

Image Classification

Assignment for the position of Data scientist at TheraPanacea

Fahad Khalid, PhD

Date: 26-02-2023

Submitted to:

Sami Romdhani

Task:

The task requires developing an image classification model with the aim to minimize the Half-Total Error Rate (HTER). A dataset consisting of 100k images along with a corresponding label file, both housed within a zip file, will serve as the basis for the model's training and evaluation. The images, named numerically, align with their binary labels presented line by line in the label file. The task requires preprocessing the images, splitting the dataset, designing, and training an appropriate image classification model such as a Convolutional Neural Network (CNN). The model performance will be validated and iteratively improved until satisfactory HTER results are achieved. The final product will be a well-documented and robustly performing image classifier.

https://github.com/FahadKhalid1/Image_classification_exercise.git

Introduction:

Advancements in binary image classification have led to the development of highly accurate models for numerous tasks, including face recognition, object detection, and medical image analysis. Binary image classification involves distinguishing between two classes, which poses unique challenges and opportunities for model innovation. Deep Learning, with the advent of convolutional neural networks (CNNs), revolutionized binary image classification tasks. CNNs have multiple layers of neurons that can learn high-level features from raw pixel data, making them extremely efficient for image analysis. Architectures such as LeNet, AlexNet, VGG, and ResNet brought significant improvements in binary classification tasks. However, more recent advancements have introduced new strategies and architectures. Capsule Networks, for instance, offer an alternative to CNNs by encoding spatial relationships between high-level features in an image, which has shown promise for binary classification tasks (Sabour, S., Frosst, N., & Hinton, G. E., 2017, "Dynamic Routing Between Capsules", <https://arxiv.org/abs/1710.09829>). Capsule Networks offer an exciting alternative to CNNs. They have shown potential in tasks requiring the understanding of spatial hierarchies and relationships, like recognizing faces with varying orientations or expressions. These networks identify and group related features into "capsules," providing richer representation and improved discrimination between classes. More recently, transformer models originally designed for text, like BERT, have been adapted to image classification tasks with models like Vision Transformer (ViT). These models divide an image into a sequence of patches, treating each as an input token like words in a sentence. This approach allows for a global understanding of the image, beneficial for tasks like binary face classification where context and relational information are crucial. Finally, Generative Adversarial Networks (GANs) have provided a novel approach to handle class imbalance, a common issue in binary classification tasks. GANs can generate synthetic examples of the minority class, thereby enhancing the model's ability to accurately classify less represented classes. Image classification models, particularly for facial recognition tasks, have seen significant advancements in recent years. These advancements have revolutionized the field of computer vision, leading to improved accuracy and efficiency in image classification tasks. In summary, advancements in image classification methods have continuously evolved to improve the accuracy, robustness, and efficiency of binary face classification tasks. It's an

exciting area of research, and further advancements are likely to continue pushing the boundaries of what's possible.

The Dataset:

After a meticulous visual examination, it is evident that the dataset is primarily composed of facial images set against diverse backgrounds, each measuring 64x64 pixels. Class 1 predominantly includes images featuring unadorned faces, free from wearables such as glasses, caps, hats, or bandanas, and devoid of any facial hair. In contrast, Class 0 is distinguished by faces that are partially obscured either by various wearables or the presence of facial hair. The labeling of the dataset appears to be generally reliable. However, with a substantial sample size like 100K images, it is inevitable that a few discrepancies or mismatches are present within the dataset. The Figure below displays few examples from each class:



The training data has [87,901](#) images labeled as '1' and [12099](#) images labeled as '0'. Hence the classes are highly imbalanced.

Half total error loss:

The Half Total Error Rate (HTER) loss function is a specific type of loss function used in binary classification problems. It measures the average of two types of errors: the False Acceptance Rate (FAR) and the False Rejection Rate (FRR). In the context of binary classification:

False Acceptance Rate (FAR) is the measure of the likelihood that the system incorrectly accepts an access attempt by an unauthorized user. In other words, it's the probability that a negative instance (a '0' label) is incorrectly classified as positive (a '1' label).

False Rejection Rate (FRR) is the likelihood that the system incorrectly rejects an access attempt by an authorized user. That is, it's the probability that a positive instance (a '1' label) is incorrectly classified as negative (a '0' label).

The code below defines the 'hter loss' as:

```
# Define a custom loss function (half total error rate loss)
def hter_loss(outputs, targets):
    targets = targets.view(-1, 1) # Change the shape of targets to [batch_size, 1]
    probabilities = torch.sigmoid(outputs)
    predictions = torch.round(probabilities)
    errors = torch.abs(predictions - targets)
    false_acceptance = torch.mean(errors * (1 - targets))
    false_rejection = torch.mean(errors * targets)
    half_total_error = (false_acceptance + false_rejection) / 2.0
    return half_total_error
```

Explaining the above function:

- The outputs argument are the raw prediction values from the model, typically on a real number scale.
- The targets argument are the true labels for the batch, which are either 0 or 1.
- The predictions are obtained by applying the sigmoid function to the outputs to get probabilities, and then rounding them to the nearest integer (0 or 1).
- The errors are the absolute differences between the predictions and the true labels.
- The false acceptance rate is computed as the mean of the errors for the instances where the true label is 0.
- The false rejection rate is computed as the mean of the errors for the instances where the true label is 1.
- The half total error rate is the average of the false acceptance rate and the false rejection rate.

The models:

Keeping in view the characteristics of the dataset the following 3 models were adapted for the classification task. The pytorch module was adapted for building the codes. Tools such as **data augmentation**, random selection of images for **under sampling** of data, tensor board and **early stopping** were adapted.

- **ResNet50**, short for Residual Networks with 50 layers, is a deep learning model that addresses the problem of training very deep neural networks. Traditional neural networks often suffer from the vanishing gradient problem as the depth of the network increases, hampering the network's learning capability. However, ResNet50 overcomes this problem using residual or "shortcut" connections, which allow gradients to be backpropagated to earlier layers without being diminished (He, K., Zhang, X., Ren, S., & Sun, J., 2015, "Deep Residual Learning for Image Recognition", <https://arxiv.org/abs/1512.03385>).
- **FaceNet** is another advancement in the realm of face recognition. Developed by researchers at Google, FaceNet directly learns a mapping of face images to a compact Euclidean space, where distances correspond to a measure of face similarity. Rather than focusing on the classification, it emphasizes the embedding, providing a unified embedding for face recognition, verification, and clustering tasks. Its performance in terms of accuracy has set new records in standard benchmarks (Schroff, F., Kalenichenko, D., & Philbin, J., 2015, "FaceNet: A Unified Embedding for Face Recognition and Clustering", <https://arxiv.org/abs/1503.03832>).

- The most recent breakthrough in image classification is the emergence of **Vision Transformers (ViT)**. These models extend the transformer architecture, primarily used in natural language processing tasks, to computer vision. Unlike traditional Convolutional Neural Networks (CNNs) that process images locally, ViTs treat an image as a sequence of patches and allow for global interactions between them. Although they require significant computation power and extensive training data, ViTs have demonstrated remarkable performance in image classification tasks (Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N., 2020, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", <https://arxiv.org/abs/2010.11929>).

Model Architecture:

Please note all codes are stored in the 'src' folder of the git repository [https://github.com/FahadKhalid1/Image_classification_exercise.git].

ResNet50:

I am fine-tuning a pre-trained ResNet-50 model for a custom classification task with two classes. I load the pre-trained model, replace its final fully connected layer with new layers for my classification task, and freeze most of the existing layers to retain the pre-trained knowledge. Only the deeper layers in layer4 are allowed to adapt during training to better suit my specific task. This approach helps me leverage the knowledge already captured by the pre-trained model while adapting it for my custom classification problem. [Code below]

```
model = models.resnet50(weights='ResNet50_Weights.DEFAULT')
num_features = model.fc.in_features
# Create additional layers for your custom classification task
additional_layers = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Linear(256, 2),
)
# Combine the pre-trained model and additional layers
model.fc = additional_layers
# Freeze all layers except for Stage 4
for name, param in model.named_parameters():
    if not name.startswith('layer4'):
        param.requires_grad = False
    else:
        param.requires_grad = True
```

FaceNet (modified):

I have defined a custom neural network called FaceNet model for a specific face classification task with two classes. The architecture includes convolutional layers, batch normalization, ReLU activation, and max-pooling layers. The model takes RGB images as input and

processes them through these layers. After adaptive average pooling, the features are flattened and passed through dropout and two fully connected layers. The first FC layer reduces the feature dimension to 128, and the second FC layer maps the embeddings to the two classes. Finally, I have instantiated the model and moved it to the available device for training and inference. (Code next page)

```
class FaceNetModel(nn.Module):
    def __init__(self, embedding_dimension=128, num_classes=2):
        super(FaceNetModel, self).__init__()

        # Define own architecture based on FaceNet
        # Here is a simplified version
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(3, stride=2, padding=1)

        self.conv2 = nn.Conv2d(64, 64, kernel_size=1, stride=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.relu2 = nn.ReLU()

        self.conv3 = nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(192)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(3, stride=2, padding=1)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout()
        self.fc1 = nn.Linear(192, embedding_dimension)
        self.fc2 = nn.Linear(embedding_dimension, num_classes)

    def forward(self, x):
        x = self.conv1(x)

def forward(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu1(x)
    x = self.pool1(x)

    x = self.conv2(x)
    x = self.bn2(x)
    x = self.relu2(x)

    x = self.conv3(x)
    x = self.bn3(x)
    x = self.relu3(x)
    x = self.pool3(x)

    x = self.avgpool(x)
    x = torch.flatten(x, 1)

    x = self.dropout(x)
    x = self.fc1(x)
    x = self.fc2(x)

    return x
```

Vision Transformers:

I am creating a Vision Transformer (ViT) model using the `create_model` function with the architecture `'vit_base_patch16_224'`. The model is initialized with pre-trained weights and adapted for a custom classification task with two classes. The `num_classes` argument is set to 2 to match the number of classes in the new task. I then move the model to the appropriate device for training and inference. Next, I freeze all the parameters of the ViT model by setting `requires_grad` to `False` for each parameter in the model. This prevents the pre-trained weights from being updated during training. However, I want to fine-tune the classification head of the ViT model, so I set `requires_grad` to `True` for the parameters in the head module. This allows the classification head to learn and adapt to the new task while keeping the pre-trained weights in the rest of the model fixed. Finally, I use the `summary` function to display a summary of the model architecture, including the input and output sizes of each layer. The

input_size argument is set to (64, 3, 224, 224), indicating that the model expects input with a batch size of 64, 3 color channels (RGB), and an image size of 224x224 pixels.

```
# The model is pre-trained on a large dataset, likely ImageNet
# The number of output classes is set to 2, which suggests a binary classification task
model = create_model('vit_base_patch16_224', pretrained=True, num_classes=2).to(device)

# For the parameters in the head of the model, we enable gradients.
# The 'head' is typically the final fully connected layer in the model which maps the extracted
# features to the output classes.
# By enabling gradients here, we are allowing these parameters to be updated during training.
# This is a common strategy for transfer learning: the base model is frozen and used as a fixed
# feature extractor, while the final classification layer is trained on the new data.



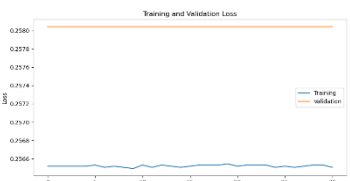
for param in model.parameters():
    param.requires_grad = False

for param in model.head.parameters():
    param.requires_grad = True

# This includes the number and types of layers in the model,
# as well as the number of parameters.
# The 'input_size' argument specifies the size of the input that the model expects.
# In this case, the model expects input of size (64, 3, 224, 224) which represents a
# mini-batch of 64 images, each of size 224x224 with 3 color channels (RGB).
summary(model, input_size=(64, 3, 224, 224))
```

Training parameters and loss curves:

The table below displays the model characteristics and training performances.

Models	Data	Parameters	Loss curves
ResNet50	<p><u>Under sampling was done to balance the dataset</u></p> <p>training set: 19364 validation set: 4841 Class count set: {0: 9714, 1: 9650} Class in validation set: {0: 2388, 1: 2453}</p>	<ul style="list-style-type: none"> num_epochs = 500 early_stopping_limit = 55 criterion = nn.CrossEntropyLoss() optimizer = optim.Adam(model.parameters(), lr=5e-5) 	
FaceNet	<p>Augmentation was done on the minority class.</p> <p>training set: 140636 validation set: 35160. Class count set: {0: 70366, 1: 70270} Class in validation set: {0: 17532, 1: 17628}</p>	<ul style="list-style-type: none"> num_epochs = 500 early_stopping_limit = 55 criterion = nn.CrossEntropyLoss() optimizer = optim.Adam(model.parameters(), lr=5e-5) 	
Vision Transformers	<p><u>Under sampling was done to balance the dataset</u></p> <p>training set: 19364 validation set: 4841 Class count set: {0: 9620, 1: 9744} Class in validation set: {0: 2482, 1: 2359}</p>	<ul style="list-style-type: none"> num_epochs = 500 early_stopping_limit = 55 criterion = nn.CrossEntropyLoss() optimizer = optim.Adam(model.parameters(), lr=5e-5) 	

Results and conclusion:

In the initial attempts, the models did not converge effectively, and there could be several underlying reasons contributing to this behavior. Due to the time constraints set for the completion of the task, further investigation and improvement efforts were not feasible. As a result, the decision was made to select the ResNet50 fine-tuned model to perform the prediction task and submit the results.

For instance, the utilization of the **Vision Transformer** model in one of the approaches presents a constraint where it only accepts images of size 224 by 224. This limitation could potentially lead to the loss of significant information, which in turn may have hindered the model's convergence. Furthermore, due to the heavy nature of Vision Transformers, down sampling of the dataset was necessary to accommodate computational limitations. However, it is important to acknowledge that image augmentation for class balancing and preserving the original dataset size could yield valuable insights and potentially different outcomes. Pre-processing the images before feeding them to the model could also play a crucial role in improving the results. Conducting such experiments in the future could provide deeper understanding and potentially enhance the performance of the classification task.