**Week 1 – Notes**

**Convolutional Neural Networks**

**Computer Vision**

There are multiple computer vision problems like image classification, object detection and neural style transfer

CNNs appeared because you cannot use classical fully connected NN to learn from images because the input size and the number of weights would be of the order of millions or even billions

A picture containing handwriting, line, diagram, font

Description automatically generated

**Edge Detection Example**

CNNs work based on finding features like edges (vertical, horizontal, and so on)

To detect them we are using filters / kernels that are applied on images through the process of convolution which means that you multiply each pixel with the corresponding filter value and in the end, you just add the values

A screenshot of a whiteboard

Description automatically generated with low confidence

Higher values denote shades of white

A screenshot of a math game

Description automatically generated with low confidence

We can see that a vertical edge detector finds regions where there’s a transition from higher to lower values

**More Edge Detection**

If the transition is inverse, from lower values to higher ones, then, in our case, the result of the convolution would be a matrix composed of 0 and -30

In the same fashion we can detect horizontal edges

There are many types of filters (the first one is called Prewitt), for example the Sobel variant is more robust because it weighs more the central value of each side

A picture containing line, text, diagram, font

Description automatically generated

Essentially, we would want to let the CNN learn the values of these filters so that it can extract the best features for the data set at hand and in addition to find edges of all kinds of inclinations

To do this, we treat each value of a filter as a weight that is learned through backprop

**Padding**

After applying the convolution, we end up with a matrix of the size n – f + 1 x n – f + 1, where n represents the size of the input image and f the size of the filter

Thus, there are some flaws:by applying many convolutions, we end up with a smaller and smaller output and we do not take into consideration values that are on the edge of the image

To solve these problems, we can use a padding with zero values around the image, then the output after the convolution will have a size of n + 2p – f + 1 x n + 2p – f + 1

There are 2 types of convolutions:

Valid – they have not padding

Same – pad so that the output size is the same as the input size (p = (f – 1) / 2)

The filter size is usually odd (by convention) because in this way we have a central value when we apply it and because we can pad symmetrically on the left and right of the image

Common filters are 1x1, 3x3, 5x5 and 7x7

**Strided Convolutions**

You may want to use a strided convolution, so to move the kernel over the input by jumping a number of values (columns or rows)

If we use a stride of 2, we move the kernel by 2 positions, also a stride of 1 is the basic movement of the kernel

The dimension of the output can be computed by using the following formula:

A picture containing font, text, number, line

Description automatically generated (nxn image, fxf filter, padding p, stride s)

[image processing - 2D convolution: Flipping the kernel? - Computer Science Stack Exchange](https://cs.stackexchange.com/questions/11591/2d-convolution-flipping-the-kernel)

Technically, we aren’t using convolutions, we are applying an operation called cross-correlation

The convolution flips the kernel on both axis before applying it and this is the correct waw (it was coined in signal processing)

The convolution is associative, while the correlation is not: (A\*B)\*C = A\*(B\*C)

Let’s say that you have an image with a single 1 in the center and 0 otherwise + a kernel with values from 1 to 9 => by applying the convolution, you get the same kernel, but by applying cross-correlation you obtain the flipped kernel

**Convolutions Over Volume**

In DL, the convolutions are applied over volumes

For example, we use input images that have 3 channels, so the kernel also has 3 channels. However, the output has only one channel

A calculator and a calculator

Description automatically generated with low confidence

The convolutions are applied in such a way that in one kernel with 3 channels, we can detect one type of edge in each color channel

In DL, we usually apply several filters in one layer, so we extract many features out of the input

For each filter we need to have as many channels as the input

**One Layer of a Convolutional Network**

We can trace a parallel between CNNs and fully connected NNs

a[l-1] are the inputs, w[l] the filters, b1 and b2 represent real values which are broadcasted to the matrices and we apply an activation function to obtain the a[l] output

a[l] is just a stacking of the results of applying several filters on the input

A picture containing text, diagram, plan

Description automatically generated

The number of params of a layer of a CNN is stable regardless of the input image size

For a layer l of a CNN, we have the following dimensions:

A picture containing text, handwriting, font

Description automatically generated

**Simple Convolutional Network Example**

In a CNN we usually use 3 types of layers: convolutions (CONV), pooling (POOL) and fully connected (FC)

The FC layer is the last one and the outputs of it are passed through a softmax or a sigmoid

It’s extremely common to have the height and width of the data smaller as smaller as we progress through the network, and to have more and more channels

A picture containing text, drawing, sketch, diagram

Description automatically generated

**Pooling Layers**

They reduce size of the representation and makes the network better detect features (the network becomes more robust)

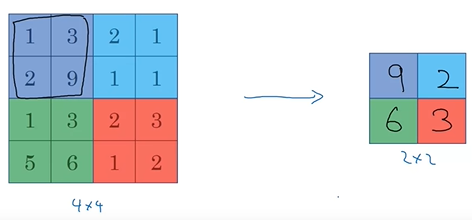
This operation only has hyper-parameters, but it does not have parameters to learn

The hyperparameters are type of pooling, filter size, stride size and the padding size (which rarely is different than 0)

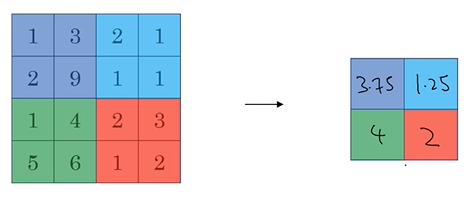
We compute the output size in the same way we do for convolutions floor((n + 2p – f) / s + 1)

There, the number of input channels is the same as the number of the output ones

Max pooling:

(f=2, s=2)

Average pooling:



Max pooling is used more frequently than the average pooling

Usual values of f and s: f=2 and s=2 (divides by 2 the size of the input) and f=3 and s=2

**CNN Example**

LeNet-5 architecture

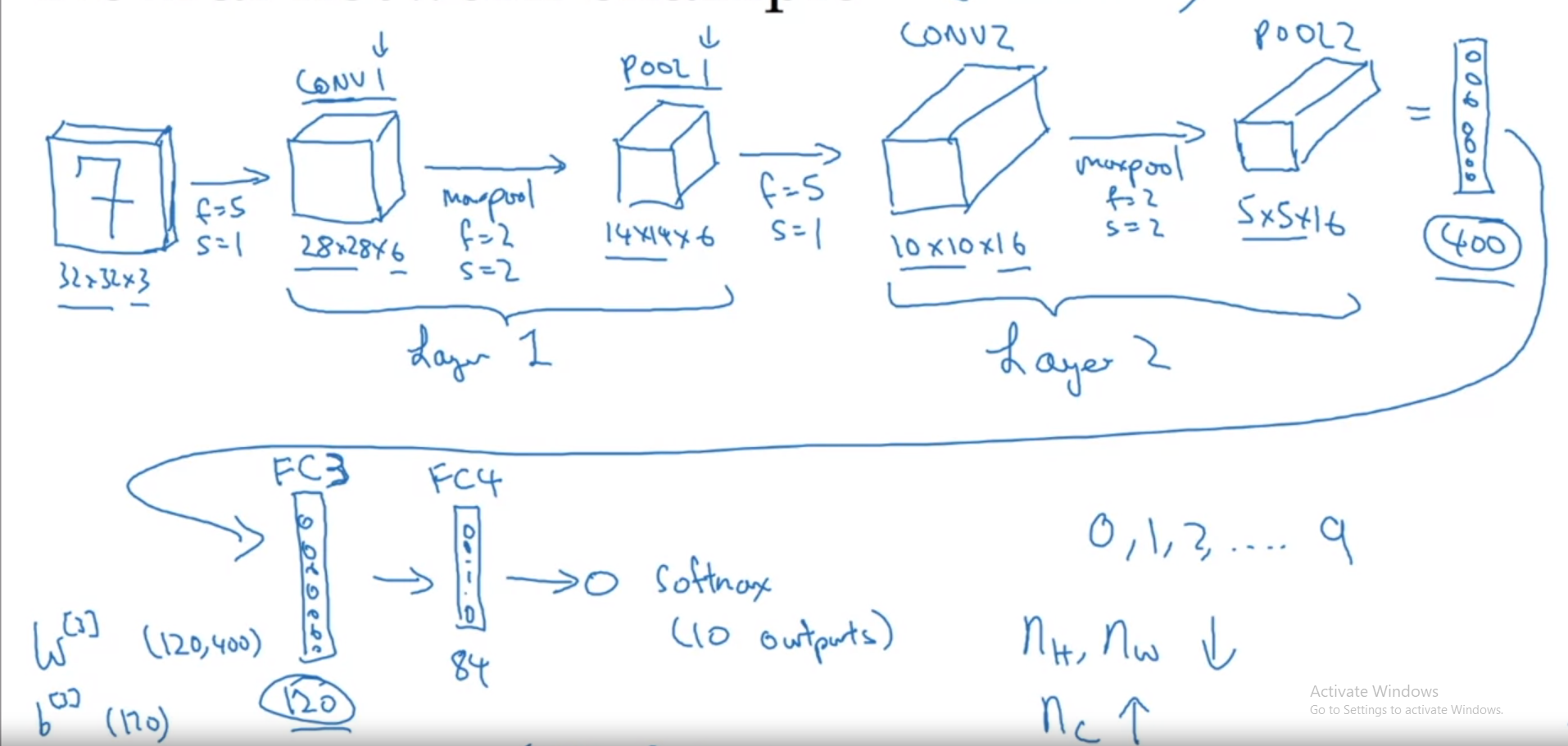
This architecture has the following layers:

Conv -> pool -> conv -> pool -> fc -> fc -> fc -> softmax

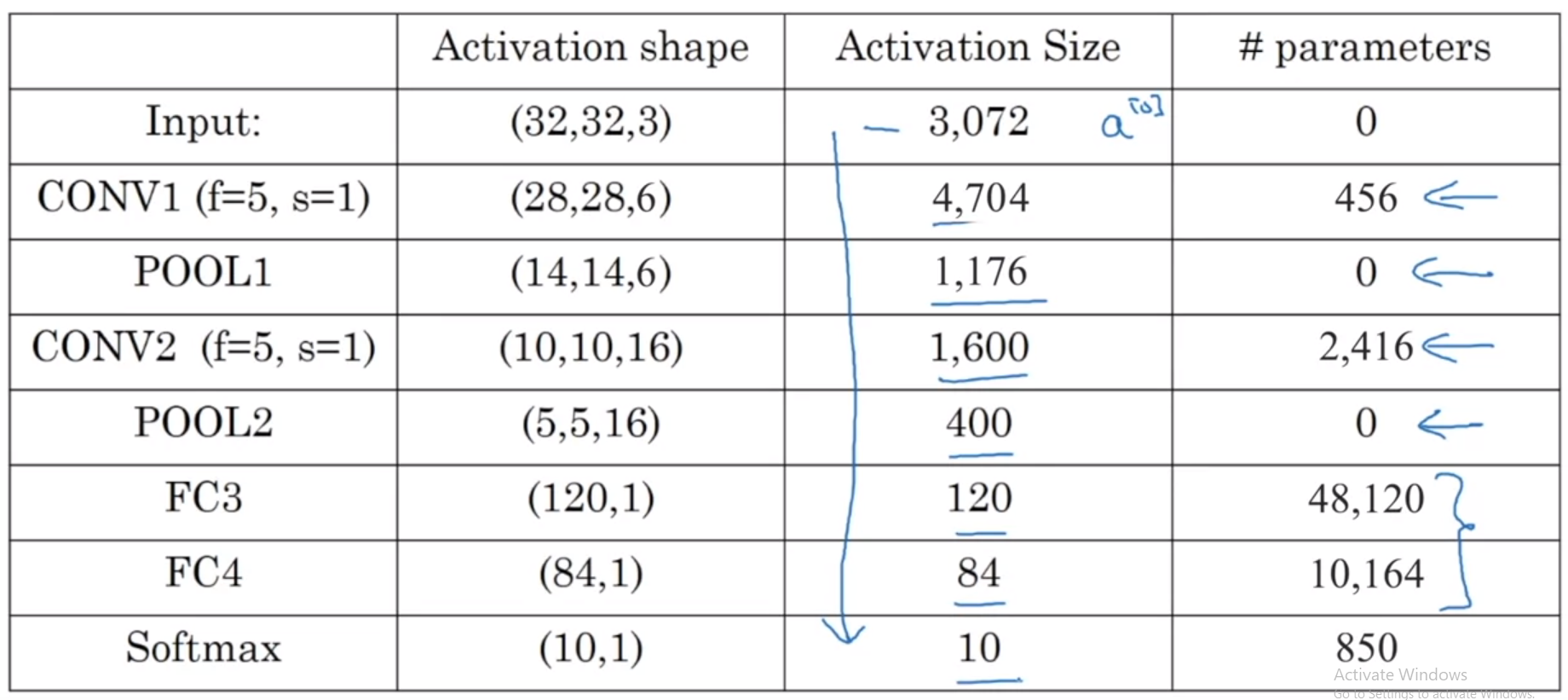
Usually, because a pooling layer doesn’t have parameters, it’s taken together with a conv layer, so that conv + pool represent one layer

We can see that the flatten isn’t considered a layer (there aren’t any params)

Additionally, as we go through the network, the nh and nw are smaller and nc is larger



Analysis of the network (activations and number of parameters):



For conv, the number of params = f \* f \* nc prev layer \* nc current layer

**Why Convolutions?**

Firstly, CNNs help reducing the number of parameters drastically, compared to a fully connected NN

The number of params is agnostic of the input size

Convolutions have 2 important properties:

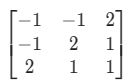
1. Parameter sharing: a feature detector that’s useful in one part of the image is probably useful in another part of the image; reduces the number of params, thus the overfitting

2. Sparsity of connections: in each layer, each output value depends only on a small number of inputs, thus the network is more robust to features (less chances of overfitting) – even if the features will be shifted in other image, the network will recognize them

The training is exactly as we did for fully connected NN: we use gradient descent to optimize the parameters, so that the cost function J will be reduced

**Quiz notes**

This represents a filter that detects 45-degree edges because there is a high delta between -1 and 1, even though on the diagonal the values are not 0



If the padding is 2, then you add 4 to the height dimension and 4 to the width dimension

Even though the pooling layer doesn’t have parameters, it affects the backprop because everything that influences the loss appears in the backpropagation, plus the gradients flow through the Pooling layers

In Pooling layers, we usually set f=s

**Assignment 1**

You can pad with np.pad

A pooling layer helps reduce computation, as well as helps make feature detectors more invariant to the position in the input

The 2D output of the convolution is called the feature map

For backpropagation you essentially backpropagate the weights through the kernels by multiplying corresponding values or parts of the kernel

When we want to backpropagate the gradients through the pooling layer, if we do that through the max pooling, we have to let the gradients pass unchanged only through the position of the biggest values of the kernel, but for average pooling, we have to multiply the gradients with 1 / (height \* width) of the kernel, because each value contributed equally during the forward pass

Through a pooling layer we propagate the derivative of A (get dA and return dA\_prev)

**Assignment 2**

Keras is an abstraction built on top of TensorFlow

TF Keras has 2 APIs: Sequential and Functional

The Sequential API is used for simple models that have exactly one input and one output tensor; it’s used to create models layer by layer; you add layers with .add() and remove them with .pop(); you can think of the Sequential API as a list of layers; for the first layer of the model you have to specify the input\_shape

When you compile your model and specify the metric as “accuracy”, TF knows what type of accuracy you want based on the type of loss

TF flow: build a model -> compile it (.compile(…)) -> fit it (.fit(…)) -> evaluate it (.evaluate(…))

The Functional API is more complex because it can handle non-linear topology, shared layers, layers with multiple inputs or outputs; you can think of this API as a way to connect layers as nodes in a graph, for example to have skip connections; each layer has an output and an input, that’s why you can see them as nodes in a graph, because you can connect them as you wish; you have to initially define input\_img = tf.keras.Input(shape=input\_shape) and after you defined all layers to combine everything with model = tf.keras.Model(inputs=input\_img, outputs=outputs); each layer outputs the values and gets inputs in an explicit way

In TF, you can define the batch size in the fit function or directly when you load the data (then you don’t have to mention it again inside the fit function)

The .fit(…) method returns the training history

[The Sequential model  |  TensorFlow Core](https://www.tensorflow.org/guide/keras/sequential_model)

[The Functional API  |  TensorFlow Core](https://www.tensorflow.org/guide/keras/functional_api)