Modul Pelatihan
TensorFlow Developer Certificate

Disusun oleh:
Henokh Lugo Hariyanto

Version 0.40 (August 2021)

Table of Contents

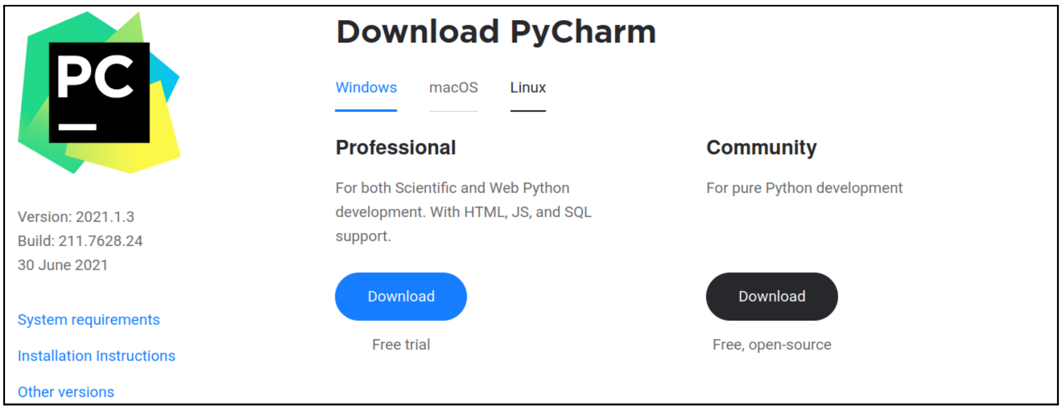
1. Python introduction with PyCharm

a. PyCharm installation

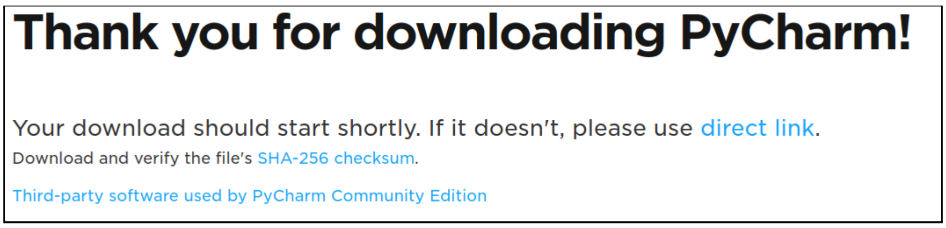
This installation will also include all Python 3.8+ and standard packages. In this tutorial, we choose standalone installation. For recommended installation using JetBrains Toolbox App, see the complete description in https://www.jetbrains.com/help/pycharm/installation-guide.html.
Please note that the PyCharm version that you should install is build 211.* or older to support TensorFlow Developer Certificate plugin version 0.1.12.

i. Installation on Windows

Download the installer .exe in https://www.jetbrains.com/pycharm/download/#section=windows. Choose the Community version. The file size is around 366 MB.



Then the next window will appear. Click “Download and verify the file’s SHA-255 checksum”. Keep the number to compare with the next procedure



Verify your downloaded file of pycharm by running in command prompt

```
$ Certutil -hashfile <path to file> SHA256
```

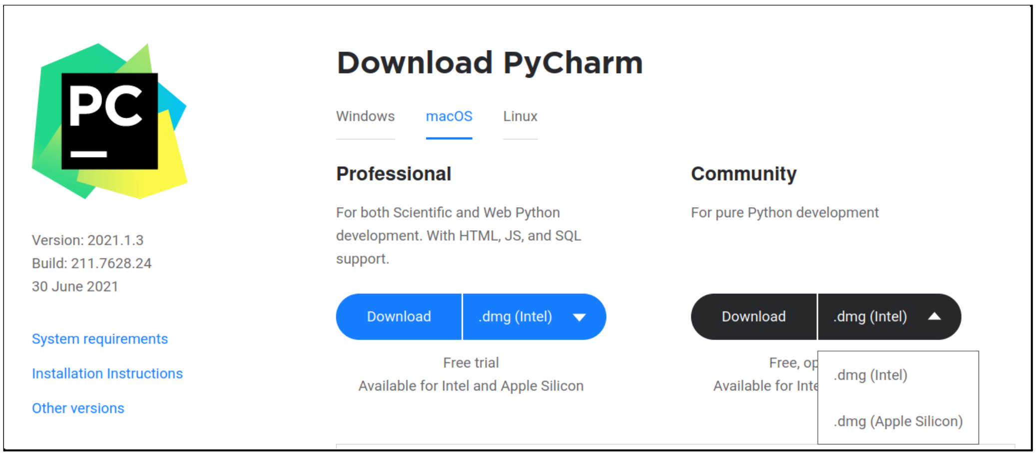
After you have verified the installer, you can proceed to run the installer. Mind the following options in the installation wizard

- **64-bit launcher:** Adds a launching icon to the Desktop.
- **Open Folder as Project**: Adds an option to the folder context menu that will allow opening the selected directory as a PyCharm project.
- **.py**: Establishes an association with Python files to open them in PyCharm.

- **Add launchers dir to the PATH**: Allows running this PyCharm instance from the Console without specifying the path to it.

ii. Installation on macOs

Download the disk image in https://www.jetbrains.com/pycharm/download/#section=mac. Choose the Community version. The file size is around 450 MB.



Then the next window will appear. Click “Download and verify the file’s SHA-255 checksum”. Keep the number to compare with the next procedure

Verify your downloaded file of pycharm by running in command prompt

```
$ shasum -a 256 <path to file>
```

After you have verified the installer, you can mount the image and drag the **PyCharm** app to the **Applications** folder.

iii. Installation on Linux

We will use the installer through snap packages. Open the terminal and run the following command:

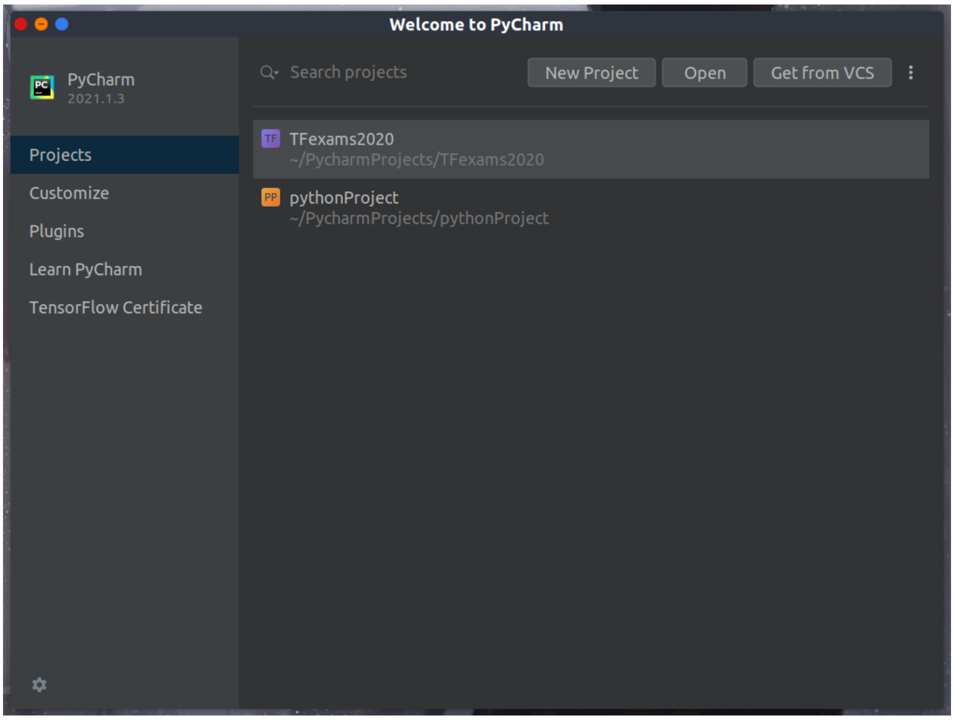
```
$ sudo snap install pycharm-community --classic.
```

For specific version, sometimes snap would have updated automatically, we can use the following command

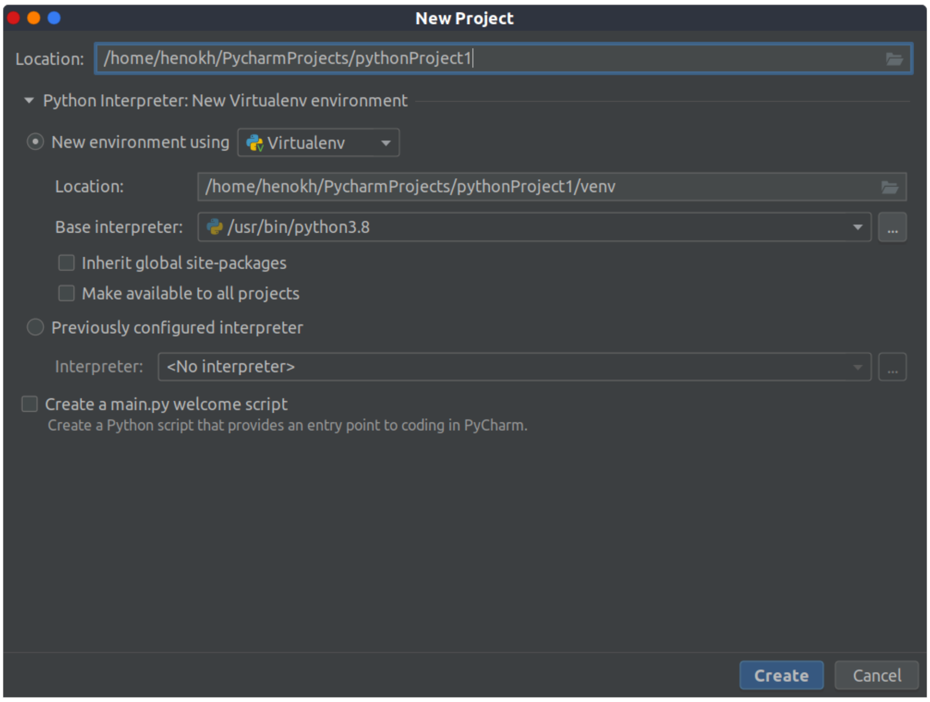
```
$ sudo snap install pycharm-community -channel=2021.1/stable --classic.
```

b. Fast Intro using PyCharm

When we open PyCharm for the first time, we will have a window like below



You can select "New Project" and then "New Project" window will appear

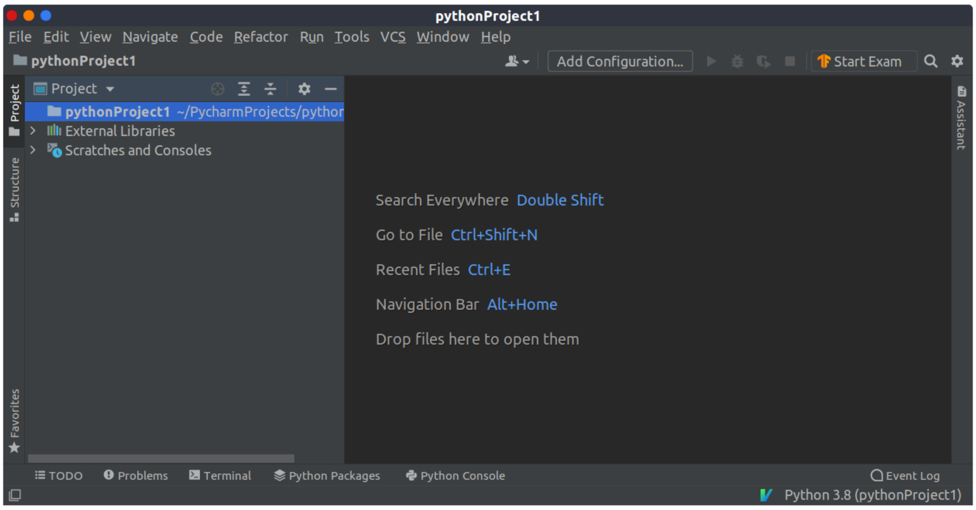


Several things that we need to set up

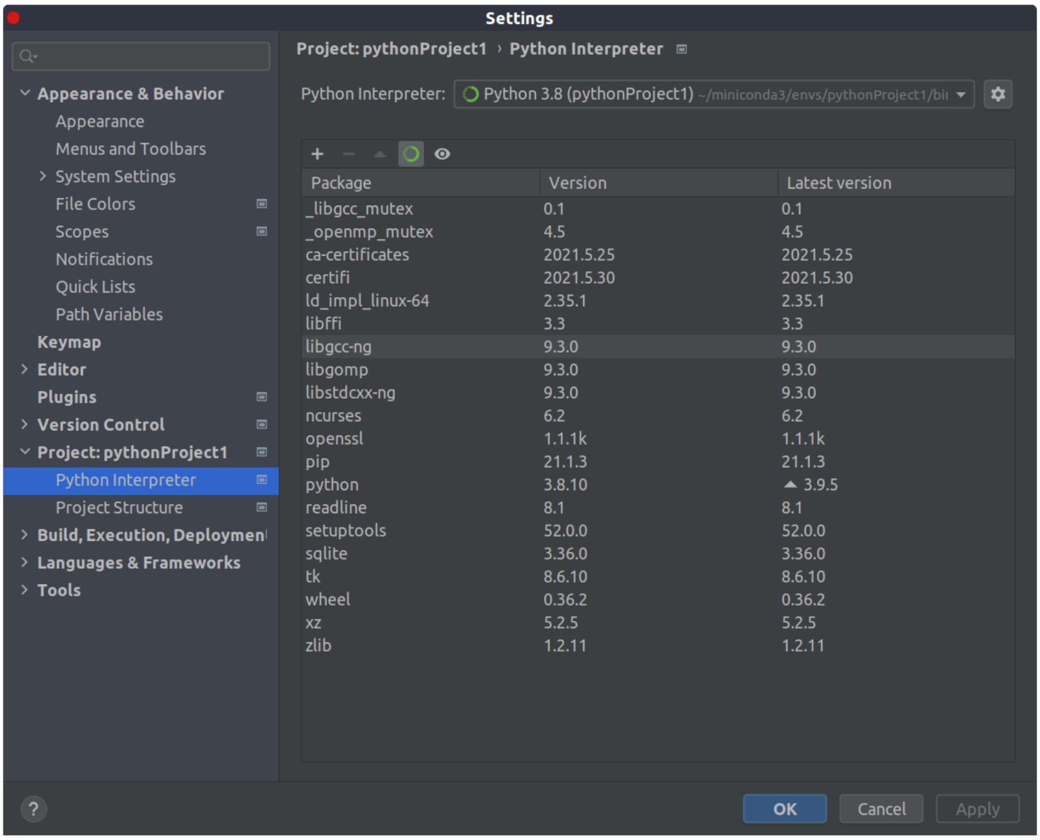
- Location: The location of folder for the project that we want to create
- New environment using: There are three available environments that we can use; Virtualenv, Pipenv, and Conda. My personal option is to use the Conda environment.

- Under the New environment section, we can also set the path for the environment in Location.
- Base interpreter: The path for your Python interpreter. For our purpose of TensorFlow Developer Certificate, we need to use Python 3.8

Click “Create” after finishing all the above things. Then the main window of PyCharm IDE will shown up.



Then go to File >> Settings….
On the left box of the “Setting window, expand Project:<your_project_name> and select Python Interpreter. On the right box, click “+” to add a new Python’s packages.

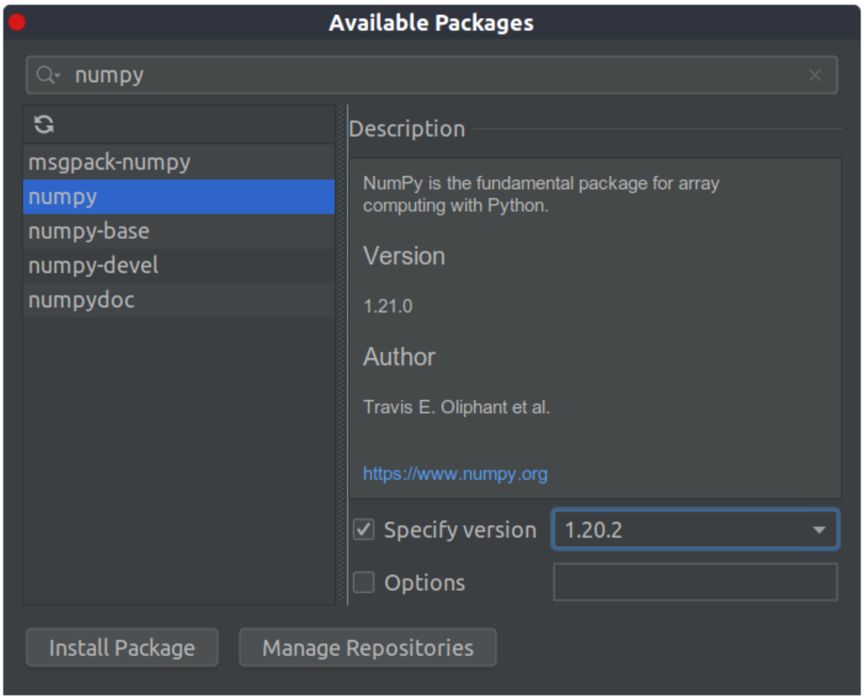


Then the Available Packages window will appear. In the search toolbars search the following packages:

- tensorflow==2.5.0

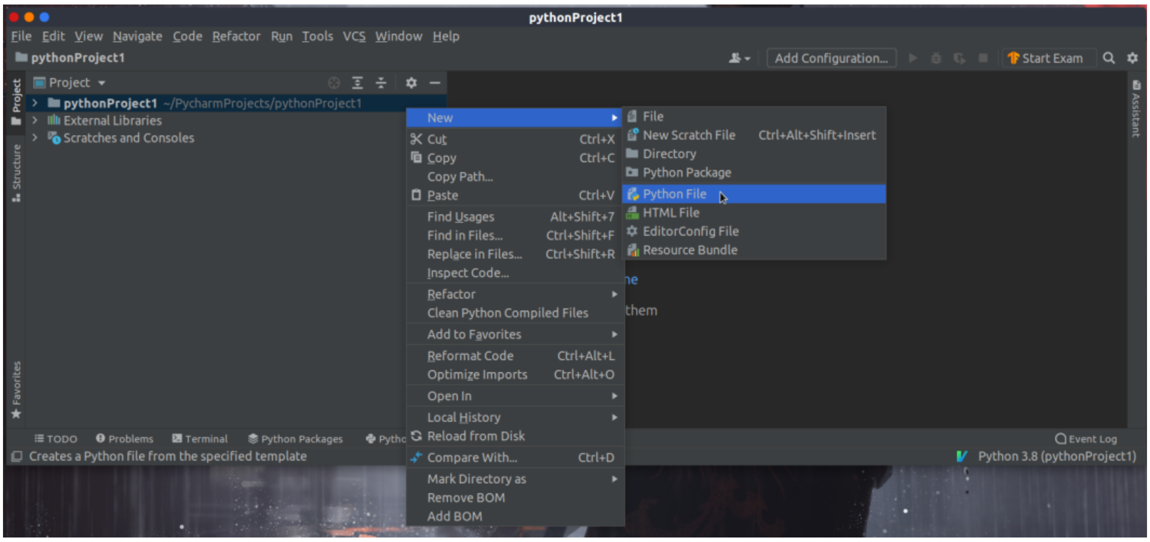
- tensorflow-datasets==4.3.0
- Pillow==8.2.0
- pandas==1.2.4
- numpy==1.19.5
- scipy=1.7.0
- Urllib3

The below image shows the “Available Packages” window for “numpy”. To select the package’s version, you can set “Specify version”



For full description see
https://www.tensorflow.org/extras/cert/Setting_Up_TF_Developer_Certificate_Exam.pdf

After we have installed all the required packages, we can create a new Python file under the directory of the new project that we have created by right clicking the project’s name and selecting “New” >> “Python File”. Put a meaningful name to that Python file that represents the content of the code.



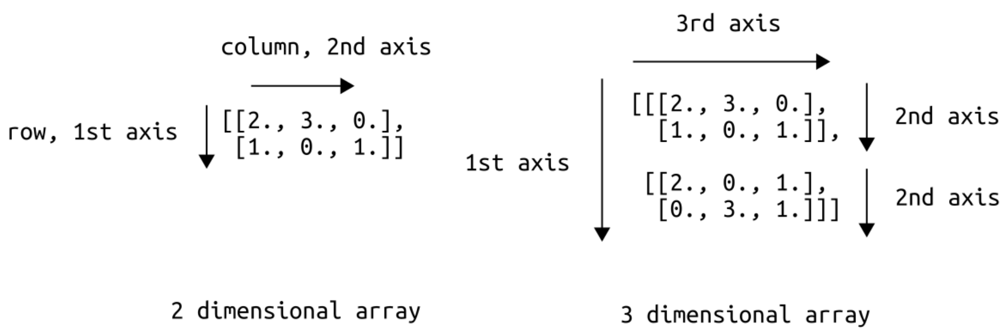
Then, we are ready to start for the next step.

c. Fast Introduction to NumPy and Matplotlib

This is a short introduction on how to use Python with NumPy and Matplotlib. For complete description go to official documentation of Python, NumPy, and Matplotlib.

i. Basics of NumPy

NumPy is one of the most useful Python's packages to handle arrays in any dimension and is full of methods to manipulate them. The image below shows the fundamental term that we have to understand when we use NumPy



In NumPy, the nth-dimension is called *nth-axis*. Numpy's class in Python is called ndarray. Several important attributes in ndarray are

- **`ndarray.ndim`**: store the number of axes
- **`ndarray.shape`**: store size of the array in each dimension
- **`ndarray.size`**: store total number of elements of the array
- **`ndarray.dtype`**: store the data type of the elements.

An example

```
In [1]: import numpy as np

In [2]: a = np.arange(15).reshape(3, 5)
```

```
In [3]: a
Out[3]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [4]: a.shape
Out[4]: (3, 5)

In [5]: a.ndim
Out[5]: 2

In [6]: a.dtype
Out[6]: dtype('int64')

In [7]: a.size
Out[7]: 15

In [8]: type(a)
Out[8]: numpy.ndarray

In [9]: b = np.array([6, 7, 8])

In [10]: b
Out[10]: array([6, 7, 8])

In [11]: type(b)
Out[11]: numpy.ndarray
```

Array creation

There are several ways to create arrays:

- **`np.array()`**: create an array by providing it with a list or tuple.
- **`np.zeros()`**, **`np.ones()`**, or **`np.empty()`**: Initialize an array when we only know the size of the array before computation
- **`np.arange()`**: create a one-dimensional array if we know the lower and upper values and also the increments.
- **`np.linspace()`**: the most useful function to create *N* elements of a one-dimensional array if we know the lower and upper values of the array.

```
In [1]: import numpy as np

In [4]: b = np.array([1.2, 4.3, 9.1, 10.])

In [5]: b.dtype
Out[5]: dtype('float64')

In [6]: c = np.array([[1, 4], [9, 6]], dtype=np.float64)

In [7]: c
Out[7]:
array([[1., 4.],
       [9., 6.]])
```

```
In [8]: np.zeros((3, 5))
Out[8]:
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])

In [9]: np.ones((2, 2, 5), dtype=np.int64)
Out[9]:
array([[[1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1]],

       [[1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1]]])

In [10]: np.empty((2, 4))
Out[10]:
array([[4.66086321e-310, 0.00000000e+000, 4.66086282e-310,
        3.80323684e+177],
       [6.93809786e-310, 4.66086282e-310, 2.03508930e+281,
        6.93810211e-310]])

In [11]: np.arange(2, 9, 2)
Out[11]: array([2, 4, 6, 8])

In [12]: np.linspace(0, 5, 11)
Out[12]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

Reshaping array

When we train a machine learning model with different sizes of input, we have to make sure that the shape of the input array matches the shape of the input array of the model. NumPy provides us with a handy method to reshape it.
We can invoke the method by **`.reshape(axis1, axis2, ...)`** or **`np.reshape(arr, axis1, axis2, ...)`**

```
In [1]: import numpy as np

# generate an integer array with 10 elements start from 0
In [2]: arr1D = np.arange(10)

In [3]: print(arr1D)
[0 1 2 3 4 5 6 7 8 9]

In [4]: arr2D = np.arange(20).reshape(2, 10)

In [5]: print(arr2D)
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]

In [6]: arr3D = np.arange(40).reshape(2, 2, 10)

In [7]: print(arr3D)
```

```
[[[ 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 36 37 38 39]]]

# axis2 inferred automatically from axis1
In [8]: arr2D = np.reshape(arr1D, (2, -1))

In [9]: print(arr2D)
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

Basic operations

All arithmetic operation applied *element-wise* means that it operates for each element of the array.

```
In [23]: a = np.array([20, 30, 40, 50])

In [24]: b = np.arange(4)

In [25]: b
Out[25]: array([0, 1, 2, 3])

In [26]: c = a - b

In [27]: c
Out[27]: array([20, 29, 38, 47])

In [28]: b**2
Out[28]: array([0, 1, 4, 9])

In [29]: 10 * np.sin(a)
Out[29]: array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])

In [30]: a < 35
Out[30]: array([ True,  True, False, False])
```

Python multiplication operator "`*`" will be interpreted as the operation of element wise when we multiply two matrices. To do the usual matrix product, we can use the "`@`" operator or method "**`.dot()`**".

```
In [16]: A = np.array([[1, 1], [0, 1]])

In [18]: B = np.array([[2, 0], [3, 4]])

# element-wise product
In [20]: A * B
Out[20]:
array([[2, 0],
       [0, 4]])
```

```
# matrix product
In [21]: A @ B
Out[21]:
array([[5, 4],
       [3, 4]])

# matrix product using .dot() method
In [22]: A.dot(B)
Out[22]:
array([[5, 4],
       [3, 4]])
```

Indexing, slicing and iterating

Indexing in NumPy means we select a specific element in an array based on its index.
Slicing is a general way of indexing on how we collectively select more than one element with several indices. We can put complicated rules to slice an array that we will discuss in the later section.
Iterating is a short-hand notation to iterate over all element of an array when we use an array with **`for-loop`**

```
In [31]: a = np.arange(10, dtype=int)**3

In [32]: a
Out[32]: array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])

# get the element at index = 2
In [33]: a[2]
Out[33]: 8

# get the elements from index 2 to (5 - 1)
In [34]: a[2:5]
Out[34]: array([ 8, 27, 64])

# get the elements from index 0 to (6 - 1) with increment of 2
In [35]: a[:6:2]
Out[35]: array([ 0,  8, 64])

# a nice way to reverse an array
In [36]: a[::-1]
Out[36]: array([729, 512, 343, 216, 125,  64,  27,   8,   1,   0])

In [38]: for element in a:
    ...:     print(element / 2.0)
    ...:
0.0
0.5
4.0
13.5
32.0
62.5
108.0
```

```
171.5
256.0
364.5
```

ii. Shape manipulation of NumPy arrays

In the Basics section, we have introduced a simple way to manipulate the shape of an array by using **`.reshape()`**. The other useful NumPy functions to manipulate an array are:

- **`np.resize()`**: This function is similar to np.reshape() but it will change the array in-place instead of returning a modified array.
- **`np.vstack()`**: This will stack two arrays along the first axis (vertical)
- **`np.hstack()`**: This will stack two arrays along the second axis (horizontal)
- **`np.row_stack()`**: Similar to np.vstack()
- **`np.column_stack()`**: Similar to np.hstack() only for 2D arrays.
- **`np.newaxis`**: Create a new axis.
- **`np.vsplit()`**: Split 2D arrays along first axis (vertical)
- **`np.hsplit()`**: Split 2D arrays along second axis (horizontal)

```
In [39]: import numpy as np

# create a generator object
In [40]: rng = np.random.default_rng()

In [41]: a = np.floor(10 * rng.random((3, 4)))

In [42]: a
Out[42]:
array([[6., 1., 6., 3.],
       [4., 4., 0., 1.],
       [4., 5., 0., 9.]])

In [43]: a.shape
Out[43]: (3, 4)

In [44]: a.resize((2, 6))

In [45]: a
Out[45]:
array([[6., 1., 6., 3., 4., 4.],
       [0., 1., 4., 5., 0., 9.]])

In [46]: a = np.floor(10 * rng.random((2, 2)))

In [47]: a
Out[47]:
array([[9., 0.],
       [0., 2.]])

In [48]: b = np.floor(10 * rng.random((2, 2)))

In [49]: b
Out[49]:
array([[8., 5.],
```

```
       [7., 4.]])

In [50]: np.vstack((a, b))
Out[50]:
array([[9., 0.],
       [0., 2.],
       [8., 5.],
       [7., 4.]])

In [51]: np.hstack((a, b))
Out[51]:
array([[9., 0., 8., 5.],
       [0., 2., 7., 4.]])

In [52]: np.row_stack((a, b))
Out[52]:
array([[9., 0.],
       [0., 2.],
       [8., 5.],
       [7., 4.]])

In [53]: np.column_stack((a, b))
Out[53]:
array([[9., 0., 8., 5.],
       [0., 2., 7., 4.]])

In [54]: a = np.array([3., 5.])

In [55]: a
Out[55]: array([3., 5.])

In [57]: a.shape
Out[57]: (2,)

# create a new axis for 1D array
In [59]: a[:, np.newaxis].shape
Out[59]: (2, 1)

In [60]: a = np.floor(10 * rng.random((2, 12)))

In [61]: a
Out[61]:
array([[2., 5., 9., 4., 0., 4., 9., 6., 8., 8., 3., 7.],
       [7., 5., 9., 9., 1., 5., 4., 3., 8., 1., 1., 8.]])

In [63]: np.hsplit(a, 3)
Out[63]:
[array([[2., 5., 9., 4.],
        [7., 5., 9., 9.]]),
 array([[0., 4., 9., 6.],
        [1., 5., 4., 3.]]),
 array([[8., 8., 3., 7.],
        [8., 1., 1., 8.]])]

In [64]: np.hsplit(a, 3, 3)

# horizontal split before third column and after (3-1)-th column
In [65]: np.hsplit(a, (3, 3))
Out[65]:
```

```
[array([[2., 5., 9.],
        [7., 5., 9.]]),
 array([], shape=(2, 0), dtype=float64),
 array([[4., 0., 4., 9., 6., 8., 8., 3., 7.],
        [9., 1., 5., 4., 3., 8., 1., 1., 8.]])]

In [66]: a = a.reshape(-1, 2)

In [67]: a
Out[67]:
array([[2., 5.],
       [9., 4.],
       [0., 4.],
       [9., 6.],
       [8., 8.],
       [3., 7.],
       [7., 5.],
       [9., 9.],
       [1., 5.],
       [4., 3.],
       [8., 1.],
       [1., 8.]])

In [68]: np.vsplit(a, 3)
Out[68]:
[array([[2., 5.],
        [9., 4.],
        [0., 4.],
        [9., 6.]]),
 array([[8., 8.],
        [3., 7.],
        [7., 5.],
        [9., 9.]]),
 array([[1., 5.],
        [4., 3.],
        [8., 1.],
        [1., 8.]])]

# vertical split before third row and after (3-1)-th row
In [69]: np.vsplit(a, (3, 3))
Out[69]:
[array([[2., 5.],
        [9., 4.],
        [0., 4.]]),
 array([], shape=(0, 2), dtype=float64),
 array([[9., 6.],
        [8., 8.],
        [3., 7.],
        [7., 5.],
        [9., 9.],
        [1., 5.],
        [4., 3.],
        [8., 1.],
        [1., 8.]])]
```

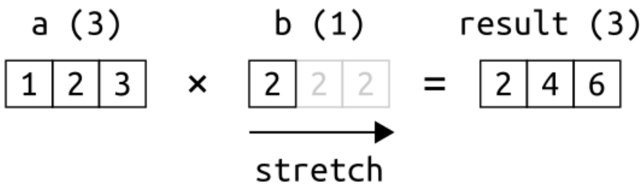
iii. Copies of NumPy arrays

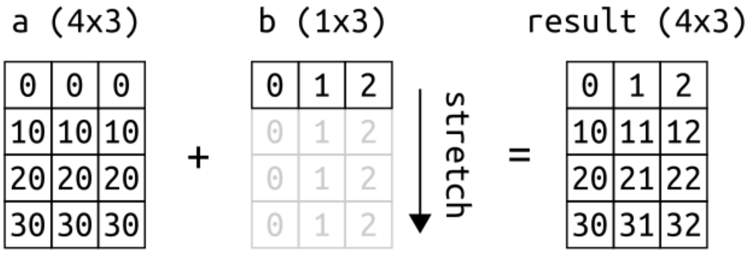
For someone without prior knowledge on how the assignment of NumPy array to variable is, should consider very carefully.
When we declare an array and assign it into another variable, we have to fully understand that what NumPy passed is not the values but its reference to the values.
To copy a NumPy array to another variable, we should use deep copy through method **`.copy()`**.
The following example will make the explanation above clear

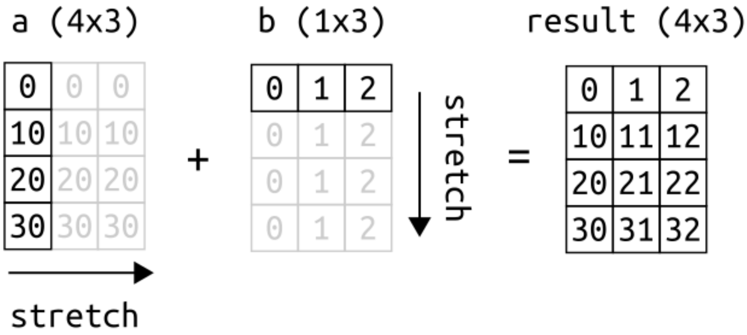
```
In [2]: import numpy as np

In [3]: a = np.array([[ 0,  1,  2,  3],
   ...:               [ 4,  5,  6,  7],
   ...:               [ 8,  9, 10, 11]])

In [4]: b = a

In [5]: b[0, 0] = 100

In [6]: a
Out[6]:
array([[100,   1,   2,   3],
       [  4,   5,   6,   7],
       [  8,   9,  10,  11]])

In [7]: a = np.array([[ 0,  1,  2,  3],
   ...:               [ 4,  5,  6,  7],
   ...:               [ 8,  9, 10, 11]])

In [8]: b = a.copy()

In [9]: b[0, 0] = 100

In [10]: a
Out[10]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

iv. Broadcasting rule of NumPy arrays operations

This is one of the powerful features in NumPy to eliminate for-loop and make the code easy to read and maintain.
Broadcasting rule is a process of stretching an array into a bigger one such that it will have the same shape to the other array under binary operation.
The following illustration describes that process.







The addition operator above can be replaced by any binary operator or even a function.

```
In [11]: a = np.array([1.0, 2.0, 3.0])

In [12]: b = 2.0

In [13]: a * b
Out[13]: array([2., 4., 6.])

# --------------------------------------------
In [14]: a = np.array([[ 0.0,  0.0,  0.0],
    ...:                [10.0, 10.0, 10.0],
    ...:                [20.0, 20.0, 20.0],
    ...:                [30.0, 30.0, 30.0]])

In [15]: b = np.array([1.0, 2.0, 3.0])

In [16]: a + b
Out[16]:
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])

# --------------------------------------------
In [17]: a = np.array([0.0, 10.0, 20.0, 30.0])

In [18]: b = np.array([1.0, 2.0, 3.0])
```

```
In [19]: a[:, np.newaxis] + b
Out[19]:
array([[ 1.,  2.,  3.],
       [11., 12., 13.],
       [21., 22., 23.],
       [31., 32., 33.]])
```

v. Advanced Indexing of NumPy arrays

In the Basics of NumPy, we have introduced how to do indexing to NumPy's array. In this section, we will understand how the concepts of indexing worked in 2D arrays and using some boolean indexing

Indexing with Arrays of Indices

If we have a 1D array, we can index the elements by 1D array or 2D array. The result will follow the shape of the indexing array.

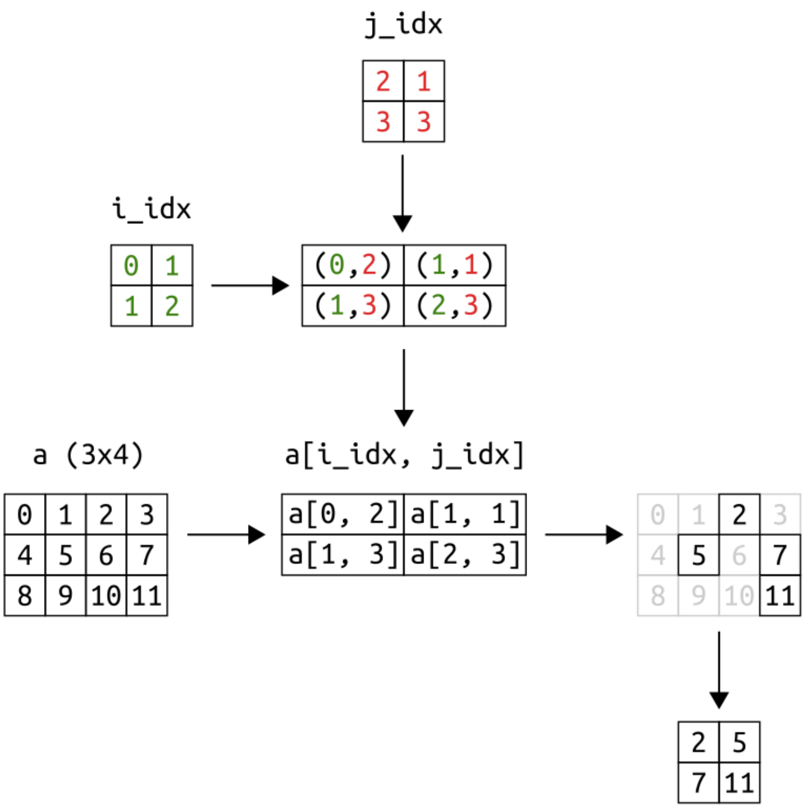
```
# the first 12 square numbers
In [20]: a = np.arange(12)**2

In [21]: a
Out[21]: array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121])

# a 1D array of indices
In [22]: i_idx = np.array([1, 1, 3, 8, 6])

# get the elements of `a` at the position `i_idx`
In [23]: a[i_idx]
Out[23]: array([ 1,  1,  9, 64, 36])

# a 2D array of indices
In [24]: j_idx = np.array([[3, 4], [9, 7]])

# get the elements of `a` at the position `j_idx`
# each index in `j_idx` will be interpreted as a single index
In [25]: a[j_idx]
Out[25]:
array([[ 9, 16],
       [81, 49]])
```

The following example will show us when working with 2D arrays and doing indexing using 1D or 2D array of indices.

```
a = np.arange(12).reshape(3, 4)

In [27]: a
Out[27]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
```

```
       [ 8,  9, 10, 11]])

# 2D array of indices for the first axis of `a`
In [28]: i_idx = np.array([[0, 1],
    ...:                   [1, 2]])

# 2D array of indices for the second axis of `a`
In [29]: j_idx = np.array([[2, 1],
    ...:                   [3, 3]])

# i_idx and j_idx must have equal shape
In [31]: a[i_idx, j_idx]
Out[31]:
array([[ 2,  5],
       [ 7, 11]])
```

The above example can be explained through this simple diagram



The 2D array of indices will be created by taking for each element from `i_idx`, `j_idx` in the same position of index. Then indexing using two 2D arrays of indices will return the same size of the 2D array of indices.

Indexing with Boolean Arrays

This is a method to select elements of an array based on whether those elements satisfy some condition or not.

```
In [32]: a = np.arange(12).reshape(3, 4)
```

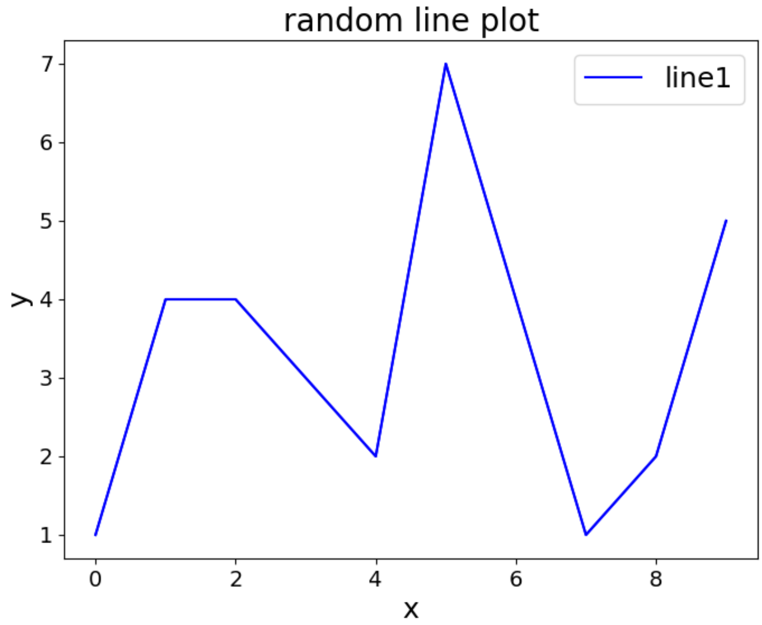
```
In [33]: b = a > 4

In [34]: b
Out[34]:
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])

# only select `True` indices
In [35]: a[b]
Out[35]: array([ 5,  6,  7,  8,  9, 10, 11])

# all elements of `a` higher than 4 become 0
In [36]: a[b] = 0

In [37]: a
Out[37]:
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

vi. A formal way to use Matplotlib

Matplotlib is a Python package to create a visualization in 2D (mostly). There is a standard way to code a plotting program using Matplotlib. This is not my style, but it is coming from the tutorial in the Matplotlib documentation.

```
import matplotlib.pyplot as plt
import numpy as np

# create data
rng = np.random.default_rng()
xdata = np.arange(10)
ydata = np.floor(10 * rng.random(10))

# define figure and axes objects
fig, ax = plt.subplots(figsize=(8, 6))

# plot the data to specific visualization
# e.g.: .plot(), .scatter, .imshow()
ax.plot(xdata, ydata, linestyle='-', color='blue', label="line1")

# setting the axes attributes
ax.set_xlabel('x')
ax.set_xlabel('y')
ax.legend(loc="best")
ax.set_title('random line plot')

# show the figure
plt.show()
```

The above script will produce the following figure


random line plot
line1
y
x
1
2
3
4
5
6
7
0
2
4
6
8

2. Introduction to TensorFlow

In Coursera courses, the course intends us to use Jupyter Notebook, but now, we try to be familiar as soon as possible with PyCharm IDE and how to use it. Then, all the codes below will be typed in Python script inside PyCharm IDE.

First, we start to write a “Hello World!” version of TensorFlow. This program will do a regression linear with simple data: $y = 2x - 1$.

Listing 2.1. simple_regression.py

```
1  import tensorflow as tf
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  from tensorflow import keras
6
7
8  def plot_data(x_data, y_data):
9      fig, ax = plt.subplots()
10
11     ax.plot(x_data, y_data, 'ro')
12
13     plt.pause(1)
14
15     return None
16
17
18 if __name__ == "__main__":
19     # Define and compile the neural networks
20     model = tf.keras.Sequential(
21         [keras.layers.Dense(units=1, input_shape=[1])])
22     model.compile(optimizer="sgd", loss="mean_squared_error")
23
24     # Providing the data
25     xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])
26     ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0])  # y = 2x - 1
27
28     # Plotting data will help us to understand faster
29     # visually about the data
30     plot_data(xs, ys)
31
32     # Train the neural network
33     model.fit(xs, ys, epochs=500)
34
35     # Predict using the model
36     print(model.predict(([10.0])))   # this result should be close to 19.
```

Lines 1-3 imports necessary packages: 1) **`tensorflow`** provides a machine learning model, optimizer, metric and fitting subroutine; 2) **`numpy`** provides an efficient way to handle an array; and 3) **`matplotlib`** for plotting the data.

In line 5, we called a specific model’s layer (**`.Dense`** layer) through **`keras`** library.

Lines 8-15 is a function declaration for a simple plotting procedure.

Line 18-36 is the main section. In the later scripts, we will move several statements into functions above this main section. Now, we stick in this format where we put all the computational procedures under the line **`if __name__ == "__main__":`**.

In lines 20-22, we initialize the model using a regular densely-connected neural network layer. This is implemented by invoking **`keras.layers.Dense()`**. We also use the format that all layers will be contained by **`tf.keras.layers.Sequential()`**. By using this container we can include as many layers as we want sequentially. The keyword argument **`units=1`** in **`.Dense()`** layer specified the dimensionality of the output. The output that we want is to predict a single number for a given single number. Then the input should have dimension or shape 1, and this is specified by keyword argument **`input_shape=[1]`**. Please note that input shape should be defined by a list even though there is only one dimension.

Lines 25-26 defines the data sets for x and $y$using NumPy's arrays as column vectors. Line 30 is visualizing the data sets.

Line 33 uses method **`.fit()`**, to fit the model into data sets. We also set the epochs. This keyword argument indicates how many iterations the model trains to the data sets **`xs`** and **`ys`**.

In the last line, we try to predict y for the input **`x = 10`**. The program returns the predicted value approximately closed to the exact value.

The theoretical background for the above program can be understood as a single fully connected layer. What does it mean by the fully connected layer? In line 21, we wrote **`keras.layer.Dense()`**. This command will create for us a linear function

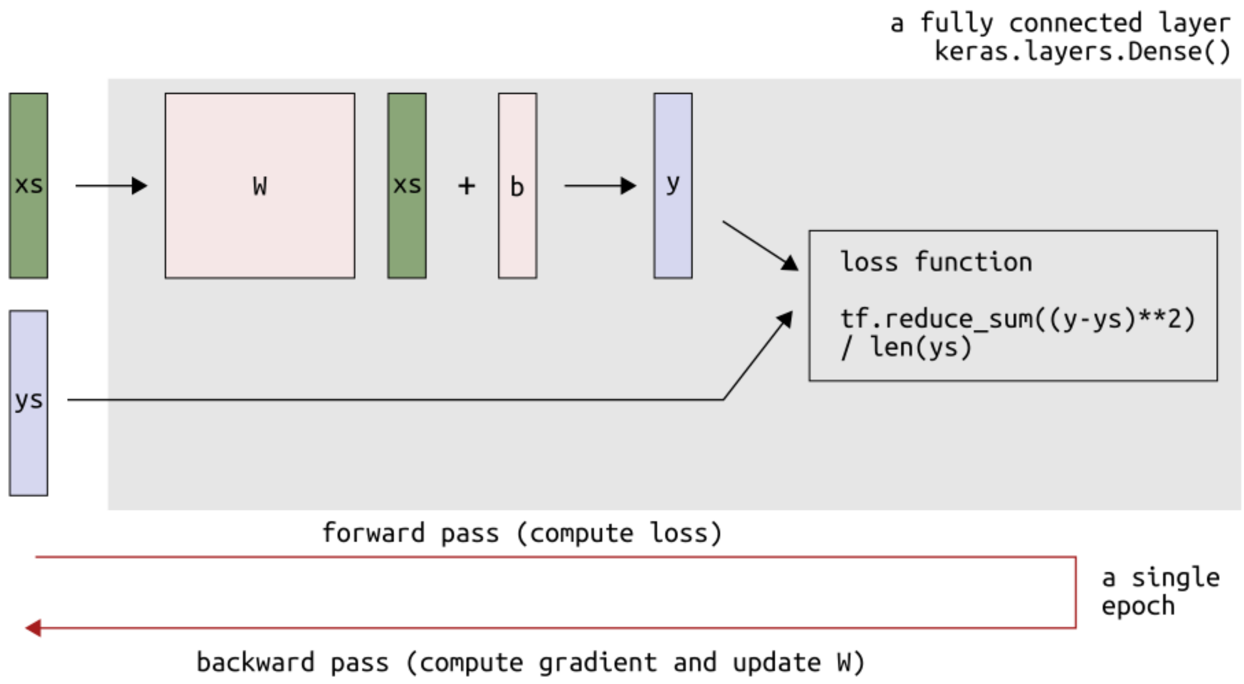
$$y_{pred} := Wx + b,$$

where $W$is a weight matrix with dimension 1x6. The number of columns represents the length of row vector x (in the script this corresponds to the variable **`xs`**) and b is a biased term. The term "fully connected" arises from the fact that we multiply all ("connected" and "fully") element of the input x with all the elements of the weight matrix W. Then this simple model of machine learning will try to find the best value of W and b such that it will give the minimum loss. The loss function (or objective function if you a hardcore optimization person) in the above script is defined as a mean square error:

$$MSE(y, y_{pred}) := \frac{1}{N}\sum_{i}^{N} (y - y_{pred})^2,$$

where y is the provided data which x corresponds to (in the script this corresponds to the variable **`ys`**). The optimization procedure is performed by using a stochastic gradient descent (in the script we set keyword argument **`optimizer='sgd'`** to the method **`model.compile()`**). A short description of stochastic gradient descent, this optimizer will compute gradients in a batch of data using backpropagation (a trick to compute gradients using computational graph and chain rules). Then will update the initial value of the weight matrix $W$by the amount of learning rate times negative of the computed gradient. The detail of the mechanism of this optimizer, including backpropagation, is beyond the scope of this module.

The previous description can be simplified by the following figure.

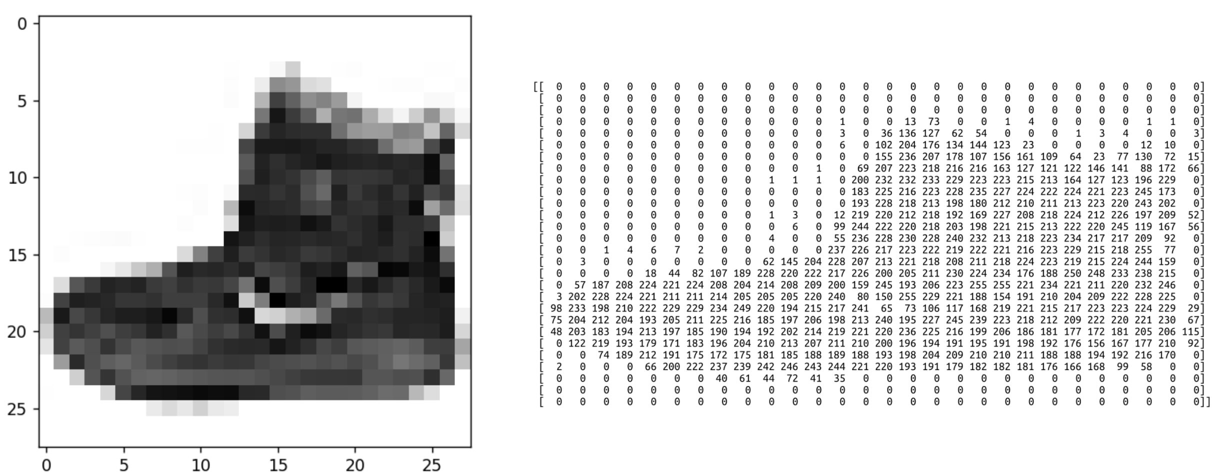


a. Classifying Fashion MNIST

Now, we want to build a model to predict what category is for a given image. For this purpose, we use the Fashion MNIST data. This data has the following specification:

- 70,000 images
- 10 categories. All categories are saved as numerical labels starting from 0 to 9.
- Images size are 28 x 28px

The following image is one of the sample images (the first entry) from Fashion MNIST whose category is “ankle boot” but it is saved as a numerical integer label 9.



All the Fashion MNIST data can be accessed through the TensorFlow module.

In Listing 2.2, we will use two dense layers to get a weight matrix as a model of the classifier that we want to build.

Listing 2.2. fashion_classifier.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import matplotlib
4  import tensorflow as tf
5
6
7  if __name__ == "__main__":
8
9      # Load the dataset
10     mnist = tf.keras.datasets.fashion_mnist
11     (training_images, training_labels), (test_images, test_labels) =
12         mnist.load_data()
13
14     # Plot an image from the dataset
15     plt.imshow(training_images[0], cmap="Greys")
16     plt.pause(1)        # increase the pause time to give time for plotting
17
18     print(training_labels[0])
19     print(training_images[0])
20
21     # Normalizing image intensities to [0, 1]
22     training_images = training_images / 255.0
23     test_images = test_images / 255.0
24
25     # Create a model
26     model = tf.keras.models.Sequential([
27         tf.keras.layers.Flatten(),
28         tf.keras.layers.Dense(128, activation=tf.nn.relu),
29         tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
30
31     # Compile the model and train it to the dataset
32     model.compile(optimizer=tf.optimizers.Adam(),
33                   loss="sparse_categorical_crossentropy",
34                   metrics=["accuracy"])
35
36     model.fit(training_images, training_labels, epochs=5)
37
38     # Evaluate the model to the test data
39     model.evaluate(test_images, test_labels)
```

Lines 1-4 imports required packages. We load the Fashion MNIST data in lines 10-12 and separate between train and test data. From the TensorFlow API this will separate 70,000 images into 60,000 images for train data and 10,000 for test data. Then in lines 14-19, we try to see whether we load the correct data or not, by picking a single image then plotting it and printing the content.

We also performed data preprocessing by scaling the grayscale value of the image into a range [0, 1] in lines 22-23. This will help the optimizer to only handle not too big numbers. Then, we create a model with two fully connected layers in lines 25-29. First we flatten the shape of the image from a matrix of 28x28 into a long row vector 28x28=784 entries. The first fully connected layer categorizes these 784 entries into 128 hidden variables and activates using ReLU (Rectifier Linear Unit) activation function to induce nonlinearity. Without this activation function, this two-layer model would be redundant and similar to a one-layer model. In general ReLu is defined as

$$ReLu\left(h_i\right) := max(0, h_i),$$

where h will be the a row vector result of linear classifier $h := Wx + b$. This output from ReLU will be the input of the second fully connected layer where instead of using ReLU as the activation, this layer uses softmax activation function. The softmax activation function is defined as

$$softmax(h_i) := \frac{exp(h_i)}{\sum_{j=1}^{C} exp(h_j)},$$

where C is the number of classes/categories.

This softmax activation function is similar like computing the probability of each output h_i but with adding a twist of exponential function to scaling-up small value in each output h_i.

In lines 31-34, we compile all the model's layers and add optimizer, loss function, and metric. This Adam optimizer is a fancier optimizer compared to stochastic gradient descent where we do not update directly negative gradients to the current weight matrix, but with some complicated pre-steps. In general, this optimizer incorporates AdaGrad and RMSProp optimizers. For a complete description of Adam optimizer, see the paper of (Kingma and Ba, 2015). Next, we use lost function **`sparse_categorical_entropy`**. This lost function will compute the following value

$$CatEntropy(s) := -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{C} t_k^{(i)}\, log(s_k^{(i)}),$$

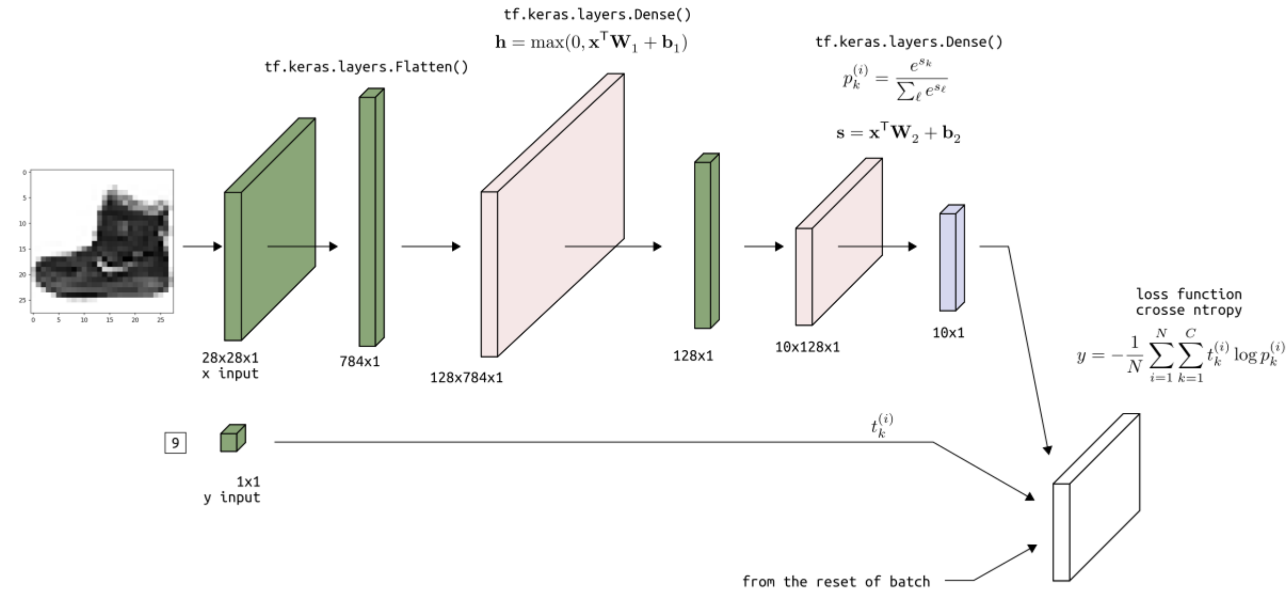
where N is the total image, C is the number of categories/classes, and $t_k^{(i)}$ is indicator function which is zero if the true category j of image i is not equal and one if they are equal. $s_k^{(i)}$ in that categorical entropy is an output score from the last layer. As a future reminder, if the provided label of the category is in one-hot encoding format, we should use **`categorical_entropy`**.

The last argument that we set in model.compile() is metric. Here, we use the simple metric **`accuracy`** that counts how many images that the model would correctly predict for computed weight in each epoch (or iteration) divided by the total number of images. We need to remember that this accuracy is not included in optimizing the weight matrix.

In line 36, we fit the model with training data with a small number of epochs. For this simple data set, we can achieve high accuracy, greater than 85% in the training data. But we have to know that this high accuracy *does not* imply how good the classifier is.

Finally in line 39, we test our classifier to the test data. If you typed correctly the script above, you will achieve a similar accuracy above 85%. In the above script, you can add more units in the first fully connected layer or set different epochs. This is what we call *hyperparameters*. Sometimes you should find this hyperparameter manually or you can use the hyperparameter optimization technique that we will discuss in the later chapter.

Like in the previous script, we can simplify all above descriptions into the following figure.



Now we introduce a handy class to interrupt the training process if some conditions are satisfied. This can be done by creating a callback class and overriding the existing method from **`tf.keras.callbacks.Callback`**..

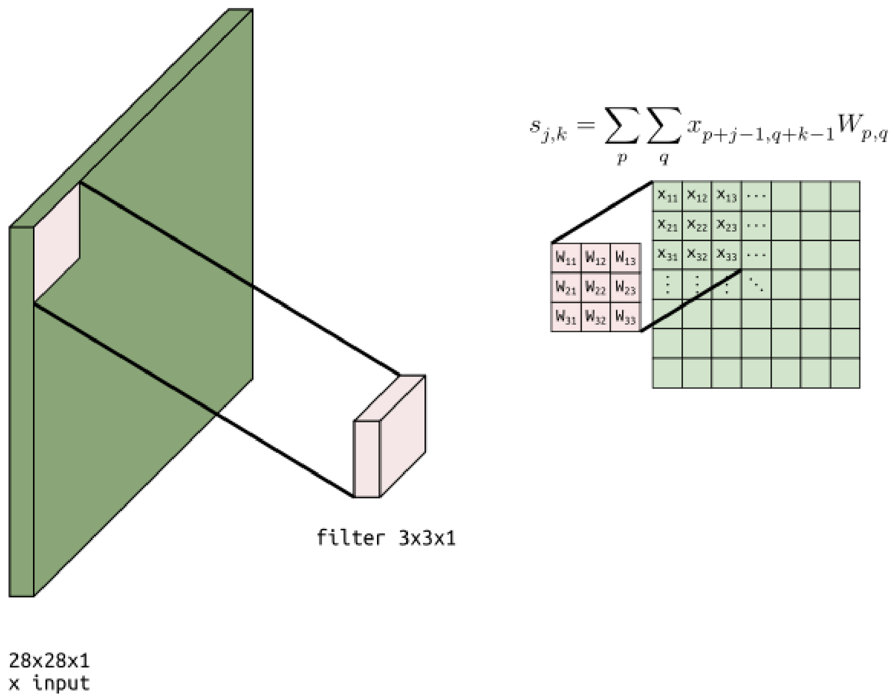
Listing 2.3. fashion_classifier_with_callback.py

```python
import tensorflow as tf


class MyCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        if logs.get("accuracy") > 0.6:
            print("\nReached 60% accuracy so cancelling training!")
            self.model.stop_training = True


if __name__ == "__main__":
    mnist = tf.keras.datasets.fashion_mnist

    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train, x_test = x_train / 255.0, x_test / 255.0

    callbacks = MyCallback()

    model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(512, activation=tf.nn.relu),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)
    ])

    model.compile(optimizer=tf.optimizers.Adam(),
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])

    # will reach 60% accuracy in 2 epochs!
    model.fit(x_train, y_train, epochs=10, callbacks=[callbacks])
```

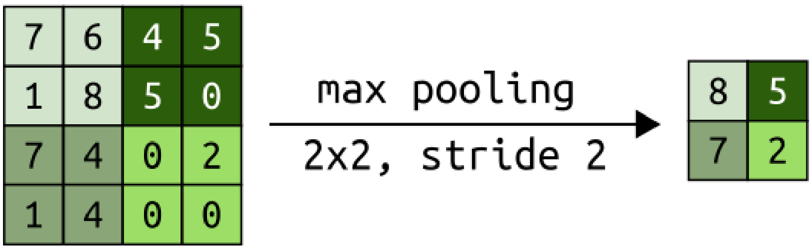
All the rest codes are the same as Listing 2.2. In lines 4-8, we add a **`MyCallback`** class where all the methods and attributes are inherited from the parent class **`tf.keras.callback.Callback`**. In that parent class, there is a method **`.on_epoch_end()`** where we can override that parent's class method by redefining that method with the description as in lines 6-8.

b. Classifying Fashion with CNN

We can also implement a model using CNN to classify the fashion category for a given image. Let us briefly introduce what CNN is. First we need to understand what convolution layer is. The following figure explains the basic principle of how to take a convolution operation.



The simple interpretation of the convolution operation is to preserve the feature of the image spatially and locally. Some simple examples of features are edges, ridges, or corners. This operation also somehow stores in the encoded way what is the best feature for each filtered area applied to the image. Every single move and operation of the convolution layer will result in a single number where we can think of that value like an activated neuron. All the weights of the filtered matrix will be learned by the model through the training process. In practice this convolution layer is subsequently followed by a max pooling layer. This layer aggregates the most activated feature in that image. The principle of max pooling layer is the same as convolution layer but instead of taking dot product for the superimposed value on the image with the filtered matrix, this max pooling layer takes the maximum value in the superimposed value by the max pooling filtered matrix. The following figure explains how the max pooling operates on the result of the convolution layer.



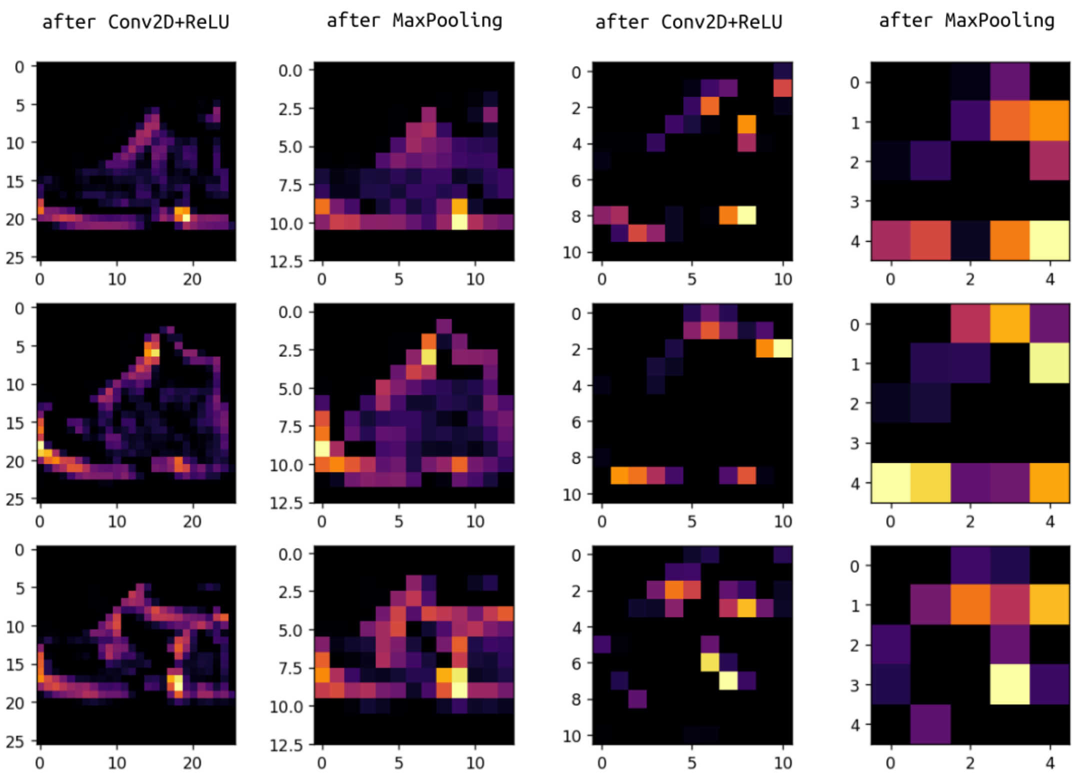
In the above figure, we take a 2x2 max pooling with stride 2. This stride means that we move the 2x2 max pooling every two steps of the element in the input matrix in row direction and column direction. In practice, the number of strides is the same as the size of the max pooling filter. This max pooling operation has an interpretation to select the most activated entry in the image to be accounted for by the computation of the weight matrix in the model. Now, let us apply that convolution and max pooling layer to the Fashion MNIST data set.

Listing 2.4. fashion_classifier_with_cnn.py

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np


def create_cnn_model():
   mnist = tf.keras.datasets.fashion_mnist

   (training_images, training_labels), (test_images, test_labels) = mnist.load_data()

   model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(64, (3, 3), activation="relu", input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(2, 2),
      tf.keras.layers.Conv2D(64, (3, 3), activation="relu"),
      tf.keras.layers.MaxPooling2D(2, 2),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(128, activation="relu"),
      tf.keras.layers.Dense(10, activation="softmax")
   ])

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

   model.fit(training_images, training_labels, epochs=10)
   test_loss = model.evaluate(test_images, test_labels)

   return [model, test_loss]


def visualizing_conv_and_max_pool(model):
   mnist = tf.keras.datasets.fashion_mnist
   (_, _), (test_images, test_labels) = mnist.load_data()

   # from this print out, we can decide the indices for
   # first_image, second_image, third_image
   print(test_labels[:100].reshape(10, 10))

   f, ax_arr = plt.subplots(3, 4, figsize=(10, 10))
   first_image = 0
   second_image = 28
   third_image = 23
   convolution_number = 2  # starting from 0 to 63

```

```
    layer_outputs = [layer.output for layer in model.layers]
    activation_model = tf.keras.models.Model(inputs=model.input,
                                             outputs=layer_outputs)

    for x in range(0, 4):
        f1 = activation_model.predict(test_images[first_image].reshape(1, 28, 28, 1))[x]
        ax_arr[0, x].imshow(f1[0, :, :, convolution_number], cmap="inferno")
        ax_arr[0, x].grid(False)

        f2 = activation_model.predict(test_images[second_image].reshape(1, 28, 28, 1))[x]
        ax_arr[1, x].imshow(f2[0, :, :, convolution_number], cmap="inferno")
        ax_arr[1, x].grid(False)

        f3 = activation_model.predict(test_images[third_image].reshape(1, 28, 28, 1))[x]
        ax_arr[2, x].imshow(f3[0, :, :, convolution_number], cmap="inferno")
        ax_arr[2, x].grid(False)

    plt.pause(2)


if __name__ == "__main__":
    cnn_model, cnn_test_loss = create_cnn_model()
    cnn_model.save("model-saved/category2.h5")

    visualizing_conv_and_max_pool(cnn_model)
```

Similar to Listing 2.3, now we add 4 layers [Conv2D+ReLU, MaxPooling, Conv2D+ReLU, MaxPooling] before the two-fully connected layers. We also add a function **`visualizing_conv_and_max_pool()`** to understand what is the influence of those four new layers. In the figure below, we can see the result of that function to the images category 9 (“ankle boot”). In the first application, we can see the layer Conv2D+ReLU tries to fire up the edges of the images. And the max pooling tries to make that edges become more significant to be accounted in the training process.



c. Classifying Emotion with CNN

In this section, we want to apply the CNN model to classify emotion from the following data sets. The dataset is prepared by Laurence Moroney (AI Lead at Google). You can download it from this link: https://storage.googleapis.com/laurencemoroney-blog.appspot.com/happy-or-sad.zip.
This data set contains two kinds of images of expression: happy or sad. Each kind of expression contains 40 images. All the images are generated from 3D emoji characters. In this example we want to show how to use a built-in preprocessing class in TensorFlow, the **`ImageGenerator`** class.. This class will be explored more in the next chapter. We also employ loading data set by a directory path instead of loading all image data to the program.

For the testing data, you can search any image with a happy or sad expression or download four sample images in here https://bit.ly/36LmNZe.

The following listing will show that with the small number of images and different unseen images as the testing data, we would highly expect the bad result on the prediction. And also increasing the number of layers will not solve the problem and severely make the prediction. The reason is the model overfitting to the data. This is the common problem that we should be aware of when we use machine learning as a classifier.

We also provide two functions that may come very handy to see the training data (**`plot_training_images()`**) and predict a category for a given image by transforming first to follow the input shape requirement of the model (**`classify_images()`**)

Listing 2.5. face_expression_classifier_with_cnn.py

```python
import os
import zipfile
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image as keras_image


class MyCallback(tf.keras.callbacks.Callback):
    def __init__(self, desired_accuracy):
        super(MyCallback, self).__init__()
        self.DESIRED_ACCURACY = desired_accuracy

    def on_epoch_end(self, epoch, logs=None):
        if logs.get("accuracy") > self.DESIRED_ACCURACY:
            print(f"\nReached {self.DESIRED_ACCURACY * 100:.2f}"
                  + " accuracy so cancelling training!")
            self.model.stop_training = True


def load_dataset(zip_file_path, extracted_zip_file_path, train_happy_dir, train_sad_dir):
    zip_ref = zipfile.ZipFile(zip_file_path, "r")
    zip_ref.extractall(extracted_zip_file_path)
    zip_ref.close()

    train_happy_names = os.listdir(train_happy_dir)
    train_sad_names = os.listdir(train_sad_dir)

    print(train_happy_names[:10])
    print(train_sad_names[:10])
    print(f"total training happy images {len(train_happy_names)}")
```

```python
    print(f"total training sad images {len(train_sad_names)}")

    return train_happy_names, train_sad_names


def do_data_preprocessing(dataset_dir):
    train_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        dataset_dir,
        target_size=(150, 150),
        batch_size=8,  # reduce this from 128 (our dataset is small)
        class_mode="binary"
    )

    return train_generator


def create_cnn_model():
    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(16, (3, 3), activation="relu", input_shape=(150, 150, 3)),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(32, (3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(64, (3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(512, activation="relu"),
        tf.keras.layers.Dense(1, activation="sigmoid")
    ])

    model.compile(loss="binary_crossentropy",
                  optimizer=Adam(learning_rate=0.001),
                  metrics=["accuracy"])

    model.summary()

    return model


def plot_training_images(train_happy_dir, train_sad_dir,
                        train_happy_names, train_sad_names):
    # Parameter for our graph; we'll output images in a 4x4 configuration
    n_rows = 4
    n_cols = 4

    # Index for iterating over images
    img_index = 0

    # Set up matplotlib fig, and size it to fit 4x4 pics
    fig = plt.gcf()
    fig.set_size_inches(n_cols * 4, n_rows * 4)

    img_index += 8
    next_happy_img = [os.path.join(train_happy_dir, fname)
                      for fname in train_happy_names[img_index - 8:img_index]]
    next_sad_img = [os.path.join(train_sad_dir, fname)
                    for fname in train_sad_names[img_index - 8:img_index]]

    for i, img_path in enumerate(next_happy_img + next_sad_img):
        # Set up subplot; subplot indices start at 1
        sp = plt.subplot(n_rows, n_cols, i + 1)
        sp.axis("off")  # don't show axes (or gridlines)

        img = mpimg.imread(img_path)
        plt.imshow(img)
        plt.pause(1.0)


def classify_images(fn_arr, model):
```

```
    for fn in fn_arr:
        path = "datasets/" + fn
        img = keras_image.load_img(path, target_size=(150, 150))
        x = keras_image.img_to_array(img)
        x = np.expand_dims(x, axis=0)

        image_i = np.vstack([x])
        classes = model.predict(image_i, batch_size=10)
        print(classes[0])
        if classes[0] > 0.5:
            print(fn + " is happy")
        else:
            print(fn + " is sad")


if __name__ == "__main__":
    zip_file_path = "datasets/happy-or-sad.zip"
    extracted_zip_file_path = "datasets/happy-or-sad"

    # Directory with our training happy images
    train_happy_dir = os.path.join("datasets/happy-or-sad/happy")

    # Directory with our training sad images
    train_sad_dir = os.path.join("datasets/happy-or-sad/sad")

    # Extract zip and get path to the data sets
    train_happy_names, train_sad_names = load_dataset(zip_file_path, extracted_zip_file_path,
                                                      train_happy_dir, train_sad_dir)

    # Plot the data sets
    plot_training_images(train_happy_dir, train_sad_dir, train_happy_names, train_sad_names)

    # Data preprocessing
    train_generator = do_data_preprocessing(extracted_zip_file_path)

    # Building a small CNN model
    cnn_model = create_cnn_model()

    # Training the model to the training data
    DESIRED_ACCURACY = 0.99
    callbacks = MyCallback(DESIRED_ACCURACY)

    history = cnn_model.fit(
        train_generator,
        steps_per_epoch=8,
        epochs=50,
        verbose=1,
        callbacks=[callbacks]
    )

    # Predict some images
    fn_arr = ["beauty-1132617_640.jpg", "girl-2961959_640.jpg",
              "woman-2126727_640.jpg", "beautiful-18279_640.jpg"]
    classify_images(fn_arr, cnn_model)
```

3. Convolutional Neural Network in TensorFlow

In the previous chapter, we have briefly introduced the convolution layer which is the building block of the convolutional neural network. Please review the previous chapter, if you have not read it yet.

a. Classifying Cats and Dogs

In this section, we will use images of cats and dogs from Kaggle competition (https://www.kaggle.com/c/dogs-vs-cats/data). But for the practical purpose, we use 1/10 of the total images as we continue building better classifiers for cats and dogs images. You can download the smaller version of the cats and dogs data set in (thanks again to Lawrence Moroney) https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip. For the testing images, you can use these 2 images of cats and 2 images of dogs from https://bit.ly/3xYRjuw.

Listing 3.1. cats_and_dogs_classifier_with_cnn.py

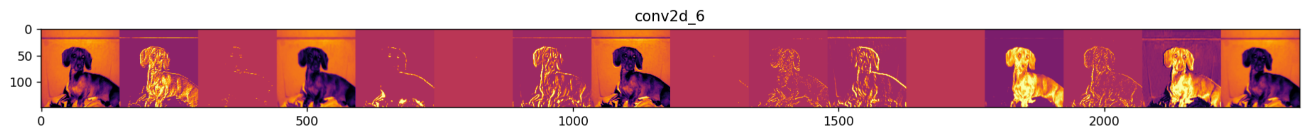
```
import os
import zipfile
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import tensorflow as tf

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
from tensorflow.keras.preprocessing import image as keras_image


def load_dataset(zip_file_path, extracted_zip_file_path):
   zip_ref = zipfile.ZipFile(zip_file_path, "r")
   zip_ref.extractall(os.path.split(extracted_zip_file_path)[0])
   zip_ref.close()

   train_dir = os.path.join(extracted_zip_file_path, "train")
   validation_dir = os.path.join(extracted_zip_file_path, "validation")

   train_cats_dir = os.path.join(train_dir, "cats")
   train_dogs_dir = os.path.join(train_dir, "dogs")

   validation_cats_dir = os.path.join(validation_dir, "cats")
   validation_dogs_dir = os.path.join(validation_dir, "dogs")

   train_cat_fnames = os.listdir(train_cats_dir)
   train_dog_fnames = os.listdir(train_dogs_dir)

   validation_cat_fnames = os.listdir(validation_cats_dir)
   validation_dog_fnames = os.listdir(validation_dogs_dir)

   print(train_cat_fnames[:10])
   print(train_dog_fnames[:10])

   print("total training cat images :", len(train_cat_fnames))
   print("total training dog images :", len(train_dog_fnames))

   print("total validation cat images:", len(validation_cat_fnames))
   print("total validation dog images:", len(validation_dog_fnames))

   return train_dir, validation_dir, train_cats_dir, train_dogs_dir, \
          train_cat_fnames, train_dog_fnames


def do_data_preprocessing(train_dir, validation_dir):
   train_datagen = ImageDataGenerator(rescale=1. / 255)
```

```python
    validation_datagen = ImageDataGenerator(rescale=1. / 255)

    train_generator = train_datagen.flow_from_directory(
        train_dir,
        batch_size=20,
        class_mode="binary",
        target_size=(150, 150)
    )

    validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        batch_size=20,
        class_mode="binary",
        target_size=(150, 150)
    )

    return train_generator, validation_generator


def create_cnn_model():
    # Building a small model from scratch: validation accuracy ~ 72 %

    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(16, (3, 3), activation="relu", input_shape=(150, 150, 3)),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(32, (3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(64, (3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(512, activation="relu"),
        tf.keras.layers.Dense(1, activation="sigmoid")
    ])

    model.summary()

    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss="binary_crossentropy",
                  metrics=["accuracy"])
    return model


def plot_cats_and_dogs(train_cats_dir, train_dogs_dir,
                      train_cat_fnames, train_dog_fnames):
    # Parameter for our graph; we'll output images in a 4x4 configuration
    nrows = 4
    ncols = 4

    # Index for iterating over images
    pic_index = 0

    # Set up matplotlib fig, and size it to fit 4x4 pics
    fig = plt.gcf()
    fig.set_size_inches(ncols * 4, nrows * 4)

    pic_index += 8
    next_cat_pix = [os.path.join(train_cats_dir, fname)
                    for fname in train_cat_fnames[pic_index - 8:pic_index]]
    next_dog_pix = [os.path.join(train_dogs_dir, fname)
                    for fname in train_dog_fnames[pic_index - 8:pic_index]]

    for i, img_path in enumerate(next_cat_pix + next_dog_pix):
        # Set up subplot; subplot indices start at 1
        sp = plt.subplot(nrows, ncols, i + 1)
        sp.axis("off")  # Don't show axes (or gridlines)

        img = mpimg.imread(img_path)
        plt.imshow(img)

    plt.pause(1.0)
```

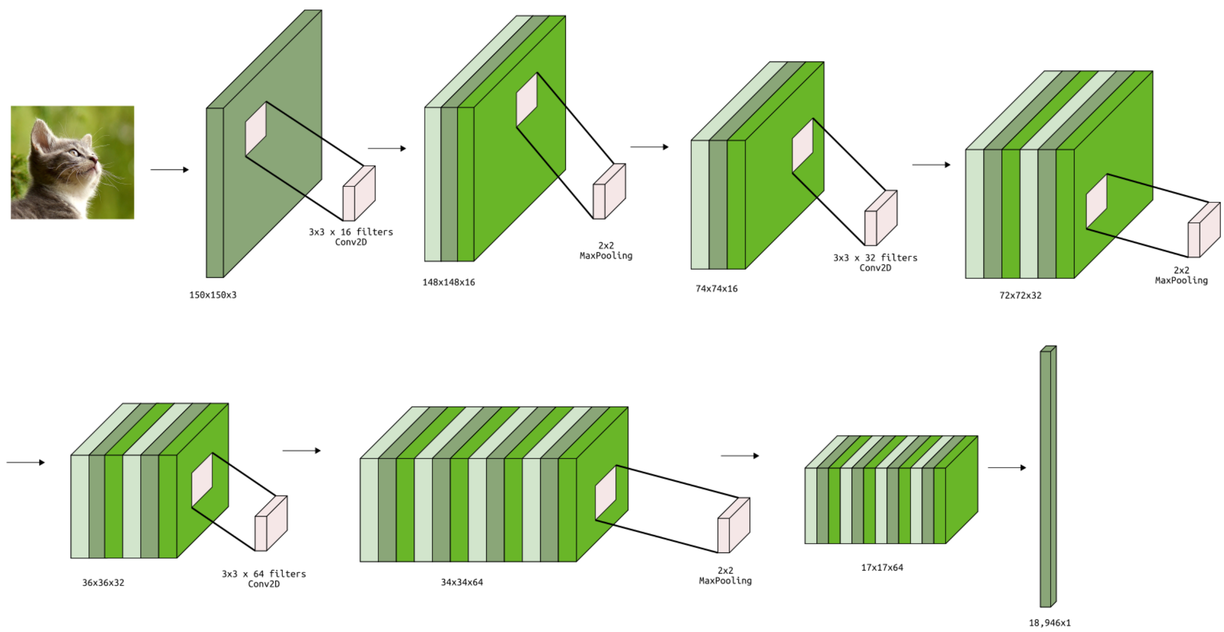
```python


def classify_images(fn_arr, model):
    for fn in fn_arr:
        path = "datasets/" + fn
        img = keras_image.load_img(path, target_size=(150, 150))
        x = keras_image.img_to_array(img)
        x = np.expand_dims(x, axis=0)

        image_i = np.vstack([x])
        classes = model.predict(image_i, batch_size=10)
        print(classes[0])
        if classes[0] > 0.5:
            print(fn + " is a dog")
        else:
            print(fn + " is a cat")


def plot_intermediate_repr(model, train_cats_dir, train_dogs_dir,
                           train_cat_fnames, train_dog_fnames):
    # Let's define a new Model that will take an image as input, and will output
    # intermediate representations for all layers in the previous model after the first.
    successive_outputs = [layer.output for layer in model.layers[0:]]
    visualization_model = tf.keras.models.Model(inputs=model.input,
                                                outputs=successive_outputs)

    # Let's prepare a random input image from the training set.
    cat_img_files = [os.path.join(train_cats_dir, f) for f in train_cat_fnames]
    dog_img_files = [os.path.join(train_dogs_dir, f) for f in train_dog_fnames]
    img_path = random.choice(cat_img_files + dog_img_files)

    img = load_img(img_path, target_size=(150, 150))  # this is a PIL image
    x = img_to_array(img)  # numpy array with shape (150, 150, 3)
    print(f"x.shape : {x.shape}")
    x = x.reshape((1,) + x.shape)
    x = x / 255.

    # Let's run our image through our network, thus obtaining all
    # intermediate representations for this image.
    successive_feature_maps = visualization_model.predict(x)

    # These are the names of the layers, so can have them as part of our plot
    layer_names = [layer.name for layer in model.layers]

    # Now let's display our representations
    for layer_name, feature_map in zip(layer_names, successive_feature_maps):
        # Just do this for the conv / maxpool layers, not the fully connected layers
        if len(feature_map.shape) == 4:
            n_features = feature_map.shape[-1]  # number of features (filters) in feature map

            # The feature map has shape (1, size, size, n_features)
            size = feature_map.shape[1]

            # We will tile our images in this matrix
            display_grid = np.zeros((size, size * n_features))
            for i in range(n_features):
                # Post-process the feature to make it visually palatable
                x = feature_map[0, :, :, i]
                x -= x.mean()
                # print('x.std()', x.std())
                x = x / x.std() if x.std() > 1e-14 else x
                x *= 64
                x += 128
                x = np.clip(x, 0, 255).astype("uint8")

                # We will tile each filter into this big horizontal grid
                display_grid[:, i * size: (i + 1) * size] = x

            # Display the grid
            plt.figure(figsize=(20, 2))
```

```python
            plt.title(layer_name)
            plt.grid(False)
            plt.imshow(display_grid, aspect="auto", cmap="viridis")
            plt.subplots_adjust(left=0.03, right=0.99)

    plt.pause(1.0)


def plot_history(train, val, title):
    epochs = range(len(train))
    plt.figure()
    plt.plot(epochs, train, label="train")
    plt.plot(epochs, val, label="val")
    plt.title(title)
    plt.legend(loc="best")
    plt.pause(1.0)


if __name__ == "__main__":
    zip_file_path = "datasets/cats_and_dogs_filtered.zip"
    extracted_zip_file_path = "datasets/cats_and_dogs_filtered"

    # Extract zip and get path to the data sets
    train_dir, validation_dir, train_cats_dir, train_dogs_dir, \
        train_cat_fnames, train_dog_fnames \
            = load_dataset(zip_file_path, extracted_zip_file_path)

    # Plot the data sets
    plot_cats_and_dogs(train_cats_dir, train_dogs_dir,
                       train_cat_fnames, train_dog_fnames)

    # Data preprocessing
    train_generator, validation_generator \
        = do_data_preprocessing(train_dir, validation_dir)

    # Building a small CNN model
    cnn_model = create_cnn_model()

    history = cnn_model.fit(
        train_generator,
        validation_data=validation_generator,
        steps_per_epoch=100,
        epochs=15,
        validation_steps=50,
        verbose=1
    )

    fn_arr = ["cat-2083492_only_head.jpg", "cat-1146504_640.jpg",
              "dog-3846767_640.jpg", "dog-3388069_640.jpg"]
    classify_images(fn_arr, cnn_model)

    # Visualizing intermediate representations
    plot_intermediate_repr(cnn_model, train_cats_dir, train_dogs_dir,
                           train_cat_fnames, train_dog_fnames)

    # Evaluating accuracy and loss for the model
    acc = history.history["accuracy"]
    val_acc = history.history["val_accuracy"]
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]

    plot_history(acc, val_acc, "Training and validation accuracy")
    plot_history(loss, val_loss, "Training and validation loss")
```

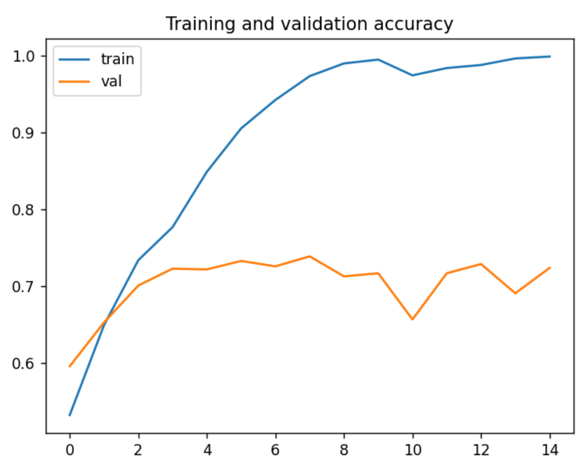
In the above listing, we add two more functions: **`plot_intermediate_repr()`** and **`plot_history()`**. The first function is to create several plots for what happened to the input images along the operation of convolutional layers and max pooling layers. The figure below shows the first effect of the first convolutional layer.

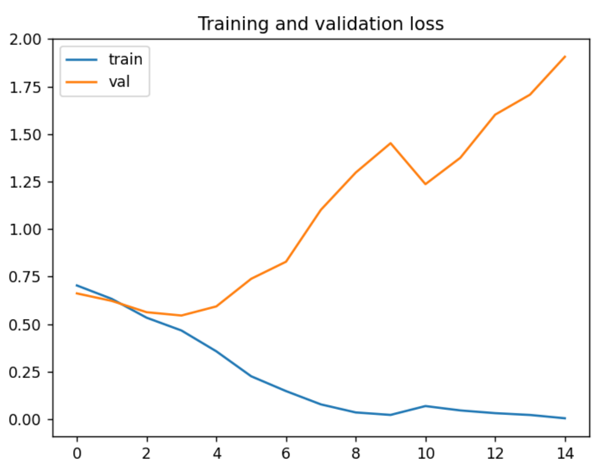


We see that the parameters or weight matrix of convolutional layers, which mainly control the learning process, try to find primitive features like edges, blobs, or corners. This is pretty amazing how the process is to be done automatically. In the below is how the tensor input image (multi-dimensional arrays) transforms its dimension after each operation in each layer.



The second function has a purpose to create a plot of loss vs. epoch and accuracy vs. epoch. This plot is very crucial as we tweak the hyperparameters such that it will give the optimal weights matrix of the model. If we run the program above, the two last functions **`plot_history()`**, we give the following figures.



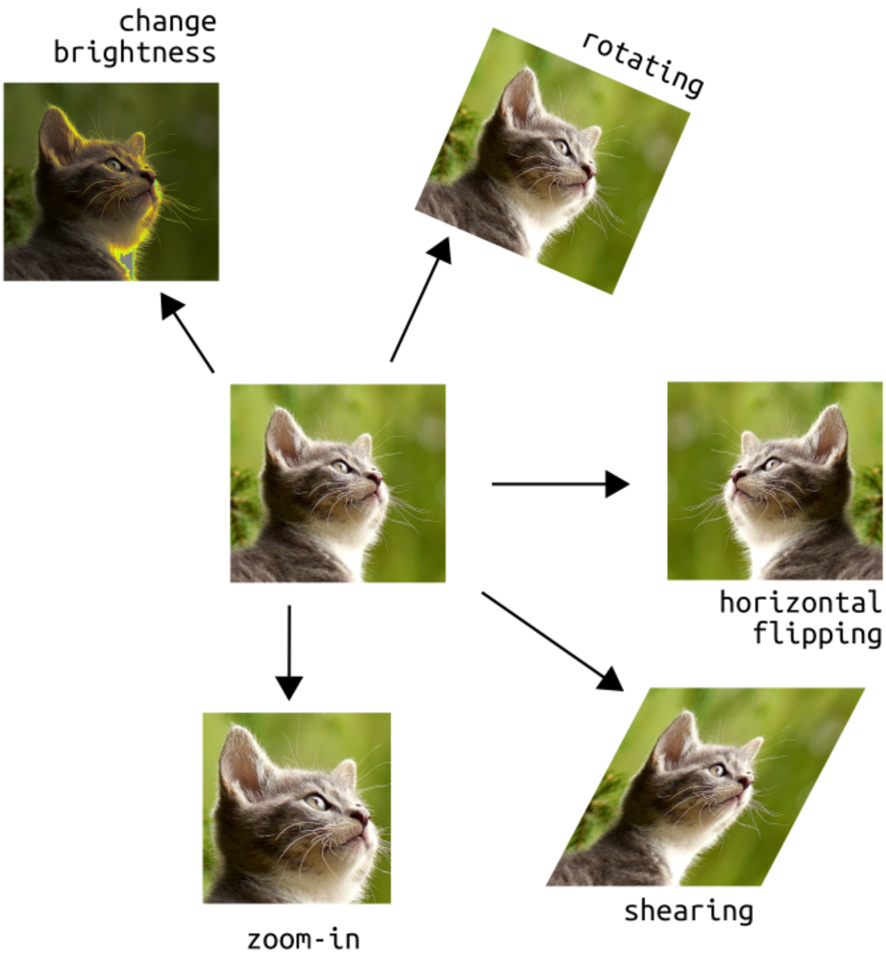


We clearly see that the model is overfitting to the training data as it indicates a large gap between the curve of training and validation data in the later epochs. In the next section, we will address this issue

b. Improving Cats and Dogs Classification

If we run the previous listing program (Listing 3.1), we will have accuracy for the validation data stalling at the level 0.72 even though the accuracy of the training data is close to 1. This is the overfitting which always caught many deep learning models. There are many ways to improve the classifier. For our case, we use two methods which are by image augmentation and dropout of some layer's connection.

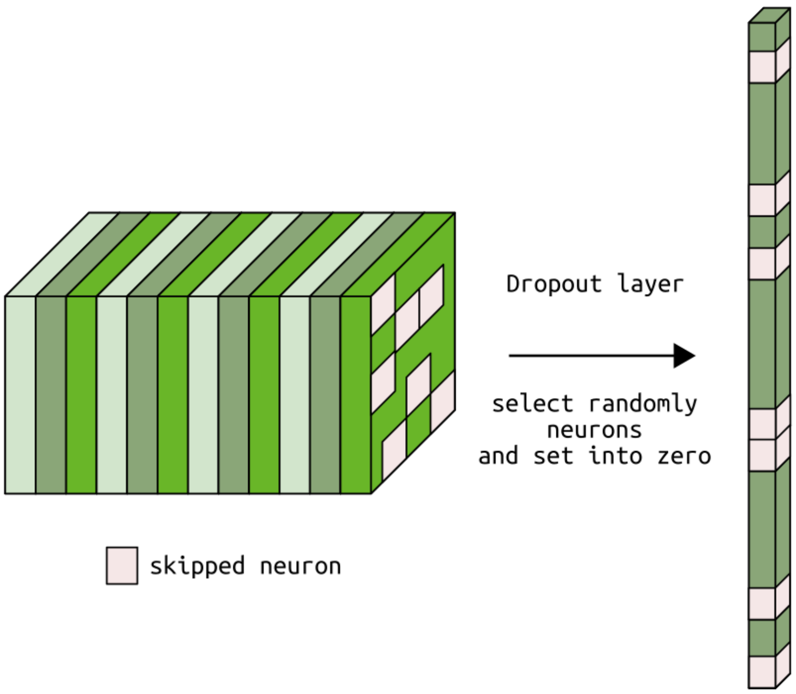
Before we improve our code, let us discuss briefly what is image augmentation and dropout layer. From its name, we can guess that image augmentation is a way to augment the existing image with some image transformations like zoom in/out (scaling+cropping/filling), rotating, shearing, flipping (mirroring), shifting, etc.



Fortunately, TensorFlow provides a handy class to do all such transformations above which is **`ImageDataGenerator()`**. We have already introduced that class in Listing 2.5. In the upcoming code, we will use several other attributes and methods from that class.

Then for the dropout layer, it serves as a masking layer which skips some entries in its input.

In practice, a dropout layer is put before fully-connected layers and after all combination of convolutional and max pooling layers. We put it after all combination of convolutional and max pooling layers to make the learning process of the model to account for all the possibilities first. Then hopefully by dropping out some specific neuron, the model only considers the most important features and withdraws all the redundant features. Inside computation process of dropout layer is show in figure below



Fully equipped with the **`ImageDataGenerator()`** and dropt out layers, we are ready to improve the previous CNN model in Listing 2.7

Listing 3.2. cats_and_dogs_classifier_with_imagedatagen_and_dropout.py

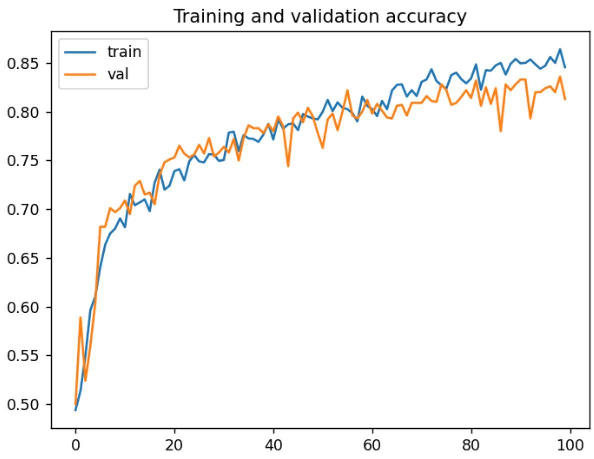
```
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image as keras_image


def plot_history(train, val, title):
   epochs = range(len(train))
   plt.figure()
   plt.plot(epochs, train, label="train")
   plt.plot(epochs, val, label="val")
   plt.title(title)
   plt.legend(loc="best")
   plt.pause(1.0)


def do_data_preprocessing(train_dir, validation_dir, aug=False):
   if aug:
       train_datagen = ImageDataGenerator(
           rescale=1./255,
           rotation_range=40,
```

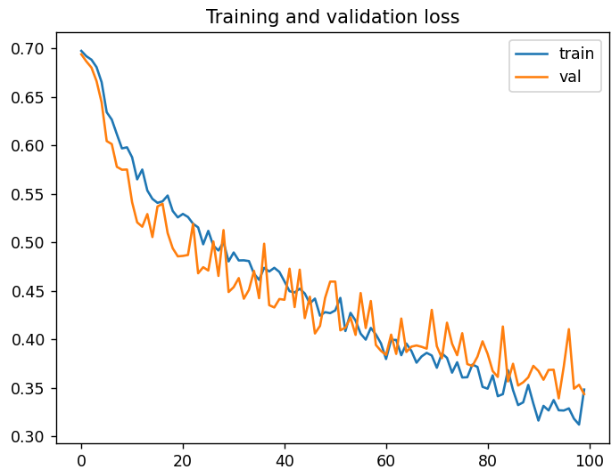
```
            width_shift_range=0.2,
            height_shift_range=0.2,
            shear_range=0.2,
            zoom_range=0.2,
            horizontal_flip=True,
            fill_mode="nearest"
        )
    else:
        train_datagen = ImageDataGenerator(rescale=1./255)

    validation_datagen = ImageDataGenerator(rescale=1./255)

    train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode="binary"
    )

    validation_generator = validation_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode="binary"
    )

    return train_generator, validation_generator


def create_cnn_model():
    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(32, (3, 3), activation="relu", input_shape=(150, 150, 3)),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(64, (3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(128, (3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Conv2D(128, (3, 3), activation="relu"),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Dropout(0.5),           # 3rd options (with image augmentation)
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(512, activation="relu"),
        tf.keras.layers.Dense(1, activation="sigmoid")
    ])

    model.compile(loss="binary_crossentropy",
                  optimizer=Adam(lr=1e-4),
                  metrics=["accuracy"])

    model.summary()

    return model


def classify_images(fn_arr, model):
    for fn in fn_arr:
        path = "datasets/" + fn
        img = keras_image.load_img(path, target_size=(150, 150))
        x = keras_image.img_to_array(img)
        x = np.expand_dims(x, axis=0)

        image_i = np.vstack([x])
        classes = model.predict(image_i, batch_size=10)
        print(classes[0])
        if classes[0] > 0.5:
            print(fn + " is a dog")
        else:
            print(fn + " is a cat")
```

```
96  if __name__ == "__main__":
97      base_dir = "datasets/cats_and_dogs_filtered"
98      train_dir = os.path.join(base_dir, "train")
99      validation_dir = os.path.join(base_dir, "validation")
100 
101     train_cats_dir = os.path.join(train_dir, "cats")
102     train_dogs_dir = os.path.join(train_dir, "dogs")
103 
104     validation_cats_dir = os.path.join(train_dir, "dogs")
105     validation_dogs_dir = os.path.join(validation_dir, "cats")
106 
107     print(len(os.listdir(train_cats_dir)))
108     print(len(os.listdir(train_dogs_dir)))
109     print(len(os.listdir(validation_cats_dir)))
110     print(len(os.listdir(validation_dogs_dir)))
111 
112     # 1st options without image augmentation and dropout layer
113     # 2nd options with image augmentation without dropout layer
114     # 3rd options with image augmentation and dropout layer
115     train_generator, validation_generator \
116         = do_data_preprocessing(train_dir, validation_dir, aug=True)
117 
118     # Build a CNN model
119     cnn_model = create_cnn_model()
120 
121     history = cnn_model.fit(
122         train_generator,
123         # steps_per_epoch=100,  # 2000 images = batch_size * steps
124         epochs=100,
125         validation_data=validation_generator,
126         # validation_steps=50,
127         verbose=1
128     )
129 
130     acc = history.history["accuracy"]
131     val_acc = history.history["val_accuracy"]
132     loss = history.history["loss"]
133     val_loss = history.history["val_loss"]
134 
135     plot_history(acc, val_acc, "Training and validation accuracy")
136     plot_history(loss, val_loss, "Training and validation loss")
137 
138     # test the model
139     fn_arr = ["cat-2083492_only_head.jpg", "cat-1146504_640.jpg",
140               "dog-3846767_640.jpg", "dog-3388069_640.jpg"]
141     classify_images(fn_arr, cnn_model)
```

In lines 23-32, we add several image transformations: rotation, shifting, shearing, zoom in/out, and flipping. Due to those transformations, there would be an empty pixel area that should be filled. In line 31, we put the keyword argument **`fill_mode="nearest"`** to fill that empty pixel area with the nearest image value. All the rest of the lines are similar to the Listing 3.1.

If we run the above program, we will get the following results for accuracy and loss. Comparing these results to the previous result of Listing 3.1, will give so much difference on how validation curves tend to be close to the training curve. This indicates that improvement is working and might give a better result. If you try to apply this model to the four sample test images of dog and cats, see lines 139-141, we will get slightly better results, unless it might give one incorrect result due to the very differently unseen data.

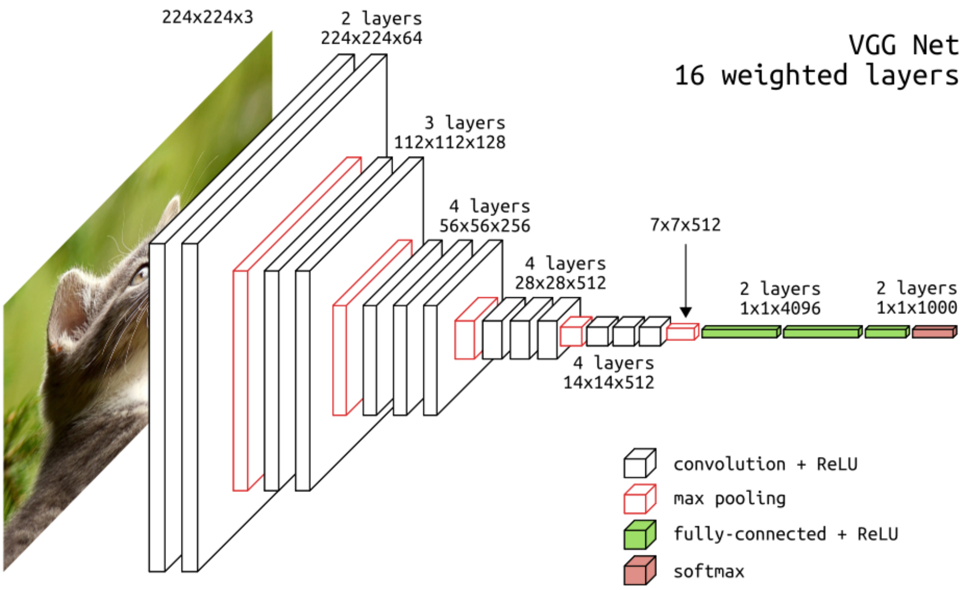




c. Transfer Learning based on VGG Net

Now, we will use the most useful technique in machine learning in which we do not need to train our model to the huge amount of data if we only have a small amount of data to be trained. This is called transfer learning. The idea is we use a pre-trained model and unfreeze some layers near the end layers. After that we train that pre-trained model using our small data. If our problem is similar to the problem that the pre-trained model has solved, we happily get the fastest training and accurate classifier.

To show the usefulness of transfer learning, we demonstrate it using VGG Net. For the detailed description about this CNN architecture, you can read the following paper by (Simonyan and Zisserman, 2014). The following figure shows a simple abstraction of VGG Net.



adapted from (Cord, 2016)

VGG Net has been trained for 1.3M images with 1000 classes which is part of the ILSVRC-2012 dataset. The pre-trained weight of VGG Net on that dataset can be downloaded in https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5. This pre-trained weight is a weight without the weight of fully-connected layers because we will change these fully-connected layers with our need for a cat and dog classifier. Now, let us look at the listing program for transfer learning.

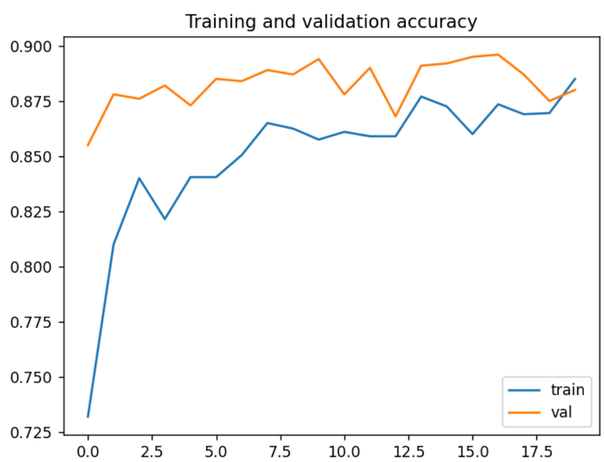
Listing 3.3. cats_and_dogs_classifier_with_transfer_learning.py

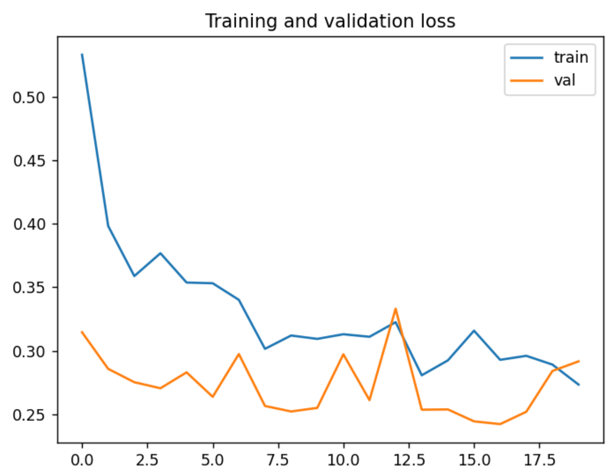
```
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import Model
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image as keras_image


def plot_history(train, val, title):
   epochs = range(len(train))
   plt.figure()
   plt.plot(epochs, train, label="train")
   plt.plot(epochs, val, label="val")
   plt.title(title)
   plt.legend(loc="best")
   plt.pause(1.0)


def do_data_preprocessing(train_dir, validation_dir, aug=False):
   if aug:
       train_datagen = ImageDataGenerator(
           rescale=1./255,
           rotation_range=40,
           width_shift_range=0.2,
           height_shift_range=0.2,
           shear_range=0.2,
           zoom_range=0.2,
           horizontal_flip=True,
           fill_mode="nearest"
       )
   else:
       train_datagen = ImageDataGenerator(rescale=1./255)

   validation_datagen = ImageDataGenerator(rescale=1. / 255)

   train_generator = train_datagen.flow_from_directory(
       train_dir,
       batch_size=20,
       class_mode="binary",
       target_size=(150, 150)
   )

   validation_generator = validation_datagen.flow_from_directory(
       validation_dir,
       batch_size=20,
       class_mode="binary",
       target_size=(150, 150)
   )

   return train_generator, validation_generator


def create_cnn_model(local_weights_file):
   pre_trained_model = VGG16(input_shape=(150, 150, 3),
```

```
                                 include_top=False,
                                 weights=None)

    pre_trained_model.load_weights(local_weights_file)

    for layer in pre_trained_model.layers:
        layer.trainable = False

    pre_trained_model.summary()

    last_layer = pre_trained_model.get_layer("block5_pool")
    print("last layer output shape: ", last_layer.output_shape)
    last_output = last_layer.output

    x = layers.Dropout(0.2)(last_output)
    x = layers.Flatten()(x)
    x = layers.Dense(1024, activation="relu")(x)
    x = layers.Dense(1, activation="sigmoid")(x)


    model = Model(pre_trained_model.input, x)

    model.compile(optimizer=Adam(learning_rate=1e-4),
                  loss="binary_crossentropy",
                  metrics=["accuracy"])

    return model


def classify_images(fn_arr, model):
    for fn in fn_arr:
        path = "datasets/" + fn
        img = keras_image.load_img(path, target_size=(150, 150))
        x = keras_image.img_to_array(img)
        x = np.expand_dims(x, axis=0)

        image_i = np.vstack([x])
        classes = model.predict(image_i, batch_size=10)
        print(classes[0])
        if classes[0] > 0.5:
            print(fn + " is a dog")
        else:
            print(fn + " is a cat")


if __name__ == "__main__":
    local_weights_file = "pre-trained/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5"

    base_dir = "datasets/cats_and_dogs_filtered"
    train_dir = os.path.join(base_dir, "train")
    validation_dir = os.path.join(base_dir, "validation")

    train_cats_dir = os.path.join(train_dir, "cats")
    train_dogs_dir = os.path.join(train_dir, "dogs")

    validation_cats_dir = os.path.join(validation_dir, "cats")
    validation_dogs_dir = os.path.join(validation_dir, "dogs")

    train_cat_fnames = os.listdir(train_cats_dir)
    train_dog_fnames = os.listdir(train_dogs_dir)

    print(len(train_cat_fnames))
    print(len(train_dog_fnames))

    train_generator, validation_generator \
        = do_data_preprocessing(train_dir, validation_dir, aug=True)

    # Build a CNN model with pre-trained VGG16 net
    cnn_model = create_cnn_model(local_weights_file)

```

```
130     history = cnn_model.fit(
131         train_generator,
132         #steps_per_epoch=100,
133         epochs=20,
134         validation_data=validation_generator,
135         #validation_steps=50,
136         verbose=1
137     )
138 
139     acc = history.history["accuracy"]
140     val_acc = history.history["val_accuracy"]
141     loss = history.history["loss"]
142     val_loss = history.history["val_loss"]
143 
144     plot_history(acc, val_acc, "Training and validation accuracy")
145     plot_history(loss, val_loss, "Training and validation loss")
146 
147     # test the model
148     fn_arr = ["cat-2083492_only_head.jpg", "cat-1146504_640.jpg",
149               "dog-3846767_640.jpg", "dog-3388069_640.jpg"]
150     classify_images(fn_arr, cnn_model)
```

In the listing above specifically in line 59, we change the input shape of the image into 150x150 instead using the standard VGG Net 224x224. Then we also provide with pre-download the weight of pre-trained VGG Net on ILSVRC-2012 dataset. Then we use the TensorFlow **`Model`** class and also VGG16 architecture from **`tensorflow.keras.applications.vgg16`**. Of course, if you are willing to build the VGG Net architecture by yourself, you certainly can do that. But, we prefer to load that architecture from the TensorFlow library. In line 68, we print the summary of VGG Net to know the last layer's name where we used it as an input in line 70. We attach two fully-connected layers with a dropout layer before them. This dropout layer serves as we don't want to inherit so many features from VGG Net pre-trained weight. If we run the listing above, for using only 20 epochs, we arrive with the similar shape of training and validation curves as we got in the previous section.





d. Classifying Images of Sign Languages

We end this chapter by performing classification of sign language dataset. First you can download the datasets and four sample images from the following links:

- training data: https://bit.ly/3kStktw
- validation data: https://bit.ly/3kUArl0
- four sample images: https://bit.ly/3iJFke1

Those datasets are acquired from its original source from Kaggle competition “Sign Language MNIST” (https://www.kaggle.com/datamunge/sign-language-mnist). We put in those links such that you don’t need to sign-in to download that data. The datasets are in .csv format where each column represents the pixel values with range from 0 to 255. In total there are 28x28 pixels. Each row represents the one image of hand-sign. The following is a table of the pairs of images and its corresponding alphabet.

If you plot one of the images from the .csv file, you will get the grayscale version and much smaller resolution. Looking carefully on the above table, you can notice that there is no representation of hand-sign for letters “J” and “Z”. These two letters are represented by the finger's motion so we can’t include them.

The CNN model that we used is similar to the CNN model that we used in cat and dog classifiers but shallower layers. We also add a function **`get_data()`** to convert data from the .csv format into a NumPy array. Without further ado, let us look at the listing program for this classifier.

Listing 3.4. hand_sign_language_classifier.py

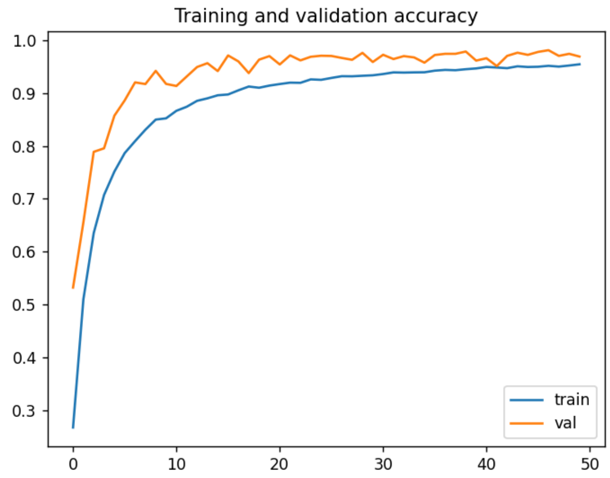
```
import os
import sys
import numpy as np
import matplotlib.pyplot as plt

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image as keras_image

import tensorflow as tf


def get_data(filename):
    with open(filename) as training_file:
        _ = training_file.readline()  # skip first line
        data = training_file.readlines()

```

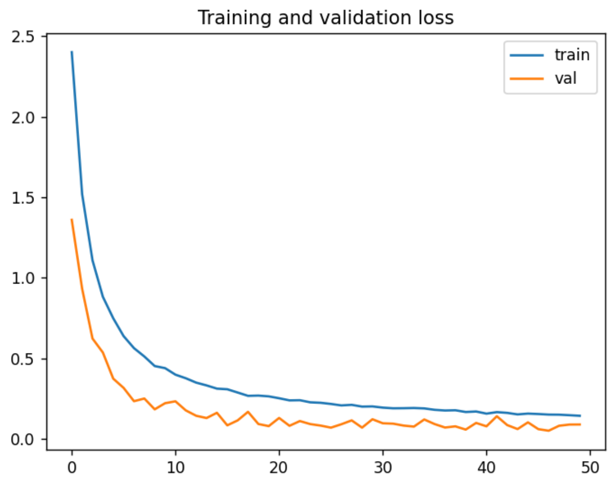
```
        labels = []
        images = []
        num_of_data = len(data)
        for i, row in enumerate(data):
            row = row.strip("\n").split(",")
            labels.append(row[0])
            images.append(np.array_split(row[1:785], 28))

            sys.stdout.write(f"\rprocessing: {(i + 1) / float(num_of_data) * 100:.2f} %")
            sys.stdout.flush()

        print("")
        labels = np.array(labels).astype(float)
        images = np.array(images).astype(float)

    return images, labels


def plot_one_image(image_data, image_label):
    fig, ax = plt.subplots()

    ax.imshow(image_data, cmap="gray", vmin=0, vmax=255)

    num_to_alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
                       'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
                       'T', 'U', 'V', 'W', 'X', 'Y']
    ax.set_title("image_label = {:g} ({:s})".format(image_label,
                                                   num_to_alphabet[int(image_label)]))

    plt.pause(1.0)


def do_data_preprocessing(training_images, training_labels,
                          validation_images, validation_labels):
    train_datagen = ImageDataGenerator(
        rescale=1. / 255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        fill_mode="nearest"
    )

    validation_datagen = ImageDataGenerator(rescale=1. / 255)

    training_generator = train_datagen.flow(
        training_images,
        training_labels,
        batch_size=20,
    )

    validation_generator = validation_datagen.flow(
        validation_images,
        validation_labels,
        batch_size=20
    )

    return training_generator, validation_generator


def create_cnn_model():
    # image size is 28x28, we don't need third conv2d
    # This will make the image size 1x1px!
    # second conv2D will make the image size 5x5px

    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(64, (3, 3), activation="relu", input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(2, 2),
        # tf.keras.layers.Conv2D(64, (3, 3), activation="relu"),
```

```
        # tf.keras.layers.MaxPooling2D(2, 2),
        # tf.keras.layers.Conv2D(128, (3, 3), activation="relu"),
        # tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Dropout(0.1),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(1024, activation="relu"),
        tf.keras.layers.Dense(26, activation="softmax")  # labels have value 0 - 24
    ])

    model.compile(
        loss="sparse_categorical_crossentropy",
        optimizer=Adam(learning_rate=0.001),
        metrics=["accuracy"]
    )

    model.summary()

    return model


def plot_history(train, val, title):
    epochs = range(len(train))
    plt.figure()
    plt.plot(epochs, train, label="train")
    plt.plot(epochs, val, label="val")
    plt.title(title)
    plt.legend(loc="best")
    plt.pause(1.0)


def classify_images(fn_arr, model):
    for fn in fn_arr:
        path = "datasets/" + fn

        # because we train using grayscale image, we need to convert
        # the sample image using color_mode="grayscale"
        img = keras_image.load_img(path, target_size=(28, 28),
                                   color_mode="grayscale")
        x = keras_image.img_to_array(img)
        x = np.expand_dims(x, axis=0)

        num_to_alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
                           'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
                           'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
        num_to_alphabet = np.array(num_to_alphabet)

        image_i = np.vstack([x])
        classes = model.predict(image_i, batch_size=10)
        print(classes[0])
        class_label = num_to_alphabet[classes[0].astype(np.int32) > 0.5][0]
        print(fn + " is a letter {:s}".format(class_label))


if __name__ == "__main__":
    # Load the dataset
    training_images, training_labels \
        = get_data(os.getcwd() + "/datasets/sign_mnist_train.csv")
    validation_images, validation_labels \
        = get_data(os.getcwd() + "/datasets/sign_mnist_test.csv")

    print(training_images.shape)
    print(training_labels.shape)
    print(validation_images.shape)
    print(validation_labels.shape)

    training_images = np.expand_dims(training_images, axis=-1)
    validation_images = np.expand_dims(validation_images, axis=-1)

    print(training_images.shape)
    print(validation_images.shape)
```

```python

    print(np.max(validation_labels), np.min(validation_labels))
    print(np.max(training_labels), np.min(training_labels))

    # Plot one of the images and its label
    image_num = 2
    plot_one_image(training_images[image_num, :, :, 0],
                   training_labels[image_num])

    # Data pre-preprocessing with ImageDataGenerator
    training_generator, validation_generator \
        = do_data_preprocessing(training_images, training_labels,
                                validation_images, validation_labels)

    # Build a CNN model
    cnn_model = create_cnn_model()

    history = cnn_model.fit(
        training_generator,
        # steps_per_epoch=len(training_images)/20,
        epochs=50,
        validation_data=validation_generator,
        # validation_steps=len(validation_images)/20
    )

    # Evaluate the model on the validation dataset
    cnn_model.evaluate(validation_generator)

    acc = history.history["accuracy"]
    val_acc = history.history["val_accuracy"]
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]

    plot_history(acc, val_acc, "Training and validation accuracy")
    plot_history(loss, val_loss, "Training and validation loss")

    # Test on sample images
    fn_arr = ["alphabet-letter-C-1298289_640.png",
              "alphabet-letter-D-1298315_640.png",
              "alphabet-letter-Y-1298311_640.png",
              "sign-language-letter-A-28717_640.png"]
    classify_images(fn_arr, cnn_model)
```

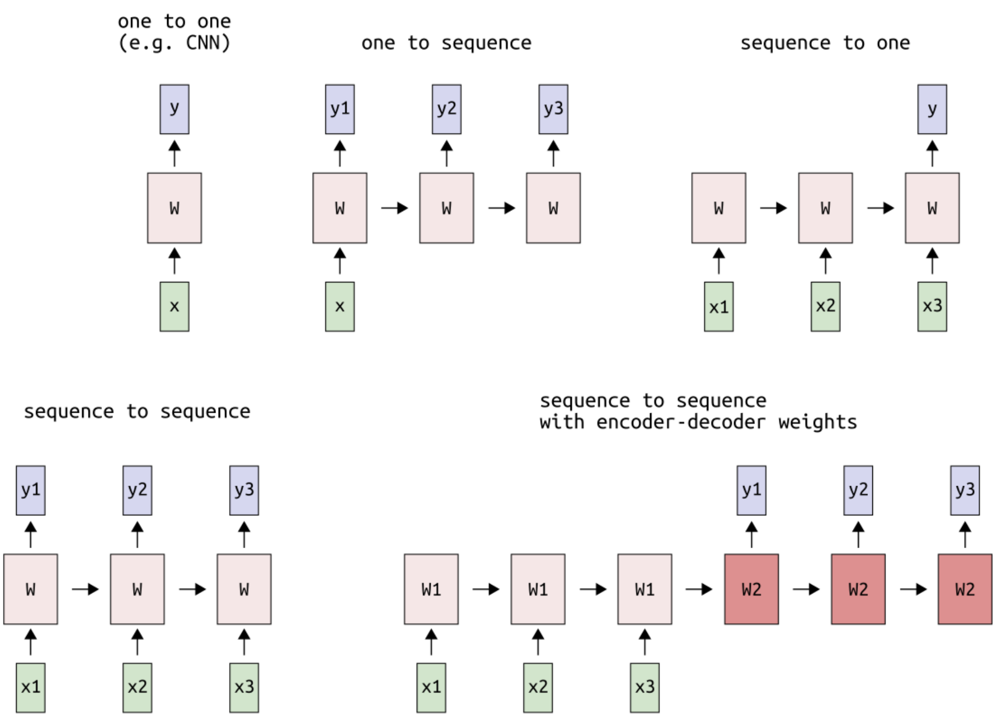
If we run the listing program above, we will get the following result for accuracy and loss curve for training and validation datasets. We can see clearly that the curve of the validation dataset is slightly higher in accuracy and slightly lower in loss. This indicates that our model is underfitting. The model is not complex enough to capture all the features in the images. For TensorFlow Developer Certification, having a result like these figures, is more than enough. But, you are free to explore how to build a complex model to resolve this issue.





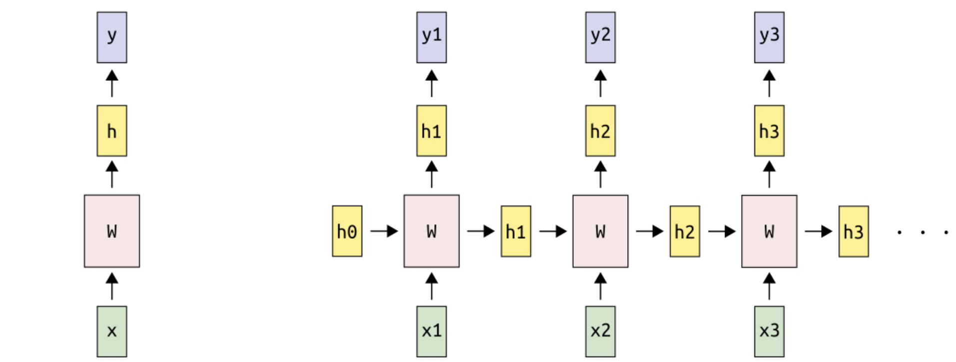
4. Natural Language Processing in TensorFlow

Before we jump into the program, first we need to understand what LSTM (Long Short-Term Memory) architecture is. Unfortunately, before we explain LSTM, we need to know the Recurrent Neural Network or RNN. Now, let us start with the RNN. This kind of network is like a generalization of computational graphs where we have recurrent relations inside the architecture as we unravel the input or output sequence. In the following figure, we summarize all types of base recurrent neural network architectures.



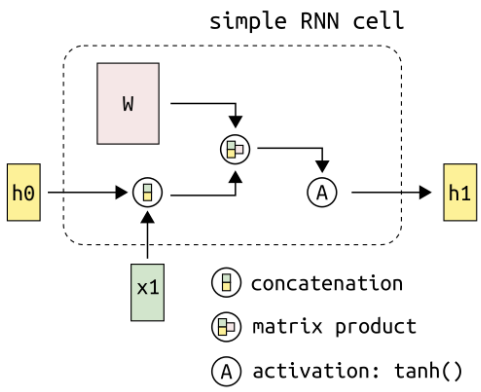
From the above figure, the horizontal arrow shows the recurrence relation. These horizontal arrows show that there are values (*hidden values*) which have been passing through the weight matrices along the horizontal arrows. In recurrent neural networks, you can feed the model with a single input or sequence of inputs. And at the end of the layers, you can obtain a single output or sequence of outputs. In the outgoing arrows of the weight matrix above, we didn't show the hidden layers. Most of the time until the end of this short book, we only use sequence to sequence of recurrent neural networks. The following figure is the comparison between CNN and RNN on how they handle input data and compute hidden output.

$$\mathbf{h} = \tanh(\mathbf{x}^\top \mathbf{W} + \mathbf{b}) \qquad \mathbf{h}_{t+1} = \tanh\left(\begin{bmatrix}\mathbf{x}_{t+1} \\ \mathbf{h}_t\end{bmatrix}^\top \mathbf{W} + \mathbf{b}\right)$$



In RNN, we concatenate matrix input x_{t+1} and hidden matrix h_t to compute h_{t+1}. Explicitly, the basic RNN is defined as the formula in the right part of the above figure. De-reconstruct this formula into computational graphs give us the following figure

$$\mathbf{h}_{t+1} = \tanh\left(\begin{bmatrix}\mathbf{x}_{t+1} \\ \mathbf{h}_t\end{bmatrix}^\top \mathbf{W} + \mathbf{b}\right)$$

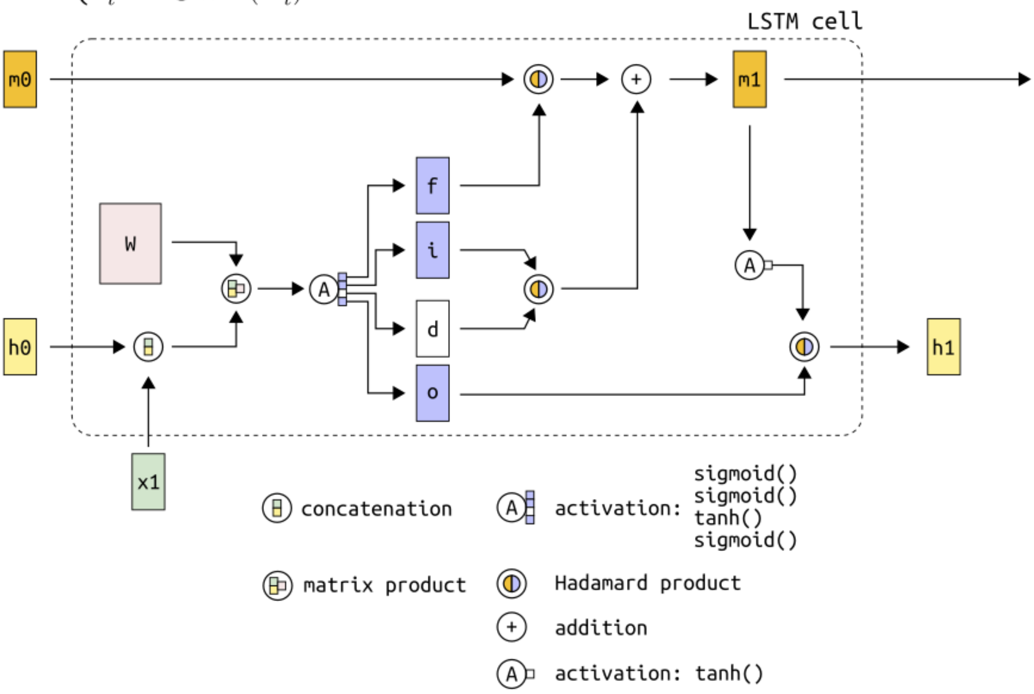


The RNN architecture got its name because we need to compute recurrence relation of h as it is shown in the equation of the figure above. Next, we move to LSTM.

The LSTM architecture is an extension of RNN but with several twists that can handle issues in basic RNN. The reason why people move from a basic RNN to LSTM, because RNN has a risk of exploding and vanishing gradient when we compute the gradient, $\partial L/\partial h_0$, through backpropagation from the output layer to the first hidden layer. Here $L$is a loss of the training data. In the backpropagation procedure, the computation of the gradient will compound up because of several times multiplication by weight matrix$W$. To resolve this exploding/vanishing gradient, they built RNN architecture such that it has a memory matrix to “remember” what was learned by the weight matrix W in a single unit of LSTM cell. The following figure is a computational graph representation of a single LSTM cell.

$$\begin{bmatrix} \mathbf{f}\\ \mathbf{i}\\ \mathbf{g}\\ \mathbf{o}\end{bmatrix} = \begin{bmatrix}\text{sigmoid}\\ \text{sigmoid}\\ \tanh\\ \text{sigmoid}\end{bmatrix}\left(\begin{bmatrix}\mathbf{x}_{t+1}\\ \mathbf{h}_t\end{bmatrix}^{\mathsf{T}} \mathbf{W} + \mathbf{b}\right)$$

$$\begin{cases} \mathbf{m}_t = \mathbf{f} \odot \mathbf{m}_{t-1} + \mathbf{i} \odot \mathbf{g} \\ \mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{m}_t) \end{cases}$$



At first glance, it is a little bit daunting how many elements in the above computational graph interacted. The big difference between the above computational graph of LSTM to the simple RNN is in how they used activation function. The core computational process is handled by how we used matrix **f**, **i**, **d**, and **o**. They have special names and have the following interpretation

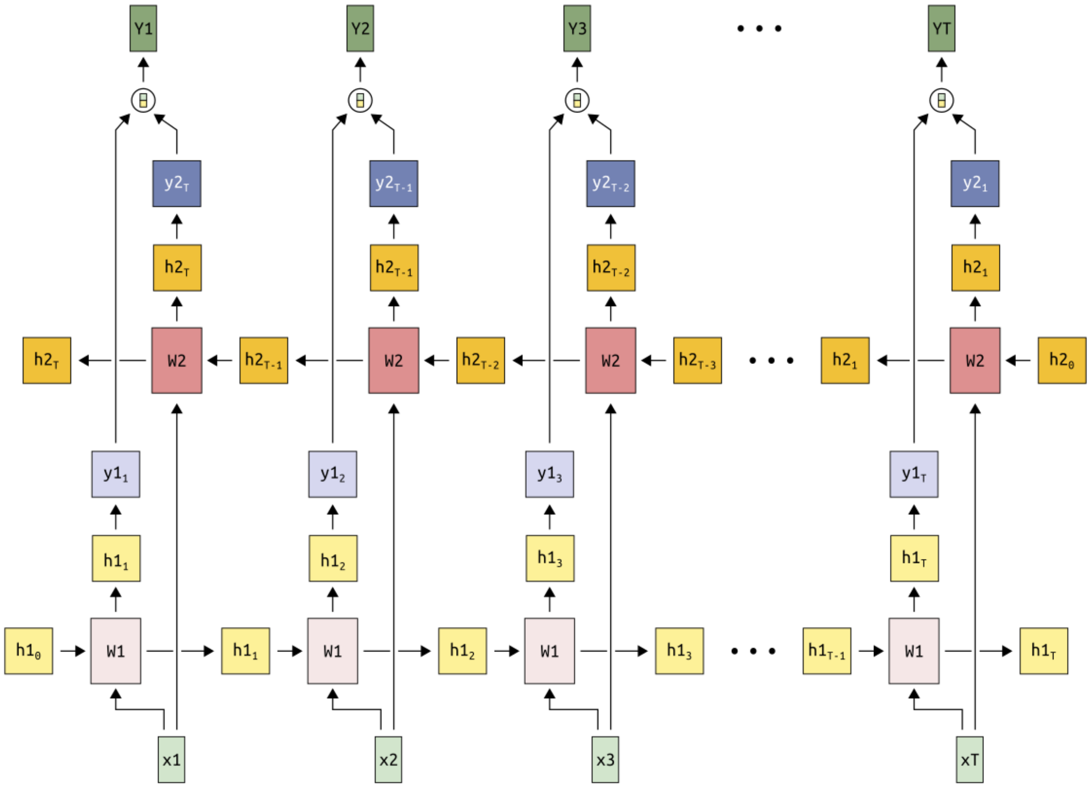
- Matrix **f** is a forget gate.
  It governs whether to erase the memory matrix or not.

- Matrix **i** is an input gate.
  it governs whether to write a hidden value to the memory matrix or not.

- Matrix **d** is a density gate.
  It governs how much the density of information to store in the memory matrix.

- Matrix **o** is an output gate.
  It governs whether to recall the memory or not.

In the next sections and later chapter we will use the fancier version of LSTM which bi-directional LSTM using the TensorFlow module. This kind of LSTM will try to learn the input sequence forward and backward. So we will have two layers of weight matrix each for forward and backward. The output of those forward and backward learning. For completeness, we show in the figure below bi-directional LSTM.



a. Detecting sarcasm in News Headlines with LSTM and CNN

First, download the dataset in the following link https://storage.googleapis.com/laurencemoroney-blog.appspot.com/sarcasm.json. The dataset has

- three keys: **`article_link`** (string), **`headline`** (string), and **`is_sarcastic`** (boolean in integer, 0 or 1).
- 26,709 items
- All headlines are in English.

Listing 4.1. headline_news_sarcasm_classifier.py

```python
import json
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences


def do_data_preprocessing(dataset_path, vocab_size, max_length,
                          padding_type, trunc_type):
    oov_tok = "<OOV>"    # out of vocabulary token
    training_size = 20_000    # in total there are 26,709 items

    with open(dataset_path, "r") as f:
        datastore = json.load(f)

    sentences = []
    labels = []

```

```python
    for item in datastore:
        sentences.append(item["headline"])
        labels.append(item["is_sarcastic"])

    training_sentences = sentences[:training_size]
    validation_sentences = sentences[training_size:]
    training_labels = labels[:training_size]
    validation_labels = labels[training_size:]

    tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
    tokenizer.fit_on_texts(training_sentences)

    word_index = tokenizer.word_index
    print(word_index)

    training_sequences = tokenizer.texts_to_sequences(training_sentences)
    training_padded = pad_sequences(training_sequences,
                                    maxlen=max_length,
                                    padding=padding_type,
                                    truncating=trunc_type)
    training_padded = np.array(training_padded)
    training_labels = np.array(training_labels)

    validation_sequences = tokenizer.texts_to_sequences(validation_sentences)
    validation_padded = pad_sequences(validation_sequences,
                                      maxlen=max_length,
                                      padding=padding_type,
                                      truncating=trunc_type)
    validation_padded = np.array(validation_padded)
    validation_labels = np.array(validation_labels)


    return training_padded, training_labels, \
           validation_padded, validation_labels, tokenizer


def create_lstm_model(vocab_size, embedding_dim, max_length):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
        tf.keras.layers.Dense(24, activation="relu"),
        tf.keras.layers.Dense(1, activation="sigmoid")
    ])

    model.compile(loss="binary_crossentropy",
                  optimizer=Adam(learning_rate=0.001),
                  metrics=["accuracy"])

    model.summary()

    return model


def plot_history(train, val, title):
    epochs = range(len(train))
    plt.figure()
    plt.plot(epochs, train, label="train")
    plt.plot(epochs, val, label="val")
    plt.title(title)
    plt.legend(loc="best")
    plt.pause(1.0)


def classify_headlines(headline_arr, model, tokenizer, max_length,
                       padding_type, trunc_type):
    for headline in headline_arr:
        test_sequence = tokenizer.texts_to_sequences(headline)
        test_padded = pad_sequences(test_sequence,
                                    maxlen=max_length,
                                    padding=padding_type,
```

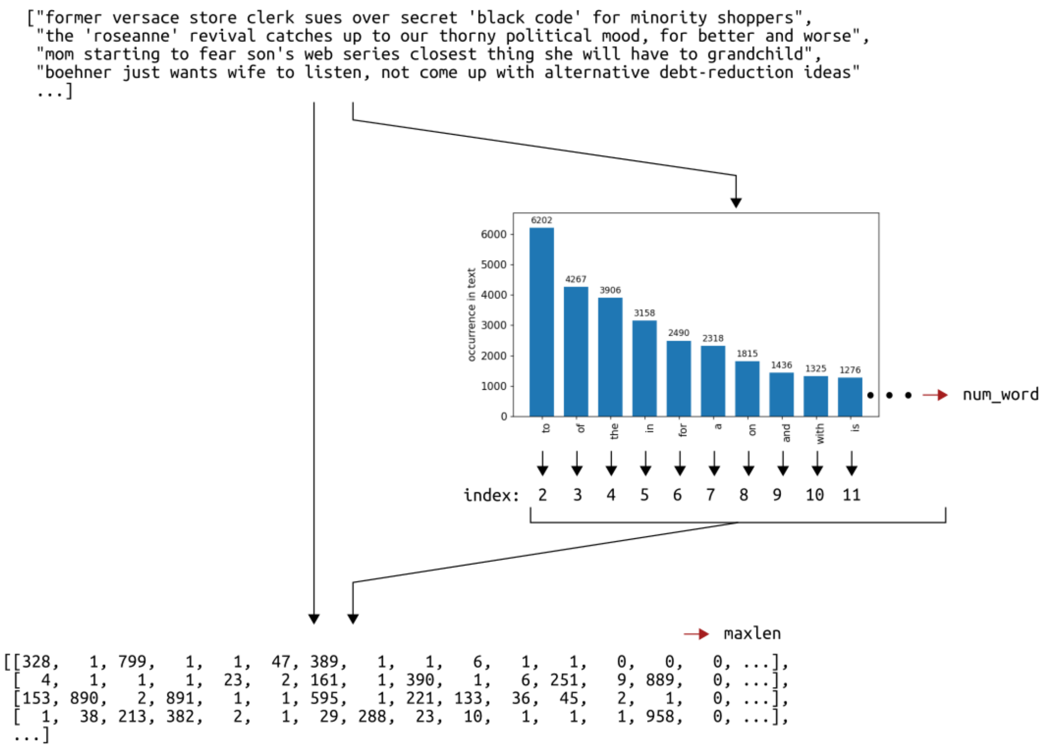
```
93                                      truncating=trunc_type)
94          test_padded = np.array(test_padded)
95  
96          classes = model.predict(test_padded)
97          print(classes[0])
98          if classes[0] > 0.5:
99              print(headline + ": a sarcasm")
100         else:
101             print(headline + ": not a sarcasm")
102 
103 
104 if __name__ == "__main__":
105     print(tf.__version__)
106 
107     dataset_path = "datasets/sarcasm.json"
108     vocab_size = 1_000
109     embedding_dim = 16
110     max_length = 120
111     trunc_type = "post"
112     padding_type = "post"
113 
114     training_padded, training_labels, validation_padded, validation_labels, \
115         tokenizer = do_data_preprocessing(dataset_path, vocab_size, max_length,
116                                           trunc_type, padding_type)
117 
118     lstm_model = create_lstm_model(vocab_size, embedding_dim, max_length)
119 
120     history = lstm_model.fit(
121         training_padded,
122         training_labels,
123         epochs=100,
124         validation_data=(validation_padded, validation_labels),
125         verbose=1
126     )
127 
128     lstm_model.save("model-saved/categ4.h5")
129 
130     acc = history.history["accuracy"]
131     val_acc = history.history["val_accuracy"]
132     loss = history.history["loss"]
133     val_loss = history.history["val_loss"]
134 
135     plot_history(acc, val_acc, "Training and validation accuracy")
136     plot_history(loss, val_loss, "Training and validation loss")
137 
138     # Test sample headlines
139     headline_arr = ["Tokyo's COVID-19 cases hit all-time high despite state of emergency",
140                     "China: No need for 2nd WHO probe on virus origin",
141                     "Turkey builds border wall to block Afghan migrants",
142                     "Line accounts of Taiwan officials hacked"]
143 
144     classify_headlines(headline_arr, lstm_model, tokenizer, max_length,
145                        padding_type, trunc_type)
```

The structure of the listing above is similar to what we have done for CNN. We start from **`do_data_processing`** in lines 12-56. We want to explain more how to represent the sentences into a sequence of integers. This kind of process is so-called *tokenizing*. Tokenizing is one of the numerous ways to encode text into a sequence of numbers.

The dataset is in JSON (JavaScript Object Notation) format, then we need to import the **`json`** module in line 1. We load the data using a standard syntax: **`with open(<path_to_file>, <flag>) as <filename>.`** Then we read each line in the **`datastore`** variable the headlines and their corresponding sarcastic status. In lines 27-30, we divide our dataset into training and validation dataset as well as their labels. The **training_sentences** (and **validation_sentences**) variable is a list of strings where each string is the headline.

Starting from line 32, we do tokenizing. To do that, we define an instance of the **Tokenizer** class with arguments **num_words** and **oov_token**. If we set **num_of_words=n**, then it dictates how many first **n-1** words, which have been sorted from the most highest occurring to the least occurring words, should be used in the method **.texts_to_sequences()** of the tokenizer object. **oov_token** serves as a string token to identify a word which is not associated with the vocabulary (out of vocabulary) that we will build using the tokenizer object.

In line 33, we build the vocabulary, the collection of all unique words, by fitting this tokenizer object to all the words in training sentences. This **.fit_on_texts()** method automatically creates a dictionary which is sorted from the most occurring words in training sentences to the least occurring words where the key is a word and the value is an index starting from 2 to the number of unique words in all sentences of the training dataset. Indices 0 and 1 are reserved for padding index and **oov_token** index [1]. Now we are ready to map each word in a sentence headline into a sequence of integers in training and validation dataset. This can be done in a vectorized way by calling the method **.texts_to_sequences** in line 38. Because the resulting sequence for each sentence does not have the same length, we need to pad and truncate such that all the sequence from all the sentences have the same length where it is set by argument **maxlen** in function **pad_sequence**. We also set the padding and truncating process at the end of the sequence. We can see that we have set this type of padding and truncating by variables **trunc_type** and **padding_type** in lines 111-112. Finally, we turn all the training sentences and labels into a NumPy array for optimal operations. We do the same thing for the validation dataset. The following figure may help to understand quickly the explanation above

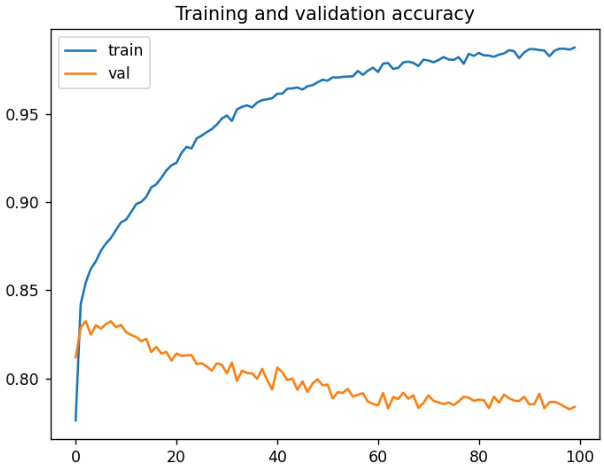


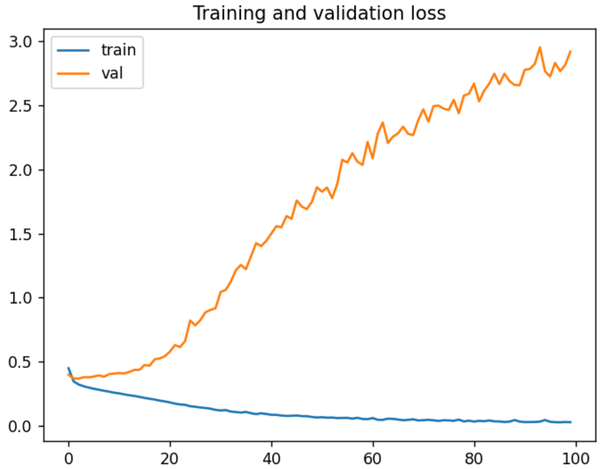
In lines 59-73, we construct a bidirectional LSTM model with the initial layer as an embedding layer. This layer serves as a converter to the training sequences of integers into smaller lengths of its

[1] https://github.com/keras-team/keras-preprocessing/blob/master/keras_preprocessing/text.py:line228

sequence. This embedding also has a training parameter which accounts for the similarity of two sentences where each sentence is represented by a vector with smaller dimension than the length of training sequences. For a complete description how this embedding layer works, see (Rong, 2016).

Running the above listing, we will achieve high accuracy and low loss for the training dataset, but unfortunately, we get a poor result in validation dataset as indicated by the following figure





b. Exploring BBC News Data

Now, we explore a different format dataset. For this case, we use the BBC News dataset. You can download the dataset, a first sample of the news, and stopwords in the following link: https://bit.ly/3fgbi0s .

Here is the specification of the BBC News Data:

- It contains two columns. The first column is the category of the news and the second column is the text containing the news. There is also a header (**`categ,text`**) in the first row.
- Total news is 2,225 where the category span from 5 topics (**`tech`**, **`entertainment`**, **`sport`**, **`politics`**, and **`business`**)

In this section, we will create a function to read this dataset such that the dataset is ready for training the model. We will use the same technique that has been described in the previous section (you can review Listing 4.1) which is using a Tokenizer class from TensorFlow. We also introduce how to use stopwords.

Listing 4.2. data_preprocessing_bbc_news.py

```
import csv

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences


def read_js_stopword(filename):
```

```
8       with open(filename, "r") as f:
9           data = f.read()
10
11      stopwords = data.split("\n")[-3]
12      stopwords = [word[1:-1] for word in stopwords[10:-3].split(", ")]
13      # print(stopwords)
14      return stopwords
15
16
17  def read_test_text(file_path):
18      with open(file_path, "r") as f:
19          test_text = f.read()
20
21      return test_text
22
23
24  if __name__ == "__main__":
25      stopwords = read_js_stopword("datasets/stopwords.js")
26
27      # Load text
28      sentences = []
29      label = []
30
31      with open("datasets/bbc-text.csv", "r") as csvfile:
32          datastore = csv.reader(csvfile)
33          datastore.__next__()   # skip the first line
34          for item in datastore:
35              text = [word for word in item[1].split() if word not in stopwords]
36              sentences.append(" ".join(text))
37              label.append(item[0])
38
39      print(f"len(sentences): {len(sentences)}")
40      test_sent = read_test_text("datasets/bbc-text-test.txt")
41
42      print(f"sentences[0] == test_sent => {sentences[0] == test_sent}")
43      print(f"labels[:10] = {labels[:10]}")
44
45      # Fit sentences to create tokens
46      sent_tokenizer = Tokenizer(oov_token="<OOV>")
47      sent_tokenizer.fit_on_texts(sentences)
48      word_index = sent_tokenizer.word_index
49
50      print(f"len(word_index): {len(word_index)}")
51
52      # Tokenizing the sentences
53      sequences = sent_tokenizer.texts_to_sequences(sentences)
54      padded = pad_sequences(sequences, padding="post")
55      print(f"padded[0]: {padded[0]}")
56      print(f"padded.shape: {padded.shape}")
57
58      # Fit Tokenizer to sentences and create a sequence of token for labels
59      label_tokenizer = Tokenizer()
60      label_tokenizer.fit_on_texts(label)
61      label_word_index = label_tokenizer.word_index
62      label_seq = label_tokenizer.texts_to_sequences(label)
63
64      print(f"label_seq[:10]: {label_seq[:10]}")
65      print(f"label_word_index: {label_word_index}")
```

In lines 7-14, we create a function **`read_js_stopword()`** to read **`stopwords.js`**. These stopwords are a list of English words: “a”, “about”, “above”, “after”, …, “yourselves”. In lines 31-37, we open the dataset **`bbc-text.csv`** and read each line then store it into **`sentences`** and **`label`** variables. As a sanity check, we verify that the resulting reading process is correct by comparing manually with the first sample of the news in line 40. We also do the same procedure like in the Listing 4.1 that we tokenize the text of news and its category in lines 46-62.

In the later section, we will classify this news into several topics using bi-directional LSTM.

c. Classifying IMDb Reviews Data

Now, we try to classify IMDb (Internet Movie Database) reviews data. This dataset is already pre-built inside TensorFlow. So we don't need to download, but we have to install the **`tensorflow_datasets`** package. You have to make sure that you have installed the correct version where it depends on the version of the other packages. Here is the short listed description of IMDb reviews data

- It contains 25,000 items of training dataset and 25,000 items of testing dataset.
- Total number of vocabulary words is around 8k. In Listing 4.3, we use **`imdb_reviews/subwords8k`**. This means that we do not separate vocabulary by a unique word as one unit word, but use a subword. We will explain later.
- There are two labels for each review which are negative review (labeled with integer = 0) and positive (labeled with integer = 1)

The classifier that we want to build is a classifier to know the sentiment for a given review whether it is a positive or negative review. We also use 4 weighted layers: embedding, bi-directional LSTM, and two fully-connected layers. The first layer is to learn the similarity between any two reviews. The second layer will learn the pattern in the sequence representation of sentences. The third and fourth layers are the nonlinear classifier function to map from a high dimensional input vector into a single output of number.

Listing 4.3. imdb_reviews_subwords8k_classifier.py

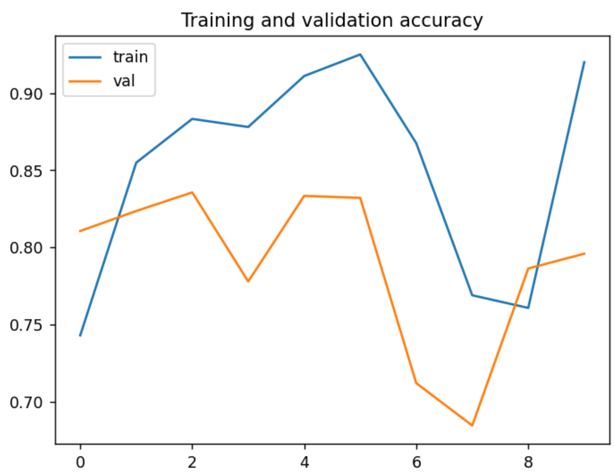
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow_datasets as tfds
import tensorflow as tf

from tensorflow.keras.optimizers import Adam


def do_data_preprocessing(train_dataset, validation_dataset, info,
                          BUFFER_SIZE, BATCH_SIZE):

   tokenizer = info.features["text"].encoder

   train_dataset = train_dataset.shuffle(BUFFER_SIZE)
   train_dataset_padded = train_dataset.padded_batch(
       BATCH_SIZE,
       tf.compat.v1.data.get_output_shapes(train_dataset)
   )
   validation_dataset_padded = validation_dataset.padded_batch(
       BATCH_SIZE,
       tf.compat.v1.data.get_output_shapes(validation_dataset)
   )

   return train_dataset_padded, validation_dataset_padded, tokenizer


def create_lstm_model(tokenizer):
   model = tf.keras.models.Sequential([
       tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
       tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
       tf.keras.layers.Dense(64, activation="relu"),
       tf.keras.layers.Dense(1, activation="sigmoid")
   ])
```

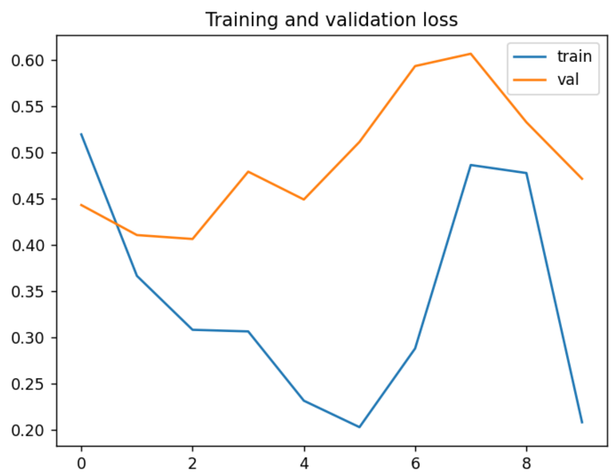
```python

    model.summary()

    model.compile(
        loss="binary_crossentropy",
        optimizer=Adam(learning_rate=0.001),
        metrics=["accuracy"]
    )

    return model


def plot_history(train, val, title):
    epochs = range(len(train))
    plt.figure()
    plt.plot(epochs, train, label="train")
    plt.plot(epochs, val, label="val")
    plt.title(title)
    plt.legend(loc="best")
    plt.pause(1.0)


def classify_reviews(test_reviews_path, model):
    with open(test_reviews_path, "r") as f:
        sample_reviews = f.read()

    sample_reviews = sample_reviews.split("\n")

    for review in sample_reviews:
        review_padded = tokenizer.encode(review)
        review_padded = tf.data.Dataset.from_tensors(review_padded)
        review_padded = review_padded.padded_batch(1)

        classes = model.predict(review_padded)
        print(f"classes[0]: {classes[0]}")
        if classes[0] > 0.5:
            print(f"[positive]\n{review[:100]}...")
        else:
            print(f"[negative]\n{review[:100]}...")


if __name__ == "__main__":
    print(tf.__version__)

    # Get the data
    dataset, info = tfds.load("imdb_reviews/subwords8k", with_info=True, as_supervised=True)
    train_dataset, validation_dataset = dataset["train"], dataset["test"]

    BUFFER_SIZE = 10_000
    BATCH_SIZE = 64

    # Pre-process the dataset
    train_dataset_padded, validation_dataset_padded, tokenizer\
        = do_data_preprocessing(train_dataset, validation_dataset,
                                info, BUFFER_SIZE, BATCH_SIZE)

    # Build a LSTM model
    lstm_model = create_lstm_model(tokenizer)

    # Train the model to the training dataset
    history = lstm_model.fit(
        train_dataset_padded,
        epochs=10,
        validation_data=validation_dataset_padded
    )

    acc = history.history["accuracy"]
    val_acc = history.history["val_accuracy"]
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]
```

```
104
105     plot_history(acc, val_acc, "Training and validation accuracy")
106     plot_history(loss, val_loss, "Training and validation loss")
107
108     # Test sample reviews
109     classify_reviews("datasets/imdb-sample-test.txt", lstm_model)
```

In lines 9-24, we define a function for doing data preprocessing. First, we get a **`SubwordTextEncoder`** class in line 12. This will be helpful to encode a text in the later section of the code. In line 14, we shuffle our training dataset to make sure all the positive and negative reviews are distributed uniformly while training the dataset. To do that we use the **`.shuffle()`** method. This shuffling uses a buffer for each sampled review that we take. And replaced the taken review from the buffer by the next (**`BUFFER_SIZE+1`**)-th review from the training dataset.
In lines 15-17, we use the **`.padded_batch`** method to give a zero padded integers for each batch in the dataset. The length for each sequence in each batch will be determined by the longest sequence in each batch. For the case of listing above, in total there would be **`np.ceil(25,000/BATCH_SIZE)=391`** batches. We do the same procedure of padding the sequence of validation dataset in lines 19-22.

For the rest of the lines, we do again the similar procedures: build model, train model, plot the accuracy and loss, then finally test the model to the sample reviews. Running the listing above we will obtain the following figure





Because of the huge amount of dataset in IMDb review data, the LSTM model tends to have a long computational time for the training process. We only set small epochs and one layer bi-directional LSTM. The reader is suggested to try with different hyperparameters such as number of filters, learning rate, adding another LSTM layer, and adding a pooling layer.

d. Classifying BBC News into topics

In this section, we extend Listing 4.2 by adding a classifier. This classifier consists of:

- an embedding layer that turns a sequence with **`max_length=500`** into a vector with length **`embedding_dim=8`**.
- a bi-directional LSTM layer with 4 channels (filters or depths).
- a drop-out layer to capture ensemble models and avoid overfitting.
- two-fully connected layers with activation functions: ReLU and softmax.

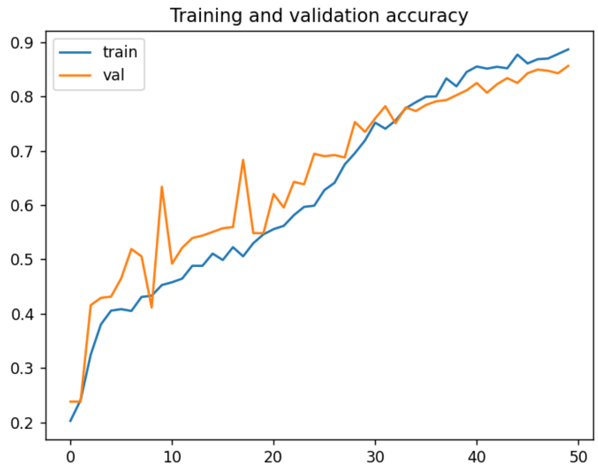
Listing 4.3. bbc_news_classifier.py

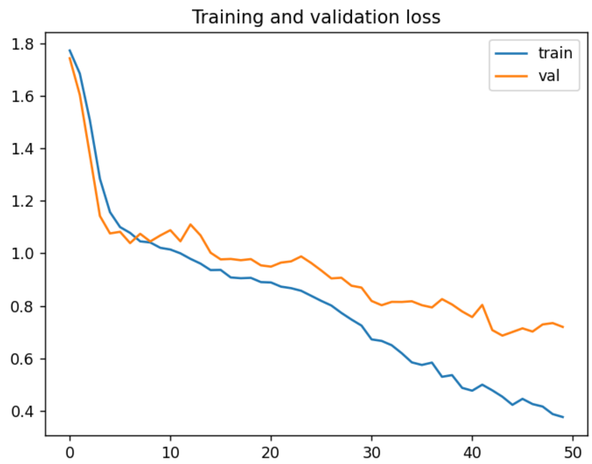
```
import csv
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences


def read_js_stopword(filename):
   with open(filename, "r") as f:
       data = f.read()

   stopwords = data.split("\n")[-3]
   stopwords = [word[1:-1] for word in stopwords[10:-3].split(", ")]
   return stopwords


def read_test_text(file_path):
   with open(file_path, "r") as f:
       test_text = f.read()

   return test_text


def read_sample_text(file_path):
   with open(file_path, "r") as f:
       data = f.readlines()

   test_text = []
   test_label = []
   for x in data:
       idx = x.find("];")
       test_text.append(x[idx+2:])
       test_label.append(x[1:idx])

   return test_text, test_label


def do_data_preprocessing(sentences, label, training_portion, vocab_size,
                          oov_tok, padding_type, trunc_type, max_length):
   verify_feature = read_test_text("datasets/bbc-text-test.txt")
   print(f"sentences[0] == verify_feature => {sentences[0] == verify_feature}")

   train_size = int(len(sentences) * training_portion)
   train_sentences = sentences[:train_size]
   train_labels = label[:train_size]

   validation_sentences = sentences[train_size:]
   validation_labels = label[train_size:]

   # Fit sentences to create tokens
   tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
   tokenizer.fit_on_texts(train_sentences)
   word_index = tokenizer.word_index
   print(f"len(word_index): {len(word_index)}")

   # Tokenizing the sentences
   train_sequences = tokenizer.texts_to_sequences(train_sentences)
   train_padded = pad_sequences(train_sequences,
                                padding=padding_type,
                                truncating=trunc_type,
                                maxlen=max_length)

   validation_sequences = tokenizer.texts_to_sequences(validation_sentences)
   validation_padded = pad_sequences(validation_sequences,
```

```python
                                     padding=padding_type,
                                     truncating=trunc_type,
                                     maxlen=max_length)

    # Fit Tokenizer to sentences and create a sequence of token for labels
    label_tokenizer = Tokenizer()
    label_tokenizer.fit_on_texts(label)

    training_label_seq = np.array(label_tokenizer.texts_to_sequences(train_labels))
    validation_label_seq = np.array(label_tokenizer.texts_to_sequences(validation_labels))

    return train_padded, training_label_seq, \
        validation_padded, validation_label_seq, \
        tokenizer, label_tokenizer


def create_language_model(vocab_size, embedding_dim, max_length):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(4)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(32, activation="relu"),
        tf.keras.layers.Dense(6, activation="softmax")  # zero index is for padding
    ])

    model.compile(loss="sparse_categorical_crossentropy",
                  optimizer=Adam(learning_rate=0.001),
                  metrics=["accuracy"])

    model.summary()

    return model


def plot_history(train, val, title):
    epochs = range(len(train))
    plt.figure()
    plt.plot(epochs, train, label="train")
    plt.plot(epochs, val, label="val")
    plt.title(title)
    plt.legend(loc="best")
    plt.pause(1.0)


def classify_news(news_arr, true_label_arr, model, tokenizer, label_tokenizer, max_length,
                  padding_type, trunc_type):

    reverse_label_index = dict([(v, k) for (k, v)
                                in label_tokenizer.word_index.items()])
    print(f"reverse_label_index: {reverse_label_index}")

    for i, news in enumerate(news_arr):
        test_sequence = tokenizer.texts_to_sequences([news])
        test_padded = pad_sequences(test_sequence, padding=padding_type,
                                    maxlen=max_length, truncating=trunc_type)
        test_padded_numpy = np.array(test_padded)

        classes = model.predict(test_padded_numpy)
        get_class_key = np.argmax(classes[0])
        class_label = reverse_label_index[get_class_key]
        print(f"{news[:200]}: {class_label} [true label: {true_label_arr[i]}]")


if __name__ == "__main__":
    vocab_size = 10_000
    embedding_dim = 8
    max_length = 500
    trunc_type = "post"
    padding_type = "post"
    oov_tok = "<OOV>"
```

```
138    training_portion = 0.8
139
140    stopwords = read_js_stopword("datasets/stopwords.js")
141
142    # Load text
143    sentences = []
144    label = []
145    with open("datasets/bbc-text.csv", "r") as csvfile:
146        datastore = csv.reader(csvfile)
147        datastore.__next__()  # skip the first line
148        for row in datastore:
149            label.append(row[0])
150            sentence = row[1]
151            for word in stopwords:
152                token = " " + word + " "
153                sentence = sentence.replace(token, " ")
154
155            sentences.append(sentence)
156
157    # Data pre-processing
158    train_padded, training_label_seq, \
159        validation_padded, validation_label_seq, \
160        tokenizer, label_tokenizer \
161        = do_data_preprocessing(sentences, label, training_portion, vocab_size,
162                                oov_tok, padding_type, trunc_type, max_length)
163
164    # Build model
165    lstm_model = create_language_model(vocab_size, embedding_dim, max_length)
166
167    history = lstm_model.fit(
168        train_padded,
169        training_label_seq,
170        epochs=50,
171        validation_data=(validation_padded, validation_label_seq),
172        verbose=1
173    )
174
175    acc = history.history["accuracy"]
176    val_acc = history.history["val_accuracy"]
177    loss = history.history["loss"]
178    val_loss = history.history["val_loss"]
179
180    plot_history(acc, val_acc, "Training and validation accuracy")
181    plot_history(loss, val_loss, "Training and validation loss")
182
183    # Test sample headlines
184    news_arr, true_label_arr = read_sample_text("datasets/bbc-text-samples.txt")
185    print(f"len(sample_sentences): {len(news_arr)}")
186    classify_news(news_arr, true_label_arr, lstm_model, tokenizer, label_tokenizer,
187                  max_length, padding_type, trunc_type)
```

Running the above program, we will achieve a good classifier as indicated by the figure below. In lines 186, we test this model to 15 news which are 3 news for each category. Sometimes it could classify correctly, sometimes it does not. For this short introduction to NLP, we do not pursue the further setting on how to improve this classifier to be a perfect classifier for the BBC News dataset. In TensorFlow Developer Certification, we only need to achieve highest accuracy in training and/or validation dataset.





e. Poem Generation with Bi-Directional LSTM

Now, we are in the last section of this chapter. This section is the most fun application of the machine learning model to text generation, especially to create a poem which can fool lay people and obviously show the simple example of the Turing test. To do that we will use 154 Shakespeare's sonnets which are concatenated to each other to build a dataset containing 2,158 rows (154 sonnets x 14 rows/sonnets + 2 rows).

The machine learning model that we employ incorporates several layers which are:

- an embedding layer with an output vector is 100 in length.
- a bi-directional LSTM layer with 150 channels.
- a dropout layer where the dropout ratio is 0.2.
- another LSTM layer with 100 channels.
- two fully-connected layers with the former layer using kernel regularizer L2 norm.

We train the model for each line in Shakespeare's sonnets. From that we will give a feed of sentences then try to generate n words that will come from the feed of the input. You can download the dataset in this link: https://storage.googleapis.com/laurencemoroney-blog.appspot.com/sonnets.txt. We also provide poems by one of the greatest Indonesian poets, Chairil Anwar. You can download his poem in this link: https://bit.ly/37h90tJ.

Listing 4.5. shakespeare-poem-generator.py

```
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
import tensorflow.keras.utils as keras_utils

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras import regularizers


def do_data_preprocessing(data_path):
   with open(data_path, "r") as f:
       data = f.read()
```

```python

    corpus = data.lower().split("\n")

    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(corpus)
    total_words = len(tokenizer.word_index) + 1

    # create input sequence using list of tokens
    input_sequences = []
    for line in corpus:
        token_list = tokenizer.texts_to_sequences([line])[0]
        for i in range(1, len(token_list)):
            n_gram_sequence = token_list[:i+1]
            input_sequences.append(n_gram_sequence)

    # pad sequences
    max_sequence_len = max([len(x) for x in input_sequences])
    input_sequences = np.array(pad_sequences(input_sequences,
                                             maxlen=max_sequence_len,
                                             padding="pre"))

    # create predictors and label
    predictors, labels = input_sequences[:, :-1], input_sequences[:, -1]

    label_categ = keras_utils.to_categorical(labels, num_classes=total_words)

    return predictors, label_categ, total_words, max_sequence_len, tokenizer


def create_language_model(total_words, ):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(total_words, 100, input_length=max_sequence_len-1),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(150, return_sequences=True)),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.LSTM(100),
        tf.keras.layers.Dense(total_words/2, activation="relu",
                              kernel_regularizer=regularizers.l2(0.01)),
        tf.keras.layers.Dense(total_words, activation="softmax")
    ])

    model.compile(loss="categorical_crossentropy",
                  optimizer=Adam(learning_rate=0.001),
                  metrics=["accuracy"])

    model.summary()

    return model


def plot_history(train, title):
    epochs = range(len(train))
    plt.figure()
    plt.plot(epochs, train)
    plt.title(title)
    plt.pause(1.0)


def generate_text(sample_text, next_words, tokenizer, max_sequence_len, model):
    for _ in range(next_words):
        token_list = tokenizer.texts_to_sequences([sample_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_sequence_len-1, padding="pre")
        predicted = np.argmax(model.predict(token_list, verbose=1), axis=-1)
        output_word = ""
        for word, index in tokenizer.word_index.items():
            if index == predicted:
                output_word = word
                break
        sample_text += " " + output_word
    print(sample_text)

```

```python

if __name__ == "__main__":

    # data_path = "datasets/chairil-anwar-poems.txt"
    data_path = "datasets/sonnets.txt"
    predictors, label_categ, total_words, \
        max_sequence_len, tokenizer = do_data_preprocessing(data_path)

    # Build a language model
    lstm_model = create_language_model(total_words)

    history = lstm_model.fit(predictors, label_categ, epochs=100, verbose=1)

    acc = history.history["accuracy"]
    loss = history.history["loss"]

    plot_history(acc, "Training accuracy")
    plot_history(loss, "Training loss")

    sample_text = "Help me Obi Wan Kenobi, you're my only hope"
    # sample_text = "saat kenyataan hanyalah bayangan dan semu"  # indonesian words
    next_words = 100

    generate_text(sample_text, next_words, tokenizer, max_sequence_len, lstm_model)
```

5. Sequence, Time Series and Prediction

For this chapter, we only focus on a single dataset which is a sunspot dataset. This dataset records . . .
The prediction has a great application to spot the sun's behaviour such that the dangerous magnetic storm can be avoided and save many electrical installations around the world. There is a stunning video explaining the importance of this prediction in Kurzgesagt channel: https://www.youtube.com/watch?v=oHHSSJDJ4oo.

Another far-fetched application of this chapter is to predict the behaviour of the market using machine learning. This has been practised by numerous quants. If you want to be rich in a clever way and in disguise, you should go deeper to understand these techniques and apply to any data that has relation to the market movement. At this moment, we do not pursue and discuss to this application.

a. Create and predict synthetic data with time series

b. Prepare features and labels

c. Predict synthetic data with Linear Regression

d. Predict synthetic data with MLP

e. Finding an optimal learning rate for a RNN

f. LSTM

6. Preparation for Taking Exam

a. Preparation

b. Taking the exam

Useful troubleshooting and Python's commands

It is recommended to use Python 3.8

Required Python package

- numpy
- tensorflow
- tensorflow-gpu
- matplotlib
- scipy

----------------------------------//----------------------------------

Adding the following commands after importing TensorFlow packages if you want to use GPU instead of CPU

```
physical_devices = tf.config.list_physical_devices("GPU")
tf.config.experimental.set_memory_growth(physical_devices[0], enable=True)
```

----------------------------------//----------------------------------

Always check the memory usage of your NVidia card through the command if you use TensorFlow with GPU

```
$ nvidia-smi -l 1
```

----------------------------------//----------------------------------

If you use PyCharm in Linux, please install the stable version that has been released in the beginning of the year when you installed it. The most recent version sometimes will not be compatible with the TensorFlow version. Use the following command

```
$ sudo snap install pycharm-community -channel=2021.1/stable --classic.
```

----------------------------------//----------------------------------

If you have the following error message while running LSTM model:

```
NotImplementedError: Cannot convert a symbolic Tensor
(bidirectional_1/forward_lstm_1/strided_slice:0) to a numpy array. This
error may indicate that you're trying to pass a Tensor to a NumPy call,
which is not supported
```

Please downgrade the numpy into version 1.19.5.

----------------------------------//----------------------------------

Determine which backend is being used by **`matplotlib`**

```
>>> import matplotlib
>>> matplotlib.get_backend()
```

If the result is "**agg**", this backend does not support for the command **`plt.show()`**

Install matplotlib backend tkinter
```
$ sudo apt install python3-tk
```

Use tkinter backend in matplotlib
```
>>> import matplotlib
>>> matplotlib.use(“TkAgg”)
```

Qt5Agg backend can be installed through Python’s package: PyQt5

To be able non-blocking figure plotting, we have to use **`plt.pause()`** instead of **`plt.show`**
```
plt.plot(x_arr, y_arr)
plt.pause(1)
```

References

(Cord, 2016) - *Deep CNN and Weak Supervision Learning for visual recognition*.

(Moroney, et.al., 2020) - *Coursera: DeepLearning.AI TensorFlow Developer.*

(Kingma and Ba, 2015) - *Adam: A Method for Stochastic Optimization*.

(Simonyan and Zisserman, 2014) - *Very Deep Convolutional Networks for Large-Scale Image Recognition.*

(Rong, 2016) - *word2vec Parameter Learning Explained*