Keras

In this module we will introduce [Keras (https://keras.io/)](https://keras.io/), a high level API for Neural Networks.

To be specific

- we will mostly restrict ourselves to the Keras Sequential model
- this will greatly simplify your learning and coding
- it will restrict the type of Deep Learning programs that you can write
    - but not a meaningful restriction for the simple programs that you will write in this course

Note:

The code snippets in this notebook are *fragments* of a larger notebook (DNN_Keras_example.ipynb)

- are illustrative: will not actually execute in this notebook but will in the complete notebook

The Keras Sequential Model

Reference: [Getting started with the Keras Sequential Model (https://keras.io/getting-started/sequential-model-guide/)](https://keras.io/getting-started/sequential-model-guide/)

Keras has two programming models

- Sequential
- Functional

We will start with the Sequential model

The Sequential model allows you to build Neural Networks (NN) that are composed of a *sequence* of layers

- just like our cartoon
- a very prevalent paradigm

This will likely be sufficient in your initial studies

- but it restricts the architecture of the Neural Networks that you can build
- use the Functional API for full generality
    - but it might appear more complicated

The idea is quite simple.

Keras Sequential implements an `sklearn` -like API

- define a model
- fit the model
- predict

Defining a model

Let's jump into some code.

We start with some preliminaries

- imports
- determining size of an example

```
import keras

from keras.models import Sequential
from keras.layers import Input, Dense

input_size = X[0].shape
output_size = np.unique(y).shape[0]
```

Next: some old friends, in new clothing

```
# Regression
model = Sequential([
    Input(shape=(input_size,)),
    Dense(1, activation=None)
    )

model.compile(loss='mse')
```

- A model uses the `Sequential` architecture
- A sequence (implemented as an array) of layers
    - `Input` layer
        - defines the shape of a single example
    - `Dense` (Fully connected) layer
        - with 1 output
        - No activation
        - Implements Regression
- Loss is `mse`

```
# Binary Classification
model = Sequential([
    Input(shape=(input_size,)),
    Dense(1, activation='sigmoid')
    )

model.compile(loss='binary_crossentropy')
```

- A model uses the `Sequential` architecture
- A sequence (implemented as an array) of layers
    - `Input` layer
    - `Dense` (Fully connected) layer
        - with 1 output: binary classification
        - sigmoid activation
        - Implements Classification
- Loss is `binary_crossentropy`

TL;DR

- Both examples are single non-Input layer
    - Dense, with 1 unit ("neuron")
- Regression example
    - No activation
    - MSE loss
- Binary classification example
    - Sigmoid activation
    - Binary cross entropy loss

Hopefully you get the idea.

Let's explore a slightly more complicated model.

```python
# Multinomial Classification
model = Sequential([
    Input(shape=(input_size,)),
    Dense(n_hidden_1, activation='relu', name="hidden_1"),
    Dense(n_hidden_2, activation='relu', name="hidden_2"),
    Dense(output_size, activation='softmax', name="outputs")
])

model.compile(loss='sparse_categorical_crossentropy')
```

- A model uses the `Sequential` architecture
- A sequence (implemented as an array) of layers
    - `Input` layer
    - 2 `Dense` layers
        - with varying number of outputs: `n_hidden_1`,`n_hidden_2`
        - `relu` activation
    - A `Dense` layer implementing Multinomial Classification
        - number of outputs equal to number of classes: `output_size`
        - `softmax` activation

The above example illustrates a common architecture

- a final *head* layer, specific to the task type
    - regression
    - classification
- pre-head layers
    - transform raw features
    - into synthetic features
    - that are best suited for the head layer

Compiling a model

A primary purpose of the `compile` statement

- associating a Loss Function with the model

As we will see later, we can also associate

- an optimizer to use in fitting
- metrics to report during training

Fitting a model

Next, just as in `sklearn`: you "fit" the model to the training data.

```
history = model.fit(X_train, y_train,
                    epochs=5,
                    batch_size=128,
                    shuffle=True,
                    validation_split=0.1)
```

Prediction

The fitted model can be used to make predictions.

```
# Evaluation
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print(f'\nTest accuracy: {test_acc:.4f}')
```

The Keras Functional Model

- More verbose than `Sequential`
- Also more flexible
    - you can define more complex computation graphs (multiple inputs/outputs, shared layers)

```python
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(32, activation='relu')(inputs)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and  Dense layers
model = Model(inputs=inputs, outputs=predictions)
```

Highlights:

- Manually invoke a single layer at a time
    - Passing as input the output of the prior layer.
- You must define an `Input` layer (placeholder for the input/define its shape)
    - `Sequential` uses the `input_shape=` parameter to the first layer
- You "wrap" the graph into a "model" by a `Model` statement
    - looks like a function definition
        - names the input and output formal parameters
    - a `Model` acts just like a layer (but with internals that you create)

As a beginner, you will probably exclusively use the Sequential model.

Keep the Functional API in the back of your mind.

Let’s code

Let's see some actual code (DNN_Keras_example.ipynb)

Some programming details

Keras: backend agnostic

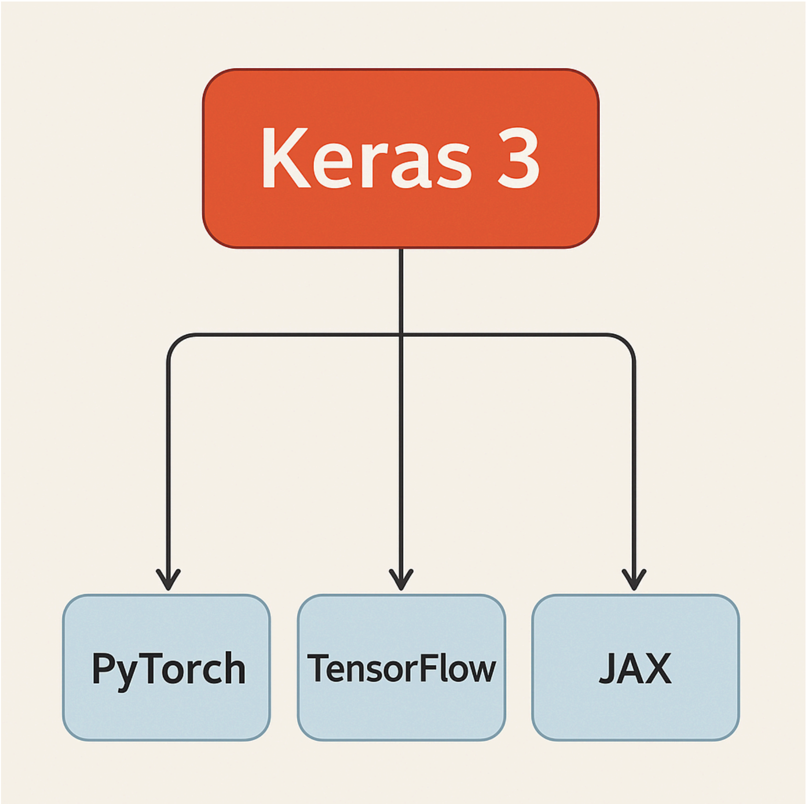
Keras is one-level higher than the "backend" APIs for Deep Learning

- TensorFlow, PyTorch, JAX

If our code uses *only* methods within the Keras module

- with no uses of backend-specific functions
- the code can run on multiple backends
    - TensorFlow, PyTorch

Keras and the backends



In order to achieve this *backend agnostic* coding

- we must avoid explicit direct calls to the other Deep Learning toolkits
- DO NOT import the backend

```
import tensorflow as tf

import torch
```

- DO NOT use functions (e.g., `func`) in the backend namespace

```
tf.func
tensorflow.func

torch.func
```

- DO use [equivalent functions (https://keras.io/guides/migrating_to_keras_3/#transitioning-to-backendagnostic-keras-3)](https://keras.io/guides/migrating_to_keras_3/#transitioning-to-backendagnostic-keras-3) implemented in Keras

```
keras.func
```

We specify which backend to use via an environment variable that is set before importing Keras

```
import os
os.environ["KERAS_BACKEND"] =  "tensorflow" # torch

import keras
```

We will *try very hard* to use only backend agnostic code

- However: some older notebooks may not have been fully converted to be backend -agnostic

Input layer specification: explicit versus implicit

In a `Sequential` model: the use of an *explicit* `Input` layer is optional

- if present: it specifies the shape of an example, e.g. the tuple `INPUT_SHAPE`

  ```
  Input(shape=INPUT_SHAPE)
  ```

- if absent: there are two *implicit* ways to for the model `model` to obtain the shape of an example
  - infer it from the first example (in the first batch) presented to the model
  - via a `build` method that passes in the example shape *with an extra leading element (with value `None`)*

    ```
    model.build(input_shape=(None,)+INPUT_SHAPE)
    ```

For example, suppose our examples are MNIST images in gray scale (one channel)

```
INPUT_SHAPE=(28,28,1)
```

The explicit `Input` layer would be

```
Input(shape=(28,28,1))
```

Using build on the object `model`

```
model.build(input_shape=(None,28,28,1)
```

Note the leading batch dimension with value `None` in the tuple passed to `build`

What is the import of knowing the shape of an example ?

- it determines the shape of the weights for the first non-Input layer !
- allowing us to allocate memory and initialize the weights

For example

- if the first non-`Input` layer is `Dense(10)`
- and shape of an example is `(784,)`
- then the `Dense(10)` layer
    - has 784*10 weights (+ 10 bias weights)

That is:

- the number of weights for a layer is a function of
    - the shape of the layer input
    - the shape of the layer output

The first non- `Input` layer is the only layer for which we don't know the shape of the layer input.

- For all other layers: the shape of the input is the shape of the output of the preceding layer

The advantage of the implicit methods of specifying the model input shape

- it is dynamic
- we can change the image dimension from `(28,28,1)` to `(100,100,1)`
- without any change in the model code

Keras implementations

Confusion warning:

There are two similar *but different* packages that implement Keras

- the one from the Keras project, imported/used as

  ```
  import keras

  # Use a Dense layer
  keras.layers.Dense(...)
  ```

- one built into TensorFlow, imported/used as

  ```
  from tensorflow import keras
  ```

  or via the `tf.keras` namespace

  ```
  import tensorflow as tf

  # Use a Dense layer
  tf.keras.layers.Dense(...)
  ```

We will be using the first, so always

```
import keras
```

This will be Keras 3, from the Keras Project.

The advantage is that Keras 3 works **unchanged** across several Deep Learning frameworks

- TensorFlow
- PyTorch
- JAX

Keras (under the covers) will use code for the chosen framework (called the *back-end*).

It is highly desirable to write code using *only* the Keras API

- *back-end agnostic*
- *not* directly to the a particular framework's back-end

We will try to do as much as possible

- however
    - older notebooks used Keras 2 (and the `tf.keras` namespace)
    - some residual "old code" may still remain

Keras 2 (old version)

There is a lot of code written in Keras 2

- including some of the older notebooks in this repo

FYI, here is information on using older code

- [Keras 2 backward compatibility (https://keras.io/getting_started/#tensorflow--keras-2-backwards-compatibility)](https://keras.io/getting_started/#tensorflow--keras-2-backwards-compatibility)

The key points:

> *> Meanwhile, the legacy Keras 2 package is still being released regularly and is available on PyPI as tfkeras (or equivalently tf-keras – note that - and are equivalent in PyPI package names). To use it, you can install it via pip install tf_keras then import it via import tf_keras as keras. > Should you want tf.keras to stay on Keras 2 after upgrading to TensorFlow 2.16+, you can configure your TensorFlow installation so that tf.keras points to tf_keras. To achieve this: > Make sure to install tf_keras. Note that TensorFlow does not install it by default. Export the environment variable TF_USE_LEGACY_KERAS=1.*

In [1]:
```
print("Done")
```

```
Done
```