Advanced Keras: motivation

We have been using the Keras Model API (mostly the `Sequential`) as a black box.

But it is highly customizable

- A `Model` is a class (as in Python object)
- It implements methods such as
  - `compile`
  - `fit`

We can change the behavior of a model in several ways

- Arguments to some methods are objects; we can pass non-default functions/objects
    - e.g., custom loss function
- We can override these (and other) methods to make our models do new things.

The `Layer` is also an abstract class (Python) in Keras.

Hence

- We can create new layer types
- We can override the methods of a given layer

In this module

- we will illustrate techniques that you can use to customize your Layers/Models.
- Illustrate the Functional model

Functional model: the basics

The `Sequential` model

- organizes layers as an ordered list
- restricts the input to layer $(l+1)$ to be the output of layer ll.

The `Functional` model

- imposes **no** ordering on layers
- imposes **no** restriction on connect outputs of one layer to the input of another

To illustrate the `Functional` model let's take a first look at model implementing a single `Transformer` block

- we will revisit this code later to illustrate other concepts

Here is the picture of a Transformer block

Transformer (Encoder/Decoder)

There are actually 3 models in this cell we will visit !

The Encoder side of the transformer:

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="encoder_inputs") x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs) encoder_outputs = TransformerEncoder(embed_dim, latent_dim, num_heads)(x) encoder = keras.Model(encoder_inputs, encoder_outputs)

This illustrates the pattern common to `Functional` models

- The output of a layer is assigned to a variable (e.g., `encoder_inputs` has the value of the model's inputs)
- The output of a layer is connected to the input of another layer via "function call" syntax
    - e.g., `encoder_inputs` is applied as the input to the `PositionalEmbedding` layer
        - `x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)`

The collection (not necessarily a sequence) of `Layer` calls defines the mapping of `Model` inputs to outputs.

To turn this collection into a `Model`

- We define the inputs to the model
- We define the output of the model

But a `Model` is a complete mapping from the mini-batch examples to the function computed by the `Model`.

For example, we define the Encoder side (sub-model of the Transformer) of the Transformer via

```
encoder = keras.Model(encoder_inputs, encoder_outputs)
```

This defines `encoder` to be a `Model` with

- input: `Layer` `encoder_inputs` (i.e., the `Input` layer)
- output: `Layer` `encoder_outputs` (i.e., the `TransformerEncoder`)

Note: the input and output of a `Model` *don't have to be* `Layer` types !

There is also a model for the Decoder side of the Transformer in the cell we will visit:

```
decoder = keras.Model([decoder_inputs, encoded_seq_inputs], decoder_outputs)
```

- input: An **array** of 2 `Layer` types -- `[ decoder_inputs, encoded_seq_inputs ]`
- output: `Layer`: `decoder_outputs`

Recall (from the Transformer picture) that the Decoder side consumes two inputs

pay close attention to the difference between $\bar{\mathbf{y}}$ (Encoder states) and $\mathbf{y}$ (Decoder outputs)

Transformer Layer (Encoder/Decoder)

- The output sequence $\bar{\mathbf{y}}_{(1..\bar{T})}$ (i.e., latent states) of the Encoder
    - Decoder-Encoder attention
    - $||\bar{\mathbf{y}}|| = \bar{T} = \text{length of Transformer input}$
- The prefix of the Decoder outputs generated up to time t
    - The Decoder output at time $(t-1)$ is appended to the Decoder inputs available at time t
    - So the inputs are the Decoder outputs $\mathbf{y}_{(1..T)}$
        - T is *full* length of Transformer output
        - Causal (Masked) Attention is used to restrict the Decoder
            - from attending at step t to any $\mathbf{y}_{(t)}$ where $t > (t-1)$
            - Can't look at an output that hasn't been generated yet !

Hence, the Decoder side takes a **pair** of inputs, as per the diagram.

Let's see if we can trace which role each element of the pair serves.

First, observe that the `Layer` sub-type `TransformerDecoder` actually implements the full Decoder.

- The second argument to `TransformerDecoder` has value `encoded_seq_inputs`
- `encoded_seq_inputs` is the second argument passed to `decoder`. See

  decoder = keras.Model([decoder_inputs, encoded_seq_inputs], decoder_outputs)

- `decoder` is called with

  decoder_outputs = decoder([decoder_inputs, encoder_outputs])

It would seem that `decoder_outputs` corresponds to $\mathbf{y}$ in our picture.

Thus, when `TransformerDecoder` is called

```
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, encoded_seq_inputs)
```

- The second argument to `TransformerDecoder` has value `encoded_seq_inputs = encoder_outputs`

So it seems that second argument to `TransformerDecoder` is $\bar{\mathbf{y}}$, the latent states of the Encoder

The first argument to `TransformerDecoder` , that is, the variable `x`

- is positionally-encoded (to enable Causal masking) `decoder_inputs`

  x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)

Hopefully: `decoder_inputs` are the `decoder_outputs` shifted by one time step

- The `PositionalEmbedding` is added to enforce Masking (causal ordering)

Finally, there is the Transformer Model, combining an Encoder and Decoder:

```
decoder_outputs = decoder([decoder_inputs, encoder_outputs]) transformer = keras.Model(
[encoder_inputs, decoder_inputs], decoder_outputs, name="transformer" )
```

The `transformer` first argument is `encoder_inputs`

- which is the $\mathbf{x}$ sequence in our picture (i.e., the input sequence)
- `encoder_inputs` causes `encoder_outputs` to be generated
- `encoder_outputs` ($\bar{\mathbf{y}}$ in our picture) is fed into the `decoder` generating `decoder_outputs` ($\mathbf{y}$ in our picture), as described above

A lot going on here !

- Hopefully:
    - `decoder_inputs` is equal to `decoder_outputs` shifted by one time step
    - Teacher forcing, enforced by the organization of the training data ?
- A complex connection of `Layer` outputs to inputs
- Custom `Layer` sub-types
    - `PositionalEmbedding`, `TransformerEncoder`, `TransformerDecoder`
    - We will soon see how to define our own `Layer` sub-classes

Here is a first look at the [Transformer code (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/neural_machine_translation_with_transformer.ipynb#s](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/neural_machine_translation_with_transformer.ipynb#s)

Model specialization

Custom loss (passing in a loss function)

In introducing Deep Learning, we have asserted that

> *It's all about the Loss function*

That is: the key to solving many Deep Learning problems

- Is not in devising a complex network architecture
- But in writing a Loss function that captures the semantics of the problem

Up until now

- We have been using pre-defined Loss functions (e.g., `binary_crossentropy`)
- Specifying the Loss function in the compile statement

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

You can [write your own loss functions (https://keras.io/api/losses/)](https://keras.io/api/losses/)

In Keras, a Loss function has the signature

```
loss_fn(y_true, y_pred, sample_weight=None)
```

Custom train step (override `train_step`)

But what if your Loss function needs access to values that are not part of the signature ?

Or what if you want to change the training loop ?

You could write your own training loop by overriding the `fit` method

- Cycle through epochs
- Within each epoch, cycle through mini-batches of examples
- For each mini-batch of examples: execute the *train step*
    - forward pass: feed input examples to Input layer, obtain output
    - compute the loss
    - Compute the gradient of the loss with respect to the weights
    - Update the weights

Rather than overriding `fit`, it sometimes suffices to override the train step: `train_step`

Let's start by looking at the "standard" implementation of a basic train step.

We will see

- How losses are computed
- Gradients are obtained
- Weights are updated

[Basic `train_step` (https://colab.research.google.com/github/tensorflow/docs/blob/snapshot-keras/site/en/guide/keras/customizing_what_happens_in_fit.ipynb#scrollTo=9022333acaa](https://colab.research.google.com/github/tensorflow/docs/blob/snapshot-keras/site/en/guide/keras/customizing_what_happens_in_fit.ipynb#scrollTo=9022333acaa)

We can modify the basic training step too.

For example: suppose we want to make some training examples "more important" than others

- Rather than Total Loss as equally-weighted average over all examples
- Pass in per-example weights

This might be useful, for example, when dealing with Imbalanced Data

`Layer` specialization

A `Layer` in Keras is an abstract (Python) object

- instantiating the object returns a function
    - That maps input to the layer to the output

We have used specific instances of `Layer` objects (e.g., `Dense`) as arguments in the list passed to the `Sequential` model type.

We can also use instances in the Functional Model.

For example

- `Dense(10)`
    - Is the constructor for a fully connected layer instance with 10 units
    - The constructor returns a function
    - The the function maps the layer inputs to the outputs of the computation defined by the layer

So you will see code fragments like > x = Input(shape=(784)) x = Dense(10, activation=softmax)(x)

- Re-using the variable x as the output of the current layer

When the function is invoked, the Layer's `call` method is used

- `call` gets invoked implicitly by "parenthesized argument" juxtaposition
    - e.g., `Dense(10) ( x )`
    - is similar to `obj`=Dense(10); result = obj.call(x)`
- The function maps the inputs to the layer to the output

Overriding `call` allows us to defined a new `Layer` sub-class.

For example, here (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/neural_machine_translation_with_transformer.ipynb#s is the code defining some new `Layer` types that will be used to create a `Transformer` layer type.

The output of `Dense(10)` is a Tensor with final dimension equal to the number of units (e.g., 10)

- The Tensor has leading dimensions too
    - e.g., the implicit "batch index" dimension
    - since the layer takes a mini-batch of examples (rather than a single example) as input
- It may have *additional* dimensions too !
    - Just like `numpy`: threading over additional dimensions
    - e.g., if input is shape $(\text{minibatch\_size} \times n_1 \times n_2)$
        - output is shape $(\text{minibatch\_size} \times n_1 \times 10)$
        - `Dense` operates over the final dimension

Studying advanced models

The best way to learn is to study the code of some non-trivial models

Factor Models and Autoencoders

Here is an example of a Functional model applied to a common problem in Finance.

- Functional model
- Threading

We will cover the Finance aspects of this in a separate module (Autoencoder_for_conditional_risk_factors.ipynb)

For now, I want to focus on the idea and the code

Here is the code, excerpted from the notebook (https://github.com/stefan-jansen/machine-learning-for-trading/blob/main/20_autoencoders_for_conditional_risk_factors/06_conditional_autoenc

```
def make_model(hidden_units=8, n_factors=3): input_beta = Input((n_tickers, n_characteristics),
name='input_beta') input_factor = Input((n_tickers,), name='input_factor') hidden_layer =
Dense(units=hidden_units, activation='relu', name='hidden_layer')(input_beta) batch_norm =
BatchNormalization(name='batch_norm')(hidden_layer) output_beta = Dense(units=n_factors,
name='output_beta')(batch_norm) output_factor = Dense(units=n_factors, name='output_factor')
(input_factor) output = Dot(axes=(2,1), name='output_layer')([output_beta, output_factor]) model =
Model(inputs=[input_beta, input_factor], outputs=output) model.compile(loss='mse', optimizer='adam')
return model
```

Autoencoder: Functional model

[Autoencoder example from github (https://colab.research.google.com/github/kenperry-public/ML_Spring_2022/blob/master/Autoencoder_example.ipynb)](https://colab.research.google.com/github/kenperry-public/ML_Spring_2022/blob/master/Autoencoder_example.ipynb)

- Functional model

Issues

- We could use a Sequential model with initial Encoder layers and final Decoder layers
  - But we would not be able to independently access the Encoder nor the Decoder as isolated models

VAE: Complex Loss; Manual Gradient updates

[Variational Autoencoder (VAE) from github (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=DEU05Oe0vJrY)](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=DEU05Oe0vJrY)

- Functional model
- [VAE: Custom train step (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=0EHkZ1WCHw9E)](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=0EHkZ1WCHw9E)
    - Complex loss

Issues

Transformer: Custom layers, Skip connections, Layer Norm

[Transformer layer (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/neural_machine_translation_with_transformer.ipynb#s
IMkSs)]

- Functional model
- Custom layers
- Layer Norm
- Skip connections

The following diagram shows the architecture, which we can compare to the code

- [Full architecture diagram (compare with code) (Transformer.ipynb#Full-Encoder-Decoder-Transformer-architecture)]

We can dig deeper to examine how the Attention layers are implemented in code:

- [Scaled dot-product attention (https://www.tensorflow.org/text/tutorials/transformer#scaled_dot_product_atten]
- [Multi-head attention (https://www.tensorflow.org/text/tutorials/transformer#mult
head_attention)]

Issues

- Build a new layer type
- Why are the components layers (e.g., Dense, MultiHeadAttention, LayerNormalization) instantiated in the class constructor
    - As opposed to being defined in the "call" method
    - Because we **need** one instance of the layer
        - Not a new instance each time the class is "called" per batch
            - This would result in brand new weights for each example batch
    - The "call" method accesses the shared layer instances and performs the computation using them

---

The Gradient Tape: Visualizing what CNN's learn

[Visualizing what Convnets learn (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/vision/ipynb/visualizing_what_convnets_learn.ipynb#scrollTo=K8

- The Gradient Tape
- Maximize utility (negative loss)
    - mean (across the spatial dimensions) of one feature map in a multi-layer CNN
    - the "weights" being solved for are the pixels of the input image !

[Gradient ascent (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/vision/ipynb/visualizing_what_convnets_learn.ipynb#scrollTo=a9

GAN

[Simple GAN (https://keras.io/examples/generative/dcgan_overriding_train_step)](https://keras.io/examples/generative/dcgan_overriding_train_step)

- [Custom train step: GAN training (https://keras.io/examples/generative/dcgan_overriding_train_step/#override-trainstep)](https://keras.io/examples/generative/dcgan_overriding_train_step/#override-trainstep)

Wasserstein GAN with Gradient Penalty

[Wasserstein GAN with Gradient Penalty (https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)](https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)

- [Gradient Tape: used for loss term, rather than weight update (https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)](https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)
- [Overide `compile` (https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)](https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)
- [Custom train step: GAN training (https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)](https://keras.io/examples/generative/wgan_gp/#create-the-wgangp-model)

Neural Style Transfer

Neural Style Transfer (https://keras.io/examples/generative/neural_style_transfer/)

- Complex Loss]((https://keras.io/examples/generative/neural_style_transfer/#compute-the-style-transfer-loss))
- Custom training loop
- Feature extractor (https://keras.io/examples/generative/neural_style_transfer/#compute-the-style-transfer-loss)

Here (https://www.tensorflow.org/tutorials/generative/style_transfer) is a tutorial view of the notebook.

In [2]:

```
print("Done")
```

Done