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, illustrated by the Transformer

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 (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/neural_machine_translation_with_transformer.ipynb#s we will visit !

Let's examine each one and try to relate the actual code to our picture.

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

Transformer Layer (Encoder/Decoder)

First model: the Encoder side of the Transformer

The Encoder side of the transformer:

```
encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="encoder_input
s")
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 a graph of function calls that maps `Model` inputs to outputs.

To turn this collection into a `Model`

- We define which function to feed Model inputs to
- We define which function's outputs are the output of 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`

- The output sequence $\bar{\mathbf{y}}_{(1..\bar{T})}$ (i.e., latent states) of the Encoder
    - Used in 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.

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

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

Let's examine the Encoder code more closely

```
encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="encoder_input
s")
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)
```

First, observe that the *Encoder* outputs ($\bar{\mathbf{y}}_{(1..\bar{T})}$) `encoder_outputs`

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

We see that these Encoder outputs become the second part of the pair of the actual arguments that are the inputs to the *Decoder*

```
decoder_outputs = decoder([decoder_inputs, encoder_outputs])
```

And the formal argument of `decoder` definition is `encoded_seq_inputs`

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

So the `encoder_outputs` ($\bar{\mathbf{y}}_{(1..\bar{T})}$) become bound to `encoded_seq_inputs` within the Decoder.

Hence, the second part of the input pair of `decoder` serves the role of $\bar{\mathbf{y}}_{(1..\bar{T})}$, the sequence of Encoder latent states

Second model: The Decoder side of the Transformer

Now, let's look at the Decoder.

```
decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="decoder_input
s")
encoded_seq_inputs = keras.Input(shape=(None, embed_dim), name="decoder_state_i
nputs")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, encoded_seq_inputs)
x = layers.Dropout(0.5)(x)
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
decoder = keras.Model([decoder_inputs, encoded_seq_inputs], decoder_outputs)
```

By tracing variable `x` backwards from the Decoder output ($\mathbf{y}_{(1..T)}$) `decoder_outputs`

```
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
```

we can see the outputs derive from `Layer` sub-type `TransformerDecoder`

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

We have already shown that `encoded_seq_inputs` corresponds to $\bar{\mathbf{y}}_{(1..\bar{T})}$

So we can guess that variable `x` in the call to `TransformerDecoder` corresponds to $\mathbf{y}_{(1..T)}$

Let's confirm that.

Tracing variable `x` backwards from its use a the first argument in the `TransformerDecoder`

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

we see that it is the positionally-encoded (to enable Causal masking) `decoder_inputs`

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

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

- That is: the training set enforces "Teacher Forcing"

Third model: the full Transformer – Encoder + Decoder

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` input is a pair

```
[encoder_inputs, decoder_inputs]
```

And it's output is

```
decoder_outputs
```

We identify the first part of the input pair (`encoder_inputs`) as the input sequence $\mathbf{x}_{(1 \ldots \bar{T})}$

Summary

A lot going on here !

- 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
- Hopefully:
    - `decoder_inputs` is equal to `decoder_outputs` shifted by one time step
    - Teacher forcing, enforced by the organization of the training data ?

That concludes are first look at the Functional model

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

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

Transformer: Custom layers, Skip connections, Layer Norm

We have already seen part of the Transformer in introducing the basics of the Functional model.

We use the rest of this example to discover other advanced Keras techniques:

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

- Custom layers (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/neural_machine_translation_with_transformer.i
- Layer Norm
- Skip connections

Custom layers: subtle point

Let's look at the constructor for the `TransformerEncoder` custom layer, as an example

```
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"), layers.Dense(embed_di
m),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.supports_masking = True
```

The custom layer consists of a collection of component layers

Why are the components layers (e.g., Dense, MultiHeadAttention, LayerNormalization) instantiated in the class constructor

- As opposed to being defined in the `call` method

Had we instantiated each component within the `call` method

- There would be a *new instance* of each component *each time the layer was called on an example* in training !
- Each instance would have *it's own weights*
- So training would not "learn" between examples

Other custom layers of interest

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)

VAE: Custom Model – Training Loop, the Gradient Tape

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

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

We use this example to illustrate

- How to sub-class the `Model` abstract class
- How to create a custom training loop
    - the "Gradient tape"

Issues

- Custom **model** (not layer) class VAE
- The *reconstruction loss* depends on the output of the Decoder part of the VAE
    - No other obvious way to define this loss aside from a **custom training** step
- Because we are computing the Loss in the training step
    - we must compute the gradient of the Loss w.r.t weights
    - Update the weights (**gradient tape**)

Visualizing what CNN's learn: Gradient Ascent and the Gradient Tape

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=K

- 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 !

We use this example to show how powerful the Gradient Tape is

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

Factor Models and Autoencoders: Threading

We use this example to show

- A Functional model applied to a common problem in Finance.
- 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_la
yer')(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
```

Not obvious what is going on here.

A picture will help:

Threading

Let's focus on the `Dense` layer corresponding to the box labelled "Beta" in the picture

- `Dense` (n_{factors})

From the diagram you will notice that

- the input to this layer is *two dimensional*: $(n_{\text{tickers}} \times n_{\text{chars}})$
- the output to this is *two dimensional*: $(n_{\text{tickers}} \times n_{\text{factors}})$

We have not yet seen multi-dimensional input/output in regard to a `Dense` layer

What is going on here ?

The layer is implementing a function with signature

- `Dense(`n_{factors}`)`
  $$:$$
  $$(n_{\text{tickers}}$$
  $$\times n_{\text{chars}})$$
  $$\mapsto$$
  $$(n_{\text{tickers}}$$
  $$\times n_{\text{factors}})$$

Tensorflow/Keras works on higher dimensional objects just like NumPy:

- threading (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense) over "extra" dimensions

If the input to layer l is shape $(d_{(l),1} \times d_{(l),2} \times \ldots d_{(l),N} \times n_{(l)})$

- And the layer type operates over a *single* dimension (usually the last dimension)
  - producing output shape $n_{(l+1)}$

Then threading treats the inputs

In our case

- Input shape is $\left(n_{\text{tickers}} \times n_{\text{chars}}\right)$
- The `Dense` layer is defined with n_{factors} units ($n_{(l+1)} = n_{\text{factors}}$)
- Hence, the output shape is $\left(n_{\text{tickers}} \times n_{\text{factors}}\right)$

The weight matrix for this layer

- $\mathbf{W}_{\beta}$ with shape $\left(n_{\text{factors}} \times n_{\text{chars}}\right)$
    - just like any `Dense` layer; number of weights is *independent* of threading
- applies the *same weights* to each of the n_{tickers} (the rows) of the input

Neural Style Transfer: Feature extractor, Training Loop

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)

Do you remember the Neural Style Transfer task

- that we used to preview the concept that Deep Learning is all about defining a Loss Function that captures the semantics of the task ?

That is, the task is

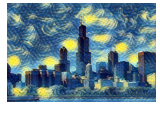
- Given
    - a Style image

    - and a Content image

- Generate an image that is the Content image re-drawn in the "style" of the Style image

We use this example to illustrate

- Complex Loss and Custom Training Loop
- Feature extractor (https://keras.io/examples/generative/neural_style_transfer/#compute-the-style-transfer-loss)

Content Loss and Style Loss

The objective of Neural Style Transfer:

- Given Content Image C
- Give Style Image S
- Create Generated Image G
- Minimizing

$$\mathcal{L} = \mathcal{L}_{\text{content}} + \mathcal{L}_{\text{style}}$$

- where
    - $\mathcal{L}_{\text{content}}$ measures the dissimilarity of the "content" of G and "content" f C
    - $\mathcal{L}_{\text{style}}$ measures the dissimilarity of the "style" of G and "style" of C

How do we measure the dissimilarity of the "content" ?

We can't just use plain MSE of the pixel-wise differences

- G is different than C, by definition (the "styles" are different)

And how do we define what the "style" of an image is ?

Suppose we have an Image Classifier Neural Network $\mathbb{C}$ (e.g., VGG19).

Further suppose $\mathbb{C}$ consists of a sequence of CNN Layers

- Let $\mathbb{C}_{(l)}$ denote the set of $n_{(l)}$ feature maps produced at layer l
    - Feature map: value of one feature, at each spatial location

We choose

- one layer l_c of $\mathbb{C}$ and call it the "content representation" layer
    - Will tend to be shallow: closer to the input
    - Features of shallow layers will be more "syntax" than "semantics"
- one layer l_s of $\mathbb{C}$ and call it the "style representation" layer
    - Will tend to be deep: closer to the output
        - Features of deep layers will be more "semantics" than "syntax"

For arbitrary image I, let

- $\mathbb{C}_{(l_c)}(I)$
  - denote the feature maps of the Classifier $\mathbb{C}$, on image I, at the "content representation" layer
- $\mathbb{C}_{(l_s)}(I)$
  - denote the feature maps of the Classifier $\mathbb{C}$, on image I, at the "style representation" layer

We can now define the dissimilarity of the "content" of Content Image C and "content" f Generated Image G

- by comparing $\mathbb{C}_{(l_c)}(C)$ and $\mathbb{C}_{(l_c)}(G)$

Similarly, we can define the dissimilarity of the "style" of Content Image C and "style" of Generated Image G

- by comparing $\mathbb{C}_{(l_s)}(S)$ and $\mathbb{C}_{(l_s)}(G)$

For any image I: $\mathbb{C}_{(l)}(I)$ consists of $n_{(l)}$ layers.

We need to define what it means to compare $\mathbb{C}_{(l)}(I)$ and $\mathbb{C}_{(l)}(I')$.

The *Gramm Matrix* $\mathbb{G}$ of $\mathbb{C}_{(l)}(I)$

- Has shape ($n_{(l)} \times n_{(l)}$)
- $\mathbb{G}_{j,j'}(I) = \mathrm{correlation}(\mathrm{flatten}(\mathbb{C}_{(l),j}(I)),$
  $\mathrm{flatten}(\mathbb{C}_{(l),j'}(I)))$
    - the correlation of the feature map j of $\mathbb{C}_{(l)}(I)$ with feature map j' of $\mathbb{C}_{(l)}(I')$

Intuitively, the Gramm Matrix measures the correlation of the values across pixel locations (flattened feature maps) of two feature maps of image I

We can now define the dissimilarity of $\mathbb{C}_{(l)}(I)$ and $\mathbb{C}_{(l)}(I')$

- As the MSE
- of $\mathbb{G}(I)$ and $\mathbb{G}(I')$

Using this dissimilarity measure, we can define the

- $\mathcal{L}_{\text{content}}$ as the dissimilarity of $\mathbb{C}_{(l_c)}(C)$ and $\mathbb{C}_{(l_c)}(G)$
- $\mathcal{L}_{\text{style}}$ as the dissimilarity of $\mathbb{C}_{(l_s)}(S)$ and $\mathbb{C}_{(l_c)}(G)$

Gradient ascent: generating G

We can find image G via Gradient Ascent

- Initialize G to noise
- Update pixel $G_{i,i',k}$ by $-\frac{\partial \mathcal{L}}{G_{i,i',k}}$

Feature extractor

One key coding trick that we will illustrate

- Obtaining the feature maps of the Classifier $\mathbb{C}$, on image I, at an arbitrary layer

We will call this tool the *feature extractor*

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

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

GAN

Simple GAN (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)

Wasserstein GAN with Gradient Penalty

Wasserstein GAN with Gradient Penalty (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)
- Overide `compile` (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)

In [2]:
```
print("Done")
```

```
Done
```