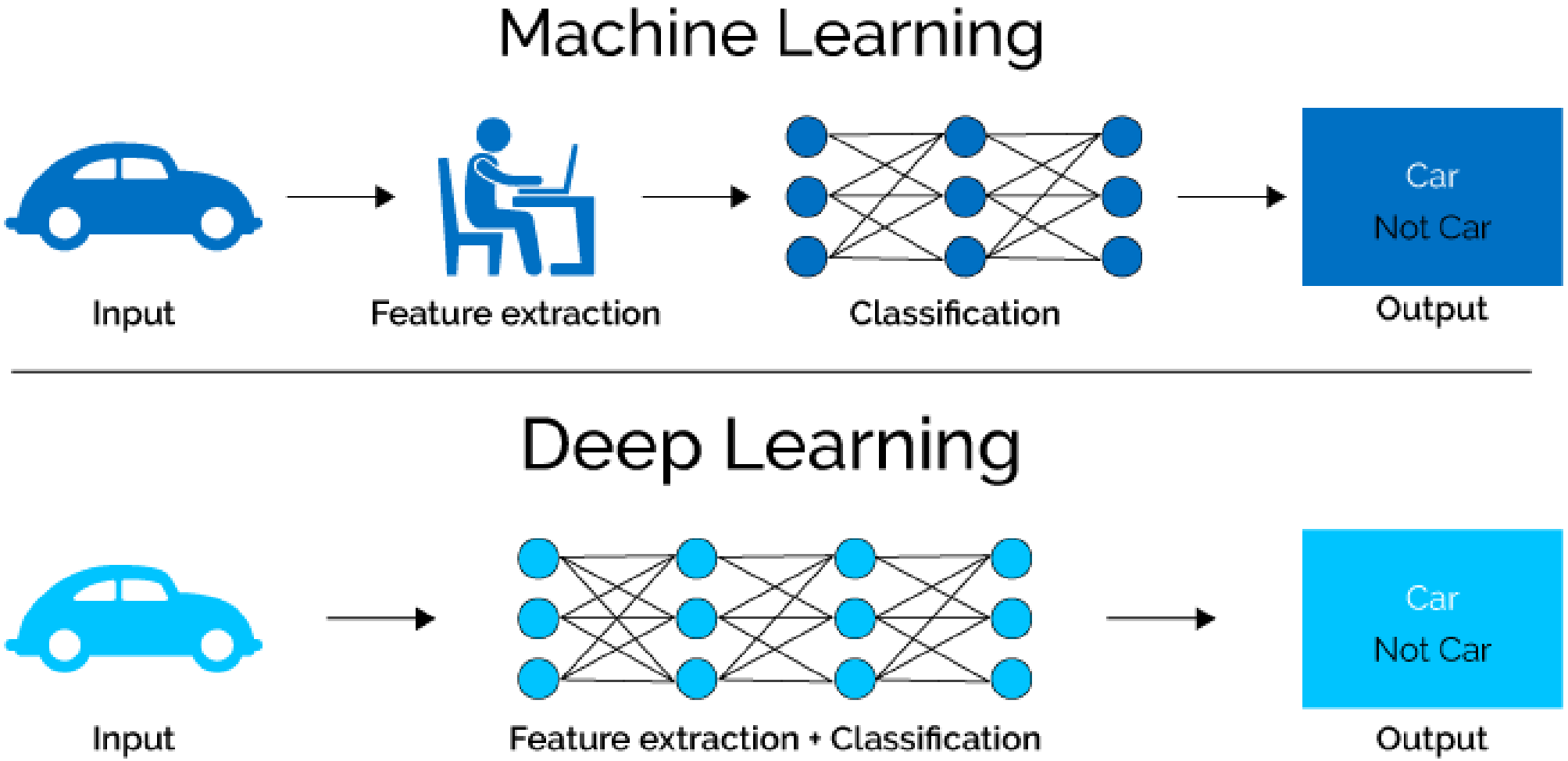
# Convolutional Neural Network (CNN)

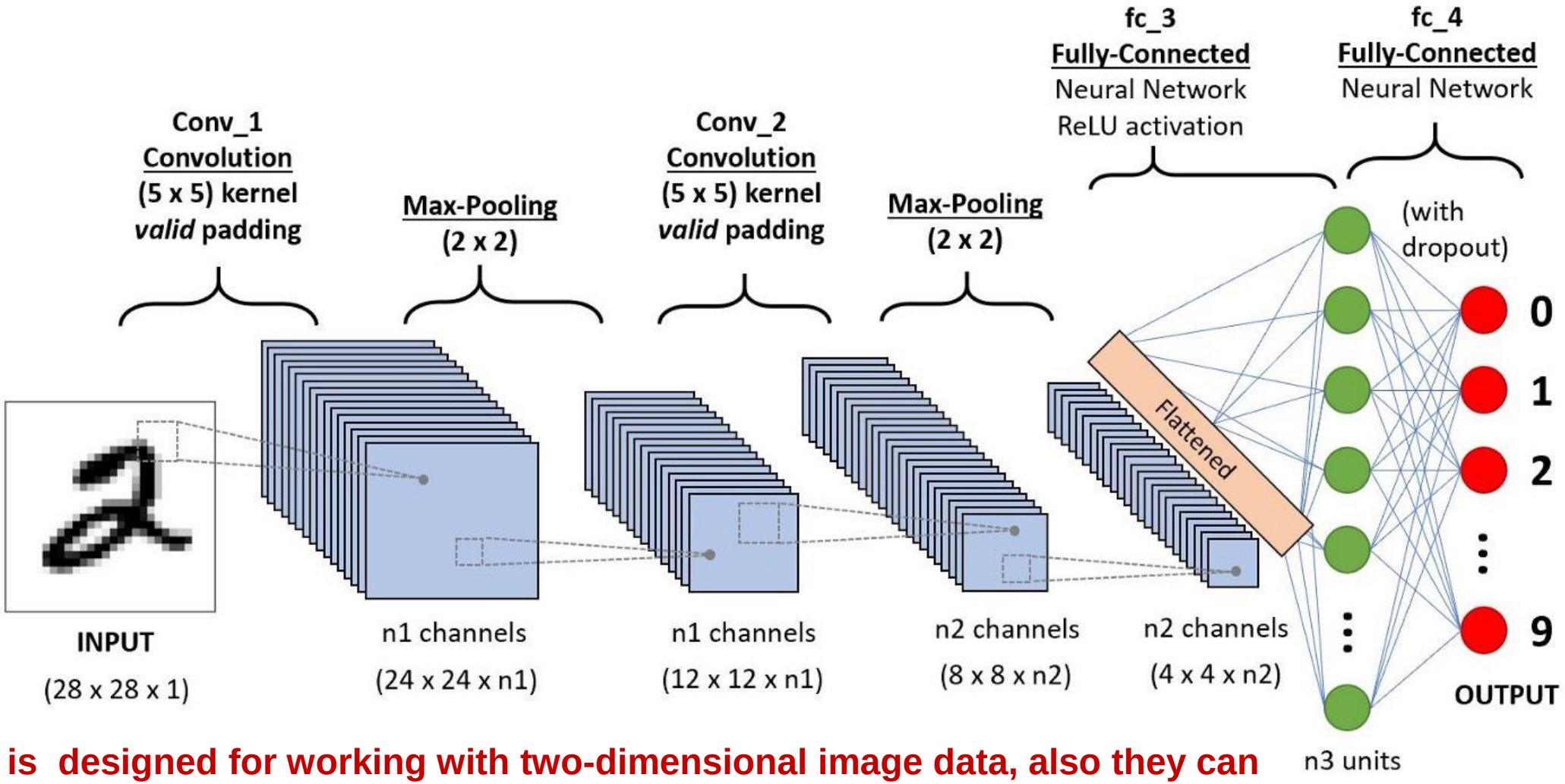
## Keras & TensorFlow

*Hichem Felouat*
*hichemfel@gmail.com*
*https://www.linkedin.com/in/hichemfelouat/*

# Machine Learning VS Deep Learning



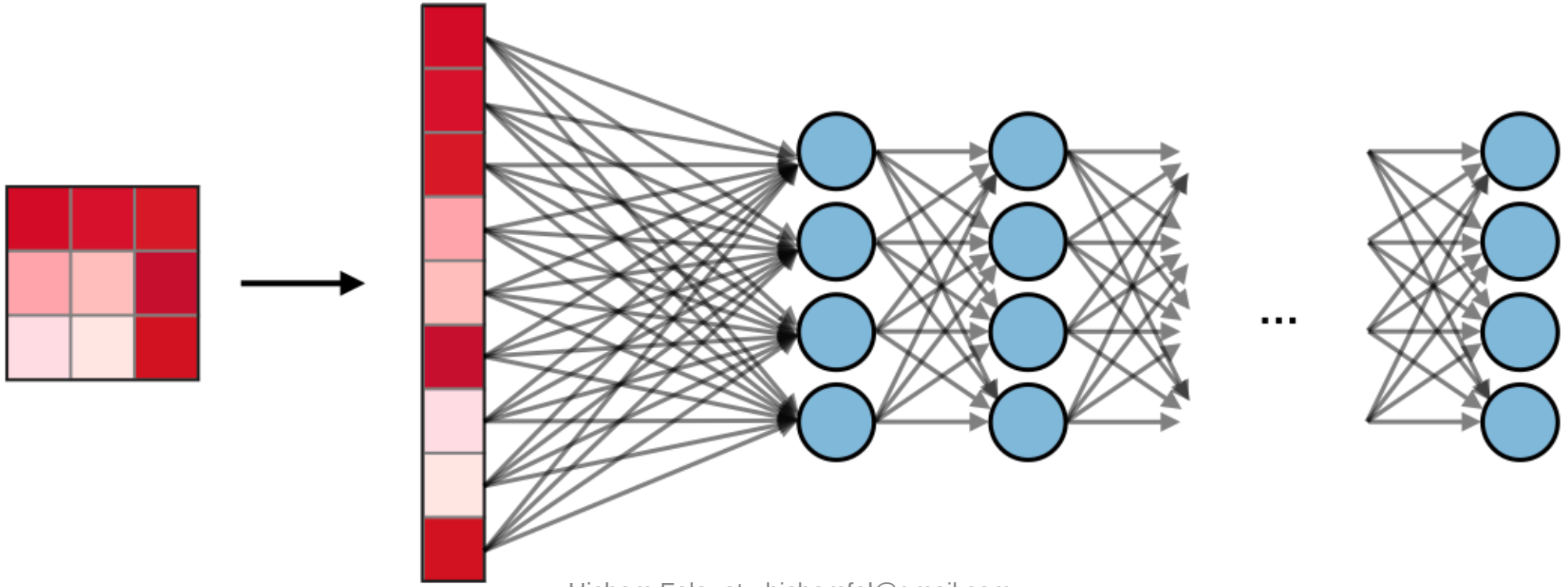
# CNN Architecture




**CNN is designed for working with two-dimensional image data, also they can be used with one-dimensional and three-dimensional data.**

# Fully Connected (FC)

**The fully connected layer (FC)** operates on a **flattened input** where each input is connected to all neurons. If present, FC layers are usually found towards the **end of CNN architectures** and can be used to **optimize** objectives such as class scores.



# Fully Connected Layers

## Typical classification model architecture

| Hyperparameter | Binary classification | Multilabel binary classification | Multiclass classification |
|---|---|---|---|
| input neurons | One per input feature | One per input feature | One per input feature |
| hidden layers neurons per hidden layer | Depends on the problem | Depends on the problem | Depends on the problem |
| output neurons | 1 | 1 per label | 1 per class |
| Hidden activation | ReLU (relu) | ReLU (relu) | ReLU (relu) |
| Output layer activation | Logistic (sigmoid) | Logistic (sigmoid) | Softmax (softmax) |
| Loss function | **binary_crossentropy** | **binary_crossentropy** | Cross entropy |

**Multiclass classification : sparse_categorical_crossentropy [Labels are Integers ]**
**categorical_crossentropy [Labels are one-hot ]**

# Convolutional Layers

- Padding ?
- Stride ?
- Activation Function ?

(0*0) + (-1*0) + (0*0)
(-1*0) + (5*60) + (-1*113)
(0*0) + (-1*73) + (0*121)
= 114

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 60 | 113 | 56 | 139 | 85 | 0 |
| 0 | 73 | 121 | 54 | 84 | 128 | 0 |
| 0 | 131 | 99 | 70 | 129 | 127 | 0 |
| 0 | 80 | 57 | 115 | 69 | 134 | 0 |
| 0 | 104 | 126 | 123 | 95 | 130 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Kernel

| | | |
|---|---|---|
| 0 | -1 | 0 |
| -1 | 5 | -1 |
| 0 | -1 | 0 |

| | | | | |
|---|---|---|---|---|
| 114 | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

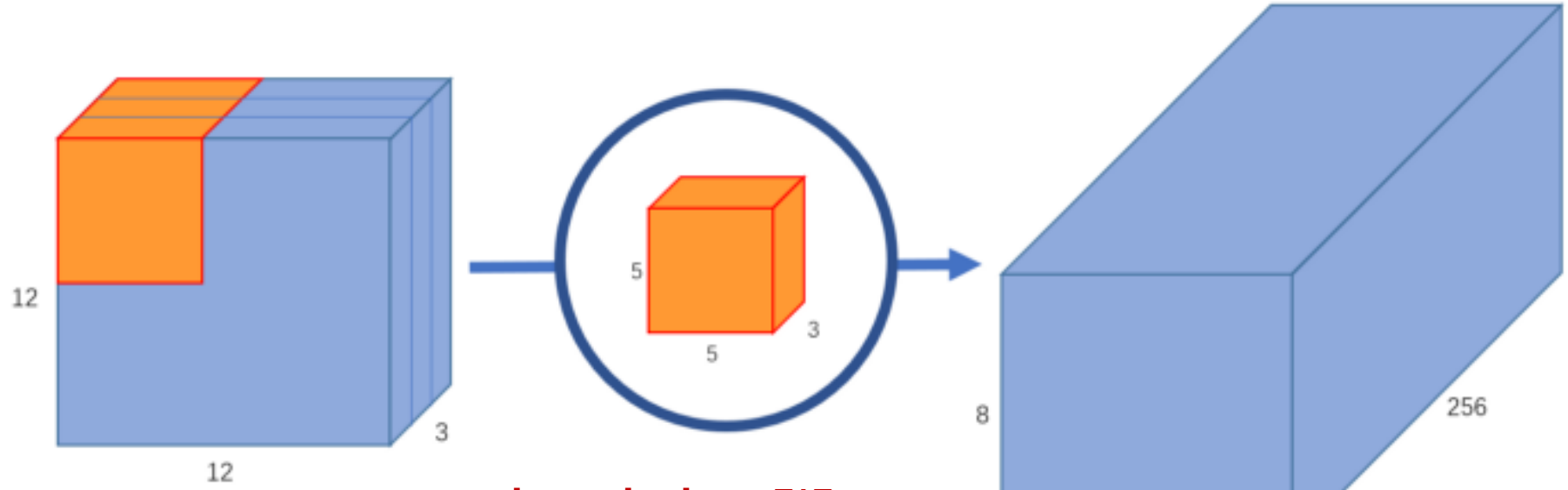
# Convolutional Layers

Since we typically use **small kernels**, for any given convolution, we might only **lose a few pixels**, but this can add up as we apply many successive convolutional layers.

- One straightforward solution to this problem is to **add extra pixels** of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, **we set the values of the extra pixels to zero**, (**padding="same"**).
- Without padding: **padding="valid"**

# Convolutional Layers

- What if we want to **increase the number of channels** in our output image? What if we want an output of size **8x8x256**?
- Well, we can create **256 kernels** to create **256 8x8x1** images, then stack them up together to create a 8x8x256 image output.




**RGB image 12*12**

**kernel_size= 5*5**

**output = 8*8*256**

# Convolutional Layers

Calculate the output size in the convolution layer? We can use this formula (output height or output width): **[(W−K+2P)/S]+1**.

W : is the input volume

K : is the Kernel size

S : is the stride

F : is n_filters

**Pooling layer:**

$$W_{out} = [(W-K)/S]+1$$

**input image:** W*H = 12*12, K = 5, S = 1, n_filters = 256

**output :**

W = [(w-k)/s]+1 = [(12-5)/1]+1 = 8

H = [(h-k)/s]+1 = [(12-5)/1]+1 = 8

output image = H*W*n_filters = **8*8*256**

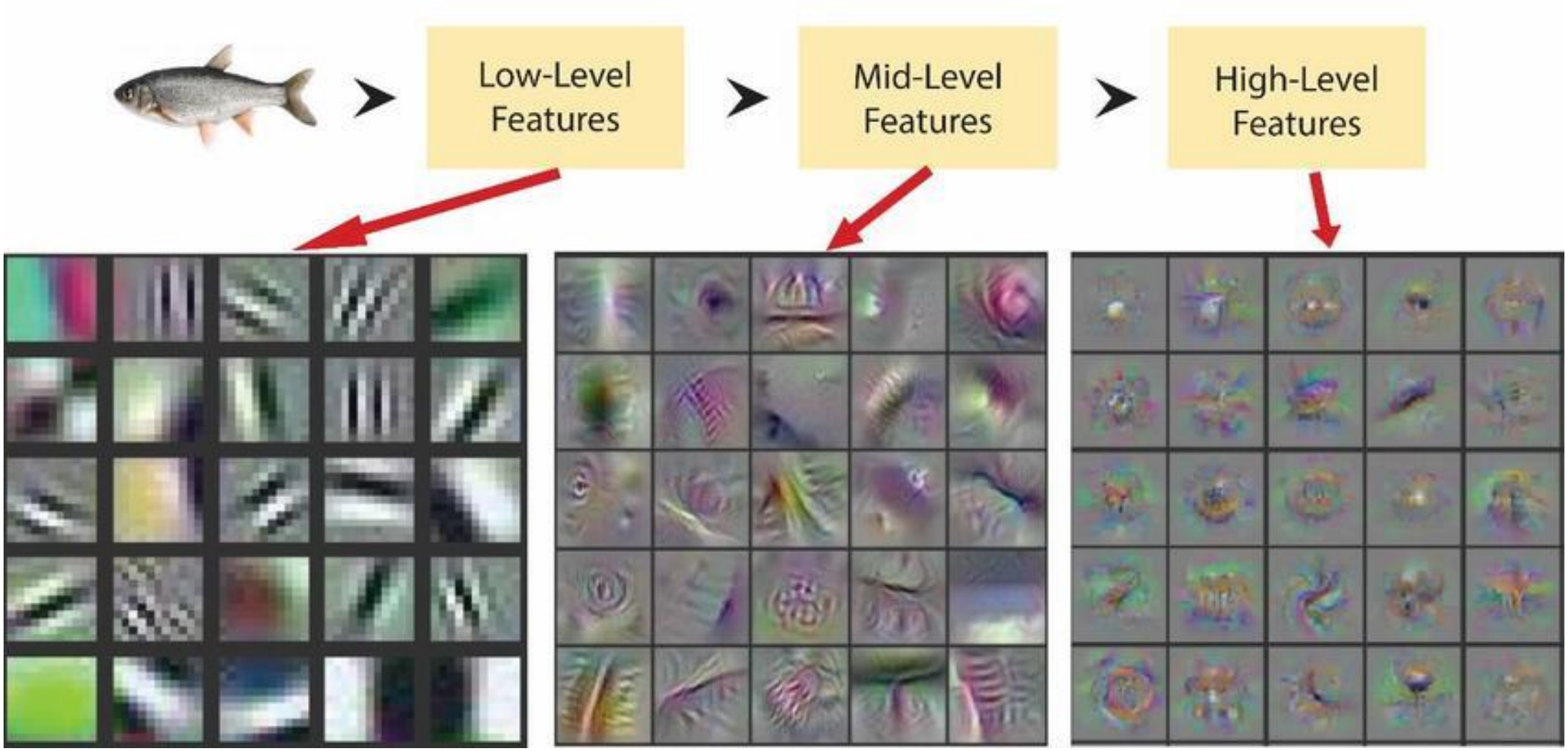
# Convolutional Layers

- The role of the Convolutional Layer is to **reduce the images into a form that is easier to process**, **without losing features** that are critical for getting a good prediction.
- This architecture allows the network to **concentrate on small low-level features** in the first hidden layer, then assemble them into **larger higher-level features** in the next hidden layer, and so on.

```
# padding : same or valid
# in the input layer, we need to specify the input size: nb_ch = number of channels (RGB = 3)
conv2D = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="same",
                                                    activation="relu",  input_shape=(H, W, nb_ch))

conv3D = keras.layers.Conv3D(filters=32, kernel_size=3, strides=1, padding="same",
                                                    activation="relu", input_shape=(128,128,128,3))
```

# Convolutional Layers

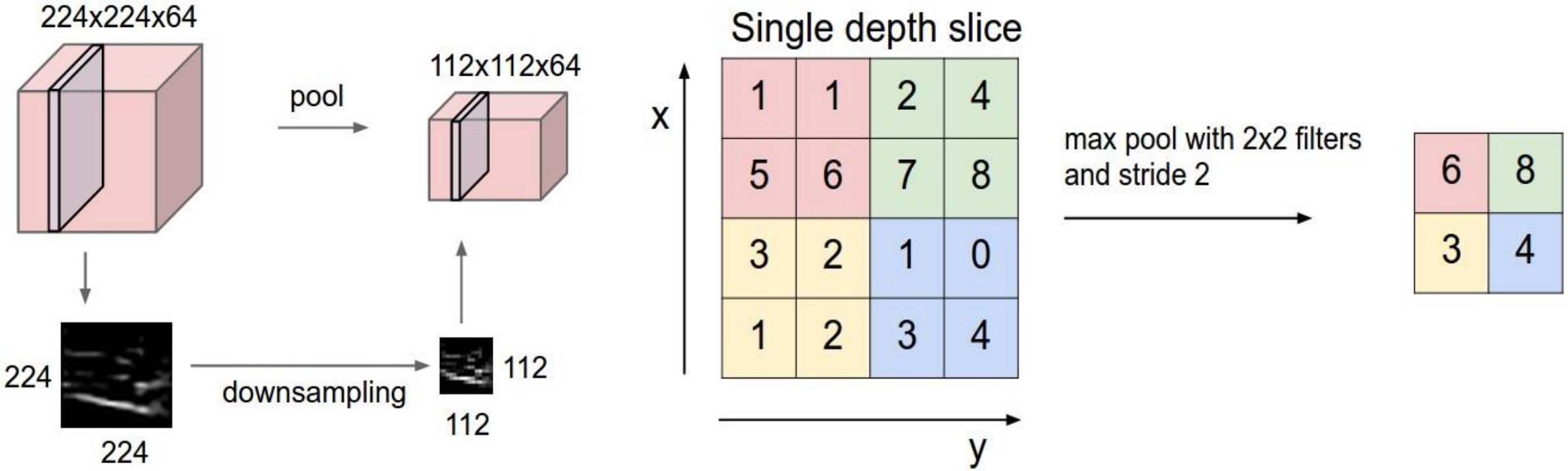


# Convolutional Layers


Original Image

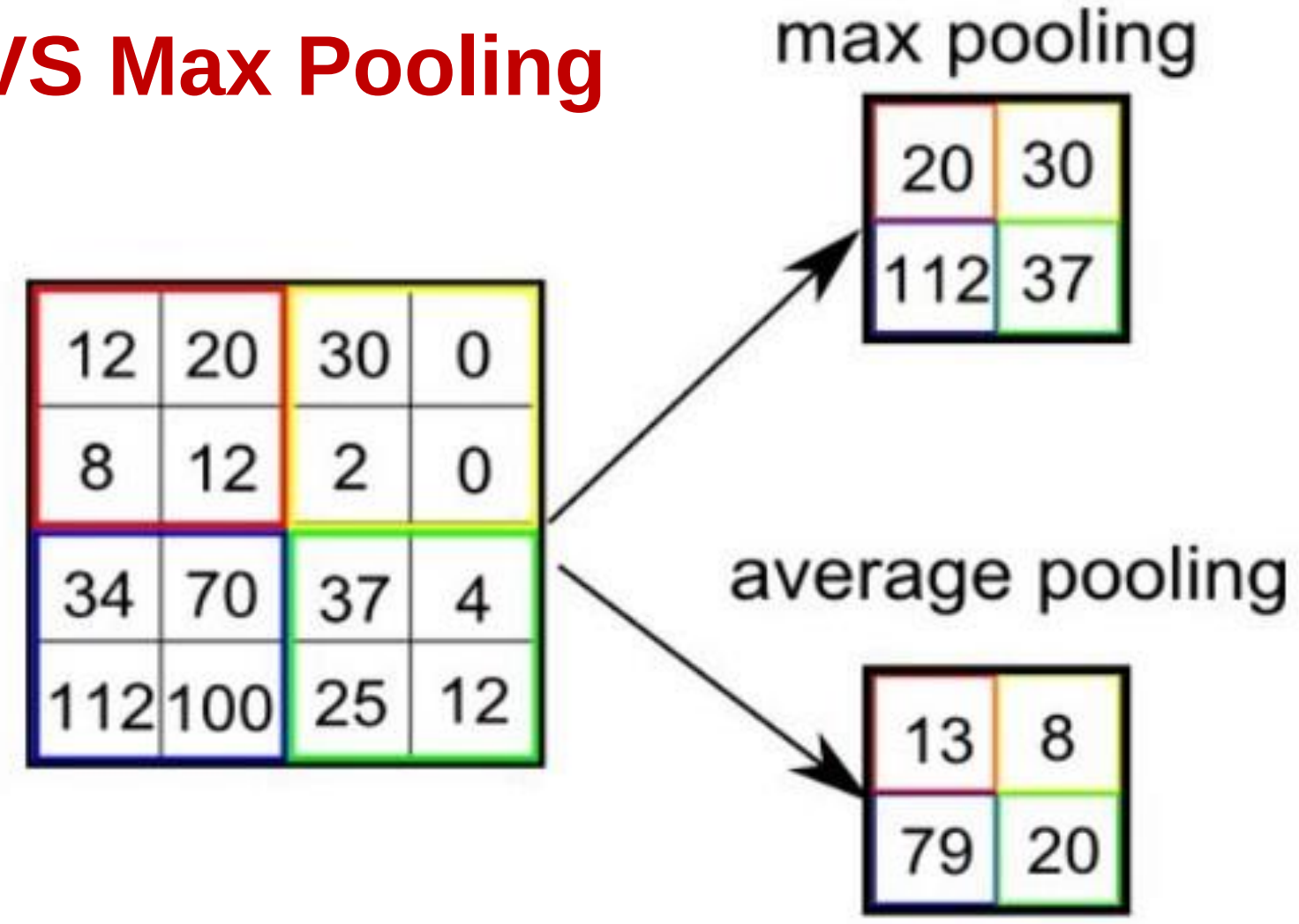

Result of Conv2D

# Pooling Layers



# Pooling Layers

- **Average VS Max Pooling**



# Pooling Layers

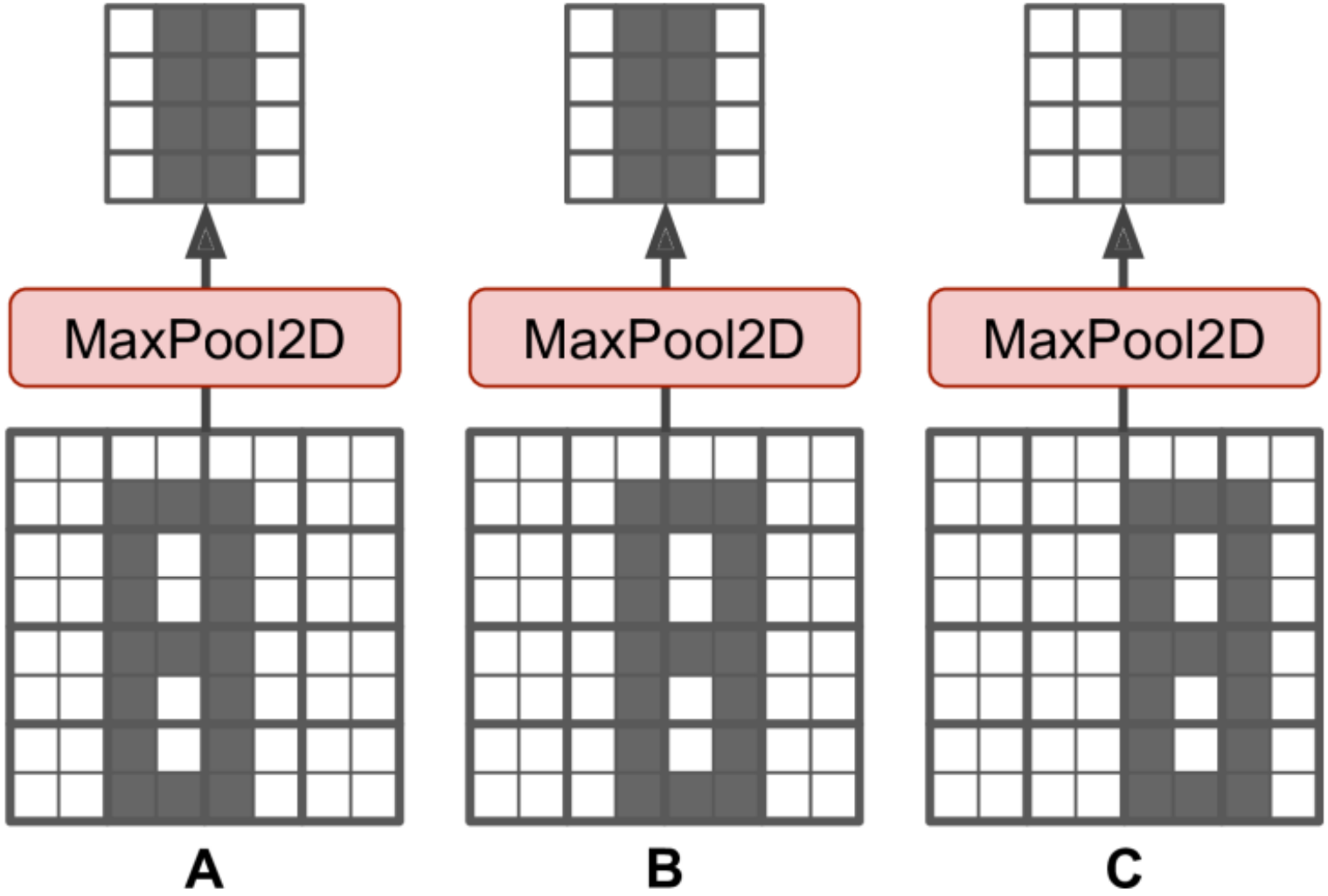
- Reduce the **computational load**, **memory usage**, and **the number of parameters** (limiting the risk of overfitting).

- Max pooling layer also introduces some level of invariance to **small translations**.

```
max_pool = keras.layers.MaxPool2D(pool_size=2)

max_pool = keras.layers.MaxPool3D(pool_size=2)
```

# Pooling Layers



# CNN

```
# Creating the model using the Sequential API
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(filters=64, kernel_size=5, strides=1, padding="same", activation="relu",
                              input_shape= (128,128,3)))
model.add(keras.layers.MaxPool2D(pool_size=2))
model.add(keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, padding="same", activation="relu"))
model.add(keras.layers.MaxPool2D(pool_size=2))
model.add(keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, padding="same", activation="relu"))
model.add(keras.layers.MaxPool2D(pool_size=2))

model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(512, activation="relu"))
model.add(keras.layers.Dense(128, activation="relu"))
model.add(keras.layers.Dense(1, activation="sigmoid"))

# The model's summary() method displays all the model's layers
print(model.summary())

# Compiling the model
model.compile(loss="binary_crossentropy", optimizer= "sgd", metrics=["accuracy"])
# Training the model
history = model.fit(X_train, y_train, epochs=30, batch_size=32, validation_split=0.2)
```

# CNN - 3D Conv3D & MaxPool3D

```
# Creating the model using the Sequential API
model = keras.models.Sequential()
model.add(keras.layers.Conv3D(filters=64, kernel_size=7, strides=1,
                              padding="same", activation="relu", input_shape= (128,128,64,3)))
model.add(keras.layers.MaxPool3D(pool_size=2))
model.add(keras.layers.Conv3D(filters=128, kernel_size=5, strides=1,
                              padding="same", activation="relu"))
model.add(keras.layers.MaxPool3D(pool_size=2))
model.add(keras.layers.Conv3D(filters=256, kernel_size=3, strides=1,
                              padding="same", activation="relu"))
model.add(keras.layers.MaxPool3D(pool_size=2))

model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(512, activation="relu"))
model.add(keras.layers.Dense(128, activation="relu"))
model.add(keras.layers.Dense(1,   activation="sigmoid"))

# The model's summary() method displays all the model's layers
print(model.summary())

```

# Avoiding Overfitting

1) Try tuning model hyperparameters such as (**the number of layers**, **the number of neurons per layer**, the **types of activation functions** to use for each hidden layer, and **the optimizer**.

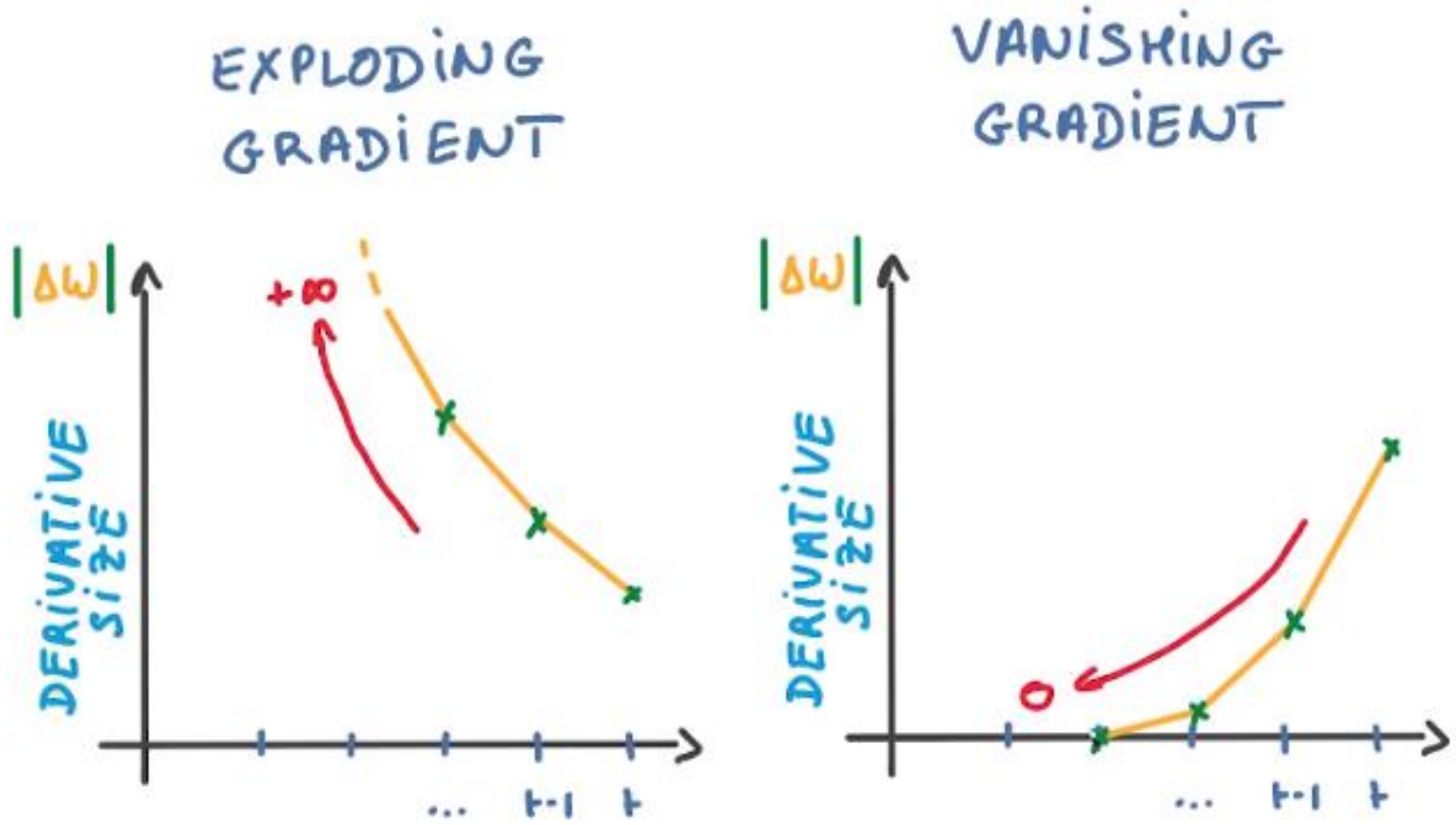
2) Try tuning other hyperparameters, such as **the number of epochs** and **the batch size**.

**3)** **Reusing parts** of a pretrained network (possibly built on an auxiliary task).

**4)** **class_weight**

# Avoiding Overfitting

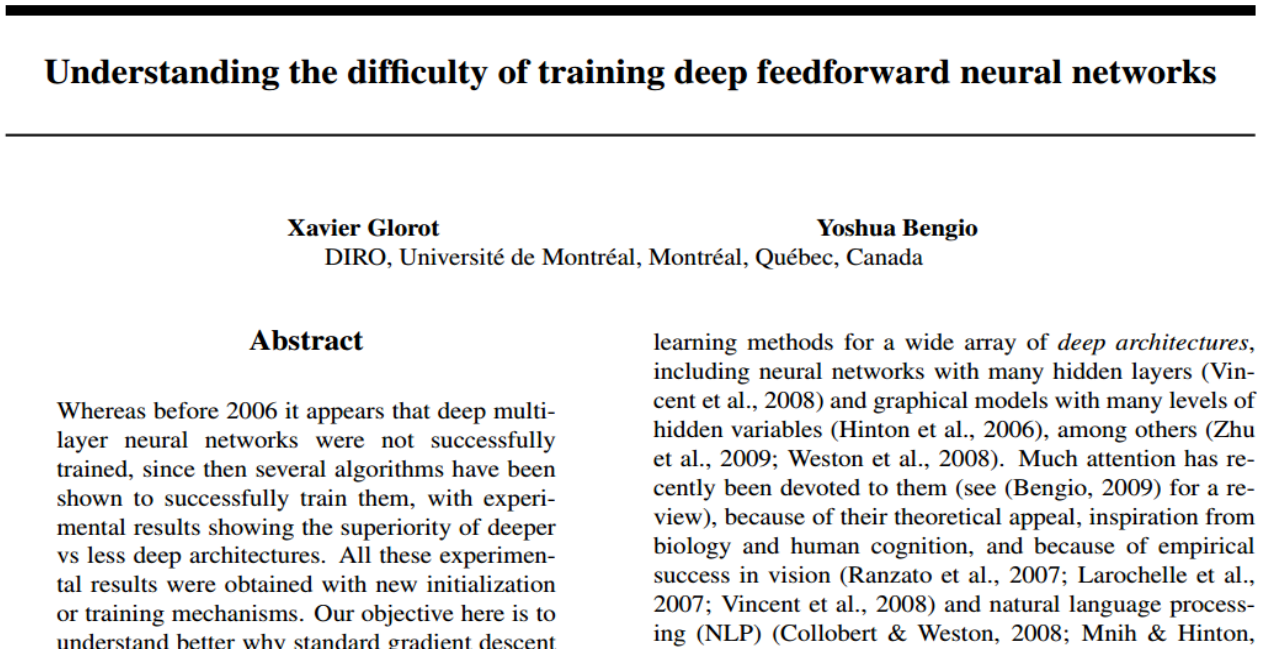
## The Vanishing / Exploding Gradient Problem



# Avoiding Overfitting - initialization

- Applying a good initialization strategy for the connection **weights**
- kernel_initializer="he_uniform" or "he_normal", "glorot_uniform"

**keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")**



**Understanding the difficulty of training deep feedforward neural networks**

**Xavier Glorot** **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

**Abstract**

Whereas before 2006 it appears that deep multi-layer neural networks were not successfully trained, since then several algorithms have been shown to successfully train them, with experimental results showing the superiority of deeper vs less deep architectures. All these experimental results were obtained with new initialization or training mechanisms. Our objective here is to understand better why standard gradient descent

learning methods for a wide array of *deep architectures*, including neural networks with many hidden layers (Vincent et al., 2008) and graphical models with many levels of hidden variables (Hinton et al., 2006), among others (Zhu et al., 2009; Weston et al., 2008). Much attention has recently been devoted to them (see (Bengio, 2009) for a review), because of their theoretical appeal, inspiration from biology and human cognition, and because of empirical success in vision (Ranzato et al., 2007; Larochelle et al., 2007; Vincent et al., 2008) and natural language processing (NLP) (Collobert & Weston, 2008; Mnih & Hinton,

# Avoiding Overfitting - activation function

- The **ReLU** activation function is not perfect. It suffers from a problem known as the **dying ReLUs**: during training, **some neurons** effectively **“die”**, meaning they stop outputting anything other than **0**.
- To solve this problem, you may want to use a variant of the ReLU function, such as the **leaky ReLU** [1], the exponential linear unit (**ELU**)[2], or the Scaled ELU (**SELU**) activation function[3].

```
model = keras.models.Sequential([ [...]
 keras.layers.Dense(10,  kernel_initializer="he_normal"),
 keras.layers.LeakyReLU(alpha=0.2),
 [...]
])
```

```
layer= keras.layers.Dense(10, activation=keras.layers.LeakyReLU(alpha=0.02), ...)
layer = keras.layers.Dense(10, activation="selu", kernel_initializer="lecun_normal")
```

1) *Bing Xu et al., “Empirical Evaluation of Rectified Activations in Convolutional Network,” arXiv preprint arXiv:1505.00853 (2015).*
2) *Djork-Arné Clevert et al., “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” Proceedings of the International Conference on Learning Representations (2016).*
3) *Günter Klambauer et al., “Self-Normalizing Neural Networks,” Proceedings of the 31st International Conference on Neural Information Processing Systems (2017): 972–981.*

# Avoiding Overfitting - activation function

**So, which activation function should you use for the hidden layers of your deep neural networks?**

Although your mileage will vary, in general:

**SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic**

- If the network's architecture prevents it from **selfnormalizing**, then ELU may perform better than SELU (since SELU is not smooth at z = 0).
- If you care a lot about **runtime** latency, then you may prefer leaky ReLU.
- If you **have spare time** and **computing power**, you can use **cross-validation** to evaluate other activation functions.
- If you have a **huge training set** use **ReLU**.
- If **speed** is your priority, **ReLU** might still be the best choice.

# Avoiding Overfitting - Batch Normalization

- **Batch normalization** is a technique designed to **automatically standardize** the inputs to a layer in a deep learning neural network.
- The layer will transform inputs so that they are standardized, meaning that they will have a **mean of zero** and a **standard deviation of one**.
- During training, the layer will keep track of statistics for each input variable and use them to standardize the data.
- Once implemented, batch normalization has the effect of dramatically **accelerating the training process** of a neural network, and in some cases **improves the performance** of the model via a modest regularization effect.

# Avoiding Overfitting - Batch Normalization

❒ **Batch normalization** – It is a step of hyperparameter $\gamma, \beta$ that normalizes the batch $\{x_i\}$. By noting $\mu_B, \sigma_B^2$ the mean and variance of that we want to correct to the batch, it is done as follows:

$$x_i \longleftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="relu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
   keras.layers.Dense(10, activation="softmax")
])
```

# Avoiding Overfitting - Batch Normalization

```
tf.keras.layers.BatchNormalization(
    axis=-1,
    momentum=0.99,
    epsilon=0.001,
    center=True,
    scale=True,
    beta_initializer="zeros",
    gamma_initializer="ones",
    moving_mean_initializer="zeros",
    moving_variance_initializer="ones",
    beta_regularizer=None,
    gamma_regularizer=None,
    beta_constraint=None,
    gamma_constraint=None,
    **kwargs
)
```

https://keras.io/api/layers/normalization_layers/batch_normalization/

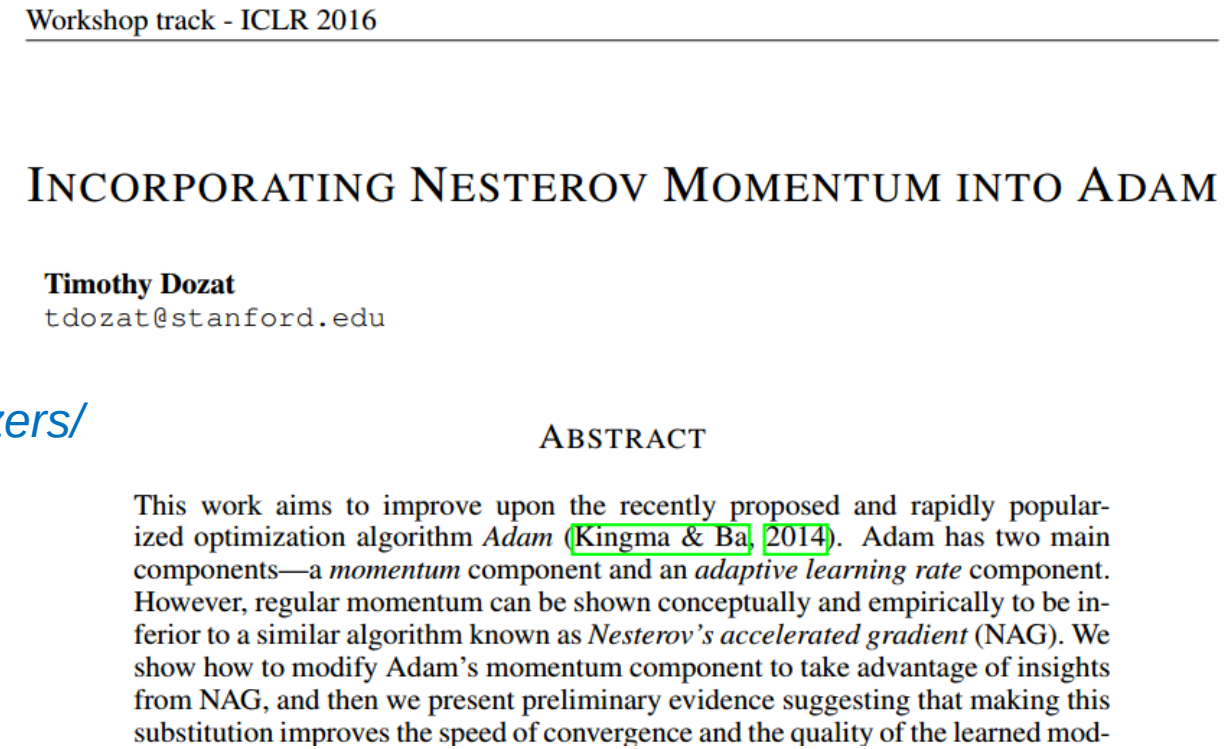
# Avoiding Overfitting - Optimizer

**Using faster optimizer (Gradient Descent, Momentum Optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, Adam, Nadam).**

**Available optimizers:**

1. SGD
2. RMSprop
3. Adam
4. Adadelta
5. Adagrad
6. Adamax
7. Nadam
8. Ftrl

*https://keras.io/api/optimizers/*



Workshop track - ICLR 2016

INCORPORATING NESTEROV MOMENTUM INTO ADAM

**Timothy Dozat**
tdozat@stanford.edu

ABSTRACT

This work aims to improve upon the recently proposed and rapidly popularized optimization algorithm *Adam* (Kingma & Ba, 2014). Adam has two main components—a *momentum* component and an *adaptive learning rate* component. However, regular momentum can be shown conceptually and empirically to be inferior to a similar algorithm known as *Nesterov's accelerated gradient* (NAG). We show how to modify Adam's momentum component to take advantage of insights from NAG, and then we present preliminary evidence suggesting that making this substitution improves the speed of convergence and the quality of the learned mod-

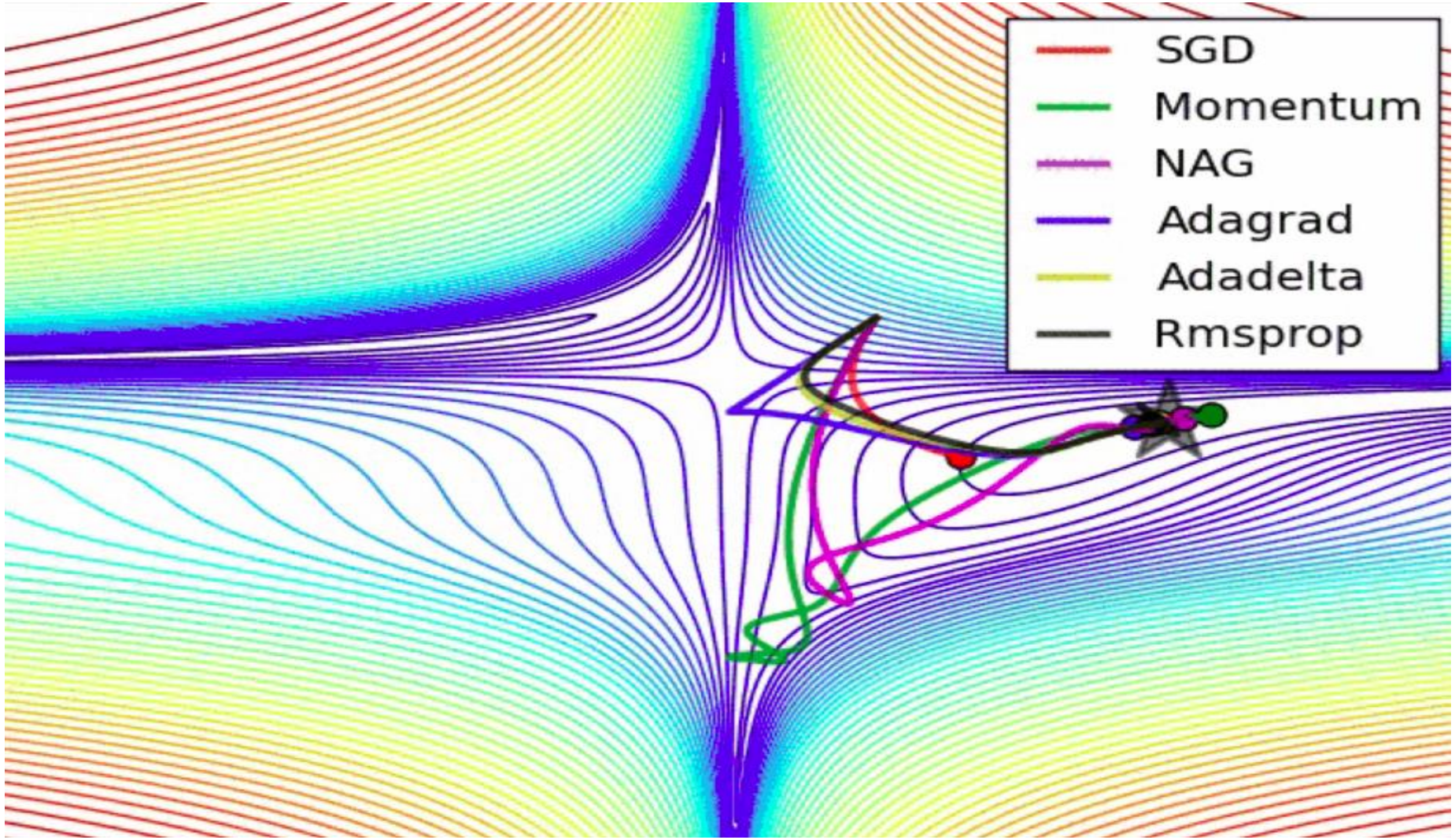
```
opt = keras.optimizers.Adam(lr=0.1, beta_1=0.9, beta_2=0.999)
```

# Avoiding Overfitting - Optimizer

| Class | Convergence speed | Convergence quality |
|---|---|---|
| `SGD` | * | *** |
| `SGD(momentum=...)` | ** | *** |
| `SGD(momentum=..., nesterov=True)` | ** | *** |
| `Adagrad` | *** | * (stops too early) |
| `RMSprop` | *** | ** or *** |
| `Adam` | *** | ** or *** |
| `Nadam` | *** | ** or *** |
| `AdaMax` | *** | ** or *** |

*** is bad, ** is average, and *** is good**

# Avoiding Overfitting - Optimizer



# Avoiding Overfitting - Optimizer

## Usage in a custom training loop

```
# Instantiate an optimizer.
optimizer = tf.keras.optimizers.Adam()

# Iterate over the batches of a dataset.
for x, y in dataset:
    # Open a GradientTape.
    with tf.GradientTape() as tape:
        # Forward pass.
        logits = model(x)
        # Loss value for this batch.
        loss_value = loss_fn(y, logits)

    # Get gradients of loss wrt the weights.
    gradients = tape.gradient(loss_value, model.trainable_weights)

    # Update the weights of the model.
    optimizer.apply_gradients(zip(gradients, model.trainable_weights))
```

*https://keras.io/api/optimizers/*

# Avoiding Overfitting - Optimizer

Another popular technique to mitigate **the exploding gradients problem** is to **clip** the gradients during backpropagation so that they **never exceed some threshold**. This is called **Gradient Clipping**. (avoid **NaN** value in metrics).

In Keras, implementing Gradient Clipping is just a matter of setting the **clipvalue** or **clipnorm** argument when creating an optimizer, like this:

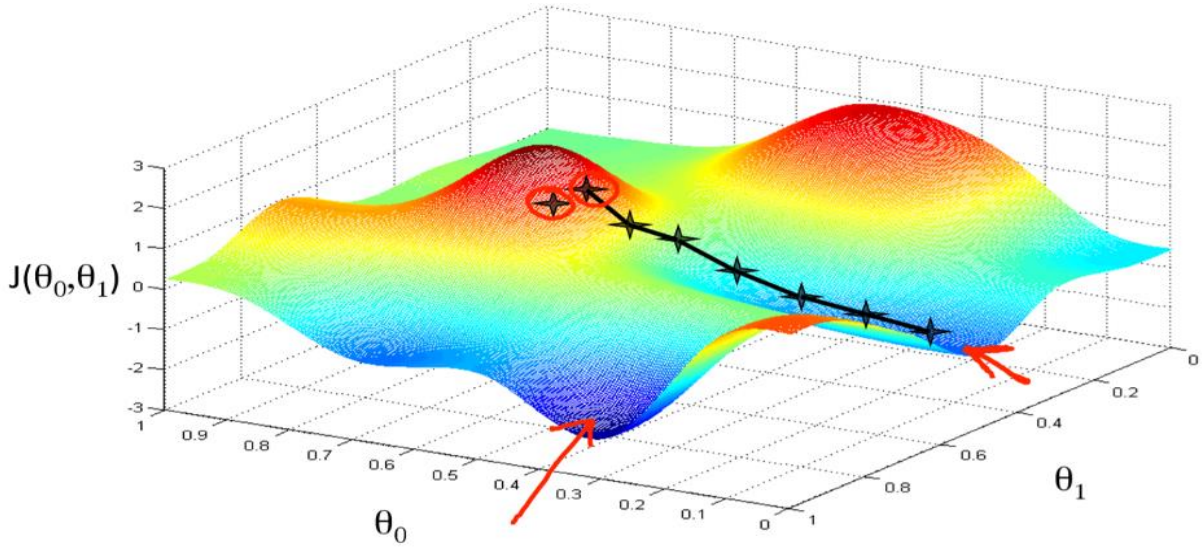
```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

**clipvalue** may **change the orientation** of the gradient vector. If you want to ensure that Gradient Clipping does not change the direction of the gradient vector, you should clip by norm by setting **clipnorm** instead of clipvalue.

# Avoiding Overfitting - learning_rate

**Update the optimizer's learning_rate attribute at the beginning of each epoch**:

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.2, patience=5,
                                                 min_lr=0.0001)
history = model.fit(X_train, y_train, [...], callbacks=[lr_scheduler])
```



# Avoiding Overfitting - learning_rate

**Learning rate scheduler** : At the beginning of every epoch, this callback gets the updated learning rate value from **schedule function** provided at **__init__**, with the current epoch and current learning rate, and applies the updated learning rate on the optimizer.

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

lr_scheduler = tf.keras.callbacks.LearningRateScheduler(scheduler)
history = model.fit(X_train, y_train, [...], callbacks=[lr_scheduler])
```

# Avoiding Overfitting - $\ell 1$ and $\ell 2$ Regularization

**Regularisation** is a process of introducing **additional information** in order to prevent **overfitting**.

## L1 Regularization:

$$loss = error(y, \hat{y}) + \gamma \sum_j |w_j| \qquad (\gamma \geq 0)$$

## L2 Regularization:

$$loss = error(y, \hat{y}) + \gamma \sum_j w_j^2 \qquad (\gamma \geq 0)$$

# Avoiding Overfitting - ℓ1 and ℓ2 Regularization

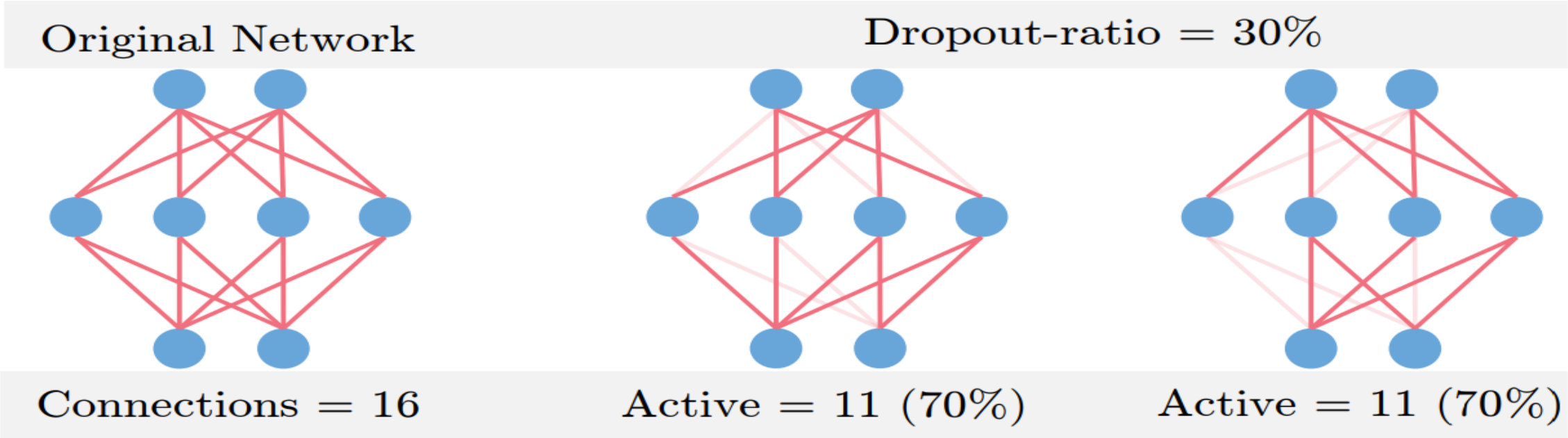
```
layer_h_n = keras.layers.Dense(100, activation="relu",kernel_initializer="he_normal",
                               kernel_regularizer=keras.regularizers.l2(0.01))
layer_output = keras.layers.Dense(10, activation="softmax",kernel_initializer="glorot_uniform")
model.add(layer_h_n)
model.add(layer_output)
```

- The **l2()** function returns a regularizer that will be called at each step during training to compute the regularization loss. This is then added to the final loss.
- **l1 : keras.regularizers.l1(0.01)**
- **l1 + l2 : keras.regularizers.l1_l2()**

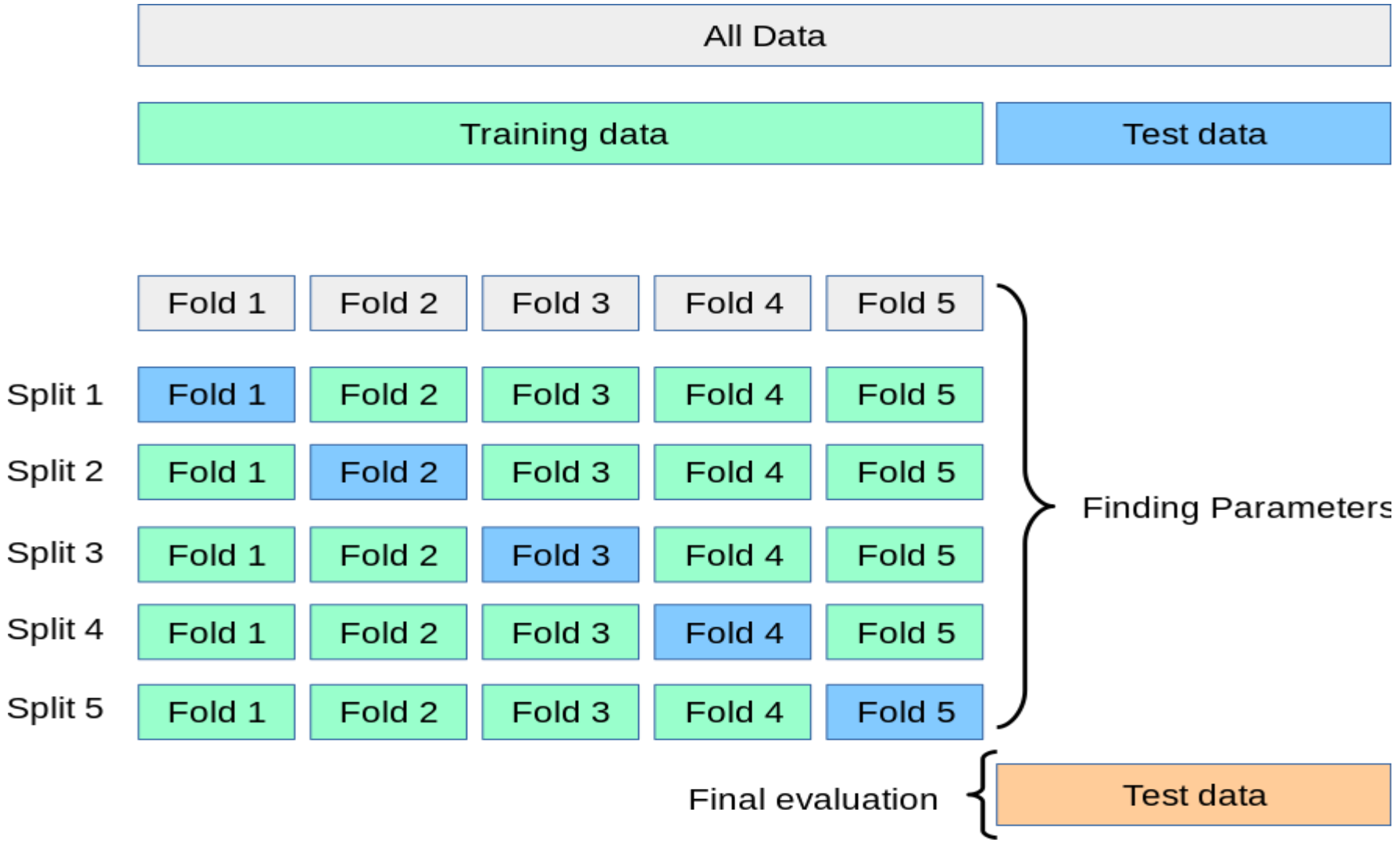
# Avoiding Overfitting - Dropout

**Dropout:** Connections between consecutive layers are **randomly dropped** based on a **dropout-ratio** and the remaining network is trained in the current iteration. In the next iteration, another set of random connections are dropped.

**model.add(keras.layers.Dropout(0.3))**



# Avoiding Overfitting - KFold CV



# Avoiding Overfitting - KFold CV

```
def create_model():
  model = tf.keras.models.Sequential()
  .
  model.compile(loss = "", optimizer = "", metrics = [""] )
  return model

from sklearn.model_selection import KFold
n_split=5
cvscores = []
for train_index,test_index in KFold(n_split).split(X):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = Y[train_index], Y[test_index]

  model = create_model()
  model.fit(x_train, y_train,epochs=5)
  print("Model evaluation : ", model.evaluate(X_test,y_test))
  scores = model.evaluate(X_test,y_test)
  print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
  cvscores.append(scores[1] * 100)

print("%.2f%% (+/- %.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
```

When this process is completed, you will end up with K accuracy values, one for each model. Then you can compute the mean and standard deviation of all the accuracy scores and use it to get an idea of how accurate you can expect the model to be.

# How to increase your small image dataset

**Data augmentation:** Create more data from available data by randomly cropping, dilating, rotating, adding small amount of noise etc.

```
trainAug = ImageDataGenerator( rotation_range=40, width_shift_range=0.2,  height_shift_range = 0.2,
                               shear_range=0.2, zoom_range=0.2, horizontal_flip=True,fill_mode='nearest')

model.compile(loss="binary_crossentropy", optimizer=opt, metrics=["accuracy"])
H = model.fit( trainAug.flow(trainX, trainY, batch_size=BS), steps_per_epoch=len(trainX) //
                BS,validation_data=(ValX, ValY), validation_steps=len(ValX) // BS, epochs=EPOCHS)
```

# Avoiding Overfitting - Injecting Noise

- **Adding noise** to an underconstrained neural network model with a **small training dataset** can have a regularizing effect and **reduce overfitting**.

- **Keras** supports the addition of **Gaussian noise** via a separate layer called the GaussianNoise layer. This layer can be used to add noise to an existing model **as an input layer or between hidden layers**.

```
# define model: 1
model = Sequential()
model.add(GaussianNoise(0.01, input_shape=(2,)))
model.add(Dense(500, activation="relu"))

# define model: 2
model = Sequential()
model.add(Dense(500, input_shape=2))
model.add(GaussianNoise(0.1))
```

```
# CNN
from keras.layers import GaussianNoise

model.add(Conv2D(32, (3,3)))
model.add(MaxPooling2D())
model.add(GaussianNoise(0.1))
```

# Avoiding Overfitting - Using Callbacks

```
checkpoint_cb = keras.callbacks.ModelCheckpoint ("my_keras_model.h5",
                                                 save_best_only=True)
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid),
                                                 callbacks=[checkpoint_cb])
# roll back to best model
model = keras.models.load_model("my_keras_model.h5")
```

- You can combine both callbacks to save checkpoints of your model (in case your computer crashes) and interrupt training early when there is no more progress (to avoid wasting time and resources):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_valid, y_valid),
                                                 callbacks=[checkpoint_cb, early_stopping_cb])
```

***https://keras.io/api/callbacks/***

# Dense layer

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

**https://keras.io/api/layers/core_layers/dense/**

# Conv2D layer

```
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

**https://keras.io/api/layers/convolution_layers/convolution2d/**

# How to Save and Load Your Model

Keras use the **HDF5 format** to save both the **model's architecture (including every layer's hyperparameters)** and the values of all **the model parameters** for every layer **(e.g., connection weights and biases)**. It also saves **the optimizer** (including its hyperparameters and any state it may have).

```
model.save("my_keras_model.h5")
```

## Loading the model:

```
model = keras.models.load_model("my_keras_model.h5")
```

## In case we use custom loss/metric function:

```
model = keras.models.load_model("my_keras_model.h5",
                                compile=False)
```

# Avoiding Overfitting - CNN

```
# Creating the model using the Sequential API
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(filters=64, kernel_size=5, strides=1, padding="same",
                              activation="relu", input_shape= (128,128,3)))
model.add(keras.layers.MaxPool2D(pool_size=2))
model.add(keras.layers.Dropout(0.4))
model.add(keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, padding="same", activation="relu"))
model.add(keras.layers.MaxPool2D(pool_size=2))
model.add(keras.layers.Dropout(0.4))
model.add(keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, padding="same", activation="relu"))
model.add(keras.layers.MaxPool2D(pool_size=2))
model.add(keras.layers.Dropout(0.4))

model.add(keras.layers.Flatten())
layer0 = keras.layers.Dense(512, activation="relu",kernel_initializer="he_normal",
                            kernel_regularizer=keras.regularizers.l2(0.01))
layer1 = keras.layers.Dense(128, activation="relu",kernel_initializer="he_normal",
                             kernel_regularizer=keras.regularizers.l2(0.01))
layer_output = keras.layers.Dense(1, activation="sigmoid",kernel_initializer="glorot_uniform")

model.add(layer0)
model.add(keras.layers.Dropout(0.2))
model.add(layer1)
model.add(keras.layers.Dropout(0.2))
model.add(layer_output)
```

# Thank you for your attention

**Hichem Felouat ...**