CME 216, ME 343 - Spring 2020

Eric Darve, ICME

Finally, we cover custom layers and models. This is done using Python subclassing (more on this later).

This is the most general technique to build DNNs.

The Sequential API and the Functional API are declarative.

A declarative programming style is one where the user expresses the *logic* of a computation without describing its *control flow.*

Said otherwise, the user describes *what* the object should do but not directly *how*.

The subclassing method is a type of imperative programming.

That is, this is an approach where the user describes *how* the program operates.

Imperative example.

You enter a restaurant and you say:

"I see that this table in the corner is empty. My wife and I are going to take it."

Declarative example.

You enter a restaurant and you say:

"A table for two, please."

The subclassing approach uses the imperative style of programming.

More information about the declarative (or symbolic) and imperative APIs in TF.

Subclassing requires using Python inheritance.

You do not really need to know the details of this.

If you know how to use the proper syntax, it is good enough for most situations.

But let us do a little more and explain what subclassing is and how it works in Python.

Inheritance is a mechanism where new classes are derived (or built on) previous classes.

The class from which a class inherits is called the parent class or superclass.

A class that inherits from a superclass is called a subclass, also called heir class or child class.

In Keras, you can subclass `tf.keras.layers.Layer` and `tf.keras.Model`.

For simplicity, we will just look at subclassing `tf.keras.Model`.

Here is the basic syntax:

```
class MyModel(keras.Model):
def __init__(self, **kwargs):
super().__init__(**kwargs) # handles standard args (e.g., name)
[...]

def call(self, input_):
[...]
```

In `__init__()` we will set up all the data-structures (layers) that are needed by our model.

`call()` defines the sequence of computations the DNN should perform.

Python uses the method `__init__` to initialize the state of a new object of that class.

This is a constructor.

It is called when a new object of that class is created.

The class `MyModel` derives from the class `keras.Model`.

Derived classes in Python inherit the methods and class attributes from their parent classes.

In our case, `MyModel` inherits from `keras.Model`.

Because we are subclassing, all the methods from `keras.Model` are available.

In particular, we can call the methods `compile`, `fit`, `predict`, and `evaluate` from `keras.Model`.

```
class MyModel(keras.Model):
def __init__(self, **kwargs):
super().__init__(**kwargs) # handles standard args (e.g., name)
[...]

def call(self, input_):
[...]
```

`super().__init__(**kwargs)` allows calling the `__init__` method of the parent class.

This ensures that the constructor of the parent class (and potentially all the relevant ancestor classes) is called.

`super()` is somewhat complicated to fully explain in this lecture.

To simplify the discuss, we will say that `super()` is referring to the parent class.

`super()` is closely connected to the concept of the method resolution order.

See the `__mro__` attribute.

`super()` is great to call a function defined by a parent class.

But it is most useful in cases of multiple inheritance.

See this demo Python code for details and examples using `super()`.

For more information about `super()` see the super() Python doc and this blog by Hettinger.

`__init__()`

```
def __init__(self, **kwargs):
super().__init__(**kwargs) # handles standard args (e.g., name)
self.hidden1 = keras.layers.Dense(4, activation="relu")
self.hidden2 = keras.layers.Dense(4, activation="relu")
self.hidden3 = keras.layers.Dense(4, activation="relu")
self.out = keras.layers.Dense(1, activation="linear")
```

As we explained above, we first call `super().__init__()` so that the parent classes are initialized.

Then we build the three hidden layers and the output `self.out`.

Note how, although we are defining 3 hidden layers and 1 output layer, we are not specifying how they are going to be used.

This will be done in `call()`.

Compared to the previous case, we changed the activation function to `relu`.

call()

```
def call(self, input_):
hidden1 = self.hidden1(input_)
hidden2 = self.hidden2(hidden1)
hidden3 = self.hidden3(hidden2)
concat = layers.Concatenate()([input_, hidden3])
return self.out(concat)
```

`call` then defines the actual sequence of calculation to perform.

`self.hidden1(input_)` uses

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

to calculate numerical values that are stored in `hidden1`.

`concat` does not need to be part of the class since it is computed from `input_` and `hidden3`.

We note that since we only define the layers and the sequence of operations, we have left a few things undefined.

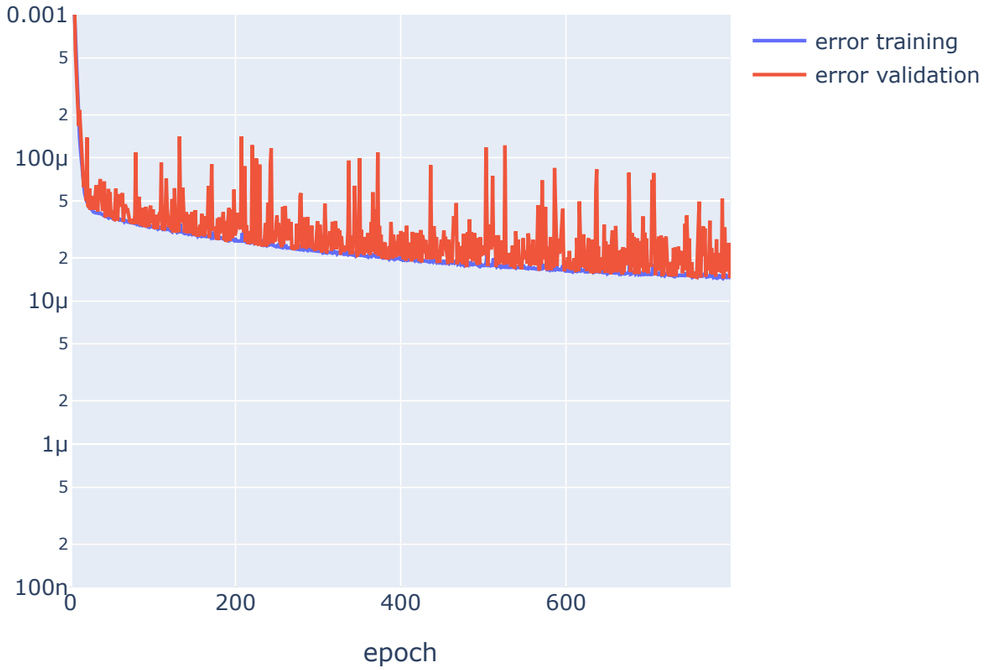
For example, the size of `input_` is not defined yet.

The shape of the input is defined later when calling `fit`.

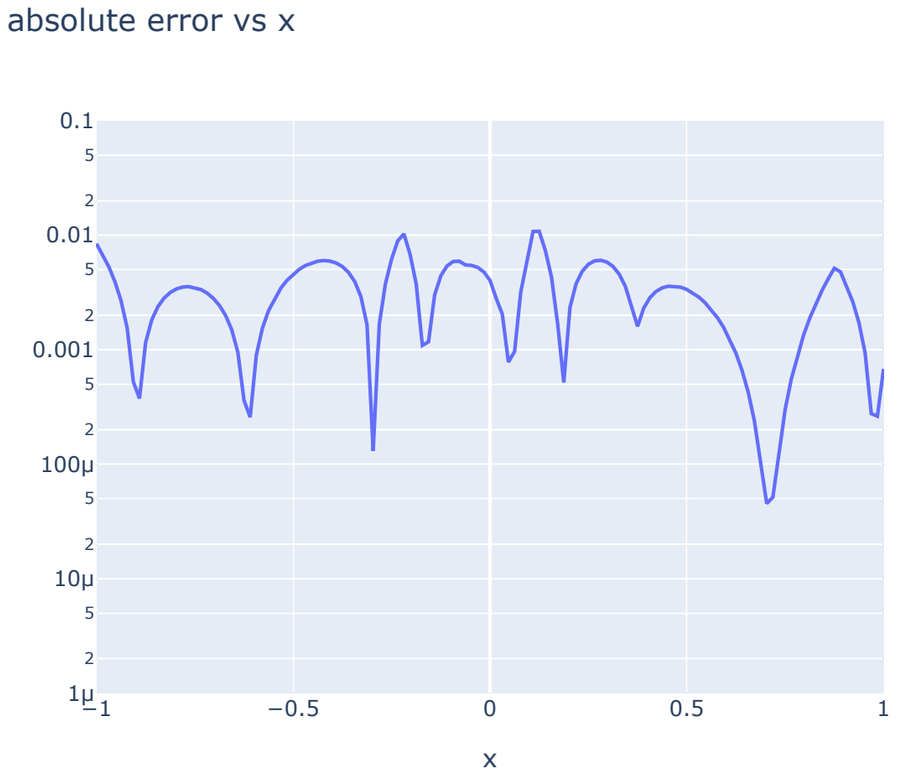
The rest of the code is the same as what we had before.

```
model.compile(loss='mse', optimizer=sgd, metrics=['mse','mae'])
history = model.fit(X_train, y_train, epochs=n_epochs,
validation_data=(X_valid, y_valid))
```

Convergence



Error


absolute error vs x
0.1
5
2
0.01
5
2
0.001
5
2
100µ
5
2
10µ
5
2
1µ
−1
−0.5
0
0.5
1
x

The `relu` activation is doing a little worse in this case.

For more details on the different APIs and subclassing, please see these two videos from the TF team.

Part 1

Part 2

Finally, we show a different example where we use a different input.

The first observation is that the function is even.

So we could use as input x^2.

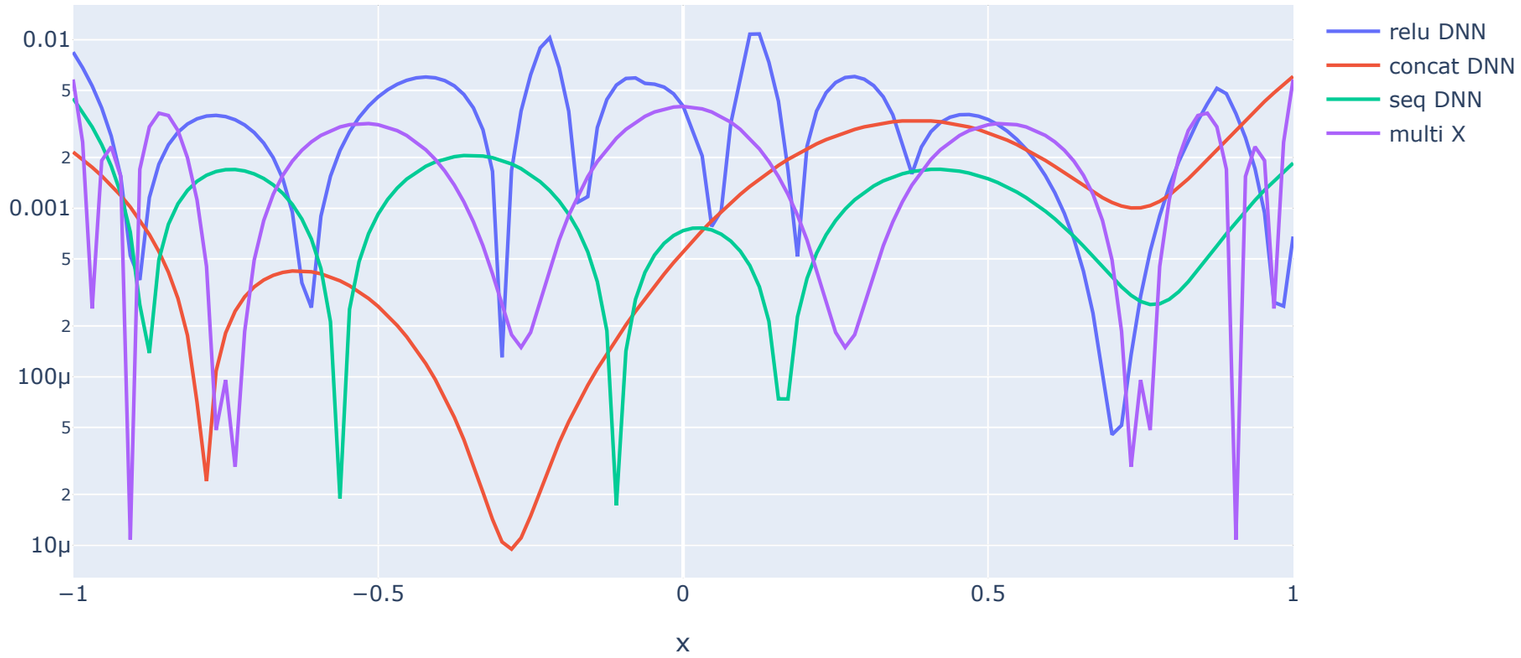
But we can use more inputs as well.

Let's try

$$(2x^2 - 1, 8x^4 - 8x^2 + 1)$$

These are the first even Chebyshev polynomials of order 2 and 4.

Let us compare all these models.


0.01
5
2
0.001
5
2
100μ
5
2
10μ
−1
−0.5
0
0.5
1
x
relu DNN
concat DNN
seq DNN
multi X

`relu` has the worst performance.

`multi X` is relatively more efficient as it exhibits an error similar to `seq DNN` but uses half of the training data.

Note how difficult it is to train these models to get high accuracy.

The convergence is rather slow.