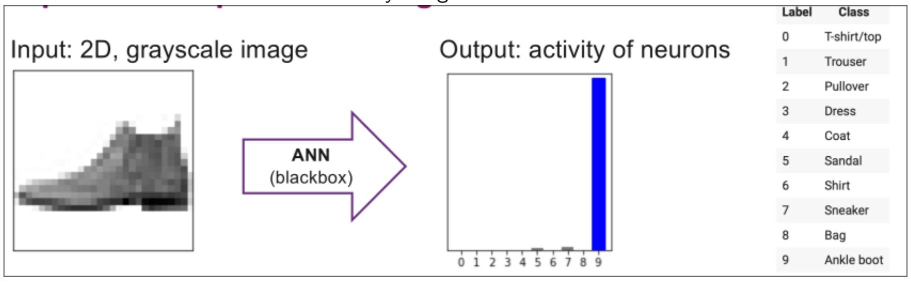
0 Contents

1 CNN Convolutional Neural Networks

1.1 Keras

A python library that wraps tensorflow for classification.
We will use this in this module to classify images like so:



Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

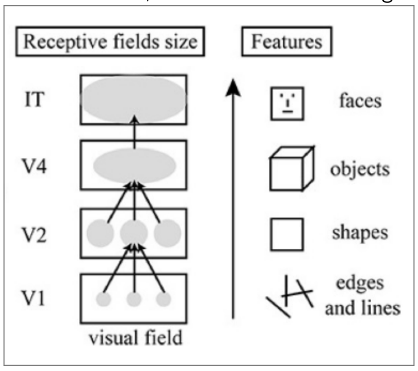
1.2 Flattening

When we convert an image into a long vector, we lose information in the human sense, or rather make it hidden.
todo, explain what is hidden and why

2 Convolution

2.1 Firing of neurons

Neurons are clearly connected to something very specific, this would then also be reflected in the artificial neural network. In other words, neuron 1 handles horizontal lines, another a line with a slight angle and so-on.

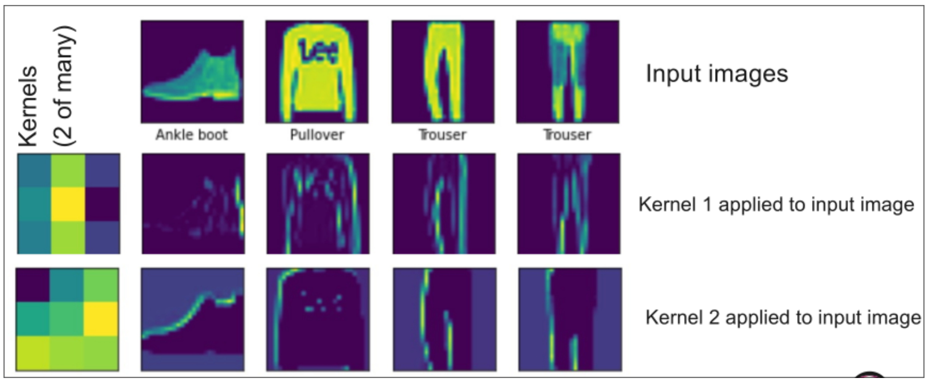


2.2 mathematical model of a feature detector

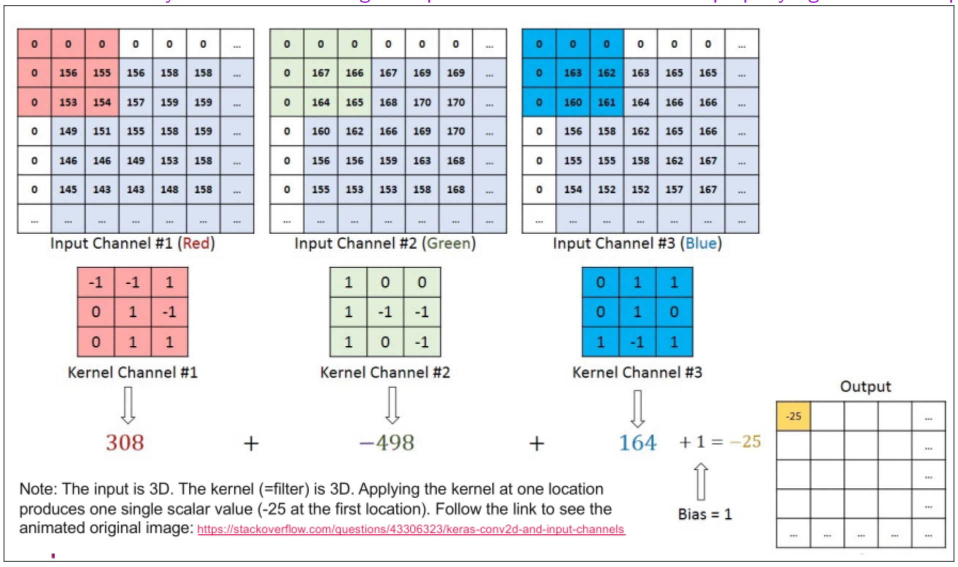
- Two Inputs
  - a picture
    Note that rgb would give you 3 channels red, green and blue
  - A filter(kernel)
    - an m by n matrix in the simplest case (1 channel, grayscale).
    - an m by n x 3 "stack of matrices" in the case of a 3 channel input (e.g. an RGB image)
    - an m by n x d "stack of matrices". The depth of the kernel must equal the number of input channels.

- One Output
  A feature map (where is the thing that we wanted to search / determine by)
  One convolution produces one feature map. Even if the input and the filter have multiple channels, the output of the convolution has one channel.
- The Operation: Convolution
  We convolve the input image with the convolutional kernel

2.3 Example of convolution


Kernels
(2 of many)
Ankle boot
Pullover
Trouser
Trouser
Input images
Kernel 1 applied to input image
Kernel 2 applied to input image

As you can see here, a filter will be used to detect something specific, like a pattern.
This means that you will be combining multiple different filters in order to properly figure out what picture the underlying image is composed of.



Input Channel #1 (Red)

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Kernel Channel #1

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #2

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #3

0	1	1
0	1	0
1	-1	1

$308 \quad + \quad -498 \quad + \quad 164 \quad +1 = -25$

Bias = 1

Output

-25				...
				...
				...
				...
...	...	...	...	...

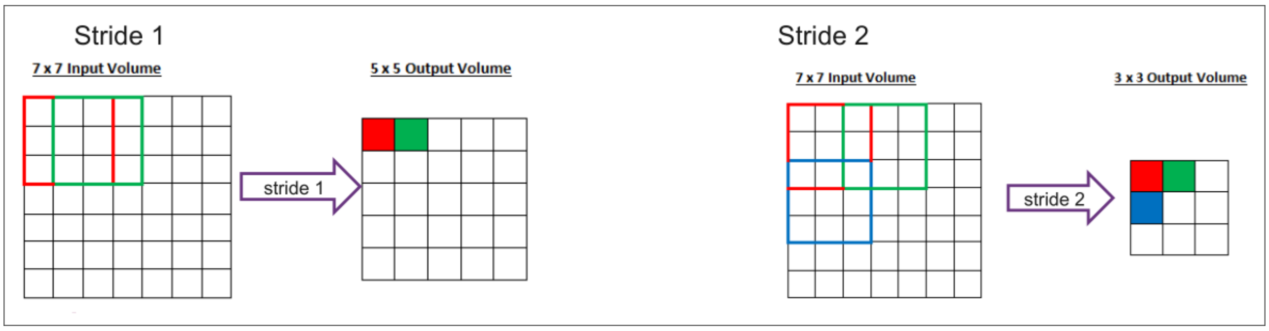
Note: The input is 3D. The kernel (=filter) is 3D. Applying the kernel at one location produces one single scalar value (-25 at the first location). Follow the link to see the animated original image: https://stackoverflow.com/questions/43306323/keras-conv2d-and-input-channels

Explanation for the first calculation: 0 * -1 + 0 * -1 + 0 * 1 + 0 * 0 + 156 * 1 + 155 * -1 + 0 * 0 + 153 * 1 + 154 * 1 = 308
multiply each number in the top red square with the number in the same position in the bottom square. We then proceed to do this for all channels (complexity of input, 3 for rgb), which *will then be combined to 1 single output value*.
This output value will then also be combined with a *bias*.
The entire reason we do this, is so that we can have an easier time calculating the images with a pc.

2.4 Reasons for convolution

- Features can be detected independent of location -> filters will always find what they are supposed/created to find
- This calculation is done in parallel, which is very fast for gpus when doing matrix calculations! -> Hence the use of tensorflow with cuda!
- Shared weights mean using the same *kernel values*, this reduces the use of a singular value for each neuron. -> more processing etc

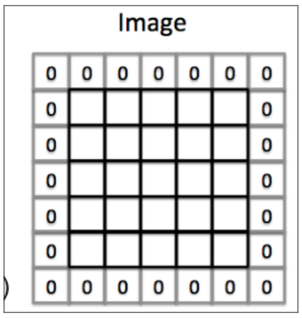
2.5 Stride



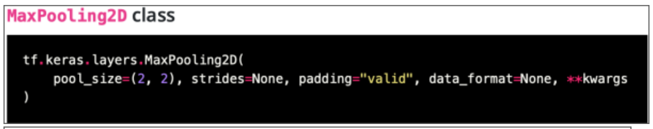
Stride is simply the offset by which we move towards the right and the bottom when we move to the next calculation.
The default value here is 1, which means Stride 1.

2.6 Padding

Padding is simply the edge of the matrix that has been predefined.
A same padding is made entirely of 0's!
Note, with padding, we can effectively go over the bounds of the matrix!

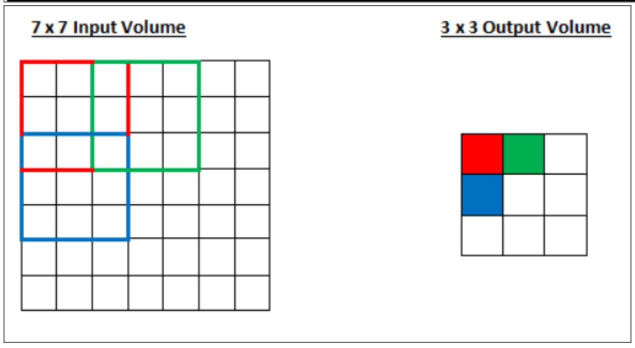


2.7 Max Pooling



MaxPooling2D class

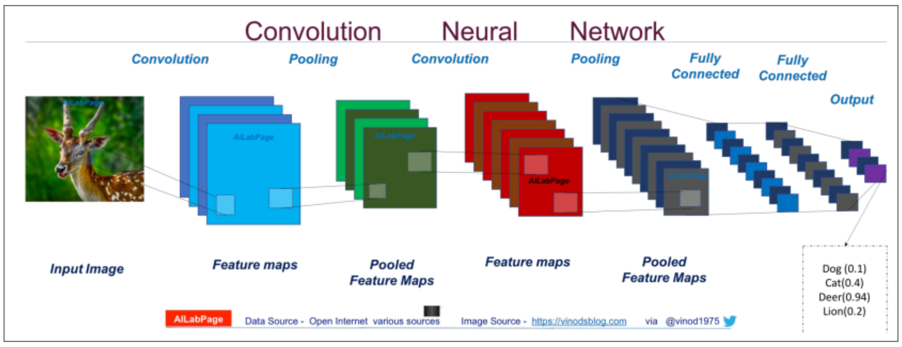
```
tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2), strides=None, padding="valid", data_format=None, **kwargs
)
```



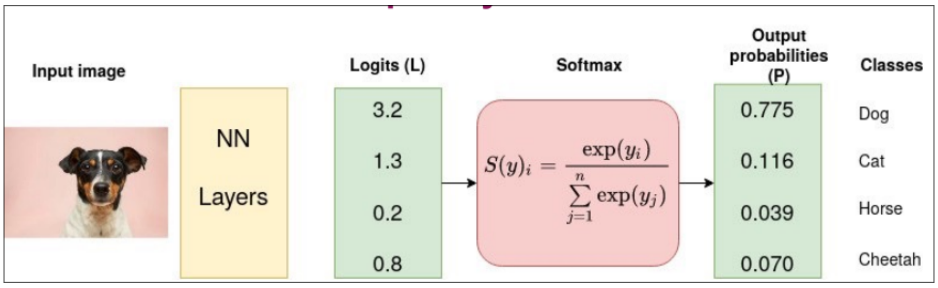
Max pooling simplyfies the image by taking the maximum value in each sector.
For example in the first red square, the highest value will be placed in the right red square.
Similar to before, we then iterate with the stride as the length to iterate and do this again and again until we are done.

2.8 Example with Keras

```
model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

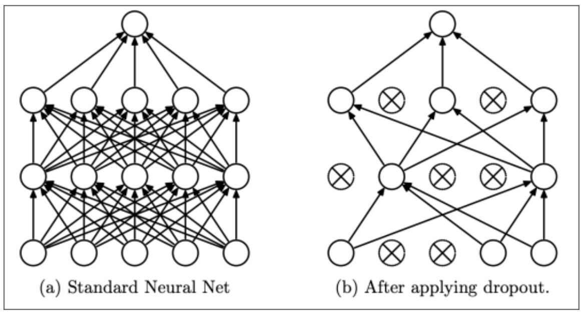


2.9 Softmax



Softmax is used to get the probabilities of each classification.

2.10 Drop Out



This is similar but not quite the same as ensemble method learning.
It is used to get models that are more robust than otherwise.
This entire ordeal is done during training, not during inference!

2.11 Why?

The entire filter is essentially just a shape, packed into a matrix.
If you for example have a filter of a 2x2 matrix and you only fill the bottom left and top right square with one and the others with 0, then you will detect whether or not the shape in question has a diagonal line inside of it.
Note that with the others being 0, you will *ignore* the rest of the shape around it. This means you will detect ALL diagonal lines, *even if they are within another shape.*.
Should you want to only detect diagonal lines that have no other color around it, then you need to make the other values in the filter matrix as negative, indicating that you only want the diagonal lines that are isolated.

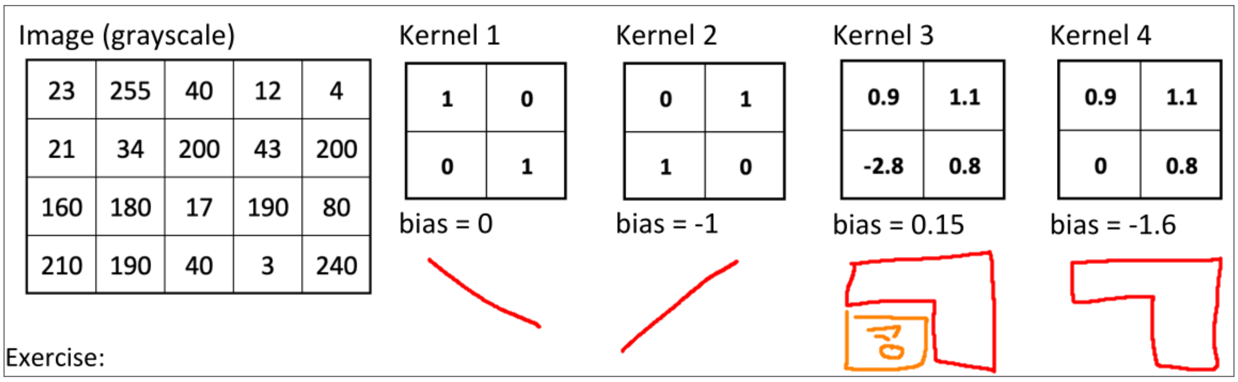


Image (grayscale)

23	255	40	12	4
21	34	200	43	200
160	180	17	190	80
210	190	40	3	240

Kernel 1

1	0
0	1

bias = 0

Kernel 2

0	1
1	0

bias = -1

Kernel 3

0.9	1.1
-2.8	0.8

bias = 0.15

Kernel 4

0.9	1.1
0	0.8

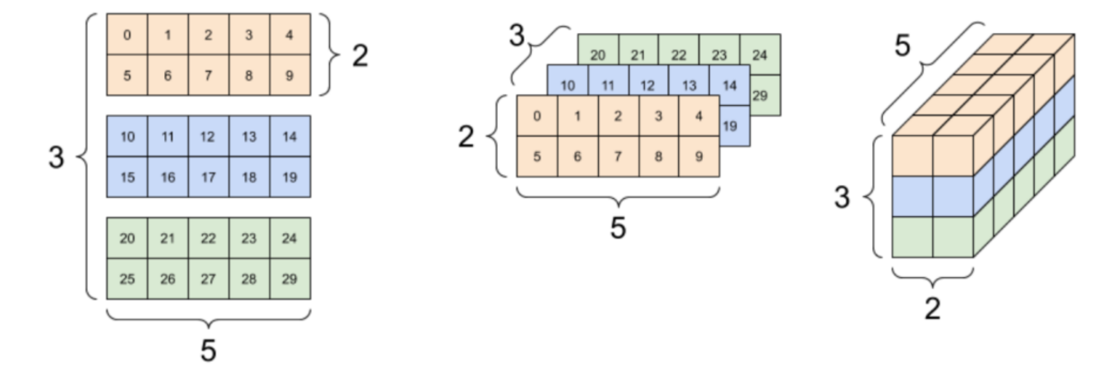
bias = -1.6

Exercise:

2.12 Tensor

A tensor is just a multidimensional array. The structure is as follows:

A 3-axis tensor, shape: [3, 2, 5]



In other words: *amount of sub-arrays, width of sub-array, length of sub-array*

Tensors are immutable, meaning you can only create new ones, you can't update them!

Terms:

- Rank
  The amount of parameters, in the example we had rank3, but what if the sub-arrays had a height as well? -> rank 4!
- Axis or Dimension
  a particular dimension of a tensor, remember 3 dimensions = rank3
  A rank-4 tensor, shape: [3, 2, 4, 5]
- Shape
  The length (number of elements) of each of the axes of a tensor.
- Size
  The total number of items in the tensor, the product of the shape vector's elements.
- Indexing
  simply the indexing of the array like in python.

```
rank_1_tensor = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
print(rank_1_tensor.numpy())
print("First:", rank_1_tensor[0].numpy())
print("Second:", rank_1_tensor[1].numpy())
print("Last:", rank_1_tensor[-1].numpy())
# First: 0
# Second: 1
# Last: 34
```

- Reshaping
  reshapes a vector by a given list. -> shape is a list!
  Remember that the order and amount of axis need to match! Otherwise you get trash!

```
# Shape returns a 'TensorShape' object that shows the size along each axis
x = tf.constant([[1], [2], [3]])
print(x.shape)
# You can reshape a tensor to a new shape.
# Note that you're passing in a list
reshaped = tf.reshape(x, [1, 3])
print(x.shape)
print(reshaped.shape)
# (3, 1)
# (1, 3)
```

- Broadcasting
  When tensors aren't the same size, you can essentially default extend them, in this case the last value will be taken.

```
x = tf.constant([1, 2, 3])
y = tf.constant(2)
z = tf.constant([2, 2, 2])
# All of these are the same computation
print(tf.multiply(x, 2))
print(x * y)
print(x * z)
# tf.Tensor([2 4 6], shape=(3,), dtype=int32)
# tf.Tensor([2 4 6], shape=(3,), dtype=int32)
# tf.Tensor([2 4 6], shape=(3,), dtype=int32)
```

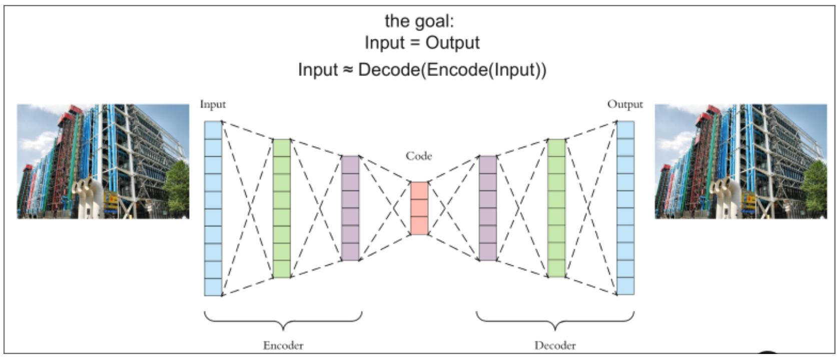
code tutorial

2.13 An example for image classification

```python
# import necessary modules
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

# show pictures
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()

# create convolutional base
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Add dense layers on top
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))

# Compile and train the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Evaluate the model
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

# print the accuracy at the end
print(test_acc)
```

full tutorial

3 Encoder and Decoder

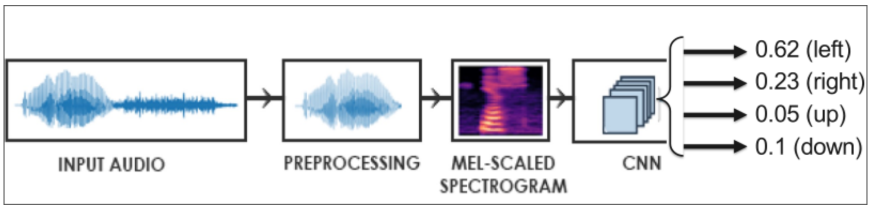


The idea here is that we can compress the data from the "noisy" picture into it's base shapes.
While this means we will lose the detail, we will also lose the "noise", that came along with the picture.
Base ideas:

- Bottleneck layer -> force the image into smaller layers
- code in the middle is *a feature vector*
- Base idea: if we know the structure, we do not need to know every pixel!
- Unsupervised learning, there are no labels!
- transfer learning: reuse part of the network for other tasks

Tutorial for Encoder

3.0.1 Audio transformation

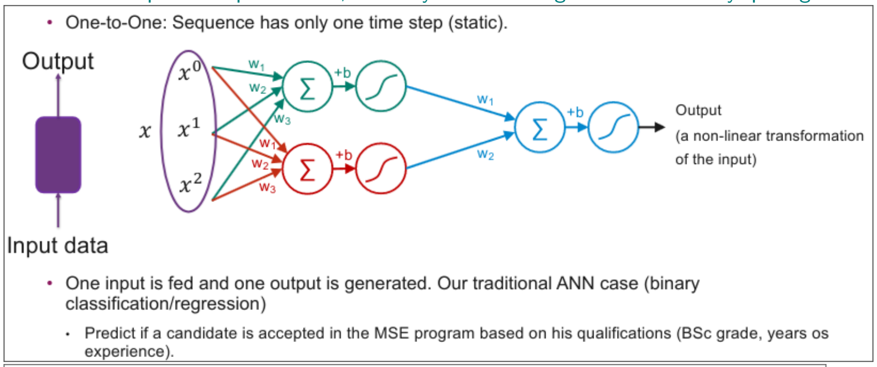


The idea here is that you take small portions of audio (11ms or something similar), then you turn the frequency table. This gives you the "modified" output on the right.

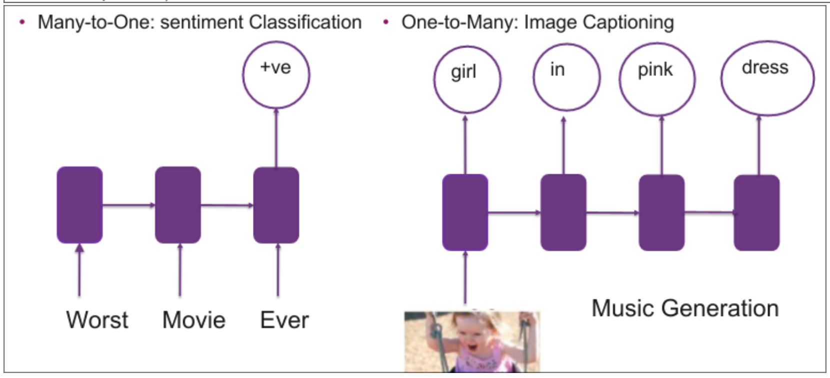
4 RNN

4.1 Sequential Data

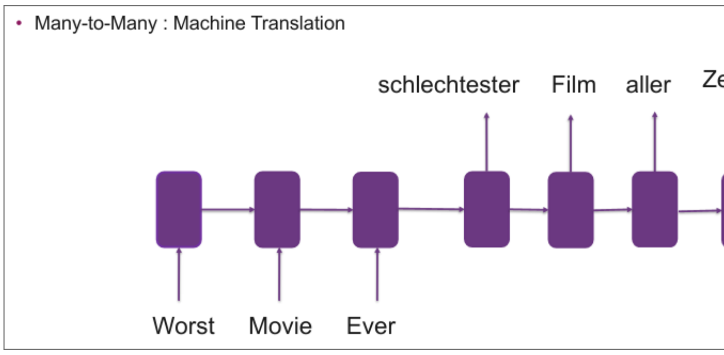
This is the 1 input 1 output criteria, basically the one thing haskell is actually quite good at!



- One-to-One: Sequence has only one time step (static).
- One input is fed and one output is generated. Our traditional ANN case (binary classification/regression)
  - Predict if a candidate is accepted in the MSE program based on his qualifications (BSc grade, years os experience).



- Many-to-One: sentiment Classification
- One-to-Many: Image Captioning

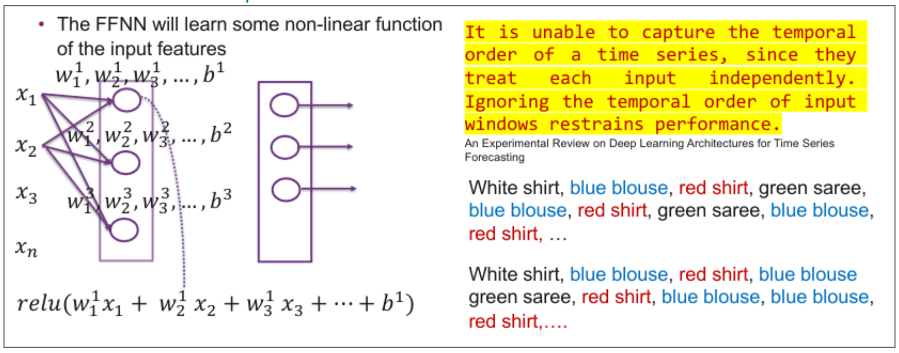


- Many-to-Many : Machine Translation

These are all fine, but the limitation is that we do not have a memory. So the usefulness is limited at best.

4.2 Dealing with memory

When you need to remember about previous states, then you can't use ANN or CNN.
For regular ANNs the problem is that they add things together and forget about the initial state.
Take this FFNN for example:



- The FFNN will learn some non-linear function of the input features

$w_1^1, w_2^1, w_3^1, \ldots, b^1$

$w_1^2, w_2^2, w_3^2, \ldots, b^2$

$w_1^3, w_2^3, w_3^3, \ldots, b^3$

$relu(w_1^1 x_1 + w_2^1 x_2 + w_3^1 x_3 + \cdots + b^1)$

`It is unable to capture the temporal order of a time series, since they treat each input independently. Ignoring the temporal order of input windows restrains performance.`

An Experimental Review on Deep Learning Architectures for Time Series Forecasting

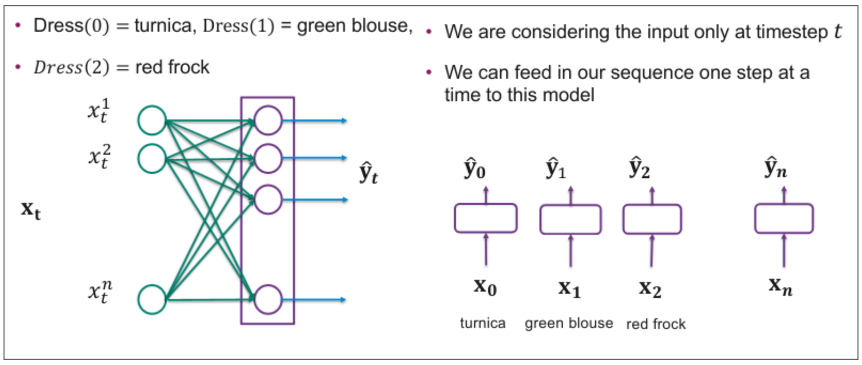
White shirt, blue blouse, red shirt, green saree, blue blouse, red shirt, green saree, blue blouse, red shirt, …

White shirt, blue blouse, red shirt, blue blouse green saree, red shirt, blue blouse, blue blouse, red shirt,….

The problem here is that the FFNN adds together the inputs, meaning that you will not remember what the inputs were at the start.
So how about CNN?
For CNN the problem is that you only see whether or not somehting was/is there with a clear information loss.
This means you can't remember the exact input, you only know, input class x was there!

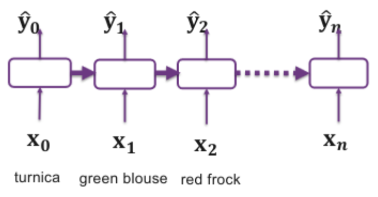
4.3 The Solution

4.3.1 How do we handle the time component?



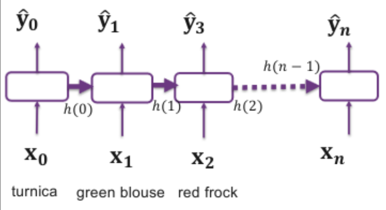
- $\mathrm{Dress}(0)$ = turnica, $\mathrm{Dress}(1)$ = green blouse,
- $Dress(2)$ = red frock
- We are considering the input only at timestep t
- We can feed in our sequence one step at a time to this model

4.3.2 Adding Connections

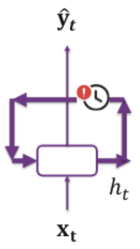


- We feed in the sequence
- $\hat{y}_2$ is the prediction for tomorrow
- But $\hat{y}_2$ is disconnected from x_0 , x_1 !
  - This is nothing but FFNN
- Remember we are modeling sequences (time series) that have dependence on other parts of the time series (past or future)

4.3.3 Recurrence

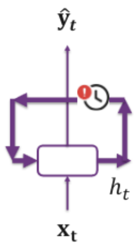


$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_3$ $\hat{y}_n$
$h(0)$ $h(1)$ $h(2)$ $h(n-1)$
x_0 x_1 x_2 x_n
turnica
green blouse
red frock



- Imagine there is a delay/lag of 1 timestep on the feedback line.
- Some part of the computation is retained (memory) and fed to the next computation
- What is this feedback h_t ?
- What is happening inside the layer, i.e,. in ▭ ?
- The neurons we saw so far had only 1 output $\hat{y}$

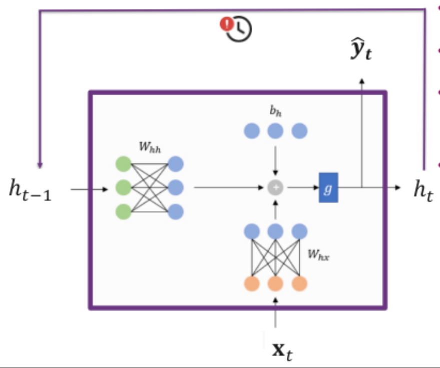
4.3.4 Memory Cell



$$x_0 = tunica, h_{-1}, y_0, h_0$$
$$x_1 = green\ blouse, h_0, y_1, h_1$$
$$x_2 = red\ frock, h_1, y_2, h_2$$

$$\boxed{h_t = f(x_t, h_{t-1})}$$
Notice that f is parameterised by weights and bias which need to be learned. These are shared across all timesteps.

4.3.5 simpleRNNCell (keras)

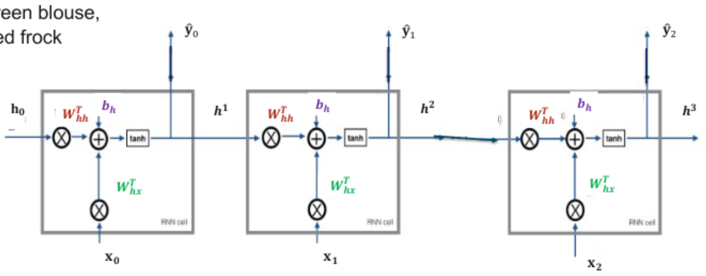


- $h_t = f(x_t, h_{t-1})$
- $h_t = g(W_{hh}^T h_{t-1} + W_{hx}^T x_t + b_h)$
- *g is a nonlinear activation function*
  - *Hyperblic tangent* $g(\theta) = \tanh(\theta)$
- $\hat{y}_t = h_t$

Mohit Mayank

4.3.6 Conputations across time/Forward Propagation

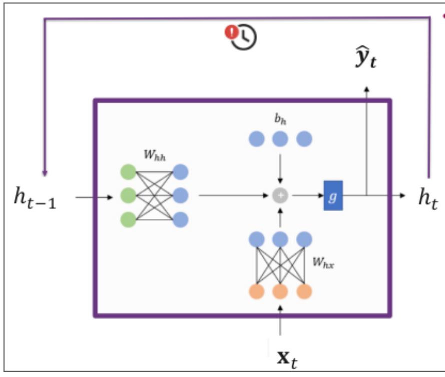
x_0 = turnica,
x_1 = green blouse,
x_2 = red frock



$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$
h_0 h^1 h^2 h^3
W_{hh}^T b_h tanh W_{hx}^T
RNN cell
x_0 x_1 x_2

Same weights are used in each cell

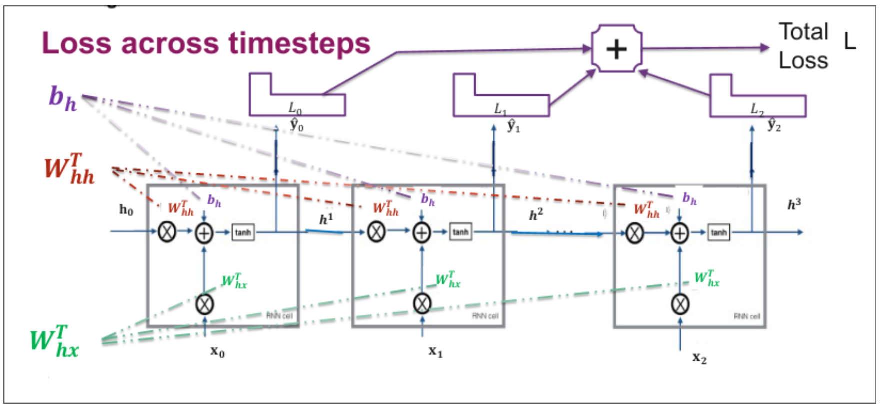
4.3.7 How to train RNN?



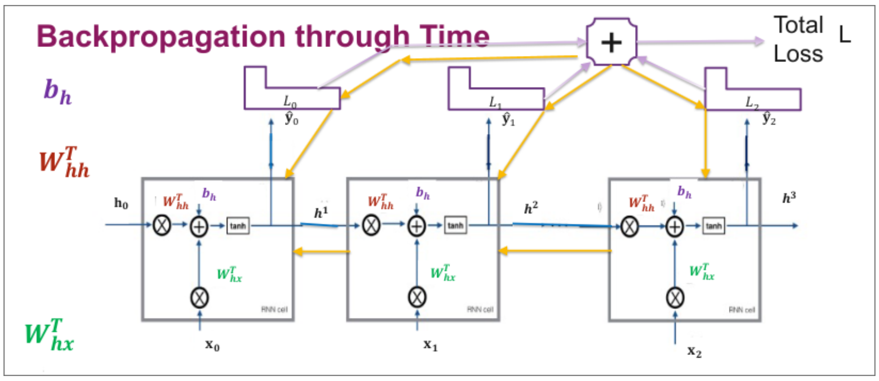
- *We have not yet defined some of the basic ingredients of DNN modeling yet*
  - *Inputs*
  - *Outputs*
  - *Loss ??*
  - *Optimization procedure ??*

Mohit Mayank

4.3.8 Loss across time steps



4.3.9 Backpropagation through Time


Backpropagation through Time
+
Total Loss
L
b_h
W^T_{hh}
W^T_{hx}
L_0
$\hat{y}_0$
L_1
$\hat{y}_1$
L_2
$\hat{y}_2$
h_0
h^1
h^2
h^3
tanh
RNN cell
x_0
x_1
x_2