Model and Transfer Learning

Ramin Zarebidoky (LiterallyTheOne)

12 Nov 2025

Model and Transfer Learning

Introduction

In the previous tutorial, we have loaded our selected **Kaggle** dataset into **train**, **validation**, and **test** subsets. Then, we have made a `DataLoader` for each subset. The summary of our code looks like below:

```
path = kagglehub.dataset_download(
 ↪  "balabaskar/tom-and-jerry-image-classification")

data_path = Path(path) / "tom_and_jerry/tom_and_jerry"

trs = transforms.Compose(
    [
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
    ]
)

all_data = ImageFolder(data_path, transform=trs)

g1 = torch.Generator().manual_seed(20)
train_data, val_data, test_data = random_split(all_data, [0.7,
 ↪  0.2, 0.1], g1)

train_loader = DataLoader(train_data, batch_size=12,
 ↪  shuffle=True)
val_loader = DataLoader(val_data, batch_size=12, shuffle=False)
test_loader = DataLoader(test_data, batch_size=12, shuffle=False)
```

As you might have noticed, we only changed the scale of our `Resize` transform to (224, 224) to make it one of the standard sizes for images in deep learning. In this tutorial, we will learn about how to define a model in **Keras**. Then, we will improve our results using a technique called **Transfer Learning**

Model in Keras

There are 3 ways to define a model in **Keras**.

- Sequential
- Functional
- Subclassing

All these three ways have their own use-cases. `Sequential` is one of the cleanest and best ways of defining a model which we will learn it in this session. As the name suggests, it would take a sequence of layers. Then, pass the data through them in order and generate the output. To define it in **Keras**, we can use this code.

```
model = keras.Sequential(
    [
    ],
)
```

It requires a list of layers, which we are going to talk about them very shortly.

> **Sources**: * https://keras.io/api/models/model/ * https://keras.io/api/models/sequential/ * https://keras.io/guides/sequential_model/

Input layer

Input layer is the layer that we use to tell **Keras** what the shape of our input is. For example, for the shape of `(3, 224, 224)`, we can use the code below:

```
input_layer = keras.layers.Input(shape=(3, 224, 224))
```

So let's add it to our sequential model:

```
model = keras.Sequential(
    [
        keras.layers.Input(shape=(3, 224, 224)),
    ],
)
```

Dense layer

`Dense layer` (fully connected layer) is a layer that all the neurons of this layer is connected to the neurons of the previous layer. To define a `Dense layer` in `Keras` we can simpy use `keras.layers.Dense`. It requires the number of the neurons. Also, we can optionally give it the activation function. For example, if we want to have 10 neurons with the `ReLU` activation, we can use the code below:

```
dense_layer = keras.layers.Dense(10, activation="relu")
```

Source: https://keras.io/api/layers/core_layers/dense/

Output layer

`Output layer` is the layer that we use to generate our output respect to our problem. In **classification** problems we mostly use `Dense layer` with `softmax` as its activation. For example, if we have 4 classes we can define an output layer like below:

```
keras.layers.Dense(4, activation="softmax"),
```

Now, let's add it to our sequential model:

```
model = keras.Sequential(
    [
        keras.layers.Input(shape=(3, 224, 224)),
        keras.layers.Dense(4, activation="softmax"),
    ],
)
```

Flatten layer

`Flatten layer` is simply flatten the output of the previous layer. If we have `5` data that their shape is `(8, 9)`, the output of a `flatten layer` would be `5` data with the shape of `(72,)`. To use a flatten layer we can use the code below:

```
flatten_layer = keras.layers.Flatten()
```

Since the output of our input layer is `(3, 224, 224)`, we should flatten this output to give it to our `dense layer`. So let's add our `flatten layer` to our sequential model like this.

```
model = keras.Sequential(
    [
        keras.layers.Input(shape=(3, 224, 224)),
        keras.layers.Flatten(),
        keras.layers.Dense(4, activation="softmax"),
    ],
)
```

Source: https://keras.io/api/layers/reshaping_layers/flatten/

Compile

`compile` is the function that we use to determine our `loss function`, `optimizer` and `metrics`. These functions are necessary in the training procedure, and we are going to talk about them individually in the next tutorials. But for now, we can use the code below to compile our model.

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

Model Details

So far, we have successfully created a model and defined its **optimizer**, **loss function**, and **metrics**. Now, let's learn about how to see the model's details. To do so, we can use a function called `summary`.

```
print(model.summary())

"""
--------
output:

Model: "sequential_3"

 Layer (type)                    Output Shape
↪  Param #

 flatten (Flatten)               (None, 150528)
↪  0

 dense_1 (Dense)                 (None, 4)
↪  602,116

 Total params: 602,116 (2.30 MB)
 Trainable params: 602,116 (2.30 MB)
 Non-trainable params: 0 (0.00 B)
"""
```

As you can see, with `summary`, we can see the layers, total parameters, trainable parameters, and non-trainable parameters. For our model, after we `flatten` the input ($3 \times 224 \times 224 = 150528$) we have 150528 neurons. When we fully connect it to 4 neurons, we would have ($150528 \times 4 = 602112$) weights and 4 biases to train.

Now, let's give one batch of our data to the model, and see the output. Our

`batch_size` was 12, and we have four classes, so we expect that our output shape to be `[12, 4]`.

```
for images, labels in train_loader:
    result = model(images)
    print(result.shape)
    break


"""
--------
output:

torch.Size([12, 4])
"""
```

As expected, our output matches our prediction.

Train the model

To train the model, we can use a function called `fit`. We can use this function like below:

```
history = model.fit(train_loader, epochs=5,
 ↪  validation_data=val_loader)

"""
--------
output:

Epoch 1/5
320/320             19s 59ms/step - accuracy: 0.3536 - loss:
 ↪  10.3647 - val_accuracy: 0.3449 - val_loss: 10.5572
Epoch 2/5
320/320             17s 55ms/step - accuracy: 0.3544 - loss:
 ↪  10.3956 - val_accuracy: 0.3449 - val_loss: 10.5387
Epoch 3/5
320/320             18s 55ms/step - accuracy: 0.3546 - loss:
 ↪  10.3916 - val_accuracy: 0.3449 - val_loss: 10.5626
Epoch 4/5
320/320             17s 53ms/step - accuracy: 0.3541 - loss:
 ↪  10.4005 - val_accuracy: 0.3449 - val_loss: 10.5625
Epoch 5/5
320/320             17s 53ms/step - accuracy: 0.3541 - loss:
 ↪  10.4005 - val_accuracy: 0.3449 - val_loss: 10.5624
"""
```

As you can see, we gave our `train_loader` for its first argument. Then, we said how many times we want it to iterate all over our data. We have determined that by an argument called `epochs`. As you can see, we set the number of `epochs` to 5. And, finally we gave our `val_loader` to an argument called `validation_data`. So, after each epoch ends, we will have a report on the **validation** subset. This function, returns a history that we can use it for plotting and reporting that we are going to learn about that in the upcoming tutorials. As you can see int the results, our accuracy and loss is not improving. This indicates that our model is not learning correctly. Before we fix that, let's learn how to `evaluate` our model on the **test** subset.

Source: https://keras.io/api/models/model_training_apis/

Evaluate the model

To evaluate our model on a given dataset, we can use a function called `evaluate`. We can use this function like below:

```
loss, accuracy = model.evaluate(test_loader)

print("loss:", loss)
print("accuracy:", accuracy)


"""
--------
output:
46/46             2s 44ms/step - accuracy: 0.3638 - loss: 10.2543
loss: 10.254292488098145
accuracy: 0.36380255222320557
"""
```

As you can see, we have evaluated our model on our `test_loader`. As its output, it would return the loss and the metrics that we have defined in the `compile` function.

Transfer Learning

Transfer learning is one of the most common techniques in **Deep Learning**. In this technique we use pretrained model (called base model), on a new dataset with a different purpose. We only use the `base_model` as a feature extractor, and we won't train it. Only the layers that we manually add will be trained. To get prepared for the transfer learning:

- Load the model without its classification layers
- Put the training of the base model to `False`
- Change the input layer according to the dataset input
- Change the output layer according to the number of classes

For example, let's load a model called `MobileNetV2` as our `base_model`, and put its `trainable` to `False`.

```
from keras.applications import MobileNetV2

base_model = MobileNetV2(include_top=False, input_shape=(224,
 ↪  224, 3))

base_model.trainable = False
```

In the code above, we have used `keras.applications` to import `MobileNetV2`. `MobileNetV2` is one of the most used and most famous models used for **Transfer Learning**. It is light and has a really great generalization. The default dataset that `MobileNetV2` is trained on is **ImageNet**. As you can see, we put the `include_top` to `False`. This removes the `classification` layers, so we can replace them with our own. We also set the `input_shape` to `(224, 224, 3)` which is the standard of **ImageNet** images. There are some pretrained models available in **Keras** which you can find them in the link below:

Different models available in **Keras**: https://keras.io/api/applications/

Permute layer

As you might have noticed, our images has a shape like `(3, 224, 224)`. But our `base_model` accepts shape of `(224, 224, 3)`. So, to fix that problem, we can use a layer called `permute`. This layer, helps us to reshape our images to the standard our `base_model` has. Let's define a `permute` layer that changes the input in a way that we want.

```
p = layers.Permute((2, 3, 1))
```

As you can see, in the code above, we have defined a permute layer. As its argument, we gave it the new position that we want our output to be. Our input was `(channel, height, width)` (`(3, 224, 224)`), we want it to become `(height, width, channel)` (`(224, 224, 3)`). So to do so, we should put the **2nd** dimension (height) at the **1st** place. Then, put the **3rd** dimension (width) at the **2nd** place. And finally, put the **1st** dimension (channel) at the **3rd** place. The result of our repositioning is like this: `(2, 3, 1)`. Now, let's test our layer with the one batch of our images.

```
for images, labels in train_loader:
    print(f"result shape: {p(images).shape}")
    break

"""
--------
output:
```

```
result shape: torch.Size([12, 224, 224, 3])
"""
```

As you can see, the output is what we expected.

Apply Transfer Learning

Now, let's add a `permute` layer and our `base_model` in the middle our previous model. The code should look like below:

```
model = keras.Sequential(
    [
        layers.Input(shape=(3, 224, 224)),
        layers.Permute((2, 3, 1)),
        base_model,
        layers.Flatten(),
        layers.Dense(4, activation="softmax"),
    ]
)
```

For the next step, let's `compile` it like before:

```
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

Now, let's see out model's detail.

```
print(model.summary())


"""
--------
output:

Model: "sequential_4"

 Layer (type)                        Output Shape
 ↪  Param #

 permute_1 (Permute)                 (None, 224, 224, 3)
 ↪  0

 mobilenetv2_1.00_224                (None, 7, 7, 1280)
 ↪  2,257,984
```

```
 (Functional)
↪

 flatten_1 (Flatten)              (None, 62720)
↪  0

 dense_2 (Dense)                  (None, 4)
↪  250,884

 Total params: 2,508,868 (9.57 MB)
 Trainable params: 250,884 (980.02 KB)
 Non-trainable params: 2,257,984 (8.61 MB)
"""
```

As you can see, now we can see the new layers that we added with the number of parameters that they have. Right now, we have $2,508,868$ number of parameters. As you can see, because we are not going to train our `base_model`, we only have $250,884$ trainable parameters. Now, let's fit our model to see if our results have improved or not.

```
history = model.fit(train_loader, epochs=5,
↪  validation_data=[val_loader])

"""
--------
output:

Epoch 1/5
320/320             40s 125ms/step - accuracy: 0.3252 - loss:
↪  10.4311 - val_accuracy: 0.4133 - val_loss: 8.8533
Epoch 2/5
320/320             42s 133ms/step - accuracy: 0.4383 - loss:
↪  8.7707 - val_accuracy: 0.4434 - val_loss: 8.7051
Epoch 3/5
320/320             46s 145ms/step - accuracy: 0.4634 - loss:
↪  8.3851 - val_accuracy: 0.4653 - val_loss: 8.2721
Epoch 4/5
320/320             45s 142ms/step - accuracy: 0.5014 - loss:
↪  7.8171 - val_accuracy: 0.5046 - val_loss: 7.7342
Epoch 5/5
320/320             47s 146ms/step - accuracy: 0.5291 - loss:
↪  7.4392 - val_accuracy: 0.5301 - val_loss: 7.3093
"""
```

As you can see, we got a better accuracy and loss. In each step, our loss, in both training and validation subsets, is decreasing, which means our model is

learning correctly. For the next step, let's evaluate our model on the **test** subset as well.

```
loss, accuracy = model.evaluate(test_loader)

print("loss:", loss)
print("accuracy:", accuracy)

"""
--------
output:

46/46             5s 111ms/step - accuracy: 0.4845 - loss: 8.0948
loss: 8.0947847366333
accuracy: 0.4844606816768646
"""
```

As you can see, the result on our unseen data (test subset) has improved as well. But, we are going to improve our result much more in the upcoming tutorials.

Your turn

- Load your dataset in 3 subsets: **train**, **validation**, and **test**.
- Choose another model other than `MobileNetV2` as your base model.
  - You can use this link to see the other models
  - https://keras.io/api/applications/
- Set the input layer according to your data
- Set the output layer according to the number of the classes
- Use the transfer learning technique correctly
- Train your model on your train subset
  - You should fill `validation_data` argument
  - 5 epochs is enough
- Report your result on your test subset

Conclusion

In this tutorial, we learned about how to define a model in **Keras** and how to use a very popular **Deep Learning** technique, called **Transfer Learning**. First, we introduced the **Sequential** model. After that, we have learned about all the necessary layers and add them to our **Sequential** model. Then, we learned about **Transfer Learning**. We used a `MobileNetV2` as our `base_model` and trained again. We saw that results have improved.