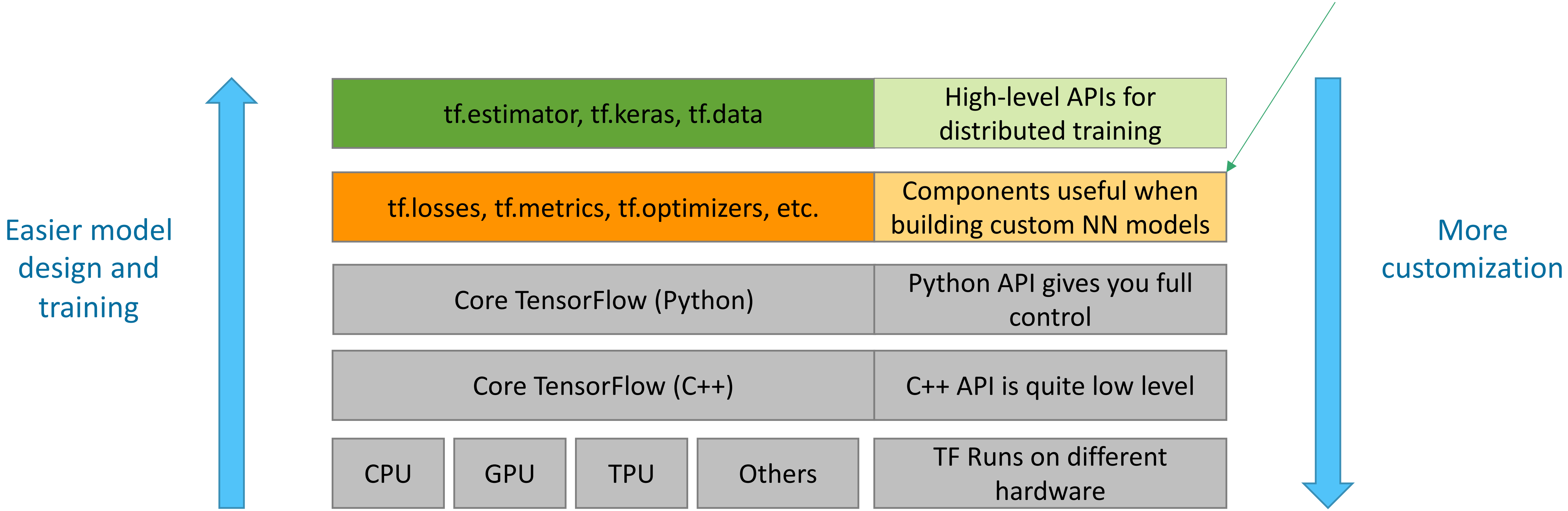
# Tensorflow 2

Keras API (Continued)

# Tensorflow API Hierarchy

- Tensorflow exposes APIs at multiple abstraction levels

These can be used both by keras and core

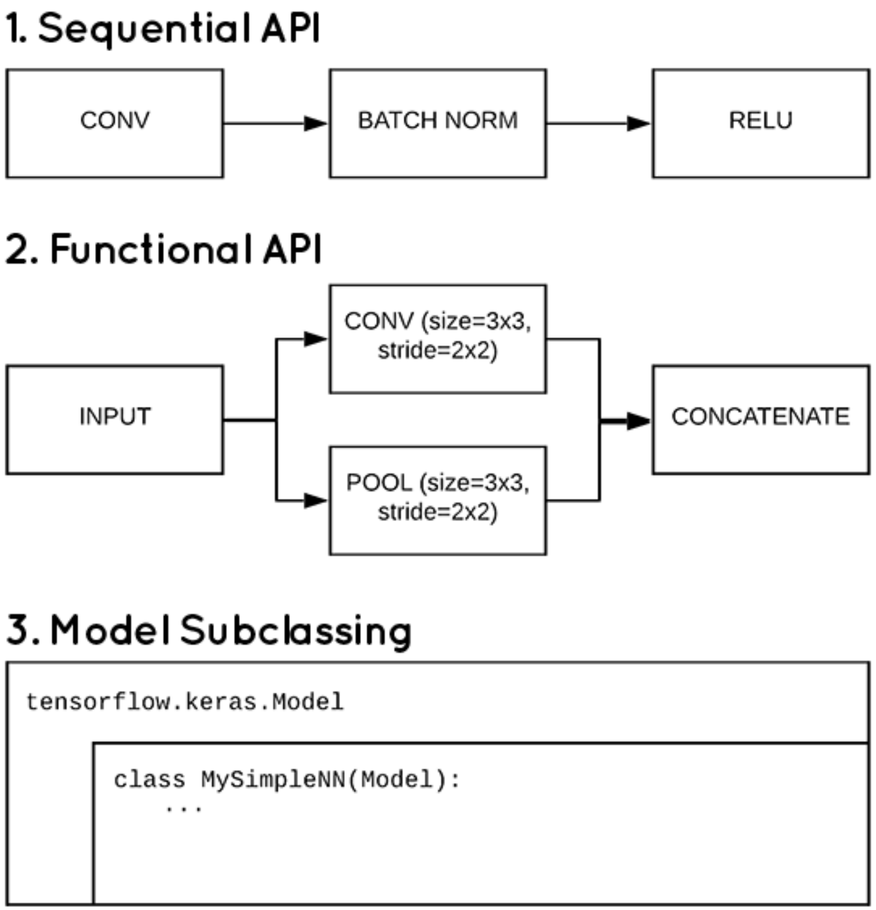


# The Keras API

- Most of the times you don’t need to write custom models and trainign loops from scratch
- Keras is a high-level API built-in in TF2, that provides a much easier flow to write “standard” models
  - No need to worry about gradient tapes, weight updates, etc. manually.

- A Keras model can be built in 3 main ways:
  - **Sequential API** → for Single-Input Single-Output models built as stacks of layers
  - **Functional API** → for MIMO models, residual connections, etc.
  - **Sub-classing API** → for maximum customization (e.g. dynamic/adaptive models)

# The Keras API

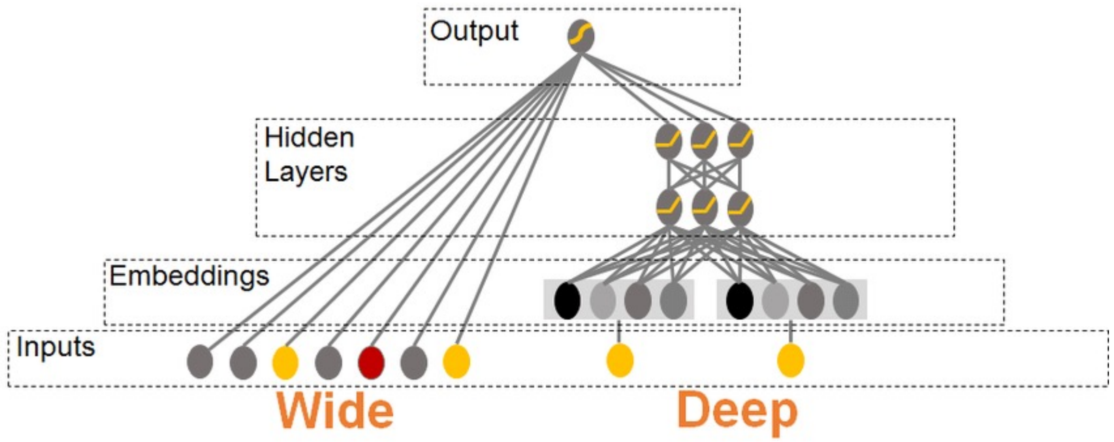


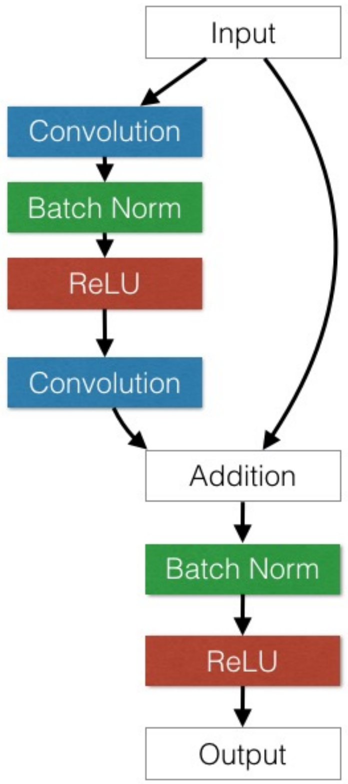
# Tensorflow 2

Keras Functional API

# The Keras Functional API

- The Sequential API is limited to single-in single-out models built as a stack of layers.
  - The most common scenario, but not the only one.
  - E.g. deep and wide models, residual connections, shared layers, etc:



# The Keras Functional API

- The Functional API can handle arbitrary (static) graphs of layers

- First create an input node (implicit in the Sequential API):

```
inputs = keras.Input(shape=(28, 28))
```

- Then create new layer instances and apply them to the input by calling them as functions:

```
flatten = keras.Flatten(input_shape=(28, 28))
x = flatten(inputs)
```

- Or more synthetically:

```
x = keras.Flatten(input_shape=(28, 28))(inputs)
```

# The Keras Functional API

- Connect many of these layer “nodes” as you wish:

```
inputs = keras.Input(shape=(28, 28))
x = keras.Flatten(input_shape=(28, 28))(inputs)
x = layers.Dense(64, activation="relu")(x)
x = layers.Dense(64, activation="relu")(x)
outputs = layers.Dense(10)(x)
```

- Finally create a `Model()` by specifying the input(s) and output(s) of the graph:

```
model = keras.Model(
    inputs=inputs,
    outputs=outputs,
    name=“my_model")
```

# The Keras Functional API

```
inputs = keras.Input(shape=(32, 32, 3), name="img")
x = layers.Conv2D(32, 3, activation="relu")(inputs)
x = layers.Conv2D(64, 3, activation="relu")(x)
block_1_output = layers.MaxPooling2D(3)(x)

x = layers.Conv2D(64, 3, activation="relu", padding="same")(block_1_output)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
block_2_output = layers.add([x, block_1_output])

x = layers.Conv2D(64, 3, activation="relu", padding="same")(block_2_output)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
block_3_output = layers.add([x, block_2_output])
```

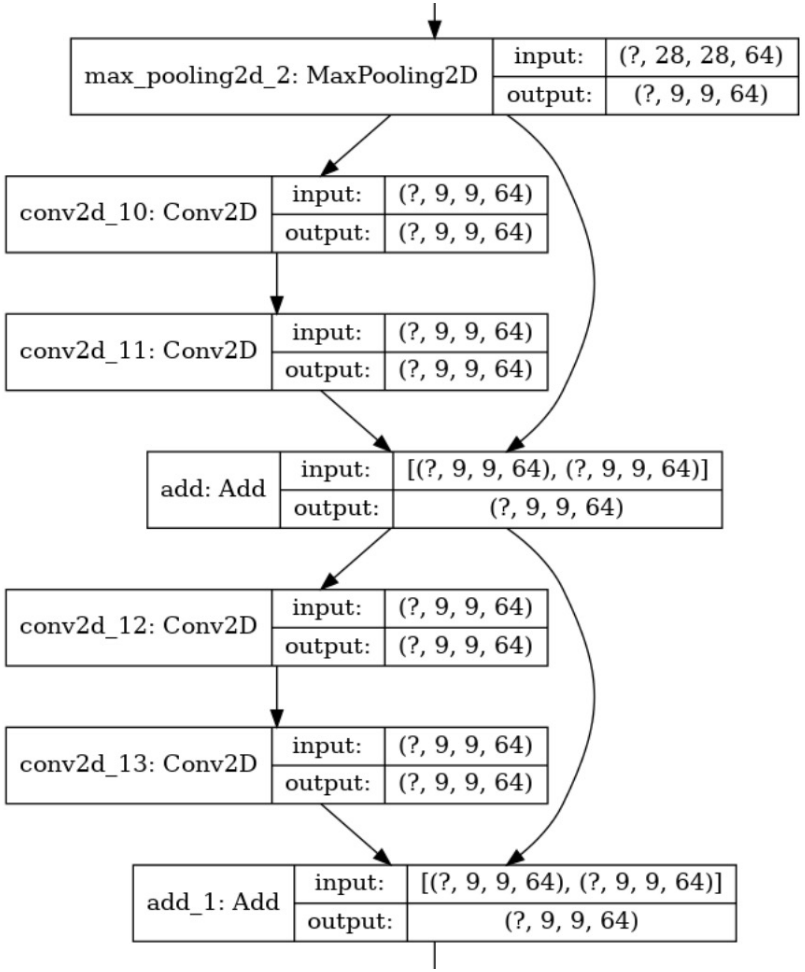
Add input and output to create a residual connection

# The Keras Functional API

```
x = layers.Conv2D(64, 3, activation="relu")(block_3_output)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(10)(x)

model = keras.Model(inputs, outputs, name="toy_resnet")
```

# The Keras Functional API



# The Keras Functional API

- Only step 2 of the whole Keras flow changes, the rest is identical!

- **Notebook:** Keras Functional API

# Tensorflow 2

Keras Sub-Classing API

# The Keras Sub-classing API

- Useful when you want to build models that:
  - Don't use standard layers provided by keras (e.g. Conv, MaxPooling, GRU, LSTM, BatchNormalization, Dropout, etc.)
  - Or cannot be represented by DAGs (e.g. tree RNNs)
  - We won't go in so many details, but we'll show you the API for completeness (and because it's similar to PyTorch)

- Custom layers are created by sub-classing the `keras.Layer` class

- Custom models are created by sub-classing the `keras.Model` class

# The Keras Sub-classing API

- **CAUTION**: in general, don’t use the sub-classing API unless you really need it


- Sequential/Functional models are much easier to:
  - Inspect (`summary()`, `plot_model()`, etc.)
  - Debug (input shape and dtype specified in advance using `keras.Input`).
  - Serialize/clone/save/restore (with subclassing, you have to write your own model saving code)

# The Keras Sub-classing API

- A custom layer = some trainable `tf.Variable` and some computations:

Base class constructor

```
class MyLinear(keras.layers.Layer):

    def __init__(self, units=32):
        super(Linear, self).__init__()
        self.units = units

    ...
```

# The Keras Sub-classing API

- A custom layer = some trainable `tf.Variable` and some computations:

```
class MyLinear(keras.layers.Layer):

    ...

    def build(self, input_shape):
        w_init = tf.random_normal_initializer()
        self.w = tf.Variable(
            initial_value=w_init(shape=(input_shape[-1], self.units),
                dtype="float32"), trainable=True)
        b_init = tf.zeros_initializer()
        self.b = tf.Variable(
            initial_value=b_init(shape=(self.units,),
                dtype="float32"), trainable=True)
    ...
```

Called only the first time a layer is invoked. Useful to support variable input shapes

Define and initialize weights

Define and initialize biases

# The Keras Sub-classing API

- A custom layer = some trainable `tf.Variable` and some computations:

```
class MyLinear(keras.layers.Layer):

    ...


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

A set of TF ops for autograd

# The Keras Sub-classing API

- You can then use this custom layer in a standard (functional or sequential) way:

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

# The Keras Sub-classing API

- You can avoid the explicit creation of `tf.Variable` thanks to the `add_weight()` method:

```
def build(self, input_shape):
    super(Linear, self).__init__()

    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)
```

- You can set `trainable=False` if you want non-trainable weights

# The Keras Sub-classing API

- Custom layers can then be used as part of other custom layers (recursively composable).
- You can define two other functions (`get_config()` and `from_config()`) to support serialization.
- You can track custom losses and metrics, and many other advanced functionalities.
- See full details here: https://www.tensorflow.org/guide/keras/custom_layers_and_models

# The Keras Sub-classing API

- Sub-classes of `keras.Models` define entire models (i.e. objects you will train)

- Should I use the Layer class or the Model class?
  - will I need to call fit() on it?
  - Will I need to call save() on it?
  - If so, go with Model.

# The Keras Sub-Classing API

- Example (very similar to PyTorch):

```
class ResNet(tf.keras.Model):

    def __init__(self):
        super(ResNet, self).__init__()
        self.block_1 = ResNetBlock()
        self.block_2 = ResNetBlock()
        self.global_pool = layers.GlobalAveragePooling2D()
        self.classifier = Dense(num_classes)

    def call(self, inputs):
        x = self.block_1(inputs)
        x = self.block_2(x)
        x = self.global_pool(x)
        return self.classifier(x)
```

A custom layer

# The Keras Sub-Classing API

- **Notebook:** Keras Subclassing API

# Tensorflow 2

(Some) Advanced Keras Features

# (Some) Advanced Keras Features

1. Saving Models
2. Advanced options for `compile()`:
   - Custom metrics and losses
3. Advanced options for `fit()`:
   - Creating/passing validation sets
   - Class and sample weights
   - Using callbacks
4. Tensorboard

# (Some) Advanced Keras Features

1. Saving Models
2. Advanced options for `compile()`:
   - Custom metrics and losses
3. Advanced options for `fit()`:
   - Creating/passing validation sets
   - Class and sample weights
   - Using callbacks
4. Tensorboard

# Saving Models

- A Keras model consists of multiple components:
  - An **architecture**, or configuration, which specifies what layers the model contain, and how they're connected.
  - A set of **weights** values (the "state of the model").
  - An **optimizer** (defined by compiling the model).
  - A set of **losses and metrics** (defined by compiling the model)

- You can save all these components or only some of them.

# Saving Models

- Saving everything:

```
model.save('path/to/location’)

# load back
model = keras.models.load_model('path/to/location’)
```

- The reconstructed model is already compiled and has retained the optimizer state. You can safely resume training.

# Saving Models

- Saving <u>only the architecture</u> (no weights, no compile information):

```
json_config = model.to_json()
new_model = keras.models.model_from_json(json_config)
```

- The reconstructed model weights are re-initialized and there is no optimizer info, etc.

# Saving Models

- Saving only the weights (e.g. for checkpointing):

```
model.save_weights("ckpt")
load_status = model.load_weights("ckpt")
```

- By default uses TF Checkpoint format.

# (Some) Advanced Keras Features

1. Saving Models
2. Advanced options for `compile()`:
   - Custom metrics and losses
3. Advanced options for `fit()`:
   - Creating/passing validation sets
   - Class and sample weights
   - Using callbacks
4. Tensorboard

# Custom Losses and Metrics

- You can pass the `compile()` function a custom loss function:

```
def my_mse(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_pred – y_true))


model.compile(optimizer="adam", loss=my_mse, metrics=…)
```

# Custom Losses and Metrics

- You can pass the `compile()` function one or more custom metrics, and combine them with built-in ones:

```
model.compile(optimizer="adam", loss=…, metrics=[my_mse, "mae"])
```

# Custom Losses and Metrics

- **Notebook:** Regression with Custom Loss & Metrics

# (Some) Advanced Keras Features

1. Saving Models
2. Advanced options for `compile()`:
   - Custom metrics and losses
3. Advanced options for `fit()`:
   - Creating/passing validation sets
   - Class and sample weights
   - Using callbacks
4. Tensorboard

# Creating Validation Sets

- Validation data can be either passed to the `fit()` function....

```
history = model.fit(
    X_train,
    Y_train,
    epochs=10,
    batch_size=32,
    validation_data=<validation_data>,
    )
```

- The main accepted formats are: a tuple of (x_val, y_val) or a TF dataset

# Creating Validation Sets

- …or we can let fit() pick a validation set for us:

```
history = model.fit(
    X_train,
    Y_train,
    epochs=10,
    batch_size=32,
    validation_split=<fraction in [0-1], e.g. 0.2>,
    )
```

- In this case, a fraction of the training data will be used for validation (and the model won't be trained on it)

# Creating Validation Sets

- By default, validation is performed after every epoch:

```
Epoch 1/40
8/8 [==============================] - 0s 14ms/step - loss: 582.3579 - mae: 22.8670 - val_loss: 581.9780 - val_mae: 2
2.9370
Epoch 2/40
8/8 [==============================] - 0s 3ms/step - loss: 539.1675 - mae: 22.0812 - val_loss: 537.9955 - val_mae: 2
2.1408
```

- You can change this with the parameter `validation_freq`

# Class and Sample Weights

- You can pass a dictionary of: `{class_index : class_weight}` to `fit()`, in order to weigh the loss function based on the true class of a sample.
- As you probably know, this is one of the ways to deal with class imbalance, letting the training algorithm give “more importance” to under-represented samples.

```
history = model.fit(
    X_train,
    Y_train,
    epochs=10,
    batch_size=32,
    class_weight={0: 1., 1: 50., 2: 2.}
)
```

# Class and Sample Weights

- Alternatively, you can provide fit() with a weight for every training sample, which is useful (for example) when you know that some samples' labels have higher "confidence" than others
- The basic way to pass sample weights is using an array of the same length as the training data

```
history = model.fit(
    X_train,
    Y_train,
    epochs=10,
    batch_size=32,
    sample_weight=<tensor or np.array>)
```

# Callbacks

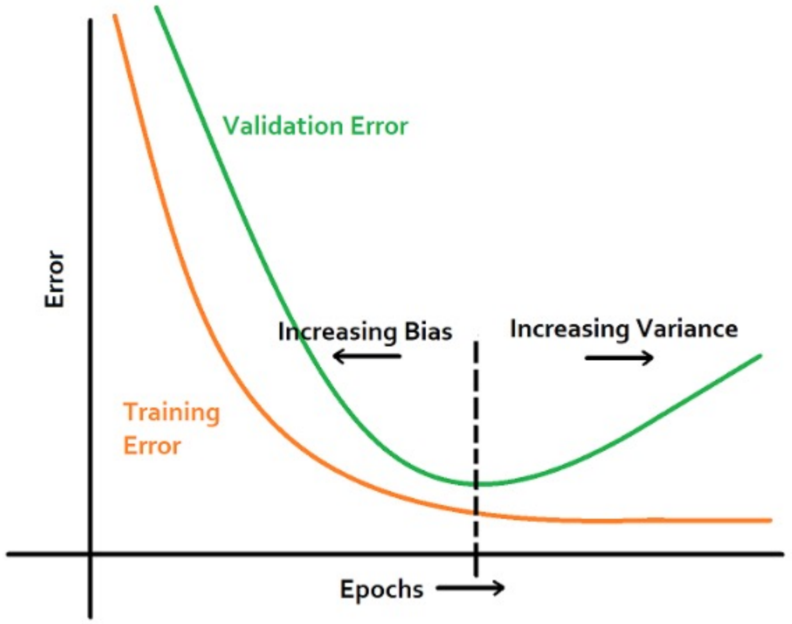
- Callbacks are a powerful tool to customize the behavior of Keras during training, evaluation and inference.
- Callbacks can be passed to `fit()`, `evaluate()` and `predict()`
- All callbacks are sub-classes of `keras.callbacks.Callback` and override a set of methods called at various stages of the three functions above

# Callbacks

- Keras provides you with a large set of pre-cooked callbacks, such as:
  - `EarlyStopping`: Stop training when a monitored metric has stopped improving
  - `ModelCheckpoint`: Save the Keras model or model weights with some frequency
  - `LearningRateScheduler`: change the LR during training
  - `TensorBoard`: Enable visualizations for TensorBoard (see later)
  - Many others…

# Callbacks (Early Stopping)

- Early stopping is a good practice for training deep learning models, useful to:
  - Avoid computationally intensive training epochs when they do not improve the predictive capabilities of the model.
  - Stop training in the so-called “optimal capacity point, where bias and variance are both low.



# Callbacks (Early Stopping)

- Some parameters:

```
tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',     Quantity to monitor
    patience=0,   Number of not-improving epochs before stopping
    verbose=0,   Print a message explaining why you stopped
    mode='auto',   Specify if the target quantity should be minimized/maximized
    …)             (auto uses the name)
```

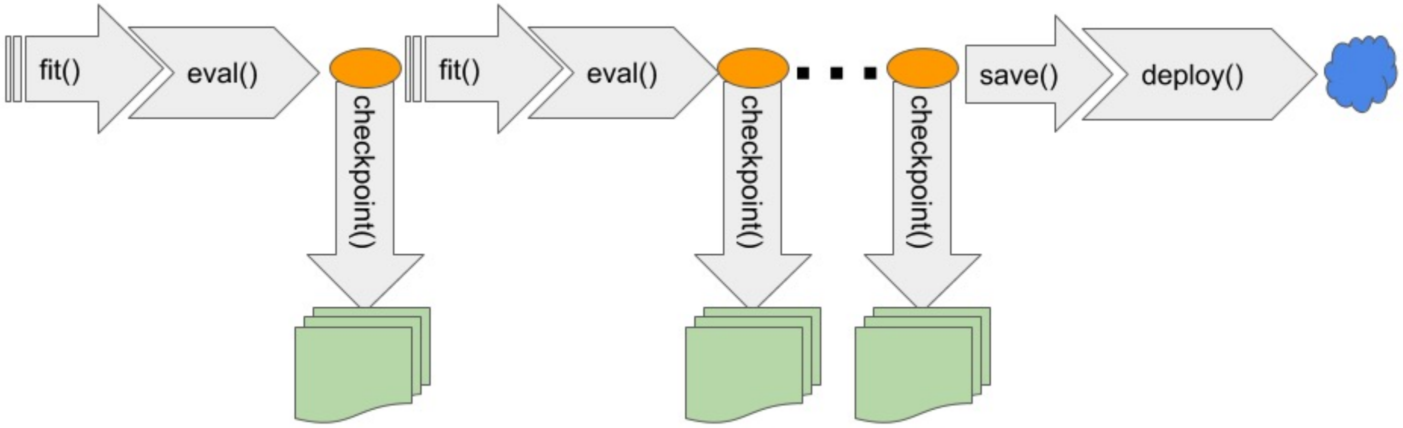
# Callbacks (Early Stopping)

- Example of usage:

```
history = model.fit(
    X, Y, epochs=200, …,
    callbacks=[EarlyStopping(
        monitor='val_loss',
        patience=2,
        verbose=1)
   ])
```

# Callbacks (Model Checkpoint)

- Checkpoints are simply periodic snapshots of the current status of a model, saved during training. Useful to:
  - Avoid losing everything in case of crashes
  - Analyzing the model behavior over the training epochs
  - Retreiving the overall best model for deployment



# Callbacks (Model Checkpoint)

- Some parameters:

```
tf.keras.callbacks.ModelCheckpoint(
     filepath,  Destination
     monitor='val_loss', Metric to monitor (e.g. for save_best_only)
     verbose=0,
     save_best_only=False,  Don't overwrite best checkpoint version
     save_weights_only=False, Save with model.save() or with model.save_weights()
     mode='auto',  Same as EarlyStopping
     save_freq='epoch',  'epoch' or an integer (interpreted as number of batches)
)
```

# Callbacks (Model Checkpoint)

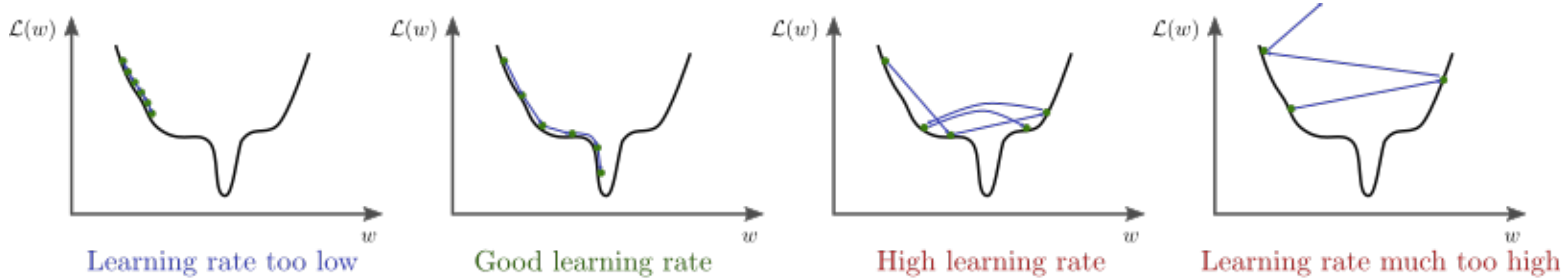
- Example of usage:

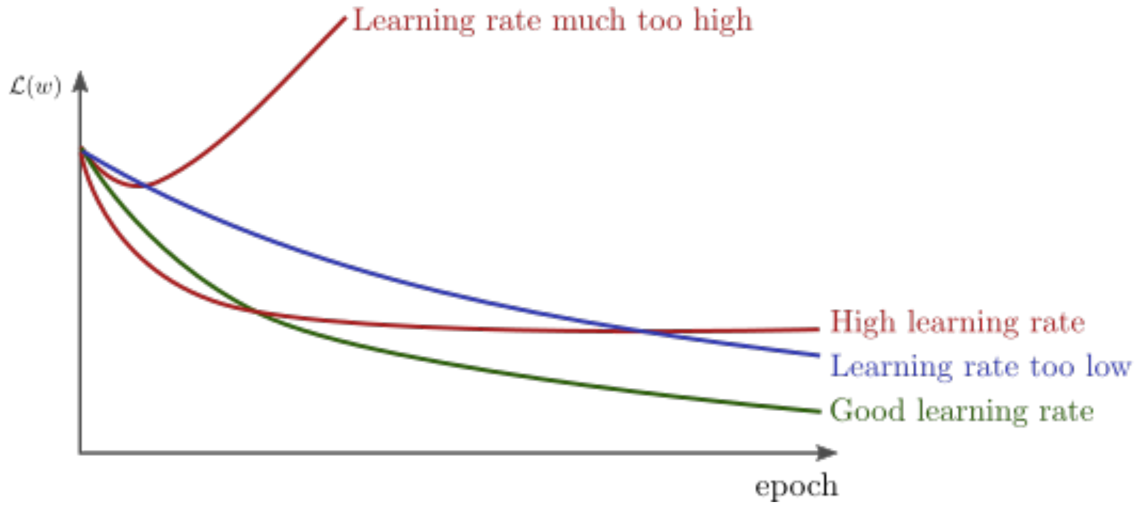
```
cp_callback = ModelCheckpoint(
    './callback_test_chkp/chkp_{epoch:02d}_{val_loss:.2f}’,
    save_best_only=False,
    save_weights_only=False,
    save_freq='epoch'
)
```

The filepath can use named formatting options for the epoch, loss and metrics values.

# Callbacks (Learning Rate Scheduling)

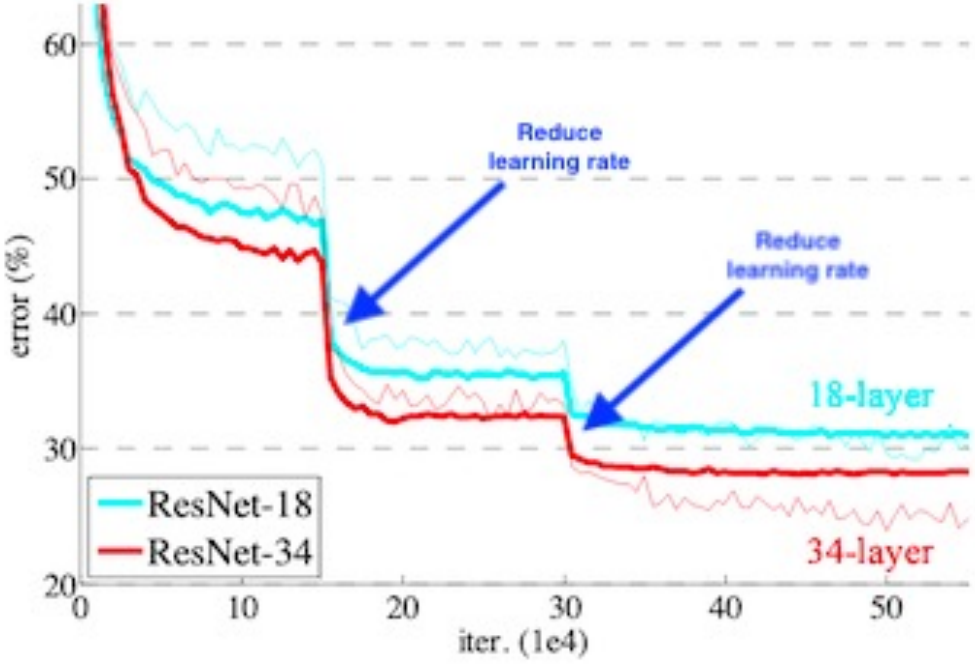
- The Learning Rate (LR) is one of the most important hyper-parameters of a deep learning model.



# Callbacks (Learning Rate Scheduling)

- LR Scheduling refers to dynamically changing the LR during training. There are many possible schedules proposed (still very much an art…)
- Example: reduce LR on plateau



# Callbacks (Learning Rate Scheduling)

- Some parameters:

```
lr_callback = LearningRateScheduler(
            schedule,
            verbose=0,
    )
```

A Python function that takes the epoch and the current LR and returns the next LR

# Callbacks (Learning Rate Scheduling)

- Example of usage:

```
def my_schedule(epoch, lr):
  if epoch < 10:
    return lr
  else:
    return lr * tf.math.exp(-0.1)


lr_callback = LearningRateScheduler(my_schedule, verbose=0)
```

# Callbacks (Learning Rate Scheduling)

- There’s also a pre-cooked LR callback for “Reduce on plateau” technique:

```
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.3,
    patience=5,
    min_lr=1e-04,
    verbose=1
)
```

# Callbacks (Custom)

- You can creating your own callbacks by subclassing the `Callback` class.

- Methods that can be overridden:
  - **on_(train|test|predict)_begin(self, logs):** Called at the beginning of fit/evaluate/predict.
  - **on_(train|test|predict)_end(self, logs):** Called at the end of fit/evaluate/predict.
  - **on_(train|test|predict)_batch_begin(self, batch, logs):** Called right before processing a batch during training/testing/predicting.
  - **on_(train|test|predict)_batch_end(self, batch, logs):** Called at the end of training/testing/predicting a batch.
  - **on_epoch_begin(self, epoch, logs):** Called at the beginning of an epoch during training.
  - **on_epoch_end(self, epoch, logs):** Called at the end of an epoch during training.

# Callbacks (Custom)

- Example: `on_epoch_end(self, epoch, logs)`

epoch number

`self.model` is a pointer to the model instance, which can be used to stop training, implement LR scheduling, etc.

Stores metrics and losses values as a dictionary of: {quantity_name: val}, e.g. logs['mae'] or logs['loss']

# Callbacks (Custom)

- Example: custom early-stopping the first time the epoch accuracy reaches 90%:

```
class myEarlyStopping(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
            if(logs.get('accuracy')>0.9):
                    print("\nReached 90% accuracy so cancelling training!")
                    self.model.stop_training = True
```

# Callbacks

- **Notebook:** Callbacks

# Using Datasets with Keras

- Of course, the Keras API can digest `tf.data.Datasets` naturally.

- One important caveat: by default, you should not call repeat() to make an infinite dataset. Repetition is handled internally by Keras
  - If you already have an infinite dataset, you must pass the additional parameters `steps_per_epoch` parameter in `fit()` and `steps` in `evaluate()`

- Note that the predict() function does not need labels, but if you pass a dataset that contains two components, the labels are simply ignored.

# Using Datasets with Keras

- Moreover, the `keras.preprocessing` package contains many pre-cooked input loading functions that produce Datasets, hiding all the details of `tf.data`

- Example:

```
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

Train/val split, batching, resizing in a single step.

# Using Datasets with Keras

- **Notebook:** TF Data and Keras