TensorFlow 2.0 Tutorial: Part #2

High-Level APIs (Sequential, Functional, and Model Subclassing) and more!

Iran University of Science and Technology (IUST)
Department of Computer Engineering

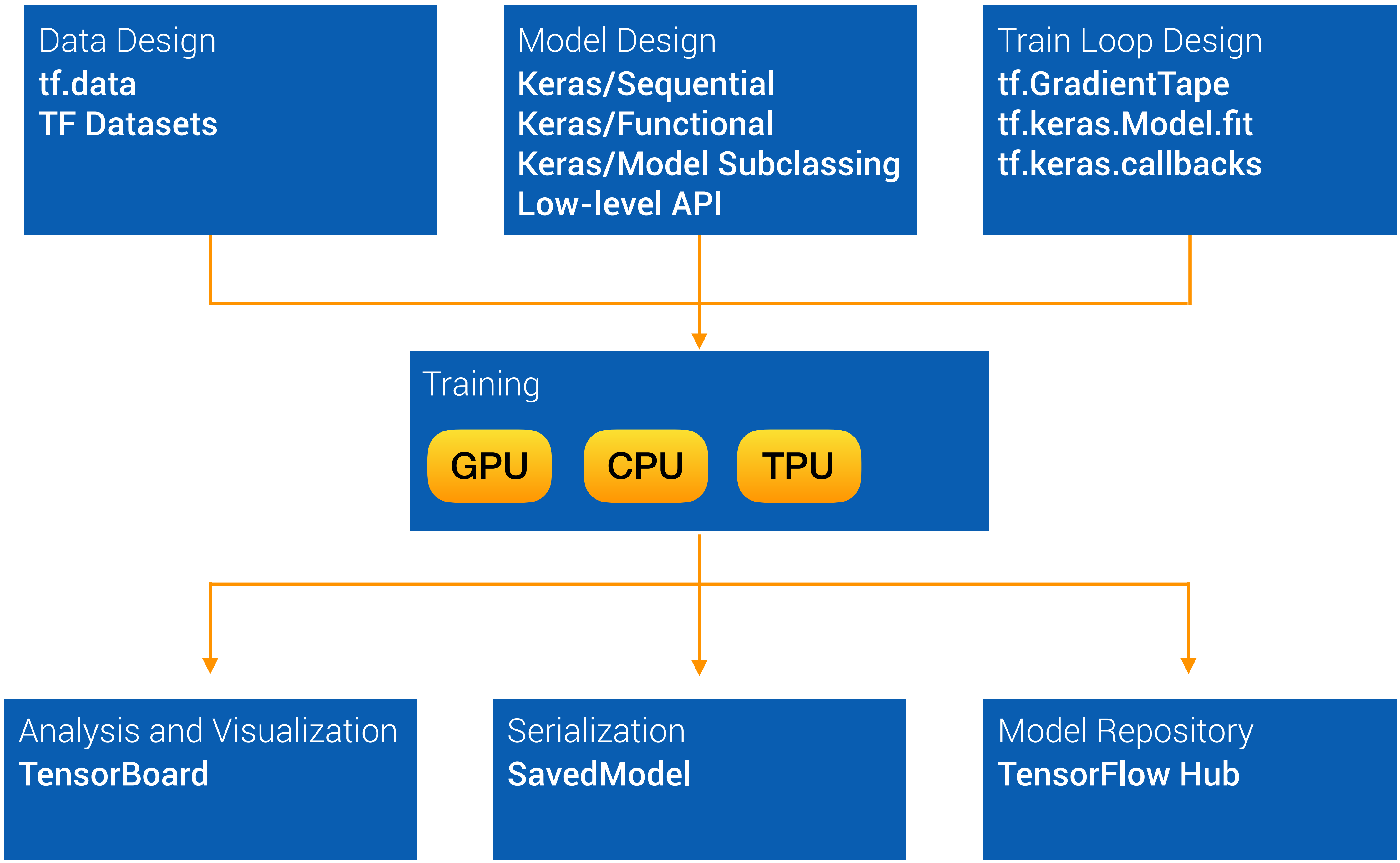
Notebook URL:

https://github.com/kazemnejad/tensorflow-2-tutorial/blob/master/part_02.ipynb

Slides URL:

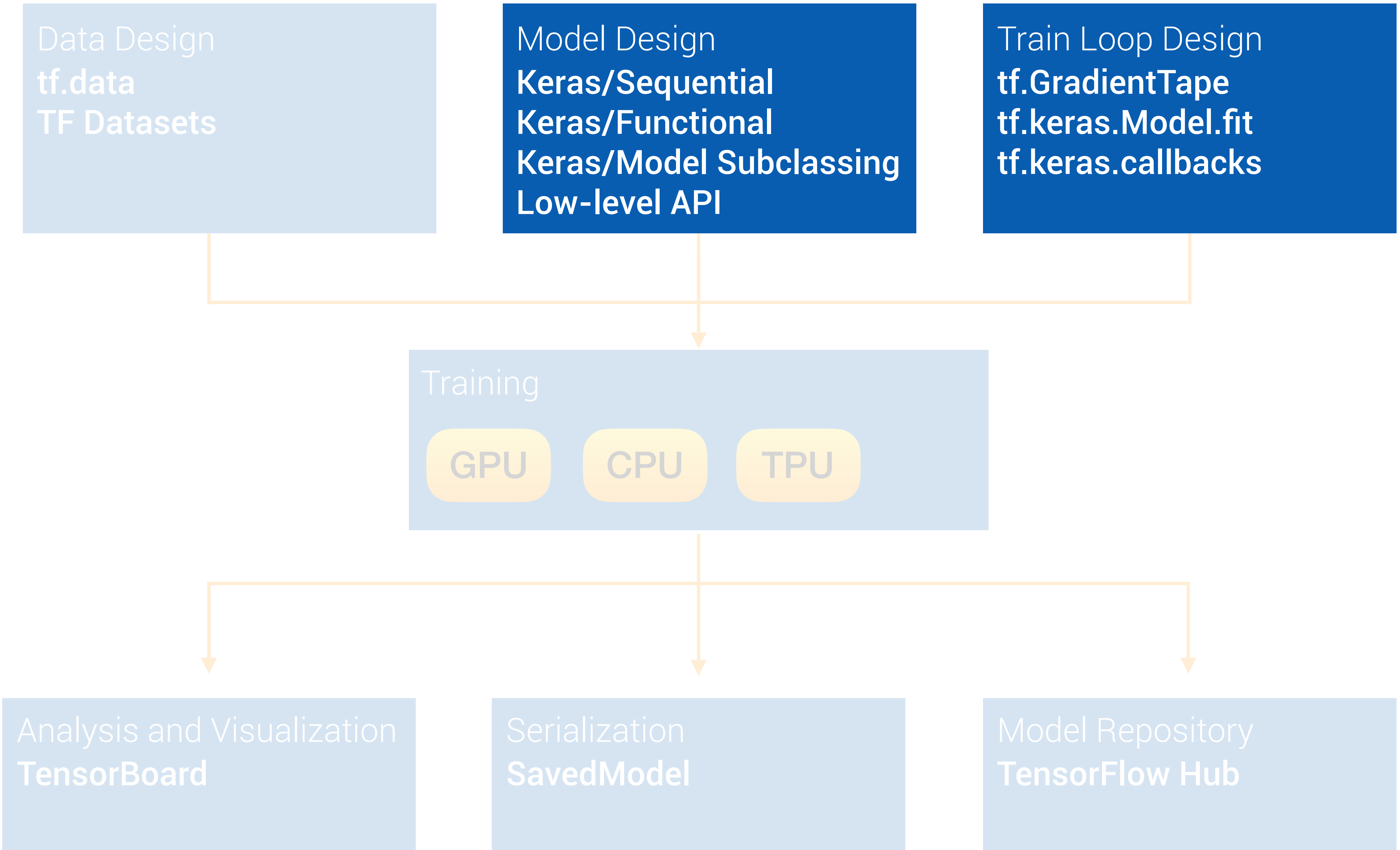
https://github.com/kazemnejad/tensorflow-2-tutorial/blob/master/part_02_slides.pdf

TensorFlow Overview



[source]

TensorFlow Overview



Package

keras.* vs **tf.keras.***

- tf.keras is a re-implementation of the Keras API.
- tf.keras has better Integration with rest of the framework.
- Distributed training is much easier in tf.keras.
- tf.keras supports Eager execution (dynamic graph).
- There is no one-to-one relation. However, most of the useful stuffs are also present in TensorFlow.

Model Design

Keras **Sequential** API
 + standard layers
 + custom layers, losses, and metrics

`Stack of layers`
`For Simple models`

TensorFlow Higher Level APIs

- Keras API (`tf.keras.*`)
  - Engine
    - **Base Layer**, **Base Model**, **Sequential**
  - Layers (various subclasses of Base Layer)
  - Losses, Metrics
  - Callbacks
  - Optimizers
  - Regularizes, Constraints

TensorFlow Higher Level APIs

- `tf.Module()` (Base neural network module class)
- Keras API (`tf.keras.*`)
  - Engine
    - **Base Layer**, **Base Model**, **Sequential**
  - Layers (various subclasses of Base Layer)
  - Losses, Metrics
  - Callbacks
  - Optimizers
  - Regularizes, Constraints

```python
import tensorflow as tf
from tensorflow.keras import layers


model = tf.keras.Sequential()
model.add(layers.Dense(32, activation=‘relu’, input_shape=(784,))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

```python
import tensorflow as tf
from tensorflow.keras import layers


model = tf.keras.Sequential([
    layers.Dense(32, activation='relu', input_shape=(784,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

```python
# Change the activation function (optional)
layers.Dense(64, activation='sigmoid')

# A linear layer: h = W.x + b

# Set the kernel (W) initializer. Default value: glorot_uniform
layers.Dense(64, kernel_initializer='orthogonal')
# Set the bias (b) initializer. Default value: zeros
layers.Dense(64, bias_initializer='random_uniform_initializer')

# Set the kernel regularizer
layers.Dense(64, kernel_regularizer=tf.keras.regularizers.l1(0.01))
# Set the bias regularizer
layers.Dense(64, bias_regularizer=tf.keras.regularizers.l2(0.01))
```

```python
# Change the activation function (optional)
layers.Dense(64, activation=tf.keras.activations.sigmoid)

# A linear layer: h = W.x + b

# Set the kernel (W) initializer. Default value: glorot_uniform
layers.Dense(64,
      kernel_initializer=tf.keras.initializers.GlorotUniform())
# Set the bias (b) initializer. Default value: zeros
layers.Dense(64,
      bias_initializer=tf.keras.initializers.RandomUniform())

# Set the kernel regularizer
layers.Dense(64, kernel_regularizer=tf.keras.regularizers.l1(0.01))
# Set the bias regularizer
layers.Dense(64, bias_regularizer=tf.keras.regularizers.l2(0.01))
```

What does a Layer do?

What does a Layer do?

- Computation from **a batch** of inputs to **a batch** of outputs.

What does a Layer do?

- Computation from **a batch** of inputs to **a batch** of outputs.
- Manages **state** (trainable **weights**, non-trainable weights).

What does a Layer do?

- Computation from **a batch** of inputs to **a batch** of outputs.
- Manages **state** (trainable **weights**, non-trainable weights).
- Tracks **losses** and **metrics.**

What does a Layer do?

- Computation from **a batch** of inputs to **a batch** of outputs.
- Manages **state** (trainable **weights**, non-trainable weights).
- Tracks **losses** and **metrics.**
- Automated compatibility checks (static **shape inference)**

What does a Layer do?

- Computation from **a batch** of inputs to **a batch** of outputs.
- Manages **state** (trainable **weights**, non-trainable weights).
- Tracks **losses** and **metrics.**
- Automated compatibility checks (static **shape inference**).
- Can be frozen (useful in **fine-tuning** and Transfer Learning).

What does a Layer do?

- Computation from **a batch** of inputs to **a batch** of outputs.
- Manages **state** (trainable **weights**, non-trainable weights).
- Tracks **losses** and **metrics.**
- Automated compatibility checks (static **shape inference**).
- Can be frozen (useful in **fine-tuning** and Transfer Learning).
- Can be serialized and deserialized (useful for storing the model).

[source]

What does a Layer do?

- Computation from **a batch** of inputs to **a batch** of outputs.
- Manages **state** (trainable **weights**, non-trainable weights).
- Tracks **losses** and **metrics.**
- Automated compatibility checks (static **shape inference**).
- Can be frozen (useful in **fine-tuning** and Transfer Learning).
- Can be serialized and deserialized (useful for storing the model).

[source]

Model Design

Keras **Sequential** API
+ standard layers

```
Stack of layers
For Simple models
```

Model Design

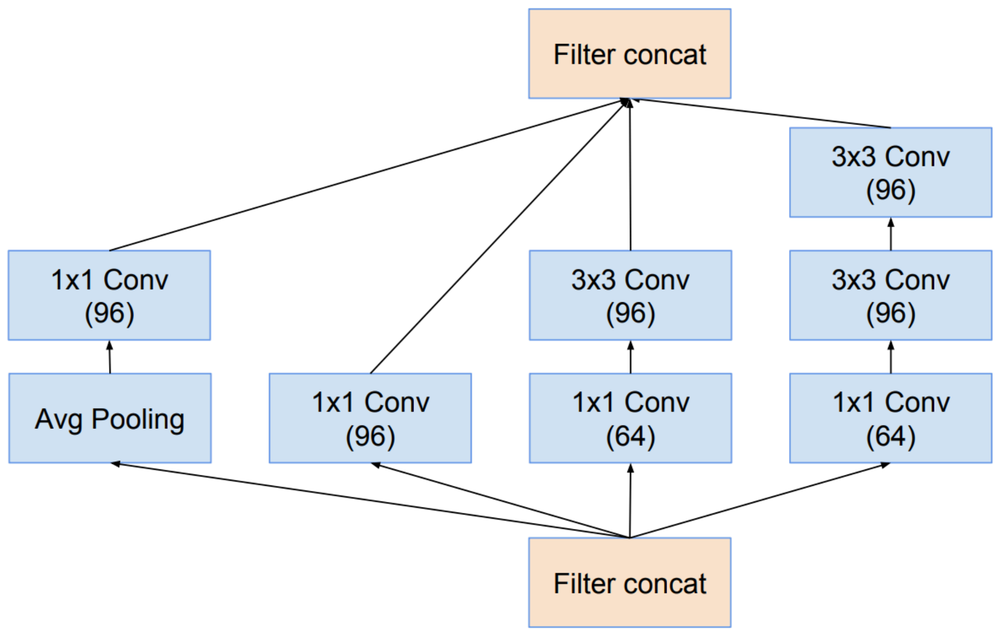
Keras **Sequential** API
+ standard layers

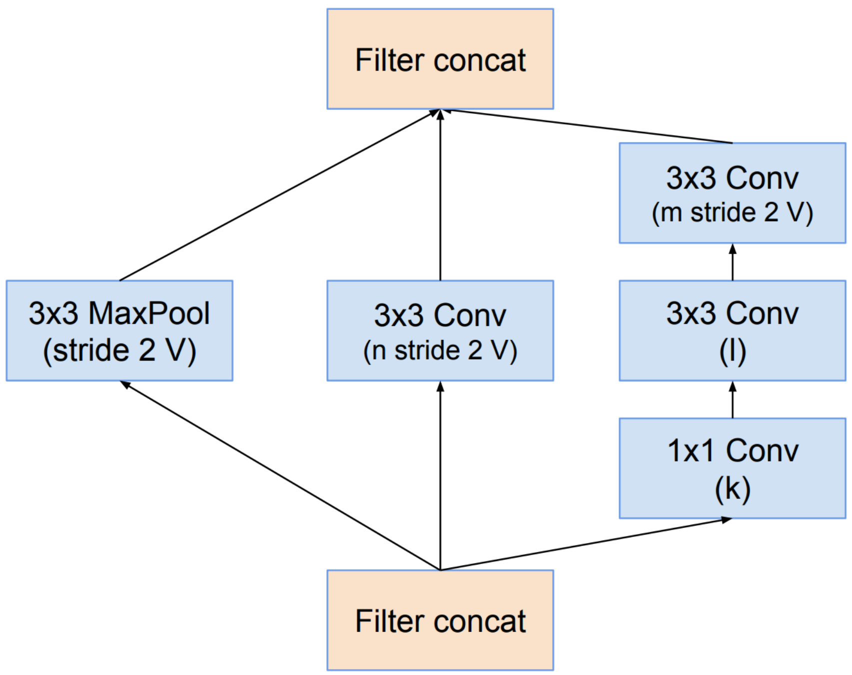
`Stack of layers`
`For Simple models`

Keras **Functional** API
+ standard layers

`**DAG** of layers`
`For Simple models`

Functional API (Creating a **DAG**)





Szegedy et al.

Functional API: A way to define DAGs of layers

```
import tensorflow as tf
from tensorflow.keras import layers


inputs = tf.keras.Input(shape=(784,))

x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Functional API: A way to define DAGs of layers

```
import tensorflow as tf
from tensorflow.keras import layers

inputs = tf.keras.Input(shape=(784,))

x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

You should first specify the model's input

Functional API: A way to define DAGs of layers

```
import tensorflow as tf
from tensorflow.keras import layers


inputs = tf.keras.Input(shape=(784,))

x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Then, define the model

Functional API: A way to define DAGs of layers

```
import tensorflow as tf
from tensorflow.keras import layers


inputs = tf.keras.Input(shape=(784,))

x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

And finally, build the model

Functional API: A way to define DAGs of layers

```python
import tensorflow as tf
from tensorflow.keras import layers


inputs = tf.keras.Input(shape=(784,))

x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Functional API: A way to define DAGs of layers

```
model.summary()
```

Functional API: A way to define DAGs of layers

```
model.summary()
```

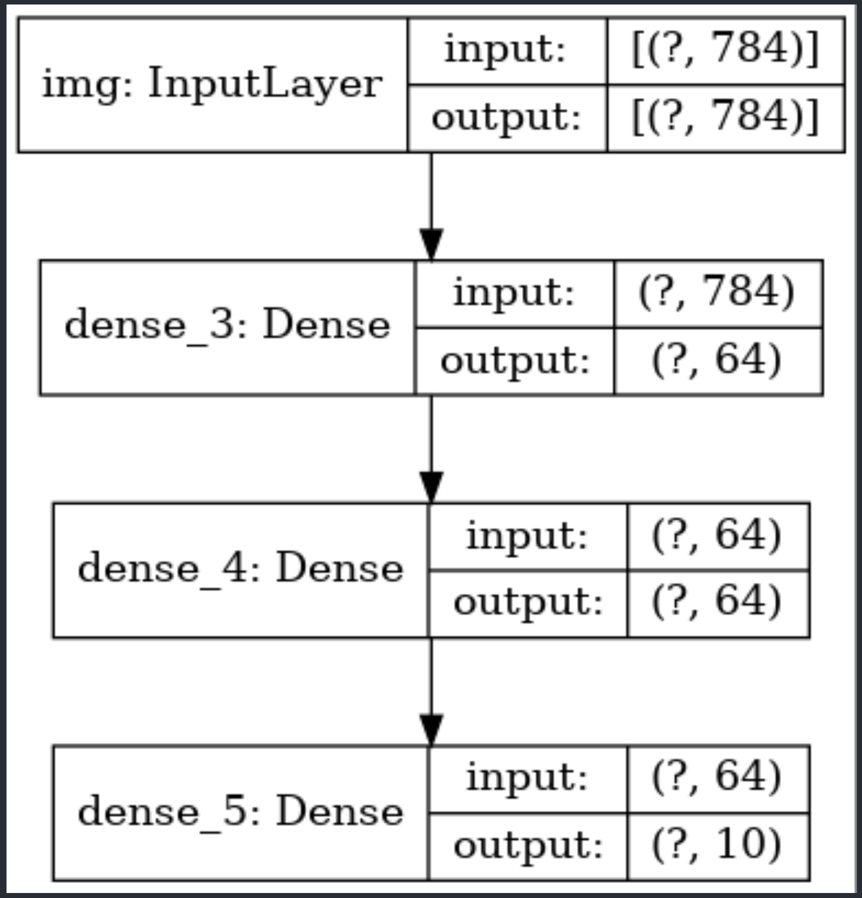
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img (InputLayer)             [(None, 784)]             0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_4 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_5 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

Functional API: A way to define DAGs of layers

```
keras.utils.plot_model(model, 'plot.png', show_shapes=True)
```

Functional API: A way to define DAGs of layers

```
keras.utils.plot_model(model, 'plot.png', show_shapes=True)
```



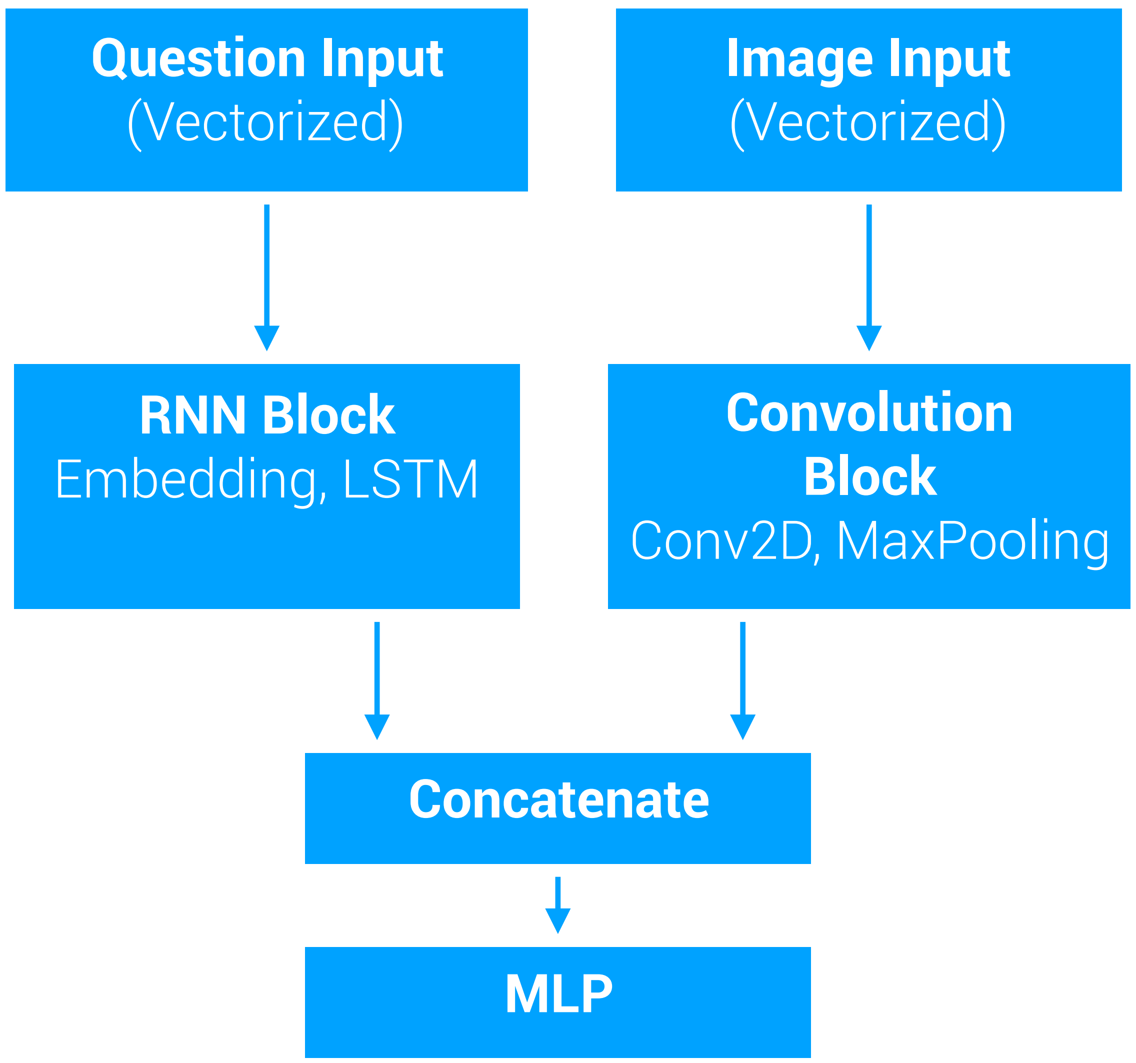
Example!
Visual Question Answering

Question:
What animal are these?

Answer:
Koala

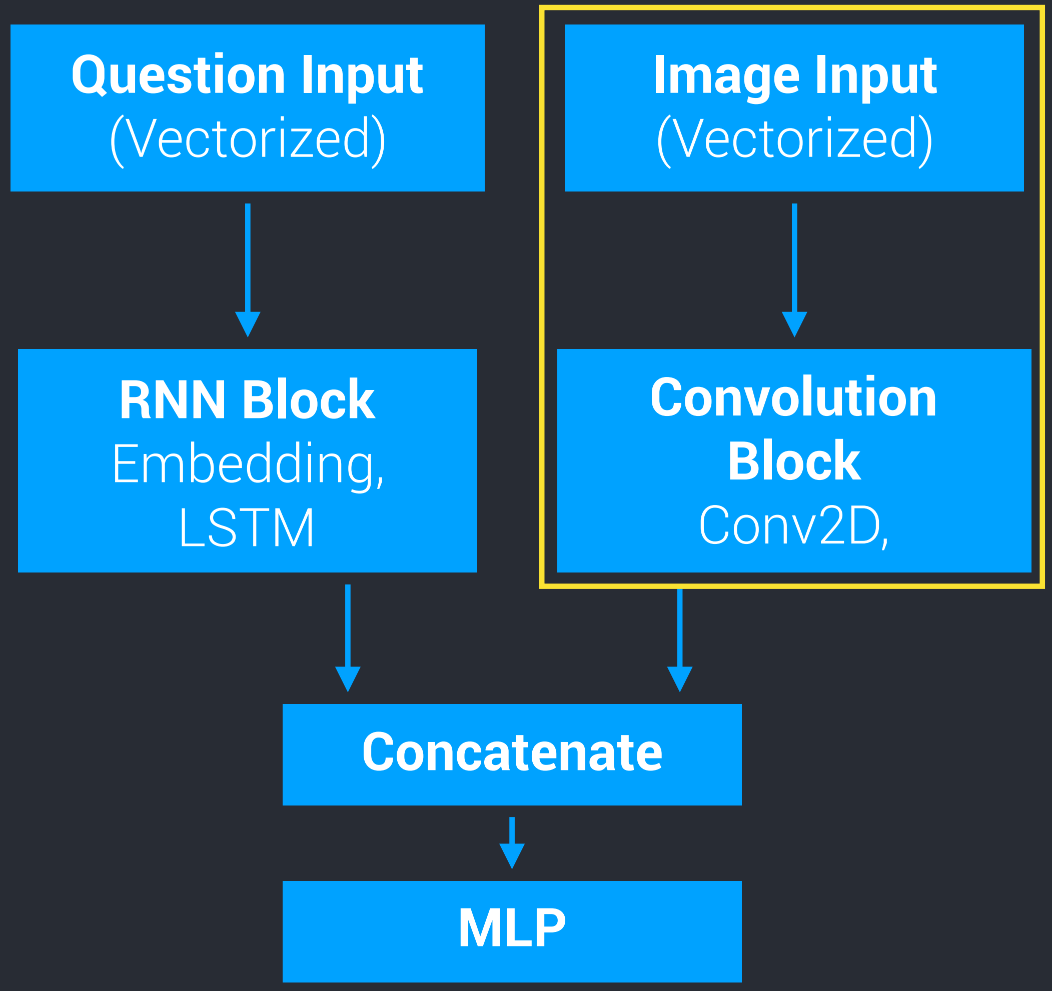
Photo by Adrian Pereira on Unsplash

Example!
Visual Question Answering



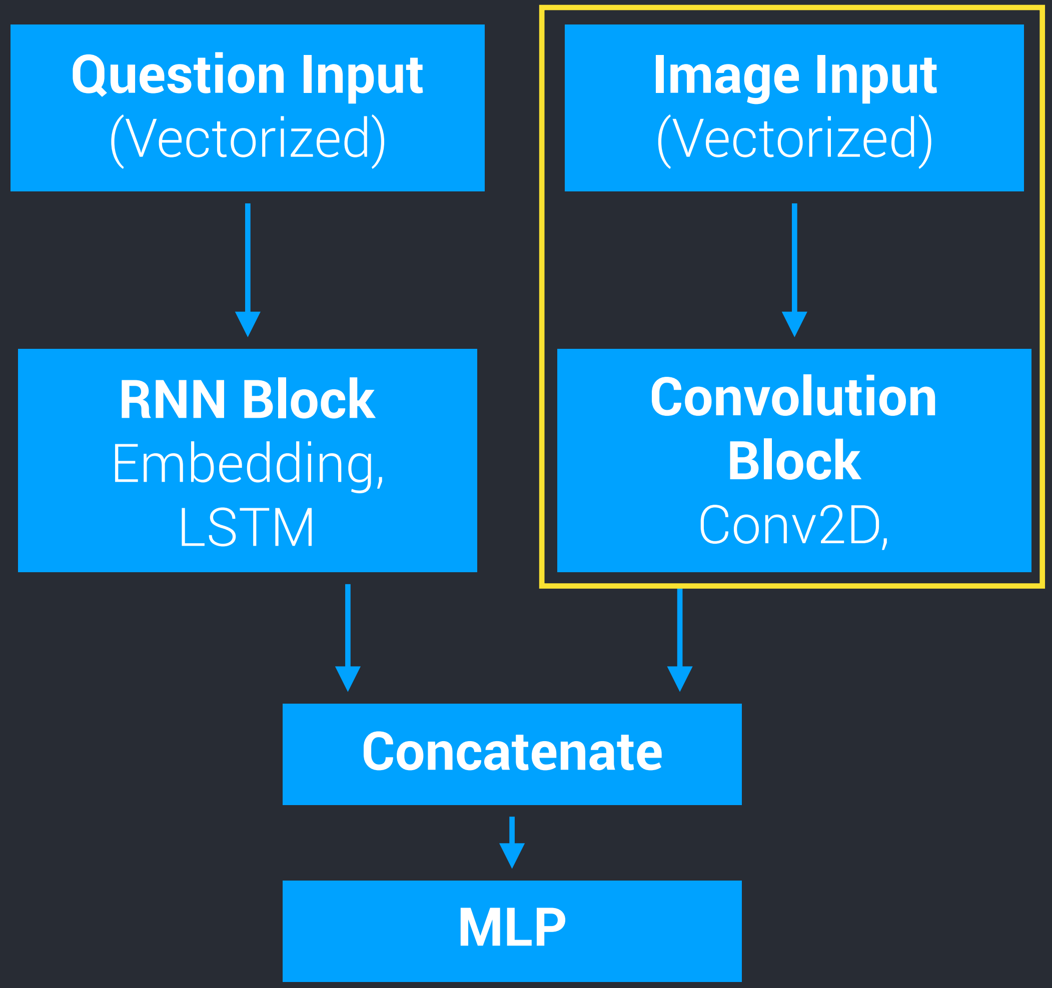
VQA Example!

```
# image input
image_input = Input(shape=(128, 128, 3))
```



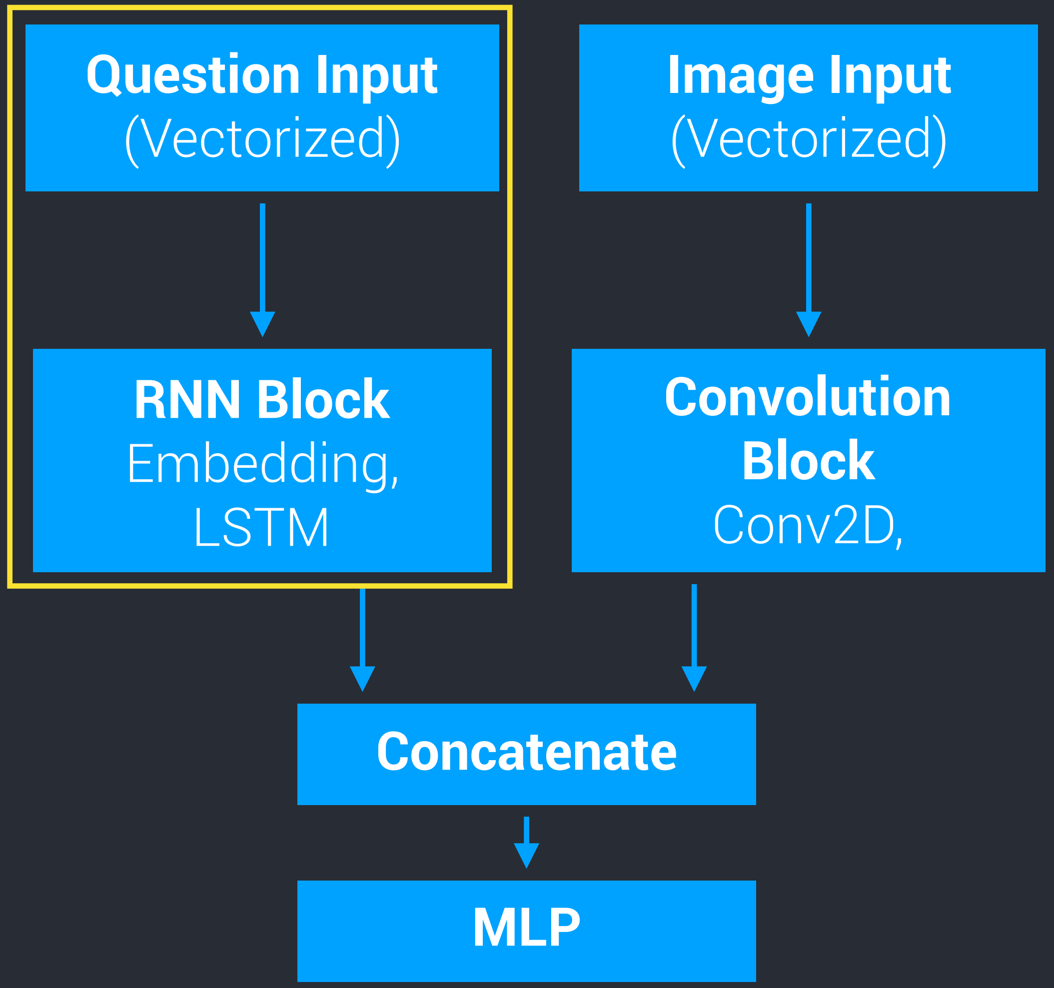
VQA Example!

```
# image input
image_input = Input(shape=(128, 128, 3))

# Encode the image into an abstract
# representation
encoded_image = Conv2D(64, (3, 3),
       activation='relu')(image_input)
encoded_image = MaxPooling2D()(encoded_image)
encoded_image = Flatten()(encoded_image)
```



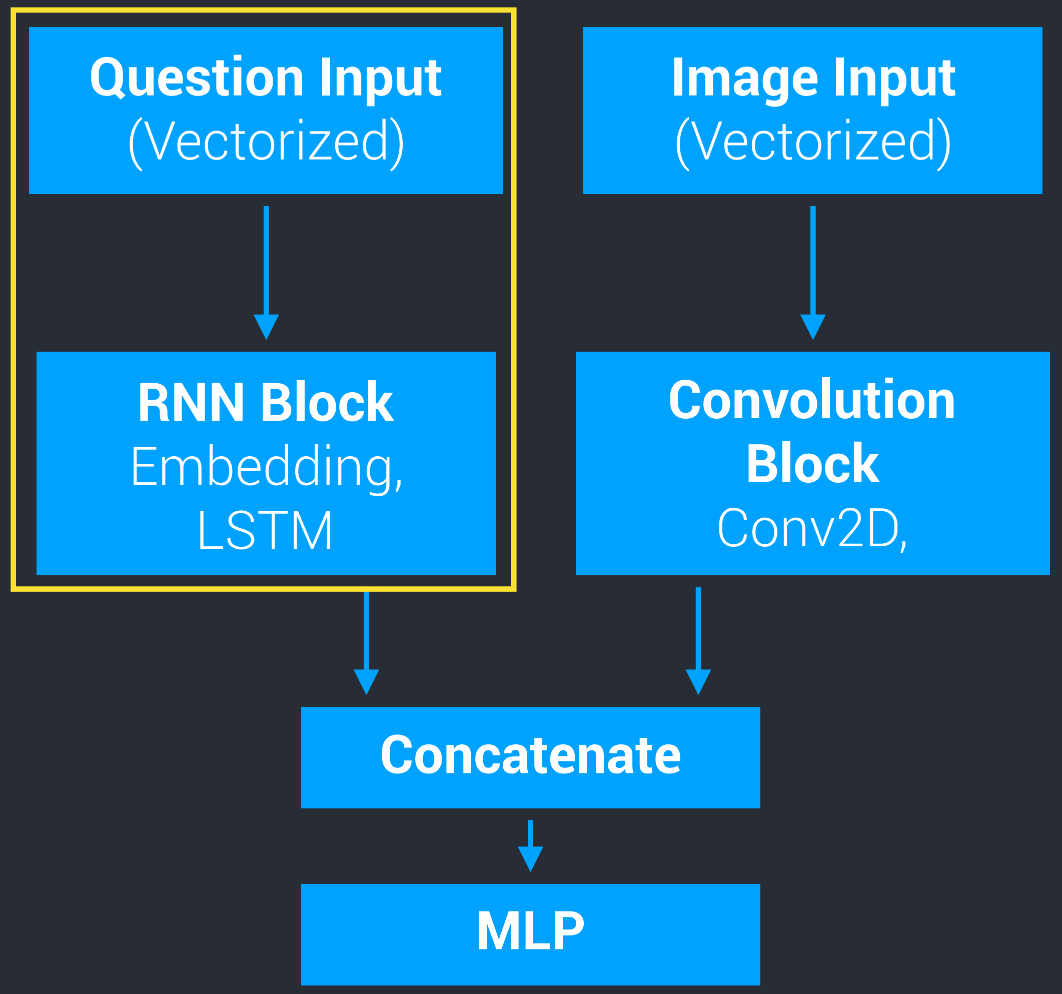
VQA Example!

```
# Vectorized input question
question_input = Input(shape=(None,),
                       dtype='int32')
```



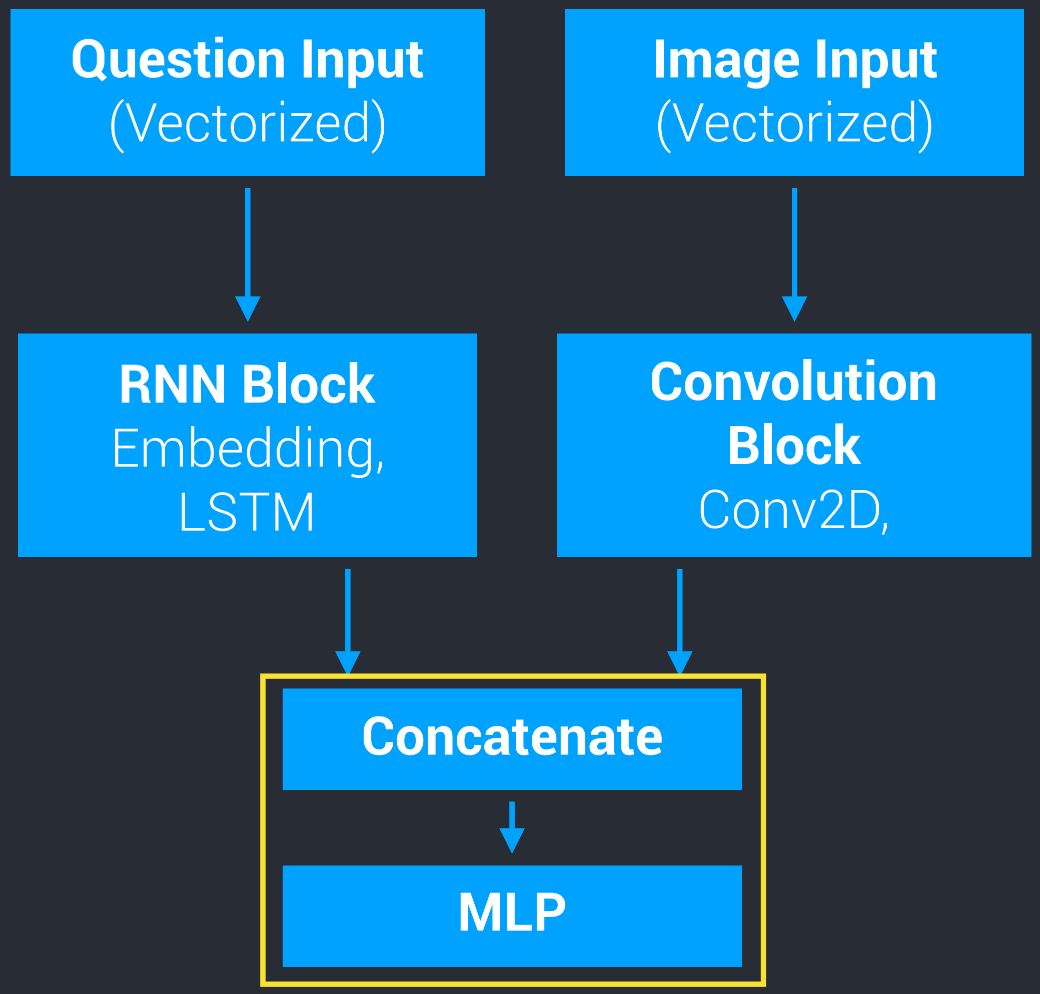
VQA Example!

```
# Vectorized input question
question_input = Input(shape=(None,),
                       dtype='int32')

# Encode the question into a single-
# vector representation
embedded = Embedding(
    input_dim=5000,
    output_dim=128,
    mask_zero=True)(question_input)

encoded_question = LSTM(128)(embedded)
```



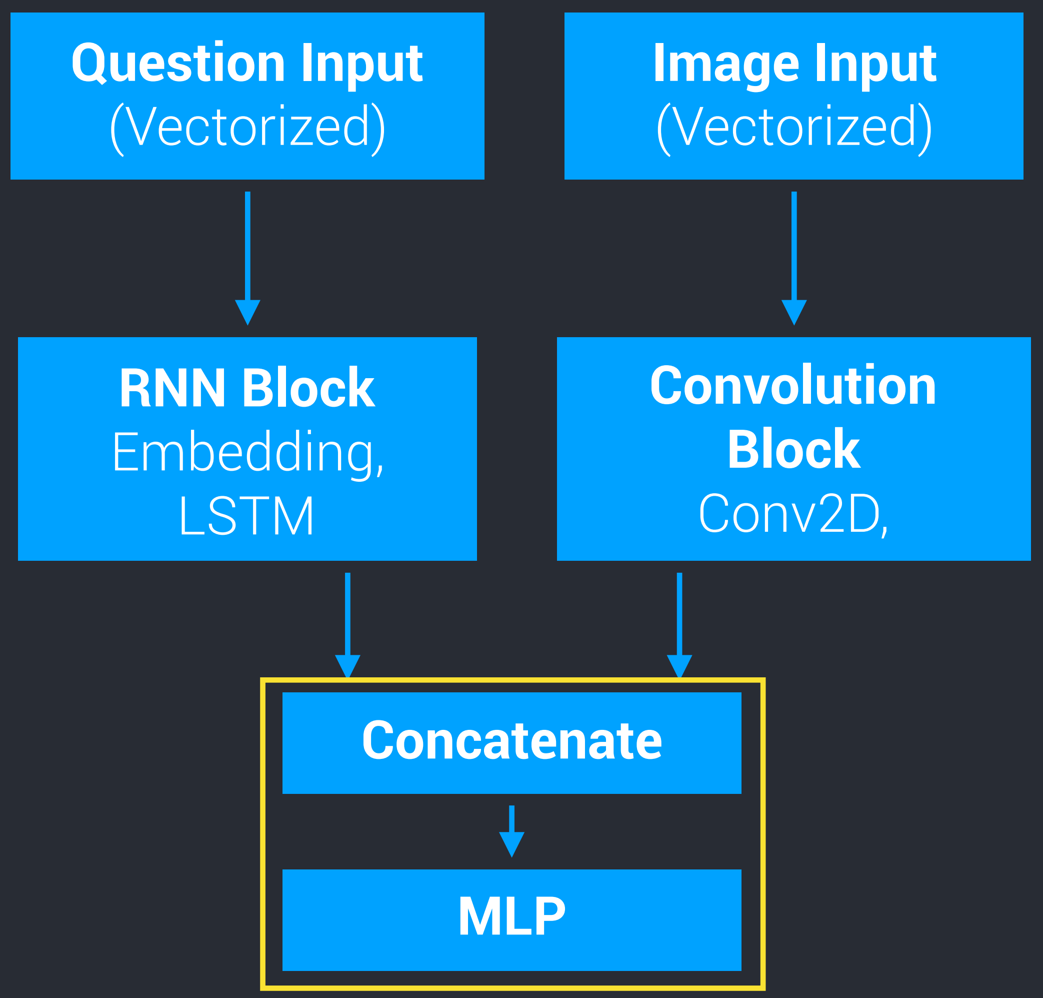
VQA Example!

```
# Concat the vector representations
merged = layers.concatenate([
    encoded_image, encoded_question])
```



VQA Example!

```python
# Concat the vector representations
merged = layers.concatenate([
    encoded_image, encoded_question])

# Use an MLP to produce the output
output = Dense(1000,
    activation='softmax')(merged)
```



Quiz #1: Product Review Classifier

Suppose that we have an online store (e.g., Amazon), and users can put a comment on products if they have bought them. Then, we want to find 1) whether the user would like to recommend the product 2) the sentiment of that review. Your model is given the title, the body, and the category of the review.
Here are the details of inputs and outputs:

Inputs

- **Title:** Vectorized & padded input (can consist of multiple word)
- **Body**: Vectorized & padded review content
- **Product Category**: one category out of 12 (one-hot representation)

Output

- **Sentiment score:** 5 possibilities
- **Recommend:** Wether the user recommends the product

Photo by Adrian Pereira on Unsplash

Writing Custom Layers

Custom Layer Outline

```
class MyLayer(layers.Layer):
    def __init__(self, arg1,arg2, ...):
        super(Linear, self).__init__()
        ...

    def build(self, input_shape):
        ...

    def compute_output_shape(self, input_shape):
        ...

    def compute_mask(self, inputs, mask=None):
        ...

    def call(self, inputs):
        ...

    def get_config(self):
        ...
```

Custom Layer Outline

```python
class MyLayer(layers.Layer):
    def __init__(self, arg1,arg2, ...):
        super(Linear, self).__init__()
        ...

    def build(self, input_shape):
        ...

    def compute_output_shape(self, input_shape):
        ...

    def compute_mask(self, inputs, mask=None):
        ...

    def call(self, inputs):
        ...

    def get_config(self):
        ...
```

Required!

A Very Basic Layer

A Very Basic Layer

```python
class Linear(layers.Layer):
```

A Very Basic Layer

```
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):









    def call(self, inputs):
        ...
```

A Very Basic Layer

```python
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()

        initializer = tf.initializers.GlorotUniform()
        self.w = tf.Variable(initializer([input_dim, units]),
                             name="kernel")

        initializer = tf.initializers.Zeros()
        self.b = tf.Variable(initializer([units]),
                             name="bias")

    def call(self, inputs):
        ...
```

A Very Basic Layer

```
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()

        initializer = tf.initializers.GlorotUniform()
        self.w = tf.Variable(initializer([input_dim, units]),
                             name="kernel")

        initializer = tf.initializers.Zeros()
        self.b = tf.Variable(initializer([units]),
                             name="bias")

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
```

A Very Basic Layer

```python
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()

        initializer = tf.initializers.GlorotUniform()
        self.w = tf.Variable(initializer([input_dim, units]),
                                       name="kernel")

        initializer = tf.initializers.Zeros()
        self.b = tf.Variable(initializer([units]),
                                       name="bias")

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

A Very Basic Layer

```python
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()

        initializer = tf.initializers.GlorotUniform()
        self.w = tf.Variable(initializer([input_dim, units]),
                             name="kernel")

        initializer = tf.initializers.Zeros()
        self.b = tf.Variable(initializer([units]),
                             name="bias")

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

What is the difference between our custom and Keras' Dense layer?

```python
Linear(units=..., input_dim=...)
Dense(units=...)
```

A Very Basic Layer

```
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()

        initializer = tf.initializers.GlorotUniform()
        self.w = tf.Variable(initializer([input_dim, units]),
                             name="kernel")

        initializer = tf.initializers.Zeros()
        self.b = tf.Variable(initializer([units]),
                             name="bias")

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

What is the difference between our custom and Keras' Dense layer?

```
Linear(units=..., input_dim=...)
Dense(units=...)
```

Do we have to also specify the input dimension for the Dense layer?

A Very Basic Layer - v2.0: Adding Laziness!

A Very Basic Layer - v2.0: Adding Laziness!

```python
class Linear(layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super(Linear, self).__init__()

        initializer = tf.initializers.GlorotUniform()
        self.w = tf.Variable(initializer([input_dim, units]),
                             name="kernel")

        initializer = tf.initializers.Zeros()
        self.b = tf.Variable(initializer([units]),
                             name="bias")

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

A Very Basic Layer - v2.0: Adding Laziness!

```
class Linear(layers.Layer):
    def __init__(self, units=32):
        super(Linear, self).__init__()



    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b






x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

A Very Basic Layer - v2.0: Adding Laziness!

```
class Linear(layers.Layer):
    def __init__(self, units=32):
        super(Linear, self).__init__()
        self.units = units


    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
```

```
x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

A Very Basic Layer - v2.0: Adding Laziness!

```
class Linear(layers.Layer):
    def __init__(self, units=32):
        super(Linear, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True)
        self.b = self.add_weight(shape=(self.units,),
                                 initializer='zeros',
                                 trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

A Very Basic Layer - v2.0: Adding Laziness!

```
class Linear(layers.Layer):
    def __init__(self, units=32):
        super(Linear, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True)
        self.b = self.add_weight(shape=(self.units,),
                                 initializer=‘zeros',
                                 trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

x = tf.ones((2, 2))
linear_layer = Linear(4)
y = linear_layer(x)
print(y)
```

A Very Basic Layer - v2.1: Make it Serializable

```
class Linear(layers.Layer):
    def __init__(self, units=32):
        super(Linear, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True)
        self.b = self.add_weight(shape=(self.units,),
                                 initializer='zeros',
                                 trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
```

A Very Basic Layer - v2.1: Make it Serializable

```
class Linear(layers.Layer):
    def __init__(self, units=32):
        super(Linear, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True)
        self.b = self.add_weight(shape=(self.units,),
                                 initializer='zeros',
                                 trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

    def get_config(self):
        config = super(Linear, self).get_config()
        config.update({'units': self.units})
        return config
```

A Very Basic Layer - v2.1: Make it Serializable

```
layer = Linear(64)
config = layer.get_config()
print(config)
```

A Very Basic Layer - v2.1: Make it Serializable

```python
layer = Linear(64)
config = layer.get_config()
print(config)
```

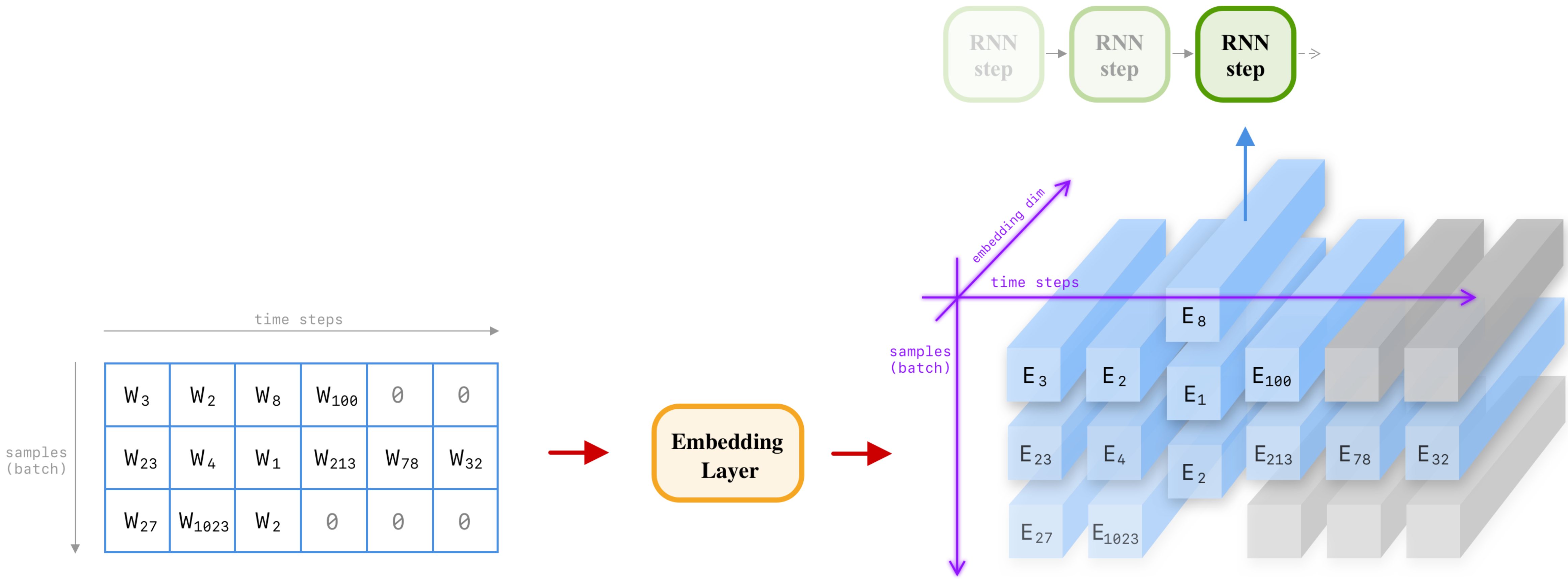
```
{'name': 'linear', 'trainable': True, 'dtype': 'float32', 'units': 64}
```

A Very Basic Layer - v2.1: Make it Serializable

```
layer = Linear(64)
config = layer.get_config()
print(config)

new_layer = Linear.from_config(config)
```

Masking

Masking



time steps
samples (batch)
W3 W2 W8 W100 0 0
W23 W4 W1 W213 W78 W32
W27 W1023 W2 0 0 0
Embedding Layer
RNN step
RNN step
RNN step
embedding dim
time steps
samples (batch)
E8
E3 E2 E1 E100
E23 E4 E2 E213 E78 E32
E27 E1023

Masking

Keras Layers fall into **3 categories** when it comes to making:

1. Mask **Consumers**
2. Mask **Propagators**
3. Mask **Generators**

Masking in Keras

```
class ConsumerLayer(layers.Layer):
    def call(self, inputs):
        ...
```

Masking in Keras

```
class ConsumerLayer(layers.Layer):
    def call(self, inputs, mask=None):
        ...
```

Masking in Keras

```python
class ConsumerLayer(layers.Layer):
    def call(self, inputs, mask=None):
        ...


class MaskPassThroughLayer(layers.Layer):
    def __init__(self, ...):
        self.support_masking = True
```

Masking in Keras

```python
class ConsumerLayer(layers.Layer):
    def call(self, inputs, mask=None):
        ...


class MaskPassThroughLayer(layers.Layer):
    def __init__(self, ...):
        self.support_masking = True


class GeneratorLayer(layers.Layer):
    def __init__(self, ...):
        self.support_masking = True

    def compute_mask(self, inputs, mask=None):
        ...
```

Masking in Keras: Example

```
class CustomEmbedding(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim, mask_zero=False):
        super(CustomEmbedding, self).__init__()
        self.mask_zero = mask_zero

    def build(self, input_shape):
        ...
    def call(self, inputs):
        ...
```

Masking in Keras: Example

```
class CustomEmbedding(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim, mask_zero=False):
        super(CustomEmbedding, self).__init__()
        self.supports_masking = True
        self.mask_zero = mask_zero

    def build(self, input_shape):
        ...
    def call(self, inputs):
        ...
```

Masking in Keras: Example

```
class CustomEmbedding(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim, mask_zero=False):
        super(CustomEmbedding, self).__init__()
        self.supports_masking = True
        self.mask_zero = mask_zero

    def build(self, input_shape):
        ...
    def call(self, inputs):
        ...

    def compute_mask(self, inputs, mask=None):
        if not self.mask_zero:
            return None
        return tf.not_equal(inputs, 0)
```

Masking in Keras: Example

```
layer = CustomEmbedding(10, 32, mask_zero=True)
x = np.array(
    [[2, 3, 4, 0, 0],
     [3, 3, 4, 9, 20],
     [9, 11, 1, 0, 0]], dtype=np.int32)

y = layer(x)
mask = layer.compute_mask(x)
```

Masking in Keras: Example

```
layer = CustomEmbedding(10, 32, mask_zero=True)
x = np.array(
    [[2, 3, 4, 0, 0],
     [3, 3, 4, 9, 20],
     [9, 11, 1, 0, 0]], dtype=np.int32)

y = layer(x)
mask = layer.compute_mask(x)
```

```
tf.Tensor(
[[ True  True  True False False]
 [ True  True  True  True  True]
 [ True  True  True False False]], shape=(3, 5), dtype=bool)
```

`training` argument in the `call` method

```
class CustomDropout(layers.Layer):
    def __init__(self, rate, **kwargs):
        super(CustomDropout, self).__init__(**kwargs)
        self.rate = rate

    def call(self, inputs, training=None):
        ...
```

`training` argument in the `call` method

```
class CustomDropout(layers.Layer):
    def __init__(self, rate, **kwargs):
        super(CustomDropout, self).__init__(**kwargs)
        self.rate = rate

    def call(self, inputs, training=None):
        if training:
            ...
```

Nested Layers

```python
class CustomDropout(layers.Layer):
    def __init__(self, rate):
        super(CustomDropout, self).__init__()
        self.rate = rate

    def call(self, inputs, training=None):
        if training:
            return tf.nn.dropout(inputs, rate=self.rate)
        return inputs
```

Nested Layers

```python
class CustomDropout(layers.Layer):
    def __init__(self, rate):
        super(CustomDropout, self).__init__()
        self.rate = rate

    def call(self, inputs, training=None):
        if training:
            return tf.nn.dropout(inputs, rate=self.rate)
        return inputs


mlp = MLPBlock()
y = mlp(tf.ones(shape=(3, 64)))

print('trainable weights:', len(mlp.trainable_weights))
# ?
```

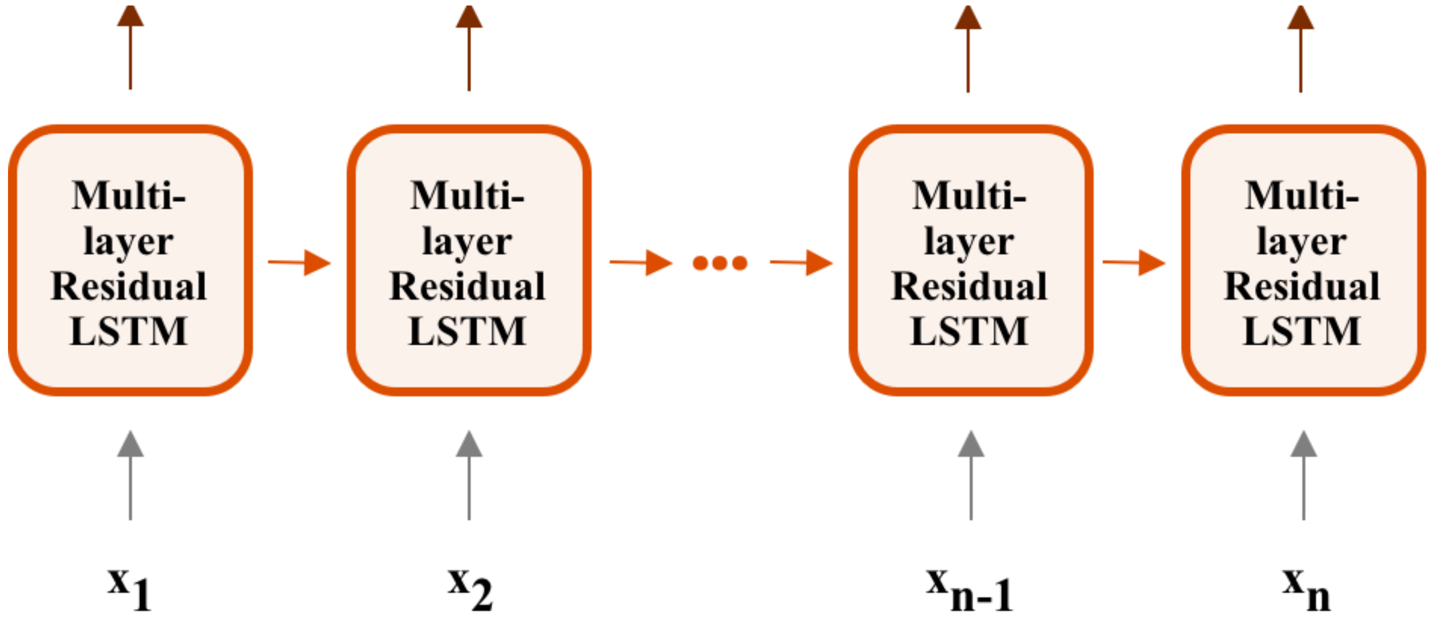
Nested Layers

```
class CustomDropout(layers.Layer):
    def __init__(self, rate):
        super(CustomDropout, self).__init__()
        self.rate = rate

    def call(self, inputs, training=None):
        if training:
            return tf.nn.dropout(inputs, rate=self.rate)
        return inputs


mlp = MLPBlock()
y = mlp(tf.ones(shape=(3, 64)))

print('trainable weights:', len(mlp.trainable_weights))
# 6
```

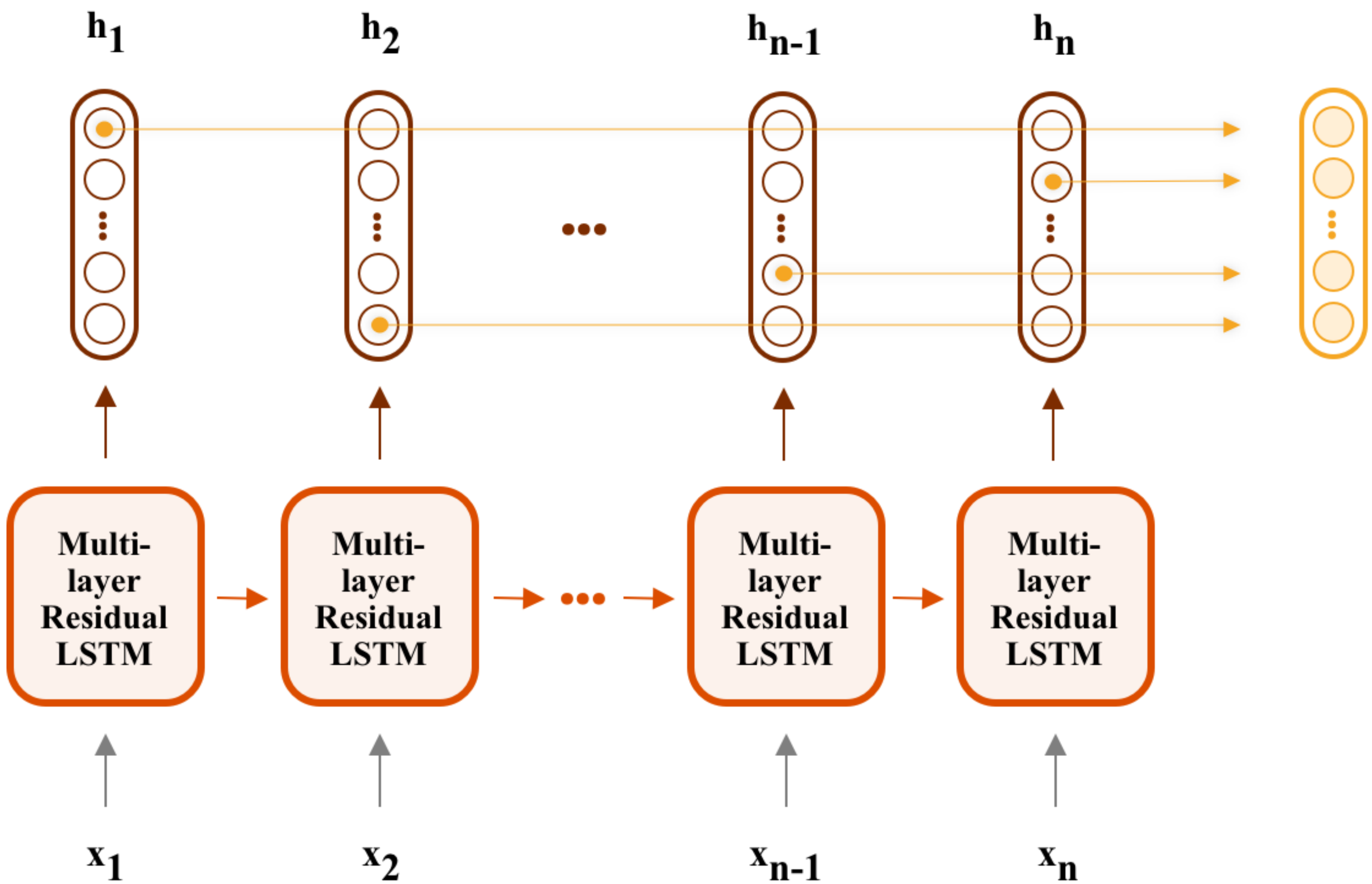
Quiz #2: Max-pooling through time

One technique that originates from Computer Vision is called Max pooling. As you might remember, this technique reduces the impact of spatial information in the image. For example, If your CNN says, "Yay! I found a wheel at the position (x,y).", your Max-pooling will convert this sentence to "Yay! I found a wheel in this image." Intuitively, we can use the max-pooling procedure in any configuration beside an image. Here is an example of Max-pooling application in recurrent networks:

Quiz #2: Max-pooling through time



Quiz #2: Max-pooling through time



Quiz #2: Max-pooling through time

In this setup, we'd like to perform max-pooling over the hidden states $\hat{h} = MaxPool([h^{(1)}, \ldots, h^{(n)}])$ where h is the max-pooled version. Every dim of h the maximum of that particular dim across all of the hidden states.

$$\hat{h}_i = \max_{1 \leq k \leq n} h_i^{(k)}$$

Although the default Keras framework provides the implementation, it lacks the masking support. Implement this mechanism as a Keras layer.

Model Design

Keras **Sequential** API
+ standard layers

Stack of layers
For Simple models

Keras **Functional** API
+ standard layers

DAG of layers
For Simple models

Model Design

Keras Sequential API + standard layers

Stack of layers
For Simple models

Keras Functional API + standard layers

DAG of layers
For Simple models

Model Sub-classing + standard layers

Define model by Python
For very customized models

Custom Model

```python
class MyModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MyModel, self).__init__()
        ...



    def call(self, inputs):
        # Define your forward pass here
```

Custom Model

```
class MyModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MyModel, self).__init__()
        self.dense_1 = layers.Dense(32, activation=‘relu')
        self.dense_2 = layers.Dense(num_classes,
                                    activation='softmax')

    def call(self, inputs):
        # Define your forward pass here
        x = self.dense_1(inputs)
        return self.dense_2(x)
```

Custom Model

```
class MyModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MyModel, self).__init__()
        self.dense_1 = layers.Dense(32, activation='relu')
        self.dense_2 = layers.Dense(num_classes,
                                    activation='softmax')

    def call(self, inputs):
        # Define your forward pass here
        x = self.dense_1(inputs)
        return self.dense_2(x)
```

Custom Model

```
class MyModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MyModel, self).__init__()
        self.dense_1 = layers.Dense(32)
        self.dense_2 = layers.Dense(num_classes,
                                    activation='softmax')

    def call(self, inputs):
        # Define your forward pass here
        x = self.dense_1(inputs)
        x = tf.nn.relu(x)
        return self.dense_2(x)
```

Models vs. Layers

Models vs. Layers

Models are exactly the same as layers!

Models vs. Layers

Models are exactly the same as layers! plus:

Models vs. Layers

Models are exactly the same as layers! plus:

+ Training (`model.fit(), .compile(), .evaluate, and etc.`)
+ Save and load on the disk
+ Summary/Visualization

Models vs. Layers

Layer corresponds to what we refer to in the literature as a "layer" (as in **"convolution layer"** or **"recurrent layer"**) or as a "block" (as in **"ResNet block"** or **"Inception block"**).

Model corresponds to what is referred to in the literature as a "model" (as in **"deep learning model"**) or as a "network" (as in **"deep neural network"**)

Model Sub-Classing (MSC) vs. Functional API

Model Sub-Classing (MSC) vs. Functional API

- ✓ MSC is much more flexible in the graph definition (recall that in Functional API everything should an instance of a Layer when we are connecting different nodes of the model. Hence, no support for low-level TF Ops)

Model Sub-Classing (MSC) vs. Functional API

- ✓ MSC is much more flexible in the graph definition (recall that in Functional API everything should an instance of a Layer when we are connecting different nodes of the model. Hence, no support for low-level TF Ops)
- ✓ MSC supports changing the runtime branch between training and evaluation (via training parameter)

Model Sub-Classing (MSC) vs. Functional API

- ✓ MSC is much more flexible in the graph definition (recall that in Functional API everything should an instance of a Layer when we are connecting different nodes of the model. Hence, no support for low-level TF Ops)
- ✓ MSC supports changing the runtime branch between training and evaluation (via training parameter)
- ✓ The mask argument should be passed manually in the MSC.

Model Sub-Classing (MSC) vs. Functional API

- ✓ MSC is much more flexible in the graph definition (recall that in Functional API everything should an instance of a Layer when we are connecting different nodes of the model. Hence, no support for low-level TF Ops)
- ✓ MSC supports changing the runtime branch between training and evaluation (via training parameter)
- ✓ The mask argument should be passed manually in the MSC.
- ✓ Model saving & restoring is easier in the Functional API.

Model Design

Keras **Sequential** API
+ standard layers

`Stack of layers`
`For Simple models`

Model **Sub-classing**
+ standard layers

`Define model by Python`
`For very customized models`

Keras **Functional** API
+ standard layers

`**DAG** of layers`
`For Simple models`

Model Design

Keras **Sequential** API

+ standard layers

+ custom layers, losses, and metrics

`Stack of layers`

`For Simple models`

Keras **Functional** API

+ standard layers

+ custom layers, losses, and metrics

`DAG of layers`

`For Simple models`

Model **Sub-classing**

+ standard layers

+ custom layers, losses, and metrics

`Define model by Python`

`For very customized models`

Model Training

Model Training

Keras **built-in** loops

model.fit()
+ callbacks

`Fast prototyping`

Keras built-in training loops

```
model = MyModel()
```

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])
```

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])
```

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])
```

- .compile() is about configuring the training process.
- Specify the optimizer, loss, and metrics.

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])

history = model.fit(data,
          epochs=10, batch_size=128,
          validation_data=val_data,
          callbacks=[EarlyStopping(),
                     TensorBoard(),
                     ModelCheckpoint()])
```

Keras built-in training loops

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()])

history = model.fit(data,
          epochs=10, batch_size=128,
          validation_data=val_data,
          callbacks=[EarlyStopping(),
                     TensorBoard(),
                     ModelCheckpoint()])

results = model.evaluate(test_data, batch_size=128)
```

Keras built-in training loops (run in dynamic graph mode)

```
model = MyModel()

model.compile(optimizer=Adam(),
              loss=BinaryCrossentropy(),
              metrics=[AUC(), Precision(), Recall()]
              run_eagerly=True)

history = model.fit(data,
          epochs=10, batch_size=128,
          validation_data=val_data,
          callbacks=[EarlyStopping(),
                     TensorBoard(),
                     ModelCheckpoint()])

results = model.evaluate(test_data, batch_size=128)

predictions = model.predict(x_test[:3])
```

Model Training

Keras **built-in** loops

model.fit()
+ callbacks

`Fast prototyping`

Model Training

Keras **built-in** loops
model.fit()
+ callbacks

`Fast prototyping`

GradientTape
custom training loops

`Complete control`

Gradient tapes

Tensorflow "records" all operations executed inside the context of a *tf.GradientTape* onto a "tape".

Tensorflow then uses that tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using reverse mode differentiation.

Gradient Tape

```python
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    y = x * x

dy_dx = g.gradient(y, x) # Will compute to 6.0
```

Gradient Tape

```python
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    y = x * x

dy_dx = g.gradient(y, x) # Will compute to 6.0
```

How to calculate second-oder derivatives?

Gradient Tape

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
    g.watch(x)
    with tf.GradientTape() as gg:
        gg.watch(x)
        y = x * x
    dy_dx = gg.gradient(y, x)     # Will compute to 6.0

d2y_dx2 = g.gradient(dy_dx, x)  # Will compute to 2.0
```

Gradient Tape; Real-world usage

Gradient Tape; Real-world usage

```
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)
```

Gradient Tape; Real-world usage

```python
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)

# Training loop
for x, y in dataset:
```

Gradient Tape; Real-world usage

```
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)

# Training loop
for x, y in dataset:
    with tf.GradientTape() as g:
        preds = model(x)
        loss = loss_fn(y, preds)
```

Gradient Tape; Real-world usage

```
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)

# Training loop
for x, y in dataset:
    with tf.GradientTape() as g:
        preds = model(x)
        loss = loss_fn(y, preds)

    g_m, g_b = g.gradient(loss, [m, b])
```

Gradient Tape; Real-world usage

```
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)

# Training loop
for x, y in dataset:
    with tf.GradientTape() as g:
        preds = model(x)
        loss = loss_fn(y, preds)

    g_m, g_b = g.gradient(loss, [m, b])

    m.assign_sub(g_m * lr)
    b.assign_sub(g_b * lr)
```

Gradient Tape; Real-world usage

```
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)

# Training loop
for x, y in dataset:
    with tf.GradientTape() as g:
        preds = model(x)
        loss = loss_fn(y, preds)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(
                     zip(grads, model.trainable_variables))
```

Gradient Tape; Real-world usage

```
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)


def train_step(x, y):
    with tf.GradientTape() as g:
        preds = model(x)
        loss = loss_fn(y, preds)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(
                     zip(grads, model.trainable_variables))

    return loss.numpy()
```

Gradient Tape; Real-world usage

```
lr = 0.05
m = tf.Variable(0.)
b = tf.Variable(0.)

@tf.function
def train_step(x, y):
    with tf.GradientTape() as g:
        preds = model(x)
        loss = loss_fn(y, preds)

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(
                     zip(grads, model.trainable_variables))

    return loss.numpy()
```

Assignment: Low-level MNIST CNN-Classifier

We have implemented an MNIST classifier several times. However, we want to implement a multi-layer CNN classifier using raw TensorFlow operations and matrix multiplication. No pre-defined CNN blocks are allowed. Moreover, you should organize your model using TF 2.0 Model sub-classing APIs (your assignment will be evaluated based on the cleanness and your effective usages of high-level Model APIs). Additionally, the only available training option is the TF's GradientTape.

Assignments criteria:

- Matrix implementation of a convolutional layer.
- Model organization using TF 2.0 sub-classing API
- Model training using TF 2.0 GradientTape

Thank you!