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

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

initialize(W) # Training loop to implement mini-batch SGD for epoch in range(n_epochs):` for X_batch, y_batch in next_batch(X_train, y_train, batch_size, shuffle=True): # Forward pass y = NN(X_batch) # Loss calculation loss = loss_fn(y, y_batch) # Backward pass grads = gradient(loss, W) # Update W = W - grads * learning_rate

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

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#s
IMkSs)

- 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:

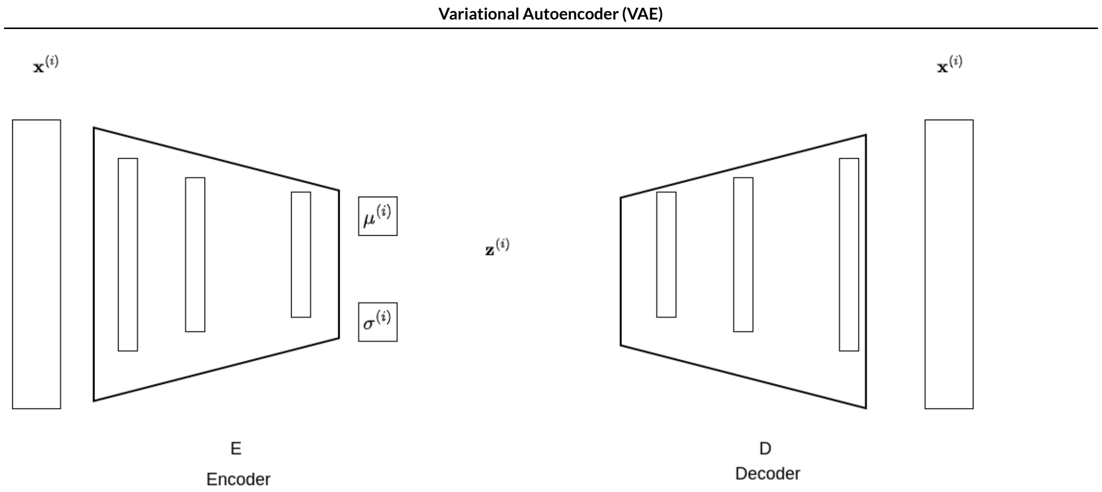
- Scaled dot-product attention (https://www.tensorflow.org/text/tutorials/transformer#scaled_dot_product_atten
- Multi-head attention (https://www.tensorflow.org/text/tutorials/transformer#multi-head_attention)

VAE: Custom Model – Custom Layer, Training Loop, the Gradient Tape

We use this example to show

- The Functional model
- A custom `Layer`: `Sampling`
- A custom training step
- Inverting a Convolution: `Conv2DTranspose`

Recall the architecture of a Variational Autoencoder (VAE)



A key step is drawing a random latent vector $\mathbf{z}^{(\mathbf{i})}$ from a distribution with mean μ and standard deviation σ.

This [cell (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=Wx_jzzcPtcfz)](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=Wx_jzzcPtcfz)

- Creates a custom `Layer` type called `Sampling` to perform the random sampling
- By overriding the base `Layer` `call` method

In this cell (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vae.ipynb#scrollTo=4rFiHtbCtcf0) we can see that the Encoder and Decoder are both implemented as Functional models

- The Encoder produces a *pair of outputs*: (μ, σ)

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**)

[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)

- [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

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=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 !

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=a9

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)](Autoencoder_for_conditional_risk_factors.ipynb)

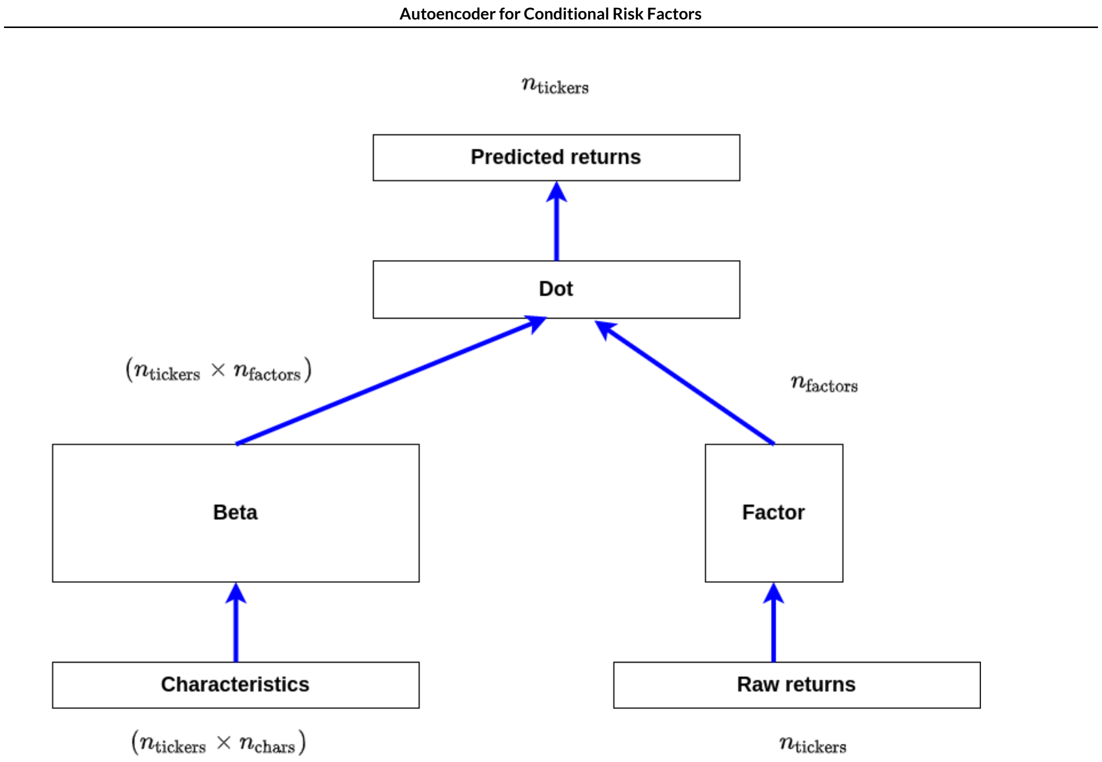
For now, I want to focus on the idea and the code

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

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

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

The weight matrix for this layer

- $\mathbf{W}_\beta$ with shape $(n_{\text{factors}} \times n_{\text{chars}})$
    - 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

We use this example to illustrate

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

Autoencoder: Functional model

[Autoencoder example from github (https://colab.research.google.com/github/kenperry-public/ML_Spring_2024/blob/master/Autoencoder_example.ipynb)](https://colab.research.google.com/github/kenperry-public/ML_Spring_2024/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

We use this example to show

- A custom training step
- Inverting a Convolution: `Conv2DTranspose`

Recall: the training of a GAN is an iterative process among two "players"

- the Discriminator
- the Generator

Custom `train_step`

Here is a summary from our introductory module on [GANs (GAN_Overview.ipynb)](GAN_Overview.ipynb)

Competitive training

Iteration t

- Train $D_{\Theta_{D,(t-1)}}$ on samples
  - $\tilde{\mathbf{x}} \in p_{\text{data}} \cup p_{\text{model},(t-1)}$
    - where $G_{\Theta_{G,(t-1)}}(\mathbf{z}) \in p_{\text{model},(t-1)}$
  - Update $\Theta_{D,(t-1)}$ to $\Theta_{D,(t)}$ via gradient $\frac{\partial \mathcal{L}_D}{\partial \Theta_{D,(t-1)}}$
    - D is a maximizer of $\int_{\mathbf{x} \in p_{\text{data}}} \log D(\mathbf{x}) + \int_{\mathbf{z} \in p_{\mathbf{z}}} \log (\, 1 - D(G(\mathbf{z}))\,)$
- Train $G_{\Theta_{G,(t-1)}}$ on random samples $\mathbf{z}$
  - Create samples $\hat{\mathbf{x}}_{(t)} \in G_{\Theta_{G,(t-1)}}(\mathbf{z}) \in p_{\text{model}}$
  - Have Discriminator $D_{\Theta_{D,(t)}}$ evaluate $D_{\Theta_{D,(t)}}(\hat{\mathbf{x}}_{(t)})$
  - Update $\Theta_{G,(t-1)}$ to $\Theta_{G,(t)}$ via gradient $\frac{\partial \mathcal{L}_G}{\partial \Theta_{G,(t-1)}}$
    - G is a minimizer of $\int_{\mathbf{z} \in p_{\mathbf{z}}} \log(\, 1 - D(G(\mathbf{z}))\,)$
      - i.e., want $D(G(\mathbf{z}))$ to be high
  - May update G multiple times per update of D

In Keras, one can override a `Model`'s `train_step` method in order to replace the treatment of a single mini-batch of examples.

Let's see how this is applied to a [simple GAN (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/dcgan_overriding_train_step.ipynb#scrollTo=O](https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/dcgan_overriding_train_step.ipynb#scrollTo=O)

Key points to observe;

- Discriminator trained first
    - Create examples from real and fake images, to be fed to Discriminator

```
# Decode them to fake images
generated_images = self.generator(random_latent_vectors)

# Combine them with real images
combined_images = tf.concat([generated_images, real_images], axi
s=0)
```

- Train the Discriminator on the combined real/fake images and update it's weights

```
        # Train the discriminator
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weight
s)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )
```

- Train the Generator
    - Have it create fake images from random, latent vectors
    - Let the Discriminator evaluate these fakes
    - Update Generator weights to better be able to fool Discriminator

```
    # Assemble labels that say "all real images"
    misleading_labels = tf.zeros((batch_size, 1))

    # Train the generator (note that we should *not* update the weights
    # of the discriminator)!
    with tf.GradientTape() as tape:
        predictions = self.discriminator(self.generator(random_latent_vectors))
        g_loss = self.loss_fn(misleading_labels, predictions)
    grads = tape.gradient(g_loss, self.generator.trainable_weights)
    self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weight
s))
```

`Conv2DTranspose`: Inverting a Convolution

A CNN layer

- creates new features, for each element of the spatial dimension of the layer input
- May "down-sample" the input (reduce spatial dimension)
    - Using a stride greater than 1

We can invert the Convolution, and "up-sample" (increase the spatial dimension)

- with the `Conv2DTranspose` layer type

- We can see the Discriminator using Convolutional layers with down-sampling
- And the Generator using transposed Convolutional layers with up-sampling

in this cell (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/dcgan_overriding_train_step.ipynb#scrollTo=O

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)](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)

In [2]:

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

```
Done
```