Training neural networks

Training: supervised learning

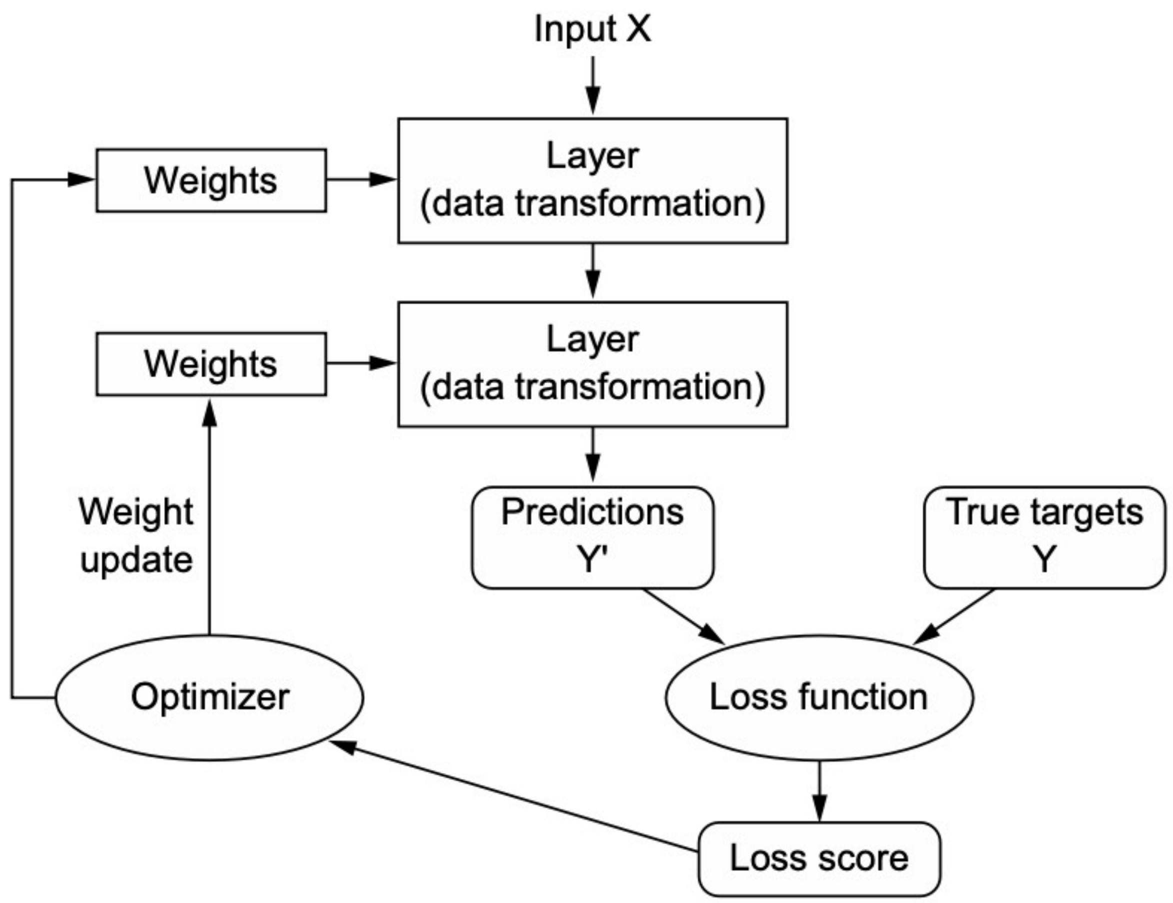
Data: Training examples consisting of **inputs** and their desired **outputs**

Objective: To set the values of the **connection weights** that minimize a cost function

Multiple **gradient-descent** algorithms exist:

- The most used historically is ***Backpropagation*** and derivatives
- Other algorithms: Marquardt-Levenberg, Rprop, Quickprop
- Most recent: RMSprop, Adam, SGD, Adagrad

Training: supervised learning



Overall scheme of the training/ learning process

Backpropagation steps

Forward propagation – calculates the output value(s) for the respective input vector and the error (given the known outputs);

Backpropagation – given the error on that sample, it is propagated back, adjusting the weights of the connections to reduce this error. This process is based on the calculation of the **gradient** of the cost function, using the **chain rule** of partial derivatives for composite functions

This process is repeated multiple times in order to improve the network's performance.

***Back-propagation* algorithm**

It is based on the gradient of the error surface that defines the direction of maximum descent (**gradient descent**)

Important parameter: Learning rate- sets the size of the steps in that direction

A sequence of these movements leads to a **minimum** (desirably global)

The training takes place over a number of **epochs** (iterations)

Initial network configuration (connection weights) - randomly generated

Stopping criterion: fixed number of epochs, elapsed time or convergence criterion based on a subset of validation examples

Training: chain rule and computation graphs

As we have already seen, a NN implements the **composition of functions (layers):**

- from the calculation of the activation from the weights and inputs of each neuron
- through the application of the activation functions
- By the interconnection of neurons in successive layers, according to the topology of the network

Thus, the cost function of a NN includes many parameters (weights) defined at various levels in the network

Training: chain rule and computation graphs

To facilitate the calculation of the derivatives of this cost function, in relation to the weights to be optimized, the chain rule is used to calculate derivatives of composite functions

Computationally, the process of calculating the output of the NN and the cost function is organized in a **computation graph**, used for the calculation of derivatives by reversing the direction of calculation and allowing to work layer by layer

Chain Rule

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x)$$

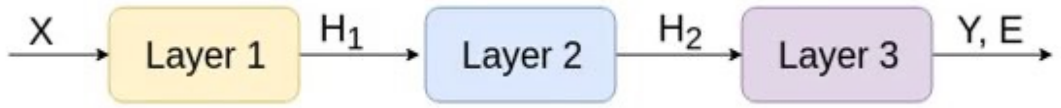
$$\begin{tikzcd}
{} \arrow[r, "x"] & \boxed{g} \arrow[r, "v"] & \boxed{f} \arrow[r, "y"] & {} \\
{} & \boxed{g'} \arrow[l] & \boxed{f'} \arrow[l] & {} \arrow[l]
\end{tikzcd}$$

$$\frac{dy}{dx} = \frac{dy}{dv} \cdot \frac{dv}{dx}$$

Note that:

$$v = g(x)$$
$$y = f(g(x))$$

Training: example with feedforward NN



$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Dense layer

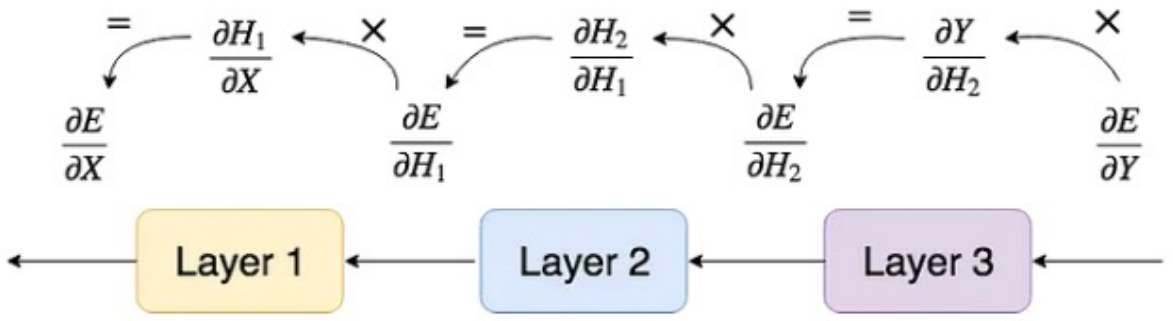
$$y_j = b_j + \sum_i x_i w_{ij}$$

Forward propagation

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} \\ \vdots \quad \ddots \quad \vdots \\ w_{i1} \dots w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

$$Y = XW + B$$

Training: example with feedforward NN


$\frac{\partial E}{\partial X}$
$\frac{\partial H_1}{\partial X}$
$\frac{\partial E}{\partial H_1}$
$\frac{\partial H_2}{\partial H_1}$
$\frac{\partial E}{\partial H_2}$
$\frac{\partial Y}{\partial H_2}$
$\frac{\partial E}{\partial Y}$
=
×
Layer 1
Layer 2
Layer 3

Backward propagation

STEP 1: compute the layer input error for layer i (the output error from the layer i+1), dE/dX, to pass on to the layer i-1 (this will be the dE/dY of the next layer)

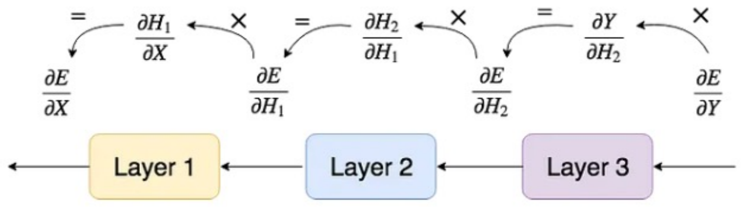
$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \dots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$

Chain rule

$$\begin{aligned} \frac{\partial E}{\partial x_i} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \dots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i} \\ &= \frac{\partial E}{\partial y_1} w_{i1} + \dots + \frac{\partial E}{\partial y_j} w_{ij} \end{aligned}$$

$$\begin{aligned} \frac{\partial E}{\partial X} &= \begin{bmatrix} (\frac{\partial E}{\partial y_1} w_{11} + \dots + \frac{\partial E}{\partial y_j} w_{1j}) & \dots & (\frac{\partial E}{\partial y_1} w_{i1} + \dots + \frac{\partial E}{\partial y_j} w_{ij}) \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \dots & \frac{\partial E}{\partial y_j} \end{bmatrix} \begin{bmatrix} w_{11} & \dots & w_{i1} \\ \vdots & \ddots & \vdots \\ w_{1j} & \dots & w_{ij} \end{bmatrix} \\ &\boxed{= \frac{\partial E}{\partial Y} W^t} \end{aligned}$$

Training: example with feedforward NN



$\frac{\partial H_1}{\partial X}$
$\frac{\partial H_2}{\partial H_1}$
$\frac{\partial Y}{\partial H_2}$
$\frac{\partial E}{\partial X}$
$\frac{\partial E}{\partial H_1}$
$\frac{\partial E}{\partial H_2}$
$\frac{\partial E}{\partial Y}$
=
×
Layer 1
Layer 2
Layer 3

Backward propagation

STEP 2: compute the weights' errors: dE/dW = X.T * dE/dY

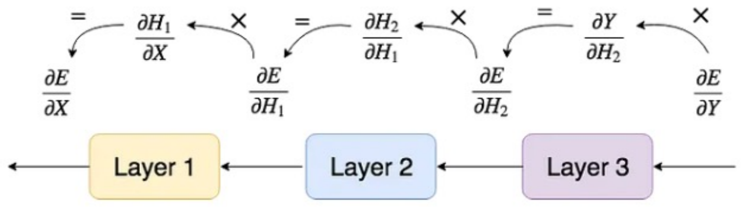
$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \cdots & \frac{\partial E}{\partial w_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{i1}} & \cdots & \frac{\partial E}{\partial w_{ij}} \end{bmatrix}$$

Chain rule ➡

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{ij}} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} \\ &= \frac{\partial E}{\partial y_j} x_i \end{aligned}$$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1}x_1 & \cdots & \frac{\partial E}{\partial y_j}x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1}x_i & \cdots & \frac{\partial E}{\partial y_j}x_i \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} = \boxed{X^t \frac{\partial E}{\partial Y}}$$

Training: example with feedforward NN



$\frac{\partial H_1}{\partial X}$
$\frac{\partial H_2}{\partial H_1}$
$\frac{\partial Y}{\partial H_2}$
$\frac{\partial E}{\partial X}$
$\frac{\partial E}{\partial H_1}$
$\frac{\partial E}{\partial H_2}$
$\frac{\partial E}{\partial Y}$
Layer 1
Layer 2
Layer 3

Backward propagation

STEP 3: compute the biases' errors: dE/dB = dE/dY

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial b_1} & \frac{\partial E}{\partial b_2} & \dots & \frac{\partial E}{\partial b_j} \end{bmatrix}$$

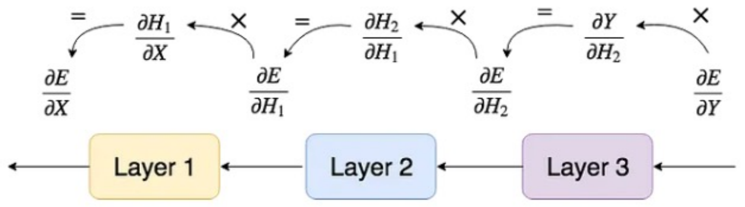
Chain rule ➡

$$\begin{aligned} \frac{\partial E}{\partial b_j} &= \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_j} + \dots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j} \\ &= \frac{\partial E}{\partial y_j} \end{aligned}$$

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \dots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$\boxed{= \frac{\partial E}{\partial Y}}$$

Training: example with feedforward NN



$\frac{\partial E}{\partial X}$
$\frac{\partial H_1}{\partial X}$
$\frac{\partial E}{\partial H_1}$
$\frac{\partial H_2}{\partial H_1}$
$\frac{\partial E}{\partial H_2}$
$\frac{\partial Y}{\partial H_2}$
$\frac{\partial E}{\partial Y}$
=
×
Layer 1
Layer 2
Layer 3

Backward propagation

STEP 4: update weights and biases using gradient descent

$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i}$$

$$b_i \leftarrow b_i - \alpha \frac{\partial E}{\partial b_i}$$

Training: activation functions

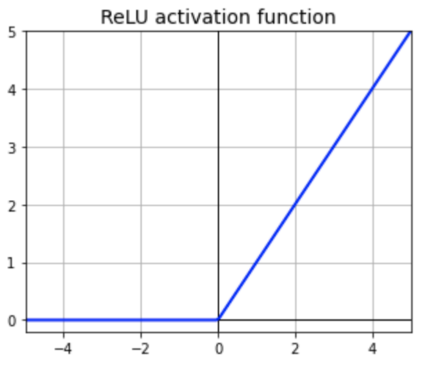
Activation functions will work similarly to other layers, being defined their forward and backpropagation computation graphs

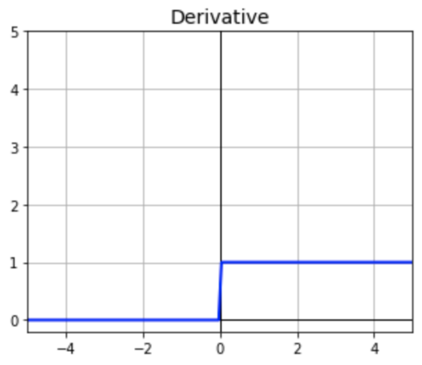
Since they do not have weights, only the error propagation step (step 1) is done for these cases

RelU

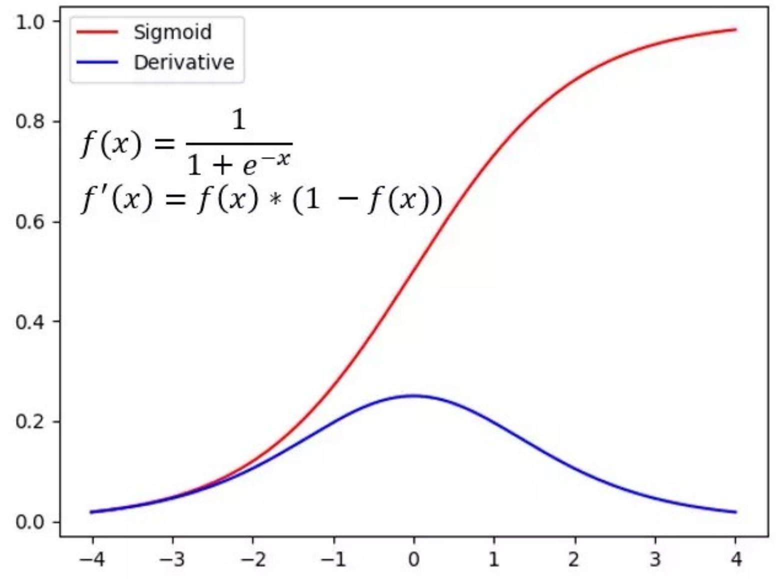
$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



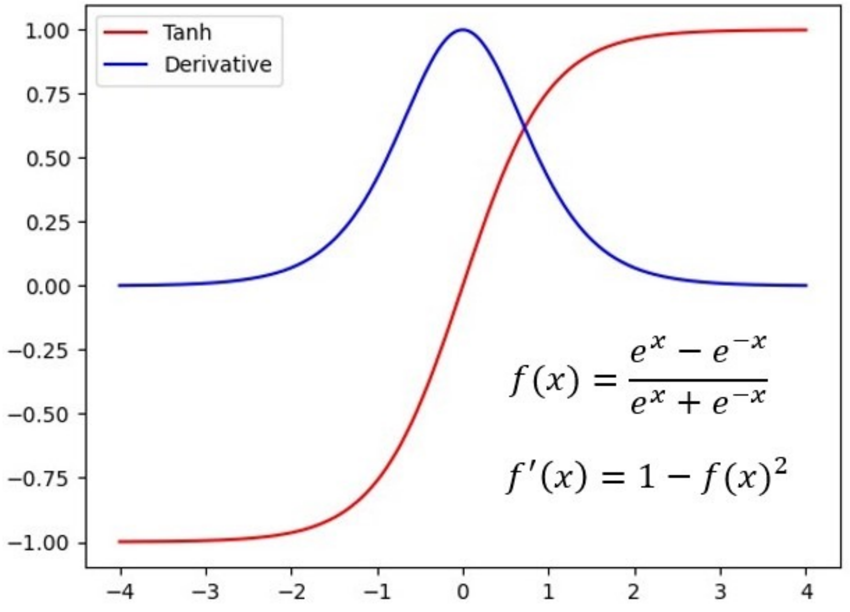


Sigmoid



Training: activation functions

Tanh



Softmax

$$f(x_i) = \frac{e^{x_i}}{\sum_j^n e^{x_j}} = \frac{e^x}{\sum e^x}$$

$$f'(x) = f(x)(1 - f(x))$$

Training: cost function

The cost function is the one that will be minimized in backpropagation

Used functions in NN are similar to those used in linear and logistic regression, being the sum of the error function for the set of individual examples

The error function (***loss function***) for each example will typically be given by:

- Sum/ mean of the squared errors (SSE/ MSE), for regression problems
- (Binary) cross entropy (used in logistic regression) for binary classification
- A generalization of the previous one to handle multiple classes

Training: loss functions in backprop

Mean squared error

$$L = \frac{1}{n}\sum_{i}^{n}\left(y_i - y_i^*\right)^2$$

$$\frac{\partial L}{\partial Y} = \left[\frac{\partial L}{\partial y_1} \quad \ldots \quad \frac{\partial L}{\partial y_i}\right]$$

$$= \frac{2}{n}[y_1^* - y_1 \quad \ldots \quad y_i^* - y_i]$$

$$= \frac{2}{n}(Y^* - Y)$$

Training: loss functions in backprop

Binary cross entropy

$$L = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log(y_i) + (1 - y_i)\log(1 - y_i))$$

$$\frac{\partial L}{\partial Y} = \frac{-Y}{Y*} - \left(\frac{1-Y}{1-Y*} * -1\right)$$

$$= \frac{-Y}{Y*} + \frac{1-Y}{1-Y*}$$

Categorical cross entropy

$$L = -\sum_{i=1}^{i=N} y_i \log(y_i^*)$$

$$\frac{\partial L}{\partial Y} = -\frac{Y}{Y*}$$

Improvements on training algorithms: batches

Doing the training process using **mini-batches** (batches with part of the examples) instead of considering all available examples to calculate errors and their derivatives and update weights

Batch sizes to consider change typically between 64 and 512; *batch* should be of a size to allow keeping it as a whole in GPU memory

Batch size can be tuned/ optimized

If batch = 1 -> *stochastic gradient descent*

If batch = number of examples in the dataset -> *batch gradient descent*

Improvements on training algorithms: momentum

Recent algorithms use **momentum** – calculating moving average of the error gradients for the last N updates

Way to remove update noise, allowing to use higher learning rates and accelerate the convergence of the training process

Adds a new parameter (β) defining the weight to the previous value (new batch contributes with 1 – β) – typical value of 0.9

Retained gradient $\longrightarrow V_t = \beta V_{t-1} + (1-\beta)\,\dfrac{\partial L}{\partial W}$

$$W_t = W_{t-1} - \alpha V_t$$

Improvements on training algorithms: RMSProp e Adam

RMSProp also uses moving averages, but here of the squared errors (uses also a parameter β – typical value of 0.999)

Weight update divides the derivative by the square root of the moving
ε
average (adding a small value ε to avoid zero values)

Adam combines momentum with the strategy of RMSProp, so having two parameters for the update - β1 e β2 - and also ε

Python implementation - numpy

We will create a module that will include a few files with different components:

- *"neuralnet"* – it will include the main class to create a neural network that will "comunicate" with the other modules;
- "layers" – it will include the layers that will make the NNs (dense, etc);
- "activations" – it will include the activation layers (sigmoid, tahn, relu, etc);
- "optimizer" – it will include the training algorithms used to update the parameters of the NNs during training (SGD);
- "losses" – it will include the loss functions to optimize during training (CrossEntropy, MeanSquaredError, etc)
- “data” – dataset (similar to previous ones)
- “metrics” - error metrics

These components will include some core implementations, but also make up a base for further developments

Python implementation - Layers

- We will first implement a base layer with some shared and **abstract** methods for all other specific layers.
- class Layer:
    - methods:
        - forward_propagation – abstract method that computes the output of a layer given the input;
        - backward_propagation – abstract method that back propagates the output error;
        - output_shape – abstract method that returns the output shape of the layer;
        - parameters – abstract mehtod that returns the number of parameters of the layer;
        - set_input_shape – sets the input shape of the layer;
        - input_shape – returns the input_shape of the layer;
        - layer_name – returns the name of the layer;

PYTHON IMPLEMENTATION

Python implementation – Dense Layer

- A dense layer, also known as a fully connected layer, connects each neuron to every neuron in the previous and next layers, forming a dense network of connections.
- class DenseLayer(Layer):
    - arguments:
        - n_units – number of neurons in the NN;
        - input_shape – tuple with the input_shape (for tabular data it will be (n_features,))
    - estimated parameters:
        - input – the layer input;
        - output – the layer output;
        - weights – the layer weights;
        - biases – the layer biases;
    - methods:
        - initialize – initializes the layer with random weiights and biases and with the optimizer;
        - parameters – returns the number of parameters of the layer;
        - **forward_propagation - perform forward propagation on a given input;**
        - **backward_propagation - perform backward propagation on the given output error;**
        - output_shape - returns the shape of the output of the layer.

Complete these functions !

PYTHON IMPLEMENTATION

Python implementation – Activation layers

- class ActivationLayer(Layer):
  - methods:
    - forward_propagation - perform forward propagation on a given input by applying the activation function;
    - backward_propagation - perform backward propagation on the given output error by applying the derivative of the activation layer on the input and multiplying it with the output error;
    - activation_function – abstract method that computes the activation function on the given input (used on the forward propagation);
    - derivative – abstract method that computes the derivative of the activation function on the given input (used on the back propagation);
    - output_shape - returns the shape of the output of the layer (which is the same as the input_shape);
    - parameters – returns the number of parameters of the layer (returns 0 as activation layers do not have learnable parameters).

PYTHON IMPLEMENTATION

Python implementation – Activation layers

- class ReLUActivation(ActivationLayer):
  - methods:
    - activation_function – applies the ReLU function on the input (argument)
    - **derivative – applies the ReLU derivative on the input (argument)**

Complete these functions !

- class SigmoidActivation(ActivationLayer);
  - methods:
    - activation_function – applies the sigmoid function on the input (argument)
    - **derivative – applies the sigmoid derivative on the input (argument)**

Complete these functions !

PYTHON IMPLEMENTATION

Python implementation – Loss functions

- We will first implement a base loss function class with two abstract methods for all other specific layers (the class will not need an __init__).
- class LossFunction:
  - methods:
    - loss – abstract method that computes the loss based on the true and predicted labels
    - derivative - abstract method that computes the derivative of the loss based on the true and predicted labels
- class MeanSquaredError(LossFunction):
    - loss - compute the mean squared error loss function;
    - **derivative - compute the derivative of the mean squared error loss function**

Complete these functions !

- class BinaryCrossEntropy(LossFunction):
    - loss - compute the binary cross-entropy loss function;
    - **derivative - compute the derivative of the binary cross-entropy loss.**

Complete these functions !

PYTHON IMPLEMENTATION

Python implementation – optimizer

- Optimizers are algorithms that adjust the model's parameters during training to minimize the loss function.
- We will implement the SGD optimization algorithm, with the option of considering the momentum

  - arguments:
    - learning_rate – the learning rate to use for updating the weights;
    - momentum – the momentum to use for updating the weights.
  - estimated_patameters:
    - retained_gradient – accumulated gradient from previous epochs;
  - methods:
    - update - Update the retained_gradient, computes and returns the updated weights;

Python implementation – neural network

- Now that we have all the components we can build a NN. The NeuralNetwork class will connect all the other classes together.
- class NeuralNetwork:
    - arguments:
        - epochs - the number of epochs to train the neural network;
        - batch_size - the batch size to use for training the neural network;
        - optimizer - the optimizer to use for training the neural network;
        - learning_rate - the learning rate to use for training the neural network;
        - momentum – parameter of momentum;
        - verbose - whether to print the loss and the metric at each epoch;
        - loss - the loss function to use for training the neural network;
        - metric - the metric to use for training the neural network;

PYTHON IMPLEMENTATION

Python implementation – neural network

- class NeuralNetwork:
    - estimated parameters:
        - layers – the layers of the NN.
        - history – dictionary containning the loss and metric at each epoch.
    - methods:
        - add – add a layer to the NN.
        - get_mini_batches – handle training batches yielding sets of exemples for training
        - **forward_propagation – forward propagation of the full feedforward NN, going through all layers to compute final output for a set of examples**
        - **backward_propagation – backward propagation of the full feedforward NN for a set of examples, going through all layers in reverse order to calculate errors and update weights**
        - fit – train the NN (will call both forward and backward propagation).
        - predict – make predictions on new data (calls forward propagation);
        - score – evaluate the NN on a given dataset (calls predict);

Complete these functions !

Python implementation – neural network

- NeuralNetwork.fit:
  - arguments:
    - dataset – the dataset to train the NN on
  - algorithm:
    1. Compute mini batches of size self.batch_size;
    2. Perform forward propagation for all layers;
    3. Compute the loss derivative error;
    4. Backpropagate the error through all layers using backward propagation;
    5. Repeat steps 2, 3 and 4 for all mini batches;
    6. Compute the loss based on true labels and predictions;
    7. Compute the metric based on true labels and predictions;
    8. Save the loss and metric in the history dictionary;
    9. Print the metric and loss if verbose is set to True;
    10. Repeat the previous steps for all epochs.

Training algorithms in Keras

Jupyter Notebook:
ex_mnist_train_algs.ipynb

Several algorithms have been proposed and are available in tensorflow; the notebook shows its use in the MNIST case including the algorithms: Adam, RMSprop, SGD

Training algorithms include a set of parameters (dependent on the algorithm) which may be tuned/ optimized where the most relevant is the **learning rate (α)**

Overfitting and regularization

Methods to address overfitting in neural networks

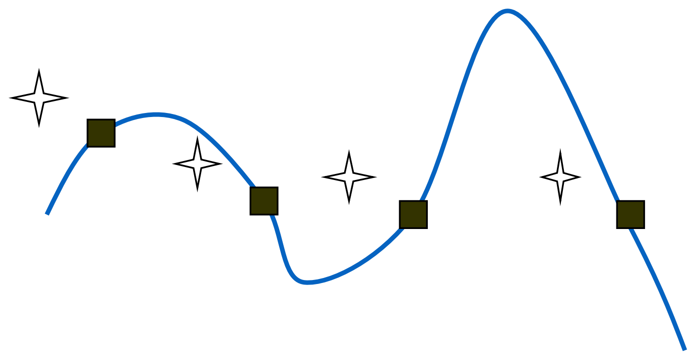
Overfitting and generalization in NNs

To train the network many epochs (training iterations) can prevent generalization and promote *overfitting*

It is preferable to stop the training in these cases – **early stopping**; error is measured in a validation dataset; if the validation error improves for a number of consecutive epochs, the training is stopped

The probability of overfitting increases if:

- Few training cases
- Many weights (complexity)



Overfitting and generalization

In DL, overfitting has been addressed using various techniques, such as **regularization** (L1 or L2, similar to linear/logistic regression, called **decay** in NNs), or explicit control of model **capacity** (e.g. number of hidden layers and number of neurons in each)

A specific technique developed for DL models is **dropout**: in each iteration of the training the output of some randomly selected neurons is put to zero. The proportion of zeroed neurons in each iteration is a parameter defined by each layer where dropout applies

IMDB dataset

Jupyter Notebook:
ex_dnn_imdb.ipynb

IMDB dataset	Dataset with texts of reviews in IMDB about movies; classified into 2 classes: positive or negative; 25k reviews for training + 25k for test; balanced – 50% positive and negative examples We will only consider the 10k words that are most common

This dataset will be used here, considering a one-hot encoding scheme where the 10K most common words will be considered as binary features.

So, if a word appears in a review the value of the feature will be 1, if not it will be 0

Notice that in this way, the order of the words is neglected

IMDB – loading and exploring data

```
(train_data, train_labels), (test_data, test_labels) =
            imdb.load_data(num_words=10000)

print(train_data.shape, test_data.shape)
print(len(train_labels), len(test_labels))
```

Load data

Check dimensions

Accessing the original text

```
print(train_data[0], train_labels[0])
print(max([max(sequence) for sequence in train_data]))

word_index = imdb.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[2]])
print(decoded_review)
```

IMDB dataset: pre-processing

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)

y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

One-hot encoding of inputs: Each review represented by the words it contains ignoring the order

Output to binary (0 / 1)

DNNs for the IMDB

```
from tensorflow.keras import models, layers, Input

hidden = 16
model = models.Sequential()
model.add(Input((10000,)))
model.add(layers.Dense(hidden, activation='relu'))
model.add(layers.Dense(hidden, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

DNN – 2 hidden layers
16 neurons in each; RelU
Output layer – sigmoid – binary classification

DNNs for the IMDB

```
model.compile(optimizer='rmsprop',
            loss='binary_crossentropy',
            metrics=['accuracy'])
```

Loss function – binary cross entropy (used in logistic regression)

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Defines validation set

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Training the network

DNN for IMDB

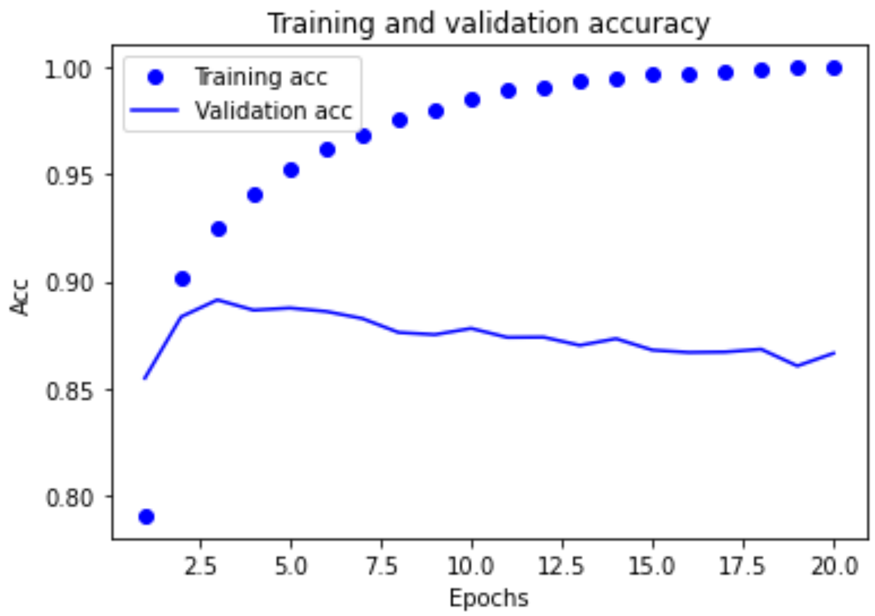
```
history_dict = history.history
acc_values = history_dict[‘acc']
val_acc_values = history_dict['val_acc']
epochs = range(1, len(acc_values) + 1)
```

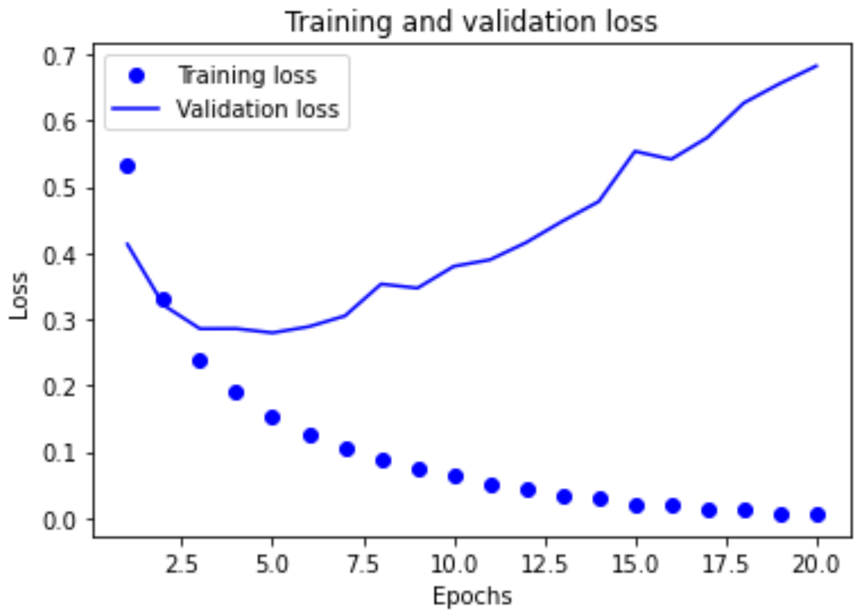
Plot of errors (accuracy)
Can also do that for the loss function

```
import matplotlib.pyplot as plt
plt.plot(epochs, acc_values, 'bo', label='Training acc')
plt.plot(epochs, val_acc_values, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel(‘Acc')
plt.legend()
plt.show()
```

DNN for IMDB

Plot of accuracy and loss in training and validation sets





What do these graphs suggest ?

DNN for IMDB

```
preds = model.predict(x_test) # probabilities
class_preds = preds > 0.5

print(preds[0:5])
print(class_preds[0:5])
```

Predict for new examples in test set

Evaluate in the test set

```
results = model.evaluate(x_test, y_test, verbose= 0)
print(results)
```

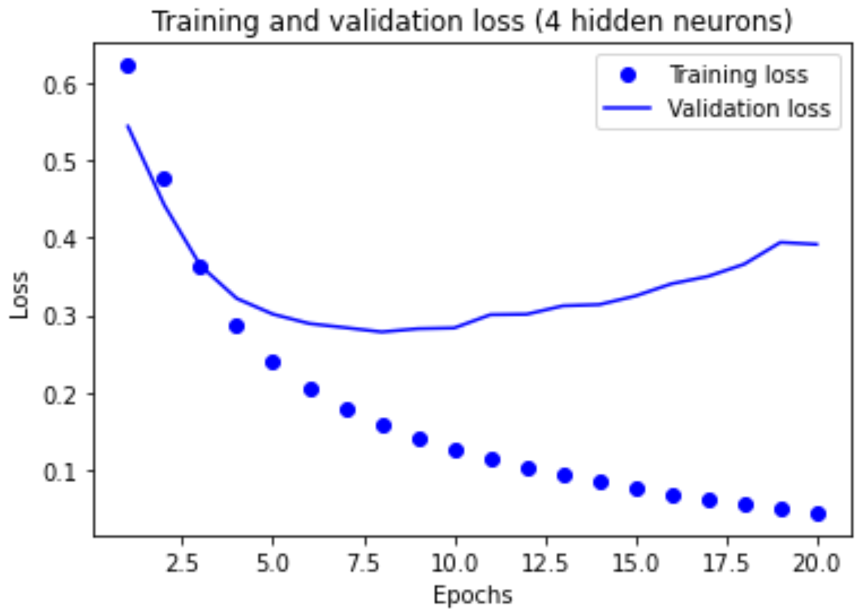
Around 85% of accuracy in test set !

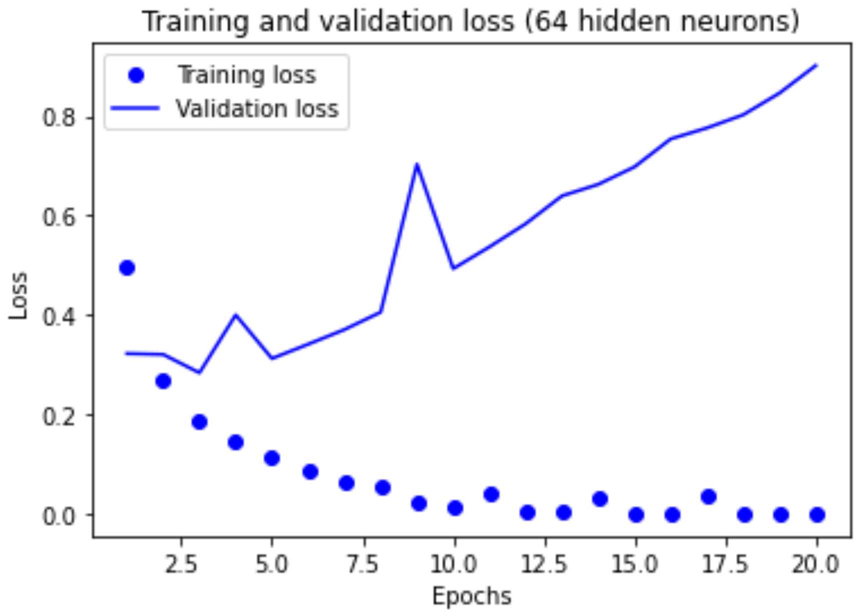
Overfitting: IMDB

Controlling the model’s capacity

```
hidden = 4   # test with 4, 8, 16, 64,...

model = models.Sequential()
...
```



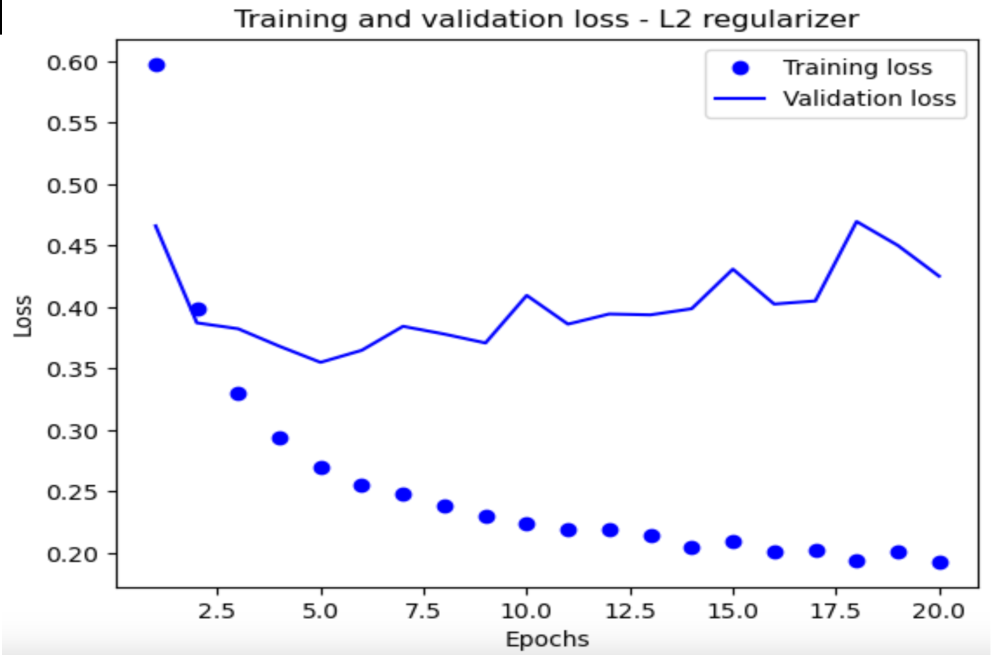


Overfitting: IMDB

Regularization

```
from tensorflow.keras import regularizers
model.add(Input((10000,))))
model.add(layers.Dense(hidden, activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dense(hidden, activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dense(1, activation='sigmoid'))
```

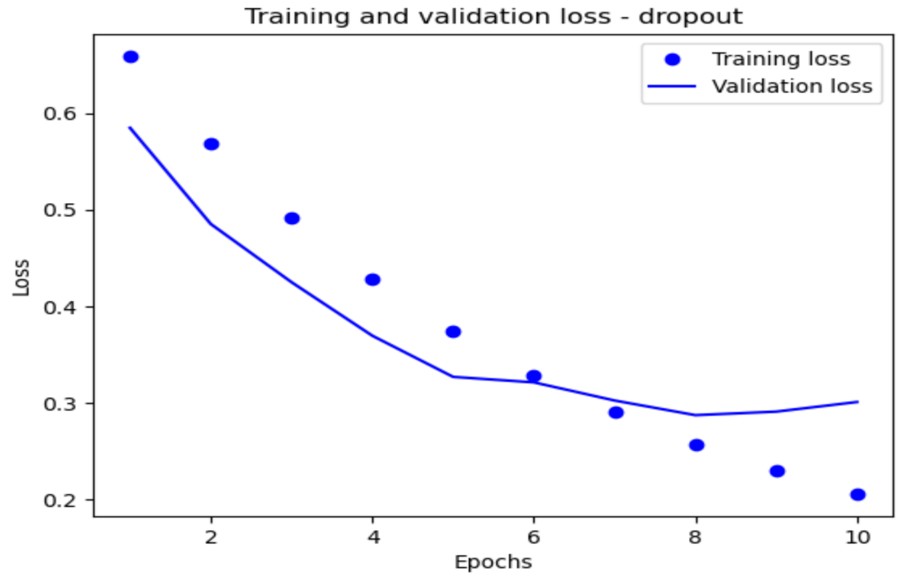
Can change to L1 or L1+ L2:
regularizers.l1(0.001)
regularizers.l1_l2(l1=0.001, l2=0.001)



Overfitting: IMDB

Dropout

```
model_dr = models.Sequential()
model_dr.add(Input((10000,)))
model_dr.add(layers.Dense(hidden, activation='relu’))
model_dr.add(layers.Dropout(0.5))
model_dr.add(layers.Dense(hidden, activation='relu'))
model_dr.add(layers.Dropout(0.5))
model_dr.add(layers.Dense(1, activation='sigmoid'))
```



Test set accuracy of 88%

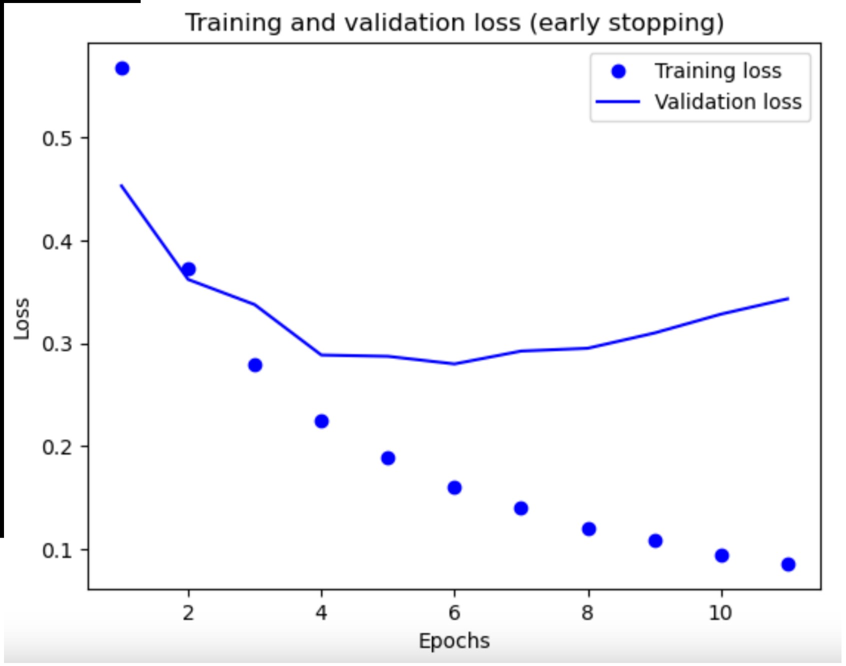
Overfitting: IMDB

Early stopping

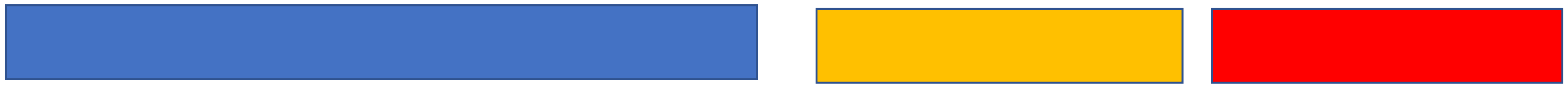
```
from tensorflow.keras.callbacks import EarlyStopping
…
early = EarlyStopping(monitor='val_loss', min_delta=0,
                      patience= 5,
                      verbose= True, mode='auto’)
callbacks = [early]
history = model.fit(partial_x_train,
            partial_y_train,
            epochs=20,
            batch_size=512,
            validation_data=(x_val, y_val),
            callbacks = callbacks)
```

Patience – number of epochs allowed for the error not to improve

Monitor – metric



Evaluating DL models



Training

Used to train models

Validation (*dev set*)

Used to optimize models/ hiper-parameters

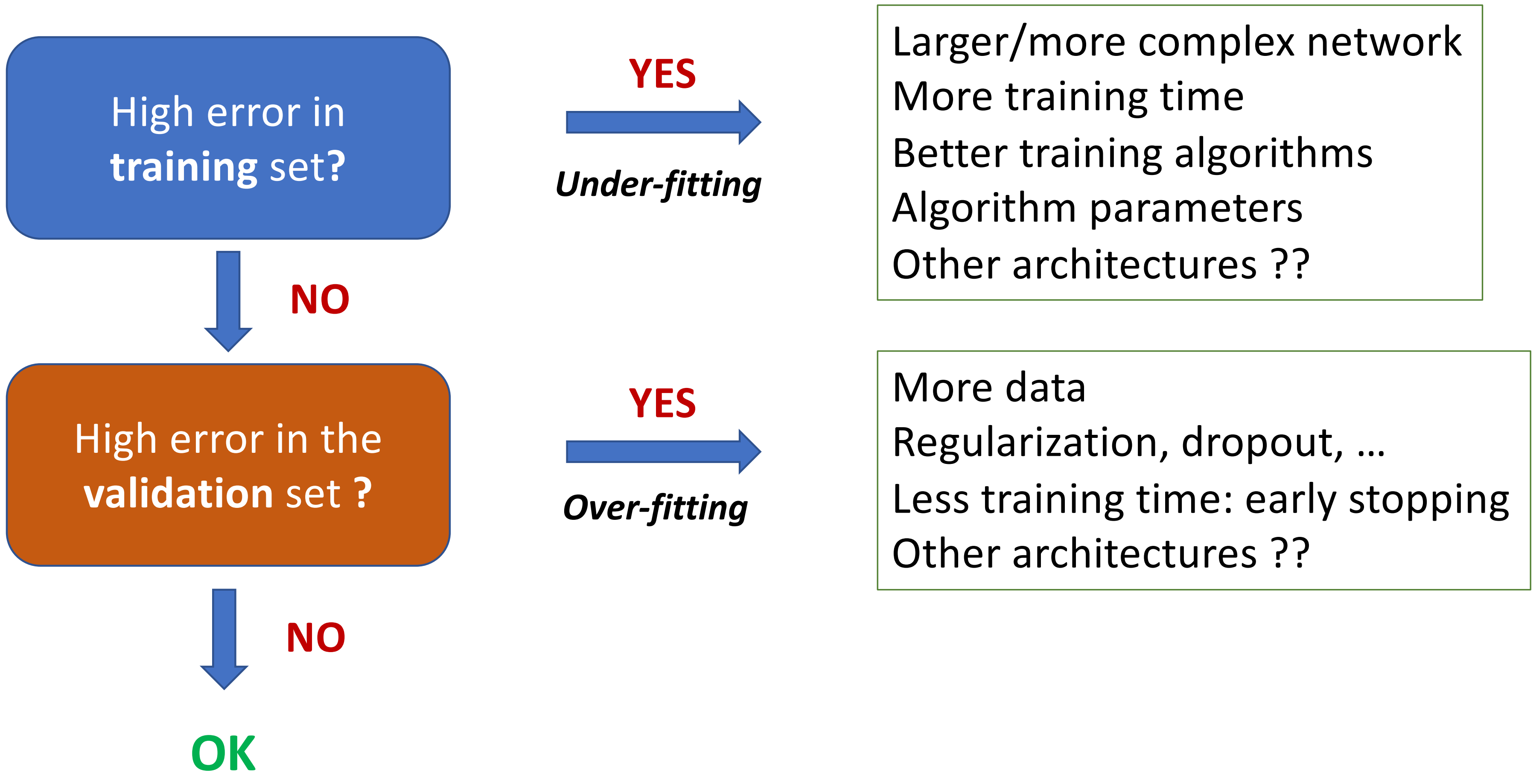
Test

Used to estimate error in the final model and comparing models

Over and underfitting

Error in **training** *Bias*	⬆	⬇	⬇
	Under-fitting		
Error in **validation** *Variance*	⬆	⬆	⬇
		Over-fitting	OK

Workflow


High error in **training** set?
YES
Under-fitting
Larger/more complex network
More training time
Better training algorithms
Algorithm parameters
Other architectures ??
NO
High error in the **validation** set ?
YES
Over-fitting
More data
Regularization, dropout, …
Less training time: early stopping
Other architectures ??
NO
OK

Hyperparameter optimization in DNNs

As with all supervised learning approaches, DL models should be the subject of a hyper-parameter optimization process to ensure their best performance. In this case, the most relevant hyper-parameters include:

- Number of layers
- Number of neurons in each layer
- Activation functions to use
- Training algorithm and its parameters (e.g. learning rate)
- Use of dropout and its rate
- Use of regularization and associated parameters
- Using early stopping or not; related parameters

Hyperparameter optimization

Jupyter Notebook:
ex_optimiz_hyperpar.ipynb

```
def setup_model(topo, dropout_rate, input_size, output_size):
    …
```

Sets models for multiclass classification, receiving topology and dropout rate
This can be changed for binary classification

```
def train_dnn(model, alg, lr, Xtrain, Ytrain, epochs = 5, batch_size = 64):
…
```

Trains model (DNN) for multiclass problems; receives algorithm, learning rate, epochs, batch_size
Adapt to binary classification/ other metrics

```
def dnn_optimization(opt_params, Xtrain, Ytrain, Xval, Yval, iterations = 10, verbose = True):
…
```

Hyperparameter optimization (opt_params gives the list of options for each parameter)
Algorithm used: randomized search

Example: hyperparameter optimization

```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
…
opt_pars = {"topology":[[100], [100,50], [250], [250,100]],
          "algorithm": [ "adam", "rmsprop", "sgd_momentum"],
          "lr": [0.01, 0.001],
          "dropout": [0, 0.2, 0.5]}

best_config, best_acc = dnn_optimization(opt_pars, train_images,
                              train_labels, test_images, test_labels)
print(best_config)
print(best_acc)
```