Deep Learning

Practical Exercises - Week 14

Practical Methodology

ICAI

FS 2024

1 Topic

Over the course of this semester, we have learned how to create and train an entire model ourselves. However, in many cases it is better to adopt existing, pre-trained neural networks to solve a particular task (don't reinvent the wheel). But how could a network which was trained with images of flowers, animals, and groceries be used as a gesture classifier? In practice this works as follows:

- There are many high-performance pre-trained neural networks that are publicly available, including their weight values (e.g., VGG16/19, EfficientNet or ResNet). These pre-trained networks are used as feature extractors, i.e. we use the feature maps after the convolution part as input features for our classifier. Thus, from the pre-trained network we copy only the first part (from the input layer to the last convolutional layer).
- The output of the pre-trained network is used as input to the classifier which is often a shallow feed-forward network. During training only the weights of feed-forward classifier are updated.

This way of learning a neural network is often referred to as transfer learning.

2 Transfer Learning

In this week's lab session we will build and train a convolutional neural network to classify three different hand gestures: *Rock*, *Paper* and *Scissors*. Thereby, we will use the pre-trained ResNet50 as feature extractor / backbone network. This convolutional neural

network was trained as image classifier on the ImageNet dataset (see this link for more information on ResNet). On top of the ResNet convolutional layers we stack three fully-connected layers, which we train using the rock_paper_scissors dataset from Laurence Moroney. At the end, you will be able to test this gesture recognition algorithm by using your own webcam (at home) or the lab cam (in our lab room).

2.1 Prepare Data

Everything related to loading, preprocessing and augmenting the data is already prepared in the Python module `data.py`. The `DataLoaderRPS` class creates the train, valid and test subsets, shuffles the dataset, creates mini-batches and performs data augmentation. The augmentation part comprises zooming and rotating the images. Feel free to add more augmentation methods as you like (see here if you need some inspiration). The `DataLoaderRPS` class also implements a method which displays 16 sample images of the first validation mini-batch. If you call this method you will see an output like in 1.

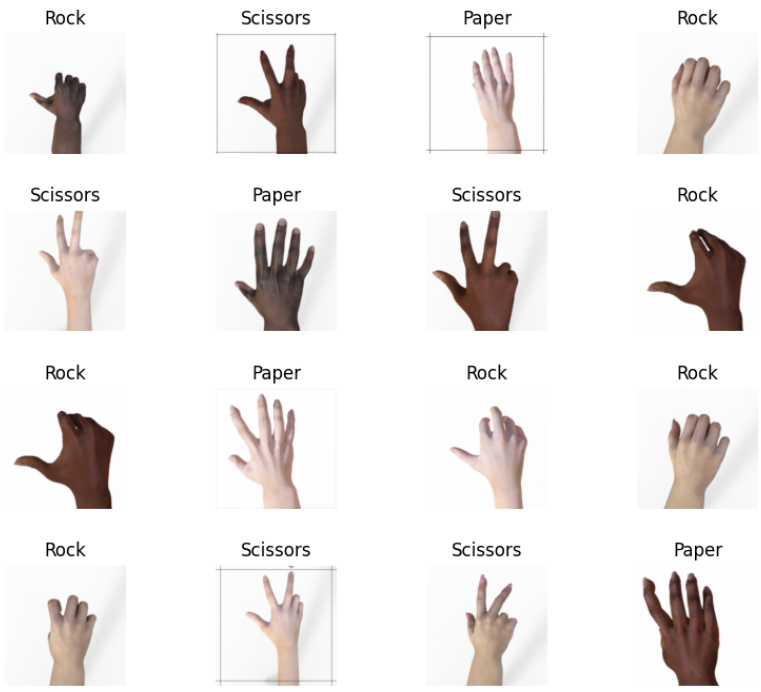

Figure 1: Example images of the rock_paper_scissors dataset.

From these examples you can see that the dataset contains only images with white background. So we can already say that our classifier will probably not generalize well to images with a different background (make sure that you have a white background when testing the classifier with your webcam).

2.2 CNN Model

As always, the neural network is defined in the Python module `model.py`. In this subchapter you will be guided step by step through the methodology of transfer learning. Using this guide as inspiration, you will then need to implement the model in the `model.py` module.

Each pre-trained model in `tf.keras.applications` expects the pixel values of the input image in a specific range and the color channels in a certain order. To ensure that we meet these requirements we call the corresponding `preprocess_input()` function.

```
def __init__(self, **kwargs):
    ...
    self.preprocess_layer = tf.keras.applications.resnet.preprocess_input
    ...
def call(self, x, training=False):
    ...
    x = self.preprocess_layer(x)
    ...
```

Next, we need to define the base model using one of the pre-trained convnets in TensorFlow's model zoo. When the ResNet50 model is instantiated, we have to decide at which layer to extract the feature maps. Here we follow the common practice of using the very last layer before the flattening operation. By specifying the `include_top=False` argument, we load a network that does not include the classification layers at the top. Also, by setting the `weights` argument to `'imagenet'`, we specify that the network uses the weights resulting from training ResNet50 on the ImageNet dataset. Furthermore, we need to declare the ResNet50 weights as non-trainable by setting the object attribute `self.backbone.trainable` to `False`, since we only want to train the head of our network.

```
def __init__(self, **kwargs):
    ...
    IMAGE_SHAPE = (300, 300, 3)
    self.backbone = tf.keras.applications.ResNet50(
        include_top=False, weights="imagenet", input_shape=IMAGE_SHAPE
    )
    self.backbone.trainable = False
    ...
def call(self, x, training=False):
    ...
    x = self.backbone(x)
    ...
```

The output of our base model is a tensor of shape `(None,10,10,2048)`. We will reduce this to a feature vector of shape `(None,1,1,2048)` by averaging over the spacial dimensions (reduce each of the 2048 feature maps of size 10x10 to one number by averaging over all 100 values). We can do this by using the layer `GlobalAveragePooling2D` as follows:

```
    def __init__(self, **kwargs):
    ...
    self.avg_pool_1 = GlobalAveragePooling2D()
    ...
```

```
def call(self, x, training=False):
    ...
    x = self.avg_pool_1(x)
    ...
```

Based on this 2048-dimensional feature vector we will classify each image as *Rock*, *Paper* or *Scissors*. We accomplish this by adding a classification head consisting of three Dense layers as well as three Dropout layers. By now you know how to implement such a fully-connected neural network. If you need some hints, check out lab 10/11 which were about deep feedforward networks.

2.3 Training

The training, validation and testing procedure is implemented in the Python script `main.py`. In this regard, the following comments:

- The `from_logits` argument of the loss function is set to `True`, which means that the classifier is expected to output the pre-softmax activations (no activation function for the last dense layer).
- Two callback functions are passed to the `model.fit()` function. One callback function logs some metrics which can be displayed with TensorBoard and the other callback function stores the model weights if the validation accuracy increases during training. You need these weights so that you can test your model with your webcam later.
- At the end, the model is tested with the best performing network weights (according to the validation accuracy). You should achieve an accuracy of approximately 73%.

2.4 Evaluation

The Python script `evaluate.py` prepares everything you need to apply the trained classifier to your webcam images. This comprises the following parts:

- Access and read your webcam using OpenCV's `VideoCapture` class.
- Instantiate an object of your model class which is defined in `model.py` and load the weights which were save during training in `./ckpt`.
- The webcam images are cropped to a size of 300x300 pixels and classified by the network. The predicted class written to the webcam image.
- You may need to play a little with the background to get good classification results.

3 Exercise

Now it is your task to implement the image classifier as described above. More specifically, you are asked to complete the following subtasks:

- Define the model following the description of 2.2.
- Train the model for 50 epochs by running the Python script `main.py`. During training you can see how the model improves by observing the development of the metrics on TensorBoard (`tensorboard --logdir ./logs` and `localhost:6006`).
- Test the real-time performance of your model by applying it to your webcam images (see 2.4). You might have to change the `DEVICE_INDEX` constant. Does it classify your hand gestures correctly?
- Try to improve the performance by making some modifications to the model or training process. Here some ideas:
  - Add more dense layers at the end of the model.
  - Test other data augmentation techniques.
  - Try to fine tune the pre-trained base model (see here if you want to know how).