IMATERIALIST IMAGE RETRIEVAL SYSTEM TECH REPORT

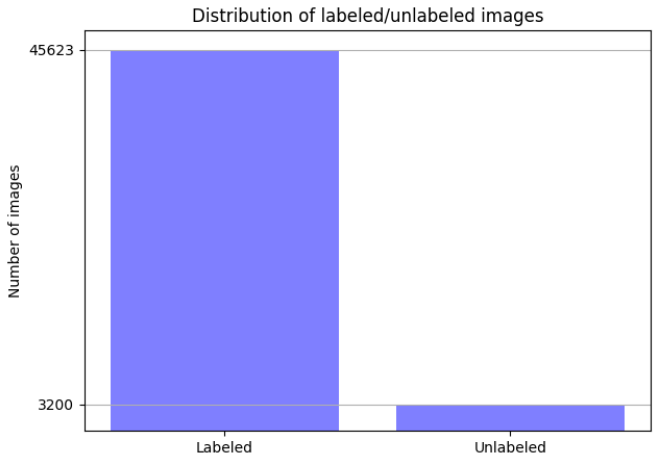
Anastasiia Havryliv

EDA

To start with, let's have a look at the data we are working with. We have the data divided into two folders: train and test, where train contains labeled images, and test - unlabeled ones.

We can see the distribution of labeled/unlabeled data on the plot below: (I want to clarify that mainly I will be using train set (labeled one) to train models, and test set of the competition will not be directly used for the results, because we don't have labels for these images and I won't be able to check all of them manually)

And from the following plot we can see that we will have 45623 labeled images for training and validation of the models:



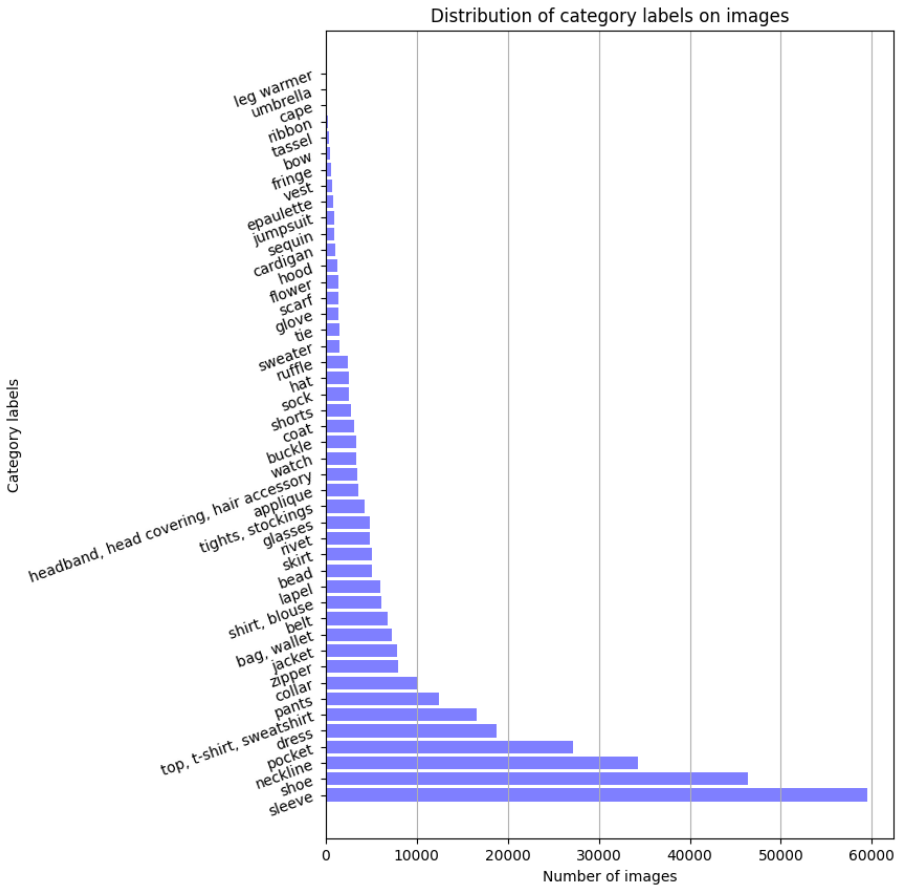
I extracted the data about labeled images (train.csv) into a dataframe to provide descriptive analysis:

	ImageId	EncodedPixels	Height	Width	ClassId	AttributesIds
0	00000663ed1ff0c4e0132b9b9ac53f6e	6068157 7 6073371 20 6078584 34 6083797 48 608...	5214	3676	6	115,136,143,154,230,295,316,317
1	00000663ed1ff0c4e0132b9b9ac53f6e	6323163 11 6328356 32 6333549 53 6338742 75 63...	5214	3676	0	115,136,142,146,225,295,316,317
2	00000663ed1ff0c4e0132b9b9ac53f6e	8521389 10 8526585 30 8531789 42 8537002 46 85...	5214	3676	28	163
3	00000663ed1ff0c4e0132b9b9ac53f6e	12903854 2 12909064 7 12914275 10 12919485 15 ...	5214	3676	31	160,204
4	00000663ed1ff0c4e0132b9b9ac53f6e	10837337 5 10842542 14 10847746 24 10852951 33...	5214	3676	32	219

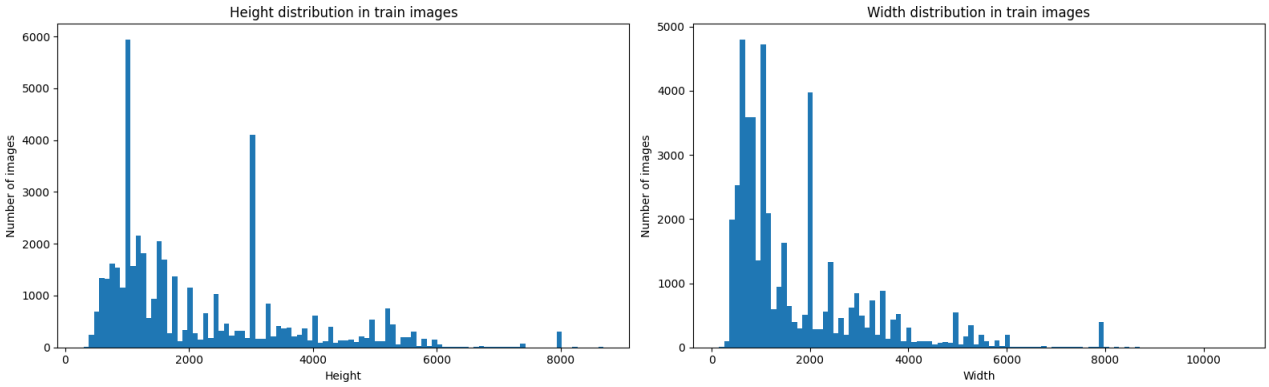
One row of the csv file provides information about the class (ClassId) represented on the corresponding image (ImageId) and pixels, on which this class is visible (EncodedPixels).

Categories and attributes are stored in the general csv file (every category (ClassId) is stored in a separate row for each picture and with attribute ids for each picture). Description of categories and attributes is stored in a separate json file ‘label_descriptions.json’.

On the bar plot below we can see the distribution of class labels on the images of the dataset. From this plot we can conclude that distribution of the labels is uneven, because for example most of the images of clothing have sleeves or shoes in them, but leg warmers or umbrellas are not so popular across the photos.



Also, there is a very uneven distribution of shape (width and height) of the images in tha dataset. We see that there are also a few outliers in the data - pictures with very large width and height.



To see the difference between images with min and max height and width better, I will show a plot of them below:

MODEL TRAINING

DATASET CREATION

Dataset (from scratch and pretrained models)

```python
class FashionDataset(Dataset):
  def __init__(self, image_dir, transform=None):
    self.image_dir = image_dir
    self.image_fns = os.listdir(image_dir)
    self.transform = transform

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

  def __getitem__(self, idx):
    image = Image.open(f'/kaggle/working/{self.image_dir.split("/")[-1]}/{self.image_fns[idx]}')

    if self.transform is not None:
      image = self.transform(image)

    label = np.array(encode_label(self.image_fns[idx], self.image_dir.split('/')[-1]))
    label = torch.Tensor(label)

    return image, label
```

On the image (left) we can see one of the variations of the dataset. I used two variations because I tried both transforms from albumentations library (where I needed to pass np.arrays to them) and transforms from torchvision, where I needed to pass PIL Image. Initially I switched from albumentations library to torchvision transforms because torchvision transforms are faster and I needed to speed up the process.

One hot encoding function (encode_label()) was customly written to encode the classes into 0 and 1 inside a tensor of the length of all classes, where 1 corresponds to a class that is represented in the image.

Dataset for metric learning

Overall the logic is simple, in the __init__ of the dataset I additionally create a list of tuples, where each tuple contains indices of anchor, positive and negative images - so called triplets. And later, in the __getitem__ function i retrieve these indices by item index from the general array, and get images by these indices from general array of images (which are also loaded to the RAM during the initialization on dataset to speed up the training process)

The algorithm of choosing triplets: First, I iteratively take one index, which will be the anchor index. Then I randomly select some other index in the range of length of the data, and check whether this image intersects with the anchor image by more than 3 classes, if yes - I append it to the positives array. If it has zero intersection with anchor image by classes - it is appended to the negatives list.

(I also added some checks to prevent very long work of the function and cases where there are no intersections between the pictures. I decided to check whether number of iterations is bigger than some number, and if so - reduce the number of matched classes for the image to be considered positive).

In the end I choose a random positive index from the positives array and random negative index from the negatives array and append tuples (anchor_index, positive_index, negative_index) to the general triplets array.

This is the way that I have chosen to select positive and negative samples to the anchor to implement Triple Loss Metric Learning.

AUGMENTATION

I didn't add a lot of augmentation to the images, the only three transforms that I did are resizing all (train, validation and test) images to 128*128, because of the diverse resolution of images in dataset, and 128*128 was the most reasonable size that kept training time relatively low and didn't use up all of the RAM.

The next transform that did was Normalizing the images, because this should help with generalization during the training process and also reduce training time.

And the last transformation is pretty standard - ToTensorV2(), as we need tensors to pass them into our model.

TRAIN/TEST/VALIDATION

The split into train and validation was pretty standard: 80% of the labeled images went to the train set and 20% randomly sampled labeled images - to the validation subset. Test images obviously weren't used for the training process, because they are unlabeled and can be only used for manual check of the predicted labels.

ResNet50

At the very beginning I decided to choose ResNet50 as a model for training. (Before ResNet50 I have also tried to train ResNet18, but as it is smaller (has less parameters) and had given bad results, I decided to switch to heavier model)

ResNet50 is considered one of the best models for image classification, so I have decided to try it for this task of multi-label classification too.

From scratch (randomly initialized weights)

```
params_resnet50 = {
    "device": device,
    "lr": 0.0005,
    "batch_size": 32,
    "num_workers": 4,
    "epochs": 9,
}
```

For the first iteration I have chosen standard parameters, where the device is cuda. Learning rate is slightly lower than the common choice (0.001), because I have successfully trained this model with learning rate 0.0005 for image classification tasks, so I decided to use the same learning rate here.

Num workers parameter was initially added to speed up the process of moving the images inside the data loader. Number of epochs was chosen as 9, because initially I had some slow processes in the creation of the dataset and training loop, so it was the most epochs I could train in more or less reasonable time.

For the last (fully connected) layer, I changed it to Linear(2048, len(categories_ids)) to transform output into the number of classes that I need, and then applied Sigmoid, as we need probabilities for the one-hot encoding and overall deal with binary classification (1 - label is on the picture, 0 - label is not on the picture)

```
model_resnet50.fc = nn.Sequential(
    nn.Linear(2048, len(categories_ids)),
    nn.Sigmoid()
)
```

For the optimizer I have chosen the safe option - Adam optimizer which is maybe most commonly used now and as a criterion I have taken BCELoss, which is a great fit for the multilabel classification.

Also as a second option we can choose BCEWithLogitsLoss, which already contains Sigmoid so if we use this loss we don't need to add Sigmoid to the fully connected layer of the model.

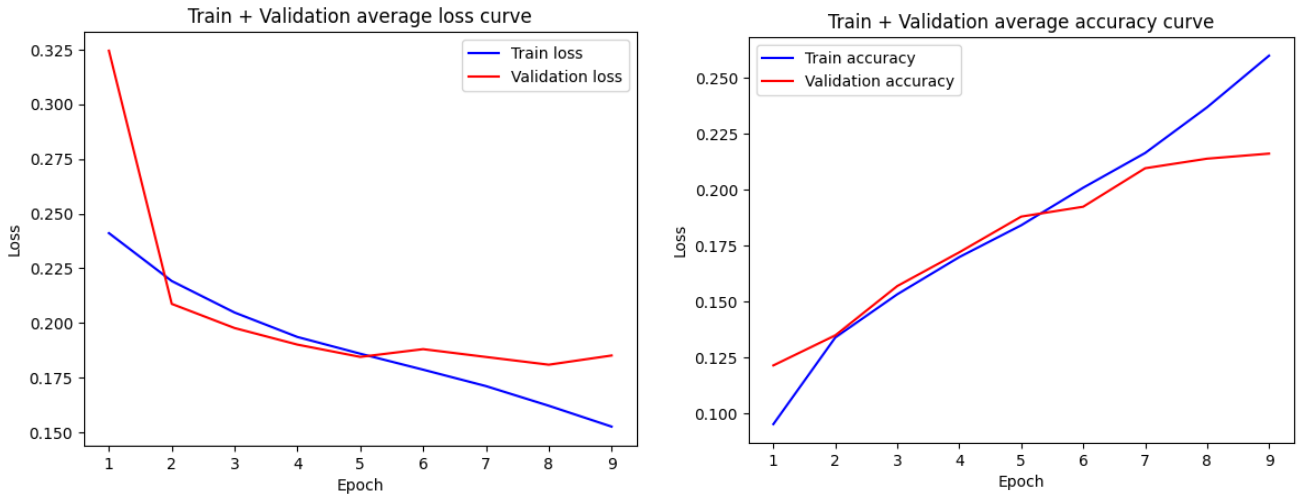
```
criterion_resnet50 = nn.BCELoss().to(device)
optimizer_resnet50 = torch.optim.Adam(model_resnet50.parameters(), lr = params_resnet50['lr'])
```

As an accuracy function I have chosen F1 Score (from torch metrics - classification.MultiLabelF1Score). It is a good metric when we have class imbalance and it also integrates precision and recall into one metric.

```
model_resnet50 = models.resnet50(weights=None)
```

(I don't pass the weights here because the model must be randomly initialized)

Results:



From the accuracy and loss plots we can not say that this is the best model we can get, but overall, losses tend to drop and accuracies tend to rise both on the train and validation subsets. (However, we can see slight overfit - plateau on the validation plot of loss/accuracy)

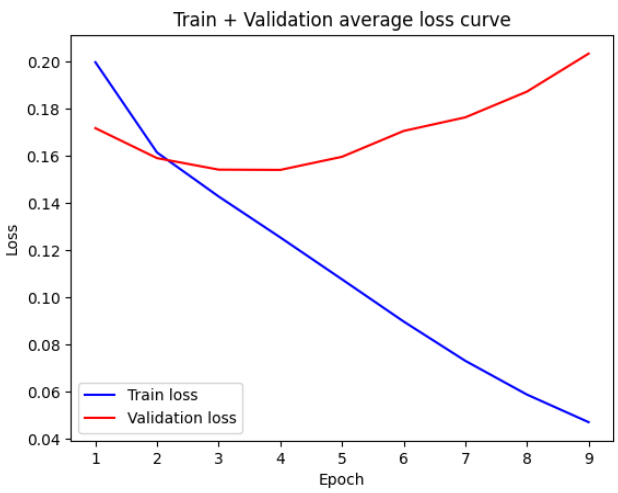
Pretrained on ImageNet1K

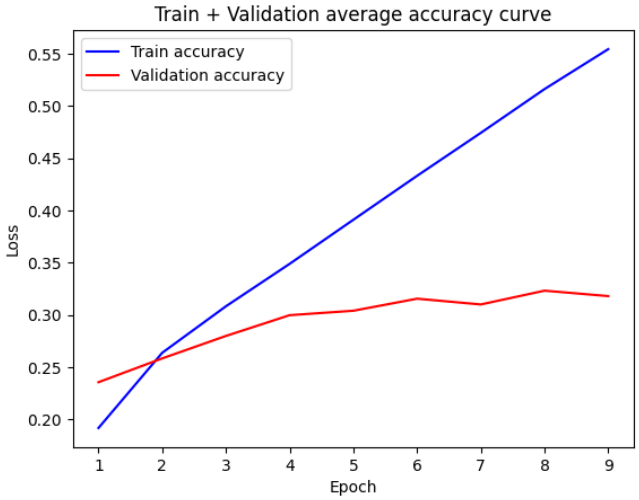
All of the parameters, optimizer and criterion used here are the same as with the randomly initialized version.

However, at the beginning I was training with all of the layers freezed, but after that, I decided to unfreeze all of the layers to let the model learn more than one fully connected layer.

(Model with all frozen layers doesn't give good results - loss doesn't drop for both train and validation subsets, and accuracy doesn't rise at all, even on the train set)

Result in described setup:





From these average loss and accuracy plots we can see that this setup with that certain model didn't work well, because we got huge overfit (judging by the loss plot), which started already from the third epoch.

MobileNetV2

This was my second choice of the model, and because of the fact that at that time I still had problems with training time, I decided to try some smaller model, with less number of parameters. So I was choosing from pytorch pretrained models and decided to tae this one because apparently it can also be used for image classification, and this model is considered to be not the worst one.

From scratch (randomly initialized weights)

```
params_mn2 = {
    "device": device,
    "lr": 0.0005,
    "batch_size": 32,
    "num_workers": 4,
    "epochs": 11,
}
```

Parameters for MobileNetV2 network are the same as for the ResNet, just the standard ones.

However, for this model I decided not to completely override the fully connected layer, but to leave the fully connected layer that was in the model before, but stack Linear (that has number of our classes as output channels) and Sigmoid() because we deal with multi-label classification and one-hot encodings here.

```
model_mn2.classifier = nn.Sequential(
    model_mn2.classifier,
    nn.Linear(1000, len(categories_ids)),
    nn.Sigmoid()
)
```

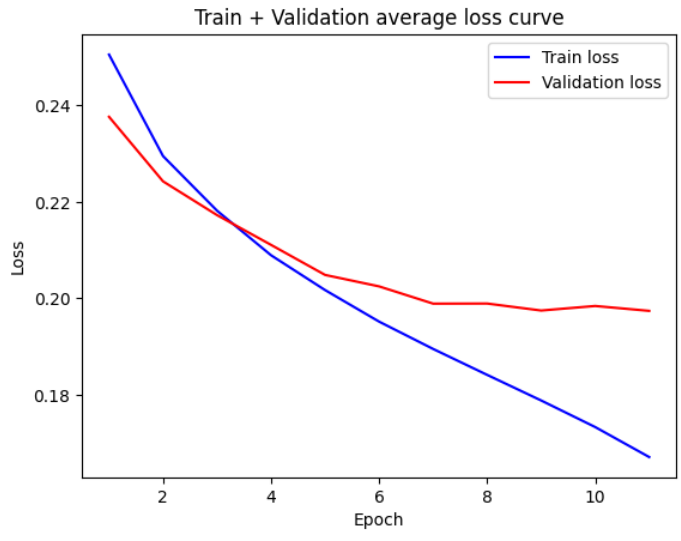
As for the optimizer and criterion, justification of this choice is the same as for the ResNet50 model.

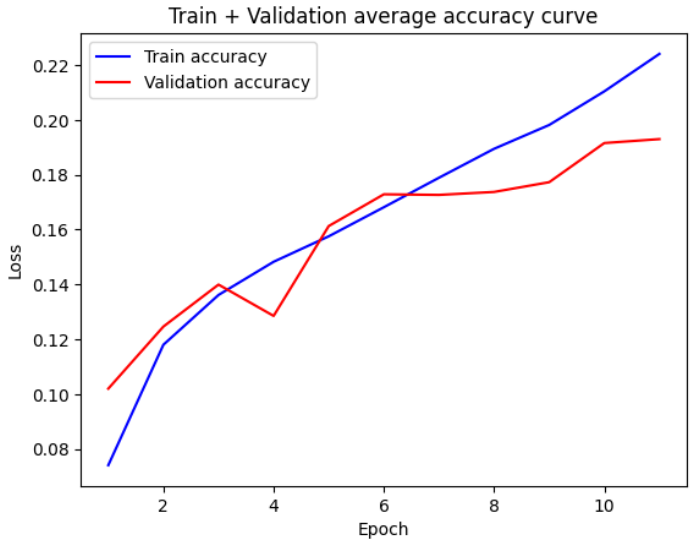
```
criterion_mn2 = nn.BCELoss().to(device)
optimizer_mn2 = torch.optim.Adam(model_mn2.parameters(), lr = params_mn2['lr'])
```

And also as it is a randomly initialized model - the weights passes to the model are None.

```
model_mn2 = models.mobilenet_v2(weights=None)
```

Results:





Here we can see something similar to the ResNet50 pretrained situation, but here we can see that validation loss doesn't move up or down after the 8-th epoch. Average accuracy plot is overall also the same, however we got some downfall of the validation accuracy on the 4-th epoch, but the situation started to look slightly better after the 5-th epoch.

Pretrained on ImageNet1K_V2

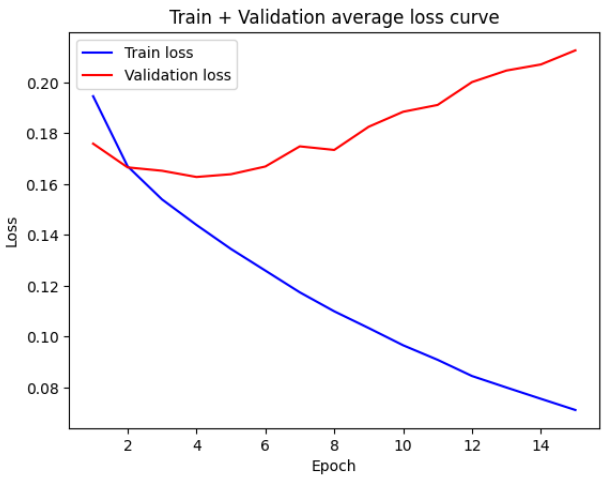
```
params_mn2 = {
    "device": device,
    "lr": 0.0005,
    "batch_size": 32,
    "num_workers": 4,
    "epochs": 15,
}
```

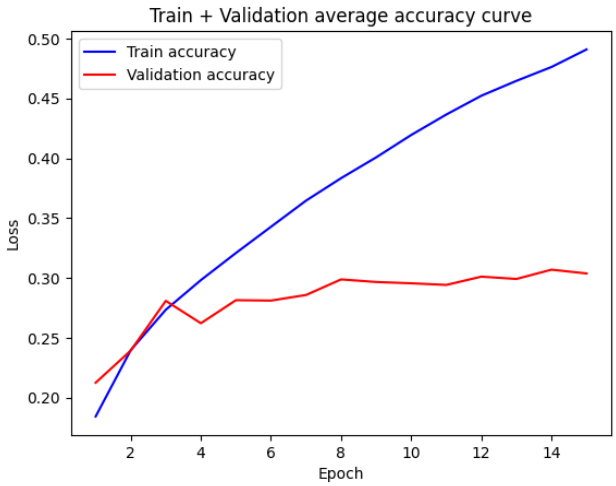
Learning rate and batch size stayed the same as in the previous iterations of this and other models, but I decided to increase the number of epochs so see if the pretrained version of MobileNetV2 will perform better on our data.

Optimizer and criterion were also left the same, along with the modifications of the fully connected layer of the model

```
criterion_mn2 = nn.BCELoss().to(device)
optimizer_mn2 = torch.optim.Adam(model_mn2.parameters(), lr = params_mn2['lr'])
```

Results:





MobileNetV2, pretrained on ImageNet1K performs maybe the worst among all these models. We can see that already after second epoch validation loss starts to rise (and validation accuracy stops increasing after the second epoch)

Metric learning

(The data preparation was described in one of the previous sections)

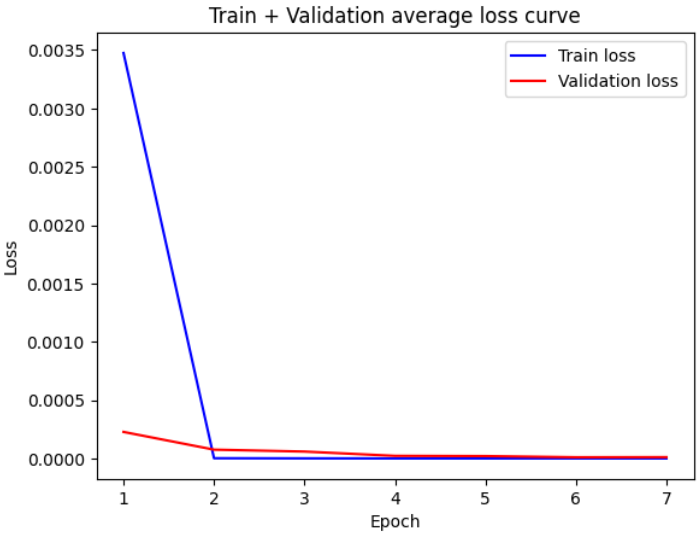
For accuracy score, I calculated the distance between output of the model given the anchor image and output of the model given positive image, the same was done with outputs of the model of the positive image. Then from these two distances i calculated with triplets were labeled correctly and divided thi number by the total number of triplets in the batch to get the accuracy score.

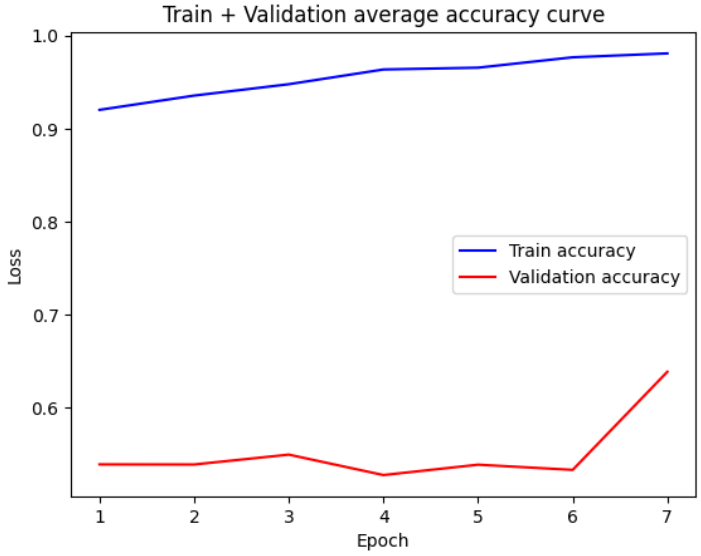
```python
def calculate_accuracy(output_anchor, output_positive, output_negative, margin=0.0):

    positive_distance = torch.nn.functional.pairwise_distance(output_anchor, output_positive)
    negative_distance = torch.nn.functional.pairwise_distance(output_anchor, output_negative)
    correct_triplets = (positive_distance < negative_distance + margin).sum().item()

    total_triplets = output_anchor.size(0)
    accuracy = correct_triplets / total_triplets

    return accuracy
```

For calculation of the loss I used nn.TripletMarginWithDistanceLoss, which allows us to pass the function for distance calculation as a parameter. So I passed torch.nn.fucntional.pairwise_distance to the loss calculator to ensure that both of the metrics are calculated correctly and using the same losses/accuracies functions.

As a model

Results:





From plots we can see that loss drops almost to zero already during the second epoch. I think i can be possible, because in every epoch we try to minimize the distance between the anchor and positive, but at the same time try to minimize the distance between the anchor and the negative. So maybe they compensate each other. And because we use margin=0.0 in loss and accuracy calculation functions, it possibly can reach very small numbers, close to zero.

['shirt, blouse', 'cardigan', 'pants', 'shorts', 'jumpsuit', 'cape', 'glasses', 'hat', 'tie', 'watch', 'tights, stockings', 'scarf', 'lapel', 'epaulette', 'sleeve', 'neckline', 'zipper', 'applique', 'bow', 'flower', 'fringe', 'rivet', 'ruffle', 'sequin', 'tassel']

This is an example of prediction that I get on the test set. And somehow, this model predicts me mostly the same items for every image that i try. I thought that it was because I was forming triplets, where positive and negative images were the same for all of the image pairs, so I corrected the strategy, but nothing changed. Also I tried to tune margin in the loss and accuracy functions, but unfortunately that didn't work too.

I have read about AngularLoss and that it should work better in handling such cases, but unfortunately I didn't find implementation of this loss in pytorch (it is available in the library pytorch-metric-learning, but this library is probably can not be used for multilabel classification, at least I made that conclusion from their docs).

Self-supervised learning

For self-supervised learning I also used MobileNetV2. To make the dataset bigger, and help the model adapt to different conditions I introduced Rotation transform along with the others. As for the loss, we need some specific one for self-supervised learning because we have to avoid using labels from the labeled set of images for the model to learn patterns by itself.

I didn't find the implementation of this loss in one of the libraries (except from pytorch-metric-learning, but most probably this library doesn't work with multi label classification), so I won't be able to show plots for this task.

FAISS

I also tried implementing an image retrieval system with my trained models (the weights of the models I saved after each iteration). It uses the same dataset that has been used for from-scratch and pretrained models.