

    def __getitem__(self, index):
        # Get the image and label from the parent class
        path, label = self.samples[index]
        image = self.loader(path)  # Load image using default loader

        image_id = os.path.basename(path).split('.')[0]
        # Get the corresponding bounding box annotation
        _, bbox = get_bounding_box(image_id)

        original_width, original_height = image.size

        resize = transforms.Resize((224, 224))

        # apply resize to the image
        image = resize(image)

        # Resize the bounding box to the same proportions
        resized_width, resized_height = image.size

        width_scale = resized_width / original_width
        height_scale = resized_height / original_height

        x_min, y_min, x_max, y_max = bbox

        x_min = int(x_min * width_scale)
        y_min = int(y_min * height_scale)
        x_max = int(x_max * width_scale)
        y_max = int(y_max * height_scale)

        bbox = (x_min, y_min, x_max, y_max)

        # must apply color jitter before converting to tensor and adding 4th channel
        if self.augment:
            jitter = transforms.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.2)
            image = jitter(image)

        bitmask = generate_bitmask(image.size, bbox)

        # Convert bitmask to a tensor and add it as the 4th channel
        bitmask_tensor = torch.tensor(bitmask, dtype=torch.float32).unsqueeze(0)
        if not isinstance(image, torch.Tensor):
            image = transforms.ToTensor()(image)
        image = torch.cat((image, bitmask_tensor), dim=0)

        # ToTensor normalizes pixels to [0, 1],
        # but further augments don't care about that so it's safe to apply it here

        # coefficients tailored to the dataset
        # mean 0 and std 1 for the bitmask - don't change it
        mean = np.array([0.46824582, 0.44051382, 0.40057998, 0])
        std = np.array([0.24759944, 0.23944832, 0.24277655, 1])
        normalize = transforms.Normalize(mean=mean, std=std)
```

```
        if self.augment:
            transform = transforms.Compose([
                transforms.RandomHorizontalFlip(),
                transforms.RandomVerticalFlip(),
                transforms.RandomRotation(30),
                normalize
            ])
        else:
            transform = normalize

        image = transform(image)

        # Apply transformations to the label (if any)
        if self.target_transform is not None:
            label = self.target_transform(label)

        return image, label
```

Appendix C. 4-channel ResNet-18 with 40 outputs

```
import torch
import torch.nn as nn

import torchvision.models as models

def make_model(device):
    resnet = models.resnet18(pretrained=True)  # Example with ResNet-18

    # Get the original weights of the first layer
    original_conv1 = resnet.conv1
    new_conv1 = nn.Conv2d(
        in_channels=4,  # Change to 4 channels
        out_channels=original_conv1.out_channels,
        kernel_size=original_conv1.kernel_size,
        stride=original_conv1.stride,
        padding=original_conv1.padding,
        bias=original_conv1.bias
    )

    # Copy pretrained weights for the first 3 channels (RGB)
    with torch.no_grad():
        new_conv1.weight[:, :3, :, :] = original_conv1.weight
        new_conv1.weight[:, 3:, :, :] = 0  # Initialize weights for the new channel to 0

    # Replace the first layer in ResNet
    resnet.conv1 = new_conv1

    # Modify the final layer for our task (40 classes)
    resnet.fc = nn.Linear(resnet.fc.in_features, 40)

    return resnet.to(device)
```