+----cursor-parking-lot-----+
+--30L-area–+ **<3** |
**|kiss from Boracchi** |
+----------------+-------------+

# LSTM

## Do you use word embedding with LSTM networks? How?

Yes, word embeddings are also commonly used in LSTM (Long Short-Term Memory) networks. In an LSTM network, the input is also a sequence of words, represented as word embeddings. These embeddings are typically learned during the training process and are used to represent the meaning of each word in a continuous, lower-dimensional space. The LSTM network then processes these embeddings, using its internal memory cells, to generate a new representation of the input that is suitable for a given task, such as language generation or text classification.

The word embeddings are usually fed as an input to the LSTM cell along with the previous hidden state, and the LSTM cell will output a new hidden state and an output. This process is repeated for each word in the input sequence, and the final hidden state can be used for the task at hand.

In general, LSTM networks can be used with a variety of input, not just word embedding, like character-level representation, audio, video etc.

## Can word embedding overfit?

Yes, it is possible for word embeddings to overfit in a kind neural network. This can happen when the embeddings are learned from a small or specific dataset, and then applied to a different, larger or more diverse dataset. In such cases, the embeddings may not generalise well to the new dataset, and the performance of the neural network may suffer as a result.

Overfitting can occur when the model becomes too complex and starts memorising the training data. For example, if the dataset is small, the model may learn the specific characteristics of the training data rather than the general features that are relevant to the task.

To avoid overfitting with word embeddings, it's common to use pre-trained embeddings that have been learned from a large, diverse dataset. These pre-trained embeddings can be fine-tuned on the specific task or dataset, but the initial embeddings will already contain useful information and general features that are relevant to the task.

Additionally, other techniques such as regularisation methods, or early stopping can be used to avoid overfitting in neural networks.
—
I would say it can overfit, like every network, when:
- The dataset isn't large enough: the network learns easily only the training set (so only the contexts of these) no making prediction absolutely erratic if the same words are found in different contexts
- If the network is too complex: the more complex the network, the easier it learn the training data so tends to overfit

You can detect it by keeping part of the dataset for testing after training and verifying performance over it

- Word embeddings, such as those learned by the word2vec model, can overfit if the training data is not diverse enough or if the model is too complex
- Overfitting occurs when a model is trained too well on the training data, and as a result, it performs poorly on unseen data.
- For example, if the training data only consists of a specific domain-specific language, the model will be more likely to overfit to that domain and may not perform well on other domains
- Additionally, if the model has too many parameters, it can easily fit to the noise in the training data, which can lead to poor generalisation performance on unseen data.
- To prevent overfitting, it is recommended to use a large and diverse dataset for training and to regularise the model. Techniques such as dropout and L2 regularisation can be used to prevent overfitting.

–

- Yes, word embedding models can overfit. Overfitting occurs when a model is trained too well on the training data, to the point that it performs poorly on new, unseen data. This can happen when the model is too complex or when there is not enough training data
- One way to detect overfitting in word embedding models is to evaluate the model's performance on a held-out validation set. If the model performs well on the training data but poorly on the validation data, it is likely overfitting. Another way is to evaluate the embeddings similarity with pre-trained embeddings, if the embeddings are not similar, it means overfitting has occurred.
- To prevent overfitting, one can use techniques such as regularisation, early stopping, and dropout. Regularisation is a technique that helps to prevent overfitting by adding a penalty term to the loss function that discourages the model from having too many large weights. Early stopping is a technique that stops the training process when the model's performance on the validation set stops improving. Dropout is a technique that randomly drops out a certain percentage of neurons during training to prevent the model from relying too much on any one neuron.

## What is an LSTM and why does it help with vanishing gradient?

An LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) architecture that is capable of learning long-term dependencies in sequential data. Unlike traditional RNNs, LSTMs have memory cells that can store information for longer periods of time, making them well-suited for tasks such as language modelling, speech recognition, and time series prediction. LSTMs are commonly used in natural language processing and other sequential data tasks.

The idea is to have some gate mechanism where you can “write”,”keep” or “read” information in the memory cells whenever the gate is turned on.
The gates are based on the Constant Error Carousel. Decisions (e.g. read/write) are based on the output of a Sigmoid, while the loop is ruled by a ReLu activation function.

The LSTM can help with the vanishing gradient issue thanks to the constant error carousel and its complex internal structure of the gates and memory handling mechanisms.

LSTMs solve the problem of vanishing gradient by using a unique additive gradient structure that includes direct access to the forget gate’s activations, enabling the network to encourage desired

behaviour from the error gradient using frequent gates update on every time step of the learning process.

## What is the Constant Error Carousel?

The Constant Error Carousel (CEC) as you might well know is the magic of the LSTM in that it prevents vanishing gradients. It's denoted as follows:

$c_{t+1} = c_t *$ forget gate + new input $*$ input gate

In the case of regular RNNs during backpropagation, the derivative of an activation function, such as a sigmoid, will be less than one. Therefore over time, the repeated multiplication of that value against the weights $f'(x) * W$ will lead to a vanishing gradient.

In the case of an LSTM, we only multiply the cell state by a forget gate, which acts as both the weights and the activation function for the cell state. As long as that forget gate equals one, the information from the previous cell state passes through unchanged.

So in our case, the parameters we derived $h_t$ and $h_{t-1}$ are the hidden states which as part of the input vector $I^t$, will be filtered through the forget gate, and affect the cell state accordingly. The $\delta W$ we solved for is the parameter we use to update all our weights/gates.

## Which are the configurations of a sequence-to-sequence?

We can build different configurations with sequence-to-sequence:

- One-to-one: translates one input into one output (e.g Image Classification)
- One-to-many: one input and you want to predict many outputs. (e.g. Captioning)
- Many-to-one: sequence of input and you want one output (e.g. Text Classification)
- Many-to-many: (e.g. Video Classification-Video Captioning)
- Synced many-to-many: the input and output sequence have same length
- Sequence-to-Sequence: enter a sequence, you have a state and the machine is started with this state, and from there it generates a sequence (e.g. Text prediction, Question-Answer)

## When could we prefer RNN instead of LSTM?

RNN are suitable in the cases where we need to handle very short sequences, because these kinds of networks suffer from the vanishing gradient issue. RNNs are better suited with respect to the LSTM layer because they have much fewer parameters, and are less prone to overfitting when having a small dataset.

LSTM cells, instead, are better suited for longer sequences and when a more complex handling of the memory is required. LSTM cells are more computationally intensive to be trained and at inference time, compared to the RNNs.

## Why do we need an attention mechanism? Aren't Long Short-Term Memories enough?

LSTMs have some limitations when it comes to processing very large input sequences. They have a fixed size internal memory, which can make it difficult to store and retrieve long-term dependencies in the input sequence. For this reason we need to add an attention mechanism, in order to make the model focus on specific parts of the input, rather than on the entire sequence. This is useful for example during translation tasks, where the model needs to focus on different parts of the input sequence, at different times, in order to generate the correct translation.

## Describe briefly the seq2seq model. Can it be used with RNN or can it only be used with LSTM? Why?

- The seq2seq model follows the classical encoder-decoder architecture ⇒ The encoder takes a sequence of input data (such as a sentence in one language) and converts it into a fixed-length representation, called the “context vector”. The decoder then uses this context vector to generate a sequence of output data (such as a translation of the input sentence in another language).
- At training, the decoder does not feed the output of each time step to the next; the input to the decoder time steps are the targets from the training set. At inference times, the decoder feeds the output of each time step into the input of the next.
- Both the decoder and the encoder can use RNNs, although they are rarely used ⇒ They suffer from the vanishing gradient

# CNN

## How data representation is learned via convolutional neural networks?

In a convolutional neural network (CNN), data representation is learned through a process called convolution. The input to a CNN is typically an image, which is represented as a multi-dimensional array of pixel values. The convolution operation involves taking a small, fixed-sized filter (also called a kernel or weight matrix) and sliding it across the input image, element-wise multiplying it with the underlying section of the image, and then summing the results. This process is repeated for multiple filters, each with a different set of weights, creating a feature map.

By applying multiple filters to the input image, CNNs are able to learn different features at different levels of abstraction, such as edges, textures, and shapes. In this way, CNNs can extract useful information from the input image and create a condensed, abstract representation of the data.

After multiple convolution layers, the CNN will have learned a variety of features at different levels of abstraction. These features are then passed through additional layers such as pooling, fully connected layers to make a prediction or a decision.

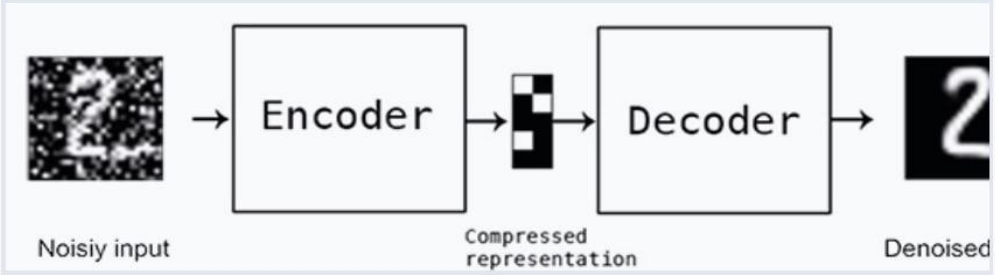
Additionally, CNNs often use a technique called weight sharing, which means that the same set of filter weights is used across the entire input image. This helps to reduce the number of parameters in the model and make it more computationally efficient.

In summary, the main process of data representation learning in CNNs is the convolution operation, which is performed multiple times using different filters, with the goal of extracting useful features from the input data.

# Exercise

## QUESTION 3: CONVOLUTIONAL NEURAL NETWORKS

In the picture, the general schema of a Denoising Autoencoder is reported with 256x256 greyscale images as input and a desired compressed representation of 8 float numbers.




computed them, e.g., 3x5x5=45 and not just 45. (Yes you can use the calculator, but we are more interested in the formula than on the numbers)
(30 Points)

```
net = tfkl.Conv2D(128, (3, 3), padding='same')(inp) # par=3x3x1x128 + 128 = 1280, out=256x256x128
net = tfkl.MaxPooling2D((2,2), padding='same')(net) # par=0, out=128x128x128
net = tfkl.Conv2D(64, (3,3), padding='same')(net) # par=3x3x128x64 + 64 = 73792, out=128x128x64
net = tfkl.MaxPooling2D((2,2), padding='same')(net) # par=0, out=64x64x64
net = tfkl.Conv2D(32, (3,3), padding='same')(net) # par=3x3x64x32 + 32 = 18464, out=64x64x32

net = tfkl.UpSampling2D((2, 2))(net) # par=0, out=128x128x32
net = tfkl.Conv2D(64, (3, 3), padding='same')(net) # par= 64x3x3x32+64=18496, out=128x128x64
net = tfkl.UpSampling2D((2,2))(net) # par=0, out=256x256x64
net = tfkl.Conv2D(128, (3, 3), padding='same')(net) # par=128*64*3*3 + 128=73.856, out= 256x256x128
out = tfkl.Conv2D(1, (3, 3), padding='same')(net) # par=3x3x1x128 + 1=1153, out= 256x256x1
```

## Describe the training procedure for the Denoising Autoencoder in the picture in terms of the dataset you need and the loss function you would use.

I'd use a dataset of clean images, such as the MNIST digit dataset in this case. To generate an (x, y) pair I would take a clean image as y, and then add some noise to get a noisy x. Then I would train the network to output the original image from the noisy one. Since it looks like we are dealing with black and white digits, the output would be a 2d tensor of values between 0 and 1 that are strongly polarised to either 0 or 1, so I would use cross entropy on every pixel as a loss function

```
import keras
from keras.utils  import np_utils
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Flatten, GlobalAveragePooling2D, Dropout
from keras.layers import Conv2D, MaxPooling2D, AveragePooling2D, Dropout
from keras.applications import MobileNet, VGG16

patchsize          = 64
pool_size          = (2,2)
kernel_size        = (3,3)
nb_filters         = 8

model = Sequential()

model.add(Conv2D(nb_filters,  (3,3)  , input_shape=(patchsize, patchsize, 3), padding
= "same"))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = pool_size))
model.add(Conv2D(nb_filters*2, (3,3), padding = "same"))
model.add(Activation('relu'))
model.add(Conv2D(nb_filters*2, (3,3), padding = "same"))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = pool_size))
model.add(Conv2D(nb_filters*3, (5,5), padding = "same"))
model.add(Activation('relu'))
model.add(Conv2D(nb_filters*3, (5,5), padding = "same"))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = pool_size))
model.add(GlobalAveragePooling2D())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(5))
model.add(Activation('softmax'))
```

Answer the following questions providing a short explanation for each

1. Is this a linear model?
2. Is this a classification or a regression network?
3. What is the input size? What is the output size?
4. Is this a fully convolutional neural network?
5. Once trained, can we feed to the network a larger image? Can we feed an image with more channels? Why?
6. Is there any form of regularization?

1.No, there are non linear activations (ReLU)
2. Classification since there's a softmax layer at the end
3. Input size = (None, 64, 64, 3) // Output size = (None, 5)
4. No because it has (3) dense (fully connected) layers
5. - Can we feed the network with a larger image? Yes independently from the input size, the GAP layer has output with 24 neurons (number of channels)
5. - Can we feed the network with more channels? No, because the structure of the network cannot process it. It would need an additional preprocessing layer to modify the correct number of channels.
6. Yes, Dropout and GAP layer

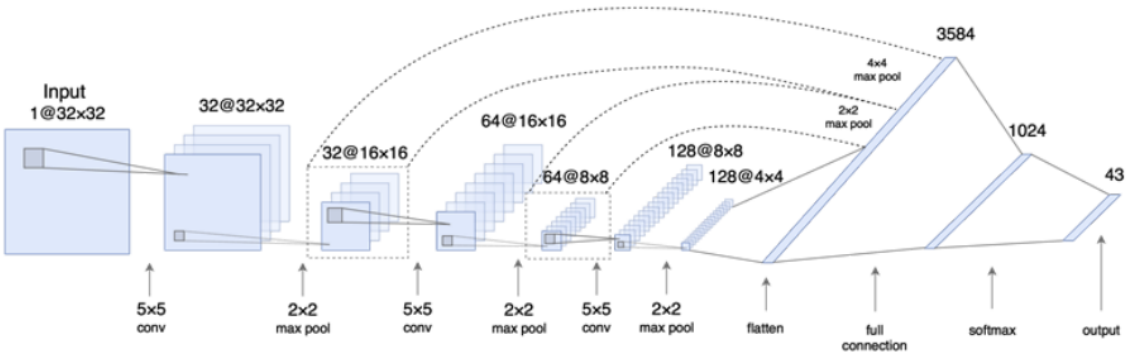
```
import keras
from keras.utils  import np_utils
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Flatten, Conv2D,
from keras.layers import MaxPooling2D, BatchNormalization

pool_size          = (2,2) # size of pooling area for max pooling
nb_filters         = 64

model = Sequential()

model.add(Conv2D(nb_filters, (11,11), input_shape=(64, 64, 3), padding = "same"))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = pool_size))
model.add(Conv2D(nb_filters*2, (5,5), padding = "same")) #2nd conv Layer starts
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = pool_size))
model.add(Conv2D(nb_filters*4, (3,3), padding = "same")) #3rd conv Layer starts
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = pool_size))
model.add(Conv2D(nb_filters, (1,1), padding = "same"))   #4th conv Layer starts
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = pool_size))
model.add(Flatten())
model.add(Dense(64))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(21))
model.add(Activation('softmax'))

model.summary()
```

1. Is this a classification or a regression network? How many output?
2. The following is the output of the model.summary() command. A few numbers have been replaced by dots. Please, fill all the numbers in and in particular indicate which computation you are performing to get to each number. Note: the first dimension is the batch size, and as such this is set to None.
3. How big is the receptive field of the pixel at the center of the feature map after the first convolutional layer? How about the pixel at the center of the second convolutional layer?
4. This network has relatively large filters in the first layers. Can we replace that layer with others to reduce the number of parameters and at the same time preserve (or increase) the receptive field?

Answers:

1. Multi-class classification with 21 output
2. [compute model parameters]
3. Assuming stride=1, the receptive field after the first layer is 11x11 and after the second conv layer is 20x20 (1-> conv2d -> 5x5 -> maxpool -> 10x10 -> conv2d -> 20x20)
4. I assume we can replace only the first layer, which has a filter of size 11x11. We can replace the first conv layer and place 3 blocks of conv 5x5 + maxpool 3x3. in this way we reduced the number of parameters (in total we have 64 * (5*5*3+1) parameters * 3 conv layers, instead of the 64 * (11*11*3+1) of the initial conv layer). The receptive field after these layers is the same, because after the 3 conv5x5 and maxpool3x3 becomes = 11x11.

Note: other possible answers are possible. We could reduce the filter size to 3x3 so to have less parameters but increase the size of max pooling to 7x7 for example. In fact, max pooling has no parameters but increasing its size, increases the receptive field.




## Describe the network in the IMAGE in terms of the characteristic elements composing it, the rationale behind the architecture, the task it might be supposed to do, the loss function you would use to train it. Justify all your statements.

We have an input of 32x32 with one channel. This could be a grey-scale image. Afterwards we have three convolutional layers, all followed by a max pooling to reduce dimensions. It is obvious that padding is used in the convolutional layers to keep the dimension the same. After these 3 iterations, a flattening layer is applied to convert to a multi-layer perceptron to classify these images. There are 2 dense layers of size 1024 and of 43.This could be a classification network of 43 classes since the output is of size 43. I would use a categorical cross entropy loss function since it is the most common choice for classification.

Additionally skip connections are used to better exploit local features in addition to the global ones. Objective of the net: classification of the object in one of 43 possible classes

## Enumerate the building blocks of the networks as if you were going to implement it in Keras and for each of them tell the number of parameters providing a short description on how you compute them, e.g., 3x5x5=45 and not just 45.

<u>Keras</u>
input_layer = tfkl.Input(shape = (32,32,1))

conv1 = tfkl.Conv2D(filters = 32, kernel = (5,5), strides = (1,1), padding = ‘same’, activation = ‘relu’)(input_layer)
pool1 = tflk.MaxPooling2D(pool_size = (2,2))(conv1)

conv2 = tfkl.Conv2D(filters = 32, kernel = (5,5), strides = (1,1), padding = ‘same’, activation = ‘relu’)(pool1)
pooling2 = tflk.MaxPooling2D(pool_size = (2,2))(conv2)

conv3 = tfkl.Conv2D(filters = 32, kernel = (5,5), strides = (1,1), padding = ‘same’, activation = ‘relu’)(pool2)
pool3 = tflk.MaxPooling2D(pool_size = (2,2))(conv3)

```
# skip connections of the outputs of the max pool layers, requires to have all the same size
concat = tfkl.Concatenate() ( [pool1, pool2, pool3] )

flattening_layer = tfkl.Flatten(concat)

dense1 = tfkl.Dense(units=1024, activation='relu')(flattening_layer)

output_layer = tfkl.Dense(units = 43, activation='softmax')
```

Summary

1. input of 1 channel and 32x32 image. (32, 32, 1). It has 0 parameters
2. conv2d_1 (None, 32, 32, 32) has 32x(5x5x1)+32 = 832 parameters (with padding since the dimensions stay 32x32)
3. max_pooling2d_1 (None, 16, 16, 32) has zero parameters
4. conv2d_2 (None, 16, 16, 64) has 64x(5x5x32) + 64 = 51.264 parameters (with padding since dimensions stay 16x16)
5. max_pooling2d_2 (None, 8, 8, 64) has zero parameters
6. conv2d_3 (None, 8, 8, 128) has 128x(5x5x64) + 128 = 204.928 parameters (with padding)
7. max_pooling2d_3 (None, 4, 4, 128) has no parameters
8. Concatenate has no paramaters
9. flatten_1 (None, 3584) has no parameters ->(128*4*4 + 64*4*4 +32*4*4)= 2048+1024+512= 3584
    - *Why do they concatenate the 2x2 and 4x4 max pool after the flattening? What is its goal? These are skip connections that may improve the performance of the model, depending on its application.*
10. dense_1 (None, 1024) has 1024x3584+1024= 3671040 parameters
11. dense_2 (None, 43) has 1024x43+43 = 44.075 parameters

## How to compute the last parameters from flatten to the end?

Flatten has no parameters
Softmax neither
The rest of the network is the Fully Connected layer which is, well, fully connected. Each neuron in the layer is connected to each neuron in the previous layer plus a weight for the bias so in total you have for each layer
**Weights = $N_{neurons}$ * ($N_{previous\ neurons}$ + 1)**

## Illustrate the major differences between a classification and an object detection network, and what are the major advantages of these latter over baselines built upon a CNN for classification. Consider R-CNN as a reference for an object detection network.

**Classification network:**

- Given a fixed set of categories and an output, assign the image a single (most probable) class

- The architecture consists of a CNN (feature extractor) + FC (classifier) head that has cross entropy as a loss function and a softmax at the end to output posterior probabilities for each of the given classes.
- It therefore outputs an array of numbers that describe the probability of finding the classes in a given image ⇒ The most probable class will then be the predicted class

**Object detection network:**

- Given a fixed set of categories and an input image which contains an unknown and varying number of instances, draws a bounding box on each object instance
- The training set therefore consists of labelled images with bounding boxes for each object in the image (together with their respective label)
- Each image therefore might require a different number of outputs depending on the number of objects detected ⇒ Big difference to the standard classification network that always outputs the same size vector

**Major advantages of these latter over baselines built upon a CNN for classification:**

- The baseline built upon a CNN for classification relies on a sliding window
  - Having defined a certain patch size, this patch is slided over the image and the classification network classifies only the content of that given patch ⇒ Maybe use a threshold s.t. only if you have a high probability you output a class
  - This is for obvious reasons extremely inefficient since it doesn't reuse features that are shared among overlapping patches and therefore the convolutions are calculated multiple times. A further disadvantage is the fact that to detect, e.g. cars, at different distances you would need the network to run several times with differently sized patches
  - The only advantage is the fact that you don’t have to retrain the classification network
- R-CNN
  - Uses a region proposal algorithm that proposes possible candidate bounding boxes. This is an external algorithm that is not a NN, but rather looks at things like edges, corners, etc. in the image. It is also noted that this algorithm has high recall and low precision ⇒ just outputs many bounding boxes
  - R-CNN runs this RegionProps algorithm and is returned with some thousand candidate bounding boxes. It now reshapes the bounding box candidates (because they have to have the same size for the FC layer that classifies) and drives the resized region proposals through a CNN and then through the FC classifier, that outputs the class probabilities. On top of that there is also a regressor that improves the region proposal location ⇒ Two different losses

## Describe what are the major changes introduced by Fast R-CNN and Faster R-CNN with respect to the baseline R-CNN.

Fast R-CNN

- Instead of cropping the regions on the image and passing these cropped images the network, rather project the region proposals from the input image over the last convolutional layer and

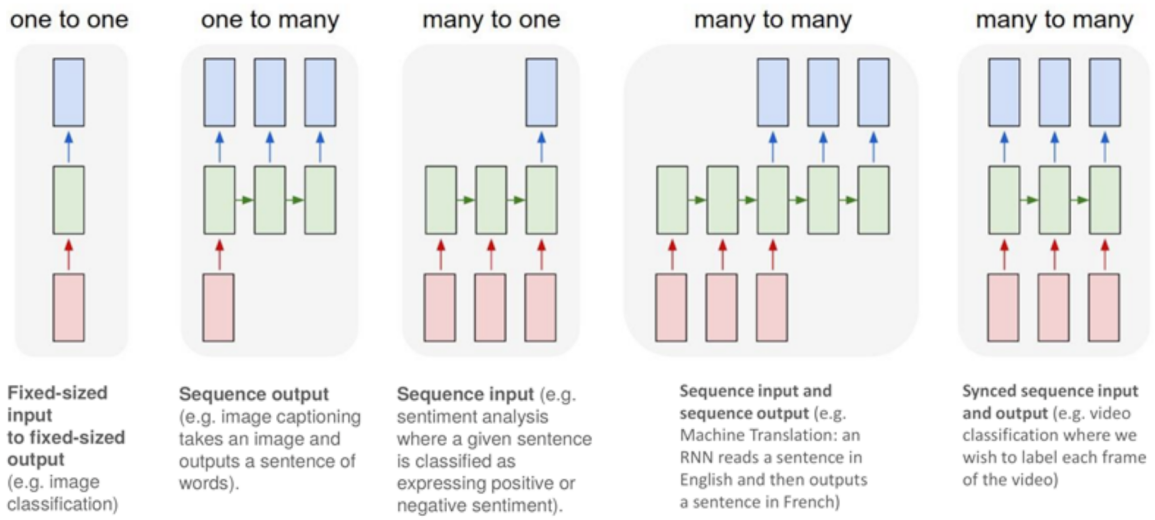
then use a ROI pooling layer to get to a fixed size ⇒ (+) CNN runs only once ⇒ reuses the convolutional computation

- (+) Another advantage is that Fast R-CNN can be trained in an end-to-end manner ⇒ Conv. part only executed once ⇒ You can backpropagate the loss through the entire network ⇒ MUCH faster

Faster R-CNN

- The main idea of Faster R-CNN is to move the region proposal algorithm inside the network and extract the region proposals straight from the convolutional features ⇒ Don't look at the image, but at the feature map directly
- The region proposal network associates to each spatial location k anchor boxes (ROI having different scales and ratios) and estimates an objectiveness score for each anchor ⇒ It therefore also has a classification and a regression head itself to find and adjust the best region proposals
- This helps Faster R-CNN to focus on the most promising regions, which makes it much faster than Fast R-CNN

## For each of the models in the IMAGE below provide its description and make an example of it.




1) classification problem, where the model tries to assign a class to the input image from a vector of options. Ex: first challenge.
2) The model tries to "reprocess" the input in a more complex way. The LSTM structure makes the model more consistent. Ex: definition of words in a dictionary.
3) The model tries to condensate a complex input into a simpler one. Ex: mixing multiple images into one.
4) LSTM model that gives a numerous output from a numerous input. Before doing so you collect all input and analyse it together. Ex: correcting a text in a more formal way.
5) Sync Many-toM-Many, it returns numerous output from a numerous input without collecting all input and analysing it together.

# EXERCISE (09-02-2022)

```
import tensorflow as tf
tfk = tf.keras
tfkl = tf.keras.layers

input_shape = (128,256,3)

input_layer = tfkl.Input(shape=input_shape, name='input')

conv1 = tfkl.Conv2D(filters = 16, kernel_size = (5,5), strides=(1,1), padding='same', acti
vation='relu', name='conv1')(input_layer)
mp1 = tfkl.MaxPooling2D(name='mp1')(conv1)

conv2 = tfkl.Conv2D(filters = 32,  kernel_size = (3,3), strides=(1,1), padding='same', acti
vation='relu', name='conv2')(mp1)
mp2 = tfkl.MaxPooling2D(name='mp2')(conv2)
dropout = tfkl.Dropout(0.3)(mp2)

conv3 = tfkl.Conv2D(filters = 128,  kernel_size = (1,1), strides=(1,1), padding='same', ac
tivation='relu', name='conv3')(dropout)
batchNorm = tfkl.BatchNormalization(name='batchNorm')(conv3) #this normalizes eac
h slice of the volume independently

convt1 = tfkl.Conv2DTranspose(filters = 32, kernel_size = (3,3), strides=(2,2), padding
='same', activation='relu', name='convt1')(batchNorm)
convt2 = tfkl.Conv2DTranspose(filters = 16, kernel_size = (3,3), strides=(2,2), padding
='same', activation='relu', name='convt2')(convt1)
output_layer = tfkl.Conv2DTranspose(filters = 3, kernel_size =(1,1), strides=(1,1), paddin
g='same', activation='sigmoid', name='output')(convt2)

model = tfk.Model(inputs=input_layer, outputs=output_layer, name='model')
# Consider now the execution of the following command
model.summary()
```

**Solution (written by professor in a mail):**

Layer (type) Output Shape Param #
==========
input (InputLayer) (None, 128, 256, 3) 0
conv1 (Conv2D) (None, 128, 256, 16) 16*5*5*3+16=1216
mp1 (MaxPooling2D) (None, 64, 128, 16) 0
conv2 (Conv2D) (None, 64, 128, 32) 32*3*3*16+32=4640
mp2 (MaxPooling2D) (None, 32, 64, 32) 0
dropout_8 (Dropout) (None, 32, 64, 32) 0
conv3 (Conv2D) (None, 32, 64, 128)  128*1*1*32+128=4224
batchNorm (None, 32, 64, 128) 128*4=512 batchNorm ha 4 parametri per layer, ma solo 2 sono trainable!
convt1 (Conv2DTranspose) (None, 64, 128, 32) 32*3*3*128+32=36896
convt2 (Conv2DTranspose) (None, 128, 256, 16) 16*3*3*32+16=4624
output (Conv2DTranspose) (None, 128, 256, 3) 3*1*1*16+3=51
==========
Total params: 52,163
Trainable params: 51,907

Non-trainable params: 256
21

## What is the receptive field of a pixel at the centre of the output of the batchnorm layer?

Solution: 12x12

Start from the end (in this case BatchNorm layer) (init) -> 1
Conv2D(1x1) 1 +(1-1) -> 1
MaxPool  1*2 -> 2
Conv2D(3x3): 2 + 3 -1 -> 4
MaxPool: 4*2 -> 8
Conv2D(5x5) -> 12

Consider the network is now compiled as follows,

model.compile(optimizer=tfk.optimizers.Adam(), loss='categorical')

and that you have a large training set of images from a surveillance camera, with whate ver annotation needed.

Select all the tasks for which a neural network expert (like you are expected to be) woul d train and use this network for:
(1 punto)

- [ ] determining how many persons appears in the image
- [ ] determining the image regions covered by cars, by persons and anything else
- [ ] determining whether it is winter or summer, whether it is raining or not
- [ ] determining which pixels contain a human, a car and what is the temperature in there
- [ ] determining where cars are parked in the image
- [ ] determining where there are empty parking slots
- [ ] tracking persons, cars and buses moving in the scene
- [ ] determining the locations of: each person, each dog, each car in the scene.
- [ ] determining all the pixels covered by road, sky or others

**Solution** (written by professor in a mail, found on telegram group)

The only correct questions are:

- determining the image regions covered by cars, by persons and anything else,
- determining all the pixels covered by road, sky or others

In fact, the network outputs a tensor having the same size as the original image and it's trained to minimise crossentropy. Therefore, the network is a segmentation network that any neural network expert like you are expected to be, would train that to solve a three-class image segmentation problem.

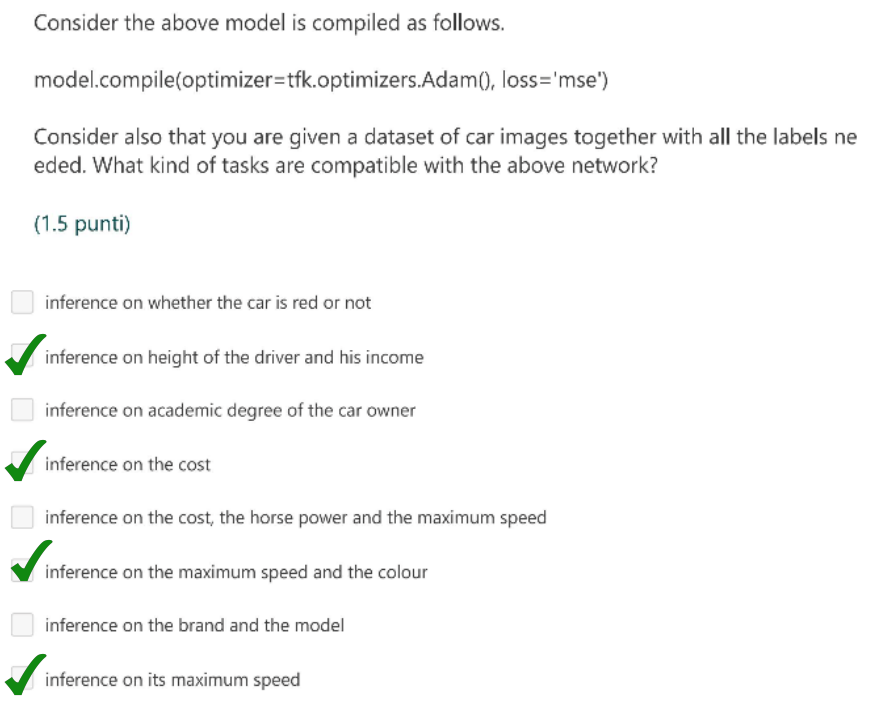
- ☐ determining how many persons appears in the image
  Wrong: that's a regression problem on the image
- ☑ determining the image regions covered by cars, by persons and anything else
  that's correct
- ☐ determining whether it is winter or summer, whether it is raining or not
  Wrong: that's a three-class classification problem
- ☐ determining which pixels contain a human, a car and what is the temperature in there
  Wrong: here the output seems compatible with the network architecture but the temperature is not a categorical variable, so that's not the right network for this task
- ☐ determining where cars are parked in the image
  Wrong: this can be either interpreted as a two-class classification problem or a localization problem, which are not segmentation problems
- ☐ determining where there are empty parking slots
  Wrong: this can be either interpreted as a two-class classification problem or a localization problem, which are not segmentation problems
- ☐ tracking persons, cars and buses moving in the scene
  Wrong: this requires drawing bounding boxes or in any case identifying instances, which is not what image segmentation does
- ☐ determining the locations of: each person, each dog, each car in the scene.
  Wrong: this requires drawing bounding boxes or in any case identifying instances, which is not what image segmentation does
- ☑ determining all the pixels covered by road, sky or others
  that's correct.

# Exercice (17-01-22)

```
tfk = tf.keras
tfkl = tf.keras.layers

input_shape = (256, 256, 3);

# Build the neural network layer by layer
input_layer = tfkl.Input(shape=input_shape, name='Input')

conv1 = tfkl.Conv2D(filters=8, kernel_size=(5, 5), strides = (1, 1),padding = 'same',activ
ation = 'relu', name='conv1')(input_layer)
pool1 = tfkl.MaxPooling2D(pool_size = (2, 2), name='mp1')(conv1)

conv2 = tfkl.Conv2D(filters=32,kernel_size=(5, 5),strides = (2, 2),padding = 'same',activ
ation = 'relu', name='conv2')(pool1)
pool2 = tfkl.MaxPooling2D(pool_size = (2, 2), name='mp2')(conv2)

btchNorm = tfkl.BatchNormalization(name='batchNorm')(pool2) #this normalizes each
slice of the volume independently

conv3 = tfkl.Conv2D(filters=64,kernel_size=(3, 3),strides = (2, 2),padding = 'same',activ
ation = 'relu', name='conv3')(btchNorm)
pool3 = tfkl.MaxPooling2D(pool_size = (2, 2), name='mp3')(conv3)

flattening_layer = tfkl.Flatten(name='Flatten')(pool3)

dropout1 = tfkl.Dropout(0.3)(flattening_layer)
dense1 = tfkl.Dense(units=64, name='Dense1', activation='relu')(dropout1)

dropout2 = tfkl.Dropout(0.3)(dense1)
dense2 = tfkl.Dense(units=32, name='Dense2', activation='relu')(dropout2)

output_layer = tfkl.Dense(units=2, activation='linear', name='Output')(dense2)

# Connect input and output
model = tfk.Model(inputs=input_layer, outputs=output_layer, name='model')

# Consider now the execution of the following command
model.summary()
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| Input (InputLayer) | (None, [256 256 3]) | [0] |
| conv1 (Conv2D) | (None, [256 256 8]) | [8(5x5x3+1)] |
| mp1 (MaxPooling2D) | (None, [128 128 8]) | [0] |
| conv2 (Conv2D) | (None, [64 64 32]) | [32(5x5x8+1)] |
| mp2 (MaxPooling2D) | (None, [32 32 32]) | [0] |
| batchNorm | (None, [32 32 32]) | [32*4] |
| conv3 (Conv2D) | (None, [16 16 64]) | [64(3x3x32+1)] |
| mp3 (MaxPooling2D) | (None, [8 8 64]) | [0] |

| | | |
|---|---|---|
| Flatten (Flatten) | (None, [8*8*64]) | [0] |
| dropout (Dropout) | (None, [8*8*64]) | [0] |
| Dense1 (Dense) | (None, [64]) | [(8*8*64+1)*64] |
| dropout (Dropout) | (None, [64]) | [0] |
| Dense2 (Dense) | (None, [32]) | [(64+1)*32] |
| Output (Dense) | (None, [2]) | [(32+1)*2] |




Explanation:
The final output layer is a dense layer with 2 neurons and **linear activation function**. It solves a **regression task over 2 (at most) values**.

- Maximum speed is a real continuous value and therefore it is inferable from the images
- The colour can be mapped from a RGB encoding to a single value. For example, if the colour is 8-bit encoded, it can be mapped on a value with a range from 0 to 16 million. Therefore a linear activation can detect the colour, so regression over colours is feasible.
- The cost of the car can be inferred because it is a real value.
- The height of the driver and his income are continuous values as well, so they can be inferred, assuming the labels are provided for both values for every image of the cars.

- The other values (car brand, car model, academic degree) cannot be transformed into a continuous value, so regression is not feasible, and a classification network (with softmax output) is required.

Side note: the neurons in the final layer are two, so at most 2 values can be inferred. This means that the network can also infer a single value, and keep the result of the other neuron always zero, so that only the relevant value is taken into account when doing regression.

Assume you want to develop a system to automatically read the hours from your old w all clock as in the picture.

Now, assume you place a webcam in front of your wall clock, gather a lot of images wit h their acquisition time and you are ready to start training your CNN.

You don't have many images however, and you want your network to be robust to diff erent positioning of the webcam in front of the wall clock, different weather / light con ditions...

What kind of data augmentation **should be avoided** during training?

(1.5 punti)

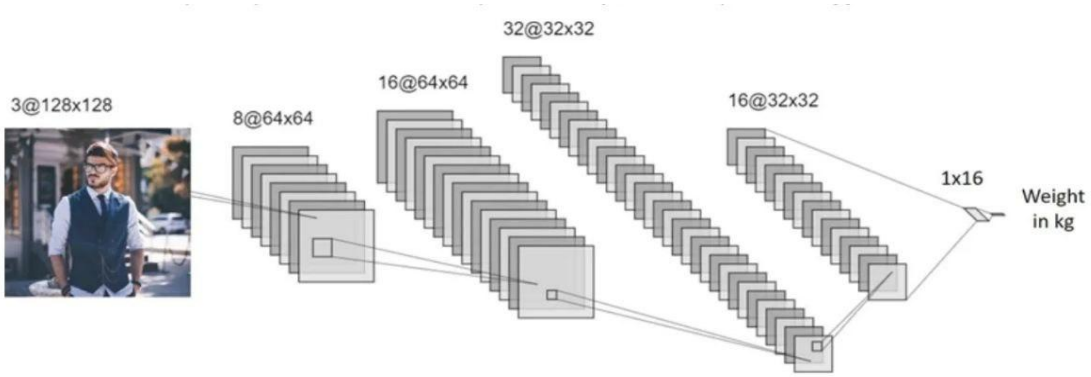
- [ ] translation
- [x] scaling of y axis
- [x] rotation
- [x] vertical flip
- [ ] noise addition
- [x] scaling of x axis
- [x] horizontal flip
- [ ] change in brightness
- [ ] image scaling

---

- [ ] blur

Rotation, vertical flip and horizontal flip, scaling of x and y dimension should be avoided because it may result in an image of a clock that represents a different time than the true one. For example, it may happen that the small hand of the clock is mistaken for the bigger one.
*In the solution, I assume that image scaling refers to a homogeneous resizing of the image along the x and y axis, so it affects the overall shape and not the time reading.*
The other transformations can be applied for augmentation without changing the time representation in the clock.

# EXERCISE (08-07-21)




## What task is addressing the network illustrated above? What loss would you use for training?

The network is trying to predict the weight of the object in the image fed in input. To do so it uses a stack of 2 convolutional layers followed by a max pooling layer and another conv2d layer. Then it uses a global average pooling to go in a dense layer to output the result. Being a regression problem I would use a mean square error loss function.

## How would you modify the network to also predict whether the human is an engineer or not? How would you train the network in this latter case?

The idea is to propose a second FC parallel to the first one and just change the output layer using a softmax and we can retrain the network with a binary cross entropy loss function. At the end you'll have a pair as output telling the weight in the above network and if being an engineer in the new one.

- Since the question says "also", it means that we should design a network that can predict both: weight (regression) and is_engineer (binary classification)
- In order to do both, we could add two separate heads to the shared CNN ⇒ One head uses the regression loss (MSE) to predict the weight, the other used binary categorical cross entropy to predict the class
- Since we however need a single scalar for our loss in order to compute a gradient of a loss function with respect to the network parameters, we must minimise the multitask loss to merge the two separate losses ⇒ This is a convex combination of the two losses, that is controlled by a hyper parameter alpha
- Alpha should be tuned with great care since it is no ordinary hyperparameter but one that actually changes the rules of the game ⇒ It directly influences the loss definition ⇒ Difficult to tune ⇒ Should be tuned using CV looking at some other loss, since the loss value of different alpha values by themselves might be meaningless

## How would you train the network in this latter case?

Since we already have the trained model provided to us, I would use transfer learning to make use of the existing CNN. The FC heads should then be trained using a training set that has both the weight and the binary class

Banale, una possibilità è tenere una rete fissa trainando l'altra, due volte, infine tutto libero per tuning finale.

# GAN

## What are GANs?

GAN is a kind of model where two networks are put against each other as adversaries. One network (Generator G) creates fake images starting from a dataset of real images, the other network (Discriminator D) has to determine if the image generated by the Generator is real or fake (i.e. generated).
The training happens cyclically, first we train G, then D, then G again, and so on. Initially, G is poorly trained and so D detects its images as fake, but the more G trains, the less reliable D will be. At some point, D will be totally unreliable, meaning that G can generate perfectly realistic images, so D will be discarded and we keep G as a great generator.

**Alternative Solution:**
Generative Adversarial Networks (GANs) are a type of deep learning model designed to generate new, previously unseen data that is similar to a training dataset. They consist of two main components: a generator and a discriminator. The generator creates new data samples, and the discriminator evaluates them to determine whether they are real or fake. The two models are trained in competition with each other, with the generator trying to produce data that can fool the discriminator, and the discriminator trying to correctly identify the real data from the fake data. GANs can be used for a variety of tasks, such as image and video synthesis, text-to-speech, and more.

# Autoencoders

## What are sparse neural auto-encoders?

Sparse Autoencoders are a variant of autoencoder neural networks that are designed to learn sparse (compact) representations of the inputs, in which the majority of the feature representations are zero or close to zero. The goal of a sparse autoencoder is to learn a compact, efficient and interpretable representation of the input data that captures the most important features.

The main difference between a regular autoencoder and a sparse autoencoder is the objective function that they optimise. A regular autoencoder tries to minimise the reconstruction error between the input and the output, while a sparse autoencoder also tries to minimise the sparsity of the activations of the hidden layer. This is achieved by adding a sparsity constraint to the objective function, that can be implemented by adding a sparsity regularisation term to the loss function, acting as a lasso regularisation term (L1 norm). The sparsity term prevents co-adaptation of the neurons, so that the neurons learn meaningful representations of the features, without forcing a specific path in the network.

## How data representation is learned via deep autoencoders?

In a deep autoencoder, data representation is learned through a process called encoding and decoding. An autoencoder is an unsupervised deep learning model that is trained to reconstruct its input. The model consists of two main parts: an encoder and a decoder.

The encoder takes the input data and maps it to a lower-dimensional representation, called the bottleneck or latent representation. This is done by passing the input through multiple layers of non-linear transformations, such as fully connected or convolutional layers. The encoder aims to learn a compressed, abstract representation of the input data that captures the most important features.

The decoder then takes the bottleneck representation and maps it back to the original input space. This is done by passing the bottleneck representation through multiple layers of non-linear transformations, such as fully connected or transposed convolutional layers. The decoder aims to reconstruct the input data as accurately as possible.

During training, the autoencoder is trained to minimise the reconstruction error between the input and the output, this will force the encoder to learn a representation that can be used to reconstruct the input.

Deep autoencoders can be used for a variety of tasks, such as dimensionality reduction, feature extraction, anomaly detection and generative modelling.

In summary, the main process of data representation learning in deep autoencoders is encoding and decoding, which is performed by passing the input data through multiple layers of non-linear transformations, with the goal of learning a compressed, abstract representation of the input data that captures the most important features and can be used to reconstruct the input.

## What is the relationship we learn in a neural autoencoder? Why do we do it?

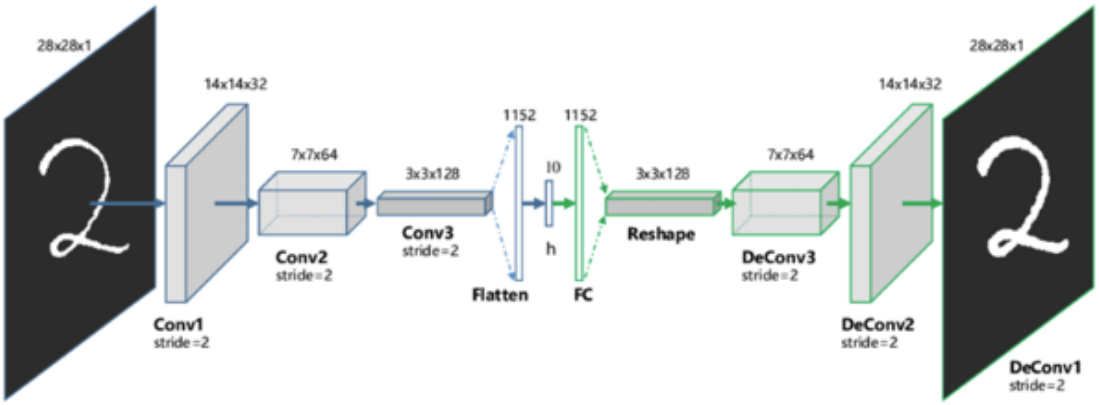
The relationship to be learned is the Identity Mapping. The Identity relationship is the correspondence between input and output maps. We design autoencoders so that the output of the decoder stage resembles as close as possible the input in the encoding stage. The reason behind this is that it may be necessary in some situations to learn to represent latent representations that are as accurate as possible, with respect to the input of the network.
Some use cases of the neural autoencoders are:

- Data compression: we are able to generate new realistic samples from the latent representation, by randomly sampling it. The latent representation is of much lower dimensionality than the input dimension.
- Unsupervised feature learning: by training on a sparse autoencoder, it can learn useful features that may be used in other tasks such as clustering.
- Anomaly detection: we are able to recreate non-anomalous samples, so that we can understand if new test samples are anomalous or realistic, depending on whether they could be produced by the decoder stage.

## How could we size the embedding of a neural autoencoder?

The size of the embedding in a neural autoencoder can be determined through a process of experimentation and evaluation. A general rule of thumb is to start with a smaller size and gradually increase it, evaluating the performance of the model at each step. The performance can be measured using metrics such as reconstruction error, classification accuracy or clustering performance. Additionally, you can also consider the computational cost and memory requirements. Another approach is to use the rule of thumb that the size of the embedded layer should be between the input and output layer size. Ultimately, the optimal size will depend on the specific task and dataset you are working with.

# EXERCISE




9

Detail the building blocks in each layer of the above network. Please indicate
- what kind of block is
- what is its size
- what is the overall number of parameters, together with a short description on how you compute them, e.g., 3x5x5=45 (not just 45!). Consider convolutions having a spatial 3x3 extent.

12

| Layer (type) | Output Shape | Param # |
|---|---|---|
| Input (InputLayer) | (None, [28 28 1]) | [0] |
| conv1 (Conv2D) | (None, [14 14 32]) | [32(3x3x1+1)] |
| conv2 (Conv2D) | (None, [7 7 64]) | [64(3x3x32+1)] |
| conv3 (Conv2D) | (None, [3 3 128]) | [128(3x3x64+1)] |
| flatten1 (Flatten) | (None, [1152]) | [0] |
| autoencoder(Dense) | (None,[10]) | [10* (1152 +1)] |
| dense1 (Dense) | (None,1152) | [1152(10+1)] |

| | | |
|---|---|---|
| reshape(Reshape) | (None,[3 3 128]) | [0] |
| deconv3 (DeConv2D) | (None, [7 7 64]) | [64(3x3x128+1)] |
| deconv2 (DeConv2D) | (None, [14 14 32]) | [32(3x3x64+1)] |
| deconv1 (DeConv2D) | (None, [28 28 1]) | [1(3x3x32+1)] |

## This network is called denoising autoencoder and it has the goal of cleaning or restoring a damaged image. How would you train it? What kind of loss function would you use?

A denoising autoencoder (DAE) is a type of autoencoder that is trained to reconstruct the original data from a corrupted version of the data. The goal of a DAE is to learn a robust representation of the data that is able to tolerate noise and other forms of corruption.

To train a denoising autoencoder, you would first need to create a corrupted version of your training data. This can be done by adding noise to the data, such as Gaussian noise, or by masking out a certain percentage of the data, such as in dropout. Then, the corrupted data is used as the input to the encoder, and the original, uncorrupted data is used as the target for the decoder.

As for the loss function, the most common loss function used for training autoencoders is the mean squared error (MSE) or the binary cross-entropy (BCE) between the original data and the reconstructed data. The MSE loss compares the element-wise squared difference between the original data and the reconstructed data, while BCE loss compares the log probability of the original data and the reconstructed data.

During the training, the model's aim is to minimise the reconstruction error between the original data and the reconstructed data, this will force the encoder to learn a robust representation of the data that is able to tolerate noise and other forms of corruption.

In summary, to train a denoising autoencoder, you would first need to create a corrupted version of your training data, and then use it as the input to the encoder, the original, uncorrupted data is used as the target for the decoder, and the most common loss function used for training autoencoders is the mean squared error (MSE) or the binary cross-entropy (BCE) between the original data and the reconstructed data.

# RNN

## How data representation is learned via recurrent neural networks?

In a recurrent neural network (RNN), data representation is learned through a process called recurrence. RNNs are designed to process sequential data, such as time series, speech, or text, and are able to capture the temporal dependencies within the data by maintaining an internal memory, called the hidden state. The hidden state is updated at each time step based on the current input and the previous hidden state, and it is used to generate the output.

In the case of text, the input to the RNN is typically a sequence of words, each represented as a word embedding. These embeddings are passed through the RNN one at a time, and at each time step, the RNN updates its hidden state based on the current input and the previous hidden state. The updated hidden state is then used to generate the output, which can be a probability distribution over a set of words (for language modelling) or a probability distribution over a set of labels (for text classification).

RNNs can be unrolled to multiple time steps, this means that the same set of weights are used for each time step, and the hidden state is updated multiple times. This allows the RNN to capture long-term dependencies in the input data, as the hidden state can maintain information from previous time steps.

In summary, the main process of data representation learning in RNNs is the recurrence operation, which is performed multiple times using the same set of weights, with the goal of capturing the temporal dependencies within the input data and maintaining an internal hidden state that can be used to generate the output.

## What is Vanishing Gradient in RNN? How is it fixed in LSTM? Why does it fix the problem?

The vanishing gradient in RNN is related to the fact that, when computing the gradient during backpropagation, we have to multiply several times for the derivative of the activation function and the weights. In case we're using activation function as the sigmoid or the tanh we have that the derivative is always less than 1 (max 0.25 for the sigmoid and max 1 for the tanh). So, by multiplying several times numbers <1 the gradient converges to 0. To solve this problem, we have to use a linear or ReLU activation function. Also, to be sure that we don't have a vanishing (or exploding) gradient due to the weights, we have to fix the weights to be equal to 1. LSTM implements a Constant Error Carousel (CEC) with linear activation and weights equal to 1 and, to solve the problem of the RNN that only "accumulates" input, it implements a gating mechanism to decide whenever to write or read from the memory cell.

**Alternative solution:**

Vanishing Gradient in RNN:

- Back propagation through time unrolls the network according to the length of the series
- The vanishing gradient problem in RNNs is caused by the fact that the gradients in the network, which are used to update the weights during training, become very small as they are backpropagated through time.

- This can happen because the gradients are multiplied by the weights in the network multiple times as they are backpropagated through the time steps, and if the weights have values that are less than 1, the gradients will become smaller and smaller with each multiplication.
- This can make it difficult for the network to learn from the data, especially for long sequences.

How it's fixed in LSTMs

- LSTMs fix this problem through the constant error carrousel, i.e. a memory cell
- Recursion is the most critical part of LSTM with respect to the vanishing gradient as it generates a multiplication of gradient for each step in the past

## Why does LSTM fix the vanishing Gradient?

- The solution is in the CEC (Constant Error Carousel), i.e., a memory cell with a recursion weight fixed to 1 (no issue with exploding or vanishing gradient) and a linear activation function (no issue with exploding or vanishing gradient) ⇒ can retain information for long periods of time
- The memory cell is a small neural network that is connected to the input, output, and other LSTM cells in the network. The connections between the memory cell and other parts of the LSTM are controlled by gates, which are themselves small neural networks. These gates act as filters, allowing some information to pass through while blocking other information. This is done with the use of activation functions such as sigmoid and tanh.
- The gates in LSTMs can also help to prevent gradients from becoming too small or too large. For example, the forget gate is responsible for deciding how much of the previous information to retain in the memory cell, while the input gate is responsible for deciding how much new information to let in. By controlling the flow of information through these gates, LSTMs can ensure that the gradients flowing through the network remain at a healthy level and prevent the vanishing gradient problem.

## Do I really need a recurrent neural network to process a stream of input data? Why can't I use a standard feed forward architecture?

Standard Feed Forward Networks are memoryless, although RNN-s are memory based models. RNN are well-suited for processing sequential data, such as a stream of input data, because they have the ability to maintain a hidden state that can be used to process information from previous time steps. This allows RNNs to make use of context from previous time steps when processing the current time step.
Standard feedforward architectures, on the other hand, do not have the ability to maintain a hidden state, and therefore cannot make use of context from previous time steps. This can make them less well-suited for tasks that involve sequential data.

**Alternative solution**
Standard feed forward neural networks can indeed process streams of input data, but they need to know which is the width of the window of dependency. Knowing that one can use delay taps to concatenate the needed inputs or in alternative define non linear hidden layers in order to unroll time and try to treat data as if it was static. This is a valid approach, but in practical cases the window of dependency is not known a priori or it is changes during time, so a better approach is to use dynamical systems such as RNNs that manage to capture temporal dependencies within the data with recurrent connections that allow hidden states to maintain information from previous time steps.

## Classic recurrent neural networks suffer the vanishing gradient problem. What is it and what is it due to? Does it only affect recurrent networks?

The vanishing gradient problem occurs in several types of neural networks, not only on RNN but also on deep forward networks. It is present in those networks that use backpropagation to update their weights. It happens when the gradient of the weight updates become very small, which can make it difficult for the network to learn the training data. The gradient is the product of the derivatives of all the intermediate functions of layers and going back in time this product decreases and gets to zero. RNNs have feedback connections that can amplify the gradients, because of the repeated multiplications of the gradients during backpropagation through time. This leads to very small gradients.

## Why do we need a recurrence mechanism? Isn't the attention mechanism enough?

The attention mechanism can put more weight to certain parts of the input, while maintaining long-term memory of the past and future inputs. The attention mechanism used alone is useless if there is no way of managing memory and handling sequences of data. That's why RNN layers or LSTM cells are important and useful to be used with the attention mechanism, since they provide an efficient way of handling sequences of data inputs, and maintain short-term memory. Without them, the attention layer would be practically useless (unless a Transformer model is employed).

The attention mechanism was built upon the RNN and LSTM layers, but the Transformer model revolutionised this assumption, showing that the Attention layer can perform well, if properly designed, without recurrent layers such as the RNN of LSTM.

*The Transformer architecture is not required for this year's exams.*

## Why do we need an attention mechanism? Isn't the LSTM cell with its recurrent mechanism enough?

The recurrence mechanism proves to be very useful when handling sequences of data. Two different mechanisms have been employed for solving the sequence problem, such as the base recurrent neurons and the LSTM cells, which work differently but solve the same task. The LSTM cells are designed to work with very long sequences of data, and can learn dependencies between data points that are far from each other.

The attention mechanism was invented to focus on specific parts of the input and maintain long-term memory of the inputs. It also overcomes the problem of the vanishing gradient when handling very long sequences of inputs, which was a problem present in the LSTM cells. The attention layer can also handle long-term dependencies between data points. It manages to give more weight to certain parts of the input, such that it employs a more effective way of managing memory of past or future data points.

## What kind of applications can be solved with a Recurrent Neural Network? Make two (2) examples and discuss their characteristics.

In principle, tasks like translation or voice to text can be tackled with RNNs. They both feature input data that are temporally correlated and sequential, which is the basic feature that points us in the direction of RNNs rather than other models. Of course, simple RNNs might prove too weak to satisfactorily perform these tasks, but they would also reasonably perform better than the even more simple feedforward NNs.
Both the tasks I mentioned are of the many2many types, but this does not necessarily need to be the case to choose RNNs. A many2one task such as sentiment analysis input a sequence of characters or words, e.g., a tweet, and classify the sequence into positive or negative sentiment. Input has variable lengths, output is of a fixed type and size.

## The current state of the art in text modelling and prediction is the Transformer. Is it implemented with a Recurrent Neural Network? Does it suffer from the Vanishing Gradient problem? Motivate your answer.

It is not implemented with RNNs since the attention mechanism allows it to elaborate the entire input sequence in parallel. However, it is worth noting that at test time it does perform a form of recurrence on the output since it uses the outputs at the previous steps to produce the next output.
The transformer utilises residual learning both in the encoder and the decoder, which in conjunction with ReLU activations and attention, make it resilient to vanishing gradients.
Of course, residual learning affects the gradient flow through the network since it allows the gradient at the output of a residual block to directly contribute to the gradient at any layer below it, since each of those layers adds something to the output.

## How sentences are embedded in seq2seq modelling?

I do encode each word into a number ranging from 0 to V(size of vocabulary) + 4 (tokens SOS,EOS,PAD,UNK)

**Alternative solution:**
After having performed some sort of word embedding on the input vocabulary and the four special tokens (SOS,EOS,PAD,UNK), we append: EOS to source_sentences, SOS to target_sentences to obtain target_input_sentences and EOS to target_sentences to have target_output_senteces.
Then, the source_sentences are passed through the encoder part of the network.
After having processed the EOS embedding of a sentence in the last cell the h and c of the LSTM are the embedding of the phrase to be passed to the decoder of the network.

## How does the training for seq2seq work?

It's a supervised learning procedure. You have batches of couples made of a source sequence and a target sequence. After encoding, the source sequence is fed to the encoder.

SOS, EOS, PAD, and UNK are special tokens that are often used in natural language processing tasks, particularly in sequence-to-sequence m
like machine translation or text summarization. Here's what each token stands for:

SOS: SOS stands for "Start of Sequence". It is a special token that is added at the beginning of an output sequence in a sequence-to-sequenc
model, indicating the start of the generation process.

EOS: EOS stands for "End of Sequence". It is a special token that is added at the end of an output sequence in a sequence-to-sequence mode

The target sequence is fed to the decoder both as input and as target. Given the encoding, each timestep takes the corresponding element of the target sequence as input, outputs a guess on which element will be next and then uses the same target sequence (in a slightly different encoding) to evaluate its guess and learn.

At inference time, where there is no target sequence, the output guess of each timestep of the decoder is directly fed as input of the next timestep

## What is the difference between a Recurrent Neural Network and a Long Short-Term Memory in terms of training algorithms? (Try to be short and focused!)

RNN and LSTM are both trained using the backpropagation through time algorithm. This is a variant to the normal backpropagation algorithm because it does the unrolling of the RNN neurons and the unrolling of the LSTM cells through time. The computation is overall the same between RNN and LSTM. There is a slight difference in how the two computations are handled.

- RNN requires truncated backpropagation in the sense that the unrolling is very limited in time, so it cannot go “too far in the past”. Therefore the sequences handled support only short term dependencies between data inputs.
- In LSTM cells training the truncated version of backpropagation through time is not always required *(is this true?)*, depending on the length of the data inputs and the distance of their dependencies. Furthermore, the LSTM cells are much more robust to the vanishing gradient problem, making them easier to be trained with the most advanced stochastic gradient descent algorithms.

*Side note: I was not able to find any trustful resource to check whether truncated BPTT is actually always used for LSTM. According to chatGPT, it seems that truncated BPTT is the favoured option, independently from the sequence length and structure, but I’m not able to verify this (and it’s surely not required to know for the exam).*

## When could we prefer the use of Recurrent Neural networks instead of Long Short-Term Memory networks? Why?

If the task at hand does not require the ability to retain long-term context, then RNNs are preferred as they are simpler and more computationally efficient than LSTMs. But if the task requires the ability to retain long-term context, then LSTMs are the preferred choice.

## How could we size the embedding of a RNN? And in the case of LSTM?

When we don't care how much information from previous steps we want to save. Since in RNN, we're obliged to use ReLu, we basically accumulate the input data and as a result, our memory should be

kinda short. but in LSTM we decide also about the amount of prev info(thanks to gates!) as well as having a long memory

## What does the sentence «You shall know a word the company it keeps» by John R. Firth (1957) mean? Why do we mention it in the course and which model uses it? Describe the model

This sentence refers to the Masked Language Modelling and Causal Language Modelling tasks. It means that a word in a text is strongly correlated with the words surrounding it. Namely, a word is dependent by its context (that is, the words around the word). In case of causal language modelling we consider words before it and in masked language modelling we consider words before and after it. We mentioned this quote because we treated the Masked Language Modelling NLP tasks. This concept is being used by the Word2Vec model both in its CBOW and SkipGram version. The model tries to predict a word considering both the words on the right and on the left. Given the words, the algorithm is able to produce the embedding of the missing word given the embedding of the context's words. On the other side, the SkipGram implementation does the opposite task, given a word embedding, it computes the most probable context for the word and outputs it

## What is word embedding and what is it used for?

Word embedding is a technique which aims to offer an efficient representation of words by projecting them from their 1-of-N encoding in a lower-dimensional feature space. Such a space is also continuous and dense, and thus maintains a representation of the similarity of words which can be intuitively represented though vector arithmetics. Word embedding is especially used to allow for more powerful architectures for speech-related tasks.
—

- Put simple: word embeddings are an efficient, comparatively low-dimensional representation of words that retains semantics ⇒ two words that are surrounded by the same words are close in meaning (and therefore in space)
- In more detail: An embedding maps/transforms a word (or phrase) from its original high-dimensional, sparse and semantic-ignoring space (one-hot-encoding of the body of words) to a lower dimensional numerical vector space ⇒ It "embeds" words in a different space
- It's main advantages are:
    - Fights the curse of dimensionality that is inherent to one-hot-encoding by: reducing the dimension (Compression), going from a discrete representation to a continuous representation (Smoothening) and by building a dense representation (Densification)
    - In an word embedding, similar words therefore are close to each other in space ⇒ if you average the encoding of the surrounding words, you get the likelihood of a given word
- What are they used for?
    - They are used for all different types of language processing and can be used as a more powerful input to the usual models, such as seq2seq models.
    - Other applications are: information retrieval, document similarity (Obama speaks to the me

## Is word embedding a supervised or an unsupervised machine learning task? Why?

Word embedding is typically considered an unsupervised machine learning task. This is because the goal of word embedding is to learn a dense, low-dimensional representation of words in a corpus of text, without the use of any labelled data or external supervision. The embeddings are learned based on the patterns and relationships among the words in the corpus. The embeddings can then be used as input to other supervised or unsupervised models, such as language translation, text classification, and sentiment analysis.

–

When people talk about word embeddings like word2vec, they are usually referring to the final product, which is **NOT** the neural network structure that is trained to (in word2vec's case) either predict current word based on context words (CBOW), or the other way around (skip-gram), **BUT** the input-to-hidden layer weight matrix. In that case, even though the network itself is trained in a supervised way, the whole process of word2vec training is seen as an unsupervised learning process. I would personally call it **self-supervised:**

- It's still a neural network that works by back-propagating error
- To calculate error, you need labelled data.
- Word2Vec reads the text and generates labelled data from it.
- Word2Vec's pre-processing part takes this sentence and makes data pairs
- Thus it forms labels and trains the network.

## How is the word embedding loss function defined?

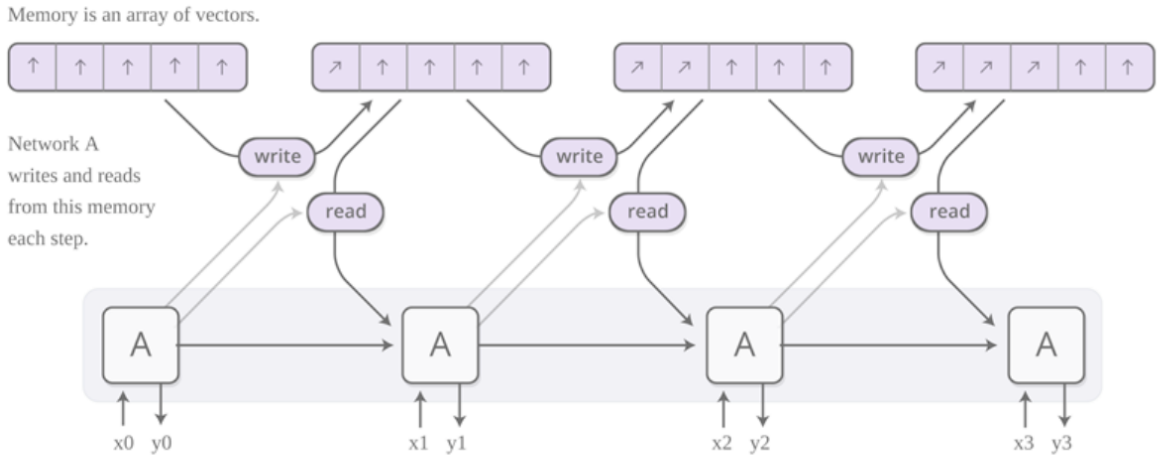
A common architecture for training word embeddings using a neural network is the continuous bag-of-words (CBOW) model. In this model, the input to the network is a context of words and the output is the target word that appears in the middle of the context. The loss function is defined as the difference between the predicted probability distribution of the target word and the true probability distribution of the target word, which is typically defined using a one-hot encoded vector.
The most commonly used loss function for training word embeddings in a neural network is the cross-entropy loss. The cross-entropy loss is defined as:

$$L = -\sum y\_i * \log(\hat{y}\_i)$$

where y_i is the true probability distribution of the target word and ŷ_i is the predicted probability distribution of the target word. The sum is taken over all words in the vocabulary.
Another loss function that is commonly used in word embedding neural networks is the negative sampling loss function.

# Neural Turing Machine

The model in the picture describes the functioning of a Neural Touring Machine from an high level perspective. With reference to this model answer the following.




## How does a Neural Turing Machine work?

Basically, it is a neural network acting as a Turing Machine. In the Turing machine we have a string of symbols and we can perform any computation by a series of operations that read and write in a sequence from this string. What the network does is that it stores information in the memory, reading from it and writing in it. It uses the memory explicitly instead of an encoding of it. So it combines a RNN with an external memory bank.
The NTM is composed of a neural network controller that interacts with an external memory matrix. The controller uses its internal state, as well as input and output, to read and write to the memory matrix, allowing it to perform tasks that require the use of long-term memory.

## How does the WRITE mechanism work in a Neural Turing Machine?

NTM write mechanism allows writing to the external memory of the network. Writing means you select one address and you change the content of that address.
It first reads the content of a memory location using a set of memory-reading weights. These weights are learned during training and are used to interpolate between memory locations. Then NTM uses its network to compute a set of write weights, which are used to update the content of the memory location by interpolating the read and write weights with the current memory content and the new value.

## How does the READ mechanism work in a Neural Turing Machine?

NTM read mechanism allows retrieving information from the external memory of the network. Reading means you select one address and extract the cell and only the cell at a specific address.

The read mechanism in an NTM typically works by first computing a set of read weights to read the contents of the memory by interpolating between different memory locations. The interpolation is done by taking a weighted sum of the memory locations, where the weights are determined by the read weights. The read weights are learned during training and are used to focus the attention of the NTM on specific parts of the memory.
The output of the read mechanism is a vector that contains the contents of the memory that the NTM has read. This vector is then used as input to the NTM's neural network, which can be used to make predictions or perform other computations.

## How is attention used in Neural Turing Machines?

In NTM attention is used to focus the model's attention on specific parts of the external memory, when reading or writing. It is in practice a soft selection but it doesn't mean we ignore the other resources.
In NTM addressing is the problem. We want to learn what to write/read and where. The solution is to read and write everywhere, just to different extents. It is similar to LSTM gates but here the gate is over the entire memory and you decide, from all the cells, in which cell to write or not. Your data will tell you where to write, in order to put attention on specific cells.
There are two types of attention mechanisms:

- Content-based: focus on places that match what you're looking for.
- Location-based: allow relative movement in memory enabling the NTM to loop.

How it works:
Based on the input, we have a softmax on the memory and the output gets changed based on the memory. This softmax is driven by the input, so it's a content-based attention.
Once we have retrieved the most similar area in the memory, we have a location-based mechanism that allows us to move from one area to another.

## What is sequence to sequence modelling? How is the attention mechanism used in sequence to sequence modelling?

Seq2Seq modelling designed to generate an output sequence from an input sequence. It is a classical encoder-decoder architecture. The encoder takes as input a sequence and maps it to a fixed-length context vector, which is passed to the decoder. The decoder takes this vector and generates an output sequence.
The attention mechanism allows the decoder to focus on specific parts of the input sequence when generating the output sequence, by computing a set of attention weights, which are used to interpolate between the different hidden states of the encoder. The encoder maps the input sequence to a set of hidden states, and the decoder uses the attention mechanism to weight these hidden states when generating the output, by taking a weighted sum of the hidden states.

## What are neural Turing machines? How does attention work in this kind of model?

Definition NTM:

- In a standard Turing machine you have a infinite string of symbols and you can perform any computation by a series of operations that reads and writes in a sequence from this string

- We can do the same with a recurrent model that basically stores information in the memory, reads from it and writes in it ⇒ So Neural Turing Machines combine a RNN with an external memory bank
- It uses the memory explicitly instead of an encoding of it
- An NTM is composed of a neural network controller and an external memory matrix. The controller is responsible for reading and writing to the memory matrix, while the memory matrix stores information that the controller can use to make decisions. The controller uses its internal weights to read and write information to the memory matrix and the memory matrix is updated through backpropagation

How does attention work?

- To make use of the NTM you would like to read and write from the memory, but there is a problem ⇒ Read and write are not differentiable operations since, more precisely addressing is the problem ⇒ But we need them to be differentiable in order to learn where to write
- The solution to that problem is to write/read everywhere but to a different extent, a.k.a. attention ⇒ Put attention/focus on s specific subset of your memory
- The idea is to have an Attention mechanism: basically your data will tell you where to write, in order to put your attention on specific cells ⇒ If you want to read from a specific cell, you put attention there
- There are two types of attention:
    - Content-based attention: searches memory and focus on places that match what they’re looking for
    - Location-based attention: allows relative movement in memory enabling the NTM to loop

# Miscellaneous

## Why Deep Learning?

Machine Learning means computers learning from data using algorithms to perform a task without being explicitly programmed. Deep learning uses a complex structure of algorithms modelled on the human brain.
It uses deep neural networks, which are networks with multiple layers, as opposed to shallow networks which only have a single layer. The term "deep" refers to the number of layers in the network. The use of deep neural networks allows for more complex and sophisticated models, which can be used to solve a wide range of problems, such as image and speech recognition, natural language processing, and more.

+ data-driven feature extraction (principal reason for ‘revolution’)

## What is the dying neuron problem and how would you fix it?

Definition

- The dying neuron problem is an issue that can occur in deep neural networks during training, where certain neurons in the network become inactive or "dead" and cease to contribute to the overall computation
- This can happen when the gradients flowing through the network during backpropagation are too small, causing the weights of certain neurons to become very small or zero
- This is specifically associated to the ReLU activation function ⇒ Because the slope of ReLU in the negative input range is also zero, once it becomes dead (i.e., stuck in negative range and giving output 0), it is likely to remain unrecoverable

Solution:

- Leaky ReLU, ELU
- Weight Initialization: The initialization of the weights in a neural network can also affect the gradients. By using a different initialization method, such as He initialization, it's possible to prevent the gradients from becoming too small in the first place

GoogleNet is an example of a network that suffers from the dying neuron problem.

## Example of unsupervised ML and the counterpart in DL?

We say PCA as the unsupervised learning technique in ML and AutoEncoders as the counterpart in DL

## Is data representation, i.e., features, learned via deep learning always better than hand-crafted features? Why?

It is not always the case that data representation learned via deep learning is better than hand-crafted features. The choice between the two depends on the specific task, dataset, and the resources

available. Hand-crafted features are based on domain knowledge and expert understanding of the problem. They are often designed to capture specific, task-relevant information and can be very effective in certain situations.

On the other hand, deep learning models, especially deep convolutional neural networks (CNNs) and transformers, have the ability to learn useful features from data automatically. These models can learn complex, abstract representations of data that are not easily captured by hand-crafted features. They are particularly useful for tasks with large amounts of data and when the data is noisy or has a lot of variations.

In general, it's worth trying both approaches and comparing the results, and choose the one that performs best for the specific task. In some cases, it's also possible to combine both hand-crafted features and deep-learned features, to leverage the strengths of both approaches and improve the performance of the model.

## What is overfitting? What is it due to?

Overfitting is an undesirable phenomenon that occurs when the model predicts too accurately the target of the training data but not for new data: the model has a low generalisation capability. Overfitting can happen due to several reasons, for example when the training data size is too small and does not contain enough data samples to accurately represent all possible input data values, or when the model complexity is high, so it learns the noise within the training data.

## How do you avoid overfitting when training deep neural networks?

When we train deep neural networks, we can avoid overfitting using several techniques. However, since deep neural networks may suffer long training times, Early Stopping technique may be desirable independently by the other techniques we can adopt. There are three main techniques: Dropout, Weight decay and Early Stopping. Dropout consists in randomly deactivating some neurons of the networks with a given probability. At each epoch, every neuron is deactivated with this probability. This process will eliminate the co-adaptation phenomenon. It consists of neurons relying too much on specific neurons. This technique trains multiple weak learners which will be averaged at test time using weight scaling. Weight Decay is a specification of the more general type of regularisation technique called L-norm. This technique consists in adding the sum of the absolute value of the weights to the power of $q$, all multiplied by a parameter $\gamma$. The weight decay technique consists in having $q$ = 2. This technique counteracts overfitting since when we have overfitting we also have the increase of weights in the model. Early stopping is a technique in which we monitor both the training and the validation loss during training. Since overfitting implies that the validation loss starts to increase, early stopping stops the training after the algorithm has not found a new minimum after an arbitrary number of epochs called patience.
Another possibility to reduce the risk of overfitting is to augment data. Indeed Data Augmentation allows you to make your dataset being more general possible.

## What is early stopping and why does it help with overfitting?

Early Stopping is a technique monitoring both the training loss and the validation loss during training to detect overfitting, since overfitting is characterised by having the training loss continuing to decrease and the validation loss starting to increase. Early Stopping monitors the two losses and has

a term called patience which represents the number of epochs it will wait before stopping a training because of detected overfitting. Early Stopping stops a training if the network does not find a new validation loss minimum in “patience” number of epochs.

Early stopping prevents overfitting by limiting the number of epochs during which the network is trained, therefore preventing the training process from lasting longer than necessary, while retaining the best model found by the optimizer so far.

## What is the pain behind weight initialization? What is the relief?

Weight initialization is an important phase in training a neural network. The pain behind the weight initialization is that with wrong initialization we might not learn, learn too slowly or have problems regarding the gradient. For instance, with initialization of weights to zero we won’t be able to learn anything because the gradient will be null. With weight initialization with too big values, we will learn slowly. Moreover, with certain initialization we might have problems in terms of vanishing and exploding gradient.The relief is that there are some initializations we can use to be relatively sure that we won’t have problems with vanishing or exploding gradients. An example of that initialization is the one described by Glorot and Bengio in their paper. They initialise weights by sampling from a null mean gaussian with variance 2/(nin+nout). With this initialization all the layers will have the same variance and the problem of the vanishing and exploding gradient will be reduced. We can also use Xavier initialization, which helps keep the variance the same with each pass through the layers. Input and weights follow a Gaussian distribution with mean zero and variance 1/I where I is the number of input neurons.

## What is the pain behind the vanishing gradient? What is the relief?

The vanishing gradient problem arises due to the use of the backpropagation algorithm to compute gradients. Backpropagation computes gradients by recursively applying the chain rule of calculus to compute gradients of the error with respect to the weights. However, in deep networks, the gradients are multiplied many times over as they are propagated backwards through the layers, which can cause them to become very small.

It is the reason why you need to find a gradient ≥ 1, keeping in mind that if it is too big, the gradient becomes unstable.

This problem can be mitigated by using techniques such as weight initialization and activation functions that mitigate the problem, such as He initialization and ReLU activation function. Additionally, various optimization algorithms have been developed to address this issue, such as Adam, Adadelta, Adaptive gradient, Resilient propagation and RMSprop.

## What is the pain behind text encoding? What is the relief?

Text encoding methods are ways in which we represent words and texts in a treatable way. For instance, words can be represented using 1-hot-encoding and a text may be represented using the Bag Of Words representation (in which the order is not preserved) or in an N-gram. These representations are sparse, are not good at representing similarities and tend to live in extremely huge spaces. Since these problems come also with the fact that dependencies can be hardly recognized if the reference is far behind in the text, embeddings have been proposed. The main solution to solve these problems is embedding. Words are projected in a compressed and lower-dimensional space in which words are closer in space. Moreover, using word embeddings, two

words are similar depending on their distance. Then, texts can be represented using sentence embeddings.

## What is the pain behind the input of variable sized images in CNNs? What is the relief?

A CNN is composed of a feature extraction part and a fully connected part computing the result based on the features extracted by the convolutional part. The problem is that while convolutions does not depend on the dimensions of the input, the fully connected part is strictly dependent on the dimensions of the input. The dense layers have parameters and connections static and dependent by the number of neurons in the input layer. The solution to the problem of feeding into the same network images of different sizes is to use methods that are not dependent on the number of parameters in the input layer and have constant output. For instance, we can use Global Average Pooling layers after the convolutional part such that the dense part is not strictly connected to the image in input.
However, there is another solution we can use. Since the Dense layer is characterised by a linear relation, we can express it as a convolution. Then, we can replace all the dense layers with convolutions of size 1x1xN. In that way, we will be building a fully convolutional network

## What is weight decay and why does it help with overfitting?

Weight decay (also known as ridge regression in machine learning) is a L-norm regularisation technique. The general L-norm regularisation technique is defined as:

$$L(x) = E(x) + \gamma \sum |wi \mid q \; N \; i=0$$

Where $wi$ are the parameters of the model. Weight decay is the L-norm technique for which we have $q$ = 2. This technique helps with overfitting because it has been shown that overfitting makes the parameters of the model grow exponentially. Therefore, adding a regularisation term depending on the magnitude of the parameters of the model, reduces their magnitude and the risk of overfitting. However, it is not the only regularisation technique we use and it does not push the parameters to zero (it isn't an embedded feature selection method).

## When would you prefer weight decay with respect to early stopping? How can you tune the gamma parameter of weight decay?

Weight decay is better in cases where we have data scarcity. If not too much data is available even before augmenting it is preferable to avoid early stopping as it requires a lot of validation data to stop model training at the correct time, while weight decay works regardless of the quantity of data available.
The gamma parameter in weight decay defines how much the regularisation will weigh in the loss function. It must not be too high, or it will overcome the gradient and just force down the weights senselessly, nor too low, in which case the regularisation will be too weak. As is the case with many hyperparameters, the best way to tune it is with cross validation. In general it is best to start with small values, and try increasing it by an order of magnitude if it works well, or decrease it by the same if it degrades the performance of the network.

## Discuss the good and the bad of sigmoid, hyperbolic tangent, and ReLu. When should you use each of them? Why? Provide their derivatives.

The good idea behind the sigmoid function is to be able to perform binary classification while being able at the same time to describe the probability of being part of one or the other class, not to mention that it is differentiable. The bad idea of sigmoid activation function is that its output value is in the interval (0,1). It means it is susceptible to the vanishing gradient problem.

$$\sigma(x) = 1\ 1 - e - x\ \sigma\ '\ (x) = \sigma(x)(1 - \sigma(x))$$

The hyperbolic tangent is somehow related to the sigmoid function since it can be defined as $\tanh(x) = 2\sigma(2x) - 1$. The hyperbolic tangent represents the binary classification problem as the sigmoid. It resembles well the identity function since tanh(0) = 0. This makes the training of the neural network easier and more similar to the training of a linear model. The problem of the hyperbolic tangent is the same as the one of sigmoid.

$$\tanh(x) = e\ x - e - x/e\ x + e - x\ \tanh'\ (x) = 1 - \tanh 2\ (x)$$

The ReLU is a non-linear function defined as $ReLU(x) = \max(0, x)$. The idea of the ReLU function is to avoid the vanishing gradient problem by using a function whose derivative is $ReLU\ '\ (x) = 1x>0$ . However, the problem is that with this function we have the dying neuron problem. If the function ends in the plateau, the neuron will be stuck there. Moreover, the function is not differentiable in $x = 0$. We should use ReLU instead of sigmoid or hyperbolic tangent in case we deal with a task susceptible to the vanishing gradient problem in the hidden layers. It prevents the vanishing gradient from happening. Sigmoid and hyperbolic tangent are somehow interchangeable, hyperbolic tangent provides similar results to sigmoid while being faster than the hyperbolic tangent. We could use the binary classification problem as an output activation function to be able to classify between the two different classes we have.

## Which conceptual difference does Deep Learning differ significantly from being just another paradigm of Machine Learning similarly to supervised learning, unsupervised learning, reinforcement learning, etc.?

Every machine learning algorithm is composed of two steps: feature extraction and elaboration. In classical machine learning algorithms, the features are predefined depending on the problem by hand. In Deep Learning the features are learnt by the DL algorithm as well as the elaboration of the features. Given that, DL algorithms need extremely bigger datasets with respect to machine learning algorithms.

## Make an example of an application where Classical SUPERVISED learning is used and then present its Deep counterpart.

An application for which we can implement a classical supervised machine learning algorithm is image classification. The Iris dataset contains images of different types of iris flowers. They can be classified using some known features about the flowers using classical algorithms such as SVM. A DL algorithm can perform the exact same task without the necessity of defining manually the features

describing the flower. Convolution and pooling are some of the operations we can use to detect with a neural network the features of the flower. Convolutional Neural Networks are able to learn the features of the flowers. They can also classify the features in the fully connected part of the network by using dense layers (in case image size is fixed). There are many DL models we can use to perform this task. For instance, we can fine tune VGG16

## Make an example of an application where Classical UNSUPERVISED learning is used and then present its Deep counterpart.

PCA in classical machine learning and Autoencoders in Deep Learning. They both handle dimensionality reduction but PCA is a linear method that tries to retain as much as variance as possible, while autoencoder is a non-linear neural network that tries to reconstruct its input. It also learns a compressed representation of its features.

## What is Xavier initialization and why does it help with vanishing gradient?

Xavier initialization, also known as Glorot initialization, is a method for initialising the weights of a neural network.
Xavier initialization helps with the vanishing gradient problem by initialising the weights with small random values that are chosen from a distribution with a specific variance, which helps to ensure that the gradients are not too small, allowing the network to learn more effectively.
Initialising weights with

$$w \sim N\left(0, \frac{1}{n_{in}}\right)$$

leads to a faster convergence and more stability.

## Can the choice of the activation functions help with the vanishing gradient issue? Why?

Yes, some activation functions have been developed to avoid vanishing gradients, an example is Relu or Leaky Relu that put all negative inputs to 0.
If we use Relu function in place of a sigmoid function, the value of the partial derivative of the loss function will be having values of 0 or 1 which prevents the gradient from vanishing

## Can transfer learning help with the vanishing gradient issue? Why?

Transfer learning is a technique used to exploit pre-trained subnets that have already been tested extensively on a large dataset. These subnets work as very well tuned feature extractors, and are very deep architecture. Therefore they don't suffer from the vanishing gradient issue, because of the extensive testing done, even though they are very deep. So, doing transfer learning and adding some

layers in the end, after fine-tuning the network, will result in a training that won't suffer from the vanishing gradient issue, as the supernet doesn't have this problem as well.
–

- Yes it can help because we use the pre-trained weights of the feature extractor as our weight initialization
- When performing transfer learning only the new FC layers at the top are trained (the feature extractor remains "frozen" at the weights he came with)
- This in turn reduces the amount of layers that the backpropagation has to take ⇒ Vanishing gradient issue is a bit smaller
- It is however important to note that transfer learning can't solve all the vanishing gradient problem and it's not a general solution for it, but it can be a useful approach in some cases.

## Describe the Word2Vec architecture

The architecture takes as input a context around the word that we want to predict. In the middle of that context sits the word that we want to predict (in the version called continuous bags of words).

- This input layers is one-hot-encoded

The input words are first converted to their corresponding word embeddings, which are then averaged to form a single, fixed-length context vector

This context vector is then passed through a fully connected layer, followed by a softmax activation function to produce a probability distribution over the vocabulary for the target word

The model is trained using backpropagation to minimise the negative log-likelihood loss between the predicted probability distribution and the actual target word.⇒ It is trained unsupervised

- Idea: you use the regularity of the neighbourhood of a word to structure the embedding ⇒ Words share neighbourhood ⇒ Should be close close each other
- Goal: find the encoding s.t. if you average the encoding of the surrounding words, you get the likelihood of the target word

## What is ReLU and why does it help with vanishing gradient?

ReLU (Rectified Linear Unit) is a popular activation function used in neural networks. The ReLU activation function is defined as f(x) = max(0, x), where x is the input to the activation function. In other words, the output of the ReLU function is the input value if it is positive, and zero if it is negative.

The vanishing gradient problem is often caused by the use of activation functions that saturate, such as the sigmoid function, which produces output values that are close to either 0 or 1. This can make it difficult for the network to learn, because the gradients of the weights in the network are very small.

ReLU activation functions, on the other hand, do not saturate, meaning that the gradients of the weights in the network are not as likely to become very small. This makes it easier for the network to learn and improves the performance of the network. Additionally, ReLU is computationally efficient since it only requires a comparison and max operation.

The only problem with ReLU is that the positive slope is unbounded, and this could result potentially in the exploding gradient problem. The main solution to the exploding gradient in ReLu is the application of the gradient clipping. Other solutions, such as weight normalisation could be employed to solve this problem.

## Guideline how to choose the error function?

It depends on architecture and problem you're solving
MSE and categorical cross entropy will do for "normal" regression and classification, but i think for segmentation there's a few possible choices but he didn't get into it in the course (intersection over union, hausdorff, ...), for GANs you have the min-max problem with a million possible formulations, with siamese networks he showed contrastive loss and triplet.

## How do you initialise the weights of a deep neural network? Why?

In general we use Glorot & Bengio or He initializations. These techniques take into account the amount of neurons in input/output. The idea is that if we have many inputs, then the variance of the value that enters an output neuron will become higher, and vice versa if we have few inputs. We want this variance to be correct, or the output neurons might saturate

## What is dropout and why does it help with overfitting in neural networks?

Dropout is a regularisation technique for neural networks that helps to prevent overfitting. It works by randomly dropping out (i.e., setting to zero) a certain percentage of the activations within a layer during training.

Dropout helps to prevent overfitting by forcing the network to learn multiple, redundant representations of the data, rather than relying on a small number of neurons to learn the task. When some of the neurons are dropped out, the network has to learn to perform the task using the remaining neurons, and this encourages the network to spread the responsibility of learning the task across multiple neurons.

During the training, dropout works by randomly shutting off some of the neurons in the network, this means that the network can't rely on the same set of neurons to make predictions, it has to learn how to use different neurons in different situations, this makes the network less sensitive to the specific weights of the neurons, and this results in a more generalizable model.

During the testing, dropout is usually turned off, and all neurons are used to make predictions.

In summary, dropout is a regularisation technique that helps to prevent overfitting by randomly dropping out a certain percentage of the activations within a layer during training, this forces the network to learn multiple, redundant representations of the data, rather than relying on a small number of neurons to learn the task, and this results in a more generalizable model.

## Can the use of Dropout help with the vanishing gradient issue? Why?

No, dropout helps reduce overfitting, but not vanishing gradient which are solved by evaluating the activation function and not by simply dropping some neurons.
–

- No, dropout helps reduce overfitting
- Dropout works by randomly deactivating some neurons during the training process
- This enforces the network to learn an independent features preventing hidden units to rely on other units (co-adaptation)
- Trains weaker classifiers, on different mini- batches and then at test time we implicitly average the responses of all ensemble members
- ⇒ Smoothens because it averages

## What is the Attention Mechanism?

The attention mechanism is a technique used in neural networks that require a memory to compute their output. In summary it works by focusing the attention of the network on a certain input part (as per a value in an input sequence or in a memory location for a TNM) to better compute the output.

Consider the Inception Net module. Which of the following statements are true?

(1 punto)

- [ ] it uses multiple convolutional filters of different sizes in parallel
- [ ] it was the first to introduce skip connections
- [ ] it has been the first module used for semantic segmentation
- [ ] it was the first learnable upsampling filter
- [ ] it leverages 1x1 convolutions to reduce the computational burden

Solution:

The only correct solutions are the first and last options.

1. correct
2. false, the first one introducing skip connection was ResNet
3. false, it was UNet
4. false

5. correct, 1x1xN convolutions are required as bottleneck layers for improved computational efficiency

## What is Hebbian learning?

Hebbian learning is a rule for adjusting the strength of connections between neurons in a neural network, based on the activity of those neurons. The basic idea is that if two neurons are active at the same time, the connection between them should be strengthened. This is often referred to as "Hebb's rule" or "Hebbian learning rule." The weight of the connection between A and B neurons is calculated using:

$$\begin{cases} w_i^{k+1} = w_i^k + \Delta w_i^k \\ \Delta w_i^k = \eta \cdot x_i^k \cdot t^k \end{cases}$$

where:
η is the learning rate;
x is the input of neuron A at time k;
t is the desired output of neuron B at time k.

Which of these techniques might help with overfitting?
(1 punto)

- [x] Weight Decay
- [x] Early Stopping
- [x] Dropout
- [ ] ReLu and Leaky ReLu
- [ ] Stochastic Gradient Descend
- [x] Xavier Initialization
- [x] Batch Normalization

Solution: Weight Decay + Early Stopping + Dropout + Xavier Initialization + Batch Normalisation

Which of these techniques might help with vanishing gradient?
(1 punto)

- [ ] Dropout
- [ ] ReLu and Leaky ReLu
- [ ] Early Stopping
- [ ] Xavier Initialization
- [ ] Stochastic Gradient Descend
- [ ] Weight Decay
- [ ] Batch Normalization

Solution: ReLu + Xavier + Batch

One of your friends shows up with a brand new model she has invented, but which she is not able to train. You suspect this could be due to VANISHING/EXPLODING GRADIEN T, what would you look at to check THIS hypothesis?

HINT: Six and only six are correct; if you mark more, these will be counted as errors, if y ou mark less you are loosing points ...
(3 punti)

- [ ] The activation functions used for neurons
- [ ] The output values of neurons during training
- [ ] The distribution of the weights during training
- [ ] The initialization of the weights
- [ ] The depth of the network
- [ ] The balance of classes
- [ ] The gradient descent algorithm used
- [ ] The average gradient norm during training
- [ ] The value of the learning rate
- [ ] Whether the model contains Convolutional layers
- [ ] Whether the model contains Recurrent Layers
- [ ] The loss function used

Looking at the solution of the exam in question the correct answers are:

- the activation function used
- the distribution of the weights during training
- the initialization of the weights
- the depth of the network
- the average gradient norm
- whether recurrent layer

## What is Batch Normalisation?

It improves gradient flow through the network, by normalising each slice of volume independently. It requires 4 parameters (2 are trainable and 2 are not trainable).

You can check the tensorflow documentation to see that there are 4 parameters in total:

- Moving average (mu): used for computing the moving average of the batch inputs, this is needed for the normalisation computation. Therefore this counts as a non-trainable parameter, because it depends on the training data distribution.
- Moving standard deviation (sigma): used for computing the moving variance of the batch inputs, this is needed for the normalisation computation. This also counts as a non-trainable parameter, because it depends on the training data distribution.
- Scaling factor (gamma): trainable parameter, used to adjust the normalised output scale
- Offset factor (beta): trainable parameter, used to adjust the the normalised output bias

Batch normalization adds after standard normalization

$$x_i' = \frac{x_i - E[x_i]}{\sqrt{\text{var}[x_i]}}$$

a further a parametric transformation

$$y_{i,j} = \gamma_j x_i' + \beta_j$$

Where parameters $\gamma$ and $\beta$ are learnable scale and shift parameters.

## What is Binary-cross entropy?

Binary cross-entropy is a loss function that is commonly used in machine learning, particularly in supervised learning problems where the goal is to classify an example as belonging to one of two classes. It is used to measure the dissimilarity between the predicted probability distribution and the true probability distribution. The true probability distribution is represented by a vector where each element corresponds to a class and has a value of 1 if the example belongs to that class and 0 otherwise. The predicted probability distribution is represented by a vector of predicted probabilities for each class.

The binary cross-entropy loss function is defined as:
-loss(y, ŷ) = -(y * log(ŷ) + (1-y) * log(1- ŷ))

where y is the true probability distribution and ŷ is the predicted probability distribution.

The binary cross-entropy loss function is often used in binary classification problems, such as in image classification problems where the goal is to classify an image as either containing a certain object or not.

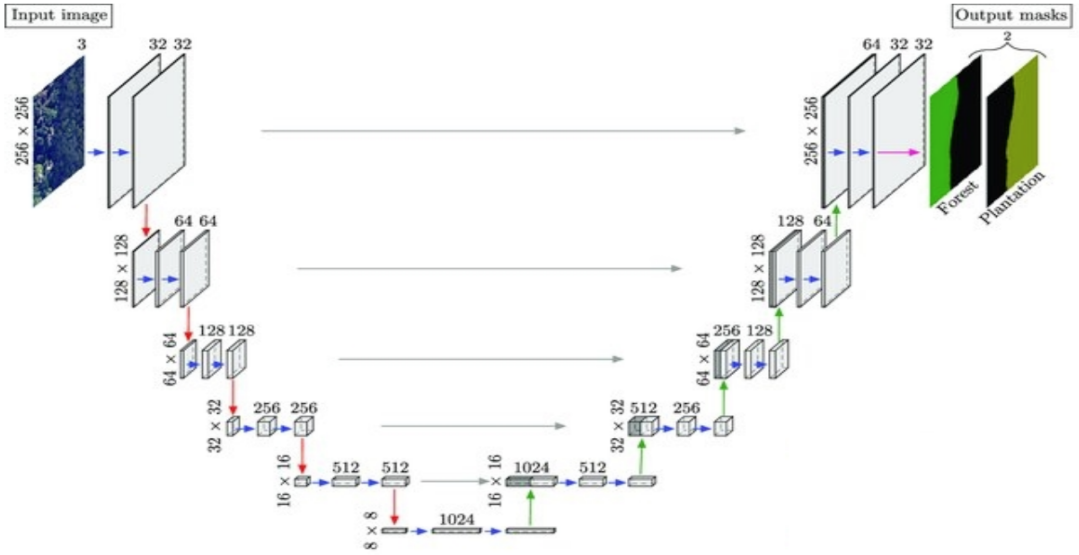
## Briefly describe what Siamese networks are and what they are used for?

A Siamese Neural Network is a class of neural network architectures that contain two or more identical subnetworks.
Why do we need it? Imagine a situation when you are building a face identification system. Building the pure classification model is not feasible for this matter because you might have only 10 images of each person in your face recognition base and every time a new person enters in the system you need to retrain the whole network from the beginning.
We pass multiple (usually two) images through identical NNs, then as output they produce embeddings, and then they are passed as inputs to a contrastive loss function.
The objective of this kind of network is to make the difference between embeddings of the same person less than the distance between different people.

# EXERCISE

QUESTION 3: CONVOLUTIONAL NEURAL NETWORKS

With reference to the deep neural network in the image below, answer the following questions.




## Enumerate

Consider that blue arrows refer to convolutions having a spatial extent 3x3,while the magenta one does not perform spatial averages. You can use the calculator, but we are more interested in the formulas than on numbers! (Note: for some of the blocks you might need to infer the number of channels or the size from the preceding/following blocks)

It is a U-net taking as input an image of size 256*256 to perform image segmentation between forest and plantation

Number of parameters of the U-Net:

Contracting part:
Input | (None,[256,256,3]) | 0
Conv2D | (None,[256,256,32]) | 32*3*3*3+32
Conv2D | (None,[256,256,32]) | 32*3*3*32+32
MaxPool2D | (None, [128,128,32]) | 0
Conv2D | (None,[128,128,64]) | 64*3*3*32+64
Conv2D | (None,[128,128,64]) | 64*3*3*64+64
MaxPool2D | (None,[64,64,64]) | 0
Conv2D | (None,[64,64,128]) | 128*3*3*64+128
Conv2D | (None,[64,64,128]) | 128*3*3*128+128
MaxPool2D | (None,[32,32,128]) | 0
Conv2D | (None,[32,32,256]) | 256*3*3*128+256
Conv2D | (None,[32,32,256]) | 256*3*3*256+256
MaxPool2D | (None,[16,16,256]) | 0

Conv2D | (None,[16,16,512]) | 512*3*3*256+512
Conv2D | (None,[16,16,512]) | 512*3*3*512+512
MaxPool2D | (None,[8,8,512]) | 0
Conv2D | (None,[8,8,1024]) | 1024*3*3*512+1024
Conv2D | (None,[8,8,1024]) | 1024*3*3*1024+1024

Upsampling part:
Transpose2DConv | (None,[16,16,512]) | 512*2*2*1024+512
Aggregation | (None,[16,16,1024]) | 0
Conv2D | (None,[16,16,512]) | 512*3*3*1024+512
Conv2D | (None,[16,16,512]) | 512*3*3*512+512
Transpose2DConv | (None,[32,32,256]) | 256*2*2*512+256
Aggregation | (None,[32,32,512]) | 0
Conv2D | (None,[32,32,256]) | 256*3*3*512+256
Conv2D | (None,[32,32,256]) | 256*3*3*256+256
Transpose2DConv | (None,[64,64,128]) | 128*2*2*256+128
Aggregation | (None,[64,64,256]) | 0
Conv2D | (None,[64,64,128]) | 128*3*3*256+128
Conv2D | (None,[64,64,128]) | 128*3*3*128+128
Transpose2DConv | (None,[128,128,64]) | 64*2*2*128+64
Aggregation | (None,[128,128,128]) | 0
Conv2D | (None,[128,128,64]) | 64*3*3*128+64
Conv2D | (None,[128,128,64]) | 64*3*3*64+64
Transpose2DConv | (None,[256,256,32]) | 32*2*2*64+32
Aggregation | (None,[16,16,64]) | 0
Conv2D | (None,[256,256,32]) | 32*3*3*64+32
Conv2D | (None,[256,256,32]) | 32*3*3*32+32

Conv2D (1x1) | (None,[256,256,2]) | 2*1*1*32+2

## What is the task this network is expected to solve? What is the rationale behind this architecture and which loss function would you use to train the network? Justify all your statements.

A full-image training by a weighted loss function

When training a U-Net for image segmentation, the most commonly used loss function is the cross-entropy loss. This loss function compares the predicted segmentation mask with the ground truth mask and calculates the error between them.
The cross-entropy loss function is defined as:

$$L = -\sum y_i * \log(\hat{y}_i)$$

where $y_i$ is the ground truth label and $\hat{y}_i$ is the predicted label for the ith pixel. The sum is taken over all pixels in the image.

# Compute Numbers of parameters

## How to compute the number of parameters of a CNN

Layers have filters $F$, kernel size $(k_w \times k_h)$, stride $(s_w \times s_h)$ and padding $p = \{0,1,2\}$

- Depending on the type of the layer there may be a different number of parameters
  - Generally, $|par| = 0$, if the layer is composed by an algorithm with no parameters
  - Conv $|par| = |F| * (k_w * k_h * d_{in} + 1)$
  - Dense $|par| = d_{out} * (d_{in} + 1)$
  - LayerNormalization $|par| = 2 * d_{in}$ normalize ∀example in a batch independently
  - BatchNormalization $|par| = 4 * d_{in}$ normalize across batches
    - $\gamma, \beta$ are learned during training
    - moving mean and moving variance non-trainable
- Dimension of the output image $(w_{out}, h_{out}, d_{out})$ of a layer is computed as:

$$(w_{out}, h_{out}, d_{out}) = \left( \left\lfloor \frac{w_{in} + (k_w - 1) * (p - 1)}{s_w} \right\rfloor, \left\lfloor \frac{h_{in} + (k_h - 1) * (p - 1)}{s_w} \right\rfloor, ? \right)$$

  - Pooling by default has $k = s = 2$ and valid $\quad d_{out} = d_{in}$
  - Conv by default, $s = 1$ and valid $\quad d_{out} = |F|$
  - Flatten $(\cdot) = (w_{in} * h_{in} * d_{in})$
  - GlobalPooling $(\cdot) = (d_{in})$
- Receptive field of a layer $(l)$ is the size of input data used to obtain 1 output data after layer $(l)$. It's computed per dimension in a backward recursive way

$$r^{(l)} = r_0 = \begin{cases} r_{l+1} = 1 & initial\ condition \\ r_x = k_x + (r_{x+1} - 1) * s_x & \forall x = 0, \dots, l \end{cases}$$

  - Pool layers leads to $\quad r_x = r_{x+1} k_x$
  - Conv layer leads to $\quad r_x = r_{x+1} + (k_x - 1)$
  - Different padding does not influence the receptive field.
  - In CNN usually the kernel and stride size are the same for all dimensions.