An introduction to Tensorflow 2.0

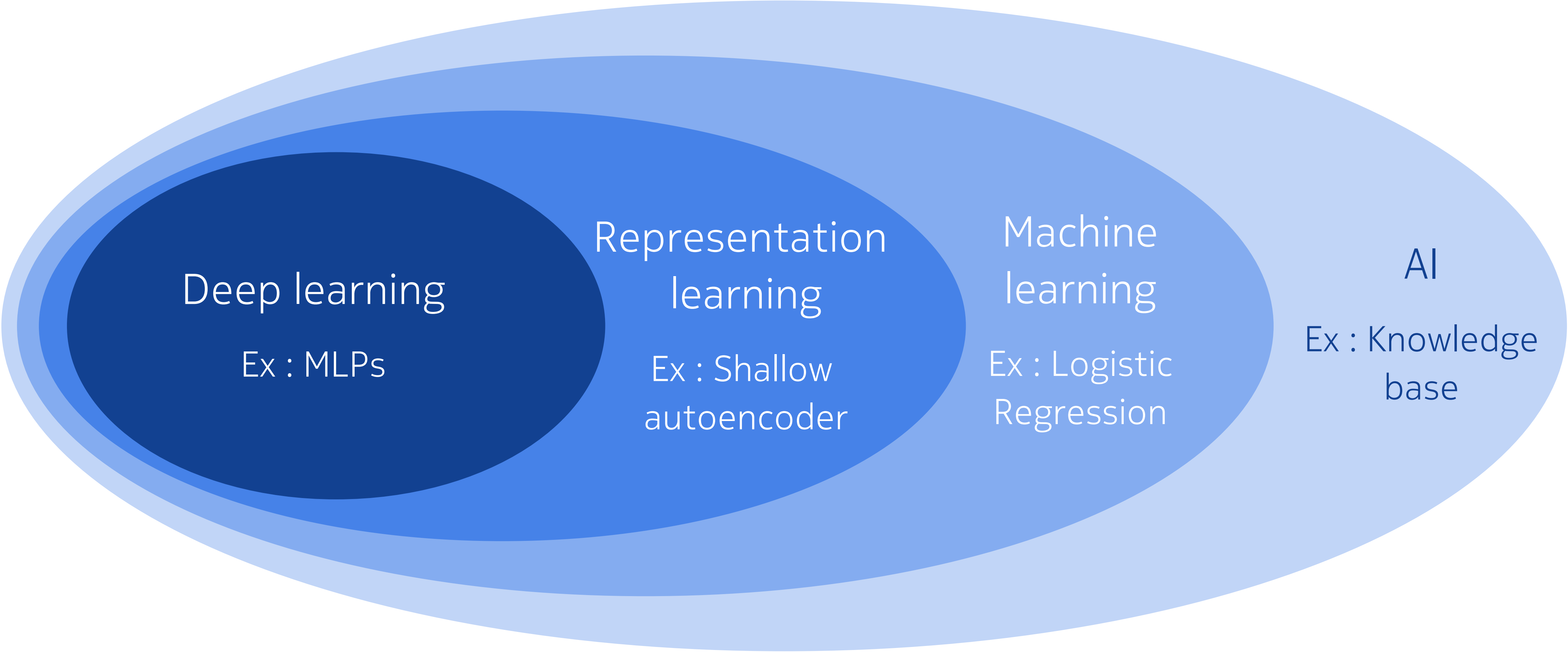
Names
Training school
Date

NOKIA

An introduction to Tensorflow 2.0

- Introduction to Deep Learning
- Tensorflow for beginners
- Tensorflow for experts
- Building a custom training loop

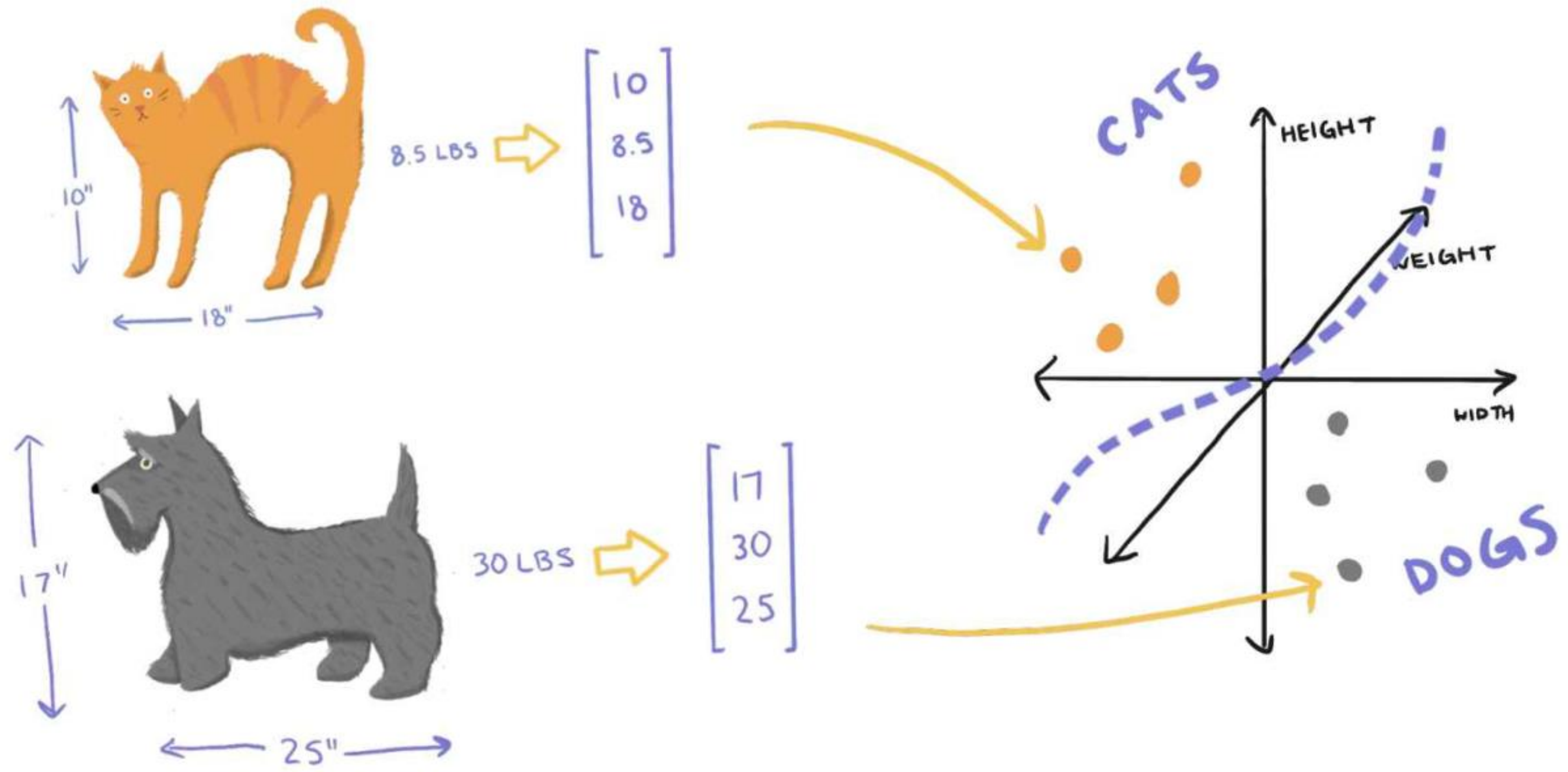
1. Introduction to Deep Learning



1. Introduction to Deep Learning

We usually have:

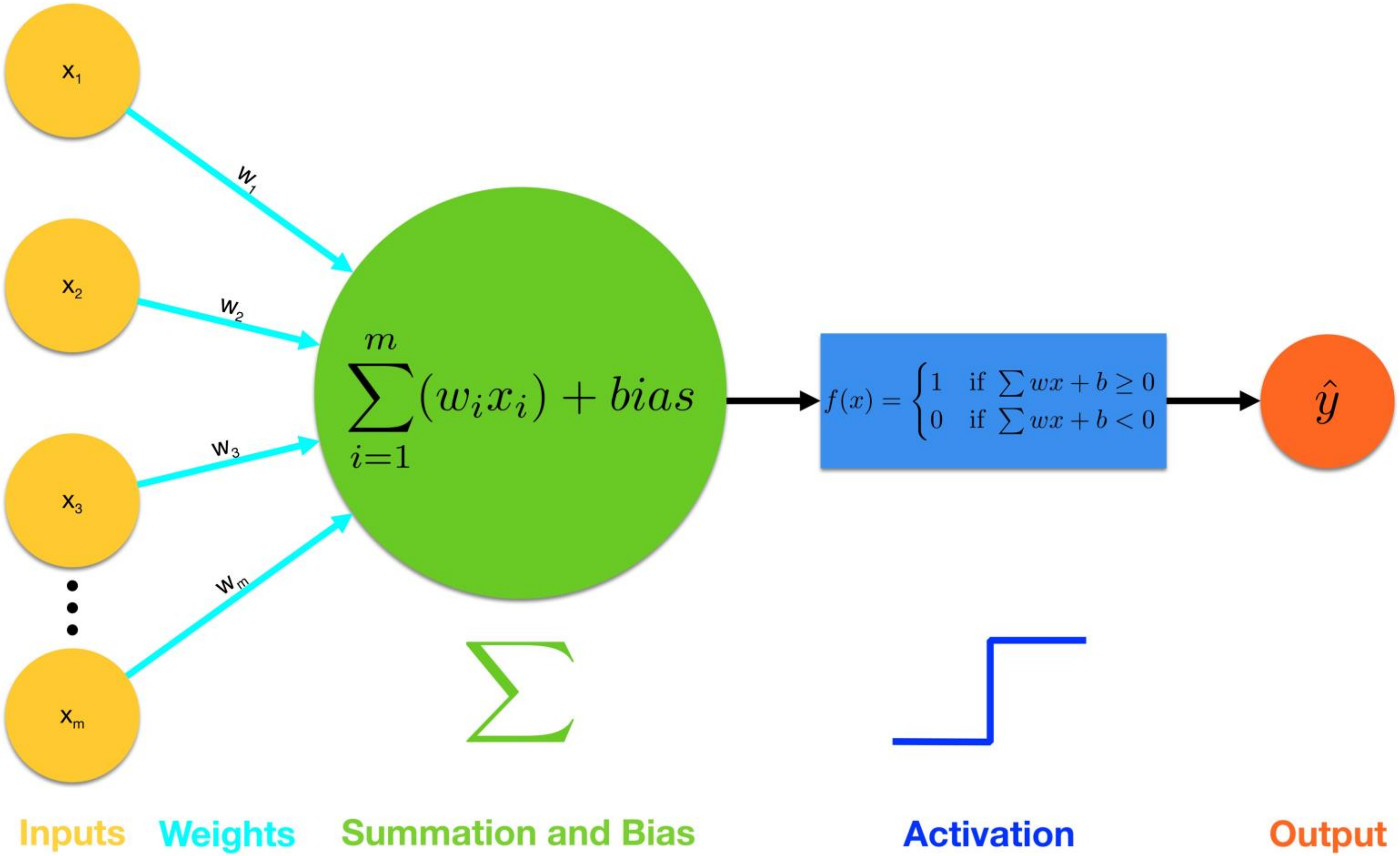
- A set of **Features** : height, weight, width
- A set of **Labels :** Cats, Dogs



NOKIA

1. Introduction to Deep Learning

The basic element is a **Neuron**

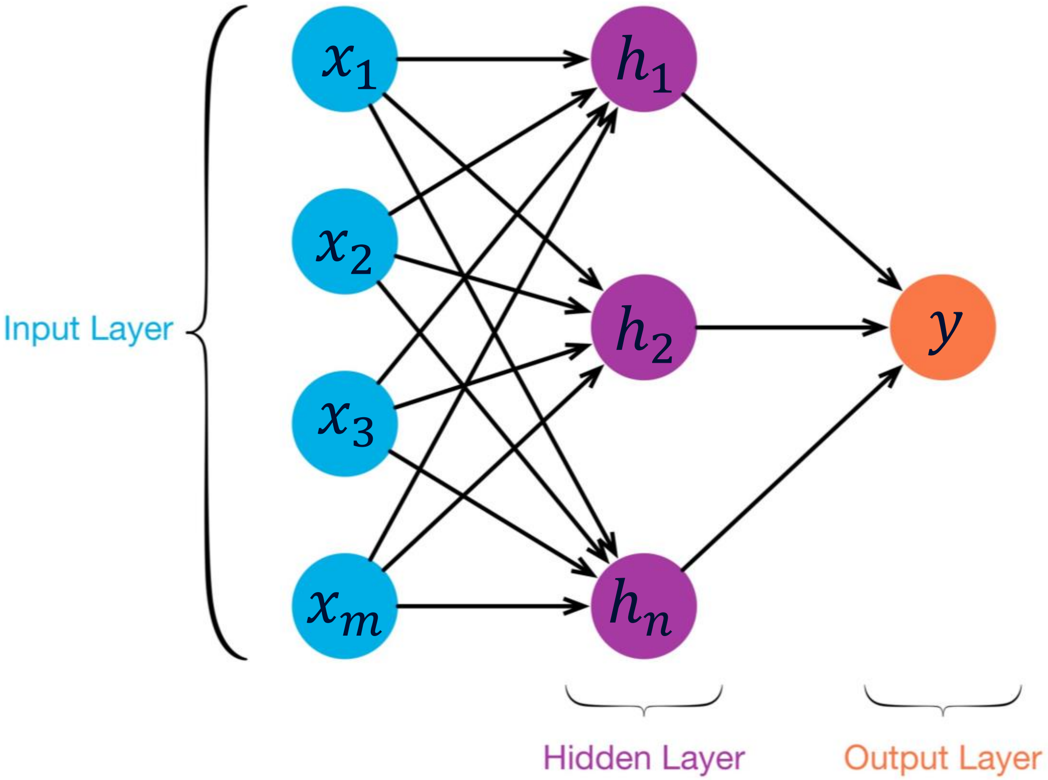


$$\hat{y} = f(w^T x + b)$$

https://towardsdatascience.com/introducing-deep-learning-and-neural-networks-deep-learning-for-rookies-1-bd68f9cf5883
NOKIA

1. Introduction to Deep Learning

Neural Network with 1 hidden **dense layer**



$$y = f(\boldsymbol{W_o}(f(\boldsymbol{W_h x} + \boldsymbol{b_h}) + \boldsymbol{b_o})$$

https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f

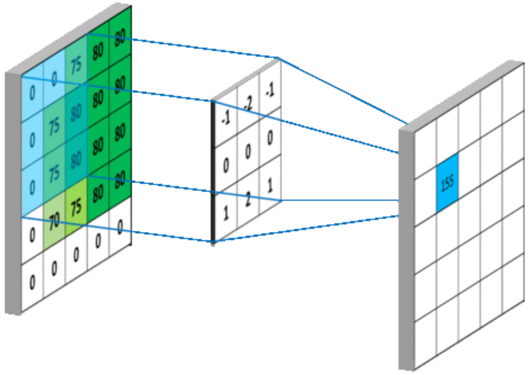
1. Introduction to Deep Learning

Other types of layers:

- **2D Convolutional :**

  Multiplies the 2D inputs by N kernels creating N 2D outputs

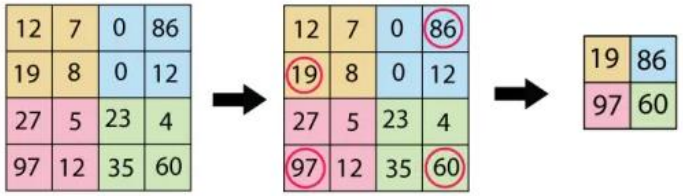
  Parameters : # of kernels N, kernel size, …



- **2D Max Pooling :**

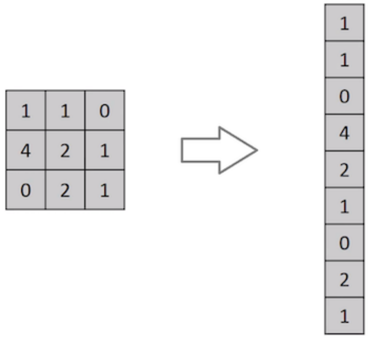
  Down-sample the inputs by taking the maximum of sub-regions

  Parameters : sub-regions sizes, …



- **Flatten :**

  Converts the inputs into a dimension [batch_size, -1]

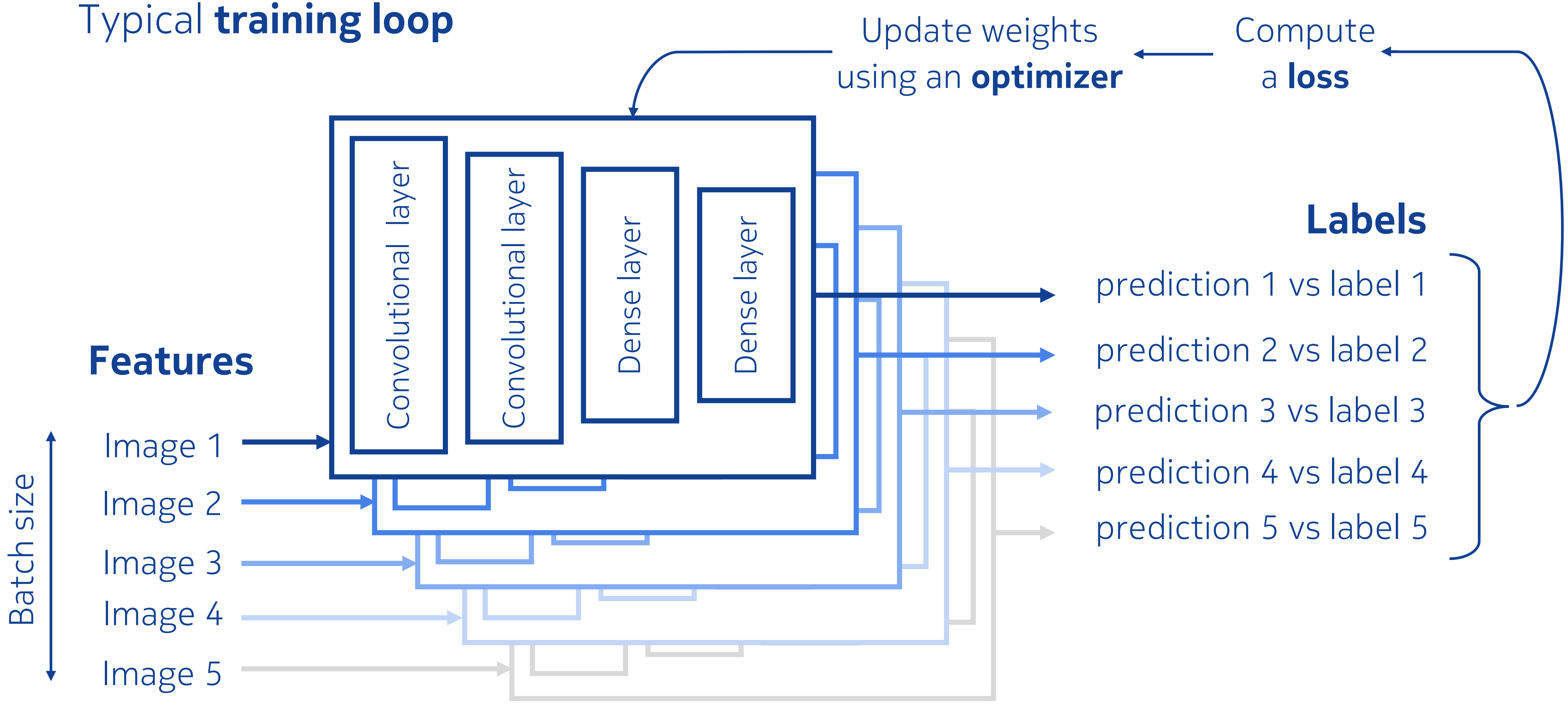


- **Etc.**

https://mlnotebook.github.io/post/CNN1/

1. Introduction to Deep Learning

Typical **training loop**



Multiple copy of the **same** Neural Network

1. Introduction to Deep Learning

The **loss function** depends on the task

(for a batch: compute forward path → compute **loss** & gradients → apply optimizer)

- For a regression problem: Mean Squared Error, etc.

$$\mathrm{MSE} = \frac{1}{B_s}\sum_{i=1}^{B_s}(y_i - \hat{y}_i)^2$$

- For a classification problem : Cross Entropy, etc.

$$\mathrm{CE} = \frac{1}{B_s}\sum_{i=1}^{B_s}\sum_{j=1}^{C} -p_{ij}\ \log(\hat{p}_{ij})$$

Regularization adds a penalty terms depending on the weights:

- L1 regularization increase sparsity

$$Loss = \underbrace{\frac{1}{B_s}\sum_{i=1}^{B_s}(y_i - \hat{y}_i)^2}_{\text{MSE}} + \underset{\text{Tuning parameter}}{\lambda} \underbrace{\sum_{j=1}^{\#\theta}|\theta_j|}_{\text{L1}}$$

- L2 regularization avoids overfitting

$$Loss = \underbrace{\frac{1}{B_s}\sum_{i=1}^{B_s}(y_i - \hat{y}_i)^2}_{\text{MSE}} + \underset{\text{Tuning parameter}}{\lambda} \underbrace{\sum_{j=1}^{\#\theta}{\theta_j}^2}_{\text{L2}}$$

NOKIA

1. Introduction to Deep Learning

Training is done by Stochastic Gradient Descent (**SGD**) or a variant

(for a batch: compute forward path → compute loss & gradients → apply **optimizer**)

SGD updates the weights in the negative gradient direction

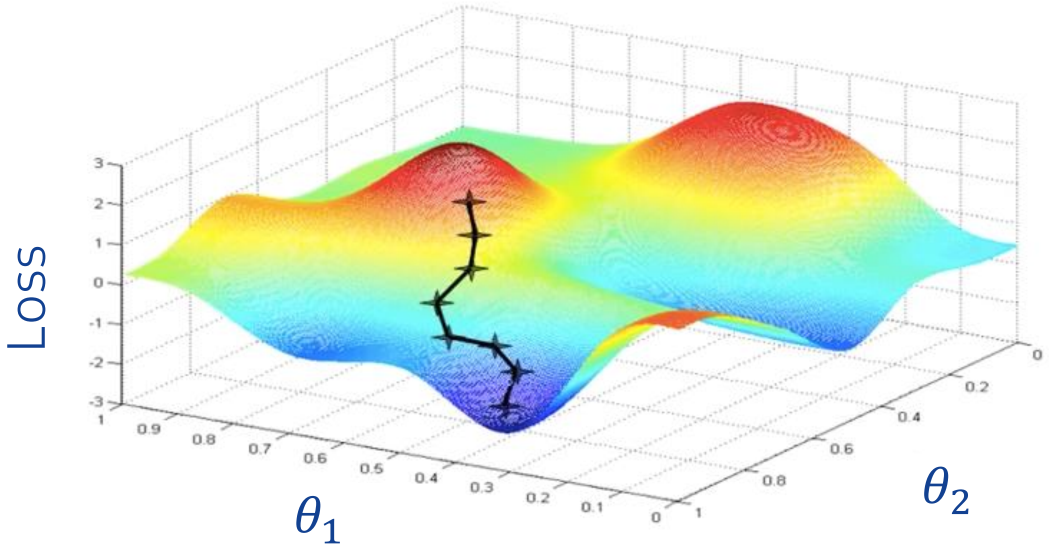
$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t; \hat{y}, y)$$

Learning rate

Gradient of the Loss function

Predictions

Labels



The most used variant is **Adam**:

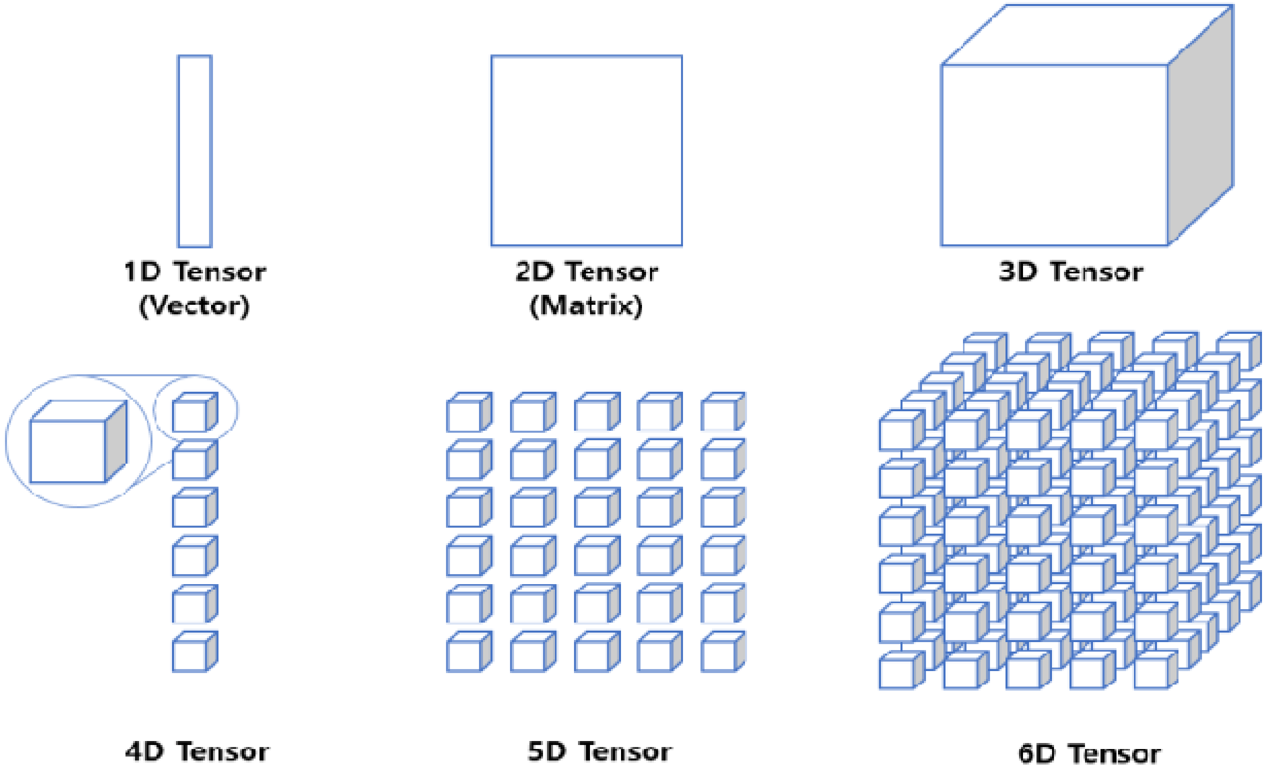
- Individual adaptive learning rate for each parameters
- Exponential moving average of gradients
- Computationally efficient

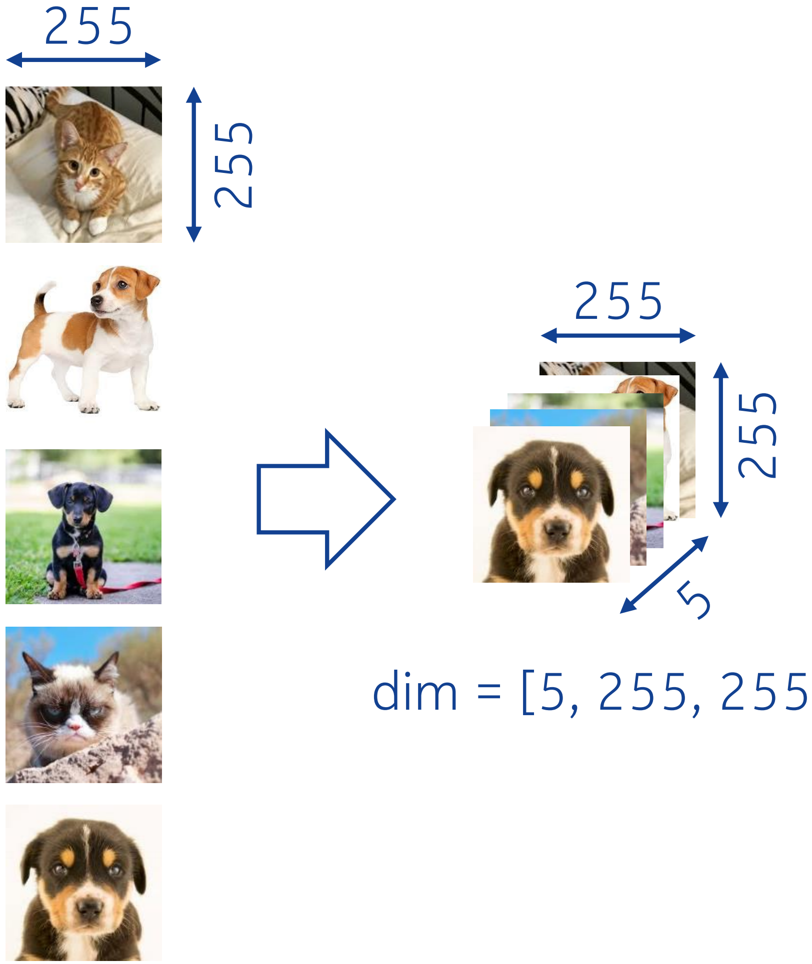
NOKIA

2. Tensorflow for beginners

A **Tensor** is a N-dimensional Matrix

The first dimension is usually the batch size





dim = [5, 255, 255]

NOKIA

2. Tensorflow for beginners

Tensorflow 2.0 is **very pythonic**

Lots of equivalent functions between Numpy & Tensorflow

Numpy

```
import numpy as np

a = np.array([[2, 2], [2, 2]], dtype=np.int32)
b = np.array([[3, 3], [4, 4]], dtype=np.int32)
c = a*b

print(c)
```

```
[[6 6]
 [8 8]]
```

Tensorflow

```
import tensorflow as tf

a = tf.constant([[2, 2], [2, 2]], dtype=tf.int32)
b = tf.constant([[3, 3], [4, 4]], dtype=tf.int32)
c = a*b

print(c)
```

```
tf.Tensor(
[[6 6]
 [8 8]], shape=(2, 2), dtype=int32)
```

Supported TF data types : bool, string, int8/16/32/64, float32/64, complex64/128, etc.

2. Tensorflow for beginners

The parameters of a NN are created as **Variables**

```
cst = tf.constant([[2, 2], [2, 2]]) # cst is a fixed Tensor
var = tf.Variable([[2, 2], [2, 2]]) # var will be updated during training

print('cst:', cst, '\n')
print('var:', var, '\n')
```

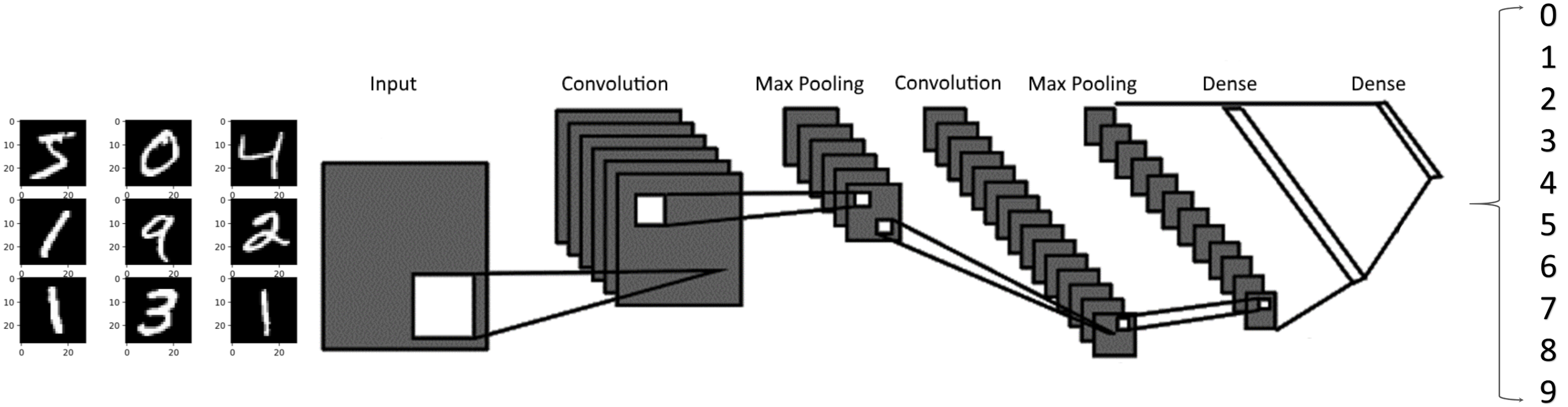
```
cst: tf.Tensor(
[[2 2]
 [2 2]], shape=(2, 2), dtype=int32)

var: <tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
array([[2, 2],
       [2, 2]], dtype=int32)>
```


2. Tensorflow for beginners

Let's play with MNIST : A large database of handwritten digits

Goal : predict the digit given an image



2. Tensorflow for beginners

Preparing the dataset

```
# Load dataset, contains 4 Numpy arrays
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Convert Numpy arrays to Tensors
x_train = tf.convert_to_tensor(x_train, dtype=tf.float32) # [60000, 28, 28]
y_train = tf.convert_to_tensor(y_train, dtype=tf.int32) # [60000]

x_test = tf.convert_to_tensor(x_test, dtype=tf.float32) # [10000, 28, 28]
y_test = tf.convert_to_tensor(y_test, dtype=tf.int32) # [10000]

# Scale the dataset and add a channel dimension
x_train = x_train/255.0
x_train = tf.expand_dims(x_train, axis=-1) # [60000, 28, 28, 1]

x_test = x_test/255.0
x_test = tf.expand_dims(x_test, axis=-1) # [10000, 28, 28, 1]
```


2. Tensorflow for beginners

Keras is a high-level neural networks API

Give access to pre-made Layers

```
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```


2. Tensorflow for beginners

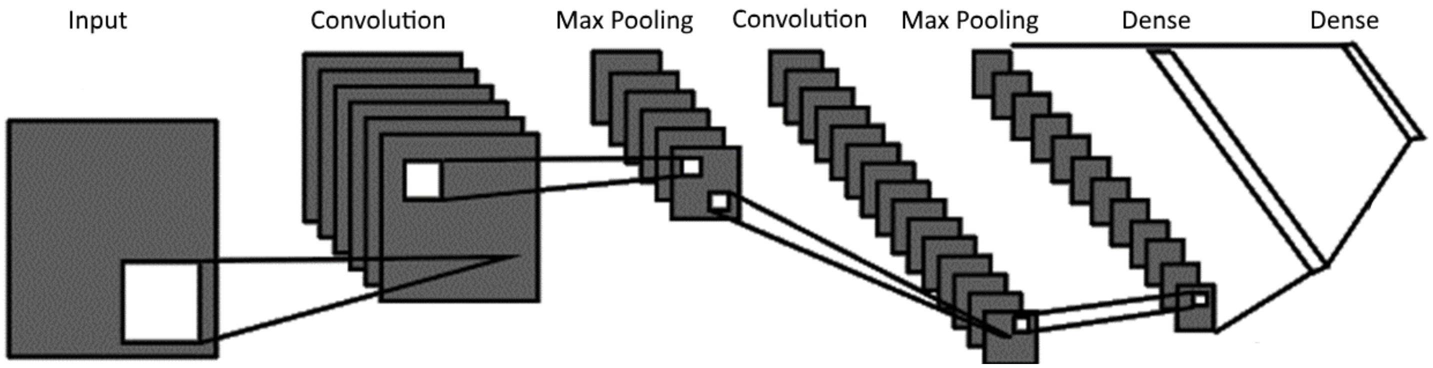
The **Sequential API** : Easily define models

```
my_model = tf.keras.models.Sequential([

    Conv2D(filters=6, kernel_size=8, activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(filters=15, kernel_size=4, activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    Dense(128, activation='relu'),
    Dense(10, activation='softmax') # Outputs a probability distribution
])
```



2. Tensorflow for beginners

Keras gives access to pre-made training functions

```
my_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy', # Only one correct class
              metrics=['accuracy']) # Percentage of good predictions

my_model.fit(x_train, y_train, epochs=3, batch_size=1024)

my_model.evaluate(x_test,  y_test, verbose=2)
```

```
Train on 60000 samples
Epoch 1/3
60000/60000 [==============================] - 4s 59us/sample - loss: 1.3336 - accuracy: 0.6400
Epoch 2/3
60000/60000 [==============================] - 0s 7us/sample - loss: 0.3372 - accuracy: 0.9027
Epoch 3/3
60000/60000 [==============================] - 0s 7us/sample - loss: 0.2316 - accuracy: 0.9322

[0.1858606786608696, 0.9435]
```

Final loss   Final accuracy

NOKIA

3. Tensorflow for experts

Creating a Tensorflow **dataset**

```
BATCH_SIZE = 1024

train_ds = tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(BATCH_SIZE)

test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(BATCH_SIZE)
```

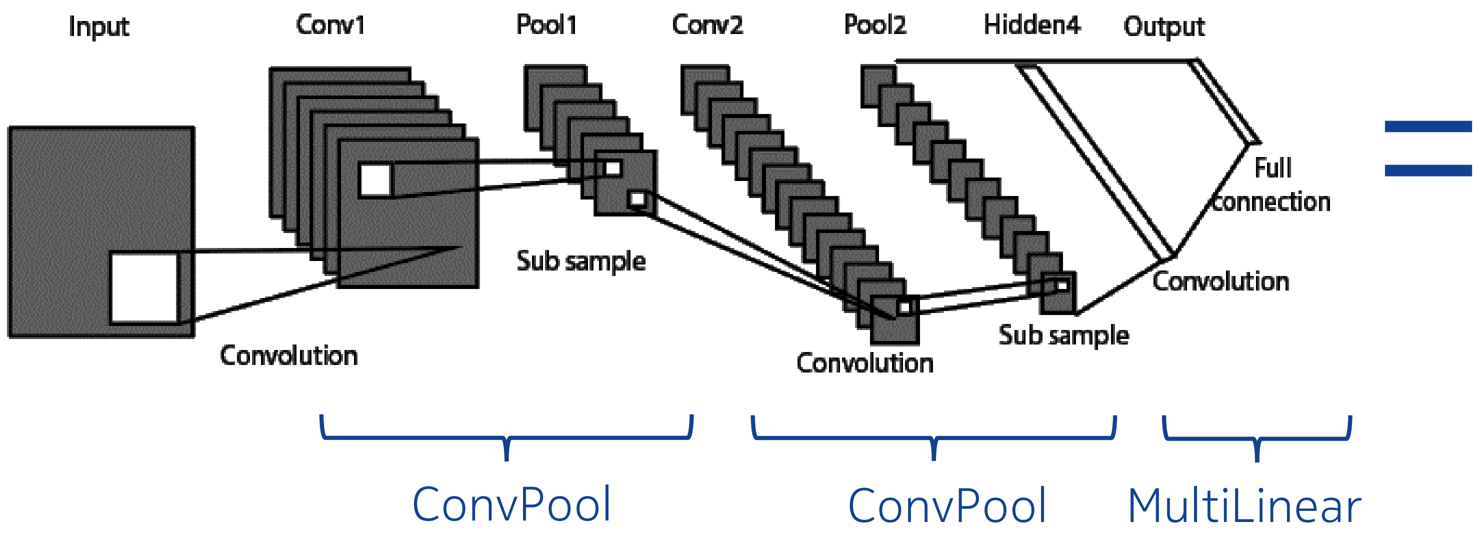
Using TF dataset
helps with parallelization

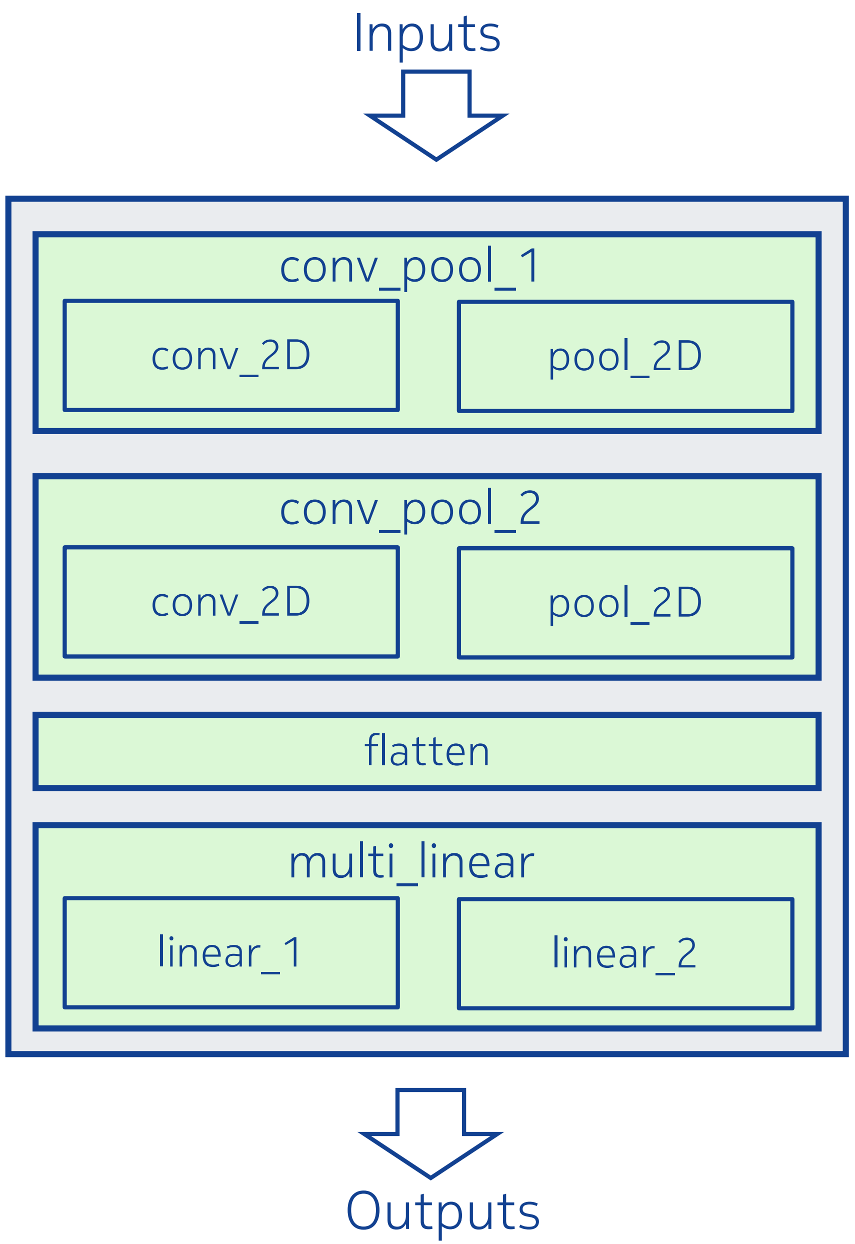
Creates a TF dataset from
given tensors

Splits the dataset into batches
for future training iterations

3. Tensorflow for experts

Creating custom models and layers





3. Tensorflow for experts

The **subclassing API**: defining new layers

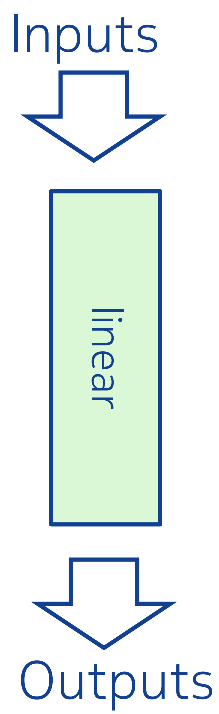
```python
from tensorflow.keras.layers import Layer

class Linear(Layer):
    """y = Wx + b"""

    def __init__(self, units=32): # Called when creating the Layer
        super(Linear, self).__init__()
        self.units = units # units = number of neurons = output shape

    def build(self, input_shape): # Called the first time the layer is used
        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='random_normal', trainable=True)

    def call(self, inputs): # What the layer actually does
        return tf.matmul(inputs, self.W) + self.b
```



NOKIA

3. Tensorflow for experts

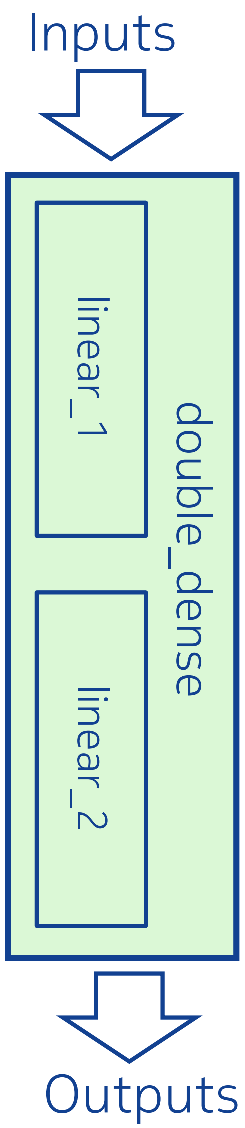
Layers are **recursively composable** with custom Layers

```
class DoubleDense(Layer):
    """" Linear-relu + Linear-Softmax """

    def __init__(self, nb_classes): # Called when creating the layer
        super(DoubleDense, self).__init__()
        self.nb_classes = nb_classes

    def build(self, input_shape):  # Called the first time the layer is used
        self.linear_1 = Linear(units=128)
        self.linear_2 = Linear(units=self.nb_classes)

    def call(self, inputs): # What the layer actually does
        x = tf.nn.relu(self.linear_1(inputs))
        x = tf.nn.softmax(self.linear_2(x)) # Outputs a probability distribution
        return x
```



3. Tensorflow for experts

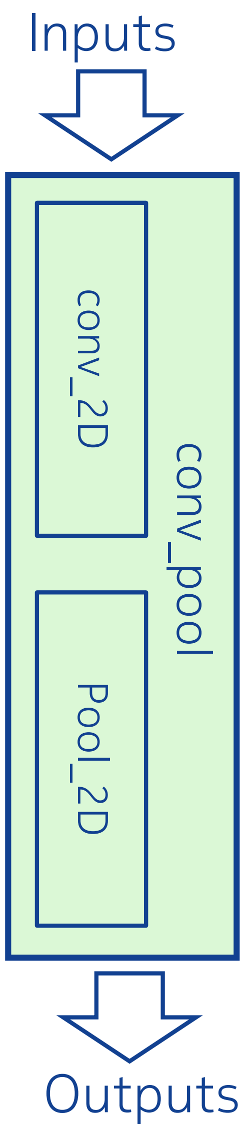
Layers are **recursively composable** with Keras layers

```python
class ConvPool2D(Layer):
    """ Conv2D-relu + MaxPooling2D """

    def __init__(self, nb_kernels, kernel_size): # Called at layer creation
        super(ConvPool2D, self).__init__()
        self.nb_kernels = nb_kernels
        self.kernel_size = kernel_size

    def build(self, input_shape):  # Called the first time the layer is used
        self.conv_2D = Conv2D(filters=self.nb_kernels,
                              kernel_size=self.kernel_size,
                              activation='relu')
        self.pool_2D = MaxPooling2D(pool_size=(2, 2))

    def call(self, inputs): # What the layer actually does
        x = self.conv_2D(inputs)
        x = self.pool_2D(x)
        return x
```



NOKIA

3. Tensorflow for experts

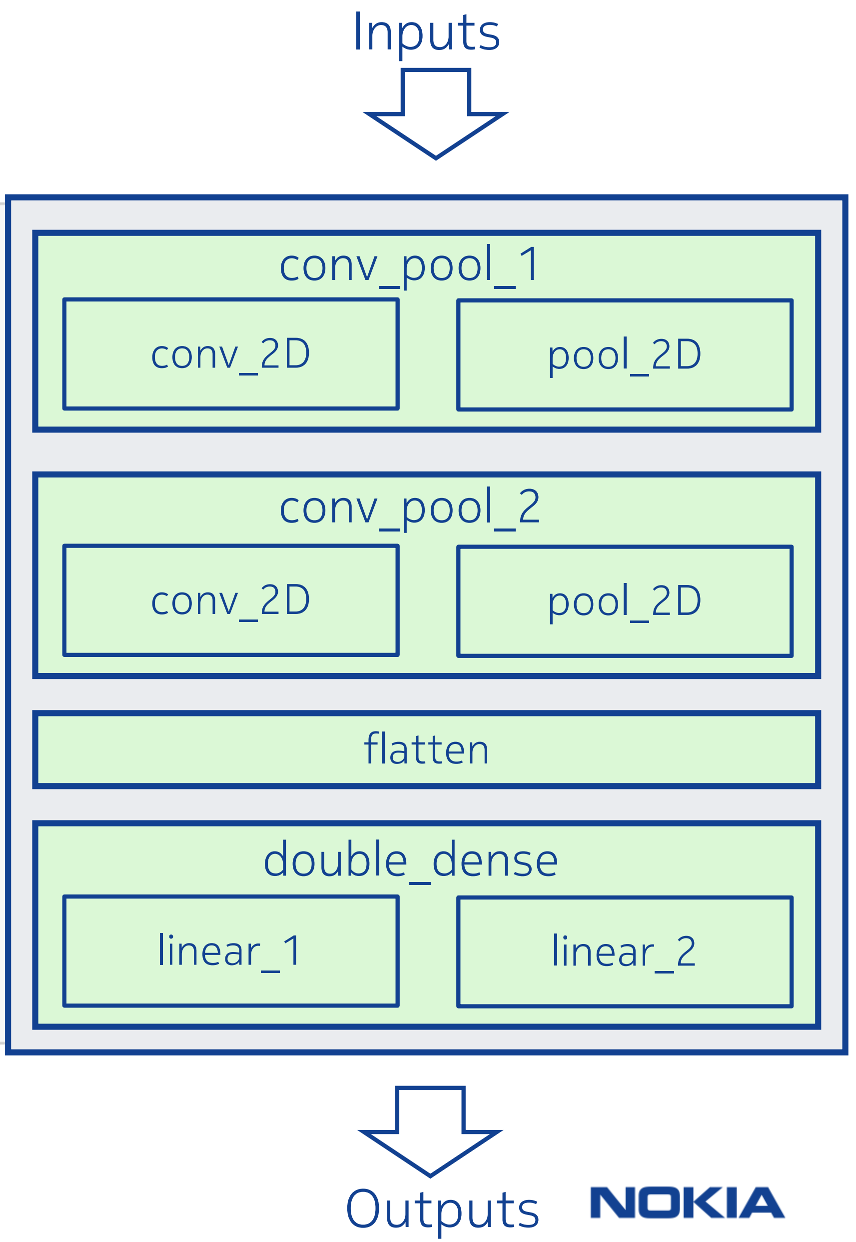
The **subclassing API**: defining new Models

```
from tensorflow.keras import Model

class MyModel(Model):
    def __init__(self, nb_classes):  # Called when creating the model
        super(MyModel, self).__init__()
        self.nb_classes = nb_classes

    def build(self, input_shape):  # Called the first time the layer is used
        self.conv_pool_1 = ConvPool2D(nb_kernels=6, kernel_size=8)
        self.conv_pool_2 = ConvPool2D(nb_kernels=15, kernel_size=4)
        self.flatten = Flatten()
        self.double_dense = DoubleDense(nb_classes=self.nb_classes)

    def call(self, inputs): # What the model actually does
        self.x_0 = self.conv_pool_1(inputs)
        self.x_1 = self.conv_pool_2(self.x_0)
        self.x_2 = self.flatten(self.x_1)
        self.predictions = self.double_dense(self.x_2)
        return self.predictions
```



3. Tensorflow for experts

Creating the model using pre-made functions

```
loss_function = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()

my_model = MyModel(nb_classes=10)
my_model.compile(optimizer, loss_function)
```

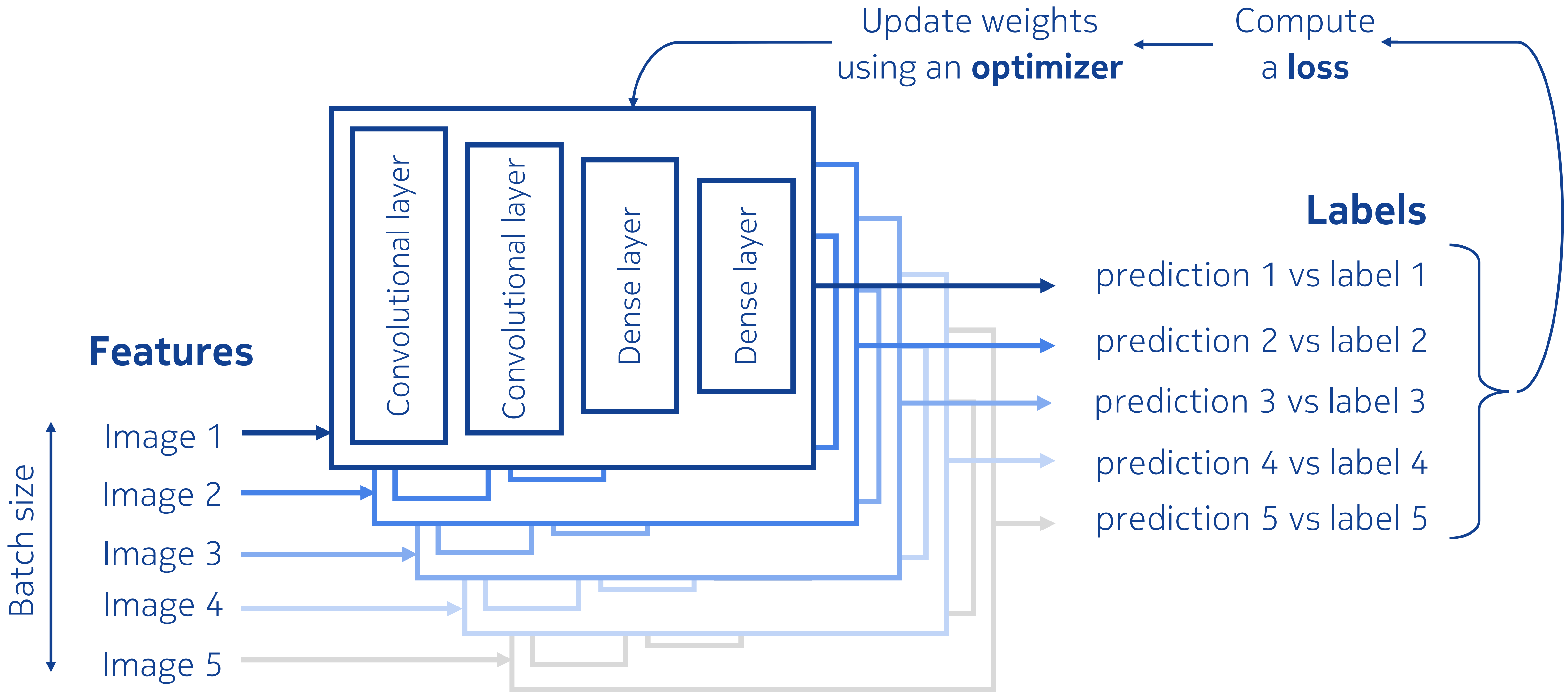
```
my_model.fit(train_ds, epochs=3)
```

```
Epoch 1/3
59/59 [==============================] - 1s 24ms/step - loss: 1.3960
Epoch 2/3
59/59 [==============================] - 1s 14ms/step - loss: 0.4227
Epoch 3/3
59/59 [==============================] - 1s 14ms/step - loss: 0.2977
```

```
my_model.summary()
```

```
Model: "my_model_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv_pool2d_8 (ConvPool2D)   multiple                  390
_________________________________________________________________
conv_pool2d_9 (ConvPool2D)   multiple                  1455
_________________________________________________________________
flatten_4 (Flatten)          multiple                  0
_________________________________________________________________
multi_linear_4 (MultiLinear) multiple                  18698
=================================================================
Total params: 20,543
Trainable params: 20,543
Non-trainable params: 0
_________________________________________________________________
```


4. Building a custom training loop



Multiple copy of the **same** Neural Network

4. Building a custom training loop

Important **metrics** can be logged

```
# Define the loss function
loss_function = tf.keras.losses.SparseCategoricalCrossentropy()

# Define the optimizer
optimizer = tf.keras.optimizers.Adam()

# Define the metrics
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```


4. Building a custom training loop

The **tf.GradientTape API** provides automatic differentiation.
All operations executed inside a gradient tape are recorded.

$$x = 3$$
$$y = x^2$$
$$\frac{dy}{dx} = 2x = 6$$

```
x = tf.Variable(3.0)
with tf.GradientTape() as tape:
    y = tf.square(x)
dy_dx = tape.gradient(y, x)
print(dy_dx)
```

```
tf.Tensor(6.0, shape=(), dtype=float32)
```


NOKIA

4. Building a custom training loop

Creating the training step

```
# One SGD step with a given batch
def train_step(images, labels):

    # Open a GradientTape
    with tf.GradientTape() as tape:

        #Forward pass
        predictions = my_model(images)
        # Loss for this batch
        loss = loss_function(labels, predictions)

    # Get gradients of loss w.r.t. the weights
    gradients = tape.gradient(loss, my_model.trainable_variables)
    # Update the weights according to our optimizer
    optimizer.apply_gradients(zip(gradients, my_model.trainable_variables))

    # Save loss and accuracy
    train_loss(loss)
    train_accuracy(labels, predictions)
```

```
# Test the model on a given batch
def test_step(images, labels):

    # Forward pass
    predictions = my_model(images)
    # Loss for this batch
    t_loss = loss_function(labels, predictions)

    # Save loss and accuracy
    test_loss(t_loss)
    test_accuracy(labels, predictions)
```

4. Building a custom training loop

Creating the training loop

```
my_model = MyModel(nb_classes=10)
start = time.time()

# Iterate over 3 epochs
for epoch in range(3):
    # Train over every batch in the training dataset
    for images, labels in train_ds:
        train_step(images, labels)

    # Test over every batch in the testing dataset
    for test_images, test_labels in test_ds:
        test_step(test_images, test_labels)

    # Print result
    template = 'Epoch {:.0f},   Loss: {:.3f}, Accuracy: {:.3f}    '+ \
               'Test Loss: {:.3f}, Test Accuracy: {:.3f}'
    print(template.format(epoch+1, train_loss.result(), train_accuracy.result()*100,
                          test_loss.result(), test_accuracy.result()*100))

    # Reset the metrics for the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()

# Display elapsed time
end = time.time()
print('TIME = ', end - start)
```

```
Epoch 1,   Loss: 1.383, Accuracy: 59.863    Test Loss: 0.500, Test Accuracy: 84.560
Epoch 2,   Loss: 0.409, Accuracy: 87.717    Test Loss: 0.296, Test Accuracy: 91.140
Epoch 3,   Loss: 0.279, Accuracy: 91.637    Test Loss: 0.219, Test Accuracy: 93.480
TIME =  6.866998195648193
```


4. Building a custom training loop

Building a **graph** to speed up training

by adding *@tf.function* before the training and testing functions

```
# One SGD step with a given batch
@tf.function
def train_step(images, labels):
```

```
# Test the model on a given batch
@tf.function
def test_step(images, labels):
```

	Without @tf.function	**With @tf.function**
Training time for 3 epoch	~6.86s	~3.46s

But we loose access to the value of the model’s attributes ☹

```
# Without @tf.function
my_model.x_2
```

```
<tf.Tensor: id=107279, shape=(784, 13
5), dtype=float32, numpy=
array([[0.78254956, 3.1596248 , 2.98792
89 , ..., 1.6538126 , 1.9493525 ,
        0.24810283],
```

```
# With @tf.function
my_model.x_2
```

```
<tf.Tensor 'my_model_9/flatten/Reshape:
0' shape=(784, 135) dtype=float32>
```


NOKIA

4. Building a custom training loop

Adding **regularization** to the loss function

- Using Keras layers' parameters :

```
def build(self, input_shape):  # Called the first time the layer is used
    self.conv_2D = Conv2D(filters=self.nb_kernels,
                          kernel_size=self.kernel_size,
                          activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l1(l=1.))
```

- And/or using custom layers' *loss* property :

```
def call(self, inputs): # What the layer actually does
    self.l1_reg = tf.reduce_sum(tf.abs(self.W)) + tf.reduce_sum(tf.abs(self.b))
    self.add_loss(self.l1_reg)
```

- And adding those losses to the training loop :

```
predictions = my_model(images)
# Loss for this batch
loss = loss_function(labels, predictions)
# Add extra losses created during this forward pass:
loss += 1e-3 * sum(my_model.losses)
```

4. Building a custom training loop

Building a **custom loss function**

- Create a new class

```
from tensorflow.keras.losses import Loss

class CustomLoss(Loss):
    """ Custom Sparse Cross Entropy loss with L1 regularization """

    def __init__(self, tuning_param): # Called when creating the layer
        super(CustomLoss, self).__init__()
        self.tuning_param = tuning_param
        self.SCE = tf.keras.losses.SparseCategoricalCrossentropy()

    def call(self, y_true, y_pred): # What the loss function actually does
        return self.SCE(y_true, y_pred) + self.tuning_param * sum(my_model.losses)
```

- Instantiate the class

```
cust_loss_function = CustomLoss(tuning_param=1e-3)
```

- Change the loss in the training loop

```
predictions = my_model(images)
# Loss for this batch
loss = cust_loss_function(labels, predictions)
```


NOKIA

Find a working example at

https://mgoutay.github.io/tutorials/2019/11/14/tf2-tutorial.html

Any questions ?

