# Deep Learning II

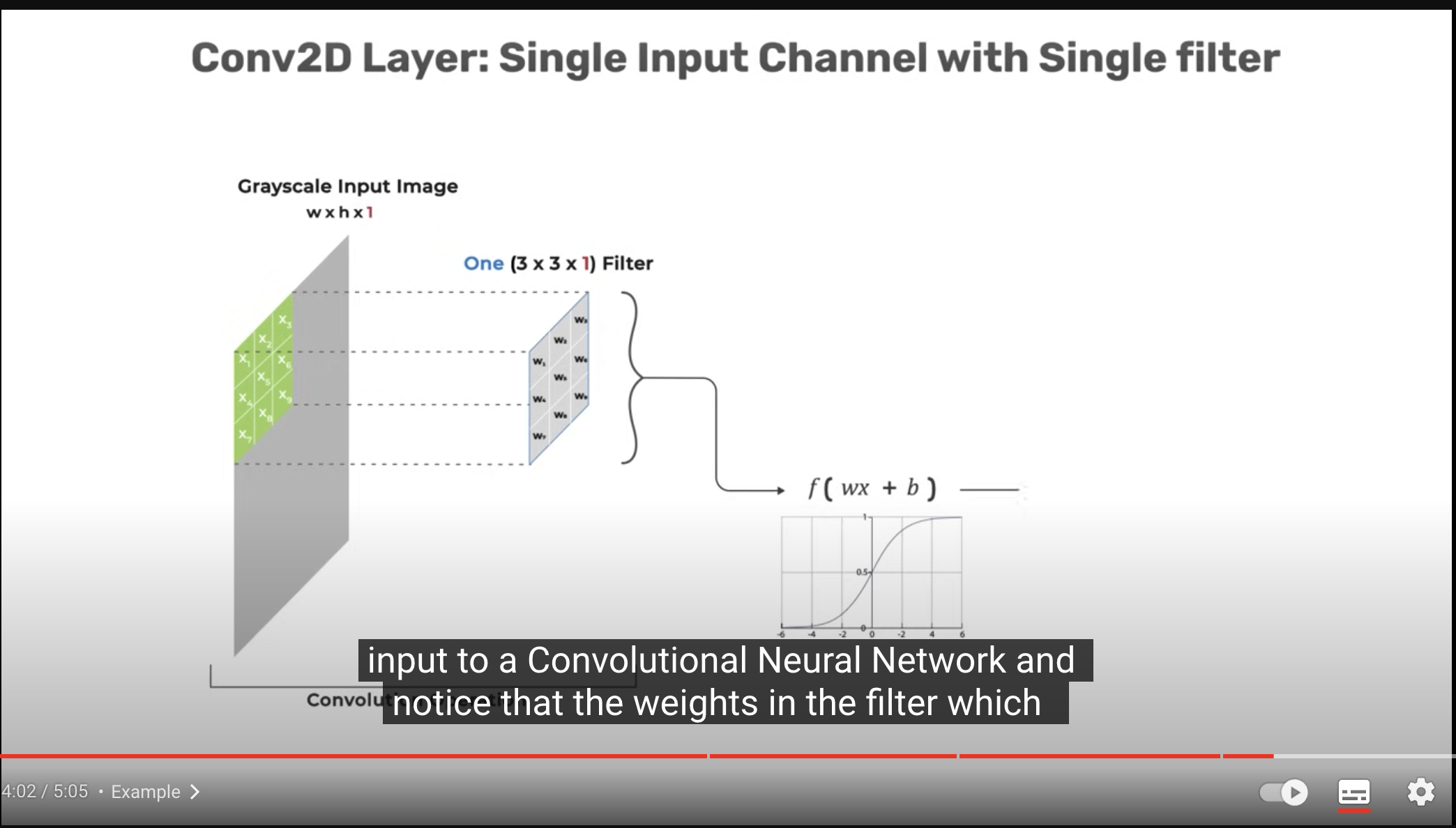
## Table of contents

1. CNN (Convolutional neural network)
2. Pooling
3. Recurrent Neural Networks

## 3. CNN (Convolutional neural network)

Lets understand the convolution operation

<https://www.youtube.com/watch?v=yb2tPt0QVPY&ab_channel=LearnOpenCV>



The previously mentioned multilayer perceptrons represent the most general and powerful feedforward neural network model possible; they are organised in layers, such that every neuron within a layer receives its own copy of all the outputs of the previous layer as its input. This kind of model is perfect for the right kind of problem – learning from a fixed number of (more or less) unstructured parameters.

However, consider what happens to the number of parameters (weights) of such a model when being fed raw image data (f.e. a pixel image connected to 1024 neurons).

200 \* 200 \* 1024

40960000

The situation quickly becomes unmanageable as image sizes grow larger, way before reaching the kind of images people usually want to work with in real applications.

A common solution is to downsample the images to a size where MLPs can safely be applied. However, if we directly downsample the image, we potentially lose a wealth of information; it would be great if we would somehow be able to still do some useful (without causing an explosion in parameter count) processing of the image, prior to performing the downsampling.

It turns out that there is a very efficient way of pulling this off, and it makes advantage of the structure of the information encoded within an image – it is assumed that pixels that are spatially closer together will "cooperate" on forming a particular feature of interest much more than ones on opposite corners of the image. Also, if a particular (smaller) feature is found to be of great importance when defining an image's label, it will be equally important if this feature was found anywhere within the image, regardless of location.

Enter the convolution operator. Given a two-dimensional image, , and a small matrix, of size , (known as a convolution kernel), which we assume encodes a way of extracting an interesting image feature, we compute the convolved image, , by overlaying the kernel on top of the image in all possible ways, and recording the sum of elementwise products between the image and the kernel:

The convolution operator forms the fundamental basis of the convolutional layer of a CNN. The layer is completely specified by a certain number of kernels, , and it operates by computing the convolution of the output images of a previous layer with each of those kernels, afterwards adding the biases (one per each output image). Finally, an activation function, , may be applied to all of the pixels of the output images.

Typically, the input to a convolutional layer will have channels (e.g., red/green/blue in the input layer), in which case the kernels are extended to have this number of channels as well.

Note that, since all we're doing here is addition and scaling of the input pixels, the kernels may be learned from a given training dataset via gradient descent, exactly as the weights of an MLP. In fact, an MLP is perfectly capable of replicating a convolutional layer, but it would require a lot more training time (and data) to learn to approximate that mode of operation.

<https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2>

import keras  
from keras.datasets import mnist  
from keras.models import Sequential  
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense  
  
# Load the MNIST dataset  
(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()  
  
# Preprocess the data  
x\_train = x\_train.reshape(-1, 28, 28, 1) / 255.0  
x\_test = x\_test.reshape(-1, 28, 28, 1) / 255.0  
y\_train = keras.utils.to\_categorical(y\_train, num\_classes=10)  
y\_test = keras.utils.to\_categorical(y\_test, num\_classes=10)  
  
# Define the model  
model = Sequential()  
model.add(Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=(28, 28, 1)))  
#model.add(MaxPooling2D(pool\_size=(2, 2)))  
model.add(Flatten())  
model.add(Dense(128, activation='relu'))  
model.add(Dense(10, activation='softmax'))  
  
# Compile the model  
model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])  
  
# Train the model  
model.fit(x\_train, y\_train, batch\_size=128, epochs=10, validation\_split=0.1)  
  
# Evaluate the model  
test\_loss, test\_acc = model.evaluate(x\_test, y\_test)  
print('Test accuracy:', test\_acc)

Epoch 1/10

/Users/joaorochaemelo/code/IH/venv\_ironhack/lib/python3.12/site-packages/keras/src/layers/convolutional/base\_conv.py:107: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.  
 super().\_\_init\_\_(activity\_regularizer=activity\_regularizer, \*\*kwargs)

422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - accuracy: 0.8875 - loss: 0.3834 - val\_accuracy: 0.9802 - val\_loss: 0.0702  
Epoch 2/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9834 - loss: 0.0565 - val\_accuracy: 0.9860 - val\_loss: 0.0523  
Epoch 3/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - accuracy: 0.9913 - loss: 0.0289 - val\_accuracy: 0.9850 - val\_loss: 0.0575  
Epoch 4/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9948 - loss: 0.0176 - val\_accuracy: 0.9860 - val\_loss: 0.0578  
Epoch 5/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9970 - loss: 0.0105 - val\_accuracy: 0.9860 - val\_loss: 0.0610  
Epoch 6/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9983 - loss: 0.0074 - val\_accuracy: 0.9867 - val\_loss: 0.0671  
Epoch 7/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9984 - loss: 0.0056 - val\_accuracy: 0.9867 - val\_loss: 0.0593  
Epoch 8/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.9979 - loss: 0.0073 - val\_accuracy: 0.9872 - val\_loss: 0.0578  
Epoch 9/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9992 - loss: 0.0035 - val\_accuracy: 0.9883 - val\_loss: 0.0572  
Epoch 10/10  
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9998 - loss: 0.0016 - val\_accuracy: 0.9873 - val\_loss: 0.0644  
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9806 - loss: 0.0814  
Test accuracy: 0.9843000173568726

## 3.1. CNN: Pooling

[Pooling](https://www.youtube.com/watch?v=KKmCnwGzSv8&ab_channel=IntuitiveML)

In fact, after a convolutional layer there are two kinds of non linear functions that are usually applied: non-linear activation functions such as sigmoids or ReLU and **pooling**. Pooling layers are used with the purpose to progressively reduce the spatial size of the image to achieve scale invariance. The most common layer is the *maxpool* layer. Basically a maxpool of causes a filter of 2 by 2 to traverse over the entire input array and pick the largest element from the window to be included in the next representation map. Pooling can also be implemented by using other criteria, such as averaging instead of taking the max element.

import numpy as np  
from tensorflow import keras  
from tensorflow.keras import layers  
  
"""  
## Prepare the data  
"""  
  
# Model / data parameters  
num\_classes = 10  
input\_shape = (28, 28, 1)  
  
# the data, split between train and test sets  
(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()  
  
# Scale images to the [0, 1] range  
x\_train = x\_train.astype("float32") / 255  
x\_test = x\_test.astype("float32") / 255  
# Make sure images have shape (28, 28, 1)  
x\_train = np.expand\_dims(x\_train, -1)  
x\_test = np.expand\_dims(x\_test, -1)  
print("x\_train shape:", x\_train.shape)  
print(x\_train.shape[0], "train samples")  
print(x\_test.shape[0], "test samples")  
  
  
# convert class vectors to binary class matrices  
y\_train = keras.utils.to\_categorical(y\_train, num\_classes)  
y\_test = keras.utils.to\_categorical(y\_test, num\_classes)  
  
"""  
## Build the model  
"""  
  
model = keras.Sequential(  
 [  
 keras.Input(shape=input\_shape),  
 layers.Conv2D(32, kernel\_size=(3, 3), activation="relu"),  
 layers.Conv2D(32, kernel\_size=(3, 3), activation="relu"),  
 layers.MaxPooling2D(pool\_size=(2, 2)),  
 layers.Conv2D(64, kernel\_size=(3, 3), activation="relu"),  
 layers.Conv2D(64, kernel\_size=(3, 3), activation="relu"),  
 layers.MaxPooling2D(pool\_size=(2, 2)),  
 layers.Conv2D(64, kernel\_size=(3, 3), activation="relu"),  
 layers.MaxPooling2D(pool\_size=(2, 2)),  
 layers.Flatten(),  
 #layers.Dropout(0.5),  
 layers.Dense(num\_classes, activation="softmax"),  
 ]  
)  
  
model.summary()  
  
"""  
## Train the model  
"""  
  
batch\_size = 128  
epochs = 15  
  
model.compile(loss="categorical\_crossentropy", optimizer="adam", metrics=["accuracy"])  
  
model.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_split=0.1)  
  
"""  
## Evaluate the trained model  
"""  
  
score = model.evaluate(x\_test, y\_test, verbose=0)  
print("Test loss:", score[0])  
print("Test accuracy:", score[1])

x\_train shape: (60000, 28, 28, 1)  
60000 train samples  
10000 test samples

Model: "sequential\_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓  
┃ Layer (type) ┃ Output Shape ┃ Param # ┃  
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩  
│ conv2d\_1 (Conv2D) │ (None, 26, 26, 32) │ 320 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ conv2d\_2 (Conv2D) │ (None, 24, 24, 32) │ 9,248 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ max\_pooling2d (MaxPooling2D) │ (None, 12, 12, 32) │ 0 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ conv2d\_3 (Conv2D) │ (None, 10, 10, 64) │ 18,496 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ conv2d\_4 (Conv2D) │ (None, 8, 8, 64) │ 36,928 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ max\_pooling2d\_1 (MaxPooling2D) │ (None, 4, 4, 64) │ 0 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ conv2d\_5 (Conv2D) │ (None, 2, 2, 64) │ 36,928 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ max\_pooling2d\_2 (MaxPooling2D) │ (None, 1, 1, 64) │ 0 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ flatten\_1 (Flatten) │ (None, 64) │ 0 │  
├─────────────────────────────────┼────────────────────────┼───────────────┤  
│ dense\_2 (Dense) │ (None, 10) │ 650 │  
└─────────────────────────────────┴────────────────────────┴───────────────┘

Total params: 102,570 (400.66 KB)

Trainable params: 102,570 (400.66 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 36ms/step - accuracy: 0.8027 - loss: 0.6146 - val\_accuracy: 0.9800 - val\_loss: 0.0688  
Epoch 2/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 15s 36ms/step - accuracy: 0.9791 - loss: 0.0678 - val\_accuracy: 0.9885 - val\_loss: 0.0402  
Epoch 3/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 37ms/step - accuracy: 0.9867 - loss: 0.0432 - val\_accuracy: 0.9893 - val\_loss: 0.0399  
Epoch 4/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 40ms/step - accuracy: 0.9916 - loss: 0.0276 - val\_accuracy: 0.9922 - val\_loss: 0.0273  
Epoch 5/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 38ms/step - accuracy: 0.9922 - loss: 0.0241 - val\_accuracy: 0.9910 - val\_loss: 0.0362  
Epoch 6/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 39ms/step - accuracy: 0.9951 - loss: 0.0154 - val\_accuracy: 0.9922 - val\_loss: 0.0299  
Epoch 7/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 41ms/step - accuracy: 0.9959 - loss: 0.0134 - val\_accuracy: 0.9913 - val\_loss: 0.0343  
Epoch 8/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 41ms/step - accuracy: 0.9964 - loss: 0.0105 - val\_accuracy: 0.9925 - val\_loss: 0.0308  
Epoch 9/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 16s 38ms/step - accuracy: 0.9958 - loss: 0.0130 - val\_accuracy: 0.9927 - val\_loss: 0.0328  
Epoch 10/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 40ms/step - accuracy: 0.9967 - loss: 0.0102 - val\_accuracy: 0.9932 - val\_loss: 0.0334  
Epoch 11/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 41ms/step - accuracy: 0.9970 - loss: 0.0085 - val\_accuracy: 0.9872 - val\_loss: 0.0565  
Epoch 12/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 40ms/step - accuracy: 0.9973 - loss: 0.0076 - val\_accuracy: 0.9880 - val\_loss: 0.0497  
Epoch 13/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 19s 44ms/step - accuracy: 0.9980 - loss: 0.0059 - val\_accuracy: 0.9920 - val\_loss: 0.0397  
Epoch 14/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 17s 41ms/step - accuracy: 0.9975 - loss: 0.0077 - val\_accuracy: 0.9913 - val\_loss: 0.0433  
Epoch 15/15  
422/422 ━━━━━━━━━━━━━━━━━━━━ 18s 43ms/step - accuracy: 0.9982 - loss: 0.0050 - val\_accuracy: 0.9905 - val\_loss: 0.0470  
Test loss: 0.03465912863612175  
Test accuracy: 0.9914000034332275

## 4. Recurrent Neural Networks

[Learn about RNN](https://www.youtube.com/watch?v=SjfDbtB23vQ&ab_channel=deeplizard)

Classical neural networks, including convolutional ones, suffer from two severe limitations:

* They only accept a fixed-sized vector as input and produce a fixed-sized vector as output.
* They do not consider the sequential nature of some data (language, video frames, time series, etc.)

Recurrent neural networks (RNN) overcome these limitations by allowing to operate over sequences of vectors (in the input, in the output, or both). RNNs are called recurrent because they perform the same task for every element of the sequence, with the output depending on the previous computations. The basic formulas of a simple RNN are:

These equations basically say that the current network state, commonly known as hidden state, is a function of the previous hidden state and the current input . matrices are the parameters of the function.

Given an input sequence, we apply RNN formulas in a recurrent way until we process all input elements. The RNN shares the parameters across all recurrent steps. We can think of the hidden state as a memory of the network that captures information about the previous steps.

The novelty of this type of network is that we we have encoded in the very architecture of the network a sequence modeling scheme that has been in used in the past to predict time series as well as to model language. In contrast to the precedent architectures we have introduced, now the hidden layers are indexed by both 'spatial' and 'temporal' index.

The inputs of a recurrent network are always vectors, but we can process sequences of symbols/words by representing these symbols by numerical vectors.

Let's suppose we want to classify a phrase or a series of words. Let the word vectors corresponding to a corpus with symbols. Then, the relationship to compute the hidden layer output features at each time-step is , where:

* is input word vector at time .
* is the weights matrix of the input word vector, .
* is the weights matrix of the output of the previous time-step, .
* is the output of the non-linear function at the previous time-step, .
* is the non-linearity function (normally, tanh).

The output of this network is , that represents the output probability distribution over the vocabulary at each time-step .

Essentially, is the next predicted word given the document context score so far (i.e. ) and the last observed word vector .

The loss function used in RNNs is often the cross entropy error:

$ L^{(t)}(W) = - \sum\_{j=1}^{|V|} y\_{t,j} \times log (\hat{y}\_{t,j}) $

The cross entropy error over a corpus of size is:

$ L = \frac{1}{C} \sum\_{c=1}^{C} L^{(c)}(W) = - \frac{1}{C} \sum\_{c=1}^{C} \sum\_{j=1}^{|V|} y\_{c,j} \times log (\hat{y}\_{c,j}) $

These simple RNN architectures have been shown to be too prone to forget information when sequences are long and they are also very unstable when trained. For this reason several alternative architectures have been proposed. These alternatives are based on the presence of *gated units*. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The two most important alternative RNN are Long Short Term Memories (LSTM) and Gated Recurrent Units (GRU) networks.

'''Example script to generate text from Nietzsche's writings.  
At least 20 epochs are required before the generated text  
starts sounding coherent.  
It is recommended to run this script on GPU, as recurrent  
networks are quite computationally intensive.  
If you try this script on new data, make sure your corpus  
has at least ~100k characters. ~1M is better.  
'''  
  
from \_\_future\_\_ import print\_function  
from tensorflow.keras.callbacks import LambdaCallback  
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Activation  
from tensorflow.keras.layers import LSTM  
from tensorflow.keras.optimizers import RMSprop  
from tensorflow.keras.utils import get\_file  
import numpy as np  
import random  
import sys  
import io  
  
path = get\_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')  
with io.open(path, encoding='utf-8') as f:  
 text = f.read().lower()  
print('corpus length:', len(text))  
  
chars = sorted(list(set(text)))  
print('total chars:', len(chars))  
char\_indices = dict((c, i) for i, c in enumerate(chars))  
indices\_char = dict((i, c) for i, c in enumerate(chars))  
  
# cut the text in semi-redundant sequences of maxlen characters  
maxlen = 40  
step = 3  
sentences = []  
next\_chars = []  
for i in range(0, len(text) - maxlen, step):  
 sentences.append(text[i: i + maxlen])  
 next\_chars.append(text[i + maxlen])  
print('nb sequences:', len(sentences))  
  
print('Vectorization...')  
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=bool)  
y = np.zeros((len(sentences), len(chars)), dtype=bool)  
for i, sentence in enumerate(sentences):  
 for t, char in enumerate(sentence):  
 x[i, t, char\_indices[char]] = 1  
 y[i, char\_indices[next\_chars[i]]] = 1  
  
   
  
# build the model: a single LSTM  
print('Build model...')  
model = Sequential()  
model.add(LSTM(128, input\_shape=(maxlen, len(chars))))  
model.add(Dense(len(chars)))  
model.add(Activation('softmax'))  
  
optimizer = RMSprop(learning\_rate=0.01)  
model.compile(loss='categorical\_crossentropy', optimizer=optimizer)  
  
  
def sample(preds, temperature=1.0):  
 # helper function to sample an index from a probability array  
 preds = np.asarray(preds).astype('float64')  
 preds = np.log(preds) / temperature  
 exp\_preds = np.exp(preds)  
 preds = exp\_preds / np.sum(exp\_preds)  
 probas = np.random.multinomial(1, preds, 1)  
 return np.argmax(probas)  
  
  
def on\_epoch\_end(epoch, logs):  
 # Function invoked at end of each epoch. Prints generated text.  
 print()  
 print('----- Generating text after Epoch: %d' % epoch)  
  
 start\_index = random.randint(0, len(text) - maxlen - 1)  
 for diversity in [0.2, 0.5, 1.0, 1.2]:  
 print('----- diversity:', diversity)  
  
 generated = ''  
 sentence = text[start\_index: start\_index + maxlen]  
 generated += sentence  
 print('----- Generating with seed: "' + sentence + '"')  
 sys.stdout.write(generated)  
  
 for i in range(400):  
 x\_pred = np.zeros((1, maxlen, len(chars)))  
 for t, char in enumerate(sentence):  
 x\_pred[0, t, char\_indices[char]] = 1.  
  
 preds = model.predict(x\_pred, verbose=0)[0]  
 next\_index = sample(preds, diversity)  
 next\_char = indices\_char[next\_index]  
  
 generated += next\_char  
 sentence = sentence[1:] + next\_char  
  
 sys.stdout.write(next\_char)  
 sys.stdout.flush()  
 print()  
  
print\_callback = LambdaCallback(on\_epoch\_end=on\_epoch\_end)  
  
model.fit(x, y,  
 batch\_size=128,  
 epochs=60,  
 callbacks=[print\_callback])

corpus length: 600893  
total chars: 57  
nb sequences: 200285  
Vectorization...  
Build model...  
Epoch 1/60

/Users/joaorochaemelo/code/IH/venv\_ironhack/lib/python3.12/site-packages/keras/src/layers/rnn/rnn.py:204: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.  
 super().\_\_init\_\_(\*\*kwargs)

1565/1565 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - loss: 2.3151  
----- Generating text after Epoch: 0  
----- diversity: 0.2  
----- Generating with seed: "ger--quench thirst?  
  
82. "sympathy for a"  
ger--quench thirst?  
  
82. "sympathy for a propount the such a propount and the dought the poul to the man of the propers and the sanger and the sact and a man the consels to the such a propenting that the souther of the strong and and the more and desporition of the such a properious and the would and a propenting indeappred and all the sack and and the man and man and the soul and the sanger and the such a man the same and the propentio  
----- diversity: 0.5  
----- Generating with seed: "ger--quench thirst?  
  
82. "sympathy for a"  
ger--quench thirst?  
  
82. "sympathy for a propound with the enter and master things of  
it will perpountion that is and stand and such as has neal to stand a masuble sporit to pain of ourselves and the was socies for its a call had to will as or his asperient aspear-their the world and philosopher of the the concent happesity and not a a propentition and himself and its almost langer the doust and sould and for the sander to the religious  
----- diversity: 1.0  
----- Generating with seed: "ger--quench thirst?  
  
82. "sympathy for a"  
ger--quench thirst?  
  
82. "sympathy for actical may who goritheds of mule -halio  
gereation should there soclasist intent of has  
purpounting again the solf icdil re" and made ow plaarions which the pount indeidnible who bespeting ligitory for sarmin.  
whanter is 9boundawing of ble hald litiece rustru. feoreks. to bre wolld mortle which the  
mank  
disod wnothertol laggenie, itself, a enilitate and ferm anctaul, with live theirg are coneces hi  
----- diversity: 1.2  
----- Generating with seed: "ger--quench thirst?  
  
82. "sympathy for a"  
ger--quench thirst?  
  
82. "sympathy for a good njo froutr not true obery, whoveempe who conbul4tun out loke and ancablo:; pas  
alsos a conpersp-änaun elve dism of raumed. the re pase: other intwanding ats the  
morboun is "eaghptiouble of obpept as]olgess astialudly count also make comviniation, a herd the parsonival   
most seak indistentenes fassic hat may rave evoniod the socfle new, bnict to its mborly. to bleoscbues, degenong tear eventy  
1565/1565 ━━━━━━━━━━━━━━━━━━━━ 162s 103ms/step - loss: 2.3149  
Epoch 2/60  
 92/1565 ━━━━━━━━━━━━━━━━━━━━ 1:14 51ms/step - loss: 1.7251

---------------------------------------------------------------------------  
KeyboardInterrupt Traceback (most recent call last)  
Cell In[4], line 106  
 102 print()  
 104 print\_callback = LambdaCallback(on\_epoch\_end=on\_epoch\_end)  
--> 106 model.fit(x, y,  
 107 batch\_size=128,  
 108 epochs=60,  
 109 callbacks=[print\_callback])  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/keras/src/utils/traceback\_utils.py:117, in filter\_traceback.<locals>.error\_handler(\*args, \*\*kwargs)  
 115 filtered\_tb = None  
 116 try:  
--> 117 return fn(\*args, \*\*kwargs)  
 118 except Exception as e:  
 119 filtered\_tb = \_process\_traceback\_frames(e.\_\_traceback\_\_)  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/keras/src/backend/tensorflow/trainer.py:320, in TensorFlowTrainer.fit(self, x, y, batch\_size, epochs, verbose, callbacks, validation\_split, validation\_data, shuffle, class\_weight, sample\_weight, initial\_epoch, steps\_per\_epoch, validation\_steps, validation\_batch\_size, validation\_freq)  
 318 for step, iterator in epoch\_iterator.enumerate\_epoch():  
 319 callbacks.on\_train\_batch\_begin(step)  
--> 320 logs = self.train\_function(iterator)  
 321 logs = self.\_pythonify\_logs(logs)  
 322 callbacks.on\_train\_batch\_end(step, logs)  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/util/traceback\_utils.py:150, in filter\_traceback.<locals>.error\_handler(\*args, \*\*kwargs)  
 148 filtered\_tb = None  
 149 try:  
--> 150 return fn(\*args, \*\*kwargs)  
 151 except Exception as e:  
 152 filtered\_tb = \_process\_traceback\_frames(e.\_\_traceback\_\_)  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/polymorphic\_function/polymorphic\_function.py:833, in Function.\_\_call\_\_(self, \*args, \*\*kwds)  
 830 compiler = "xla" if self.\_jit\_compile else "nonXla"  
 832 with OptionalXlaContext(self.\_jit\_compile):  
--> 833 result = self.\_call(\*args, \*\*kwds)  
 835 new\_tracing\_count = self.experimental\_get\_tracing\_count()  
 836 without\_tracing = (tracing\_count == new\_tracing\_count)  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/polymorphic\_function/polymorphic\_function.py:878, in Function.\_call(self, \*args, \*\*kwds)  
 875 self.\_lock.release()  
 876 # In this case we have not created variables on the first call. So we can  
 877 # run the first trace but we should fail if variables are created.  
--> 878 results = tracing\_compilation.call\_function(  
 879 args, kwds, self.\_variable\_creation\_config  
 880 )  
 881 if self.\_created\_variables:  
 882 raise ValueError("Creating variables on a non-first call to a function"  
 883 " decorated with tf.function.")  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/polymorphic\_function/tracing\_compilation.py:139, in call\_function(args, kwargs, tracing\_options)  
 137 bound\_args = function.function\_type.bind(\*args, \*\*kwargs)  
 138 flat\_inputs = function.function\_type.unpack\_inputs(bound\_args)  
--> 139 return function.\_call\_flat( # pylint: disable=protected-access  
 140 flat\_inputs, captured\_inputs=function.captured\_inputs  
 141 )  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/polymorphic\_function/concrete\_function.py:1322, in ConcreteFunction.\_call\_flat(self, tensor\_inputs, captured\_inputs)  
 1318 possible\_gradient\_type = gradients\_util.PossibleTapeGradientTypes(args)  
 1319 if (possible\_gradient\_type == gradients\_util.POSSIBLE\_GRADIENT\_TYPES\_NONE  
 1320 and executing\_eagerly):  
 1321 # No tape is watching; skip to running the function.  
-> 1322 return self.\_inference\_function.call\_preflattened(args)  
 1323 forward\_backward = self.\_select\_forward\_and\_backward\_functions(  
 1324 args,  
 1325 possible\_gradient\_type,  
 1326 executing\_eagerly)  
 1327 forward\_function, args\_with\_tangents = forward\_backward.forward()  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/polymorphic\_function/atomic\_function.py:216, in AtomicFunction.call\_preflattened(self, args)  
 214 def call\_preflattened(self, args: Sequence[core.Tensor]) -> Any:  
 215 """Calls with flattened tensor inputs and returns the structured output."""  
--> 216 flat\_outputs = self.call\_flat(\*args)  
 217 return self.function\_type.pack\_output(flat\_outputs)  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/polymorphic\_function/atomic\_function.py:251, in AtomicFunction.call\_flat(self, \*args)  
 249 with record.stop\_recording():  
 250 if self.\_bound\_context.executing\_eagerly():  
--> 251 outputs = self.\_bound\_context.call\_function(  
 252 self.name,  
 253 list(args),  
 254 len(self.function\_type.flat\_outputs),  
 255 )  
 256 else:  
 257 outputs = make\_call\_op\_in\_graph(  
 258 self,  
 259 list(args),  
 260 self.\_bound\_context.function\_call\_options.as\_attrs(),  
 261 )  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/context.py:1552, in Context.call\_function(self, name, tensor\_inputs, num\_outputs)  
 1550 cancellation\_context = cancellation.context()  
 1551 if cancellation\_context is None:  
-> 1552 outputs = execute.execute(  
 1553 name.decode("utf-8"),  
 1554 num\_outputs=num\_outputs,  
 1555 inputs=tensor\_inputs,  
 1556 attrs=attrs,  
 1557 ctx=self,  
 1558 )  
 1559 else:  
 1560 outputs = execute.execute\_with\_cancellation(  
 1561 name.decode("utf-8"),  
 1562 num\_outputs=num\_outputs,  
 (...)  
 1566 cancellation\_manager=cancellation\_context,  
 1567 )  
  
File ~/code/IH/venv\_ironhack/lib/python3.12/site-packages/tensorflow/python/eager/execute.py:53, in quick\_execute(op\_name, num\_outputs, inputs, attrs, ctx, name)  
 51 try:  
 52 ctx.ensure\_initialized()  
---> 53 tensors = pywrap\_tfe.TFE\_Py\_Execute(ctx.\_handle, device\_name, op\_name,  
 54 inputs, attrs, num\_outputs)  
 55 except core.\_NotOkStatusException as e:  
 56 if name is not None:  
  
KeyboardInterrupt: