

# keras

Ramin Zarebidoky (LiterallyTheOne)

26 Nov 2025

## Introduction

### Keras

**Keras** is a high-level API for building and training **Deep Learning Models**. It is designed to be a stand-alone project. But with the help of **TensorFlow**, **PyTorch**, and **Jax**, It can run on top of different hardware (e.g., **CPU**, **GPU**).

The fascinating thing about **Keras** is that it is super easy to get started with. You can train and test a model with only a few lines of code. It is a perfect way to learn **Deep Learning** concepts by practically seeing their effects.

### Google Colab

There are so many ways available to run a **Deep Learning** code. One of the fastest and easiest way that doesn't require any installation, is **Google Colab**. Google colab is a free cloud-based platform that is powered by **jupyter notebook**. All the packages that we want for this tutorial is already installed in **Google Colab**. Also, every code that we run in this tutorial can be run on this platform. So, I highly recommend you to start with **Google Colab**. After you have become more comfortable with the packages and concepts, switch to a local platform like your personal computer.

All the codes that we talk about in this tutorial is available in the **GitHub**. Each tutorial has a link to its respective code, which you can find it at the bottom of each page. To load and run the codes in **Google Colab**, you can follow these steps.

- Open Google colab
- From **files** select **Open Notebook**
- Go to the **GitHub** section
- Copy the **URL** of the code
- Select the **.ipynb** file that you want

Here is an example of loading this tutorial's code:

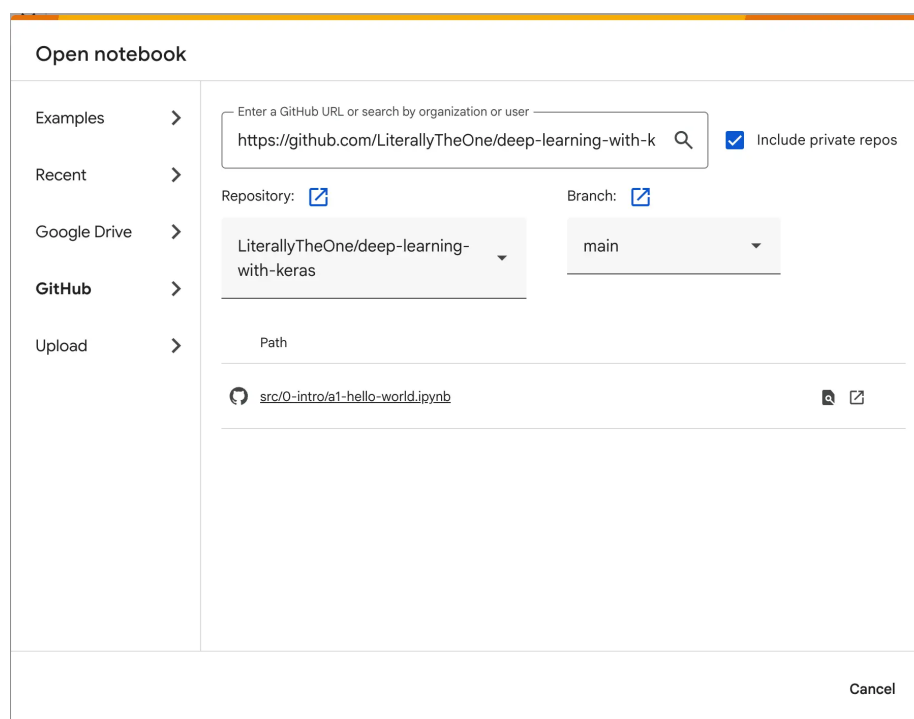


Figure 1: Colab GitHub

## Hello World

Here is a **Hello World** example that we are gradually going to complete it step by step.

```
# Setup
import os

os.environ["KERAS_BACKEND"] = "torch"

# Imports
from keras.datasets import mnist
import keras
from keras import layers

# Prepare the Data
(train_images, train_labels), (test_images, test_labels) =
    ↪ mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

# Define the model
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])

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

# Train the model
model.fit(train_images, train_labels, epochs=5, batch_size=128)

# Test the model
test_loss, test_acc = model.evaluate(test_images, test_labels)

"""
-----
output:

Epoch 1/5
469/469          2s 5ms/step - accuracy: 0.9259 - loss: 0.2622
Epoch 2/5
```

```

469/469          2s 5ms/step - accuracy: 0.9685 - loss: 0.1092
Epoch 3/5
469/469          2s 5ms/step - accuracy: 0.9797 - loss: 0.0710
Epoch 4/5
469/469          2s 5ms/step - accuracy: 0.9852 - loss: 0.0515
Epoch 5/5
469/469          2s 5ms/step - accuracy: 0.9901 - loss: 0.0363
313/313          1s 3ms/step - accuracy: 0.9801 - loss: 0.0616
"""

```

In the code above, we have trained and tested a model on a dataset called **MNIST**. In the future, we are going deeper into each step, but for now, here is a simple explanation of each of them. At first, we set up the backend of our **Keras**. We set that to `torch`, but you can set that to either `tensorflow` or `jax`. Then we imported the necessary modules. After that, we downloaded MNIST. MNIST contains of  $28 \times 28$  images of handwritten digits between 0 and 9. Then, we normalize our data. After that, we defined a simple model and compiled the model with the proper `optimizer`, `loss`, and `metrics`. With the `fit` function, we train our model. And finally, we test our model with the evaluate function. As you can see in the output, our model's `accuracy` and `loss` are shown in each training step and testing step. We have gotten 99% accuracy on our training data and 98% accuracy on our test data.

## Kaggle

Kaggle is one of the biggest platforms for data science and machine learning enthusiasts. It contains a huge number of datasets and a variety of competitions. In this tutorial, we are going to select an **Image Classification Dataset** from Kaggle. One of the simplest ways to do that is to go to the **Datasets** section in **Kaggle**, and select **Image Classification** tag in the **filters**. The dataset that we have to choose should have stored its images in a format like below:

```

class_a/
...a_image_1.jpg
...a_image_2.jpg
class_b/
...b_image_1.jpg
...b_image_2.jpg

```

As you can see, in the format above, we have some directories with images. Each directory represents a class, and the images in each directory belong to that class.

You can see the format of a **Dataset** by scrolling down to **Data Explorer**. For example, in Tom and Jerry Image classification We have a format as below:

As you can see, we have 4 directories (`jerry`, `tom`, `tom_jerry_0`, `tom_jerry_1`),

## Data Explorer

Version 3 (469.3 MB)









- ▼  tom\_and\_jerry
  - ▼  tom\_and\_jerry
    - ▶  jerry
    - ▶  tom
    - ▶  tom\_jerry\_0
    - ▶  tom\_jerry\_1
-  challenges.csv
-  ground\_truth.csv

Figure 2: Tom and Jerry data format

and each directory has its own images. So, we have 4 classes. Another example is Facial Emotion Recognition Dataset. Its data structure is as below:

## Data Explorer

Version 1 (208.62 MB)

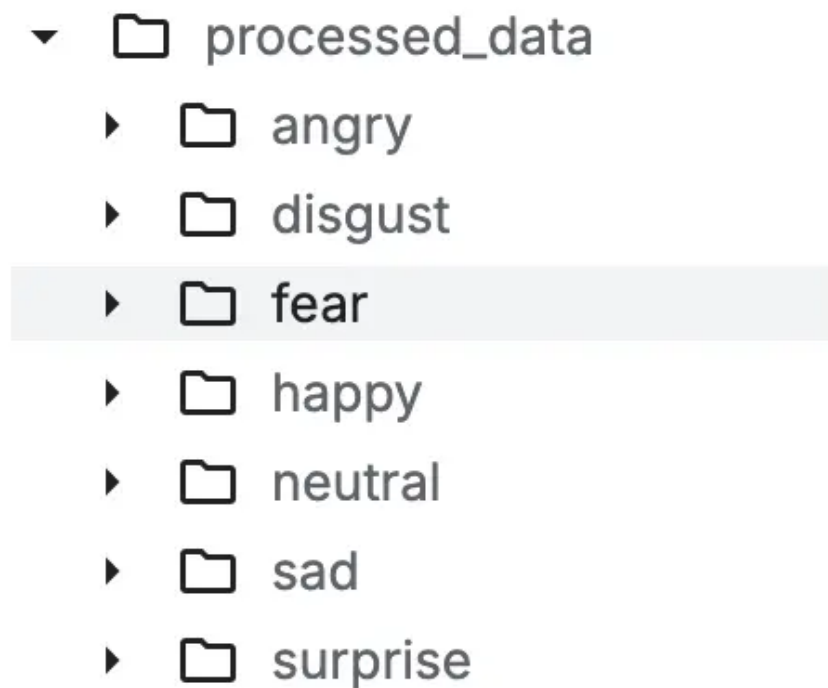


Figure 3: Facial Emotion data format

As you can see, in the image above, we have 7 directories (*angry*, *disgust*, *fear*, *happy*, *neutral*, *sad*, and *surprise*). So, we have 7 classes.

Now, you should select a dataset with these criteria:

- Image classification
- Each class has its own directory, and its images are in that directory
- It's better for our dataset size not to exceed 5GB.

## Conclusion

In this tutorial, we have introduced **Keras**. Then, we explained about **Google Colab** and how to load a notebook from **GitHub**. After that, we provided a **Hello World** example that we are going to complete it overtime. Finally, we introduced **Kaggle** and explained how to get a suitable **Dataset** from it for this tutorial.

## Load an Image Classification Dataset

### Introduction

In the previous tutorial, we learned how about **Keras**, **Google Colab**, and **Kaggle**. Our task was to select an **Image Classification Dataset** from **Kaggle**. In this tutorial, we are going to load this dataset and make it ready to give it to a model.

### Get data from Kaggle

The easiest and the recommended way to download a dataset from **Kaggle** is to use a package called **Kagglehub**. **Kaggle** itself has developed this package and made it super easy to use. You can learn more about this package in their GitHub Repository.

Now, how to use this package to download a dataset. In the dataset that you have selected, click on the **Download** button in the top right corner of the page. A window will pop up that has a code snippet on it. You should copy that code and use it in your own code. For Tom and Jerry Image classification, it is like this:

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download(
    ↪ "balabaskar/tom-and-jerry-image-classification")

print("Path to dataset files:", path)
```

The code above, will automatically download the dataset and returns its path. We said that we wanted a structure like below:

```
class_a/
...a_image_1.jpg
...a_image_2.jpg
class_b/
...b_image_1.jpg
...b_image_2.jpg
```

We know that this dataset has this structure and if you looked at the dataset in **Kaggle**, you have noticed that it is in `tom_and_jerry/tom_and_jerry` directory. But get more familiar with the **jupyter notebook** commands, let's find it with taking the list of the path that we are currently on.

```
!ls {path}

"""
-----
output:

challenges.csv    ground_truth.csv tom_and_jerry
"""
```

As you can see, we have `tom_and_jerry` directory. Now, let's take the list of this directory.

```
!ls {path}/tom_and_jerry

"""
-----
output:

tom_and_jerry
"""
```

As you can see, we have another `tom_and_jerry` directory. Let's take the list of it to see what's inside of it.

```
!ls {path}/tom_and_jerry/tom_and_jerry

"""
-----
output:

jerry          tom          tom_jerry_0 tom_jerry_1
"""
```

And as you can see, we have reached to the structure that we wanted. Let's put this path in a variable called `data_path`, to be able to use it later.

```
from pathlib import Path

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

Your dataset might have subdirectories like `train`, `validation` and `test`. If it was like this put the `train` directory in the `data_path` and store the other ones in their respective directory. For example, `val_path` for validation and `test_path` for test.

## ImageFolder

One of the best ways to use an **Image Classification Dataset** in **PyTorch** is by using `ImageFolder`. `ImageFolder` loads and assigns labels to a folder that has this structure:

```
main_directory/
...class_a/
.....a_image_1.jpg
.....a_image_2.jpg
...class_b/
.....b_image_1.jpg
.....b_image_2.jpg
```

This structure is the structure that we have right now in our `data_path` variable. Now, let's load our image folder and show one of the images.

```
from torchvision.datasets import ImageFolder
from matplotlib import pyplot as plt

all_data = ImageFolder(data_path)

for image, label in all_data:
    plt.figure()
    plt.imshow(image)
    print(label)
    break

"""
-----
output:

0
"""
```

As you can see, in the code above, we have loaded our images using `ImageFolder` and stored it in a variable called `all_data`. After that, we used a `for` to iterate through `images` and `labels`. We showed one image and one label and used `break` to end our loop. As it shown, the label is 0 and you can see the image representing that label in the above.

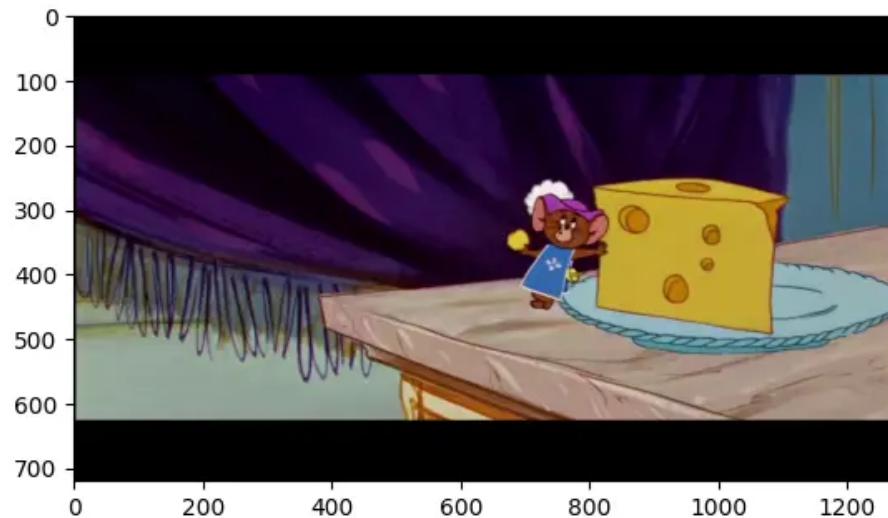


Figure 4: tom-and-jerry-example

## Transforms

**Transforms** are the way that we can transform our images to the standard that we want. For example, when we load our dataset, the images might have different sizes. But when we want to train or test our model, we want images to have the same size. To make this happen, we can use the `transforms` module in `torchvision`. For example, let's load our dataset without a resize transform with `resize transform` and see the difference.

```
from torchvision import transforms

# Without resize transform

all_data = ImageFolder(data_path)

for image, label in all_data:
    print(f"image size without resize transform: {image.size}")
    break

# With resize transform

transform = transforms.Resize((90, 160))

all_data = ImageFolder(data_path, transform=transform)

for image, label in all_data:
```

```

    print(f"image size with resize transform: {image.size}")
    break

"""
-----
output:

image size without resize transform: (1280, 720)
image size with resize transform: (160, 90)
"""

```

As you can see, in the code above, we have successfully changed the size of our images to (160, 90).

Another thing is, when we load our images with `ImageFolder`, it would load them as PIL images. But when we want to feed our images to our model, we want them to be tensors. To achieve that, `torchvision` has a `transform` that take an image and turns it into a `tensor`. To have resize and transforming to tensor transforms, we can combine them with each other like below:

```

trs = transforms.Compose(
    [
        transforms.Resize((90, 160)),
        transforms.ToTensor(),
    ]
)

all_data = ImageFolder(data_path, transform=trs)

for image, label in all_data:
    print(type(image))
    print(image.shape)
    break

"""
-----
output:

<class 'torch.Tensor'>
torch.Size([3, 90, 160])
"""

```

As you can see, we have our data in tensor, also the size of it is what we want it.

## Split into train, validation, test

Some of our datasets don't have **train**, **validation**, **test** subsets. So, to split our data into these 3 subsets, we can use a function called `random_split`. This function, takes a `Dataset`, a sequence of `lengths` to split our data, and an optional `generator`. Here is an example on how to use `random_split`:

```
import torch
from torch.utils.data import random_split

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

print(f"all_data's size: {len(all_data)}")
print(f"train_data's size: {len(train_data)}")
print(f"val_data's size: {len(val_data)}")
print(f"test_data's size: {len(test_data)}")

"""
-----
output:

all_data's size: 5478
train_data's size: 3835
val_data's size: 1096
test_data's size: 547
"""
```

In the code above, first we defined a `generator` with its seed set to 20. The reason for that is that we want every time that we run our code, have the same **train**, **validation**, and **test** subsets. Then, we used `random_split` function. for the first argument, we gave it `all_data` that we loaded it before. After that, we should give it a list of percentages or lengths. If we give it the percentages, sum of them should be equal to 1.0. If we give them the lengths, sum of them should be equal to the length of our data. For example `[0.7, 0.2, 0.1]` means to split data into 70, 20, and 10. We use that 70 for our training. We use 20 for validation. We use 10 for test. For the third argument, we gave the generator that we created earlier. As you can see in the result, we had 5478 samples, and we split them into train, validation, and test subsets. 3835 of them are for training, 1096 of them are for validation, and 547 of them are for testing.

For a **Deep Learning** project, we need these 3 subsets. If the **Dataset** provider hasn't split them already, we should split it. Otherwise, there is nothing to do.

## DataLoader

Now, we have successfully loaded our dataset into tensors. Also, we have train, validation, and test subsets. Now, we are ready to feed them into our **model** for training and testing purposes. To make this procedure easier, **PyTorch** has a module called **DataLoader**. **Dataloader** takes a loaded dataset as its argument and helps us to apply the **Deep learning** techniques. One these techniques is called **mini-batch**. So, instead of feeding our data to our model one by one, we give it a **batch** of data. For example, each time we give it **12** data. It helps our model to learn better. Another technique is called **shuffling**. By **shuffling**, we change the order of data when we want to feed it to the model. It helps the model to learn more generally. To use **DataLoader** with these 2 techniques, we can use the code below:

```
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)
```

In the code above, we have 3 dataloaders for each train, validation, and test subsets. Then, we set the **batch\_size** to 12 and for the train subset we set the **shuffle** to **true**. Now, let's show one batch of training data using **DataLoader**.

```
fig, axes = plt.subplots(3, 4)

axes_ravel = axes.ravel()

for images, labels in train_loader:
    for i, (image, label) in enumerate(zip(images, labels)):
        axes_ravel[i].imshow(transforms.ToPILImage()(image))
        axes_ravel[i].set_axis_off()
        axes_ravel[i].set_title(f"{label}")
    break
```

Output:

In the code above, we made a subplot with 3 rows and 4 columns. Then, we **ravel** it to make it a one dimensional array. This helps to use only one index instead of two. After that, we iterate thorough our **train\_loader**. It would give us 12 images and 12 labels. Then we iterate through those images and labels and show them. As you recall, our images were in **tensor** format. To bring them back to **PIL** format, we can use a transform called **ToPILImage**. As you can see in the output, we have 12 different images with their respective label on top of them.



Figure 5: batch-tom-and-jerry

## Your turn

Now, it is your turn. First, get your **Kaggle** dataset. Then, use the **ImageFolder** to load that dataset and show one of its images. After that, if you don't have any of the **train**, **validation**, and **test** subsets, make them using **random\_split**. Then, load those three subsets using **DataLoader** and set a **batch\_size** for them. Finally, show a batch of your data.

## Conclusion

In this tutorial, we have learned how to work with a dataset. At first, we got an **Image classification** dataset from **Kaggle** using **Kagglehub**. Then, we loaded that dataset using **ImageFolder**. After that, we learned how to split our data if our dataset doesn't contain **train**, **validation**, and **test** subsets. Finally, we used **DataLoader** to load our data with **Deep Learning** techniques.

## 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/](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. Here is an example of a **Dense layer** with 4 neurons which are connected to 8 input neurons.

To define a **Dense layer** in **Keras** we can simply 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/](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:

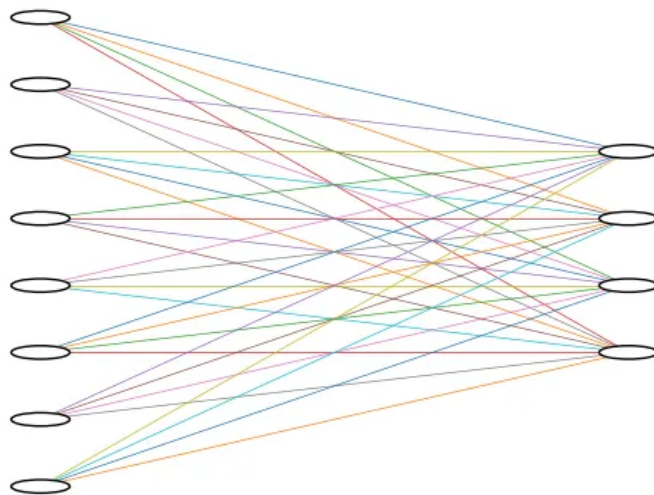


Figure 6: Dense Layer Example

```
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/](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
  ───────────  ───────────
  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 in 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/](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.

## Plot and TensorBoard

### Introduction

In the previous tutorial, we learned about **model** and **Transfer Learning**. Here is the summary of the code that we have implemented so far.

```
import os

os.environ["KERAS_BACKEND"] = "torch"
from pathlib import Path

from matplotlib import pyplot as plt

import torch
from torch.utils.data import random_split, DataLoader

from torchvision.datasets import ImageFolder
from torchvision import transforms

import keras
```

```

from keras import layers
from keras.applications import MobileNetV2

import kagglehub

import datetime

# Load the Dataset

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)

# Create the model

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

base_model.trainable = False

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

```

```

    ]
)

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

# Train the model

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

# Evaluate the model

loss, accuracy = model.evaluate(test_loader)

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

```

As you can see, in the code above, when we were training our model using `.fit` function, we were storing its result in a variable called `history`. In this tutorial, we will learn more about `history` and how to plot its results. Also, we will learn about a very powerful tool for plotting and seeing the results during training, called **TensorBoard**.

## Plot the training history

First, let's print the history to see what is inside it.

```

print(history)

"""
-----
output:

<keras.src.callbacks.history.History object at 0x12de7e300>
"""

```

As you can see, we have a `Callback` with the name of `History`. The `History` object, saves the information about the training parameters, in an attribute called `params`.

```

print(history.params)

"""

```

```

-----
output:

{'verbose': 'auto', 'epochs': 5, 'steps': 320}
"""

```

As you can see, we have trained our model for 5 epochs and each epoch contained 320 steps (mini-batches). Also, `History` saves the `loss` and the given `metrics` (in our case: `Accuracy`) as well, in an attribute called `history`.

```

print(history.history)

-----
output:

{'accuracy': [0.41955670714378357, 0.5418513417243958,
↪ 0.5614081025123596, 0.5921773314476013, 0.6033898591995239],
'loss': [8.836509704589844, 7.081783294677734,
↪ 6.813899517059326, 6.346843242645264, 6.222221374511719],
'val_accuracy': [0.540145993232727, 0.5246350169181824,
↪ 0.5729926824569702, 0.6076642274856567, 0.5894160866737366],
'val_loss': [7.019584655761719, 7.262048721313477,
↪ 6.537136554718018, 6.103211879730225, 6.454712390899658]}
"""

```

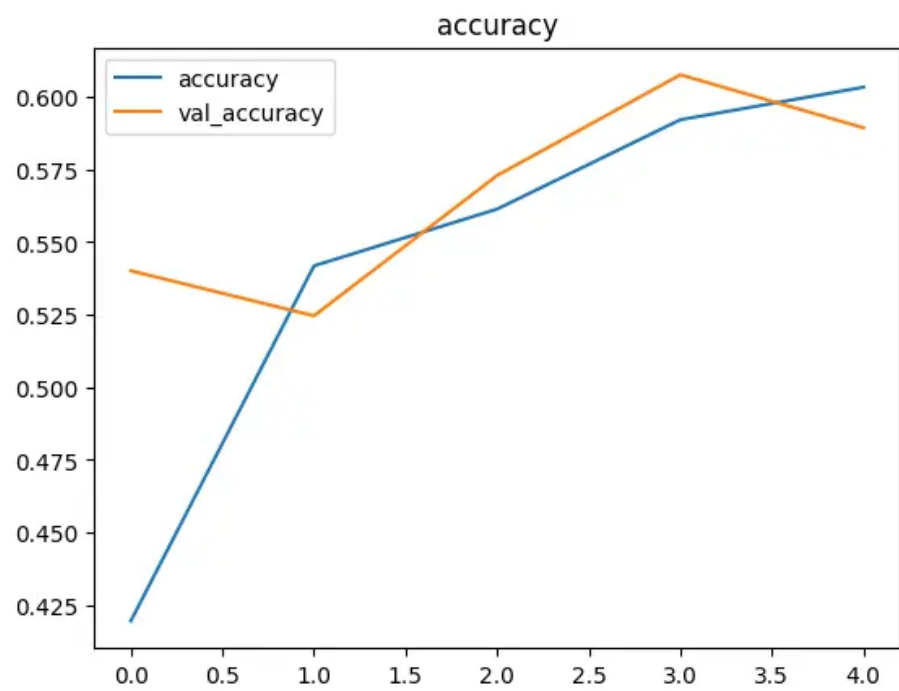
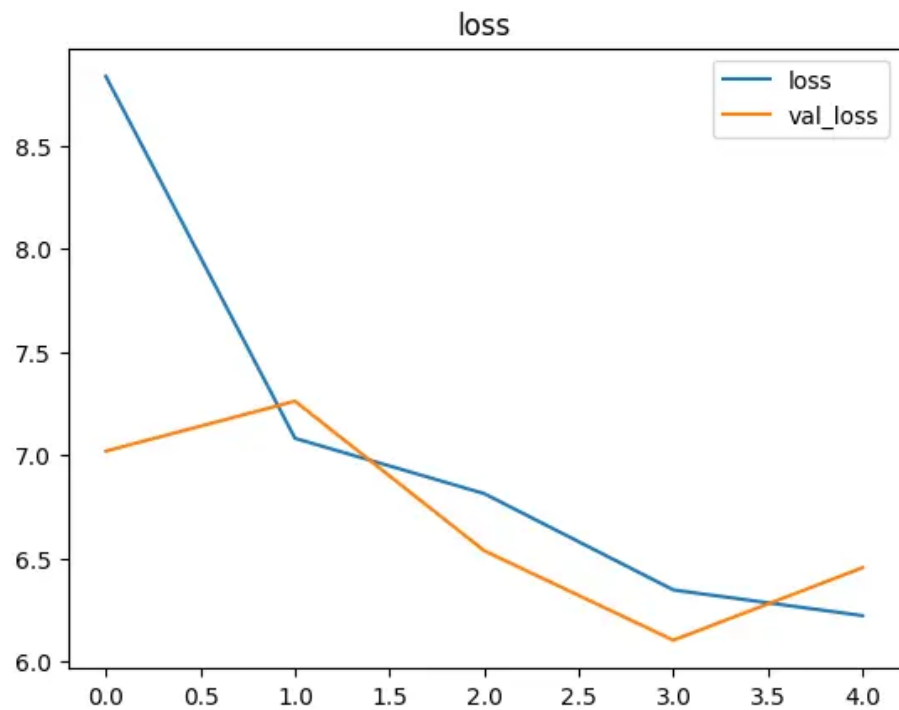
As shown, now we can access to `loss` and `accuracy` of **training** and **validation** in each epoch. So, let's plot the `loss` and `accuracy` separately.

```

plt.figure()
plt.title("loss")
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.legend(["loss", "val_loss"])

plt.figure()
plt.title("accuracy")
plt.plot(history.history["accuracy"])
plt.plot(history.history["val_accuracy"])
plt.legend(["accuracy", "val_accuracy"])

```



As you can see, we have trained our model for 5 epochs. For training subset, our **loss** and **accuracy** were improving (blue line). But for the validation subset, we had some ups and downs which is natural. We are going to learn how to analyze them in the upcoming tutorials.

Source: [https://www.tensorflow.org/api\\_docs/python/tf/keras/callbacks/History](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/History)

## TensorBoard

If we want to plot our results using **History callback** from the output of the **fit** module, we have to wait until the training is done. So, we can't have live plots and data to analyze our training procedure. One of the ways that we can solve this problem is by logging the data during training in our hard drive. Then, use a **UI** to load that log and analyze it. That's exactly what **TensorBoard** does.

**TensorBoard** is an open-source visualization toolkit for machine learning experiments. It has its own logging standard and visualization dashboard. Anytime that we log something in our code, we can see that log on the dashboard. **TensorBoard** is widely used and is one of the standard ways to log and share our training procedure. So, now we have two steps to take:

- Use our standard **TensorBoard** logging when we **fit** our model
- Open the **UI Dashboard** and see the result

### Add TensorBoard to the code

To add **TensorBoard** logging in our training procedure, **Keras** has provided us a **Callback**. We can create a new object of that **Callback** using the code below:

```
log_dir = "logs/fit/" +  
    ↪ datetime.datetime.now().strftime("%Y%m%d-%H%M%S")  
tensorboard_callback =  
    ↪ keras.callbacks.TensorBoard(log_dir=log_dir)
```

In the code above, at first we have created the destination path that we want to store our logs. The standard that we used is putting all the logs in the parent directory called **logs/fit** and name each of them based on the time that they are created. For example: **logs/fit/20251118-092033**. Then we created a new **TensorBoard** object with passing one argument to it. **log\_dir** is the destination path that our logs would be stored which we filled it with the directory name that we have created earlier.

Now, it's time to give our **TensorBoard Callback** to the **fit** function. To do so, we can use an argument called **callbacks** in the **fit** function. This argument takes a list of **Callbacks**. So, the only thing that we should do, is to add our **tensorboard\_callback** to the **callbacks** like below:

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

Now, when we fit our model, the training logs would be saved at `logs/fit`.

**Source:** <https://keras.io/api/callbacks/tensorboard/>

### TensorBoard dashboard

Now, let's open up the **TensorBoard dashboard**. The code to do that is like below:

```
tensorboard --logdir logs/fit

"""
-----
output:

Serving TensorBoard on localhost; to expose to the network, use a
↳ proxy or pass --bind_all
TensorBoard 2.20.0 at http://localhost:6006/ (Press CTRL+C to
↳ quit)
"""
```

This code would make a local host, and you can access its dashboard through web browser. Here is an example of a dashboard in a web browser.

If you want to load the **TensorBoard dashboard** in your **Jupyter notebook**, you should first load it with the code below:

```
%load_ext
tensorboard
```

And then run the loading code:

```
%tensorboard --logdir
logs
```

The output of the respective cell would work interactively, and you can access the dashboard. Now, let's get deeper into the **Scalars tab** in **TensorBoard dashboard**. We are going to learn about the other tabs in the future tutorials.

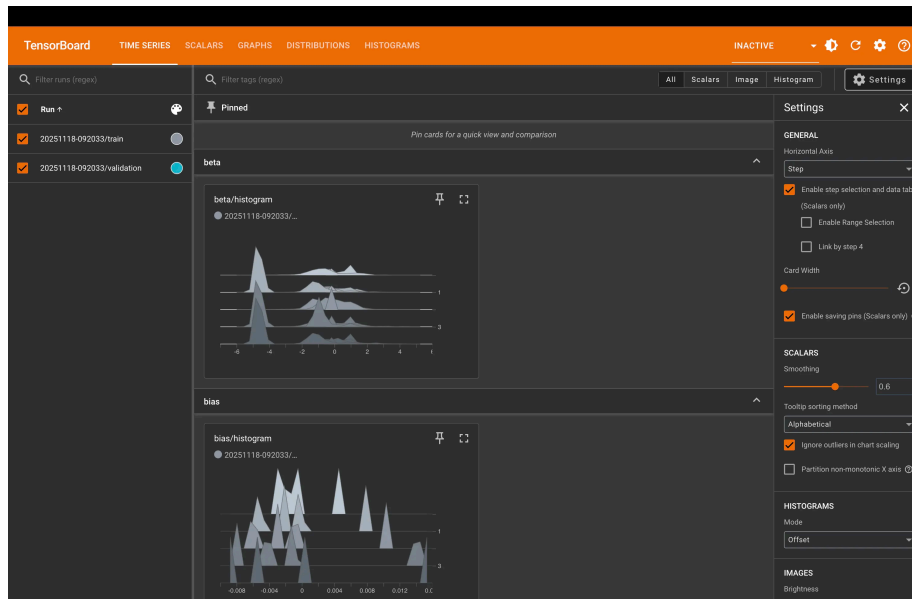


Figure 7: tensorboard dashboard

## Scalars tab

**Scalars tab** contains the plots of our loss and metrics.

As you can see, in the image above, we have 5 different sections:

- **epoch\_accuracy**
- **epoch\_learning\_rate**
- **epoch\_loss**
- **evaluation\_accuracy\_vs\_iteration**
- **evaluation\_loss\_vs\_iteration**

These sections can be opened to see the validation and train plots. In the left panel, we can select the run that we want. We might have trained our model multiple times, we can select the respective run to see the results. Also, for each run, results of train and validation are being stored separately. We can choose one of them to see its result.

## Load tensorboard files in python

There are sometimes that we want to create clean figures of our training procedure in python. In order to do so, we can use our **Tensorboard** logs. They already have the training information that we wanted. To load and use them the most straight forward method, is by using a package called **tbparse**. To use it, we can use the code below:

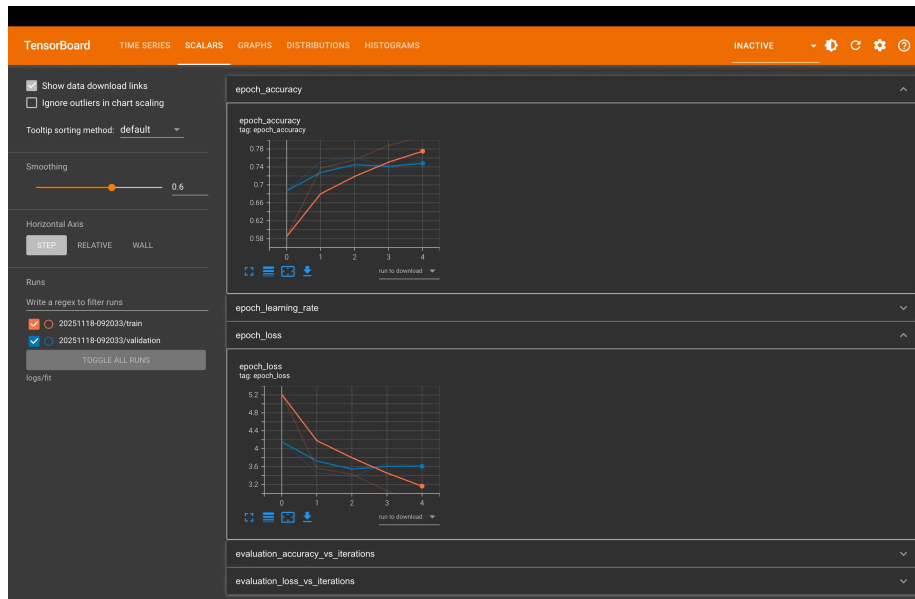


Figure 8: scalars tab

```
from tbparse import SummaryReader

log_dir = "your/log/dir"

reader = SummaryReader(log_dir)
df = reader.tensors

loss = df[df["tag"] == "epoch_loss"]
accuracy = df[df["tag"] == "epoch_accuracy"]
```

In the code above, first we imported `SummaryReader`. Then we created a `SummaryReader` object with the given `log_dir`. Because we used **PyTorch**, as our backend, our data is stored in `tensors`. So, we put them in a variable called `df`. To get the `loss`, we only should get the data which their tag is `epoch_loss`. And for the `accuracy`, we have a tag called `epoch_accuracy`.

Now, we can plot them simply. Here is an example of plotting the loss of loaded **Tensorboard** log.

```
from matplotlib import pyplot as plt

plt.figure()
plt.title("loss")
plt.plot(loss["step"], loss["value"], label="training loss")
```

We can also, load multiple **Tensorboard** logs and plot them together. You can find an example in the code that we provided in **GitHub**. (link to the code can be found in the page)

## Your turn

- Draw accuracy and loss plots
  - You should include train and validation on each one of them
- Add Tensorboard to your training procedure

## Conclusion

In this tutorial, we learned about plotting our training procedure. First, we explained the **History** object that `.fit` function returns. Then, we used its data to plot our results. Second, we address that to get the **History** object, we should wait for the `.fit` function to finishes its job. To see the result's online during training, we learned that we can use **Tensorboard**. After that, we added tensorboard to our training procedure. Finally, we learned about the **TensorBoard dashboard** and **Scalars tab**.

# Loss and Optimization

## Introduction

In the previous tutorial, we learned about **plotting** and **Tensorboard**. Here is the summary of the code that we have implemented so far.

```
# -----[ Setup ]-----
import os

os.environ["KERAS_BACKEND"] = "torch"

# -----[ Imports ]-----
from pathlib import Path

from matplotlib import pyplot as plt

import torch
from torch.utils.data import random_split, DataLoader

from torchvision.datasets import ImageFolder
from torchvision import transforms
```

```

import keras
from keras import layers
from keras.applications import MobileNetV2

import kagglehub

import datetime

# -----[ Load the data ]-----
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)

# -----[ Make the model ]-----
base_model = MobileNetV2(include_top=False, input_shape=(224,
    ↪ 224, 3))

base_model.trainable = False

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

```

```

)

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

# -----[ Train the model ]-----
log_dir = "logs/fit/" +
    ↪ datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback =
    ↪ keras.callbacks.TensorBoard(log_dir=log_dir)

history = model.fit(
    train_loader,
    epochs=5,
    validation_data=val_loader,
    callbacks=[tensorboard_callback],
)

# -----[ Evaluate the model ]-----

loss, accuracy = model.evaluate(test_loader)

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

# -----[ Plot the training procedure
    ↪ ]-----

plt.figure()
plt.title("loss")
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.legend(["loss", "val_loss"])

plt.figure()
plt.title("accuracy")
plt.plot(history.history["accuracy"])
plt.plot(history.history["val_accuracy"])
plt.legend(["accuracy", "val_accuracy"])

plt.show()

```

In this tutorial, we are going to learn more about **loss functions** and **optimiz-**

ers in Keras.

**Your turn**

**Conclusion**

## Convolutional models

### Introduction

In the previous tutorial, we learned about different loss functions and optimizers. In this tutorial, we will learn more about convolutional models that we were using in the previous tutorials.

### Convolution

Convolution is an operation in which we slide a smaller matrix (kernel) over a bigger matrix and calculate the weighted sum. Let's explain its concepts using an example. In our example, we have a  $6 \times 6$  image, and our kernel is  $3 \times 3$ , like below:

```
image_size = (6, 6)
kernel_size = (3, 3)

image = np.arange(image_size[0] *
    ↪ image_size[1]).reshape(image_size)
kernel = np.ones(kernel_size) / (kernel_size[0] * kernel_size[1])

print("image:")
print(image)
print("kernel:")
print(kernel)

"""
-----
output:

image:
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]
kernel:
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]
```

```
[0.11111111 0.11111111 0.11111111]]
"""
```

As you can see, our image is the numbers from 0 to 35, and our kernel is working as an average kernel. If we apply convolution, we are going to have a result like below:

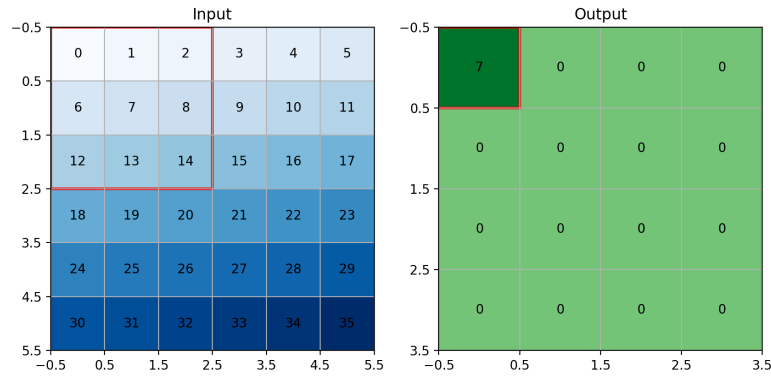


Figure 9: conv

As you can see in the GIF above, the kernel is being slid on our image, and we are getting the average of each  $3 \times 3$  block as an output. Let's calculate the first block.

$$0 \times \frac{1}{9} + 1 \times \frac{1}{9} + 2 \times \frac{1}{9} + 6 \times \frac{1}{9} + 7 \times \frac{1}{9} + 8 \times \frac{1}{9} + 12 \times \frac{1}{9} + 13 \times \frac{1}{9} + 14 \times \frac{1}{9} = 7$$

As you can see, the calculations have the same results as the code. Also, our input's shape is  $6 \times 6$ , but our output's shape is  $4 \times 4$ . The reason behind that is our kernel is  $3 \times 3$ . So, we can only slide it 4 times on our input. For now, we can calculate it like below:

$$W_{out} = (W_{in} - K_w) + 1$$

$$H_{out} = (H_{in} - K_h) + 1$$

- W: Width
- H: Height
- K: Kernel

Now, let's talk about 3 important things in **Convolution**. If you want to experience different convolutions with different options, you can use this code: `conv_gif.py`.

## Stride

Right now, we are sliding our kernel 1 square at a time. If we decide to slide it with a number different from one, we can use **stride**.

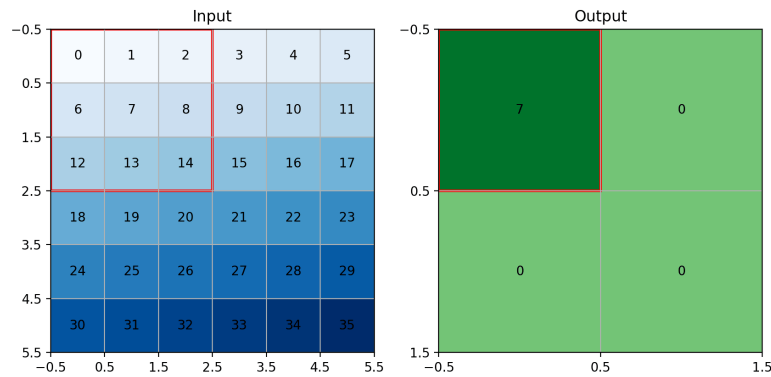


Figure 10: conv stride

As you can see in the GIF above, we put the stride to 2. So, it slides 2 squares instead of 1 in both x and y axis. As a result, our output's shape becomes half of what it was. We can calculate the output's shape as below:

$$W_{out} = \frac{(W_{in} - K_w)}{S_w} + 1$$

$$H_{out} = \frac{(H_{in} - K_h)}{S_h} + 1$$

- W: Width
- H: Height
- K: Kernel
- S: Stride

## padding

Padding is a technique that we use to fill the surrounding of the input with some values. The most common value for padding is 0, which is called **zero padding**. The main reason for that is to prevent our image from being shrunk

after some convolutions. In the previous example, you saw that the image with  $6 \times 6$  becomes  $4 \times 4$ . If the input shape and output shape are the same, it is called **zero-padding**.

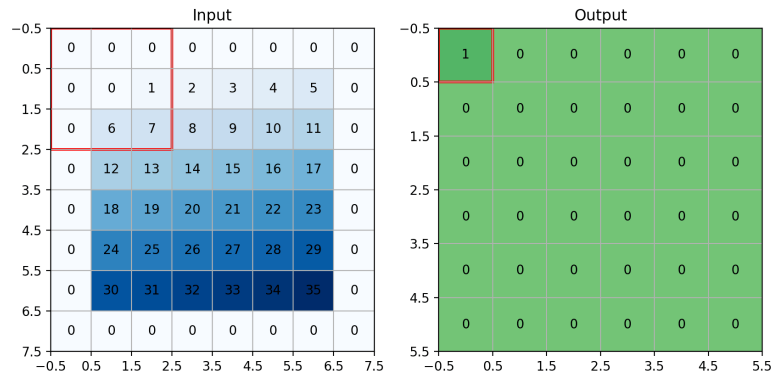


Figure 11: conv pad 1

As you can see in the GIF above, we have added zeros to the surroundings of our input. As a result, our output has the same shape as our input ( $6 \times 6$ ). We can calculate the output size as below:

$$W_{out} = \frac{(W_{in} + 2P_w - K_w)}{S_w} + 1$$

$$H_{out} = \frac{(H_{in} + 2P_h - K_h)}{S_h} + 1$$

- W: Width
- H: Height
- K: Kernel
- S: Stride
- P: Padding

### Dilation

Dilation is a technique that we use to make the kernel bigger to cover a bigger area. To do so, we insert gaps between our kernel. For example, if our kernel is like below:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

After `dilation=2`, it becomes like below:

$$\begin{bmatrix} 1 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 8 & 0 & 9 \end{bmatrix}$$

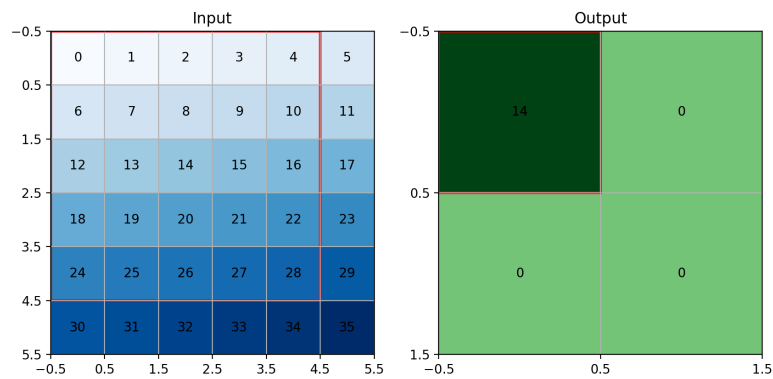


Figure 12: conv dilation 2

As you can see in the GIF above, we have `dilation=2`, so our kernel becomes  $5 \times 5$ . We can calculate the output shape with the formula below:

$$W_{out} = \frac{(W_{in} + 2P_w - D_w \times (K_w - 1) - 1)}{S_w} + 1$$

$$H_{out} = \frac{(H_{in} + 2P_h - D_h \times (K_h - 1) - 1)}{S_h} + 1$$

- W: Width
- H: Height
- K: Kernel
- S: Stride
- P: Padding
- D: Dilation

## Convolution layer

Earlier, we learned how `convolution` works. Now, let's talk about how to use it in **Keras**. We can define a `Convolution layer` in **Keras** like below:

```

from keras.layers import Conv2D

conv_1 = Conv2D(
    filters=64,
    kernel_size=(3, 3),
    padding="same",
    strides=(1, 1),
    dilation_rate=(1, 1),
)

```

In the code above, we have defined a **convolution layer**. For its output, it creates 64 channels. Also, it has a  $3 \times 3$  kernel. As you can see, we have control over **stride**, **padding**, and **dilation**. Now, let's feed our loaded images to `conv_1`, to see what happens.

```

for images, labels in train_loader:
    images = Permute((2, 3, 1))(images)
    result = conv_1(images)
    print(f"images shape: {images.shape}")
    print(f"result shape: {result.shape}")
    break

"""
-----
output:
images shape: torch.Size([12, 224, 224, 3])
result shape: torch.Size([12, 224, 224, 64])
"""

```

The results above show that the width and height of our inputs and outputs are the same. The reason behind that is that we put **padding** to **same**. Also, we have 64 channels for the results as expected. When we train a convolution layer, this kernels would be trained.

## Pooling

**Pooling** is a downsampling operation. It is mostly being used after feature extraction layers. For example, after a series of **convolution layers**. It basically reduces the spatial dimensions (height, width) while keeping the important information.

We are using a pooling layer mostly because:

- Compress information
- Avoid overfitting (which we are going to discuss in the upcoming tutorials)
- Achieving Translation Invariance
- Increase Receptive Field

The operation of **Pooling** is pretty similar to **Convolution**.

We have a specific **Kernel** that we are sliding it over a bigger matrix. The only difference between them, is that this **Kernel** is not trainable. Let's get more familiar with two important **Pooling** layers, **Average Pooling** and **Max pooling**.

### Average Pooling

**Average pooling** calculates the average of each window. Here is an example of defining and using an **Average pooling layer** in **Keras**.

```
from keras.layers import AveragePooling2D

avg_pooing_layer = AveragePooling2D((2, 2), strides=1)

a = np.arange(32, dtype=float).reshape(1, 4, 4, 2)

result = avg_pooing_layer(a).cpu().numpy()

print("differences in shapes")
print(a.shape)
print(result.shape)
print("-" * 20)

print("1st channel")
print(a[0, :, :, 0])
print("-" * 20)

print("second channel")
print(a[0, :, :, 1])
print("-" * 20)

print("result of the 1st channel")
print(result[0, :, :, 0])
print("-" * 20)

print("result of the 2nd channel")
print(result[0, :, :, 1])
print("-" * 20)

"""
-----
output:

differences in shapes
(1, 4, 4, 2)
(1, 3, 3, 2)
```

```

-----
1st channel
[[ 0.  2.  4.  6.]
 [ 8. 10. 12. 14.]
 [16. 18. 20. 22.]
 [24. 26. 28. 30.]]
-----

second channel
[[ 1.  3.  5.  7.]
 [ 9. 11. 13. 15.]
 [17. 19. 21. 23.]
 [25. 27. 29. 31.]]
-----

result of the 1st channel
[[ 5.  7.  9.]
 [13. 15. 17.]
 [21. 23. 25.]]
-----

result of the 2nd channel
[[ 6.  8. 10.]
 [14. 16. 18.]
 [22. 24. 26.]]
-----

"""

```

As you can see, in the code above, we have defined an **Average Pooling Layer** with the `pool_size` of (2, 2) and made sure that our `stride` is set to 1. After that, we made an input matrix with the size of  $4 \times 4$  that has 1 batch and 2 channels. The values of this matrix is filled by the numbers in range of [0, 31]. Then, we fed that input to our **Average pooling Layer** and printed the results. As you can see, in the result section, we can see the differences of the input and the output. First, let's examine the different shapes. The original shape is (1, 4, 4, 2) but the output's shape is (1, 3, 3, 2). The reason behind that is that we can fit  $3 \times 2 \times 2$  window on a  $4 \times 4$  matrix. As you can see, we have printed each channel and the output is the average over the  $2 \times 2$  window.

There is another common **Pooling layer** is being used as the last layer of our convolutional model (Instead of **Flatten**) is **Global Average Pooling**. This layer, calculates the average of the whole channel. Here is an example of **Global Average Pooling**.

```

from keras.layers import GlobalAveragePooling2D

avg_pooing_layer = GlobalAveragePooling2D()

```

```

a = np.arange(32, dtype=float).reshape(1, 4, 4, 2)

result = avg_pooling_layer(a).cpu().numpy()

print("difference in shapes")
print(a.shape)
print(result.shape)
print("-" * 20)

print("1st channel")
print(a[0, :, :, 0])
print("-" * 20)

print("second channel")
print(a[0, :, :, 1])
print("-" * 20)

print("result")
print(result)
print("-" * 20)

"""
-----
output:

difference in shapes
(1, 4, 4, 2)
(1, 2)
-----

1st channel
[[ 0.  2.  4.  6.]
 [ 8. 10. 12. 14.]
 [16. 18. 20. 22.]
 [24. 26. 28. 30.]]
-----

second channel
[[ 1.  3.  5.  7.]
 [ 9. 11. 13. 15.]
 [17. 19. 21. 23.]
 [25. 27. 29. 31.]]
-----

result
[[15. 16.]]
-----
"""

```

As you can see, in the code above, `GlobalAveragePooling2D` doesn't require a **Kernel**. Because, it would apply the average on the whole channel. As it shown in the outputs, the result shape is (1, 2) (batch\_size and channel). Also, you can see that the average of each channel is calculated.

## Max pooling

**Max Pooling** calculate the maximum of each window. Here is an example of **Max Pooling**.

```
from keras.layers import MaxPool2D

max_pooling_layer = MaxPool2D((2, 2), strides=1)

a = np.arange(32, dtype=float).reshape(1, 4, 4, 2)

result = max_pooling_layer(a).cpu().numpy()

print("differences in shapes")
print(a.shape)
print(result.shape)
print("-" * 20)

print("1st channel")
print(a[0, :, :, 0])
print("-" * 20)

print("second channel")
print(a[0, :, :, 1])
print("-" * 20)

print("result of the 1st channel")
print(result[0, :, :, 0])
print("-" * 20)

print("result of the 2nd channel")
print(result[0, :, :, 1])
print("-" * 20)

"""
-----
output:

differences in shapes
(1, 4, 4, 2)
(1, 3, 3, 2)
-----
```

```

1st channel
[[ 0.  2.  4.  6.]
 [ 8. 10. 12. 14.]
 [16. 18. 20. 22.]
 [24. 26. 28. 30.]]
-----

second channel
[[ 1.  3.  5.  7.]
 [ 9. 11. 13. 15.]
 [17. 19. 21. 23.]
 [25. 27. 29. 31.]]
-----

result of the 1st channel
[[10. 12. 14.]
 [18. 20. 22.]
 [26. 28. 30.]]
-----

result of the 2nd channel
[[11. 13. 15.]
 [19. 21. 23.]
 [27. 29. 31.]]
-----

"""

```

As you can see, in the code above, the syntax of **Max Pooling** is pretty similar to **Average Pooling**. The **Kernel** of our **Max Pooling** layer is also  $2 \times 2$  so the output shape would be the same as **Average pooling**. As you can see, in the output, the maximum of each window is calculated.

We have **Global Maximum Pooling** as well. Here is an example:

```

max_pooling_layer = GlobalMaxPooling2D()

a = np.arange(32, dtype=float).reshape(1, 4, 4, 2)

result = max_pooling_layer(a).cpu().numpy()

print("difference in shapes")
print(a.shape)
print(result.shape)
print("-" * 20)

print("1st channel")
print(a[0, :, :, 0])
print("-" * 20)

print("second channel")

```

```

print(a[0, :, :, 1])
print("-" * 20)

print("result")
print(result)
print("-" * 20)

"""
-----
output:

difference in shapes
(1, 4, 4, 2)
(1, 2)
-----

1st channel
[[ 0.  2.  4.  6.]
 [ 8. 10. 12. 14.]
 [16. 18. 20. 22.]
 [24. 26. 28. 30.]]
-----

second channel
[[ 1.  3.  5.  7.]
 [ 9. 11. 13. 15.]
 [17. 19. 21. 23.]
 [25. 27. 29. 31.]]
-----

result
[[30. 31.]]
-----
"""

```

As you can see, the maximum of each channel is calculated

## Activation Function

In a Neural Networks, **Activation functions** removes the linearity of the connections. Without an **Activation Function**, the model is only a linear transformation. It helps the model to learn complex relationships. In **Keras**, we can give an activation function to the layer. This was the method that we were using in the previous tutorials.

We have so many different **Activation functions**. Here are one of the most used ones.

## ReLU

ReLU stands for **Rectified Linear Unit**. It is one of the most used activation functions in **Deep Learning**. The logic behind that is pretty simple. It only changes the negative values to 0. Here is its formula:

$$ReLU(x) = \max(0, x)$$

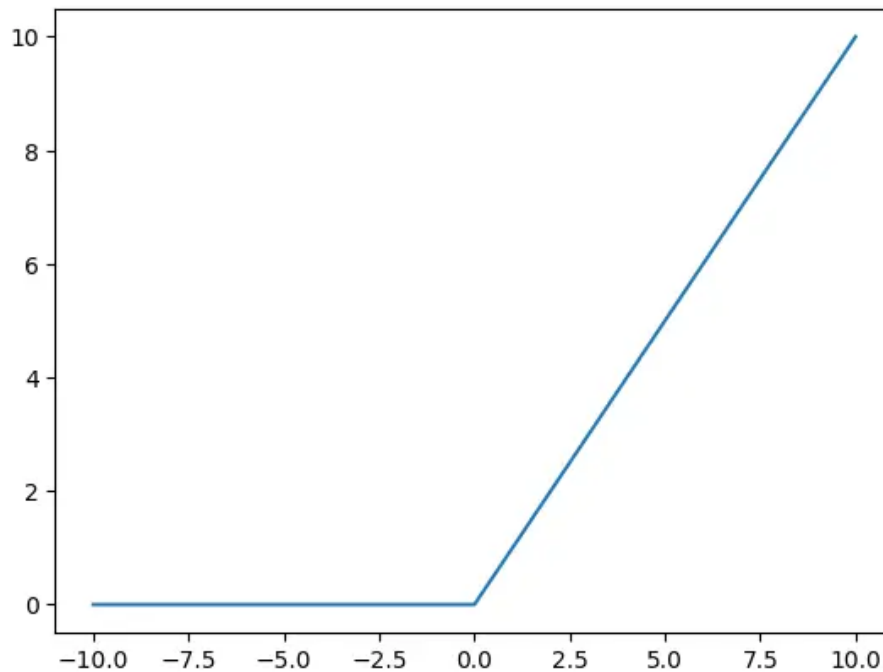


Figure 13: ReLU

You can access the **ReLU** function like below:

```
from keras.activations import relu
```

## Tanh

**Hyperbolic Tangent function** is another super useful **Activation Functions**. It maps its input into the range of  $(-1, 1)$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

You can access the **Tanh** function like below:

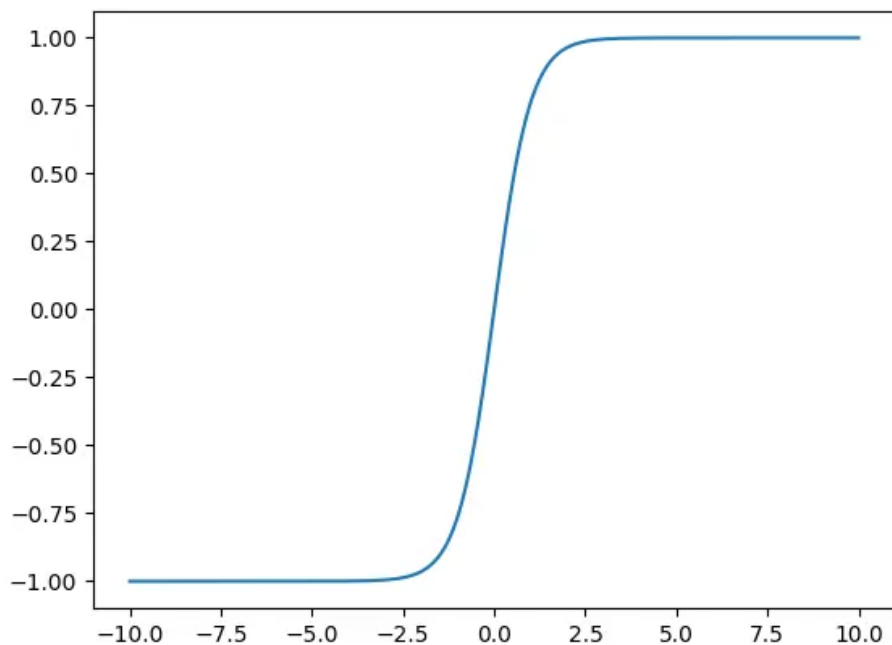


Figure 14: Tanh

```
from keras.activations import tanh
```

### Sigmoid

**Sigmoid** is another function that is mostly used in binary classification. It would map the input into the range of  $(0, 1)$ .

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

You can access the **Sigmoid** function like below:

```
from keras.activations import sigmoid
```

### Softmax

**Softmax** is mostly used as the **Activation function** of the final layer of classification. It would change the logits to the probability.

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

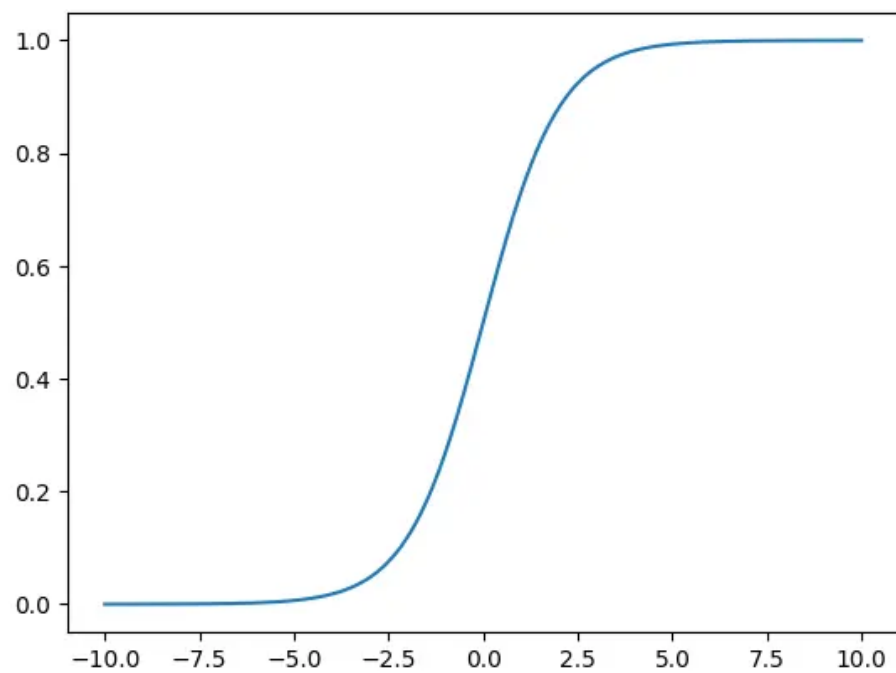


Figure 15: Sigmoid

You can access the **Softmax** function like below:

```
from keras.activations import softmax
```

## Dropout layer

**Dropout layer** is a technique that is used to avoid overfitting and achieve regularization. It accepts a percentage as its input. In each training step, it would set that percentage of neurons to zero. This helps the other neurons to get included in the training procedure.

Here is an example of the usage of the **Dropout layer**.

```
from keras import Sequential
from keras.layers import Dropout, Dense, Input

model = Sequential(
    [
        Input(shape=(1024,)),
        Dropout(0.2),
        Dense(512, activation="relu"),
        Dropout(0.2),
        Dense(128, activation="relu"),
        Dropout(0.2),
        Dense(10, activation="relu"),
    ]
)
```

## LeNet

Now that we have learned about the layers that are mostly used in **Convolutional Neural Networks**, let's build some of them from scratch. One of the most effective and simple models is **LeNet**. **LeNet** is designed to recognize the handwritten digits on grayscale  $28 \times 28$  images. Here is the architecture of **LeNet**.

**Image Source:** [https://en.wikipedia.org/wiki/LeNet#/media/File:Comparison\\_image\\_neural\\_network](https://en.wikipedia.org/wiki/LeNet#/media/File:Comparison_image_neural_network)

Here is the link to the implementation of **LeNet** part by part in **Keras**.

## AlexNet

**AlexNet** is another important **Convolutional Neural Network** that is being recognized as the beginning of the **Deep Learning**. It is officially designed to the classification task in a dataset called **ImageNet**. **ImageNet** is a huge dataset with 1000 classes. The images of this dataset is in **RGB** format and the shape of them are  $224 \times 224$ . Here is the architecture of **AlexNet**.

## LeNet

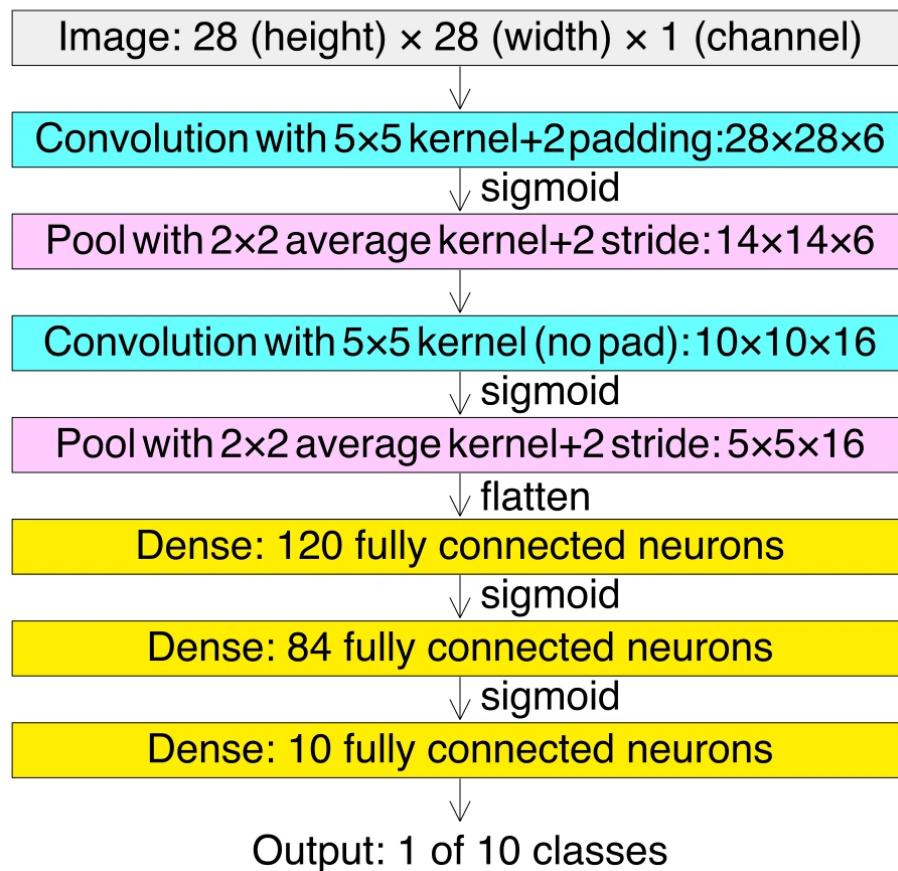


Figure 16: LeNet

## AlexNet

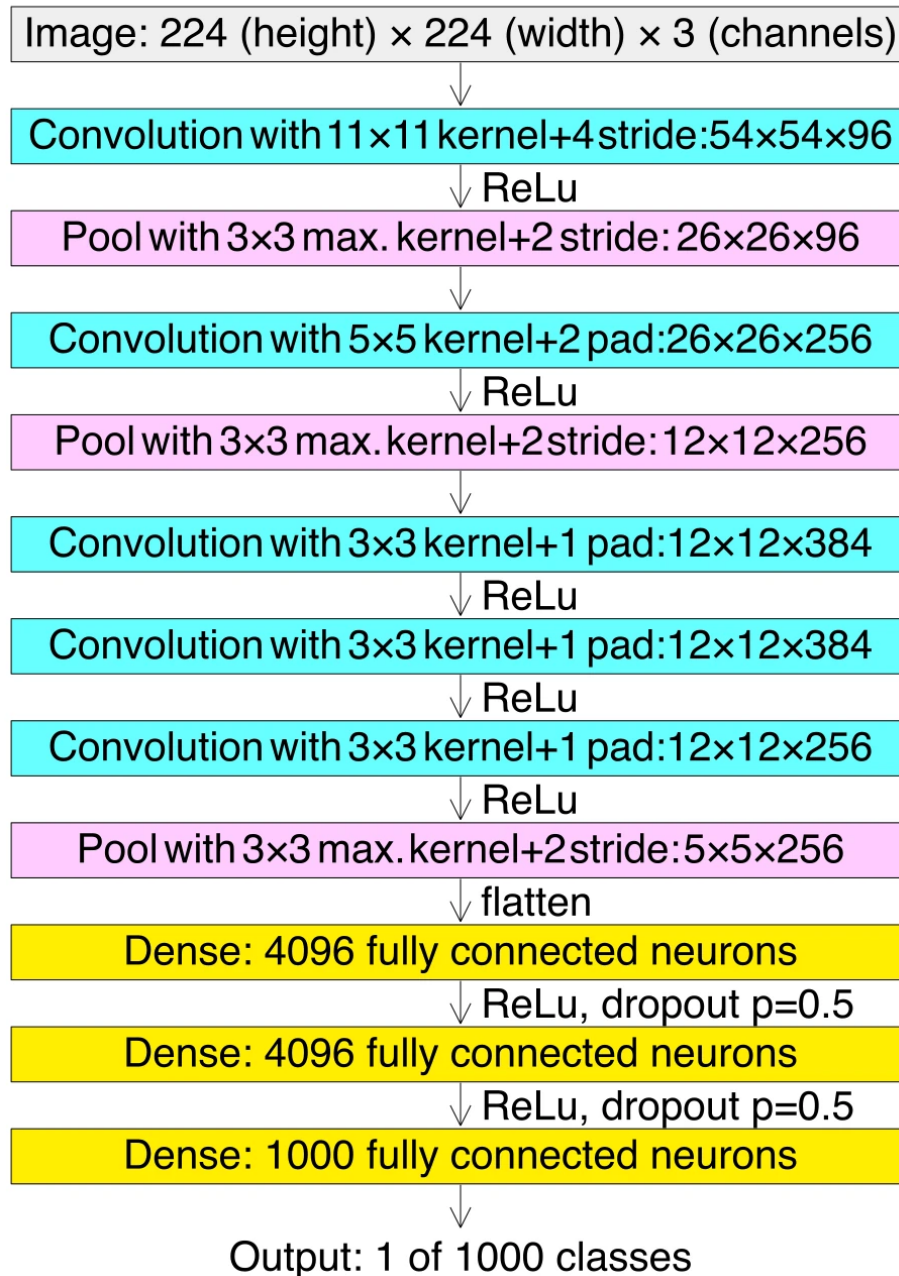


Figure 17: AlexNet

**Source:** [https://en.wikipedia.org/wiki/LeNet#/media/File:Comparison\\_image\\_neural\\_networks.svg](https://en.wikipedia.org/wiki/LeNet#/media/File:Comparison_image_neural_networks.svg)

Here is the link to the implementation of **LeNet** part by part in **Keras**.

## Your turn

Now, test the **AlexNet** on your dataset.

## Conclusion

In this tutorial, we learned the basic layers used in a **CNN**. First, we started with the **Convolution**. We explained how it works alongside with its 3 arguments: **stride**, **padding**, **dilation**. Second, we learned about **Pooling layers**. We discussed the reason behind it, and we explained about **Average Pooling** and **Max Pooling**. Third, we learned about **Activation functions**. We used to work with **ReLU** and **SoftMax**. Now, we are introduced to two other **Activation Functions**, **Tanh** and **Sigmoid**. Forth, we explained about the **Dropout layer** and why it is important. Finally, we implemented two important **CNNs**, **LeNet** and **AlexNet**, from scratch.

# Preprocessing and Augmentation

## Introduction

In the previous tutorial, we have learned about the basic layers used in **CNNs**. In this tutorial, we are going to learn about **preprocessing** and **augmentation** layers in **Keras**.

Preprocessing layers in Keras

## Preprocessing

**Data preprocessing** is a set of steps that we take before feeding the data to our model. These steps help us to have clean, consistent, and meaningful inputs. Also, they help the model to have a better accuracy, convergence speed, and generalization. When we were loading our dataset, we used two transformations: **Resize** and **ToTensor**. These two functions were related to the **PyTorch** and we were using them on the **ImageFolder**. Now, we are going to learn about some preprocessing layers in **Keras**.

## Resizing

**Resizing** layer is a layer that resizes its input to match the given size. Here is an example that we resized an image with the size of  $1920 \times 1080$  to  $224 \times 224$ .

```
from keras.layers import Resizing
```

```

resizing_layer = Resizing(224, 224)

input_image = np.random.randint(0, 256, (1, 1920, 1080, 3))

result_image = resizing_layer(input_image)

print(f"Input's shape: {input_image.shape}")
print(f"Result's shape: {result_image.shape}")

"""
-----
output:

Input's shape: (1, 1920, 1080, 3)
Result's shape: torch.Size([1, 224, 224, 3])
"""

```

## Rescaling

Rescaling layer is a layer that rescales its input to the given scale. In the example below, we have made a **Rescaling** layer with the scale of  $\frac{1}{255}$ .

```

from keras.layers import Rescaling

rescaling_layer = Rescaling(1 / 255)

input_image = np.random.randint(0, 256, (1, 224, 224, 3))

result_image = rescaling_layer(input_image)

print(f"Input's max: {input_image.max()}")
print(f"Input's min: {input_image.min()}")
print(f"Result's max: {result_image.max()}")
print(f"Result's min: {result_image.min()}")

"""
-----
output:

Input's max: 255
Input's min: 0
Result's max: 1.0
Result's min: 0.0
"""

```

## Specific model preprocessing

Each model has its own preprocessing procedure. In **Keras** we can load and use them. Here is an example of the preprocessing for MobileNetV2.

```
from keras.applications.mobilenet_v2 import preprocess_input

input_image = np.random.randint(0, 256, (1, 224, 224, 3))

result_image = preprocess_input(input_image)

print(f"Input's max: {input_image.max()}")
print(f"Input's min: {input_image.min()}")
print(f"Result's max: {result_image.max()}")
print(f"Result's min: {result_image.min()}")

"""
-----
output:

Input's max: 255
Input's min: 0
Result's max: 1.0
Result's min: -1.0
"""
```

As you can see, in the example above, the input is mapped to the range of  $[-1.0, 1.0]$ . This is the way **MobileNetV2** expects its input to be. If we want to use this preprocessing procedure in our model layers, we can use a layer called **Lambda**. **Lambda** takes a function, like `preprocess_input`, and turns it to a layer. Here is an example of how we can achieve that.

```
from keras.layers import Lambda

input_image = np.random.randint(0, 256, (1, 224, 224, 3))
input_image = np.array(input_image, dtype=float)

preprocessing_layer = Lambda(preprocess_input)

result_image = preprocessing_layer(input_image)

print(f"Input's max: {input_image.max()}")
print(f"Input's min: {input_image.min()}")
print(f"Result's max: {result_image.max()}")
print(f"Result's min: {result_image.min()}")
```

```

"""
-----
output:

Input's max: 255.0
Input's min: 0.0
Result's max: 1.0
Result's min: -1.0
"""

```

## Augmentation

**Data Augmentation** is a technique in machine learning that artificially expands our training dataset by applying different transformations. **Data Augmentation** is extremely useful when we don't have enough data or our data is not balanced. It helps us with the generalization and prevents the model from over-fitting. We have so many different **augmentation** techniques for different use-cases. Let's get to know how to use some of them in **Keras**. You can see the output of all the examples in this notebook

### RandomFlip

**RandomFlip**, technically, has a 50 chance to flip its input in the given mode. Modes can be:

- horizontal
- vertical
- horizontal\_and\_vertical

Here is an example that only flips horizontally:

```

from keras.layers import RandomFlip

random_flip_layer = RandomFlip("horizontal")

```

- The most common rotation is horizontal
- Use it when left and right rotation doesn't matter

### RandomRotation

**RandomRotation** rotates its input with the given factor. The range of the rotation would be:  $[-factor * \pi, +factor * \pi]$

For example, if we put the factor to 0.2, it would rotate the input in the range of

$$[-0.2 * \pi, +0.2 * \pi] = [-0.2 * 180^\circ, 0.2 * 180^\circ] = [-36^\circ, 36^\circ]$$

Here is an example of this layer:

```
from keras.layers import RandomRotation

random_rotation_layer = RandomRotation(0.2)
```

- Make your model robust to the rotation

## RandomZoom

RandomZoom zooms in or out respect to the **height\_factor** and **width\_factor**. Here is an example of this layer:

```
from keras.layers import RandomZoom

random_zoom_layer = RandomZoom(0.4, 0.2)
```

- Helps the model to handle scale changes
- Super effective in classification problems

## RandomTranslation

RandomZoom moves the image respect to the **height\_factor** and **width\_factor**. Here is an example of this layer:

```
from keras.layers import RandomTranslation

random_translation_layer = RandomTranslation(0.2, 0.2)
```

- Simulates small camera movements
- It is super important for the tasks that position of the object doesn't matter

## RandomContrast

RandomContrast changes the contrast respect to the given **factor**. Here is an example of this layer:

```
from keras.layers import RandomContrast

random_contrast_layer = RandomContrast(0.4)
```

- Helps us with the different lightning setups
- Useful in outdoor scenes and natural environments

## RandomBrightness

RandomBrightness changes the brightness respect to the given **factor**. Here is an example of this layer:

```
from keras.layers import RandomBrightness

random_brightness_layer = RandomBrightness(0.1)
```

- Helps us with the different lightning environments
- Specially data's taken in the different times of the day in the nature

## RandomCrop

RandomCrop crops to the given height and width randomly. Here is an example of this layer:

```
from keras.layers import RandomCrop

random_crop_layer = RandomCrop(224, 224)
```

- Simulates random object placements
- Extremely useful in large-scale training

## Add preprocessing and augmentation layers to our model

We should add our preprocessing and augmentation layers before feeding our data to the model. Here is an example:

```
"""
augmentation_layers = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomFlip("vertical"),
        layers.RandomZoom(0.1, 0.1),
        layers.RandomTranslation(0.05, 0.05),
        layers.RandomRotation(0.05),
    ]
)

model = keras.Sequential(
    [
        layers.Input(shape=(3, 224, 224)),
        layers.Permute((2, 3, 1)),
        layers.Rescaling(1.0 / 255),
        augmentation_layers,
        layers.Lambda(preprocess_input),
        base_model,
        layers.Flatten(),
        layers.Dense(4, activation="softmax"),
    ]
)
```

```

-----
output:

Model: "sequential_5"

   Layer (type)                Output Shape
   ────────────
   permute_2 (Permute)         (None, 224, 224, 3)
   ────────────
   ↪ 0

   rescaling (Rescaling)       (None, 224, 224, 3)
   ────────────
   ↪ 0

   sequential_4 (Sequential)    (None, 224, 224, 3)
   ────────────
   ↪ 0

   lambda (Lambda)             (None, 224, 224, 3)
   ────────────
   ↪ 0

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

   flatten_2 (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)

"""

```

In the example above, we have defined a `Sequential` to add our augmentation layers. Our augmentation layers consists of filliping, zooming, translation, and rotation. We also added the preprocess unit and rescaling.

We should always consider not over stack these layers. In this example, we only wanted to show you how we can add multiple augmentation layers. It might be too much for our model, which right now doesn't have so many parameters to

learn.

## Your turn

Now, choose the correct preprocessing and augmentation for your model and dataset and see the outputs.

## Conclusion

In this tutorial, we have learned about preprocessing and augmentation. First, we explained about preprocessing and how to use them in **Keras**. Then, we explored three different preprocessors. After that, we explained about the data augmentations and their use-cases. We introduced some of the most important augmentation layers. Finally, we learned how to add these layers in our model.

## Callbacks

### Introduction

In the previous tutorial, we have learned about **preprocessing and data augmentation** techniques in Keras. In this tutorial, we learn about **Callbacks** and explore some of the most important ones in **Keras**.

### Callbacks

**Callback** in **Keras** is a function that we pass it to our **fit** function. **Keras** calls that function automatically in the specific moment. We have learned about **TensorBoard Callback** before. We used to create a **TensorBoard Callback** and pass it to our fit function as below:

```
from keras.callbacks import TensorBoard

log_dir = "logs/fit/" +
    ↪ datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = TensorBoard(log_dir=log_dir)

history = model.fit(
    ...,
    callbacks=[tensorboard_callback],
)
```

In this tutorial, we are going to learn about another two important **Callbacks**, called: **EarlyStopping** and **ModelCheckpoint**.

List of available callbacks in Keras

## Early Stopping

**EarlyStopping** is a callback that stops the training procedure if there is no improvement. Here is an example of **EarlyStopping**:

```
from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    monitor="val_loss",
    patience=5,
    restore_best_weights=True,
    verbose=1,
)

history = model.fit(
    ...,
    epochs=200,
    callbacks=[tensorboard_callback, early_stopping],
)
```

In the code above, we have created an object of **EarlyStopping**. We set our **EarlyStopping** to monitor our validation loss ("val\_loss"). Then, we told it to wait for 5 epochs, if there was no improvement seen on those epochs, stop the training. With **restore\_best\_weights** set to **True**, **Keras** will load the best weights of the model that has the lowest **val\_loss**. Also, we set the **verbose** to 1, to be able to have a report of the procedure. As you can see, we have added it to our **callbacks** argument in **fit** function and increased our **epochs** to 200.

## Model Checkpoint

**ModelCheckpoint** saves the model during the training. Here is an example of **ModelCheckpoint**.

```
from keras.callbacks import ModelCheckpoint

model_checkpoint = ModelCheckpoint(
    filepath="checkpoints/best_model.weights.h5",
    monitor="val_loss",
    save_best_only=True,
    save_weights_only=True,
    verbose=1,
)

history = model.fit(
    ...,
    callbacks=[tensorboard_callback, early_stopping,
    ↪ model_checkpoint],
)
```

In the example above, we created an object of `ModelCheckpoint`. At first, we defined the path of the file, that we want to save our model. By the `monitor` argument, we said our object to look out for validation loss (`val_loss`). Then, we said, we only want to save the best model by using `save_best_only=True`. This approach helps the model not to save the model after each epoch and only saves the best one. After that, we set `save_weight_only=True`. This argument helps us to save only the weights of our model, not the way that we have compiled or other characteristics. When we do that, the capacity of the saved model decreases, and we can load our model in multiple platforms, not the specific one that we have trained our model on it. If we set this argument, we should make sure that the name of our `filepath` ends with `.weights.h5`. Then, we set `verbose=1` to have a report of what is happening.

If we want to load the best weights that we have saved, we can use the code below:

```
model.load_weights("checkpoints/best_model.weights.h5")
```

## Your turn

Add `EarlyStopping` and `ModelCheckpoint` to your callbacks.

## Conclusion

In this tutorial, we learned about **Callbacks** in **Keras**. First, we explained about what callbacks are actually are. Then, we introduced two of the most important callbacks, `EarlyStopping` and `ModelCheckpoint`.

## Fine-tuning

### Introduction

In the previous tutorials, we learned about **transfer learning** and how to use the advantage of a pre-trained model on our dataset. In this Tutorial, we learn about how to apply fine-tuning and see the results. Also, we explain the concepts of **Underfitting** and **Overfitting** and how to solve them. Finally, we make more classification layer more generalized.

### Fine-tuning

**Fine-tuning** is a technique in **Deep Learning** that we use to adapt our pre-trained model with the new **Dataset**. In the previous tutorials, we worked with **transfer learning**. **Fine-tuning** is pretty similar to **transfer learning**. The only difference is that we unfreeze some of the last layers to our `base_model` in order to train them. Here is an example:

```
base_model = MobileNetV2(include_top=False, input_shape=(224,
↳ 224, 3))

for layer in base_model.layers[:-4]:
    layer.trainable = False
```

In the code above, we froze the starting layers of our `base_model` and left the last 4 layers as `trainable`. Now, let's print the `base_model` summary with `show_trainable=True` like below:

```
print(base_model.summary(show_trainable=True))

"""
-----
output:

...

/ block_16_project      (None, 7, 7,      307,200  block_16_dept...
↳      N
  (Conv2D)              320)
↳
/ block_16_project...  (None, 7, 7,      1,280  block_16_proj...
↳      Y
  (BatchNormalizat...  320)
↳
Conv_1 (Conv2D)        (None, 7, 7,      409,600  block_16_proj...
↳  Y
                          1280)
↳
Conv_1_bn              (None, 7, 7,      5,120  Conv_1[0][0]
↳  Y
  (BatchNormalizat...  1280)
↳
out_relu (ReLU)        (None, 7, 7,      0  Conv_1_bn[0][...
↳  -
                          1280)
↳

Total params: 2,257,984 (8.61 MB)
Trainable params: 412,800 (1.57 MB)
```

```
Non-trainable params: 1,845,184 (7.04 MB)
```

```
"""
```

As you can see, the last 4 layers, are trainable. The only thing that we should do, is to train our model like before.

## Underfitting

**Underfitting** happens when our model is not training well on our **training data**. In other words, our model is not capable of learning the pattern of our data. There are different reasons that might cause this phenomenon to happen. One of the most important ones is that our model is too simple for the problem that we have. To solve this problem, we should choose a more complex model with more trainable layers.

Another reason behind **Underfitting**, is that we used too much **Regularization**. For example, we have used so many **Augmentation** layers. To solve it, we should just choose the suited **Regularization** techniques.

Sometimes, we haven't chosen the correct input features required for understanding the pattern. For example, if we want to estimate the house price, and we don't have the size of the house, our model is not going to figure out the pattern.

## Overfitting

**Overfitting** happens when our model is doing well on training data but the results on unseen data (**Validation** and **Test**) are not good. In other words, model has understood the pattern so well (including the noise), but it fails to generalize. There are some reasons that might be the cause of **Overfitting**. One of the most important ones is that our model is too complex for the dataset that we have. To fix this problem, we should choose a simpler model or lower the number of trainable layers.

Another reason is that, we train our model for a long time. To solve this problem, we said that we can use **EarlyStopping**.

One of the other reasons is that, we don't use enough regularization. For example, if our data is the images taken on the nature with so many different contrast and brightness, it is expected that our unseen data is also has this differences. So, if we want our model to learn how to deal with them, we should use the respective **Augmentation**.

## Make our classification layer more generalized

In the previous tutorials, we only used a **Fully Connected** layer for our **Classification layer**. Now, let's change the **Classification layer** based on the

things that we learned. Here is an example:

```
model = keras.Sequential(  
    [  
        ...,  
        layers.GlobalAveragePooling2D(),  
        layers.Dropout(0.5),  
        layers.Dense(128, activation="relu"),  
        layers.Dropout(0.5),  
        layers.Dense(4, activation="softmax"),  
    ]  
)
```

As you can see, in the code above, at first, I used a `GlobalAveragePooling2D` instead of `Flatten`. It helps the model to be more generalized. Then I used a `dropout` layer. This layer helps to have more generalization. Instead of using only one layer, I have used 2 layers. The first layer has 128 neurons with the activation of `relu` and the other layer has 4 neurons with the activation of `softmax` in order to guess the classification.

## Your turn

Now, Change the **transfer learning** to **fine-tuning** and make your classification layer more generalized.

## Conclusion

In this Tutorial, at first, we explained how we can apply **fine-tuning**. Then, we described **Underfitting** and **Overfitting** and how to solve them. Finally, we made our classification layer more generalized.

# Generative AI

## Introduction

## Your turn

## Conclusion