Introduction to Object Oriented programming and the Tensorflow/Keras OO framework

Albert Montillo

UTSouthwestern

Departments: Lyda Hill Department of Bioinformatics, Radiology, and Advanced Imaging Research Center

UTSW SWE course

Lyda Hill Department of Bioinformatics

Outline

1. **Introduction to Object Oriented Programing in python**
2. **Introduction to the Tensorflow/Keras OO framework**

Programming paradigms

- **Functional programming**
  - Every statement is a function and returns a value.
  - Ex: Lisp
  - While elegant, does not reflect how most think

- **Procedural programming**
  - Some program statements are grouped into procedures that act on data and may or may not return a value
  - Other statements govern flow of control and may invoke the procedures
  - Ex: C, Perl, Pascal, Fortran77
  - Reflects how most think. Allows for ready code reuse (promotes refactoring)

- **Object oriented programming (OOP)**
  - Takes refactoring one logical step further
  - All of procedural programming :
    - Allows to define not only statements that belong together (procedures/functions)
  - Also allows to define data and functions that belong together, (an object)
  - Reflects the real world. Employee has data attributes (name, age) and methods (functions) they can perform (changeDepartment(), changeSalary())
  - Further promotes code reuse: promotes refactoring classes of objects and functions
  - Ex: Python, C++, C#

Concepts and terminology of OOP

- **Object**
  - **A logical grouping of state (attribute variables) and methods (functions) that act on that state**
- **Encapsulation**
  - **Logical grouping of state and methods. Placing restrictions on what attributes and methods can be accessed directly. Python objects use encapsulation.**
- **Class**
  - **Blueprint for creating an object at run time (e.g., in memory)**
- **Inheritance**
  - **one class inherits properties (attributes, methods) from another class.**
  - **fosters code reuse**
- **Class relationships**
  - **is-a : one class is a subclass of another base class**
  - **has-a : one class contains another class as an attribute**

Object

- An entity that has state and behavior associated with it.
- Used to keep track of real-world objects
  - Person, animal, medical instrument, chair
- Used to keep track of abstract objects
  - Integer: 12
  - String: “Hello”
  - floating-point number: 3.141
- You have been using them all along

- An object consists of
  - State ..
    - The current properties of the object
    - Contained in variables called attributes
  - Behavior
    - The actions you can do to an object
    - Contained in the functions called methods that you define for the object
  - Identity
    - An object is a unique instance of an object category
    - Ex: you are a unique person
    - 12 is a unique integer

Encapsulation

- **Logical grouping of state and behavior**

- **It makes sense to sqrt(number) but not a string or person.**
- **A person has a passport ID number but a string does not.**
- **In OOP, an object uses encapsulation**

- **Local grouping of state and behavior**
  - **Typically, we place all code that defines state and behavior in the same file and same section of code within that file**

- **Why encapsulation?**
  - **Designing code with encapsulation, facilitates code maintainability**
  - **The number one expense in any SW development project for most/all computational scientific projects are scientific SW developers.**
    - **The greatest expense is maintenance of software (code)**

  - **By localizing code through OOP we make it easier to**
    - **find and fix bugs**
    - **Extend our work through code reuse**

Class

- Blueprint for creating an object where designer makes clear
  - Attributes that will store an object's state
  - Methods that will define an object's behavior
- Puts you in the driver's seat.
- Allows you to create your own data types.
- You are not stuck with integers, strings, floats any longer.

- Many instances of a class can be created from the same single class.
- These instances are called objects.

- The process of creating an instance is called instantiation.
- When you instantiate an object from a class (blueprint),
  - Memory is allocated to store the state of the new object
  - The constructor method is called to initialize the class.
    - Called __init__() .. Pronounced "dunder" init "dunder"
    - Sets the initial state of the new object by setting its attribute variables
    - Returns the new object

- Object is destroyed (memory freed) when the program ends (implicit garbage collection by python) or if you call delete the new object via "del"
  - Typically, you need not call del, python will do it for you.

Anatomy of a python Class

```
class Person(object):
    #constructor
    def __init__(self, name, idnumber):
            self.name= name
            self.idnumber = idnumber
            print(f"inside the Person constructor")

    def display(self):
            print(f"\n")
            print(f"name {self.name}")
            print(f"idnumber {self.idnumber}")
            print(f"------\n")
```

- First line:
  - Keyword class used to define a new class
  - Object is root baseclass
- Body of the class is indented after the colon :
- In the body we define
  - methods (behavior)
  - attributes (variables that hold state)
- __init__() is the constructor method
  - Defines the instance's state variables
- Other methods define class specific behavior

Example1: Class and Objects

In [12]:
```
class Person(object):
    #constructor
    def __init__(self, name, idnumber):
            self.name= name
            self.idnumber = idnumber
            print(f"inside the Person constructor")

    def display(self):
            print(f"\n")
            print(f"name {self.name}")
            print(f"idnumber {self.idnumber}")
            print(f"------\n")
```

In [13]:
```
# create a person object
person1=Person('Rahul', 8543)
person2=Person('Bob', 9923)

# print the person
person1.display()
person2.display()
```

```
inside the Person constructor
inside the Person constructor


name Rahul
idnumber 8543
------



name Bob
idnumber 9923
------
```

Example2: Class and Objects

```
In [15]: # accessing attributes of an object
         print(f"person1 name is {person1.name}")
         print(f"person2 name is {person2.name}")

         person1 name is Rahul
         person2 name is Bob

In [16]: # testing equality of objects
         person1==person2

Out[16]: False

In [17]: person3=person1

In [18]: person1==person3

Out[18]: True

In [19]: person1.run()

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         Input In [19], in <cell line: 1>()
         ----> 1 person1.run()

         AttributeError: 'Person' object has no attribute 'run'
```

Inheritance

- When one class is subclassed (derived) from another it inherits the properties (attributes and methods) of the base class.
- The class that inherits properties is called the subclass (derived, child class)
- The class from which properties are inherited is called the base class (parent, superclass).

- Transitivity
  - If class B inherits from A, (written A > B) then all subclasses of B also inherit from A.
    - E.g. if A > B and B > C then A > C (C inherits from A).

- Why inheritance?
  - Reflects real-world structure.
    - People take on more and more specialized roles: person > employee > manager > CEO
    - Taxonomy of life: mammal > primate > human
    - Taxonomy of inanimate objects: vehicle > flyingVehicle > airplane
  - Increases code reusability. If we need a new data type that is largely the same as an existing one with minor changes to state or behavior, we use inheritance rather than copy and paste code. Why is such code bloat bad?

When to use inheritance ?

- To model the “is-a” relationship
  - An employee is a person
  - An airplane is a flyingVehicle
  - An airplane is a vehicle
  - A human is a mammal

- Not for containment: has-a
  - An employee has a car
  - An airplane has a wing
  - A mammal has-a brain

- Generally used to make a more specialized version of the base class rather than to take away functionality

- Given these classes are defined:
  - Vehicle is base class of 4WheelDriveJeep
- Is this OK: 4WheelDriveJeep is base class of car ?
  - No! While tempting to reuse 4WheelDriveJeep as the base of car, we would need to “delete” functionality of 4WheelDriveJeep like “goOffroading()”
  - In OOP: A car is not a 4wheelDriveJeep with less functionality
  - You cant use a car wherever you would use a Jeep

Example: Class inheritance

```python
class Person(object):
    #constructor
    def __init__(self, name, idnumber):
            self.name= name
            self.idnumber = idnumber
            print(f"inside the Person constructor")

    def display(self):
            print(f"\n")
            print(f"name {self.name}")
            print(f"idnumber {self.idnumber}")
            print(f"------\n")
```

```python
# subclassing from Person class
class Employee(Person):
    def __init__(self, name, idnumber, salary, role):
        # invoking the __init__ of the parent class
        #super(Employee, self).__init__(name, idnumber)
        Person.__init__(self, name, idnumber)

        self.salary = salary
        self.role = role
        print(f"inside the Employee constructor")

    def personDetails(self):
        print(f"Person {self.name} has id {self.idnumber} and salary {self.salary} for their role {self.role}")
```

Observations

- **Coding conventions**
  - **Class names start with a capital letter and use CamelCase**
  - **Methods start with a lowercase letter**
  - **Attributes start with a lowercase letter**
- **self is a reference to the current instance of the class**
- **Attributes of the current instance are accessed through self**
  - **E.g. self.salary**
- **Methods (such as all the methods shown here that operate on a class instance) always take at least one argument, self. Additional arguments when needed are listed afterwards.**
- **Non __init__() methods can add attributes but this is generally not recommended practice.**

Example: Class inheritance

In [50]:
```
# subclassing from Person class
class Employee(Person):
    def __init__(self, name, idnumber, salary, role):
        # invoking the __init__ of the parent class
        #super(Employee, self).__init__(name, idnumber)
        Person.__init__(self, name, idnumber)

        self.salary = salary
        self.role = role
        print(f"inside the Employee constructor")

    def personDetails(self):
        print(f"Person {self.name} has id {self.idnumber} and salary {self.salary} for their role {self.role}")
```

In [51]:
```
e1=Employee("Joanna", 8723, 40000, "intern")
print(e1)
e1.display()
e1.personDetails()
print(f"==============")
e2=Employee("Sam", 8888, 38000, "intern")
print(e2)
e2.display()
e2.personDetails()
```

```
inside the Person constructor
inside the Employee constructor
<__main__.Employee object at 0x2aaab7eb4730>


name Joanna
idnumber 8723
------

Person Joanna has id 8723 and salary 40000 for their role intern
==============
inside the Person constructor
inside the Employee constructor
<__main__.Employee object at 0x2aaad45e7940>


name Sam
idnumber 8888
------

Person Sam has id 8888 and salary 38000 for their role intern
```

Observations

- It is good practice to invoke the constructor of the base class at the beginning of the constructor of the subclass
  - In this way base is initialized and has the expected behavior
- Objects are stored at unique memory locations
- Since a subclassed object is-a base class object, we can naturally call methods of the base class. These are inherited methods (behavior).
- The subclass has the attributes of the base class, plus optionally additional attributes for the subclass
- By default, attributes and methods in python are publically accessible

Example: Class inheritance

```
class Person(object):
    #constructor
    def __init__(self, name, idnumber):
            self.name= name
            self.idnumber = idnumber
            print(f"inside the Person constructor")

    def display(self):
            print(f"\n")
            print(f"name {self.name}")
            print(f"idnumber {self.idnumber}")
            print(f"------\n")
```

```
# subclassing from Person class
class Employee(Person):
    def __init__(self, name, idnumber, salary, role):
        # invoking the __init__ of the parent class
        #super(Employee, self).__init__(name, idnumber)
        Person.__init__(self, name, idnumber)

        self.salary = salary
        self.role = role

    def personDetails(self):
        print(f"Person {self.name} has id {self.idnumber} and salary {self.salary} for their role {self.role}")
```

Containment

- When one object has another object, we use containment.

- Examples
  - One object uses or owns another object.
  - Employee has a companyCar that they have been given to use.
  - A GAN has a generator and a discriminator subnetworks.

- Generally, the contained object classes are defined first
- Then the class doing the containing (housing) sets an attribute to the contained object.

- This allows the contained object to be acted upon through the container.
  - E.g. The employee can drive their company car.

Containment example: contained (left) and container (right) class

```
class Car():
    def __init__(self, strColor, strMake, strModel, strLicPlate):
        self.strColor=strColor
        self.strMake=strMake
        self.strModel=strModel
        self.strLicPlate=strLicPlate

    def display(self):
        print(f'{self.strColor} {self.strMake} {self.strModel} with license {self.strLicPlate} ')
```

```
car1=Car('red', 'Honda', 'CRV', 'FGT-1090')
print(car1)
car1.display()
```

```
<__main__.Car object at 0x2aaad490b040>
red Honda CRV with license FGT-1090
```

```
# subclassing from Person class
class Employee(Person):
    def __init__(self, name:str, idnumber:int, salary:int, role:str, car: Car=None):
        # invoking the __init__ of the parent class
        #super(Employee, self).__init__(name, idnumber)
        Person.__init__(self, name, idnumber)

        self.salary = salary
        self.role = role
        self.companyCar=car
        print(f"inside the Employee constructor")

    def personDetails(self):
        print(f"Person {self.name} has id {self.idnumber} and salary {self.salary} for their role {self.role}",\
              end='')
        if self.companyCar == None:
            print(' and does not drive a company car')
        else:
            print(f" and drives company car ", end='')
            self.companyCar.display()

    def addCar(self, car: Car):
        self.companyCar=car

    def driveCar(self):
        if self.companyCar == None:
            print('Error, employee does not have a company car to drive')
        else:
            print(f"Driving company car ",end='')
            self.companyCar.display()
```

Containment example: usage

```
In [99]: e3=Employee('Jane', 9090, 42000, 'resident')
         e3.personDetails()
         e3.driveCar()
         print("======")
         e3.addCar(car1)
         e3.personDetails()
         e3.driveCar()
```

```
inside the Person constructor
inside the Employee constructor
Person Jane has id 9090 and salary 42000 for their role resident and does not drive a company car
Error, employee does not have a company car to drive
======
Person Jane has id 9090 and salary 42000 for their role resident and drives company car red Honda CRV with license
FGT-1090
Driving company car red Honda CRV with license FGT-1090
```

observations

- It is good practice to use type hints
- This makes it clear what type of variable should be supplied to a method
  - User defined types can be used in a type hint

- Ex: car: Car = None
  - Let's unpack this
  - car is the input argument
  - : Car is the type hint, where “Car” is the class (type) we defined.
  - = None specifies the default value for car when the function (ctor) is called without a car argument.
  - None is Python’s way of indicating the variable is not defined. Kind of like a null pointer in C/C++

- In our example we create a person that has not been assigned a company car.
- Later we add a companyCar to that person
- Then the person can drive their companyCar.

Advanced concepts and terminology of OOP

- **Method overriding**
  - **Base class can define a method, display()**
  - **Subclass can override that method, defining subclass specific behavior.**
    - **Advanced note: Python does not support method overloading**. When we define multiple functions with the same name, the later one always overrides the prior and thus, in the namespace, there will always be a single entry against each function name.

- **Polymorphism**
  - **A function can take a base class as an input argument and operate on it calling methods of the base class. If supplied with derived class objects it will still work, with derived class dependent behavior. Since the input can take many forms, this is called polymorphism.**
  - **When there are more than one subclasses of a given base class, each overriding a base class method, the function taking a base class object as an input can get subclass specific behavior implicitly by calling the (overridden) base class method.**
  - **This in turn calls the overridden method of the subclass and not the base class method.**
  - The same action may be executed in a variety of ways because of polymorphism.
  - **Why do this?**
    - **Because it reduces the amount of code we must write.**
  - **Examples:**
    - **Write one function say to sort a list of objects regardless of their type.**
    - **Take a step in gradient decent with a subclassed model-specific train_step() method.**

Polymorphism example:

Overridden methods:

```
# Illustrate method overriding and polymorpshism
class Vehicle:
    def drive(self):
        print("driving, but fuel is undefined.")


class FuelCellCar(Vehicle):
    def drive(self):
        print("driving fueled by hydrogen")

class EVCar(Vehicle):
    def drive(self):
        print("driving fueled by electric charge")

class CombustionCar(Vehicle):
    def drive(self):
        print("driving fueled by gasoline")
```

Same action, multiple behaviors:

```
In [113]: c0=FuelCellCar()
          c1=CombustionCar()
          c2=FuelCellCar()
          c3=EVCar()
          listCars=[c0, c1, c2,c3]

          def driveCars(myList):
              for idx, car in enumerate(myList):
                  print(f"Car {idx} is ", end='')
                  car.drive()

In [114]: driveCars(listCars)

          Car 0 is driving fueled by hydrogen
          Car 1 is driving fueled by gasoline
          Car 2 is driving fueled by hydrogen
          Car 3 is driving fueled by electric charge
```

Further reading

- **Summerfield 2010, Programming in Python 3, Ch 6: Object Oriented Progamming**
- **Lutz, 2013, Learning Python, Ch 29, ch 31**

Keras overview

OOP: Defining a language within a language

- **OOP allows definition of user-defined types**

- **For many fields of research, a set of well-chosen types allows the SWE to be more productive.**
  - **The types are the core building blocks for getting work accomplished**
  - **Invoking the methods are the most performed tasks of the SWE using the types.**
  - **Such a set of such types defines a new language that the SWE can "speak".**
  - **Such a tailored-purpose language is known as a framework (library, API)**

- **Why? By creating instances of the types and manipulating them through its methods, the SWE can**
  - **focus on the bigger picture (work at a higher level of abstraction),**
  - **get more accomplished with fewer lines of code.**
  - **Fewer bugs per line of code executed (with a suitably debugged framework).**

- **Example: Tensorflow/Keras is an OOP framework for the rapid prototyping, development and deployment of deep neural network models**
  - **Open source. Developed by Google. Used by thousands of SWE and researchers.**
  - **TF2.x is particularly well designed, relatively bug free.**
  - **Excellent documentation. Many code examples, books available.**
  - **Latest version is very similar to PyTorch (another popular framework).**

OOP: Being an efficient developer

- **What is the largest expense in SW development?**
  - **You, the SWE developer, are the biggest expense.**
- **Cost per line of bug free code is very high.**
- **How can you cost-justify your employment (e.g., in industry)?**
- **How can you be an efficient, competitive research (in academia)?**
  - **By learning the best OOP SW frameworks for the challenges you face.**
  - **By extending the provided types through subclassing.**
- **Stand on the shoulders of others**
- **By developing your own subclassed types, you make yourself, and the colleagues you share your code with more productive**
  - **Hastens the progress of science (you, your lab, your scientific community)**

What is tensorflow/keras?

- **Tensorflow (TF 2.x in this course) is an intermediate level OOP python library for the construction of computational graphs.**
  - **Built in support for efficient execution of the computation on most modern HW (CPU, GPU, TPU)**
  - **Often used for development of neural networks (NNs) → specify a computational graph**
  - **Train on any (high powered) hardware. Deploy fully trained model to same or lower powered HW.**
  - **Arguably the best support for deployment of any current library (as of 2024)**

- **Keras is a high-level OOP library for the construction of NNs.**
  - **Since TF 2.x has been fully integrated into TF and is the official high-level language to develop TF based NNs.**
    - **User friendly: its classes and methods focus on the big concepts (Models, Layers, Optimizers, Datasets) making construction of complex NNs easier.**
    - **Makes the developer productive: a lot of the detailed plumbing is hidden away (encapsulated) in the implementation of the classes.**
    - **An ideal DL library for beginners.**

- **Is Keras also useful for hardcore research?**
  - **Yes. Enables fast-model prototyping**
  - **Keras offers a spectrum of different workflows, based on the principle of *progressive disclosure of complexity***
    - ***make it easy to get started yet make it possible to handle high-complexity use cases, only requiring incremental learning at each step.***

Reminder: Why ML practitioners build NN models?

- **To learn an association btwn predictors and target, and to embody the mapping in a predictor that we can use on new data, unseen-during-training.**

- **Comparison of ML practitioner to a statistician. Do YOU Agree/disagree?**
  - **Commonly use an off the shelf, linear model (e.g., multiple linear model) with existing fully implemented algorithm. Weeks to do.**
  - **ML practictioner builds a custom nonlinear model. Effectively constructing a model from one of the infinite possibilities. Lengthy search for model that best captures/maximizes the association btwn predictor and target. This can take months even years to do.**

- **Often the problem entails supervised learning. We will consider two such problems today:**
  - **MNIST: You are already familiar: Build a classifier of the handwritten digits: 0-9**
  - **Build a classifier of customer support tickets**
    - **Given inputs:**
      - **title of ticket (text), and body of the ticket (text), tags added by the user**
    - **Preprocess the text by encoding the words into binary arrays (values of 0 or 1) of size *vocabulary_size* depending on whether the word was present at least once or not.**
    - **2 outputs:**
      - ***priority score* of the ticket: scalar btwn 0…1 (sigmoid output)**
      - ***department* that should handle the ticket (posterior over departments)**

OO library design begins with noun analysis

- **To design a new OO library tailored for a given class of problems (NNs) we begin by writing a textural description of the typical use case.**

- **We would then look at the key nouns in that description. They would be candidates for the classes in our library.**

- **Description:**
  - **We need to build a neural network model composed of one or layers, which is fit to the given dataset by minimizing a loss function using an optimizer. We will evaluate the model's by computing one or more performance metrics.**

- **What do you think are the key nouns here?**
  - **Dataset**
  - **Model**
  - **Layers**
  - **Loss function**
  - **Optimizer**
  - **Metrics**

OO class design begins with behavior analysis

- **To design a class, consider what it must do. Its functions will become its methods. Those methods may need to save or alter some internal state, which in turn becomes its attributes.**

- **What behavior does a model have?**
  - **Allow for model architecture specification**
  - **Make the model compile-able by specifying how to optimize its weights.**
  - **Fit the model to the training dataset**
  - **Evaluate fully trained model on the training dataset**
  - **Use the model to predict the target for new samples not in training.**

Model class: keras library

- **Allow for model architecture specification (NN model design)**
  - **Model is composed of layers that have layer weights. Architecture specifies how layers are connected together so as to transform the inputs to the outputs**
  - **3 different ways to specify: Sequential model, Functional API, subclassing keras.Model**
  - **All of them require specifying inputs, outputs and layers with connectivity.**

- **model.compile() … specify:**
  - **Optimizer … the specific gradient decent algorithm to use to optimize the loss function**
  - **Loss function … objective function measuring dissimilarity btwn predictions and labels. Differentiate to update model weights during model fitting.**
  - **Metrics .. Which quantify model performance (RMSE, cross entropy)**

- **model.fit() … adjust the model's layer's weights to make accurate predications on the training dataset**

- **model.evaluate()**
  - **Compute the loss and the performance metrics of the fully trained model on the training dataset**

- **model.predict()**
  - **Predict target labels for new samples not seen during training (model fitting).**

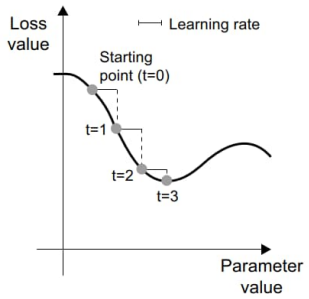
model.fit(): gradient based optimization

- **Each unit in a MLP layer transforms inputs to outputs:**

```
output = relu(dot(input, W) + b)
```

- **W and b are kernel and bias weights (trainable parameters).**
- **Initialized to small random values. Starting point but non-predictive.**

- **fit() iteratively adjusts the weights to make the prediction loss (error) smaller.**
- **Training loop repeats the following train_step() until loss is low:**



1. Draw a batch of training samples, `x`, and corresponding targets, `y_true`.
2. Run the model on `x` to obtain predictions, `y_pred` (this is called the *forward pass*).
3. Compute the loss of the model on the batch, a measure of the mismatch between `y_pred` and `y_true`.
4. Compute the gradient of the loss with regard to the model's parameters (this is called the *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example, `W -= learning_rate * gradient`—thus reducing the loss on the batch a bit. The *learning rate* (`learning_rate` here) would be a scalar factor modulating the "speed" of the gradient descent process.

- **This is the mini-batch stochastic gradient descent (mini-batch SGD) grad descent alg.**
  - **Stochastic because each batch of data is drawn at random (w/o replacement).**

model.fit(): gradient based optimization

- **Each unit in a MLP layer transforms inputs to outputs:**

```
output = relu(dot(input, W) + b)
```

- **Conceptually you can think of this pseudocode (not keras but close) for the train_step() iteration.**
- **The provided arguments are inputs .. x_train and targets … y_train (labels)**

```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(predictions, targets)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

1. Forward pass, inside a gradient tape scope
2. Retrieve the gradient of the loss with regard to weights.
3. Update the weights.

Keras: Overview of primary classes

- Such design is repeated for each class to form an overall class library design. SWEs implement the design, bug fix, maintain, improve.

With usage and refinement, the library matures into a powerful language (for NNs) within a language (Python). Occurring since initial release in 2015. TF 2.x includes following primary classes:

1. Models (`tf.keras.models`): This governs the overall type of architecture of the neural network.

2. Layers (`tf.keras.layers`): neural networks are built up as a sequence of layers. These layers can be of many types, e.g., fully-connected `dense` layers as we use here, and 2D convolutional (`conv2D`) layers.

   - Comparison of Models and Layers
     - layer … is a building block used to create models.
     - model … top-level object that is used to actually learn on training data and to export for inference on new data.
       - A model has fit(), evaluate(), and predict() methods. Layers don't.
       - You can also save a model to a file on disk via save_model().
       - Otherwise, both classes are similar: both have an __init__() used to define (sub)layers and a call() method that implements the forward pass and connects the (sub)layers

Keras library: primary classes

3. **Optimizers (`tf.keras.optimizers`): This is what keras uses to learn the neural network weights from the training data.**

4. **Sample Data (`tf.keras.datasets`): Keras comes with several popular machine learning data sets to make it easy to benchmark and test our neural network architectures.**

4 ways to use classes to specify model architecture

1. Sequential API

- Model instantiation via `model=keras.Sequential()`
- Pros/Cons: simple to use, but only allows straight line computation graph (1 input & output).

2. Functional API

- Instantiate 1+ input layers
- Instantiate transformational layers and call them to connect layers into a computation DAG
- Think: snapping together LEGO bricks.
- Pros/Cons: more flexible than Seq API, allows multi-input, multi-output. Allows for direct access to any layer (facilitates view, inspect & modify any layer at will e.g., for plotting of layer connectivity architecture, for xfer learning). But only allows specifying a DAG (e.g., no cycles, no adversary support)

- Example: Build a classifier of customer support tickets
  - Given inputs:
    - title of ticket (text), and body of the ticket (text), tags added by the user
  - Preprocess the text by encoding the words into arrays of 0/1 of size vocabulary_size depending on whether the word was present at least once or not.
  - 2 outputs:
    - *priority score* of the ticket: scalar btwn 0…1 (sigmoid output)
    - *department* that should handle the ticket (posterior over departments)

Concrete example of the functional API for model specification

1. Specify the multi-input, multi-output model via the functional API

```
vocabulary_size = 10000
num_tags = 100
num_departments = 4

title = keras.Input(shape=(vocabulary_size,), name="title")
text_body = keras.Input(shape=(vocabulary_size,), name="text_body")
tags = keras.Input(shape=(num_tags,), name="tags")

features = layers.Concatenate()([title, text_body, tags])
features = layers.Dense(64, activation="relu")(features)

priority = layers.Dense(1, activation="sigmoid", name="priority")(features)
department = layers.Dense(
    num_departments, activation="softmax", name="department")(features)

model = keras.Model(inputs=[title, text_body, tags],
                    outputs=[priority, department])
```

1. Define model inputs.

2. Combine input features into a single tensor, features, by concatenating them.

3. Apply an intermediate layer to recombine input features into richer representations.

4. Define model outputs.

5. Create the model by specifying its inputs and outputs.

2. Train the model by fitting it to the training data.

```
import numpy as np

num_samples = 1280

title_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
text_body_data = np.random.randint(0, 2, size=(num_samples, vocabulary_size))
tags_data = np.random.randint(0, 2, size=(num_samples, num_tags))

priority_data = np.random.random(size=(num_samples, 1))
department_data = np.random.randint(0, 2, size=(num_samples, num_departments))

model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "categorical_crossentropy"],
              metrics=[["mean_absolute_error"], ["accuracy"]])
model.fit([title_data, text_body_data, tags_data],
          [priority_data, department_data],
          epochs=1)
model.evaluate([title_data, text_body_data, tags_data],
               [priority_data, department_data])
priority_preds, department_preds = model.predict(
    [title_data, text_body_data, tags_data])
```

1. Dummy input data

2. Dummy target data

3. evaluate computes the loss and perf metrics for the final fitted model

4. unpack prediction results

5. predict computes the target predictions for each provided sample but not the performance metrics

[Adapted from Chollet, 2021]

41

…4 ways to use classes to specify model architecture

1. Sequential API

- Model instantiation via `model=keras.Sequential()`
- Pros/Cons: simple to use, but only allows straight line graph (1 input & output).

2. Functional API

- Instantiate 1+ input layers
- Instantiate transformational layers and call them to connect layers into a DAG
- Think: snapping together LEGO bricks.
- Pros/Cons: more flexible than Seq, allows multi-input, multi-output. Allows for direct access to any layer (facilitates view, inspect & modify any layer at will for e.g. arch. Plotting of layer connectivity, xfer learning). But only allows specifying a DAG.

3. Subclass the keras.Model

- `class myClass(keras.Model): __init__(), call(), train_step()`
- Pros/Cons:
  - Groups model spec and tailored behavior (e.g. fitting) into one place, facilitating code reuse and maintenance.
  - Allows most flexibility in specifying how layers are used together, not just DAG:
    - `Call()` uses layers inside a `for` loop, or calls them recursively. Anything is possible.
  - But you are responsible for more of model logic (increased bug potential), requires custom code for layer access (e.g. for plotting).

4. Combine 2+ above, e.g., subclassing with functional API for model spec.

- What we will illustrate in this course because it combines strengths of Keras subclassing API (maintainability) with ease of direct access of functional API.

Concrete example: Combine subclassing & func. API to specify model

1. Specify the multi-input, multi-output model via the subclassing API

```
class CustomerTicketModel(keras.Model):

    def __init__(self, num_departments):
        super().__init__()
        self.concat_layer = layers.Concatenate()
        self.mixing_layer = layers.Dense(64, activation="relu")
        self.priority_scorer = layers.Dense(1, activation="sigmoid")
        self.department_classifier = layers.Dense(
            num_departments, activation="softmax")

    def call(self, inputs):
        title = inputs["title"]
        text_body = inputs["text_body"]
        tags = inputs["tags"]

        features = self.concat_layer([title, text_body, tags])
        features = self.mixing_layer(features)

        priority = self.priority_scorer(features)
        department = self.department_classifier(features)
        return priority, department
```

1. Don't forget to call the super() constructor!

2. Define sublayers in the constructor.

3. hidden layer

4. Define the forward pass in the call() method.

5. Either unpack the inputs or as done here, require the inputs to be in the form of a dictionary with these keys

6. Use Functional API to specify the connectivity btwn layers instantiated in the constructor __init__()

7. The forward pass, call() must return the target predictions

2. Train the model by fitting it to the training data.

```
model = CustomerTicketModel(num_departments=4)

priority, department = model(
    {"title": title_data, "text_body": text_body_data, "tags": tags_data})

model.compile(optimizer="rmsprop",
              loss=["mean_squared_error", "categorical_crossentropy"],
              metrics=[["mean_absolute_error"], ["accuracy"]])
model.fit({"title": title_data,
           "text_body": text_body_data,
           "tags": tags_data},
          [priority_data, department_data],
          epochs=1)
model.evaluate({"title": title_data,
                "text_body": text_body_data,
                "tags": tags_data},
               [priority_data, department_data])
priority_preds, department_preds = model.predict({"title": title_data,
                                                  "text_body": text_body_data,
                                                  "tags": tags_data})
```

1. invoke the forward pass, call()

input is a dictionary of data components

2. Note: compile, fit evaluate update the state of the model In this code they are not returning anything explicitly

3. The structure of what you pass as the loss and metrics arguments must match exactly what gets returned by call()—here, a list of two elements.

one loss function for each output

x_train

y_train

5. The structure of the target data must match exactly what is returned by the call() method—here, a list of two elements.

6. evaluate() computes loss and performance metrics for the provided samples but does

4. The structure of the input data must match exactly what is expected by the call() method—here, a dict with keys title, text_body, and tags.

7. Predict returns predictions but not performance.

[Adapted from Chollet, 2021]

43

3 different ways to train a model

1. call model.fit()

- **Pros/Cons:**
  - **Minimal code to write. Bookkeeping including identifying the next mini batch, computing the forward pass, computing gradients of loss wrt trainable parameter weights and updating them via gradient descent optimizer is done for you.**
  - **But: it only works for DAG models. Relatively simple progression from 1+ inputs to 1+ outputs with all parts of the model optimized together (simultaneously); no internal competition.**

2. Customize model.train_step() which is itself called by model.fit()

- **Pros/Cons:**
  - **more flexible: you train a non-DAG model or keep one portion of the model fixed (e.g., Discriminator), while updating the weights of another part of the model (e.g., generator) when those parts are competing in a min-max game**
  - **Suitable for all types of architectures, including the VAE, GAN, VAE-GAN models we will build in this course.**

3. Write your own training algorithm

- **Override model.fit() entirely with your own logic.**
- **Pros/Cons:**
  - **Total control, but you have to write your own book-keeping code (mini batch construction, etc)**

Concrete example: customize model.train_step()

The steps are:

1. Create a new class that subclasses keras.Model.

2. Override the method train_step(self, data). It returns a dictionary mapping metric names (including the loss) to their current values.

3. Implement a metrics property that tracks the model's Metric instances. This enable the model to automatically call reset_state() on the model's metrics at the start of each epoch and at the start of a call to evaluate(), so you don't have to do it by hand.

…Concrete example: customize model.train_step()

1. Implement a custom train_step() to define the per mini-batch gradient step to be used with fit()

```
loss_fn = keras.losses.SparseCategoricalCrossentropy()
loss_tracker = keras.metrics.Mean(name="loss")

class CustomModel(keras.Model):
    def train_step(self, data):
        inputs, targets = data
        with tf.GradientTape() as tape:
            predictions = self(inputs, training=True)
            loss = loss_fn(targets, predictions)
        gradients = tape.gradient(loss, model.trainable_weights)
        optimizer.apply_gradients(zip(gradients, model.trainable_weights))

        loss_tracker.update_state(loss)
        return {"loss": loss_tracker.result()}

    @property
    def metrics(self):
        return [loss_tracker]
```

1. This metric object will be used to track the average of per-batch losses during training and evaluation.

2. We override the train_step method.

3. We use self(inputs, training=True) instead of model(inputs, training=True), since our model is the class itself.

4. We update the loss tracker metric that tracks the average of the loss.

5. We return the average loss so far by querying the loss tracker metric.

6. Any metric you would like to reset across epochs should be listed here.

2. Train the model by fitting it to the training data.

```
# 784 dimensions in a MNIST digit
inputs = keras.Input(shape=(28 * 28,))
features = layers.Dense(512, activation="relu")(inputs)
features = layers.Dropout(0.5)(features)
outputs = layers.Dense(10, activation="softmax")(features)
model = CustomModel(inputs, outputs)

model.compile(optimizer=keras.optimizers.RMSprop())
model.fit(train_images, train_labels, epochs=3)
```

1. While the functional API is used here. for simplicity, we recommend using _init_() and call() instead, as this keeps more model code together.

2. Invoking fit() will in turn call our CustomModel.train_step for each mini batch.

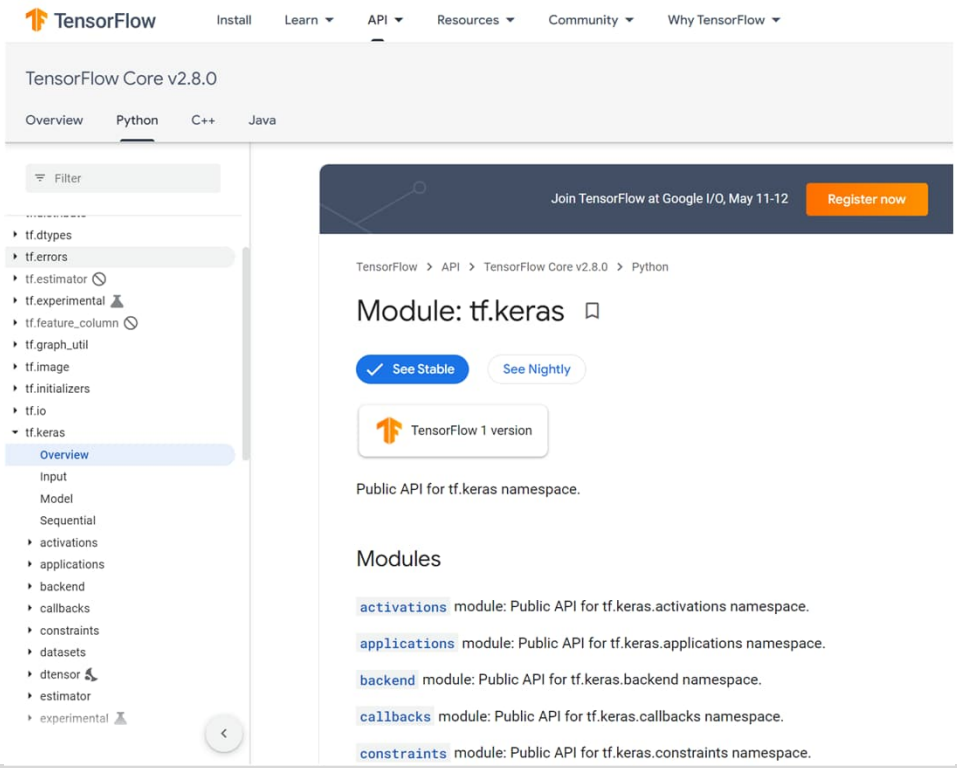
[Adapted from Chollet, 2021]

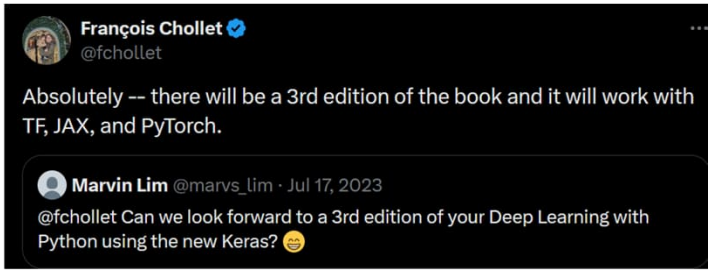
46

Further reading

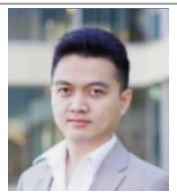
- **Chollet, 2021, Deep Learning with Python, 2nd Ed**

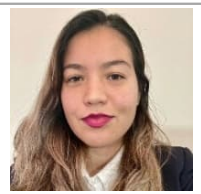
- **Read the documentation at https://www.tensorflow.org/api_docs/python/tf/keras**
- **E.g. you get stuck on the API for a Class or method then look it up at this site.**





Acknowledgements


Son Nguyen, PhD
Postdoc


Aixa Andrade Hernandez
MS, PhD student


Austin Marckx
PhD student


Krishna Chitta
Res. Sci.

---------- Recent Alumni ----------


Alex Treacher
PhD student


Atef Ali
Undergrad


Vyom Raval, BS
MD/PhD


Kevin Nguyen
MD/PhD student


Cooper Mellema
MD/PhD student

Mentees: Postdoc, PhD Students, scientists, ugraduates

- **Austin Marckx, Cooper Mellema, Son Nguyen, Kevn Nguyen, Alex Treacher, Aixa Andrade Hernandez, Krishna Chitta,**

Agencies for their Funding Support

- **NIH NIGMS, NINDS, NIA; King and Lyda Hill Foundations, Texas Alzheimer's Research and Care Consortium.**

Thank you!

Email: Albert.Montillo@UTSouthwestern.edu
Github: https://github.com/DeepLearningForPrecisionHealthLab

MegNET .... Artifact suppression
BLENDS .... fMRI augmentation
Antidepressant-Reward-fMRI .... response prediction
Parkinson-Severity-rsfMRI ... disease trajectory prediction

End of presentation