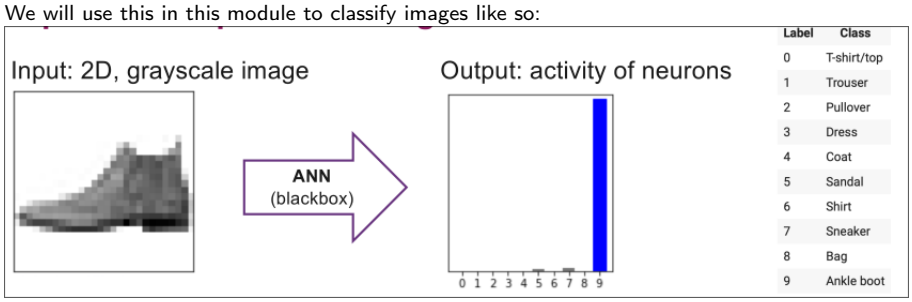
0 Contents

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

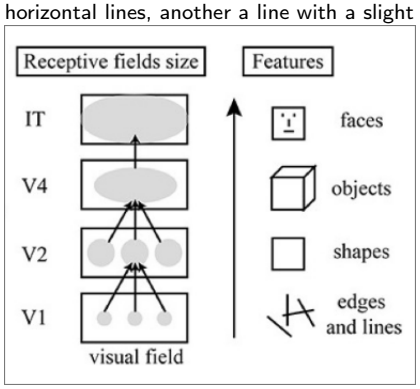
1.2 Flattening

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.

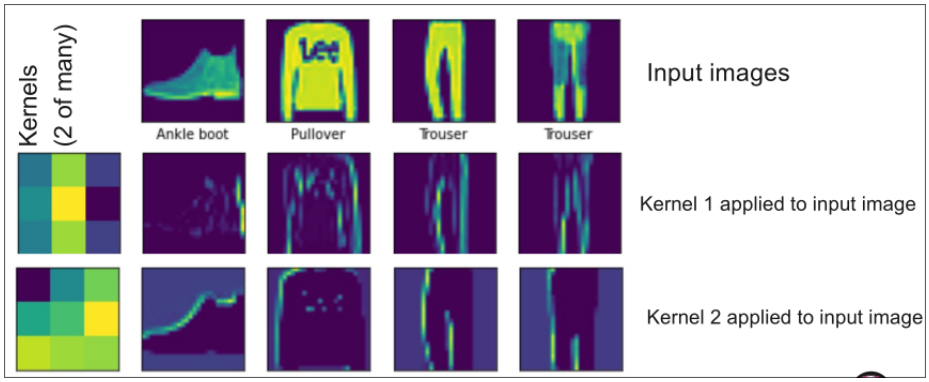


2.2 mathematical model of a feature detector

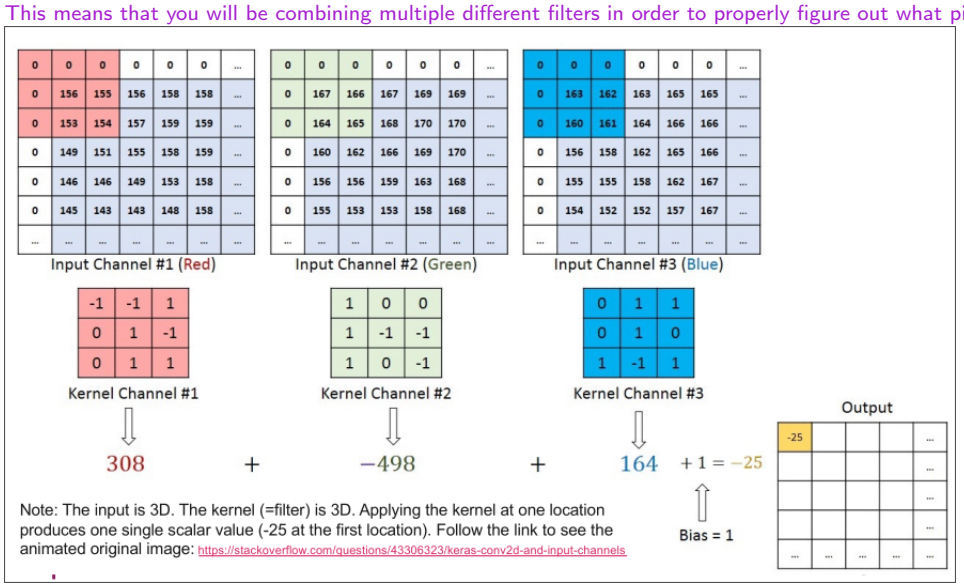
- Two Inputs
  - a picture
    Note that rgb would give you 3 channels red, green and blue
  - A filter(kernel)
    * an m by n matrix in the simplest case (1 channel, grayscale).
    * an m by n x 3 "stack of matrices" in the case of a 3 channel input (e.g. an RGB image)
    * an m by n x d "stack of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



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.

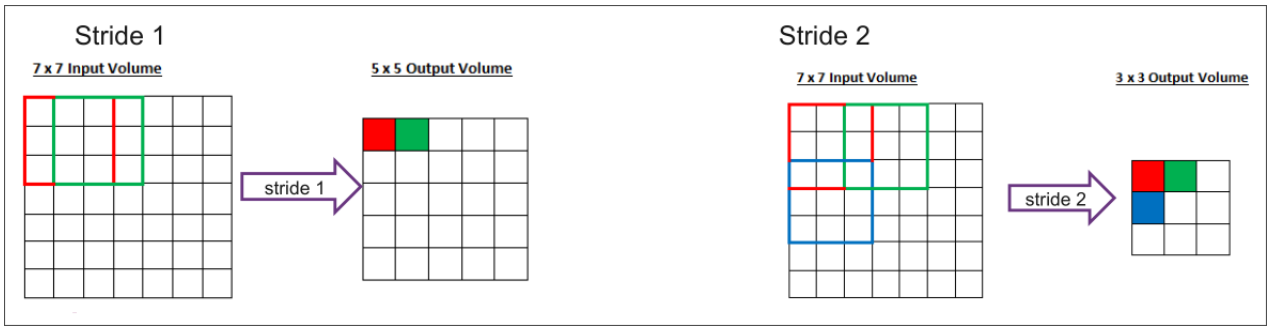


Explanation for the first calculation: 0 * -1 + 0 * -1 + 0 * 1 + 0 * 0 + 156 * 1 + 155 * -1 + 0 * 0 + 153 * 1 + 154 * 1 = 308
multiply each number in the top red square with the number in the same position in the bottom square. We then proceed to do this for all channels (complexity of input, 3 for rgb), 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

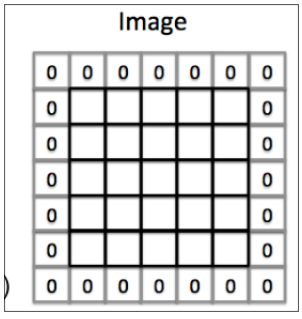
2.5 Stride



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

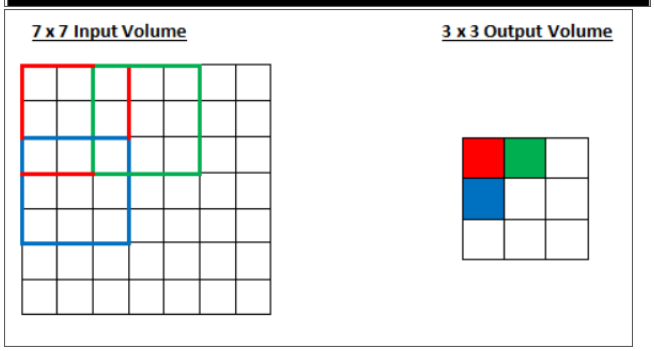
Padding is simply the edge of the matrix that has been predefined.
A same padding is made entirely of 0's!
Note, with padding, we can effectively go over the bounds of the matrix!



2.7 Max Pooling

MaxPooling2D class

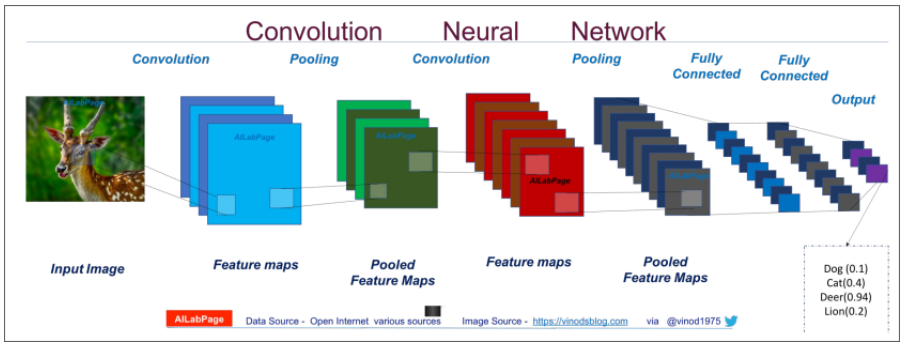
```
tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2), strides=None, padding="valid", data_format=None, **kwargs
)
```



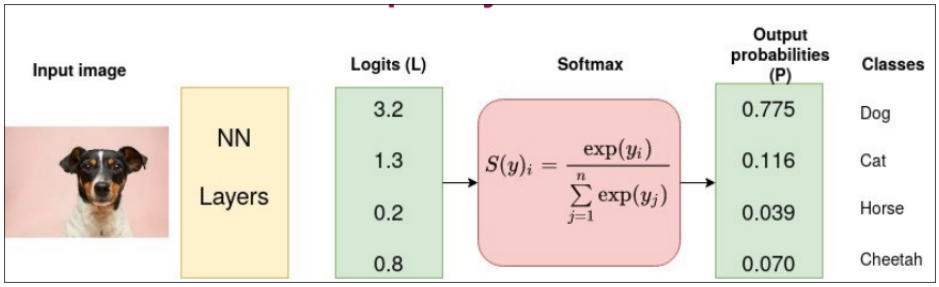
Max pooling simplyfies the image by taking the maximum value in each sector.
For example in the first red square, the highest value will be placed in the right red square.
Similar to before, we then iterate with the stride as the length to iterate and do this again and again until we are done.

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)
])
```



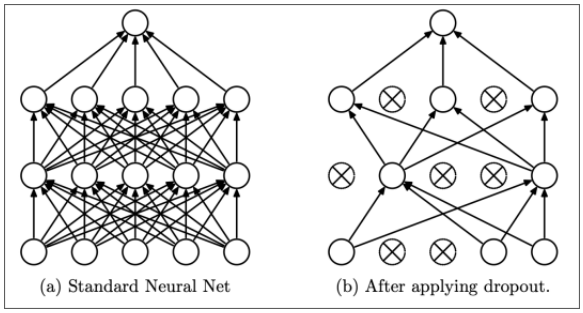
2.9 Softmax



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

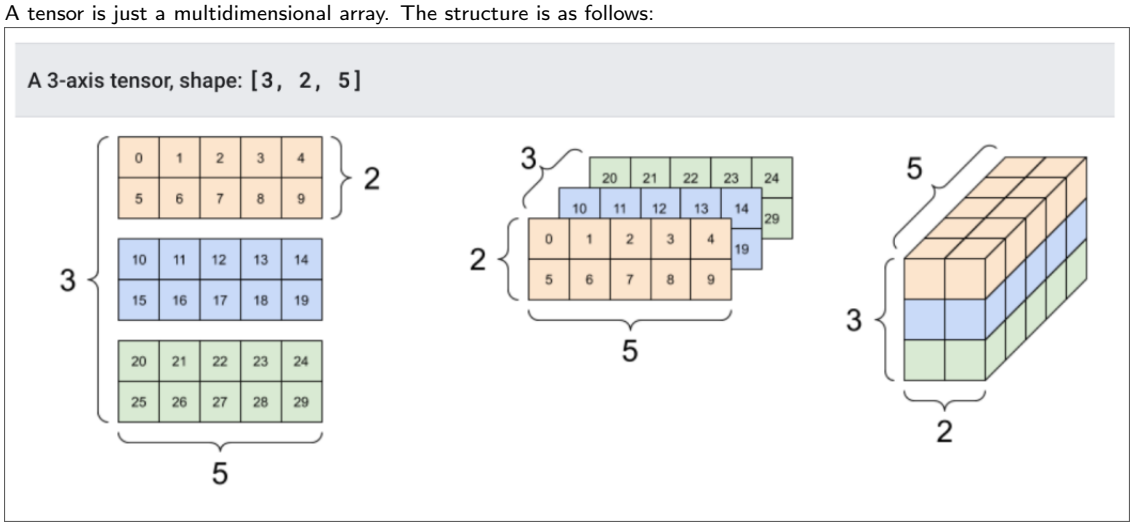
0.9	1.1
0	0.8

bias = -1.6

Exercise:

2.12 Tensor

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



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)
```

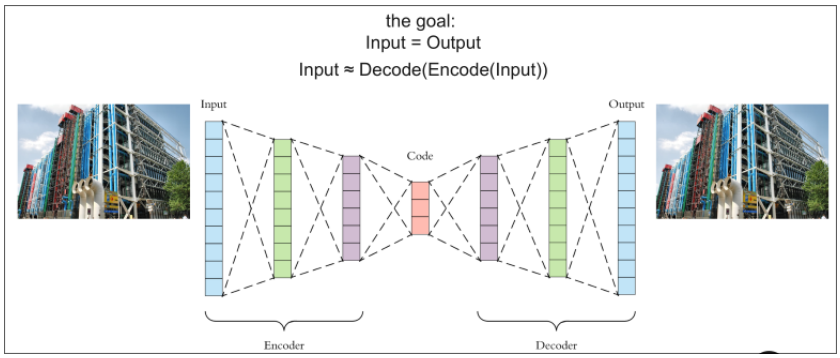
code tutorial

2.13 An example for image classification

```
# 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

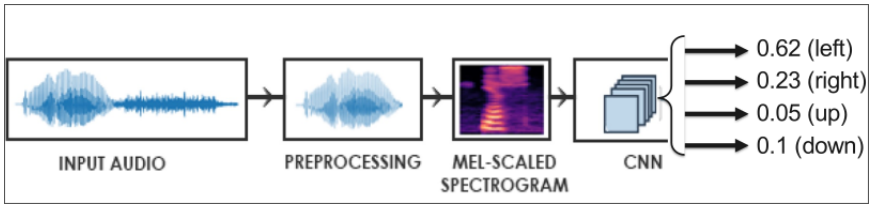


The idea here is that we can compress the data from the "noisy" picture into it's base shapes.
While this means we will lose the detail, we will also lose the "noise", that came along with the picture.
Base ideas:

- 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

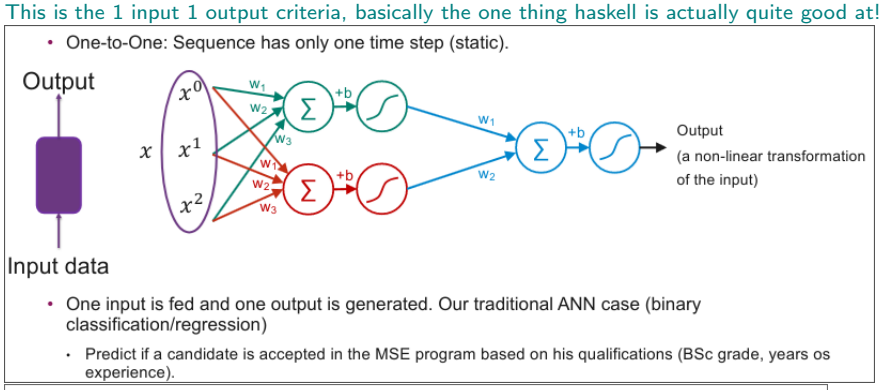


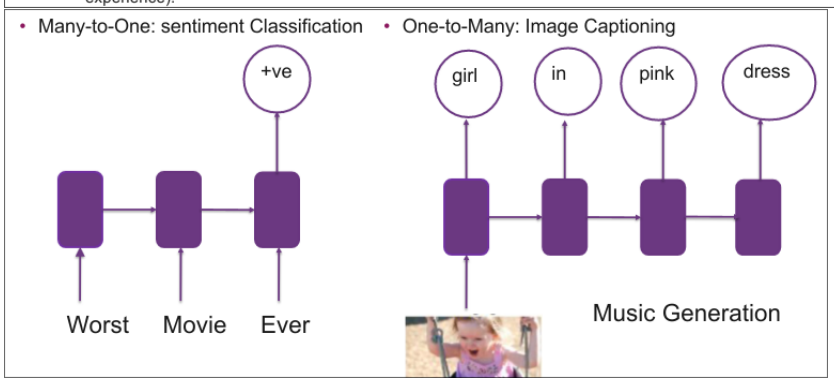
The idea here is that you take small portions of audio (11ms or something similar), then you turn the frequency table. This gives you the "modified" output on the right.

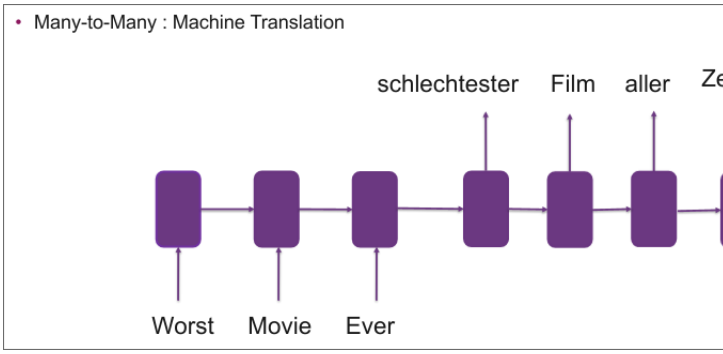
4 RNN

4.1 Sequential Data

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



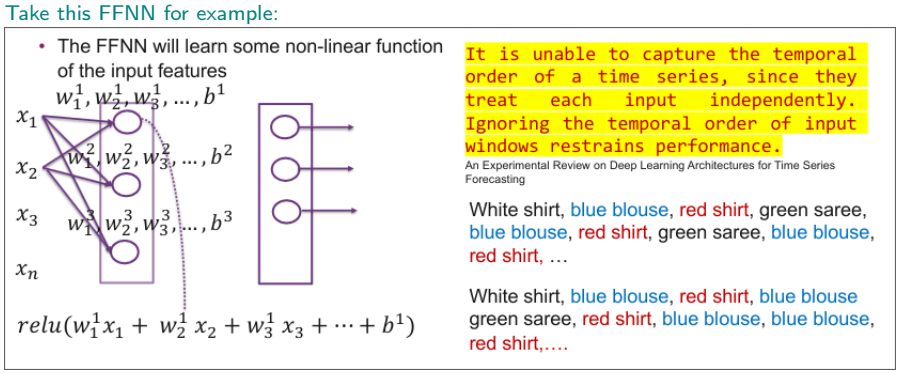




These are all fine, but the limitation is that we do not have a memory. So the usefulness is limited at best.

4.2 Dealing with memory

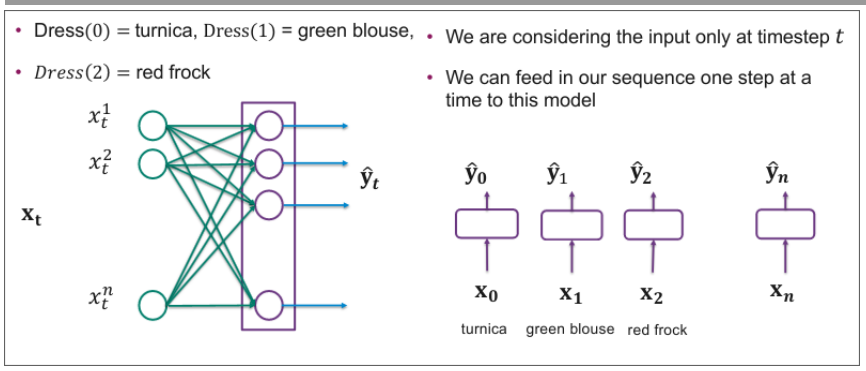
When you need to remember about previous states, then you can't use ANN or CNN.
For regular ANNs the problem is that they add things together and forget about the initial state.
Take this FFNN for example:



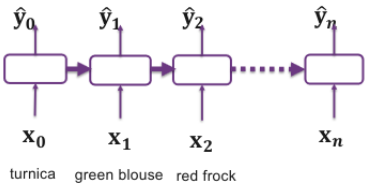
The problem here is that the FFNN adds together the inputs, meaning that you will not remember what the inputs were at the start.
So how about CNN?
For CNN the problem is that you only see whether or not somehting was/is there with a clear information loss.
This means you can't remember the exact input,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

4.3 The Solution

4.3.1 How do we handle the time component?

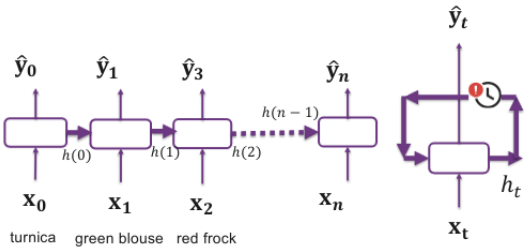


4.3.2 Adding Connections



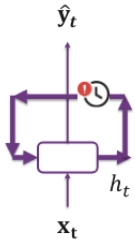
- We feed in the sequence
- $\hat{y}_2$ is the prediction for tomorrow
- But $\hat{y}_2$ is disconnected from x_0 , x_1 !
  - This is nothing but FFNN
- Remember we are modeling sequences (time series) that have dependence on other parts of the time series (past or future)

4.3.3 Recurrence



- 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

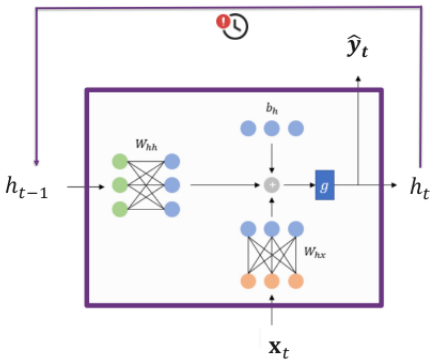


$$x_0 = tunica, h_{-1}, y_0, h_0$$
$$x_1 = green\ blouse, h_0, y_1, h_1$$
$$x_2 = red\ frock, h_1, y_2, h_2$$

$$h_t = f(x_t, h_{t-1})$$

Notice that f is parameterised by weights and bias which need to be learned. These are shared across all timesteps.

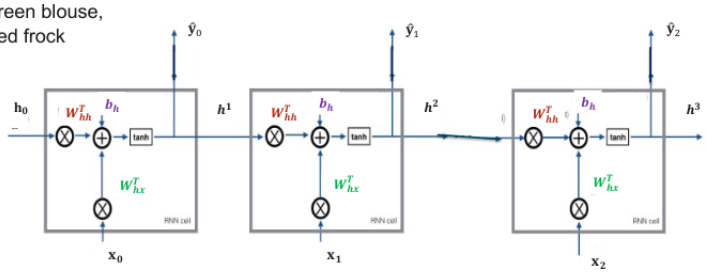
4.3.5 simpleRNNCell (keras)



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

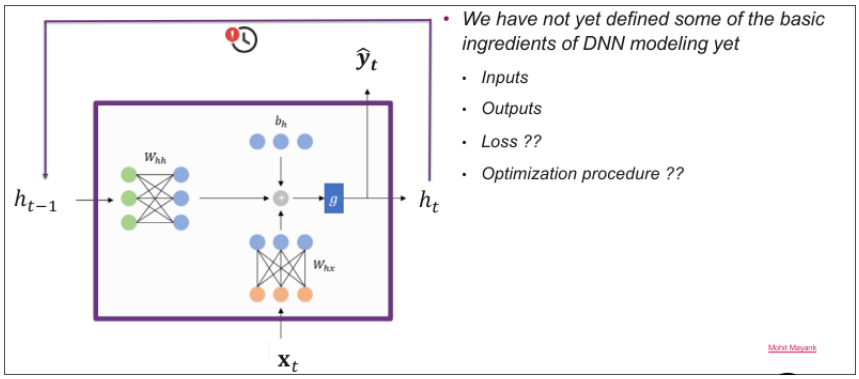
4.3.6 Conputations across time/Forward Propagation

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



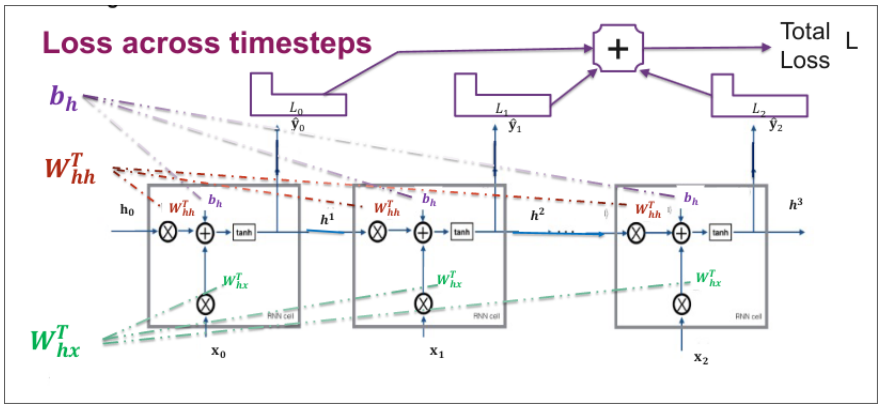
Same weights are used in each cell

4.3.7 How to train RNN?

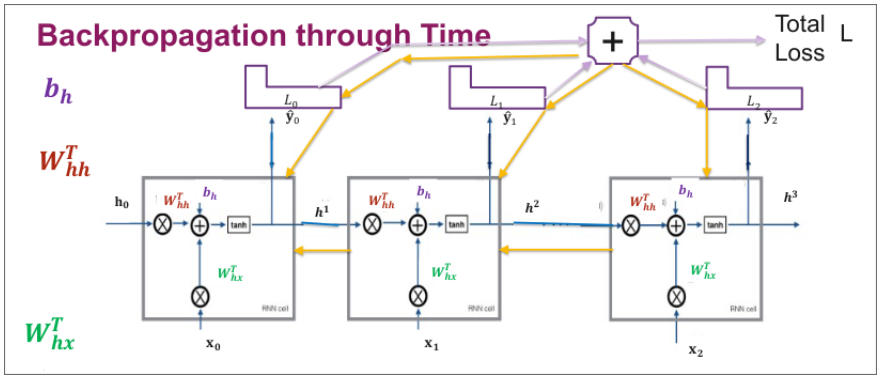


- *We have not yet defined some of the basic ingredients of DNN modeling yet*
  - *Inputs*
  - *Outputs*
  - *Loss ??*
  - *Optimization procedure ??*

4.3.8 Loss across time steps

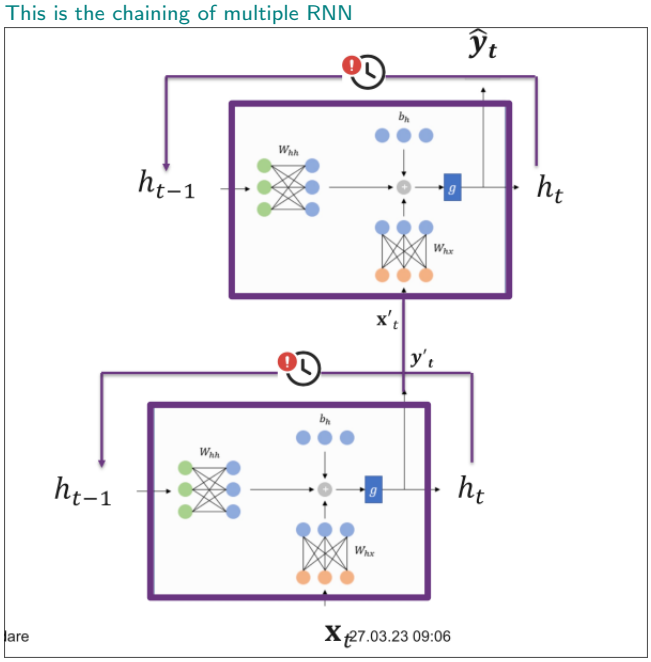


4.3.9 Backpropagation through Time

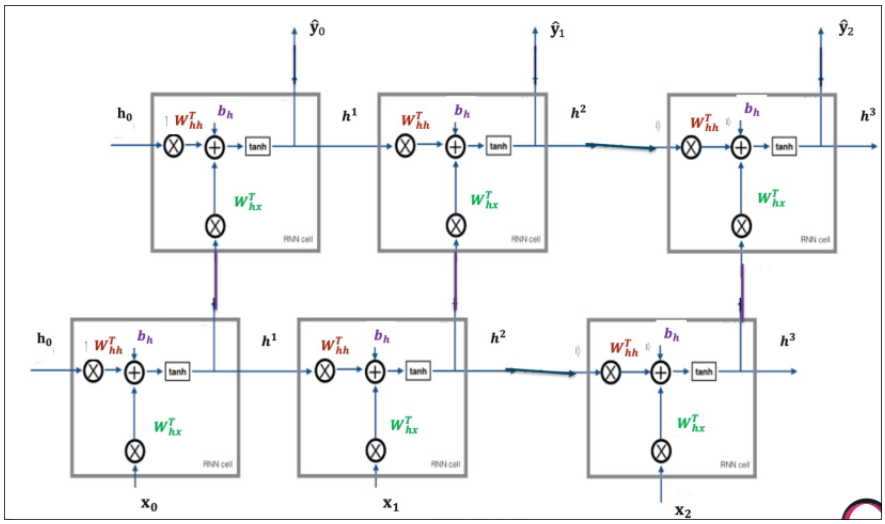


4.4 Deep RNN

This is the chaining of multiple RNN



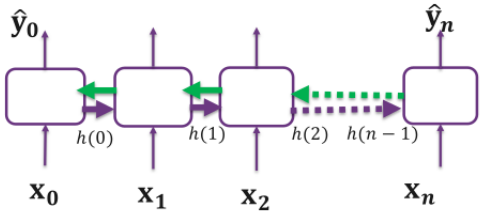
4.4.1 Unroll

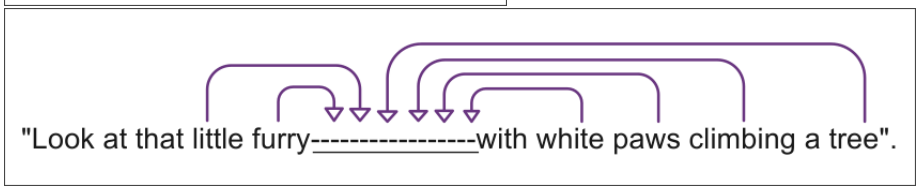


4.4.2 Bi-Directional RNN

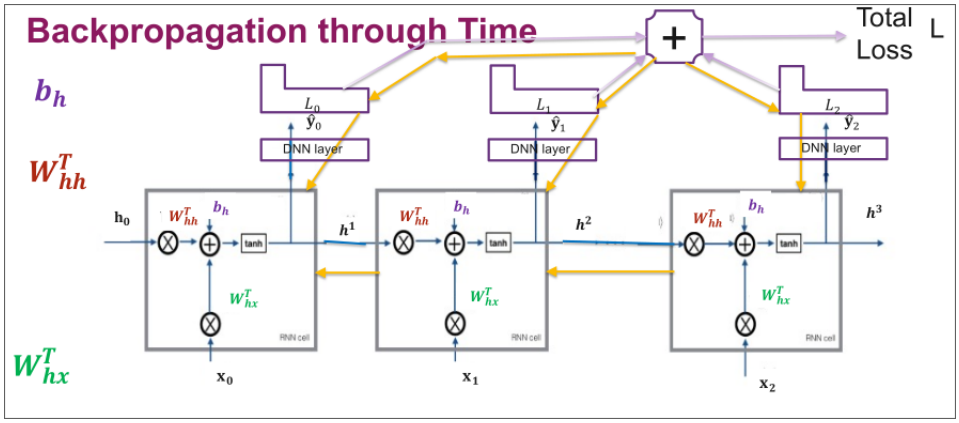
This is an RNN that can propagate from both sides, meaning you can figure out what is necessary in the middle.

- $h_t = f^f(\mathbf{x}_t, h_{t-1})$
  - $h_t = g(W_{hh}^T h_{t-1} + W_{hx}^T \mathbf{x_t} + b_h)$
- $j_t = f^b(\mathbf{x_t}, j_{t+1})$
  - $j_t = g(W_{jj}^T j_{t+1} + W_{jx}^T \mathbf{x_t} + b_j)$
- $\hat{\mathbf{y}}_t = some\ function(h_t, j_t)$





4.4.3 Backpropagation through time BPTT



4.4.4 Vanishing gradient of simple/vanilla RNNs

$$W_{hx} = W_{hx} - learning\_rate * \frac{\partial L}{\partial W_{hx}}$$

$$L = L_0 + L_1 + L_2$$

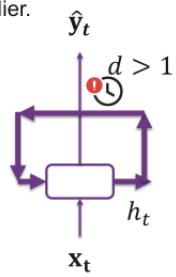
$$\frac{\partial L_2}{\partial W_{hx}} = \frac{\partial L_2}{\partial W_{y_3}} \frac{\partial y_2}{\partial h_3} \frac{\partial h_3}{\partial W_{hx}} + \frac{\partial L_2}{\partial W_{y_3}} \frac{\partial y_3}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial W_{hx}} + \frac{\partial L_2}{\partial W_{y_3}} \frac{\partial y_3}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W_{hx}}$$

4.4.5 Solutions to the problem of backpropagation

It gets harder and harder to pass the loss back to time... so wee need a soltion:

skip connections

- Adding skip connections through time
- Follows from the idea of incorporating delays
- A unit can always learn to be influenced by a value from **d** time steps earlier.



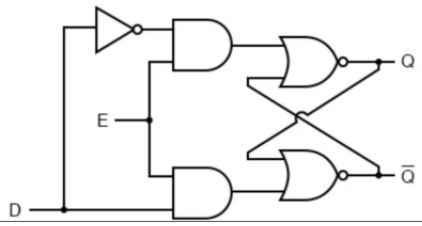
Leaky Units

- Leaky Units
  - In general RNN, we have $h_t = f(\mathbf{x}_t, h_{t-1})$
  - $h_t = \alpha\, h_{t-1} + (1-\alpha)x_t$
- These are like skip connections
- The effect is more smooth and flexible via real-values α rather than adjusting the d (integer value)
- These allow the network to accumulate information over a long duration

AI Applications FS23, M. Purandare 27.03.23 09:06

Motivation: LSTM/GRUs

- Control information that is passed to the next step
- Use more complex recurrent unit that has gates to control
- If anyone has an electrotechnics background, this is like a gated latch.
  - The latch has an enable signal that control what gets latched.



Q(n+1) = EN.D + EN'.Q(n)

E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

4.4.6 Efficient Solution: Gated RNNs

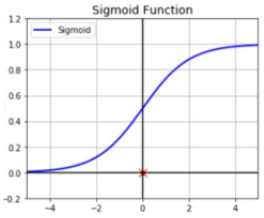
- Include long and short-term memory LSTM, and gated recurrent unit GRU models
- Generalized versions of leaky units
  Connection weights e.g. α may change at each time step
- Addition to leaky units
  Once the information is used, neural network can forget the old state
- Hot research field..

4.4.7 LSTM

LSTM

- LSTMs have similar control flow like RNNs
- LSTMs have blocks that control the information flow.
  - They have more parameters and gating units to build these blocks.
- They are able to track information/history through many time steps.
- Information is **forgotten** or **remembered** via structures called **gates.**
- **LSTM** cell performs the following specialized

- Gate



- Gates contain sigmoid activation
- Helpful to forget or update information
- The network can learn which important data to keep and which to forget

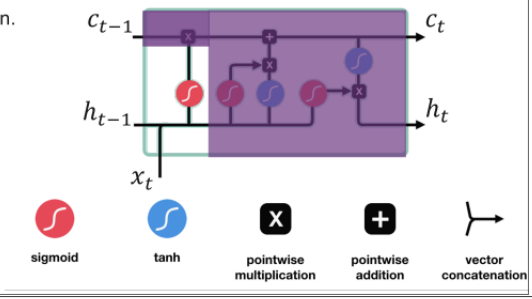
Gates

- In addition to the recurrent feedback (h_{t-1}), LSTMs have a cell state.
- LSTMs use gates to manipulate the cell state using following steps:
  1. **Forget**
  2. **Store/Compute/Input**
  3. **Cell/Update**
  4. **Output**
- Each step corresponds to a gate

Forget gate

1. They forget irrelevant parts of the previous states
2. $h_{t-1}$and $x_t$are passed through the sigmoid.
3. The o/p of the sigmoid is always between 0 and 1.
4. The values close to 0 are said to be forgotten.

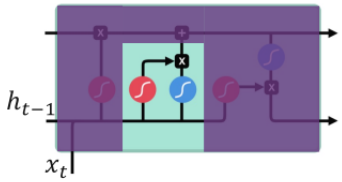
$$f_t = \sigma\left(W_{fh} \cdot h^{t-1} + W_{fx} \cdot x^t + b_f\right)$$



Input Gate

- **Compute/Store Gate**
  - Computes the new cell state
  - tanh is non-linear activation
  - Sigmoid can be thought of as the selector to choose what part of tanh is to be forgotten or which part is relevant.

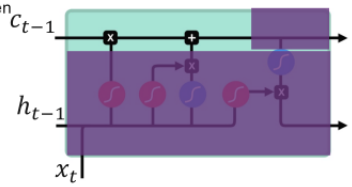
$$i_t = \sigma\left(W_{ih} \cdot h^{t-1} + W_{ix} \cdot x^t + b_i\right)$$
$$\tilde{C}_t = \tanh\left(W_{Ch} \cdot h^{t-1} + W_{Cx} \cdot x^t + b_C\right)$$

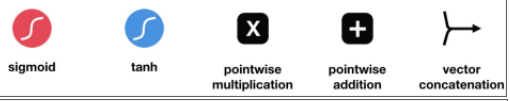


Update Gate

- The previous cell state is multiplied by the output of the forget gate
  - Again, sigmoid can be thought of as the selector
  - The previous cell state is multiplied by the output of the forget gate to choose what part of the previous cell state is to be forgotten
  - The multiplication result is further added to the output of the compute gate which preserves relevant part of the input

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

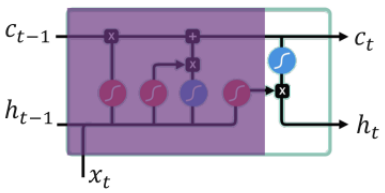




Output gate

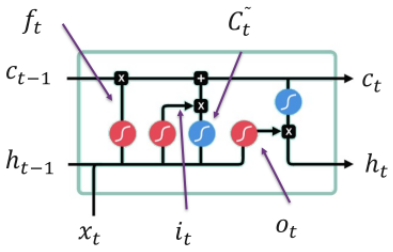
- **Output**
  - The result of the update gate is the new cell s
    - $c_t = c_{t-1}$
  - To compute h_t, $\tanh(c_t)$ and $\text{sigmoid}(h_{t-1}{:}x_t)$ are pointwise multiplied

$$o_t = \sigma\left(W_{oh} \cdot h^{t-1} + W_{ox} \cdot x^t + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$



LSTM

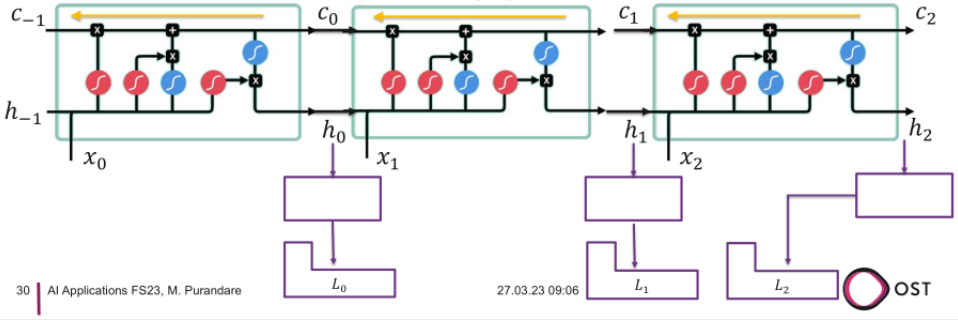
$$f_t = \sigma\left(W_{fh} \cdot h^{t-1} + W_{fx} \cdot x^t + b_f\right)$$
$$i_t = \sigma\left(W_{ih} \cdot h^{t-1} + W_{ix} \cdot x^t + b_i\right)$$
$$\tilde{C}_t = \tanh\left(W_{Ch} \cdot h^{t-1} + W_{Cx} \cdot x^t + b_C\right)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_{oh} \cdot h^{t-1} + W_{ox} \cdot x^t + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$



2023_03_27_02_37_06.png

Gradient Computation

- Interrupted gradient flow during back propagation through time
- Taking the derivatives, updating the drivatives according to loss, shifting the weights occurs respective to the cell state (c_t)
- Mitigate the vanishing gradient problem by creating paths through time that have derivatives that neither vanish nor explode. The forget gate enables that.



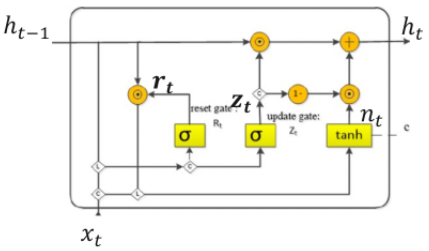
30 | AI Applications FS23, M. Purandare 27.03.23 09:06 OST

4.4.8 Gated Recurrent Units GRU

Gated Recurrent Units

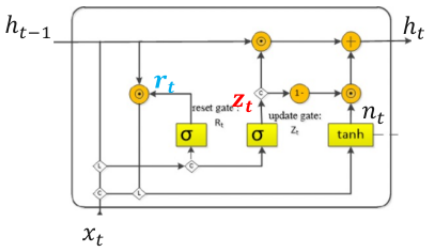
- LSTM downside
  - high number of parameters
- What pieces of LSTM are really necessary?
- GRU is another variant, quite similar to LSTM
- No cell state
- Contains two gates

  1. reset gate 2. Update gate



Gates

- Reset gate
  - Reset gates help capture short-term dependencies in sequences.
  - It is used to decide how much of the past information to forget.
- Update gate
  - Update gates help capture long-term dependencies in sequences.
  - It helps to decide how much of the past information to forget.



GRU: Reset gate

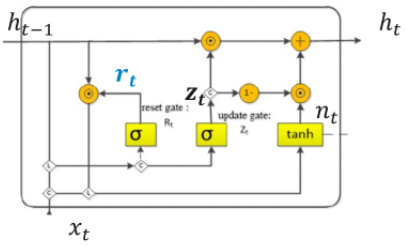
Reset gate is fairly similar to the forget gate of the LSTM cell. The resulting reset vector r represents the information that will determine what will be removed from the previous hidden time steps.

$$\mathbf{r}_t = \sigma(\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{b}_{ir} + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr}) \quad (1)$$

$$\mathbf{z}_t = \sigma(\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{b}_{iz} + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz}) \quad (2)$$

$$\mathbf{n}_t = tanh(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in} + \mathbf{r}_t \odot (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn})) \quad (3)$$

aisummer $\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{n}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1}$ (4)



GRU: Update gate

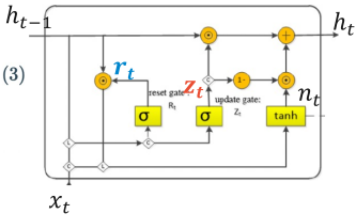
- **Update gate**
  - merges input x_t and $h_{t-1}$of the GRU

$$\mathbf{r}_t = \sigma(\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{b}_{ir} + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr}) \quad (1)$$

$$\mathbf{z}_t = \sigma(\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{b}_{iz} + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz}) \quad (2)$$

$$\mathbf{n}_t = tanh(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in} + \mathbf{r}_t \odot (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn})) \quad (3)$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{n}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1} \quad (4)$$



Output (almost)

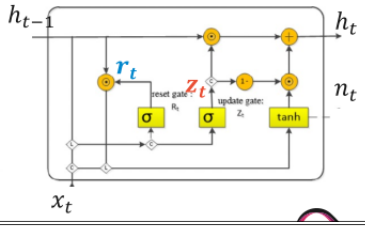
- The vector n consists of two parts
  - A linear layer applied to the input, similar to the input gate in an LSTM.
  - A reset applied to the previous hidden state. The reset **is applied directly in the hidden state**, instead of applying it in the intermediate representation of cell state **c** in case of LSTM.

$$\mathbf{r}_t = \sigma(\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{b}_{ir} + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr}) \quad (1)$$

$$\mathbf{z}_t = \sigma(\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{b}_{iz} + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz}) \quad (2)$$

$$\mathbf{n}_t = tanh(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in} + \mathbf{r}_t \odot (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn})) \quad (3)$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{n}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1} \quad (4)$$



GRU: Output

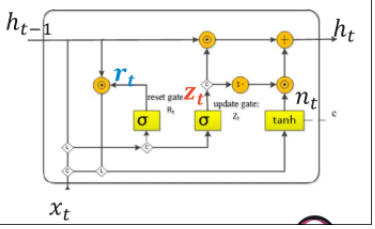
- Hidden State

$$\mathbf{r}_t = \sigma(\mathbf{W}_{ir}\mathbf{x}_t + \mathbf{b}_{ir} + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr}) \quad (1)$$

$$\mathbf{z}_t = \sigma(\mathbf{W}_{iz}\mathbf{x}_t + \mathbf{b}_{iz} + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz}) \quad (2)$$

$$\mathbf{n}_t = tanh(\mathbf{W}_{in}\mathbf{x}_t + \mathbf{b}_{in} + \mathbf{r}_t \odot (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn})) \quad (3)$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{n}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1} \quad (4)$$

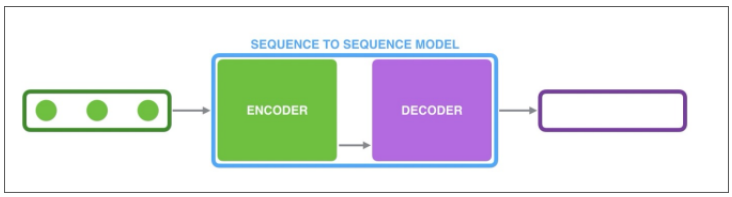


4.4.9 LSTM vs GRU

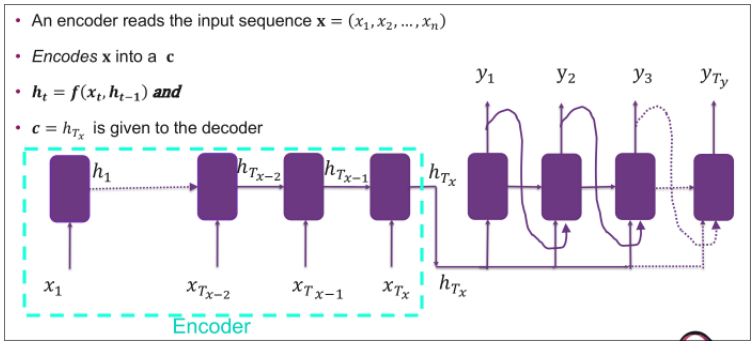
- LSTM is a bigger and more expensive model
- GRU is faster and compacter
- LSTM should only be used if you have a reason for it
- LSTM should remember longer sequences than GRU
- A GRU has one less gate than LSTM. Precisely, just a reset and update gates instead of the forget, input and output gate of LSTM.

4.5 Seq2seq

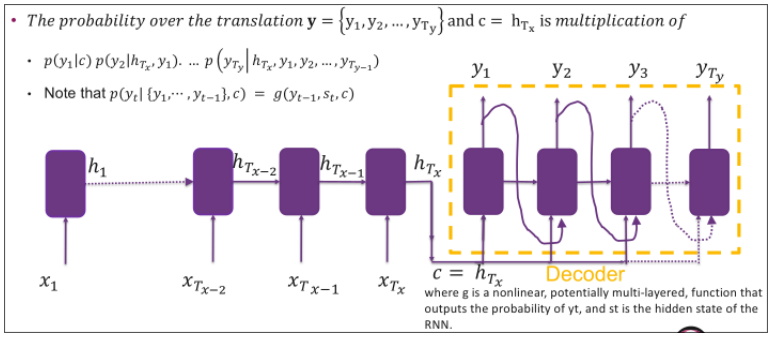
This is the split of an RNN cell into an encoder and a decoder part.
The encoder gives the sequence to the decoder as well as the *context*, which are the "important" parts of the input.
This results in less "memorizing" of old state, while not giving up on accuracy.



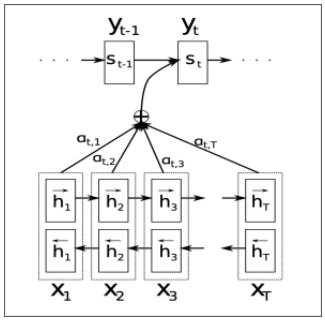
4.5.1 Encoder



4.5.2 Decoder



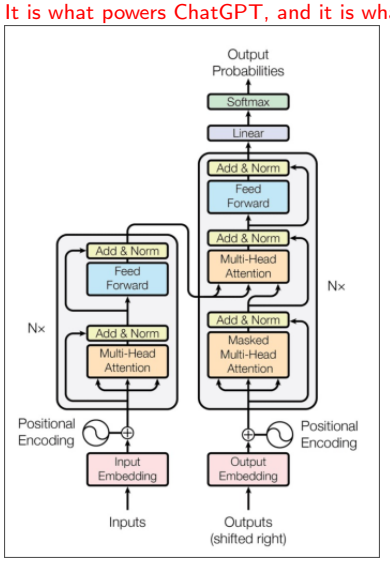
4.5.3 Problems



The biggest issue with this is the fact that this is not *parallelizeable*, it must perform each seuence in a serial manner

4.6 Transformers

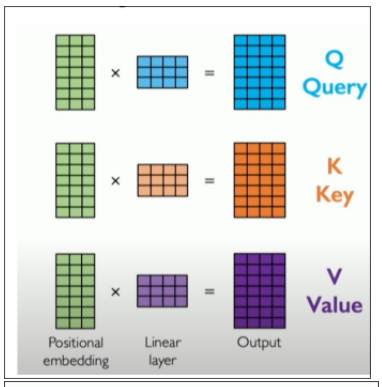
This is the solution to the seq2seq problem. Unlike the last model this is parallizeable.
It is what powers ChatGPT, and it is what makes the new generation of AI so powerful

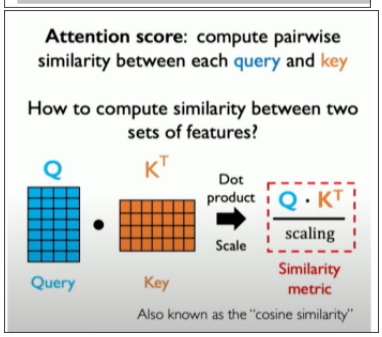


Note, *Transformers do NOT use any recurrence*

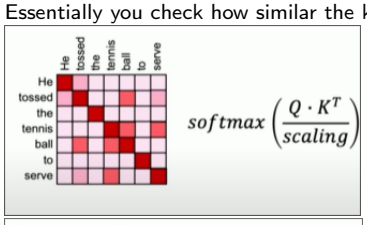
4.6.1 Attention

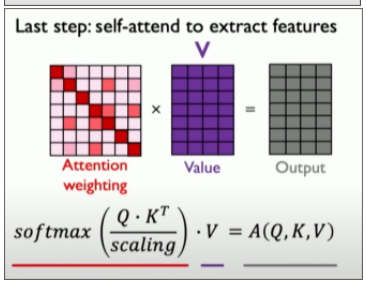
Transformers only really care about the concept of "attention", which is comprised of *Query, key, value*.



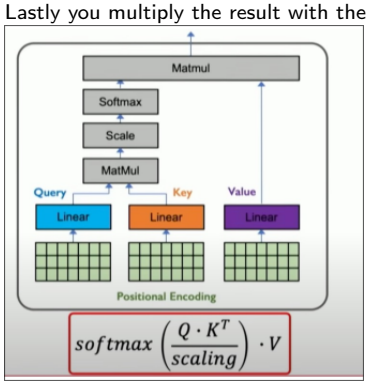


Essentially you check how similar the key is to the query.

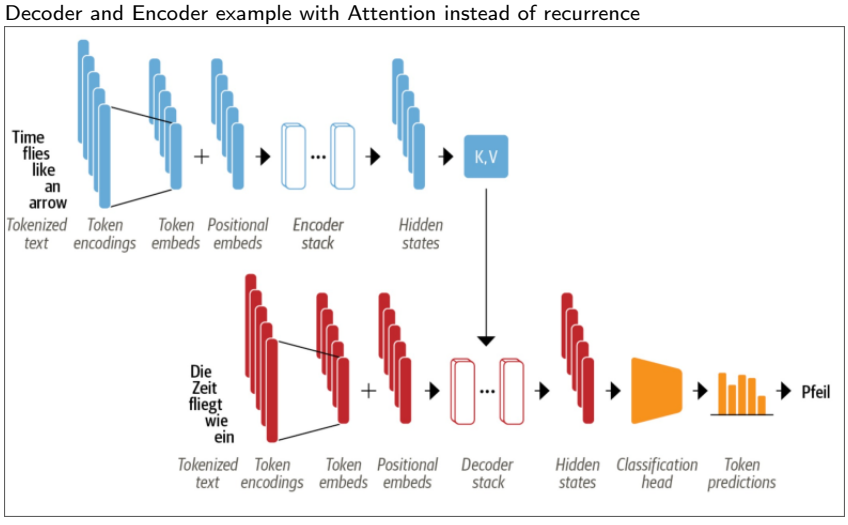




Lastly you multiply the result with the value, which results in the output.



Decoder and Encoder example with Attention instead of recurrence

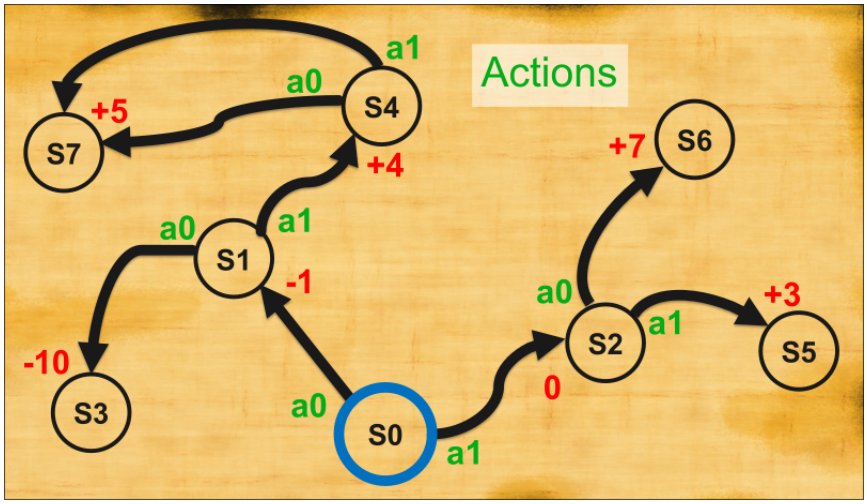


5 Reinforcement Learning

5.1 Limitations

- *Sample Efficiency*: Millions of iterations needed
- Defining the rewards can be extremely hard
- AI often does not adhere to the ultimate goal -> no understanding of context
  AI must avoid killing humans when driving -> might however accidentally kill more humans by executing extreme countermeasures

5.2 Markov Decision Process MDP

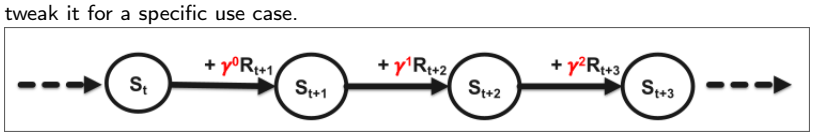


- S: Set of States
- A: Set of actions
- R: Rewards (positive or negative)
- P: transitions from state to state

Whenever the agent moves, it will always try to go with the route that has the highest outcome of rewards, aka it makes a guess on where it will end up with the highest reward number and proceeds to take actions based on that.
Note that each state above is clearly defined, this makes it easier to have an AI model that learns this way.
This also makes it hard to be used with real-world use cases or anything that "too many" inputs.
The more states something has, the harder proper reinforcement learning becomes, as not only the reward will be very vague, but also the route chosen.
The policy π is the function that calculates the probability of taking one action over another -> which is more likely to lead to a higher reward?
Note that the number of such functions is infinite in theory $\rightarrow$ no limit

5.3 Discount factory γ

This refers to the difference in time between rewards: take $10 now or $50 in 2 weeks?
How should the AI make this choice? Value the higher number or value the time, this factor makes it possible to change this behavior, as we might want to tweak it for a specific use case.



Note the exponential increase of gamma with time -> power of 1, of 2, etc.

5.4 Return R vs Reward G

$G_t = R_{t+1} + _{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$
Reward: this is the reward that is given immediately on landing in a state. The letter *r is used for constant rewards* and *R for random rewards*.Return: this is the sum of the future rewards, it is either a sum of random variables or constants. *The letter G is used*
Expected value: this is the estimation of the random variables -> we achieve this by *taking the mean of all possibilities -> throwing a dice N times and divide the sum by N*

5.5 The State Value Function V(s)

$V_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right]$

- You will see different notations for V(s):

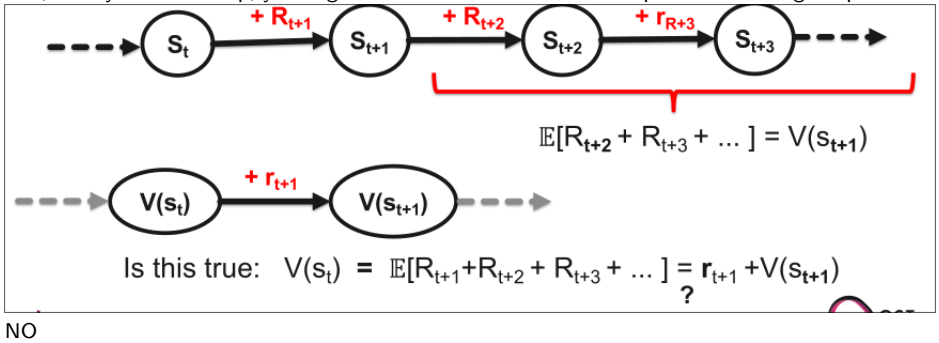
$V^\pi(s) = \mathbb{E}_{\tau \sim \pi}[G \mid s]$

$V^\pi(s) = \mathbb{E}[G \mid s; \pi]$

$V(s) = \mathbb{E}[G \mid s]$

read: The value of a state s, under policy π is defined as the expected value of the return G when starting in state s and following the policy π thereafter.

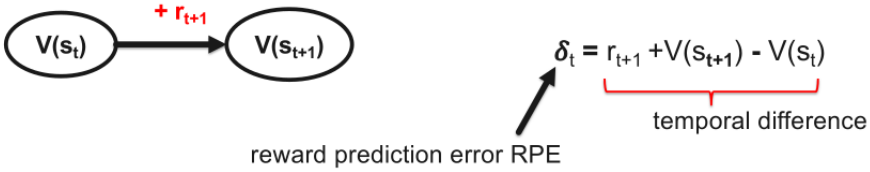
Note, whenever you start to take these actions, there will be a *delta* between the expected values and the real values.
Aka, once you take a step, you might notice that current + next expected is no longer equal to the previous sum of expected:



NO

5.5.1 Temporal Difference

$V(S_t) + \delta_t = r_{t+1} + V(S_{t+1})$
the δ is the difference between the original sum of expected values, to the current rewards, plus the rest of the expected values.



The TD-Learning update rule:

RPE = $r_{t+1} + V(s_{t+1}) - V(s_t)$

$V(s_t) \leftarrow V(s_t) + \alpha * RPE$

where α is the **learning rate**, aka 'step size' (typically a small value between 0.000001 and 0.1)

5.6 Policy Evaluation: Estmating state values V(s)

Policy Evaluation: Estimating state values V(s)

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Policy evaluation:

- For a given a policy, we estimate the state values V(s).
- The algorithm is interative: at each time-step t+1, the estimate of $V(s_t)$ is updated