Section 3

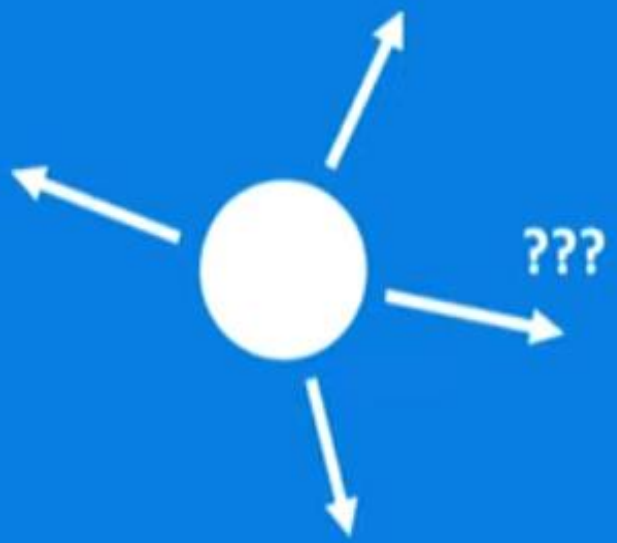
Custom Models and Training with TensorFlow,
Loading and Preprocessing Data with TensorFlow

Recurrent Neural Networks:
Back propagation through time,
Long Short Term Memory,
Gated Recurrent Units, Bidirectional LSTMs, Bidirectional RNNs

Convolutional Neural Networks:
Deep Computer vision using CNN: Convolutional Layer, Pooling Layer,
CNN Architectures: LeNet, AlexNet. GoogLeNet, VGGNet, ResNet,Xception, SENet,,
Object Detection

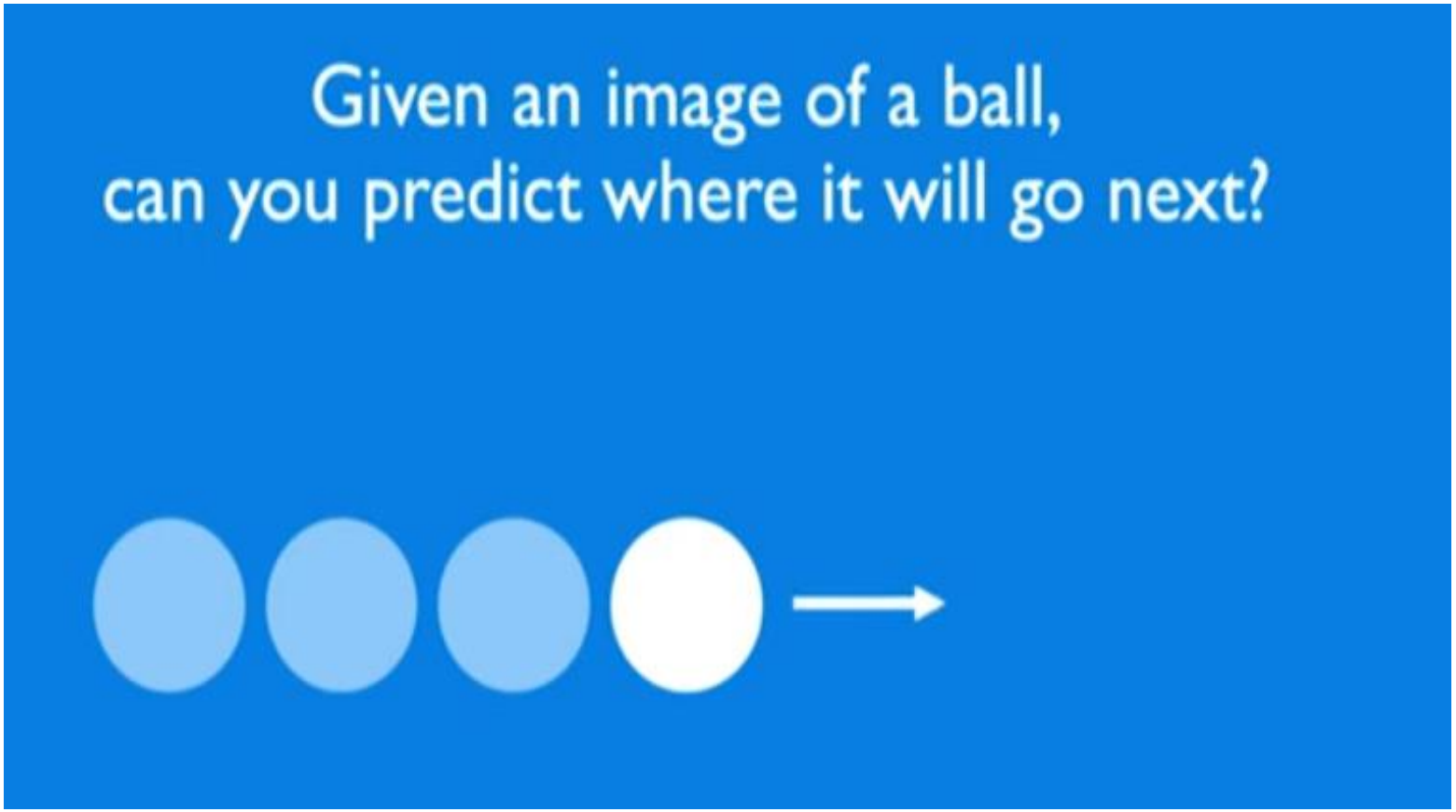
RECURRENT NEURAL NETWORKS

Given an image of a ball,
can you predict where it will go next?

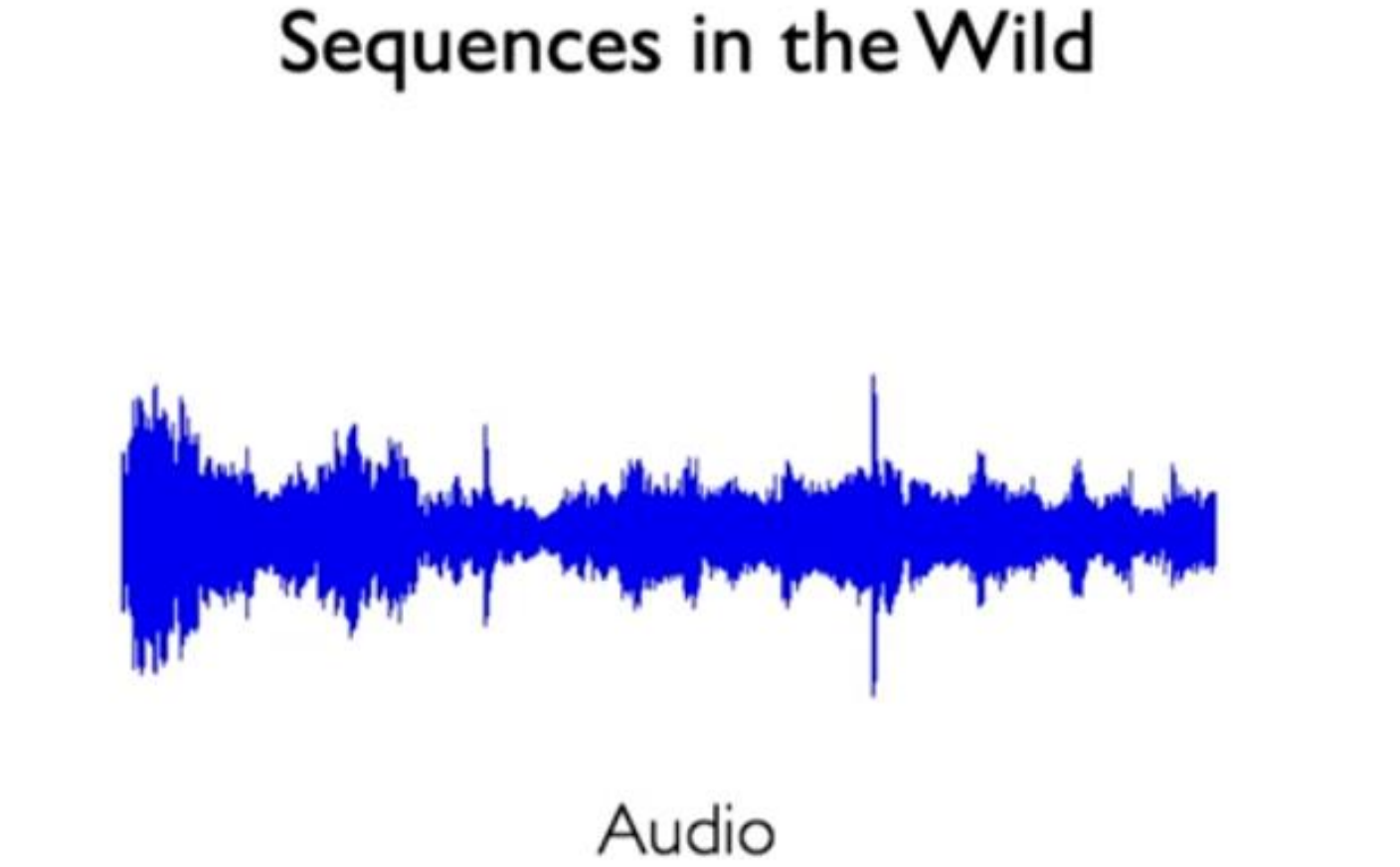


Need for sequential modelling

- When no prior information is available or in the absence of past history of movement of a ball.
- But when past information is available, the problem becomes simple.


Given an image of a ball,
can you predict where it will go next?

Examples of sequential data



Examples of sequential data

Sequences in the Wild

character: 6 . S I 9 I

word: Introduction to Deep Learning

Text

- More examples include stock prices, medial images, DNA sequencing

Sequential patterns

Text

Speech

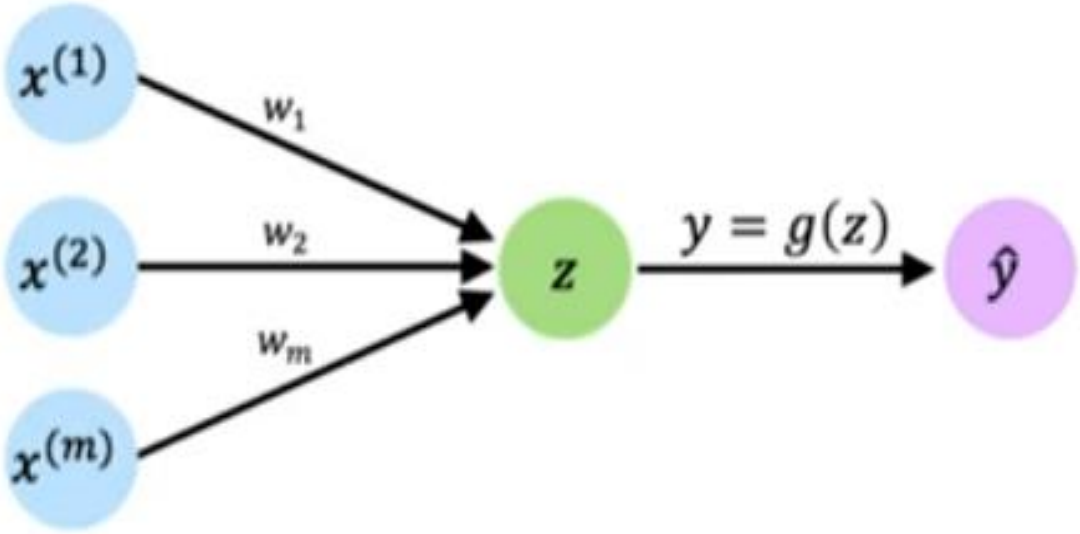
Audio

Video

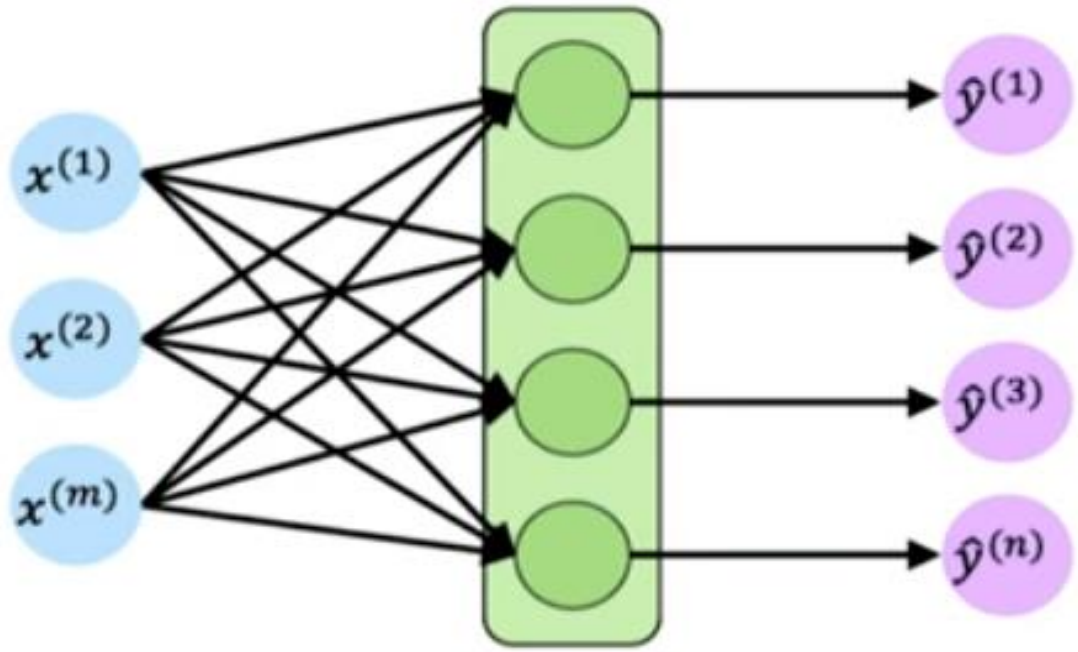
Physical processes

Anything embedded in time (almost everything)

The Perceptron Revisited



Feed-Forward Networks Revisited

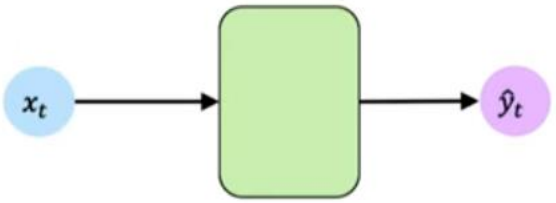


$\boldsymbol{x} \in \mathbb{R}^m$

$\hat{\boldsymbol{y}} \in \mathbb{R}^n$

- There is no notion of sequence or time in this concept of multi layer network.
- Focusing on the diagram after collapsing all the three components, i.e.
  - inputs,
  - perceptrons, and
  - outputs in the form of vectors.
  - Inputs are a vector of length m, and outputs are a vector of length n.

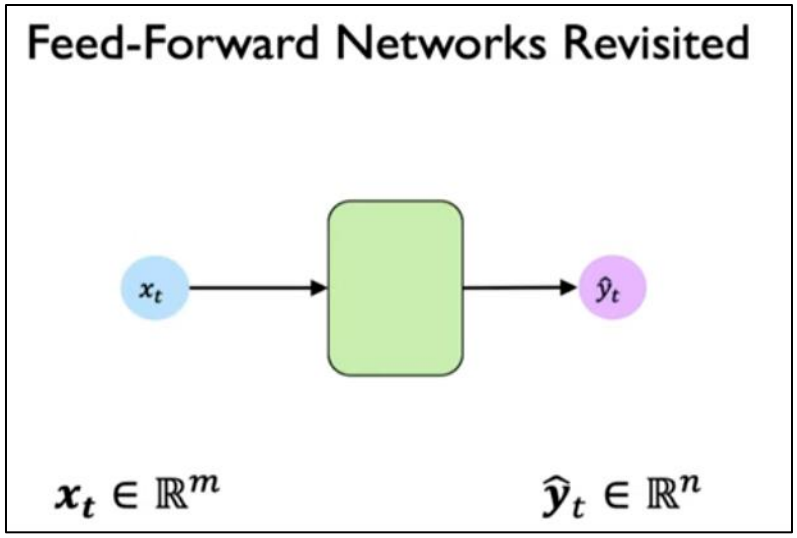
Feed-Forward Networks Revisited



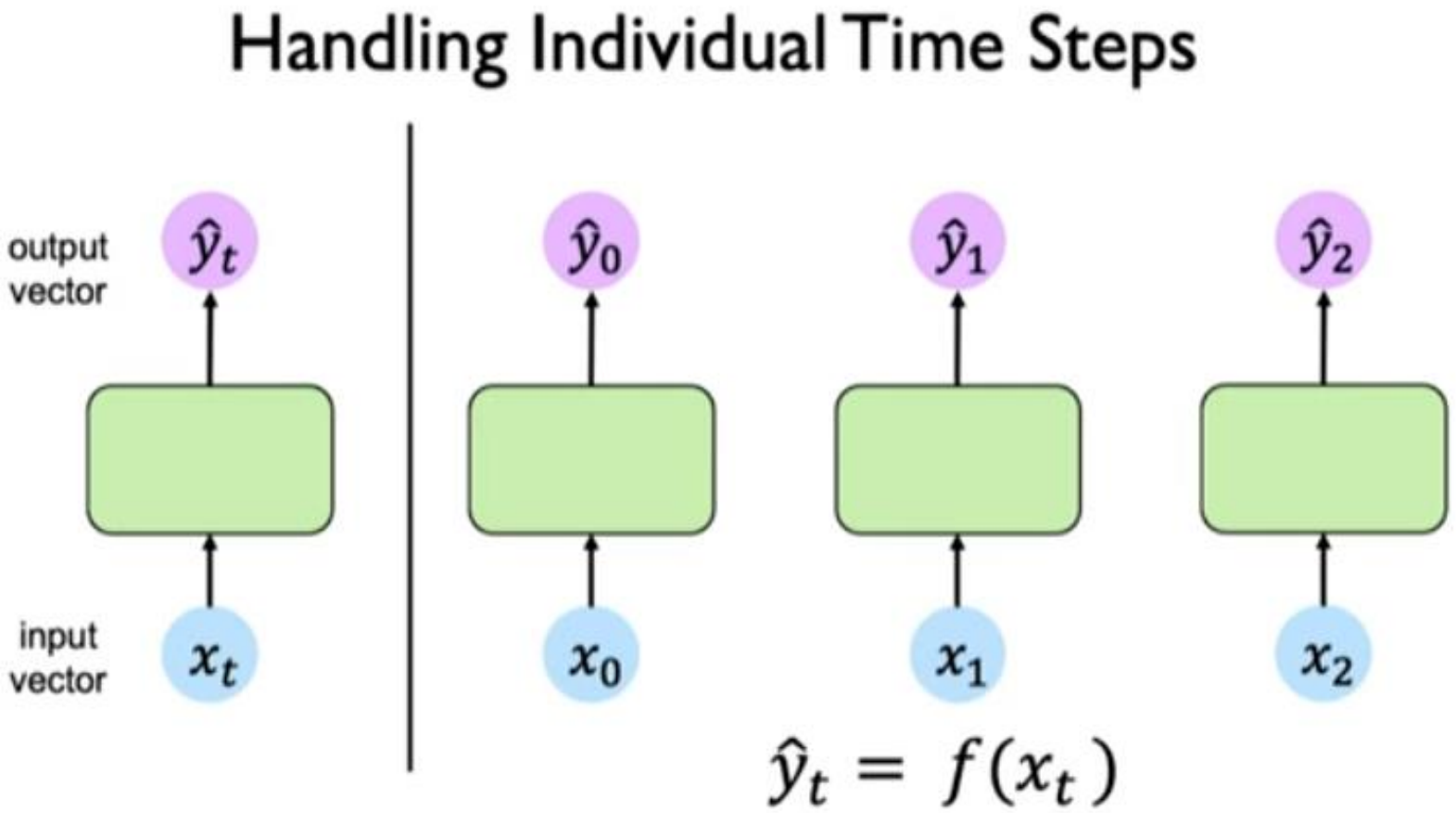
$\boldsymbol{x_t} \in \mathbb{R}^m$ $\hat{\boldsymbol{y}}_t \in \mathbb{R}^n$

No concept of sequence or time

- Individual inputs occurring at individual time steps.
- So one example is fed into the network then next and so on.
- But these individual inputs are isolated, they are not related to the inputs coming before or after them.

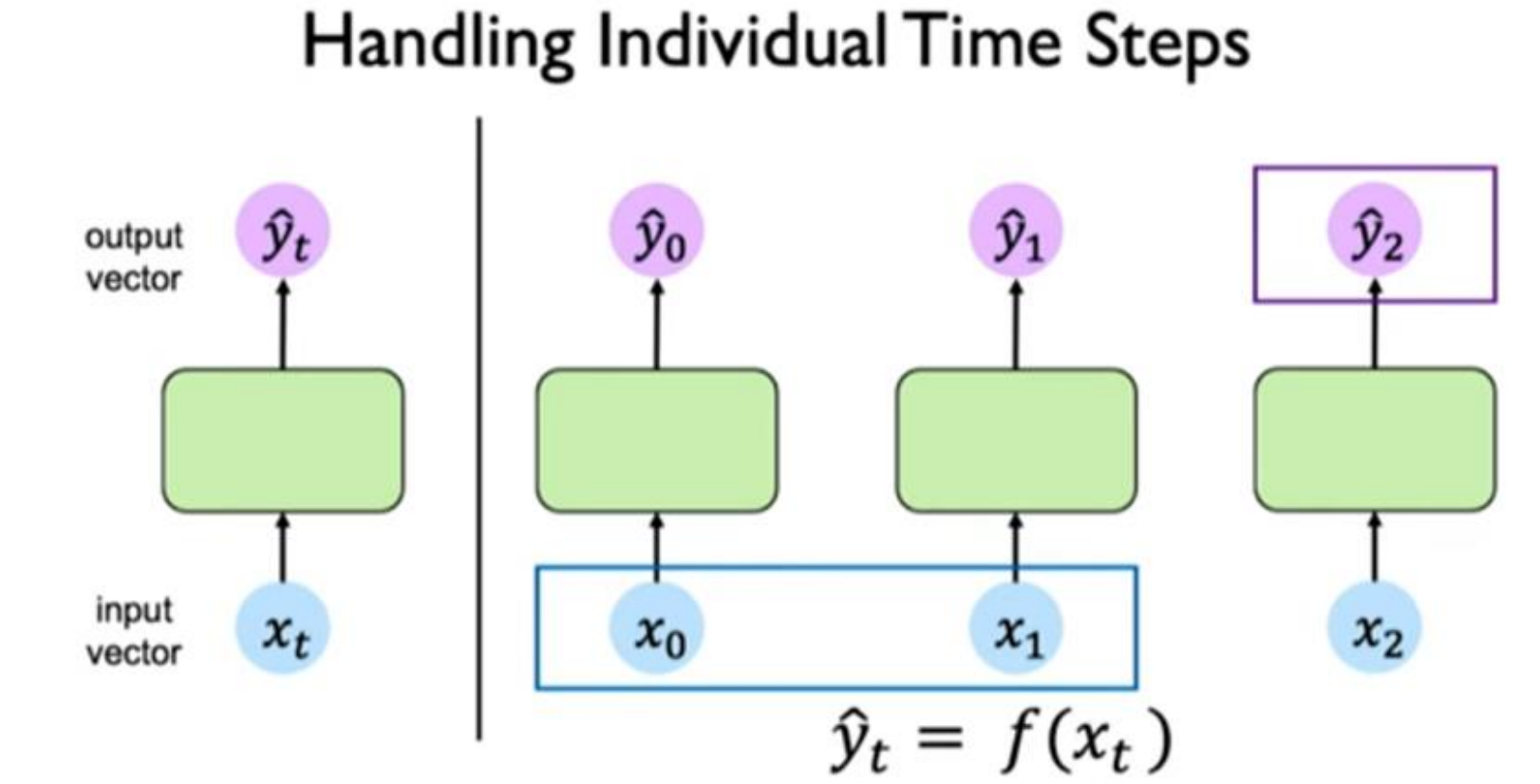


- In the feedforward network
  - output is dependent upon the current inputs only.
  - No notion of relation or interdependence across the transformations.



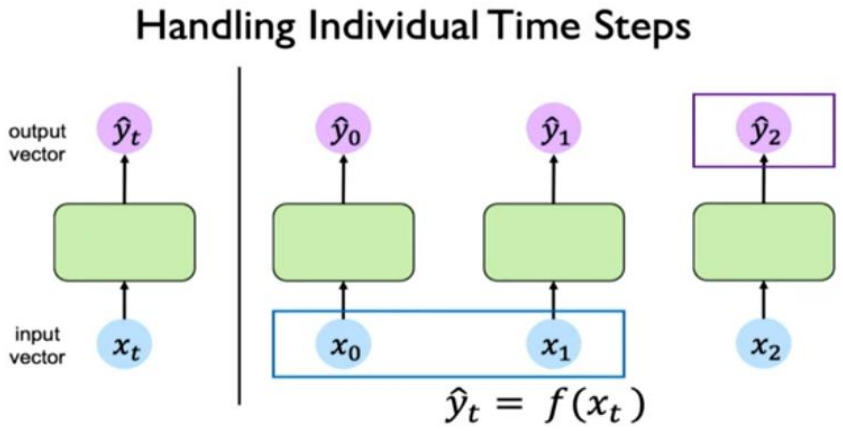
- But in sequential data
  - output at time t, depends upon the inputs at time t-1,t-2 or more historical data..
  - How can we capture this interdependence?

Let’s introduce the concept of time or sequence in this.

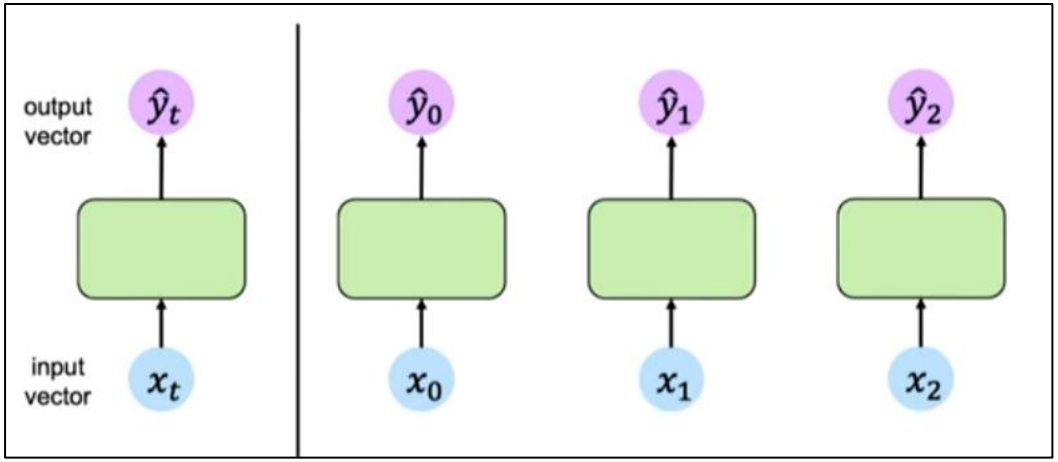


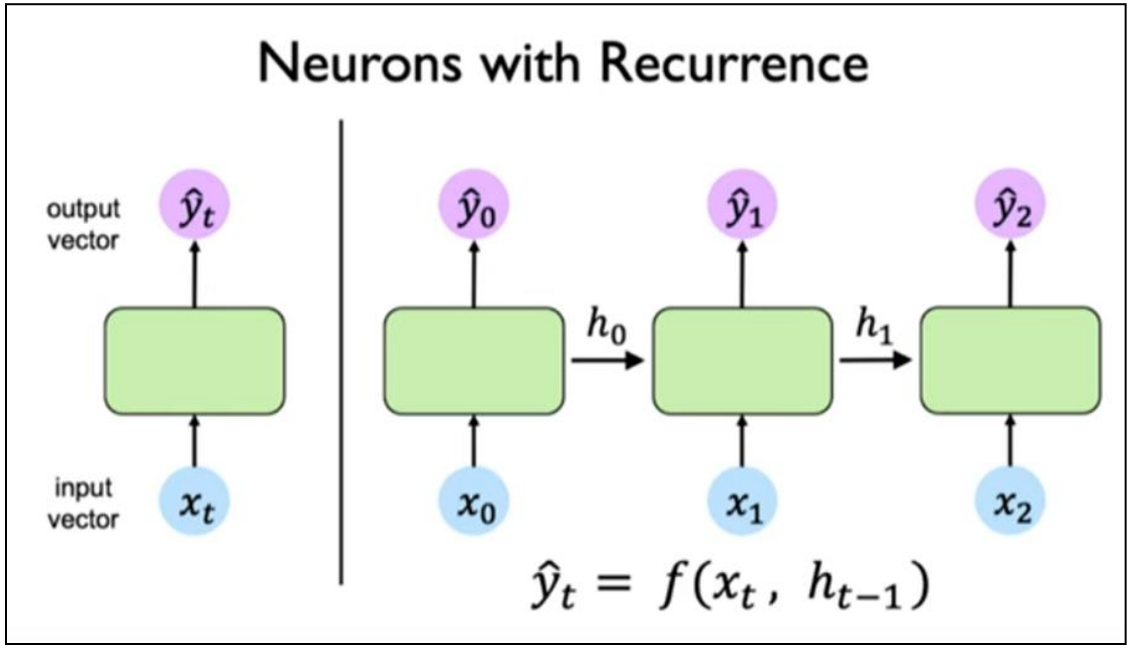
Establish a recurrence relation

- So we relate the units through a recurrence relation.
- Need to capture the historical information maintained by the neurons in their computations at the time stamp t-1, t-2 to be passed on for computation at the t time stamp.
- So in a recurrent network, internal memory maintains the state of the network at time t-1, called h_t, passes it to the network computation happening at time t.

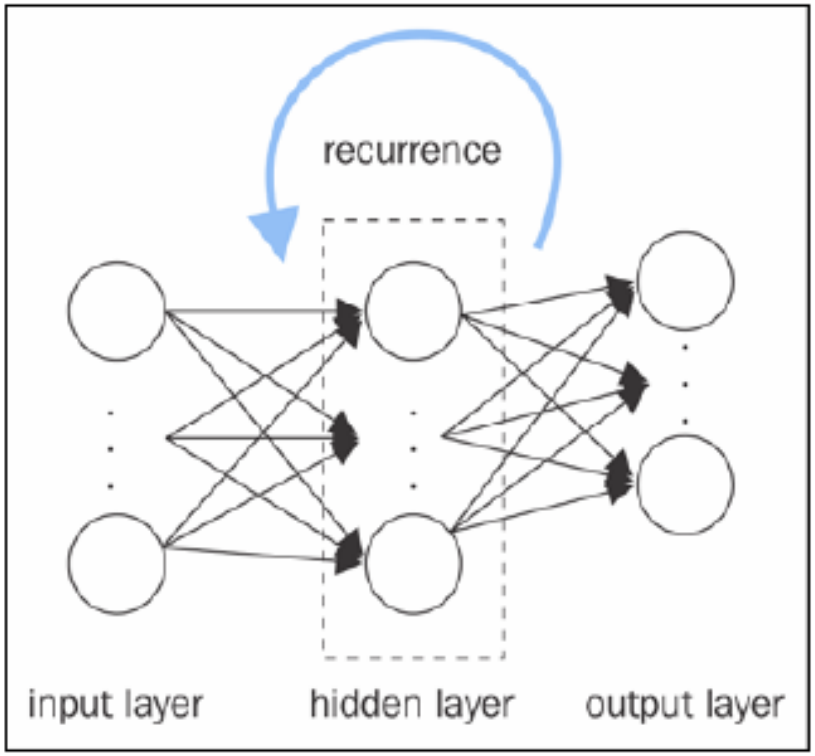


Neurons without recurrence



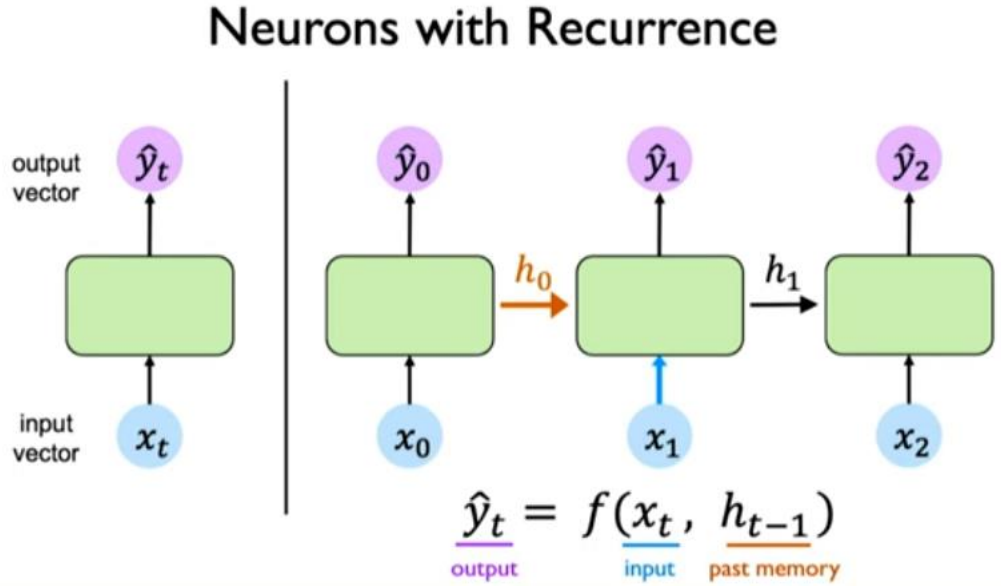


Establish a recurrence relation



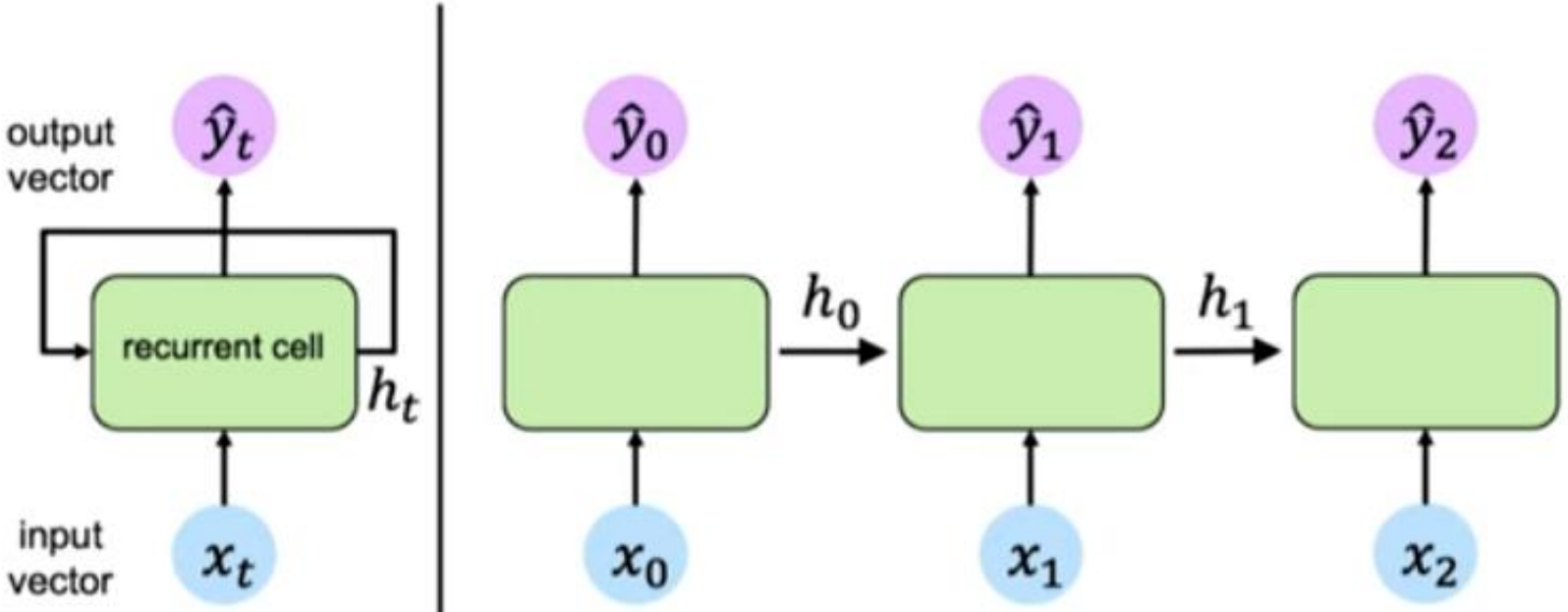
Introducing Recurrent Neural Networks

- So, in a RNN, output of the network does not depend on the input at time t, but also on the inputs in the previous timestamps retained in the memory of the network.



- Past memory retains the history of what has happened in the sequence prior to this.

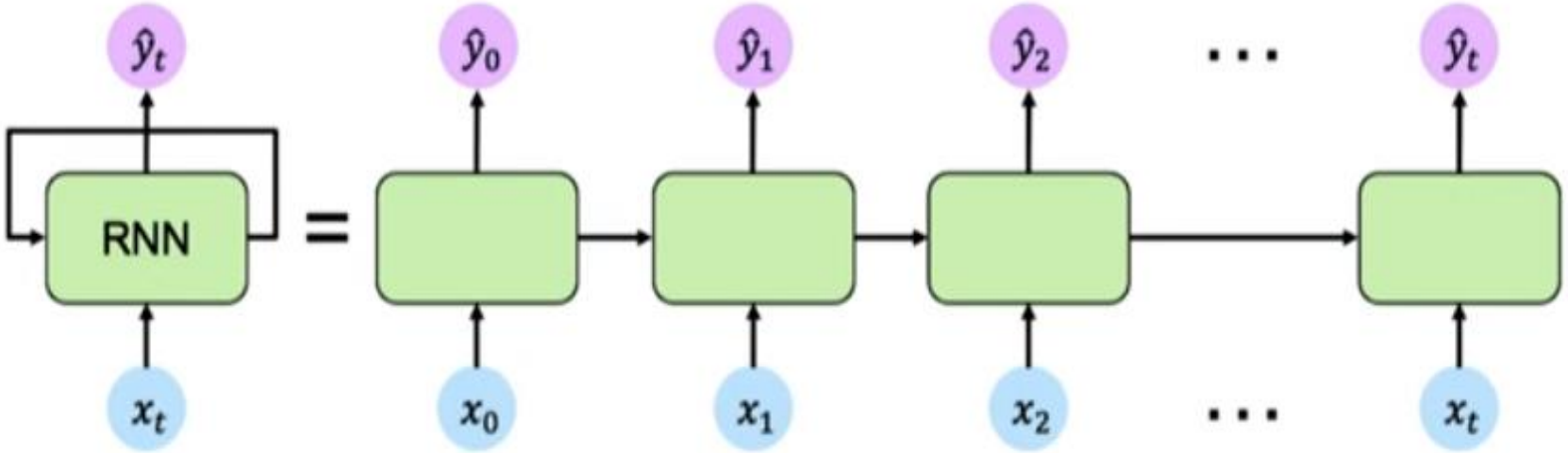
Neurons with Recurrence



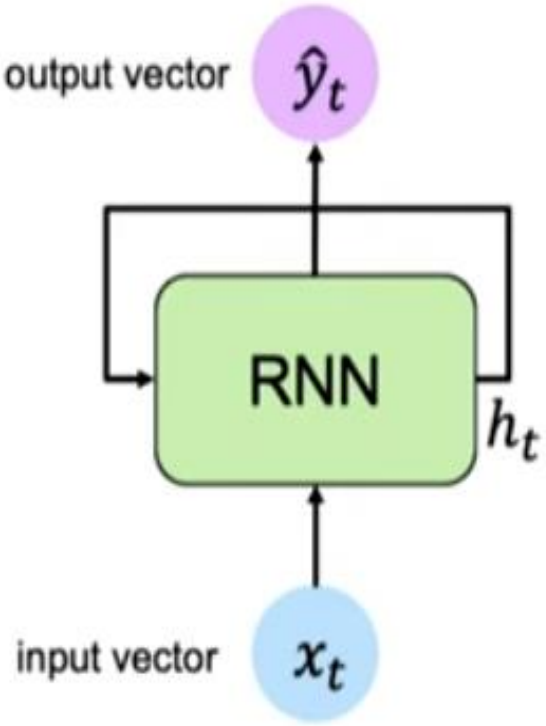
$$\underline{\hat{y}_t} = f(\underline{x_t}, \underline{h_{t-1}})$$

output input past memory

RNNs: Computational Graph Across Time



Recurrent Neural Networks (RNNs)



Apply a **recurrence relation** at every time step to process a sequence:

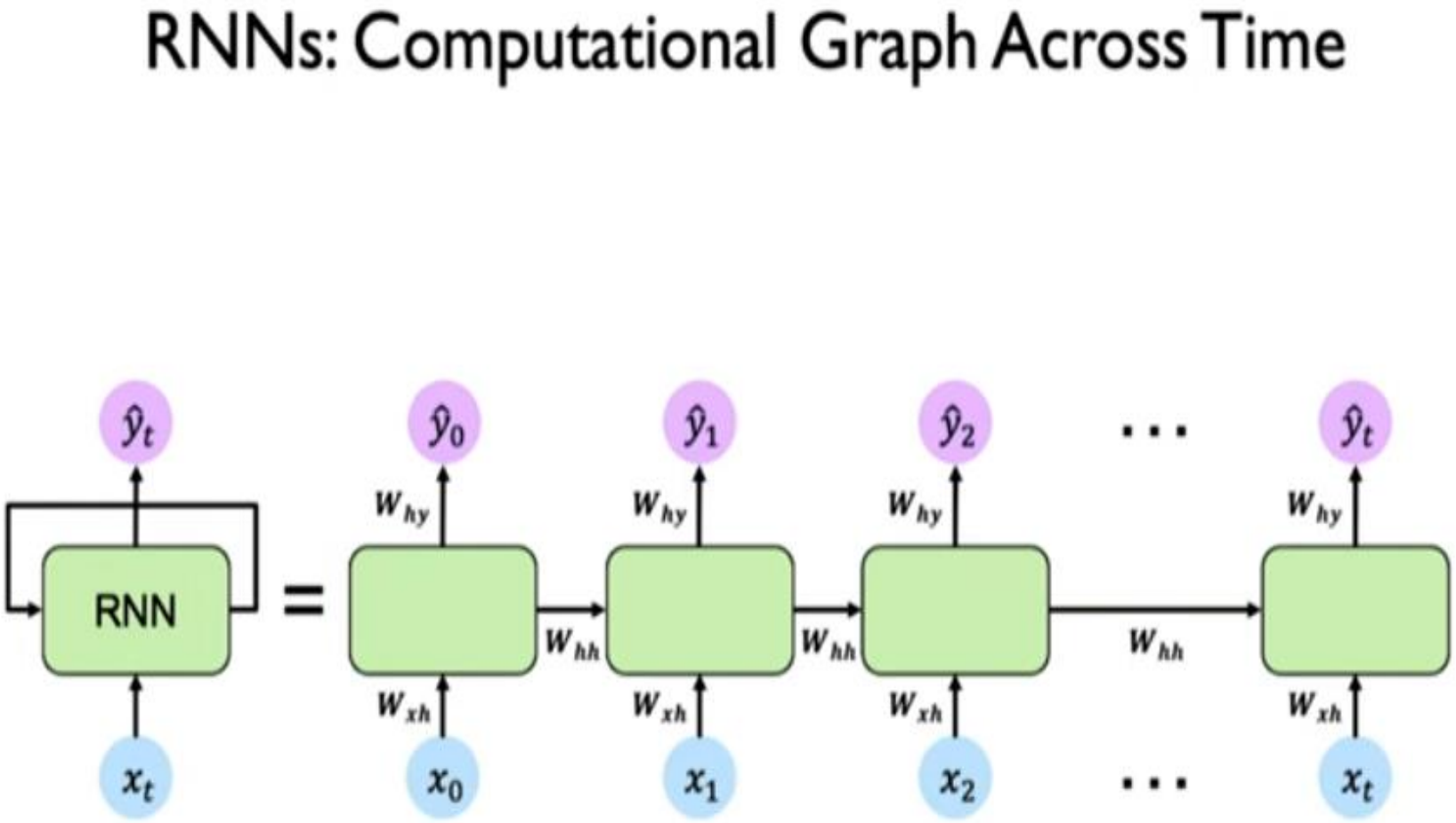
$$h_t = f_W(x_t, h_{t-1})$$

cell state | function with weights W | input | old state

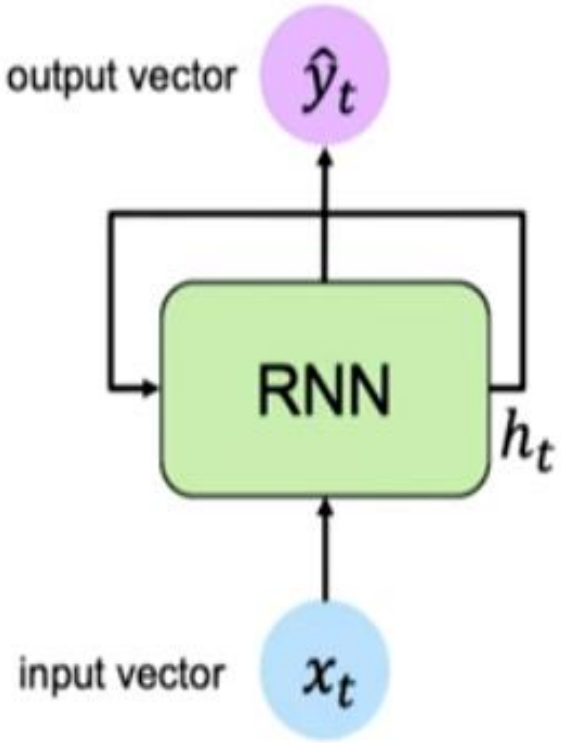
Note: the same function and set of parameters are used at every time step

RNNs have a **state**, h_t, that is updated **at each time step** as a sequence is processed

Look at the weights



RNN State Update and Output



Output Vector

$$\hat{y}_t = \boldsymbol{W}_{\boldsymbol{hy}}^{\boldsymbol{T}} h_t$$

Update Hidden State

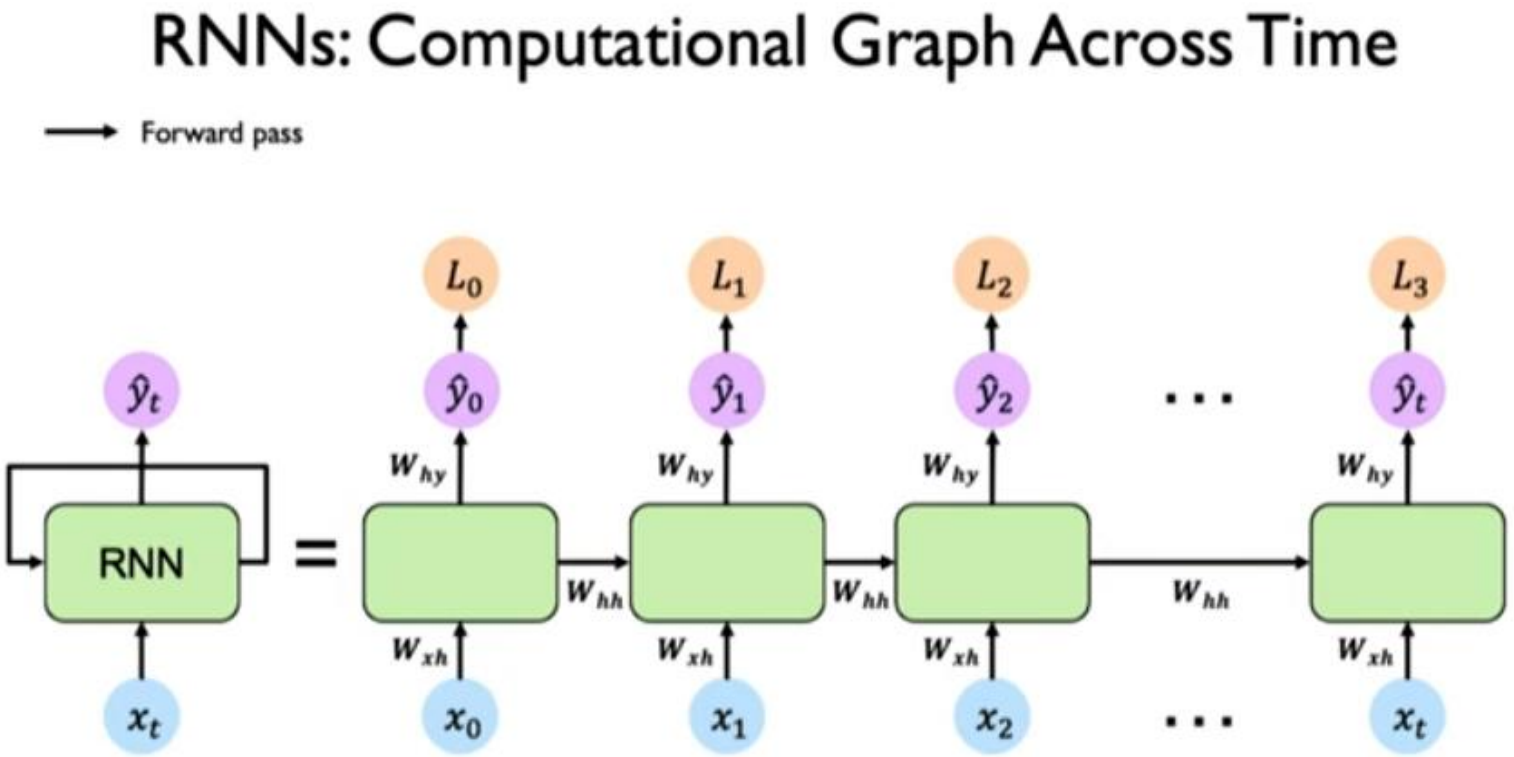
$$h_t = \tanh(\boldsymbol{W}_{\boldsymbol{hh}}^{\boldsymbol{T}} h_{t-1} + \boldsymbol{W}_{\boldsymbol{xh}}^{\boldsymbol{T}} x_t)$$

Input Vector

$$x_t$$

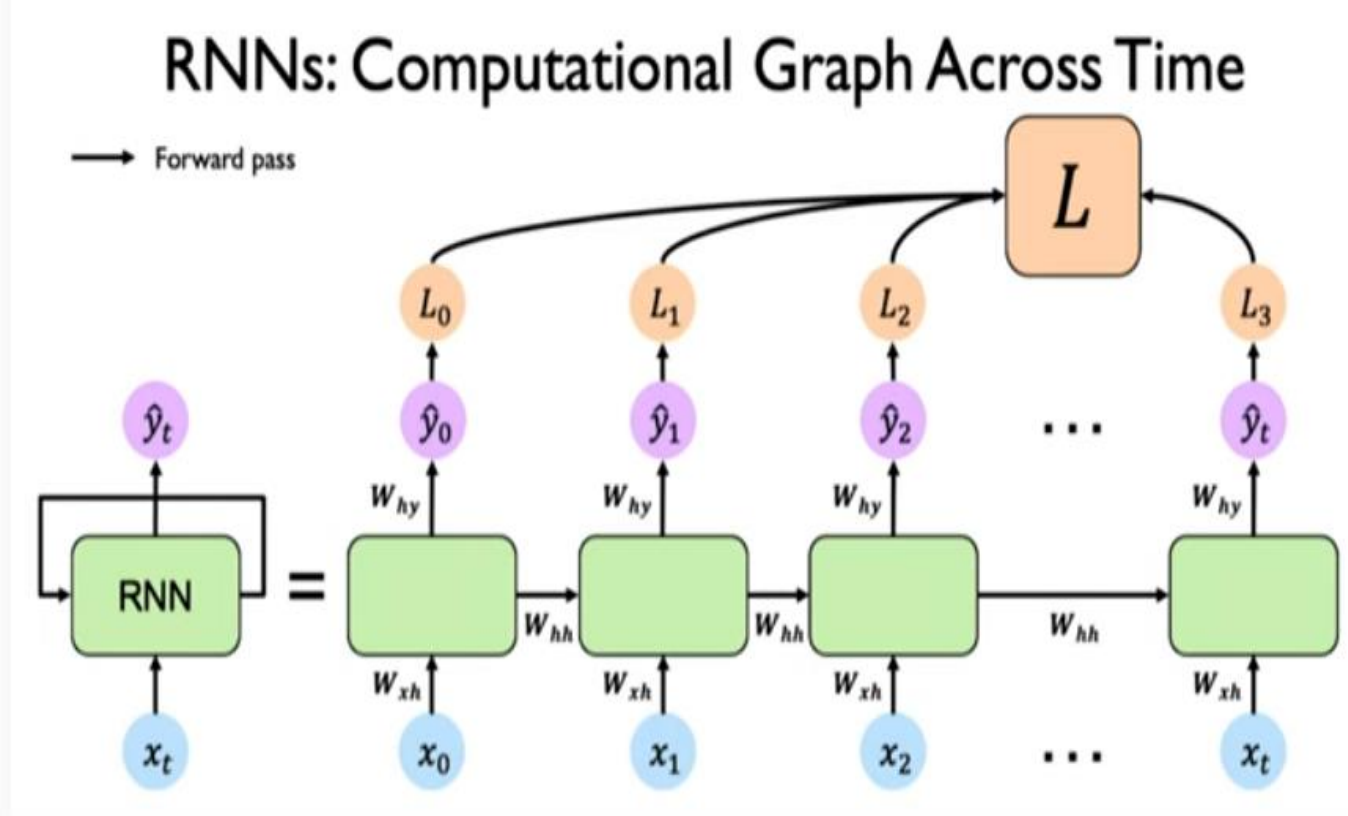
How does it train?

- During every iteration, loss is calculated.



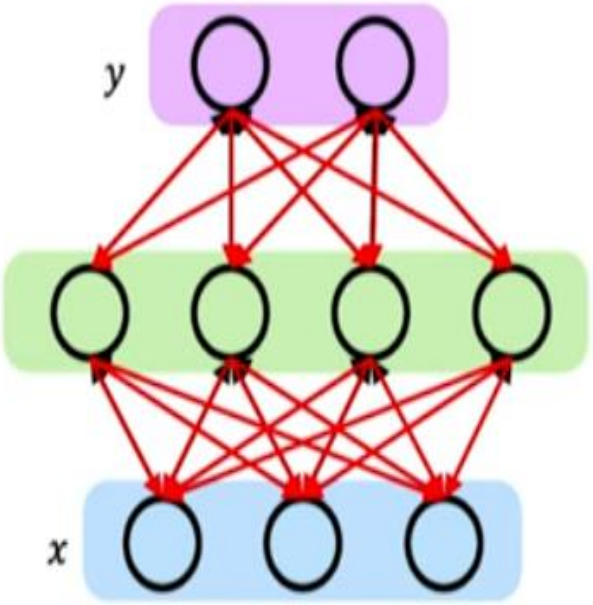
How does it train?

- Calculate the total loss.



How does it train?

Recall: Backpropagation in Feed Forward Models

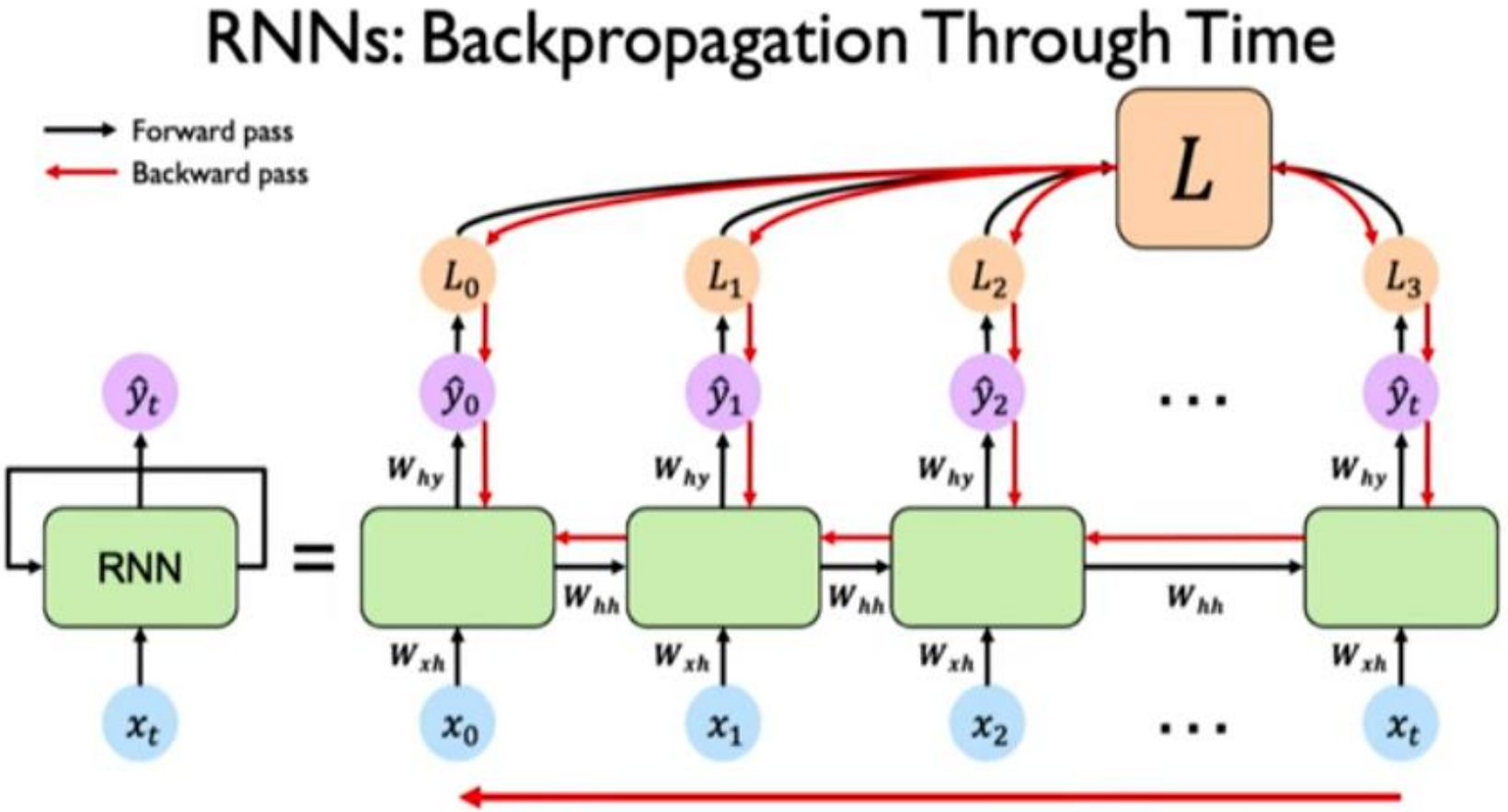


Backpropagation algorithm:

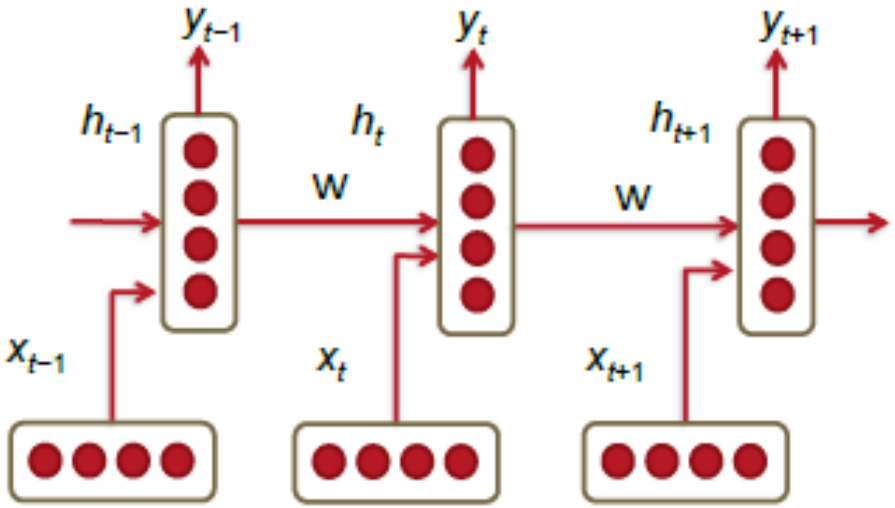
1. Take the derivative (gradient) of the loss with respect to each parameter
2. Shift parameters in order to minimize loss

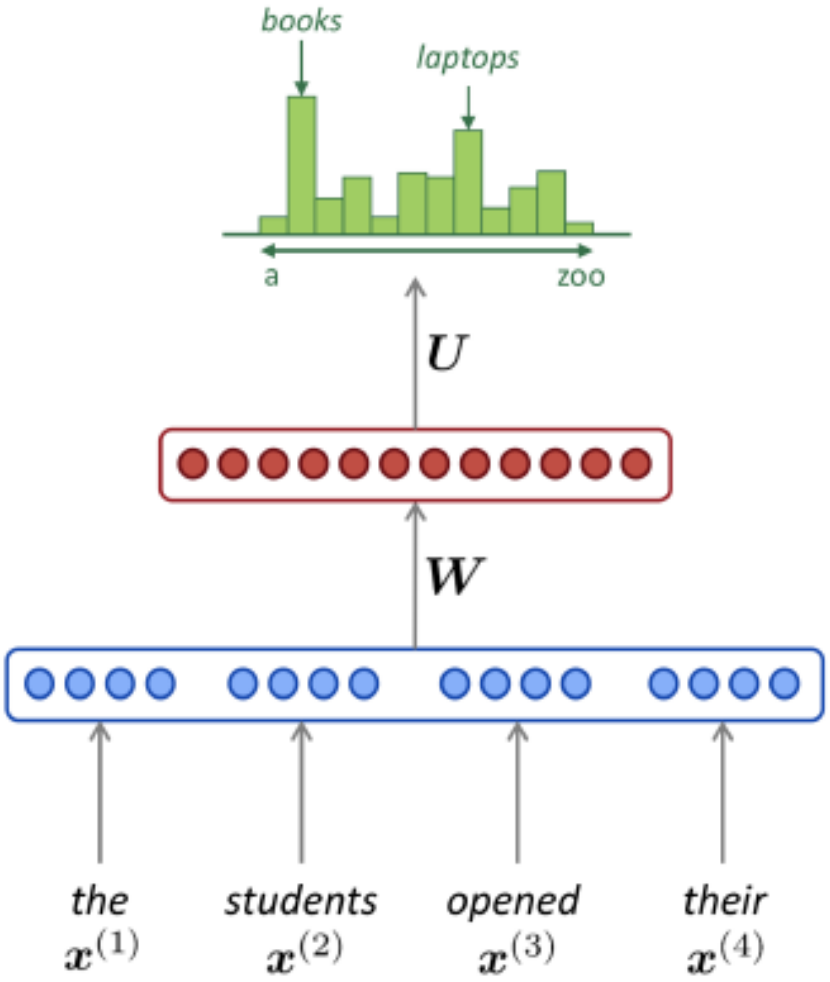
How does it train?

- Back Propagation Through Time (BPTT)
  - Error back propagates through time as well.



A simplified view





Inputs, weights, hidden layers are vectors

Implementing a RNN in Keras Part1

- #Simple RNN and MNIST dataset
- import tensorflow as tf
- import numpy as np
- # instantiate mnist and load data:
- mnist = tf.keras.datasets.mnist
- (x_train, y_train), (x_test, y_test) = mnist.
- load_data()
- # one-hot encoding for all labels to create 1x10
- # vectors that are compared with the final layer:
- y_train = tf.keras.utils.to_categorical(y_train)
- y_test = tf.keras.utils.to_categorical(y_test)

Implementing a RNN in Keras Part 2

```
# resize and normalize the 28x28 images:
image_size = x_train.shape[1]
x_train = np.reshape(x_train,[-1, image_size, image_size])
x_test = np.reshape(x_test, [-1, image_size, image_size])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
# initialize some hyper- parameters:
input_shape = (image_size, image_size)
batch_size = 128
hidden_units = 128
dropout_rate = 0.3
# RNN-based Keras model with 128 hidden units:
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.SimpleRNN(units=hidden_units,
                                    dropout=dropout_rate,
                                    input_shape=input_shape))
model.add(tf.keras.layers.Dense(num_labels))
model.add(tf.keras.layers.Activation('softmax'))
model.summary()
```

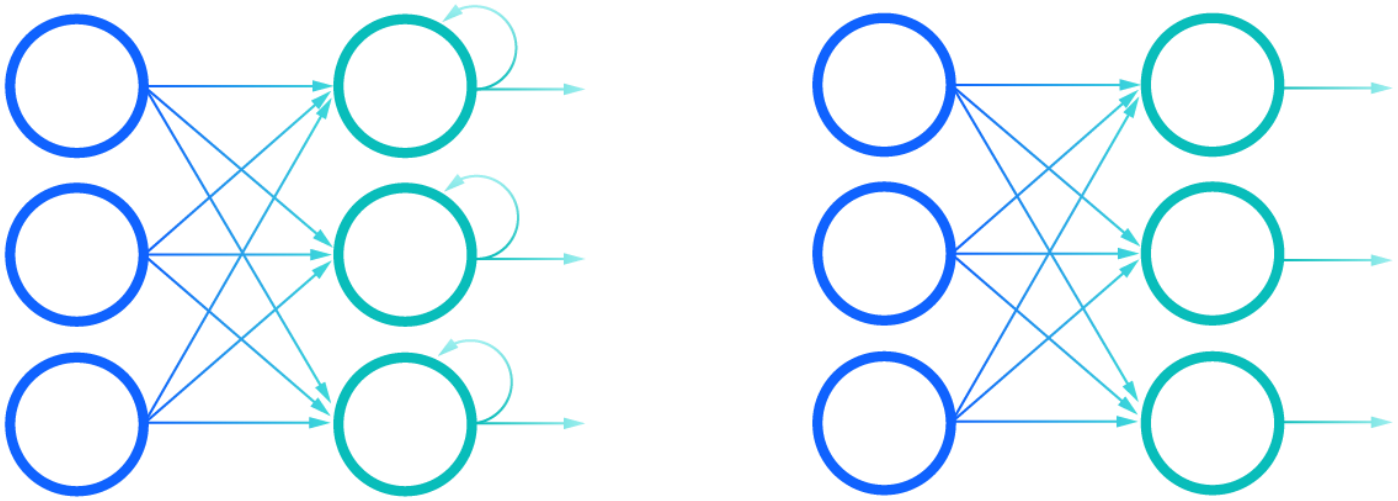
Implementing a RNN in Keras Part 3

```
model.compile(loss='categorical_crossentropy',
                        optimizer='sgd',
                        metrics=['accuracy'])
# train the network on the training data:
model.fit(x_train, y_train, epochs=8,
                batch_size=batch_size)
#calculate and then display the accuracy:
loss, acc = model.evaluate(x_test, y_test,
                                        batch_size=batch_size)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

Defining a RNN

- A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data.
- These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (nlp), speech recognition, and image captioning;
- They are incorporated into popular applications such as Siri, voice search, and Google Translate.
- Like feedforward and convolutional neural networks (CNNs), recurrent neural networks utilize training data to learn.
- They are distinguished by their “memory” as they take information from prior inputs to influence the current input and output.
- While traditional deep neural networks assume that inputs and outputs are independent of each other, the output of recurrent neural networks depend on the prior elements within the sequence.
- While future events would also be helpful in determining the output of a given sequence, unidirectional recurrent neural networks cannot account for future events in their predictions.

Recurrent Neural Network vs. Feedforward Neural Network

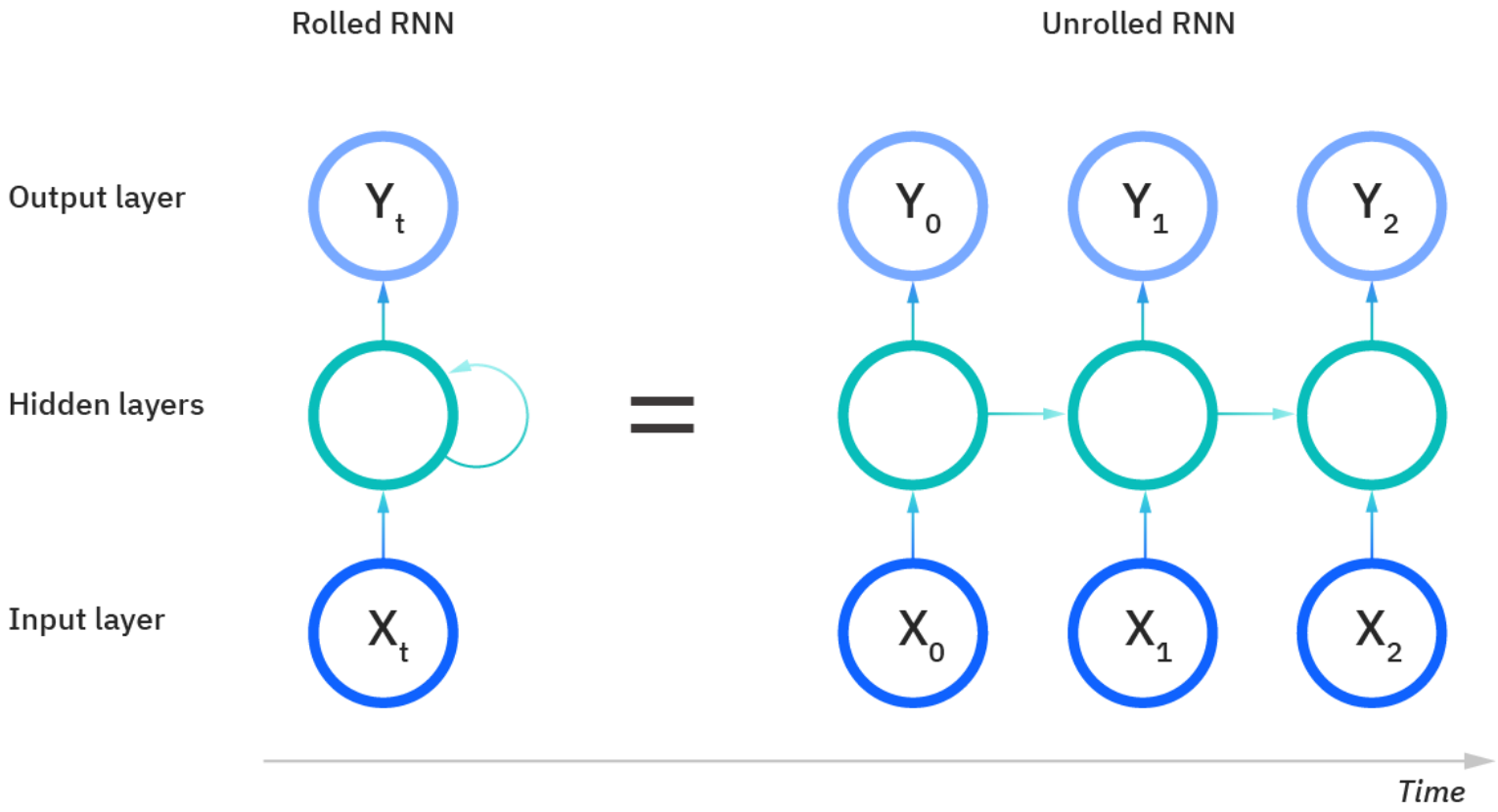


- Comparison of Recurrent Neural Networks (on the left) and Feedforward Neural Networks (on the right)

“feeling under the weather”

- Let’s take an idiom, such as “feeling under the weather”,
  - which is commonly used when someone is ill, to aid us in the explanation of RNNs.
  - In order for the idiom to make sense, it needs to be expressed in that specific order.
  - As a result, recurrent networks need to account for the position of each word in the idiom and they use that information to predict the next word in the sequence.
- Looking at the diagram (on the next page)
  - the “rolled” visual of the RNN represents the whole neural network, or rather the entire predicted phrase, like “feeling under the weather.”
  - The “unrolled” visual represents the individual layers, or time steps, of the neural network.
    - Each layer maps to a single word in that phrase, such as “weather”.
    - Prior inputs, such as “feeling” and “under”, would be represented as a hidden state in the third timestep to predict the output in the sequence, “the”.

“feeling under the weather”



backpropagation through time (BPTT) algorithm

- Recurrent neural networks leverage backpropagation through time (BPTT) algorithm to determine the gradients, which is slightly different from traditional backpropagation as it is specific to sequence data.
- The principles of BPTT are the same as traditional backpropagation, where the model trains itself by calculating errors from its output layer to its input layer.
- These calculations allow us to adjust and fit the parameters of the model appropriately.
- BPTT differs from the traditional approach in that BPTT sums errors at each time step whereas feedforward networks do not need to sum errors as they do not share parameters across each layer.

Types of recurrent neural networks

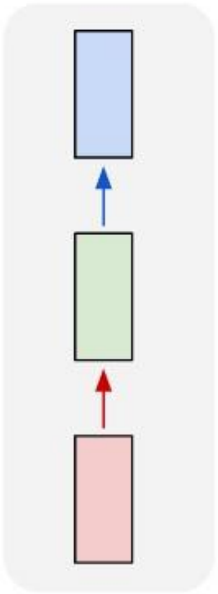
- Feedforward networks map one input to one output, and while we've visualized recurrent neural networks in this way in the above diagrams, they do not actually have this constraint.
- Instead, their inputs and outputs can vary in length, and different types of RNNs are used for different use cases, such as music generation, sentiment classification, and machine translation.
- Different types of RNNs are usually expressed using the following diagrams:

What makes Recurrent Networks so special?

- A glaring limitation of Vanilla Neural Networks (and also Convolutional Networks) is that they are too constrained:
  - they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes).
- Not only that: These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model).
- The core reason that recurrent nets are more exciting is that they **allow us to operate over *sequences* of vectors**: Sequences in the input, the output, or in the most general case both.
- A few examples may make this more concrete:

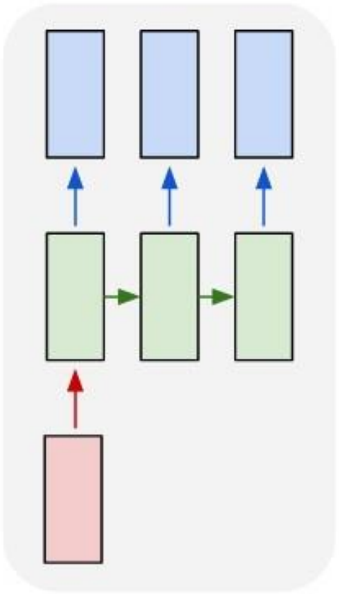
Types of RNN networks

one to one



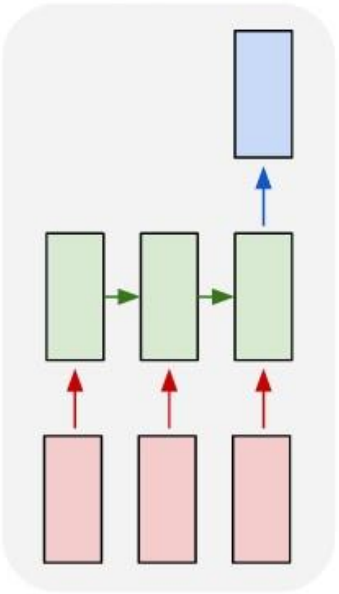
e.g. image classification

one to many



e.g. image captioning

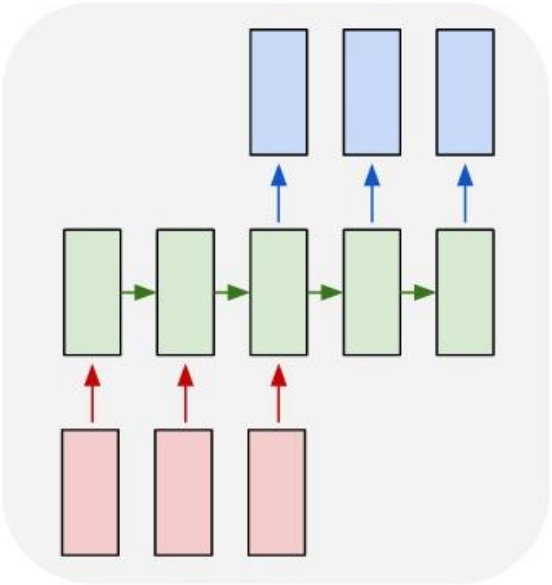
many to one



e.g. sentiment analysis

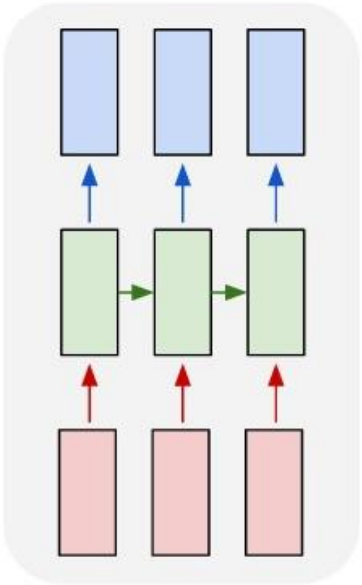
e.g. video classification to label each frame of the video

many to many



e.g. Machine Translation say English to Punjabi

many to many



The pros and cons of a typical RNN architecture are summed up in the table below:

Advantages	Drawbacks
• Possibility of processing input of any length • Model size not increasing with size of input • Computation takes into account historical information • Weights are shared across time	• Computation being slow • Difficulty of accessing information from a long time ago • Cannot consider any future input for the current state

Weights are shared across time

- To go from multilayer networks to recurrent networks, we need to take advantage of one of the early ideas found in machine learning and statistical models of the 1980s:

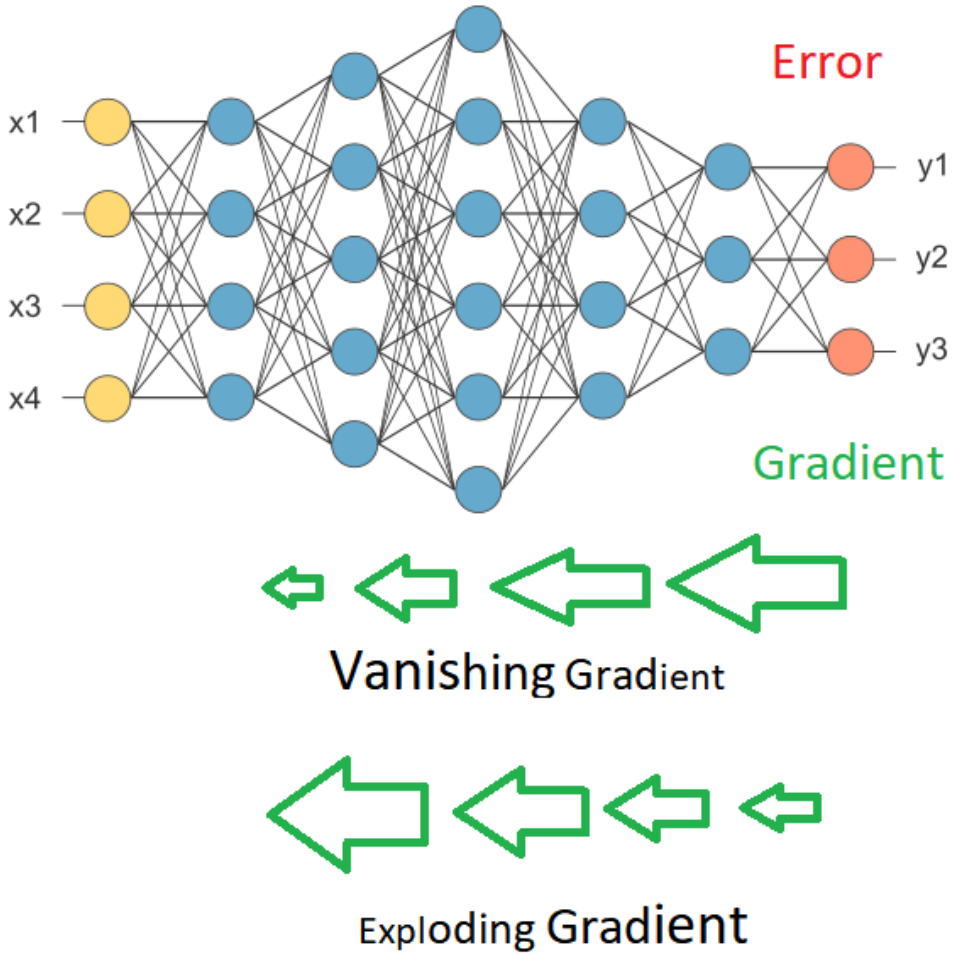
  sharing parameters across different parts of a model.

- Parameter sharing makes it possible to extend and apply the model to examples of different forms (different lengths, here) and generalize across them.
- If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training.
- Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence.
- For example, consider the two sentences

  "I went to Nepal in 2009" and "In 2009, I went to Nepal."

- If we ask a machine learning model to read each sentence and extract the year in which the narrator went to Nepal, we would like it to recognize the year 2009 as the relevant piece of information, whether it appears in the sixth word or in the second word of the sentence.
- Suppose that we trained a feedforward network that processes sentences of fixed length.
- A traditional fully connected feedforward network would have separate parameters for each input feature, so it would need to learn all the rules of the language separately at each position in the sentence.
- By comparison, a recurrent neural network shares the same weights across several time steps.

Two problems, known as exploding gradients and vanishing gradients.



Two problems, known as exploding gradients and vanishing gradients.

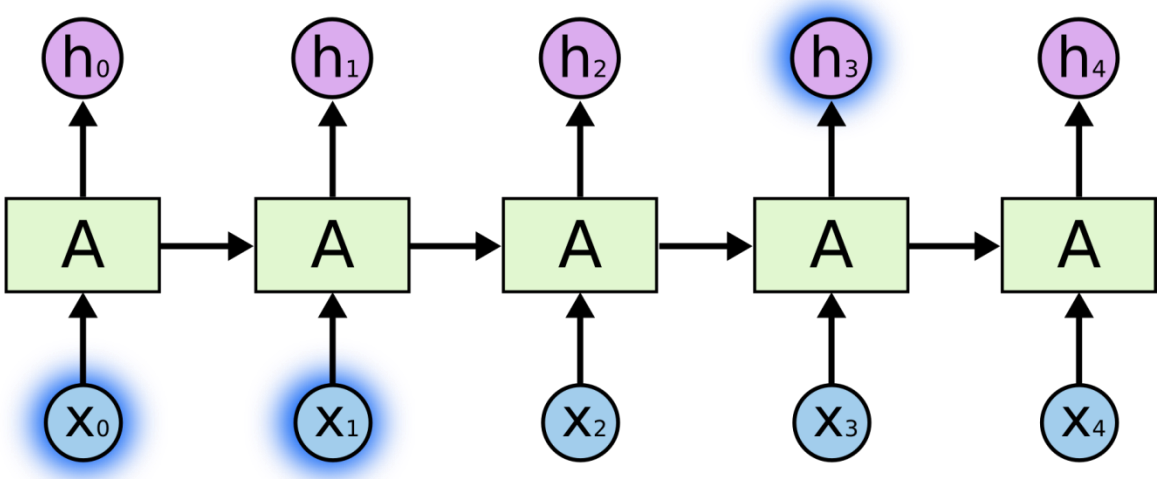
- RNNs tend to run into two problems, known as exploding gradients and vanishing gradients.
- These issues are defined by the size of the gradient, which is the slope of the loss function along the error curve.
  - When the gradient is too small, it continues to become smaller, updating the weight parameters until they become insignificant—i.e. 0.
    - When that occurs, the algorithm is no longer learning.
  - Exploding gradients occur when the gradient is too large, creating an unstable model.
    - In this case, the model weights will grow too large, and they will eventually be represented as NaN.
- So, the basic problem is that gradients propagated over many stages tend to either vanish (most of the time) or explode.
- One solution to these issues is to reduce the number of hidden layers within the neural network, eliminating some of the complexity in the RNN model.
  - But such a solution is not suitable for creating models that has to learn from information in the input in the distant past. E.g. NLP applications.
- Another solution is to design a network to handle long term dependencies.

The Problem of Long-Term Dependencies

- One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame.
- If RNNs could do this, they'd be extremely useful.
- But can they?
- It depends.

Where is the context?

- Sometimes, we only need to look at recent information to perform the present task.
- For example, consider a language model trying to predict the next word based on the previous ones.
- If we are trying to predict the last word in "the clouds are in the sky," we don't need any further context – it's pretty obvious the next word is going to be sky.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



Where is the context?

- But there are also cases where we need more context. Consider trying to predict the last word in the text

  "I grew up in France… I speak fluent *French*."
- Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back.
- It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

Theory YES
Toy problems YES
In practice NO

- In theory, RNNs are absolutely capable of handling such "long-term dependencies."
- A human could carefully pick parameters for them to solve toy problems of this form.
- Sadly, in practice, RNNs don't seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

- Thankfully, LSTMs don't have this problem!

LSTM Networks

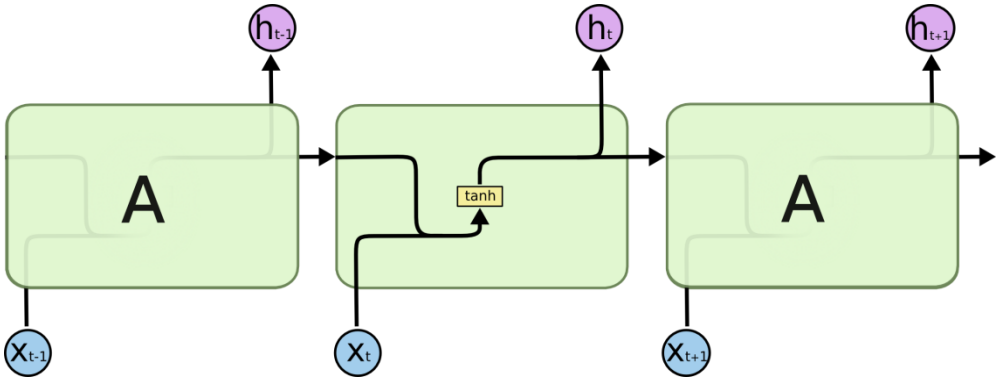
- Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies.
- LSTMs were developed in 1997 and went on to exceed the accuracy performance of state-of-the-art algorithms.
- LSTMs also began revolutionizing speech recognition (circa 2007).
- Then in 2009 an LSTM won pattern recognition contests, and in 2014, Baidu used RNNs to exceed speech recognition records.
- They work tremendously well on a large variety of problems, and are now widely used.
- They are well suited for many use cases, including NLP, speech recognition, and handwriting recognition.

LSTM – born to handle long term dependencies

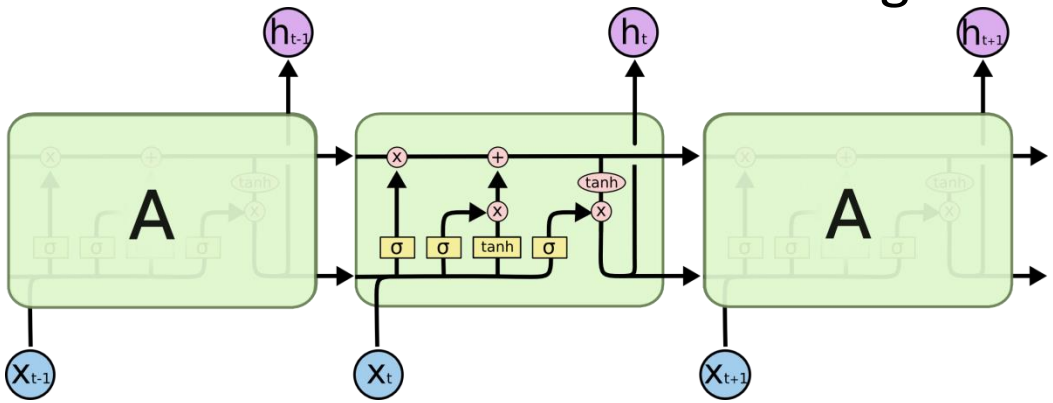
- LSTMs are explicitly designed to avoid the long-term dependency problem.
- Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!
- All recurrent neural networks have the form of a chain of repeating modules of neural network.
- In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

RNN vs LSTM

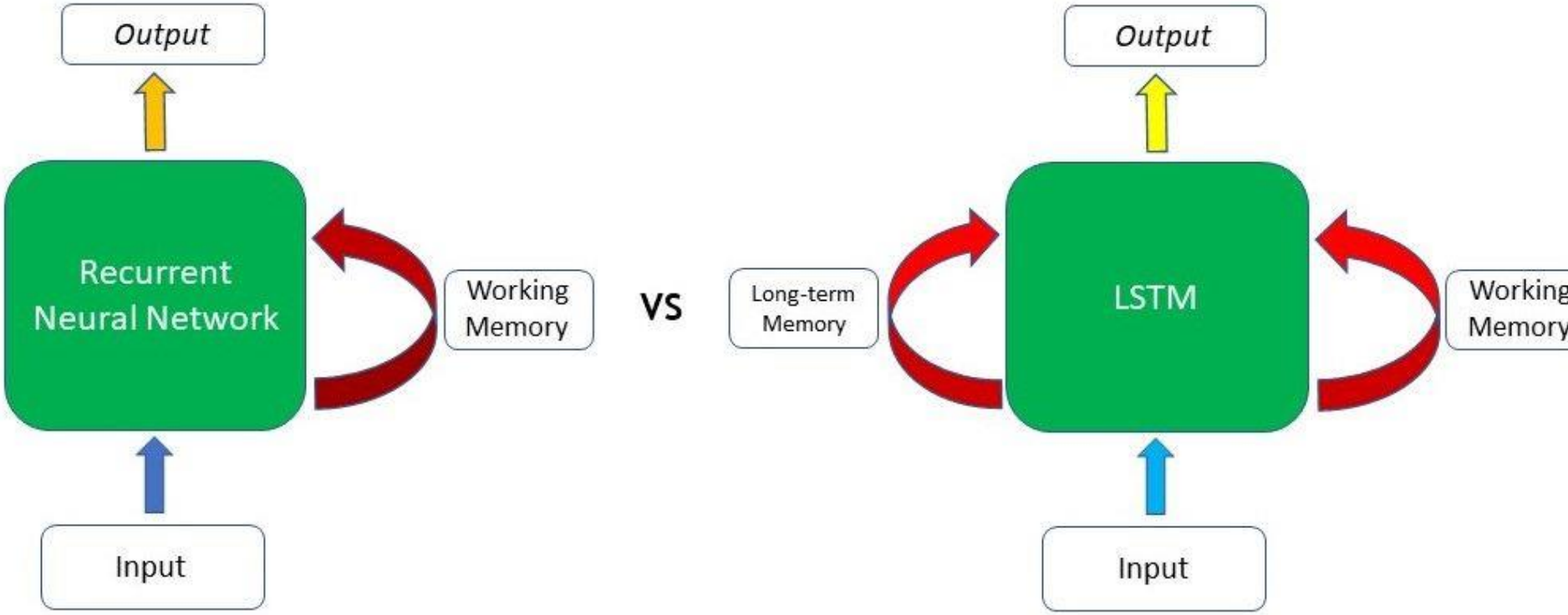
- The repeating module in a standard RNN contains a single layer.



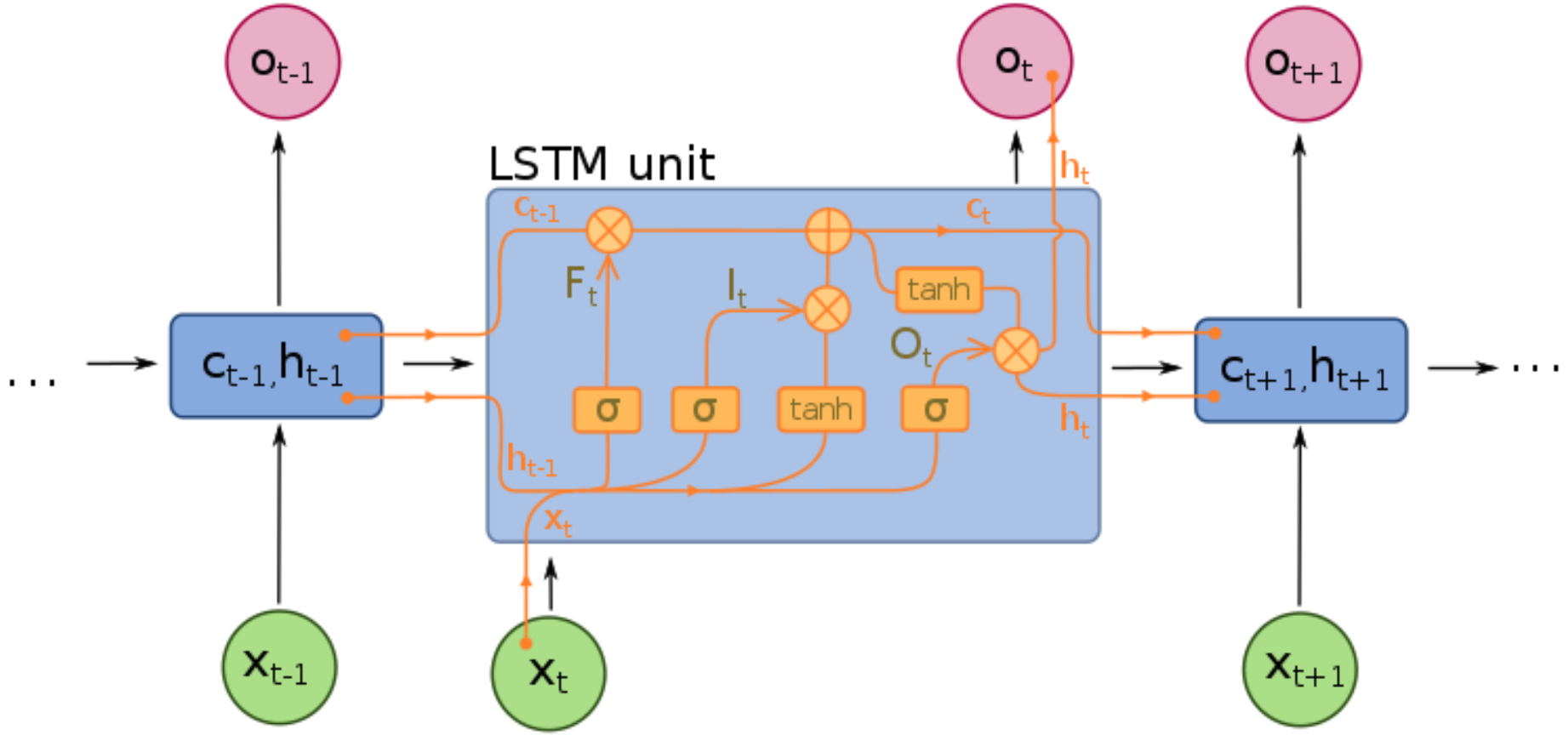
- The repeating module in an LSTM contains four interacting layers.



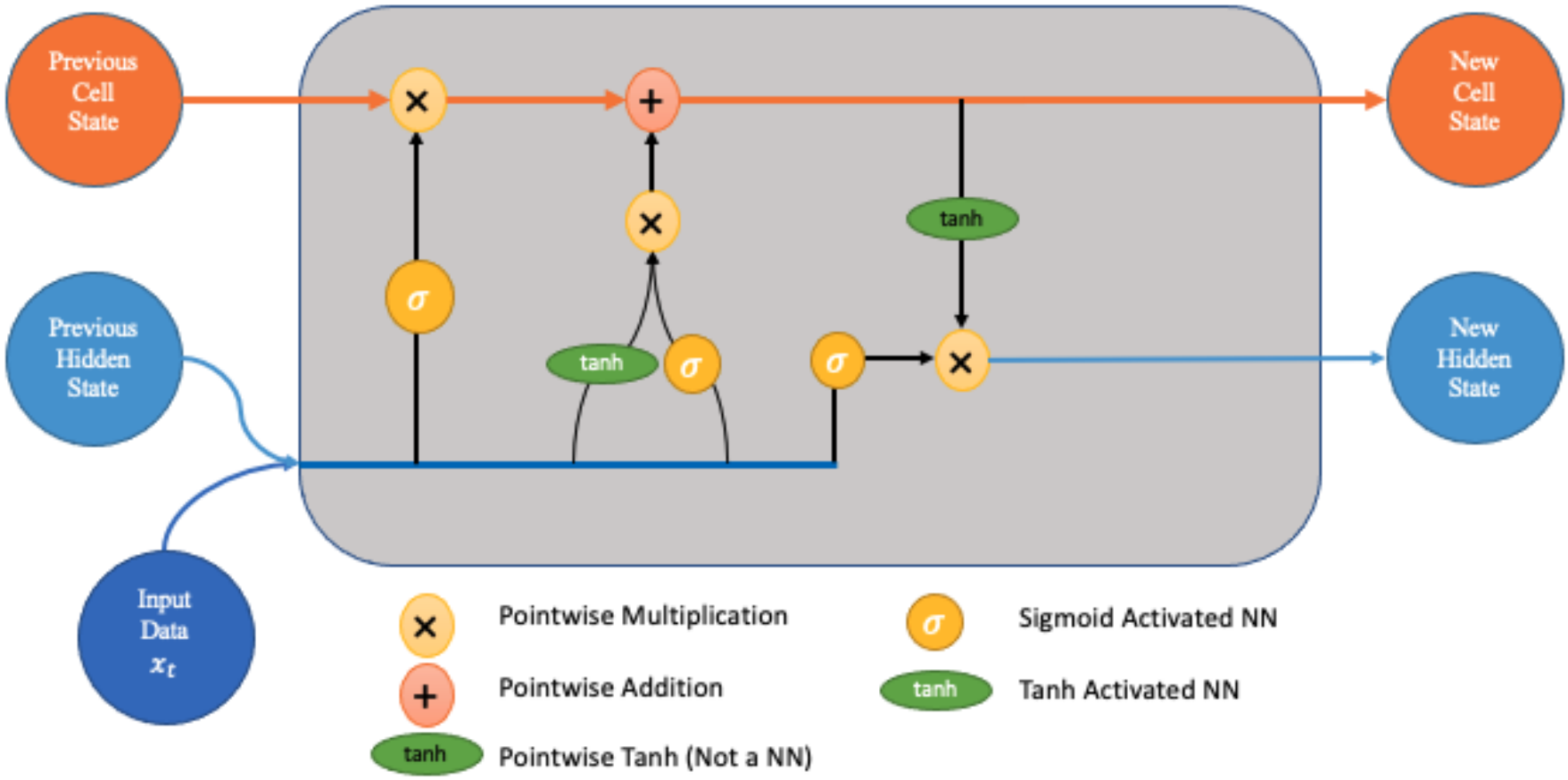
RNN vs LSTM


Output
Recurrent Neural Network
Working Memory
Input
vs
Output
Long-term Memory
LSTM
Working Memory
Input

LSTM at time t
option 1

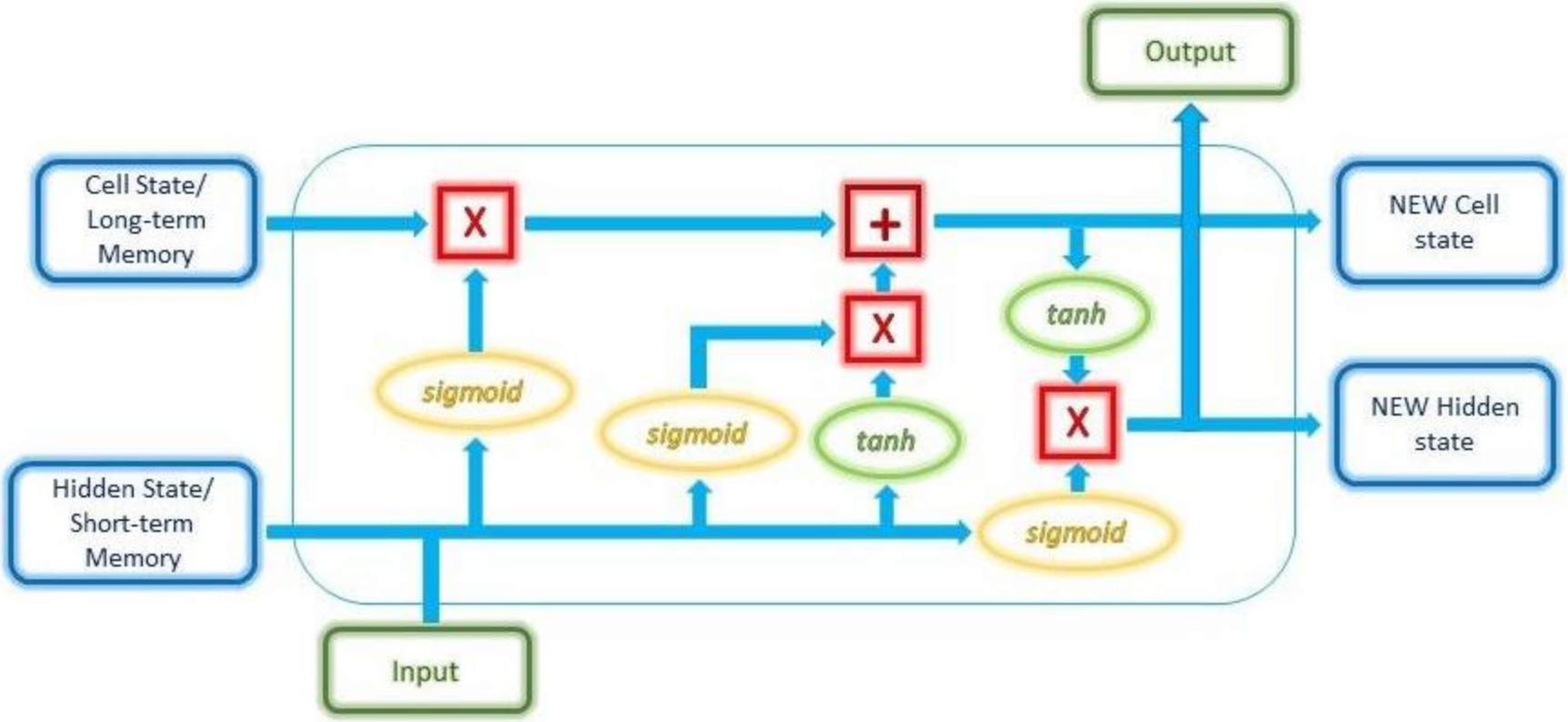


LSTM at time t
option 2

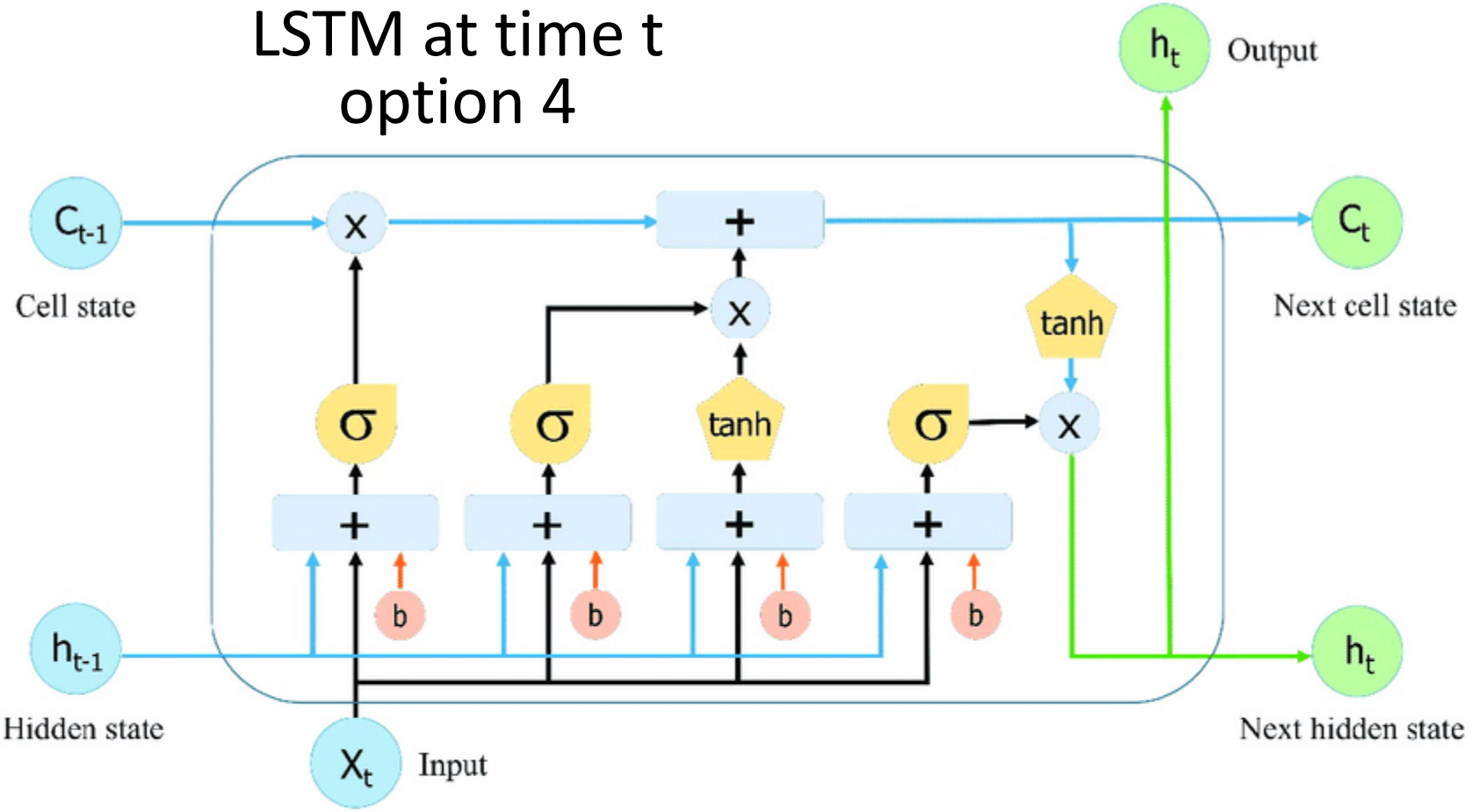


Note: the output is missing in this image, which is basically same as the new hidden state.

LSTM at time t
option 3



LSTM at time t option 4



Inputs:

- X_t Current input
- C_{t-1} Memory from last LSTM unit
- h_{t-1} Output of last LSTM unit

Outputs:

- C_t New updated memory
- h_t Current output

Nonlinearities:

- σ Sigmoid layer
- tanh Tanh layer
- b Bias

Vector operations:

- x Scaling of information
- \+ Adding information

Anatomy of an LSTM

- LSTMs are *stateful* and they contain three gates
  - (forget gate, input gate, and an output gate) that involve a sigmoid function, and also a cell state that involves the tanh activation function.
- At time period t the input to an LSTM is based on a combination of the two vectors h(t-1) and x(t).
- This pair of inputs is combined, after which a sigmoid activation function is applied to this combination (which can also include a bias vector) in the case of the forget gate, input gate, and the output gate.

Anatomy of an LSTM

1. Cell — Every unit of the LSTM network is known as a "cell". Each cell is composed of 3 inputs —

- *x(t)* — token at timestamp t
- *h(t−1)* — previous hidden state
- *c(t-1)* — previous cell state,

and 2 outputs —

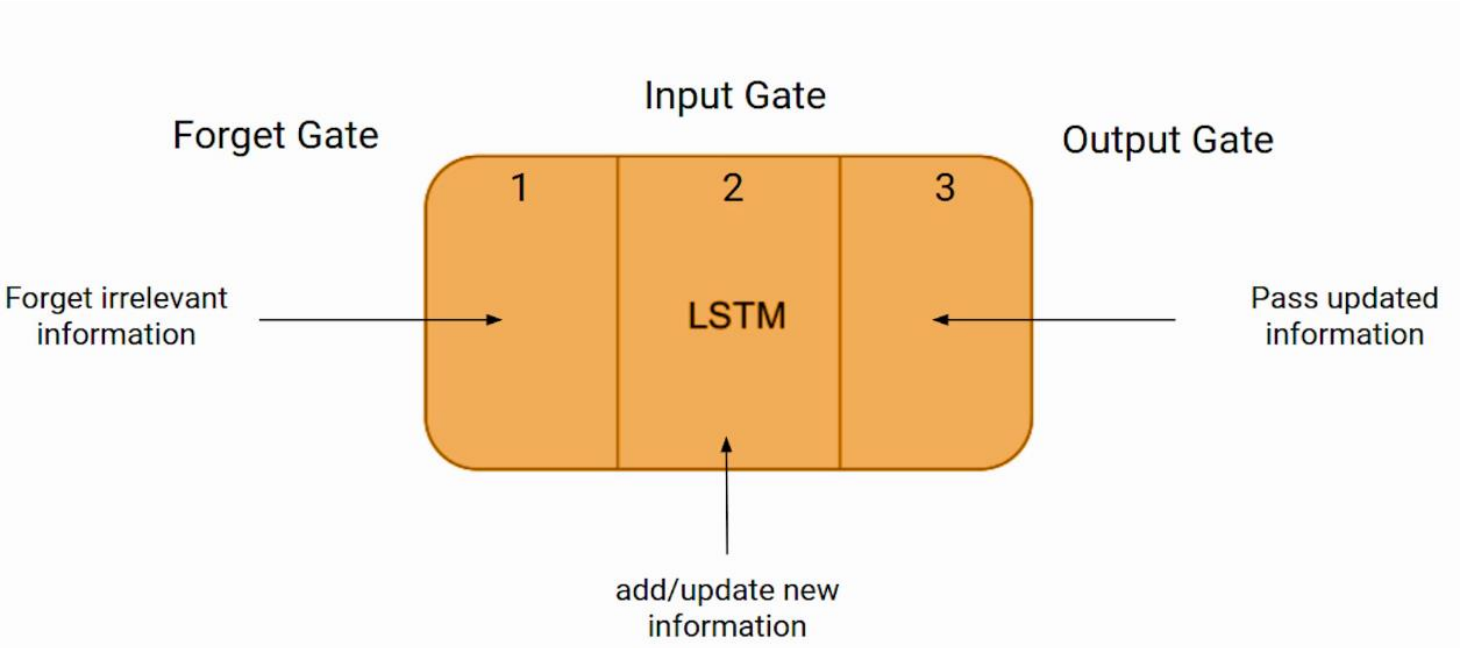
- *h(t)* — updated hidden state, used for predicting the output
- *c(t)* — current cell state

2. Gates — LSTM uses a special theory of controlling the memorizing process. Popularly referred to as gating mechanism in LSTM, what the gates in LSTM do is, store the memory components in analog format, and make it a probabilistic score by doing point-wise multiplication using sigmoid activation function, which stores it in the range of 0–1. Gates in LSTM regulate the flow of information in and out of the LSTM cells.

Gates are of 3 types —

- Input Gate — This gate lets in optional information necessary from the current cell state. It decides which information is relevant for the current input and allows it in.
- Output Gate — This gate updates and finalizes the next hidden state. Since the hidden state contains critical information about previous cell inputs, it decides for the last time which information it should carry for providing the output.
- Forget Gate — Pretty smart in eliminating unnecessary information, the forget gate multiplies 0 to the tokens which are not important or relevant and lets it be forgotten forever.

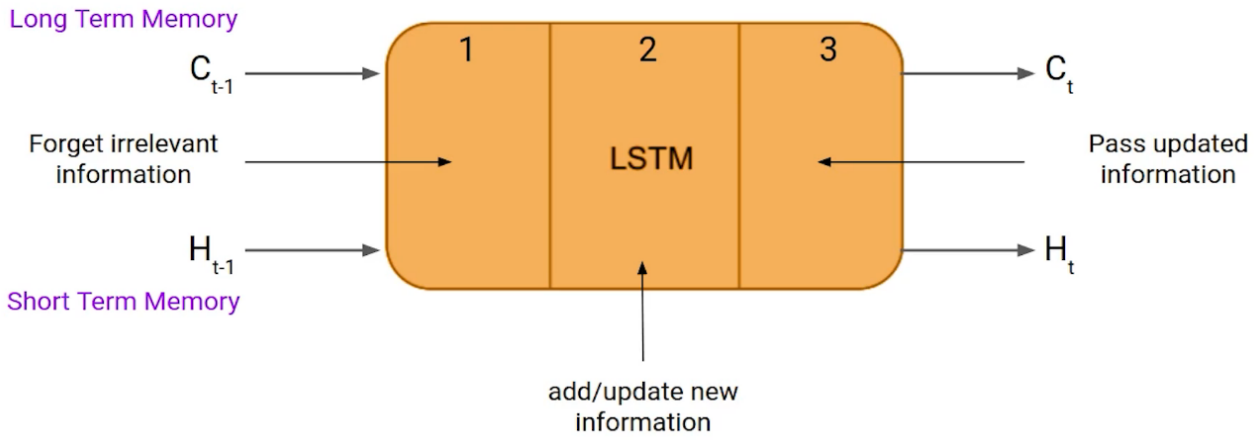
Three Gates in a LSTM cell



The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten.
In the second part, the cell tries to learn new information from the input to this cell.
At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp.
These three parts of an LSTM cell are known as gates. The first part is called **Forget gate, the** second part is known as **the Input gate** and the last one is **the Output gate**.

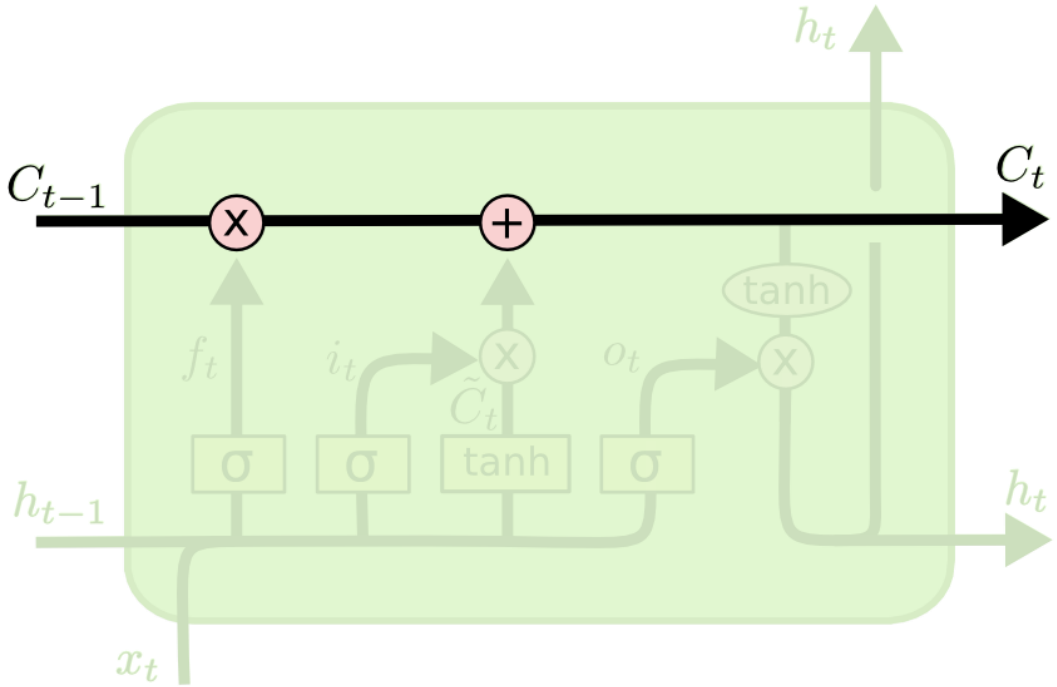
Long term memory
short term memory

- Just like a simple RNN, an LSTM also has a hidden state where $H_{(t-1)}$ represents the hidden state of the previous timestamp and H_t is the hidden state of the current timestamp.
- In addition to that LSTM also have a cell state represented by $C_{(t-1)}$ and C_t for previous and current timestamp respectively.
- Here the hidden state is known as Short term memory and the cell state is known as Long term memory.



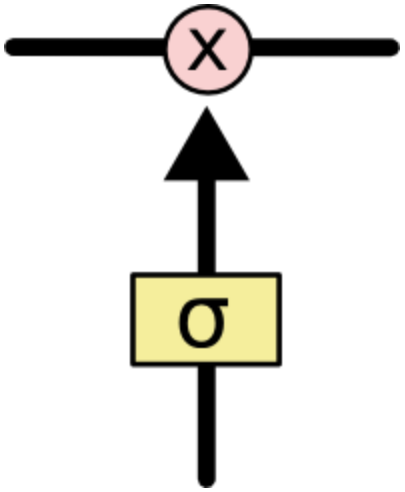
The Core Idea Behind LSTMs

- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
- The cell state is kind of like a conveyor belt.
- It runs straight down the entire chain, with only some minor linear interactions.
- It's very easy for information to just flow along it unchanged.



The Core Idea Behind LSTMs

- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
- Gates are a way to optionally let information through.
- They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

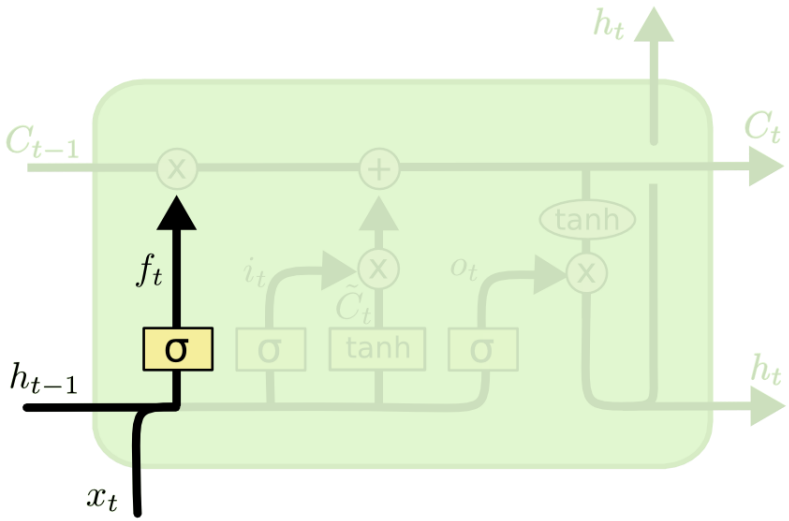


- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.
- A value of zero means “let nothing through,” while a value of one means “let everything through!”
- An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM Walk Through

STEP 1

- The first step in our LSTM is to decide what information we're going to throw away from the cell state.
- This decision is made by a sigmoid layer called the "forget gate layer."
- It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.
  - A 1 represents "completely keep this" while a 0 represents "completely get rid of this."



f_t = sigma(W_f*x_t + U_f*h_{t-1} + b_f)
or

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

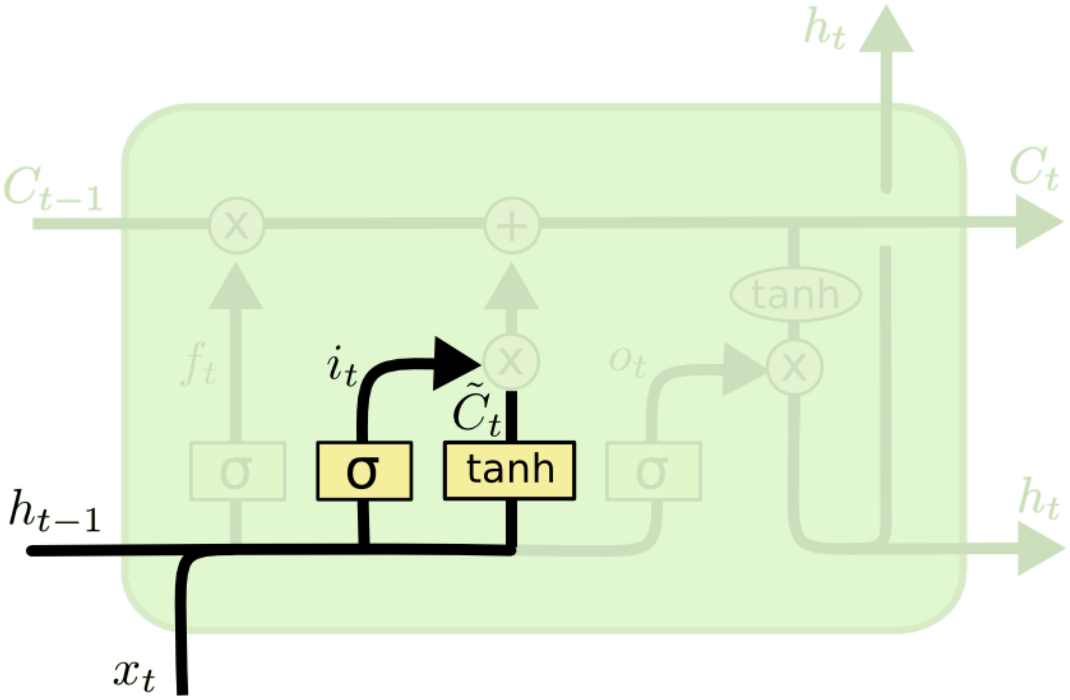
Step-by-Step LSTM Walk Through

STEP 2

- The next step is to decide what new information we're going to store in the cell state.
- This has two parts.
  - First, a sigmoid layer called the "input gate layer" decides which values we'll update.
  - Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$ that could be added to the state.
- In the next step, we'll combine these two to create an update to the state.

Step-by-Step LSTM Walk Through

STEP 2



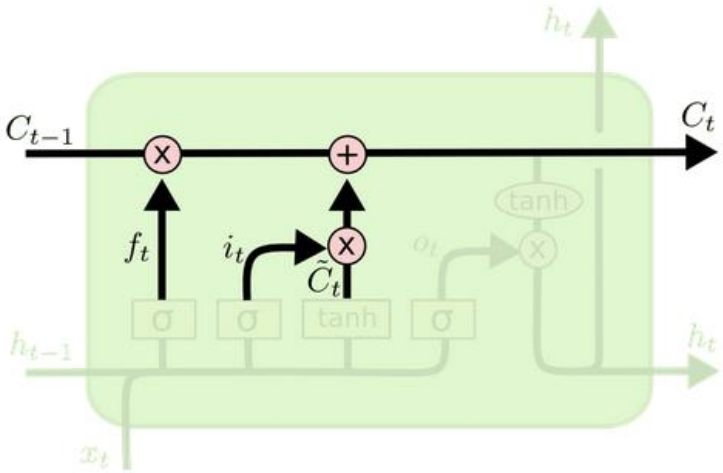
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Step-by-Step LSTM Walk Through

STEP 3

- It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.
- We multiply the old state ($\tilde{C}_t$) by f_t, forgetting the things we decided to forget earlier. Then we add $i_t *$.
- This is the new candidate values, scaled by how much we decided to update each state value.

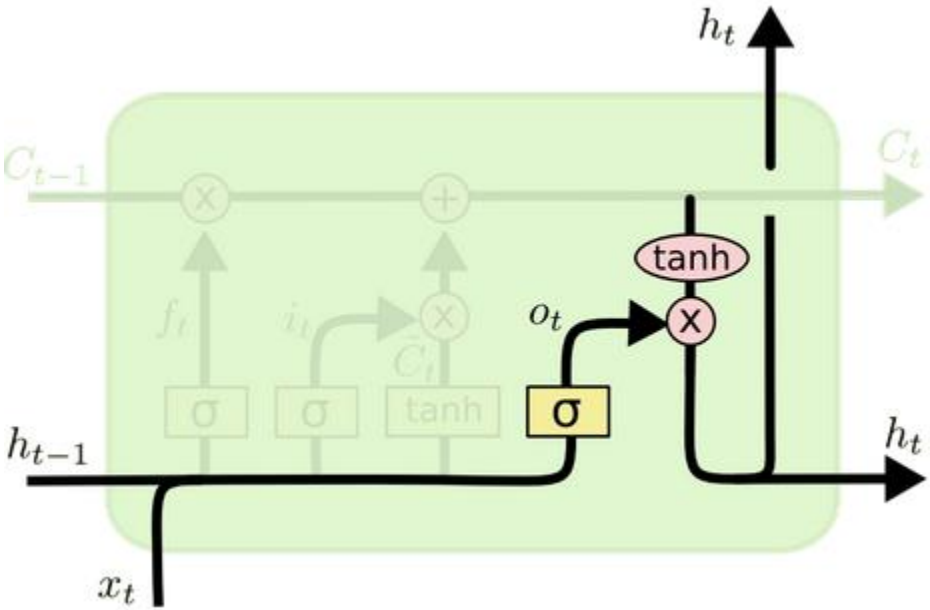


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step-by-Step LSTM Walk Through

STEP 4

- Finally, we need to decide what we're going to output.
- This output will be based on our cell state, but will be a filtered version.
- First, we run a sigmoid layer which decides what parts of the cell state we're going to output.
- Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.
  - For a language model example, if it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next.
  - For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Bidirectional LSTMs

- In addition to one-directional LSTMs, you can also define a *bidirectional* LSTM that consists of two regular LSTMs:
  - one LSTM for the forward direction and
  - one LSTM in the backward or opposite direction.
- Bidirectional LSTMs are well suited for solving NLP tasks.
- For instance, ELMo is a deep word representation for NLP tasks that uses bidirectional LSTMs.
- An even newer architecture in the NLP world is called a *transformer*, and bidirectional transformers are used in BERT, which is a very well-known system (released by Google in 2018) that can solve complex NLP problems.

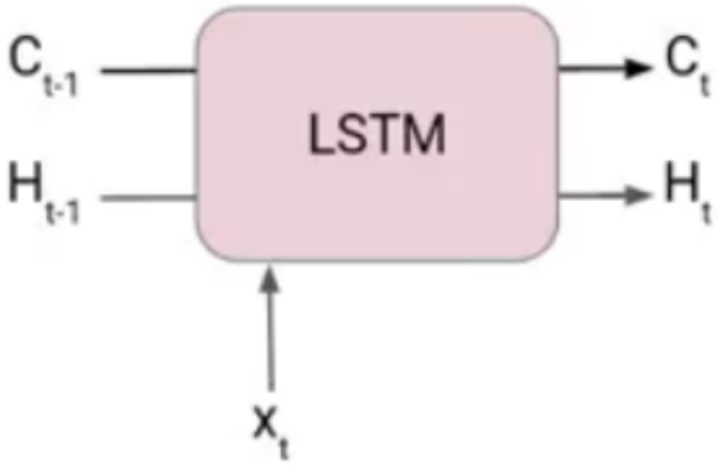
LSTM and GRU

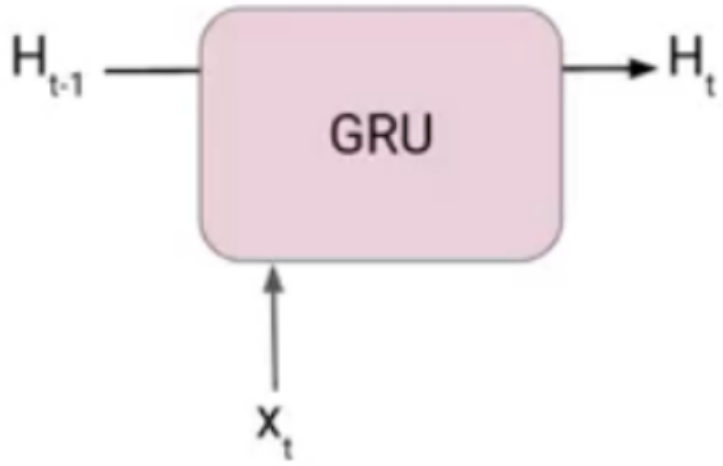
- Long Short Term Memory Network is an advanced RNN, a sequential network, that allows information to persist. It is capable of handling the vanishing gradient problem faced by RNN.
- Gated recurrent units (GRUs) were later introduced as a simpler alternative to LSTM, and have also become quite popular.

Gated Recurrent Unit

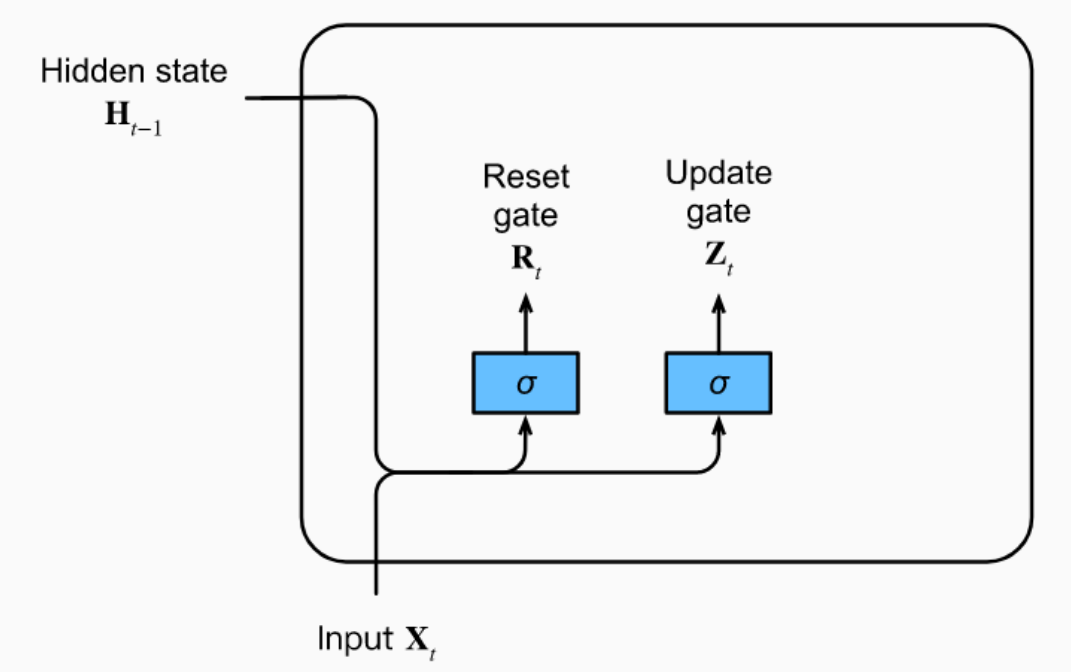
- A GRU (Gated Recurrent Unit) is an RNN that is a simplified type of LSTM. It was introduced by Kyunghyun Cho et al in the year 2014.
- The key difference between a GRU and an LSTM: a GRU has two gates (reset and update gates) whereas an LSTM has three gates (forget, input, and output gates).
- The reset gate in a GRU performs the functionality of the input gate and the forget gate of an LSTM.
- Keep in mind that GRUs and LSTMs both have the goal of tracking long term dependencies effectively, and they both address the problem of vanishing gradients and exploding gradients.

LSTM vs GRU





The reset gate and the update gate in a GRU model.



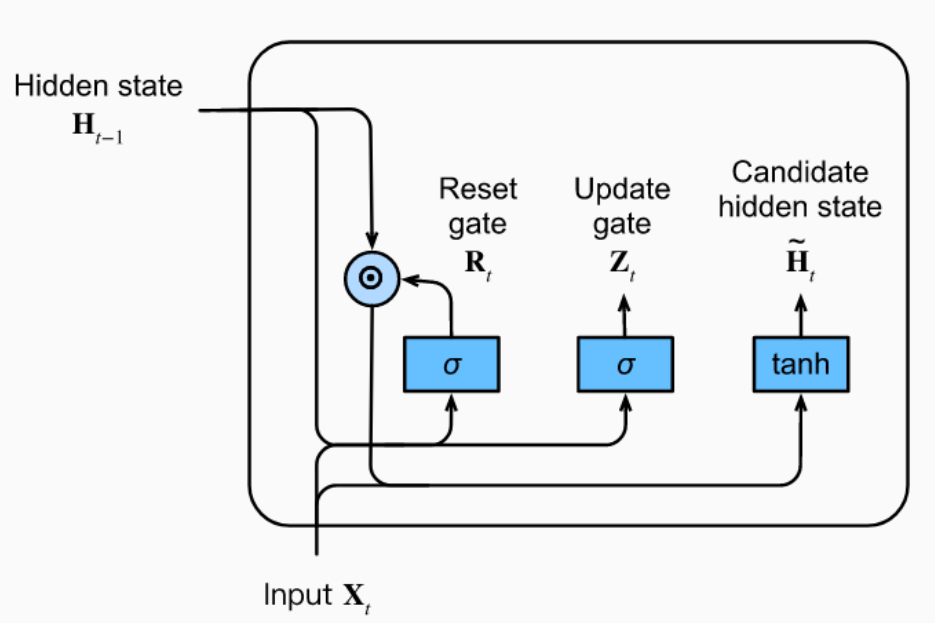
$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),$$

Sigmoid functions transform input values to the interval (0,1)

Computing the hidden state in a GRU

- To find the Hidden state Ht in GRU, it follows a two-step process.
- The first step is to generate what is known as the candidate hidden state.
- It takes in the input and the hidden state from the previous timestamp t-1 which is multiplied by the reset gate output R_t. Later passed this entire information to the tanh function, the resultant value is the candidate's hidden state.
- The most important part of this equation is how we are using the value of the reset gate to control how much influence the previous hidden state can have on the candidate state.
- If the value of R_t is equal to 1 then it means the entire information from the previous hidden state H_{t-1} is being considered. Likewise, if the value of R_t is 0 then that means the information from the previous hidden state is completely ignored.

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h),$$



Calculate the update state

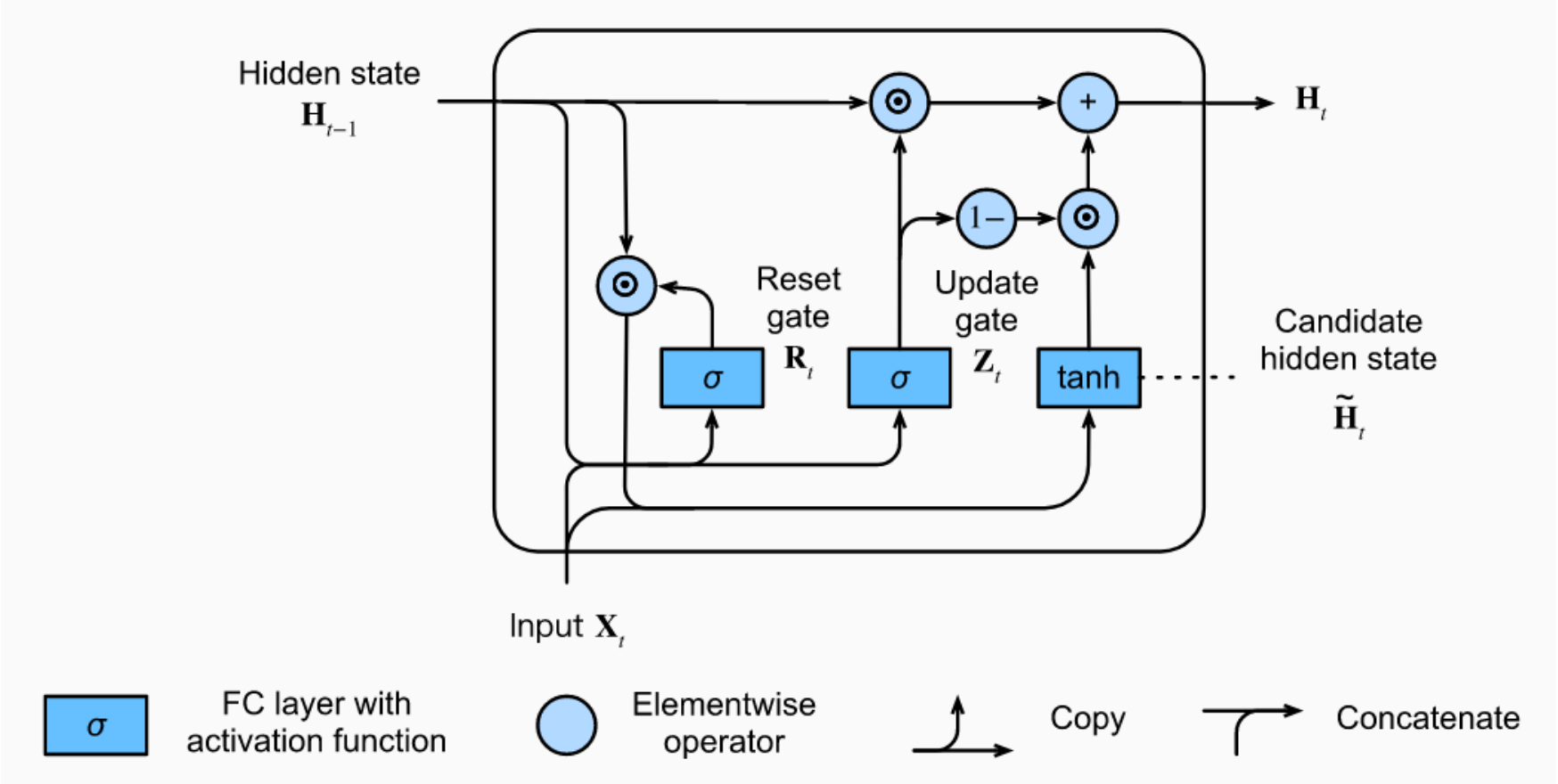
- Once we have the candidate state, it is used to generate the current hidden state H_t.
- It is where the Update gate comes into the picture.
- Now, this is a very interesting equation, instead of using a separate gate like in LSTM in GRU we use a single update gate to control both the historical information which is H_{t-1} as well as the new information which comes from the candidate state.
- the final update equation for the GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

- Now assume the value of Z_t is around 0 then the first term in the equation will vanish which means the new hidden state will not have much information from the previous hidden state. On the other hand, the second part becomes almost one that essentially means the hidden state at the current timestamp will consist of the information from the candidate state only.

- These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for sequences with large time step distances.

Complete GRU model

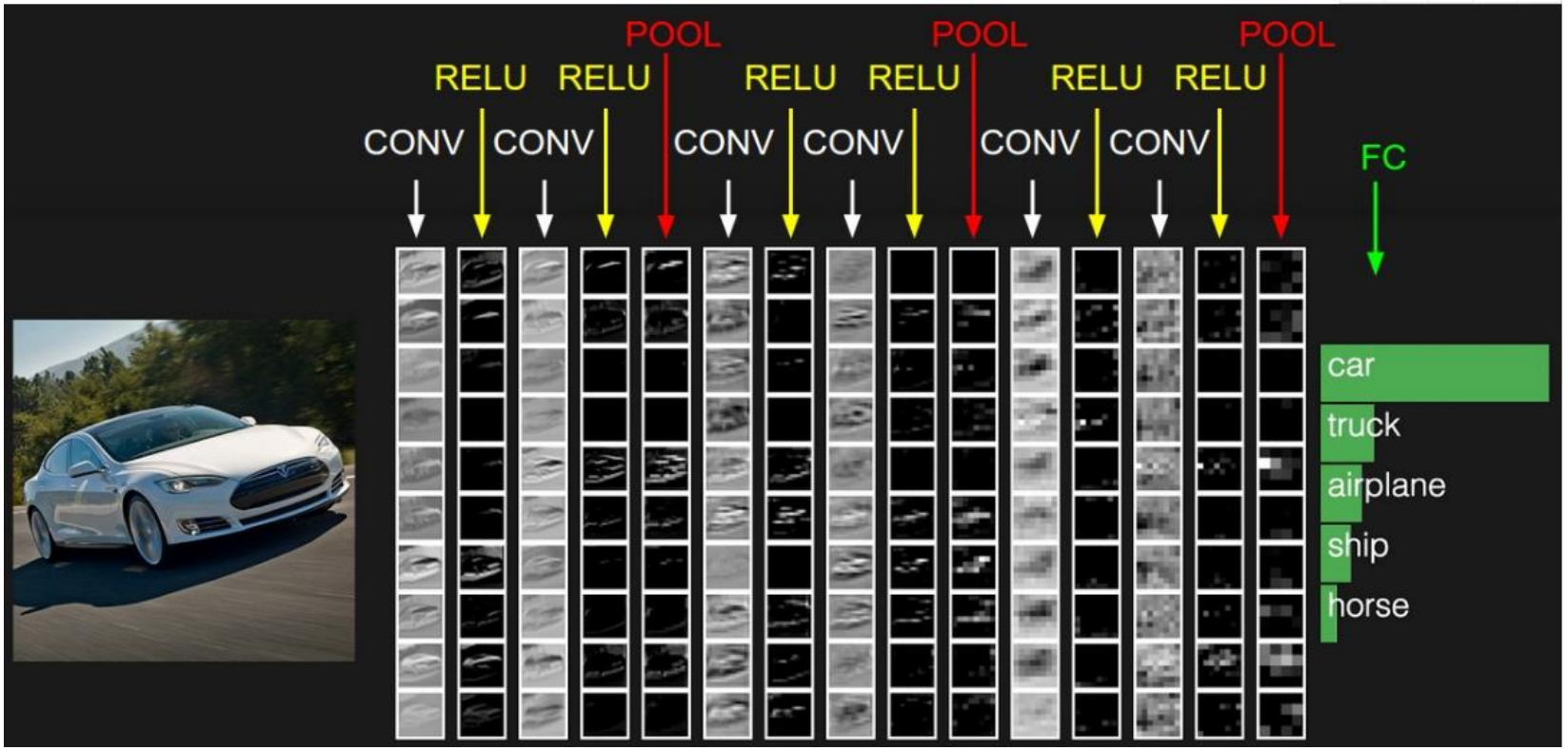


- In summary, GRUs have the following two features:
  - Reset gates help capture short-term dependencies in sequences.
  - Update gates help capture long-term dependencies in sequences.

- GRU or Gated recurrent unit is an advancement of the standard RNN i.e recurrent neural network.

CONVOLUTIONAL NEURAL NETWORKS (CNN)

A simple CNN structure



CONV: Convolutional kernel layer
RELU: Activation function
POOL: Dimension reduction layer
FC: Fully connection layer

What an image looks like in computer memory?

$\longrightarrow$

148	123	52	107	123	162	172	123	64	89 ···
147	130	92	95	98	130	171	155	169	163 ···
141	118	121	148	117	107	144	137	136	134 ···
82	106	93	172	149	131	138	114	113	129 ···
57	101	72	54	109	111	104	135	106	125 ···
138	135	114	82	121	110	34	76	101	111 ···
138	102	128	159	168	147	116	129	124	117 ···
113	89	89	109	106	126	114	150	164	145 ···
120	121	123	87	85	70	119	64	79	127 ···
145	141	143	134	111	124	117	113	64	112 ···
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In a (8-bit) greyscale image each picture element has an assigned intensity that ranges from 0 to 255.

Understanding the image data

- Image
- Relationship in pixels
- Existing ANN don’t model the relationship between different features.
  - They just consider all of them individually.
  - And don’t model their spatial relationships.
  - No two columns in tabular data have any spatial relationships.
  - They can be put in just any order.
- In an image, two pixels are together for a reason.
  - There are patterns of pixels in an image that help us to identify the image.
  - If all the pixels are scrambled, will we be able to identify the image?
  - Because in that case the patterns will be disturbed.

Understanding image data

- Image data is different
  - Topology of pixels
    - a natural topology of pixels. A meaningful Spatial relationship. Two pixels are together for a reason.
  - Translation invariance
    - To detect an object in an image, its orientation, location should not matter
  - Scale invariance
    - Big cat or small cat
  - Individual pixels don’t add much value
  - Identify Edges and shapes
  - Issues of lighting and contrast
    - Pixel density may change due to lighting and contrast
  - Understanding of human visual system

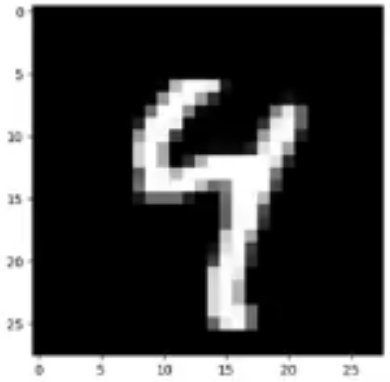
Understanding image data

Motivation – Image Data

Fully connected image networks would require a vast number of parameters.

- MNIST images are small (28 x 28 pixels), and in grayscale

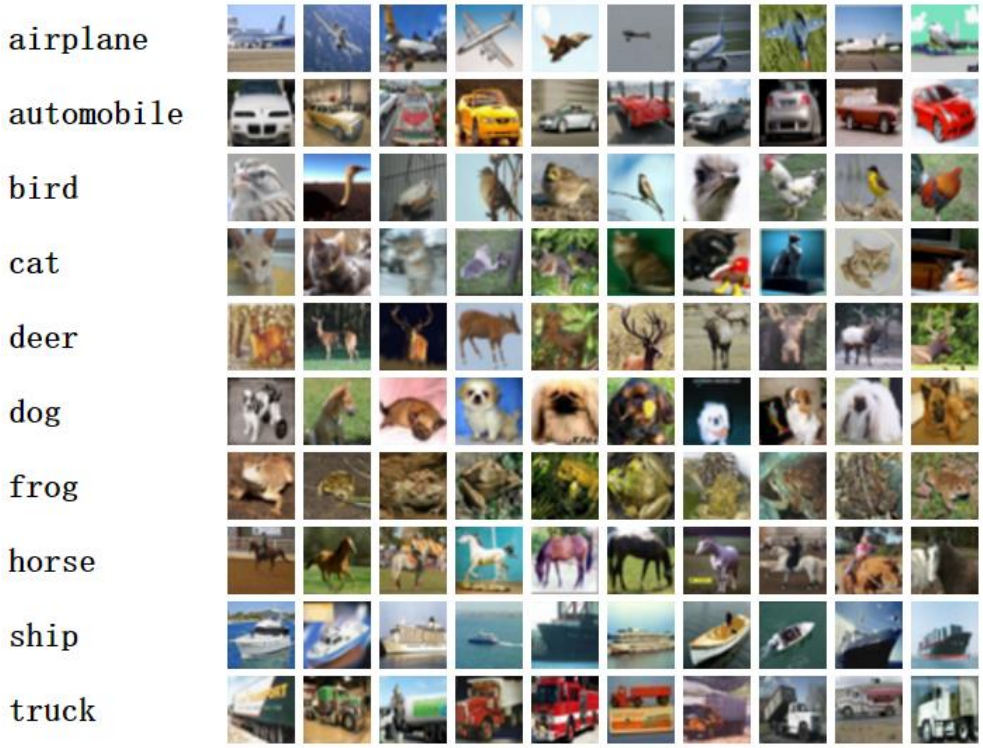
- Color images typically contain:
  [(200 x 200) pixels] x [3 color channels (RGB)] =
  120,000 values (features).



Example MNIST image

The CIFAR10 dataset

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes,
with 6000 images per class. There are 50000 training images and 10000 test images.



ImageNet

- The **ImageNet** project is a large visual database designed for use in visual object recognition software research.
- As of 2016, over ten million URLs of images have been hand-annotated by ImageNet to indicate what objects are pictured; in at least one million of the images, bounding boxes are also provided.[1]
- The database of annotations of third-party image URL's is freely available directly from ImageNet; however, the actual images are not owned by ImageNet.[2]
- Since 2010, the ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes.

Detecting object features

- Convolutional Network architecture follows that object features e.g. face are built up in stages, e.g.:
  - Identify edges
  - Then edges built-up into shape
  - Then texture

Example: Cat = [two eyes in certain relation to one another] + [cat fur texture].

- Eyes = dark circle (pupil) inside another circle.
- Circle = particular combination of edge detectors.
- Fur = edges in certain pattern.

What is Convolution?

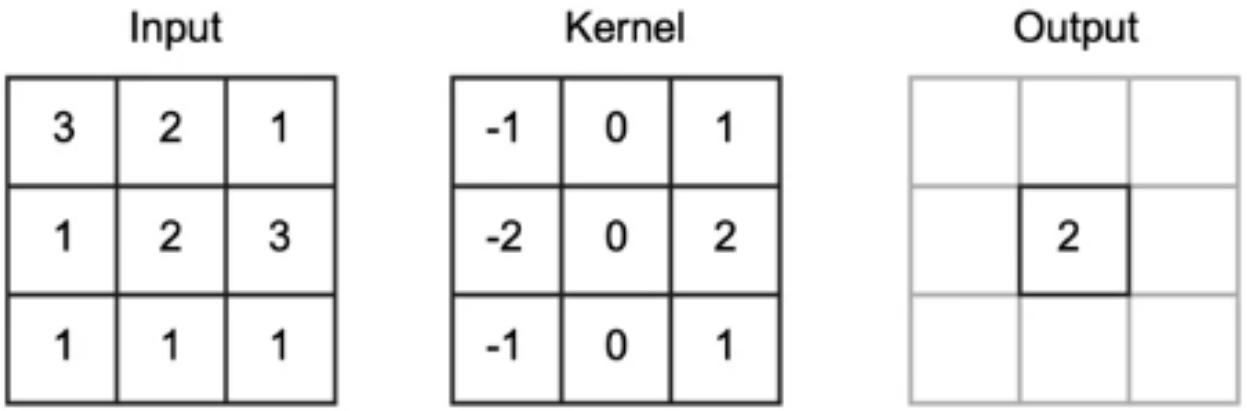
- To capture the relationship in different pixels of an image.
- Use a kernel which is a grid of weights overlaid on image, centred on one pixel.

-1	0	1
-2	0	2
-1	0	1

- Each weight is multiplied with pixels underneath it.
- And then all the terms are added $\sum_{p=1}^{P} W_p \cdot pixel_p$
- Used for traditional image processing:
  - Blur, sharpen, edge detection etc.

Convolutional kernel

Kernel: 3 × 3 Example

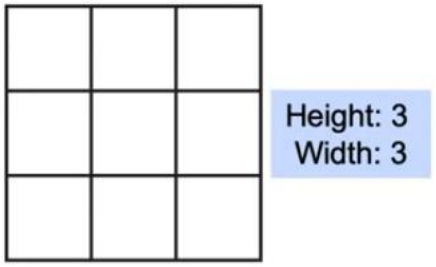


$$\text{Output} = (3 \times (-1)) + (2 \times 0) + (1 \times 1)$$
$$+ (1 \times (-2)) + (2 \times 0) + (3 \times 2)$$
$$+ (1 \times (-1)) + (1 \times 0) + (1 \times 1)$$
$$= [-3 + 1] - [2 + 6] - [1 + 1]$$
$$= 2$$

Convolution Settings – Grid Size

Grid Size (Height and Width):

- The number of pixels a kernel “sees” at once.
- Typically use odd numbers so that there is a “center” pixel.
- Kernel does not need to be square.



Height: 3
Width: 3

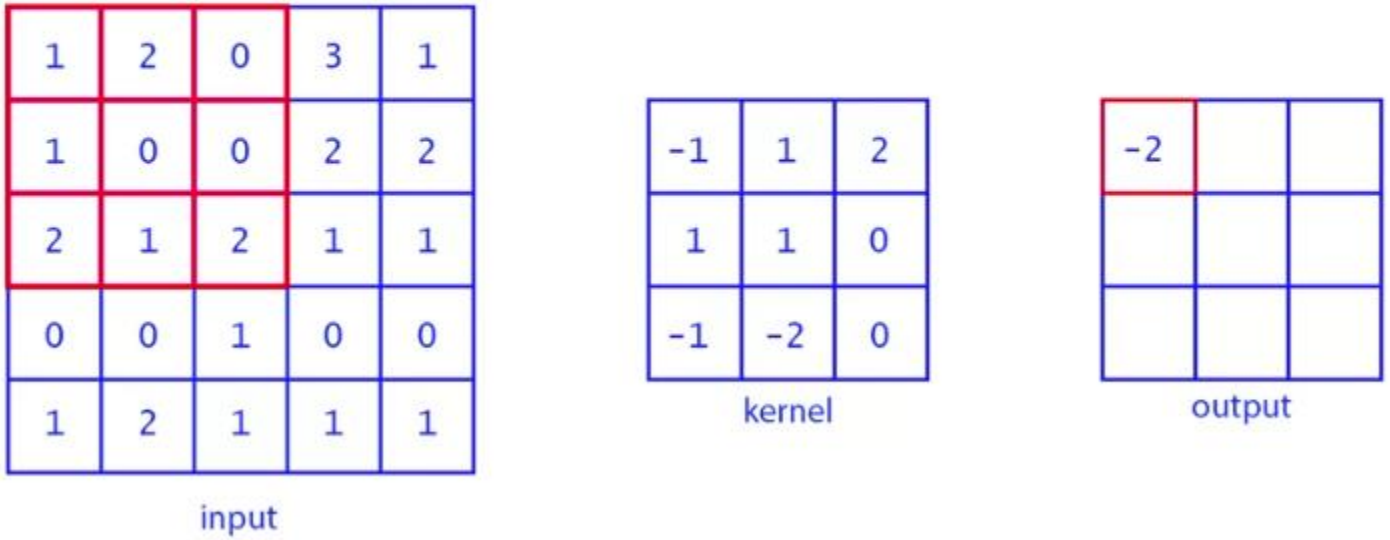
Height: 1
Width: 3

Height: 3
Width: 1

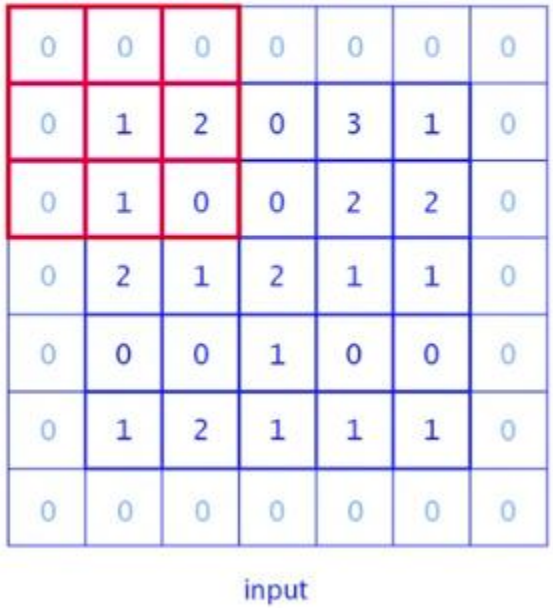
The concept of padding

- something that you may have noticed as we went through this idea of working with convolutions, is that
  - when we work with these centered values and are trying to output centered values, the edges of our image and the corners of our image tend to get somewhat overlooked.
- We're going to address this problem and introduce the concept of padding.

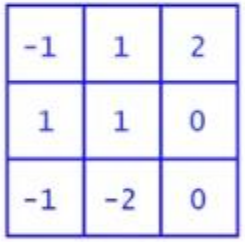
Without Padding



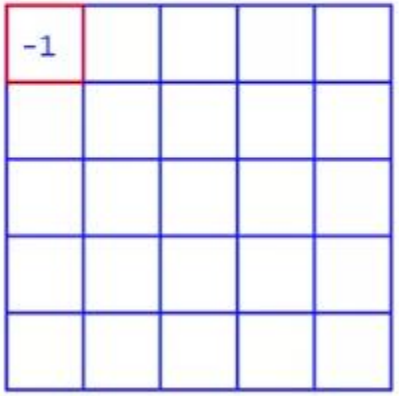
With Padding



input



kernel



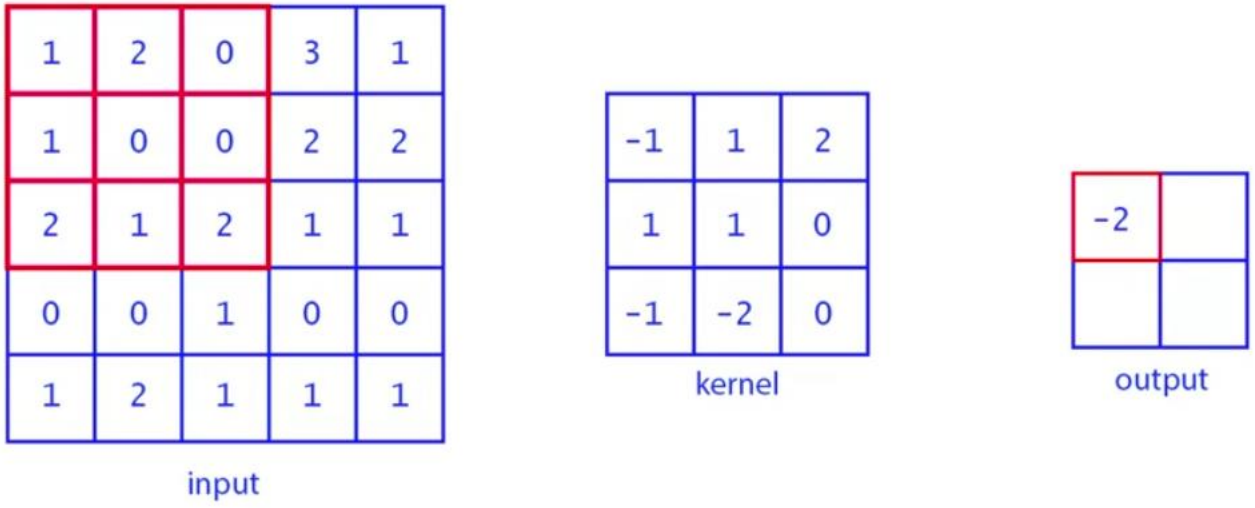
output

Convolution Settings - Padding

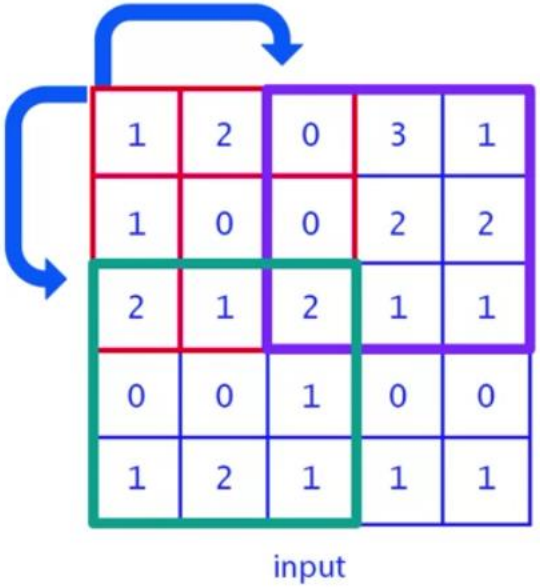
Padding

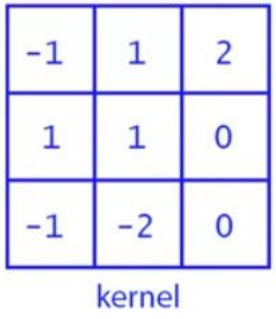
- Using Kernels directly, there will be an “edge effect”.
- Pixels near the edge will not be used as “center pixels” since there are not enough surrounding pixels.
- Padding adds extra pixels around the frame, so pixels from the original image become center pixels as the kernel moves across the image.
- Added pixels are typically of value zero (zero-padding).

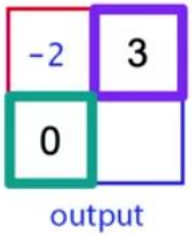
Stride 2 Example – No Padding



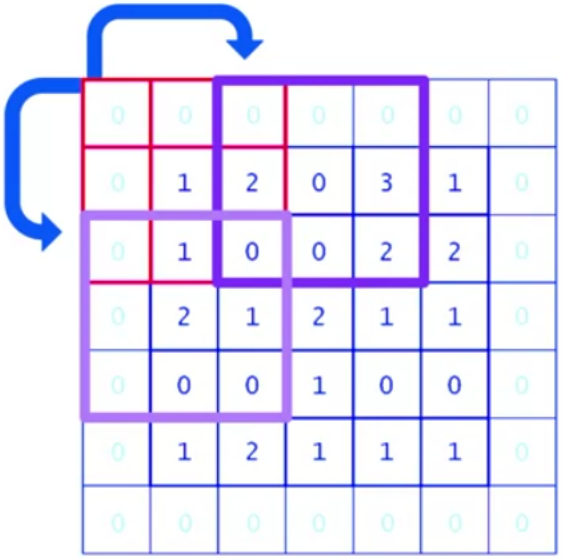
Stride 2 Example – No Padding



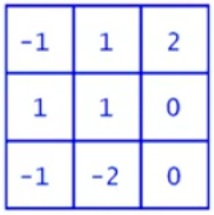




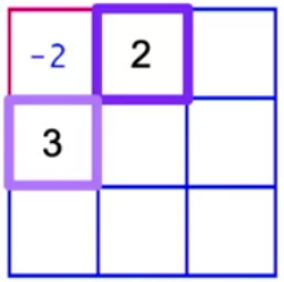
Stride 2 Example – Padding



input



kernel



output

The concept of strides

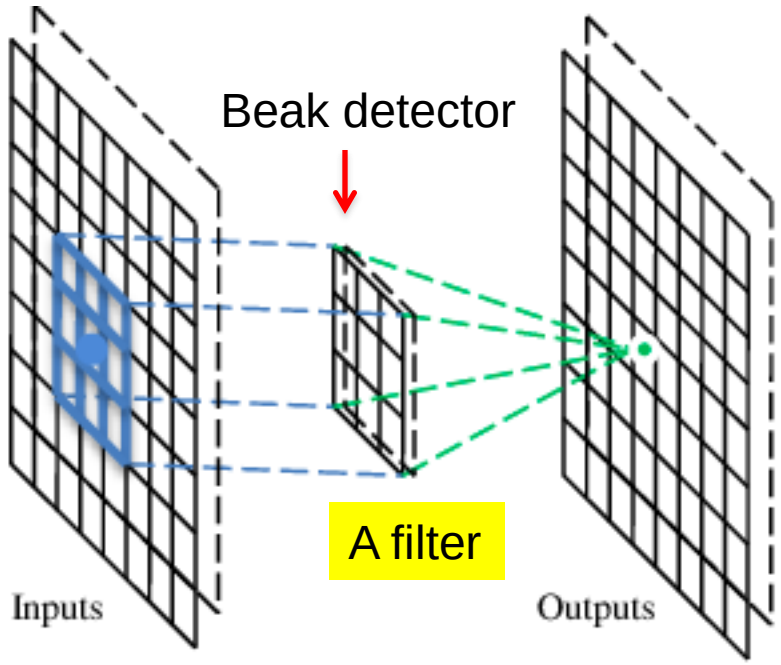
Convolution Settings

Stride

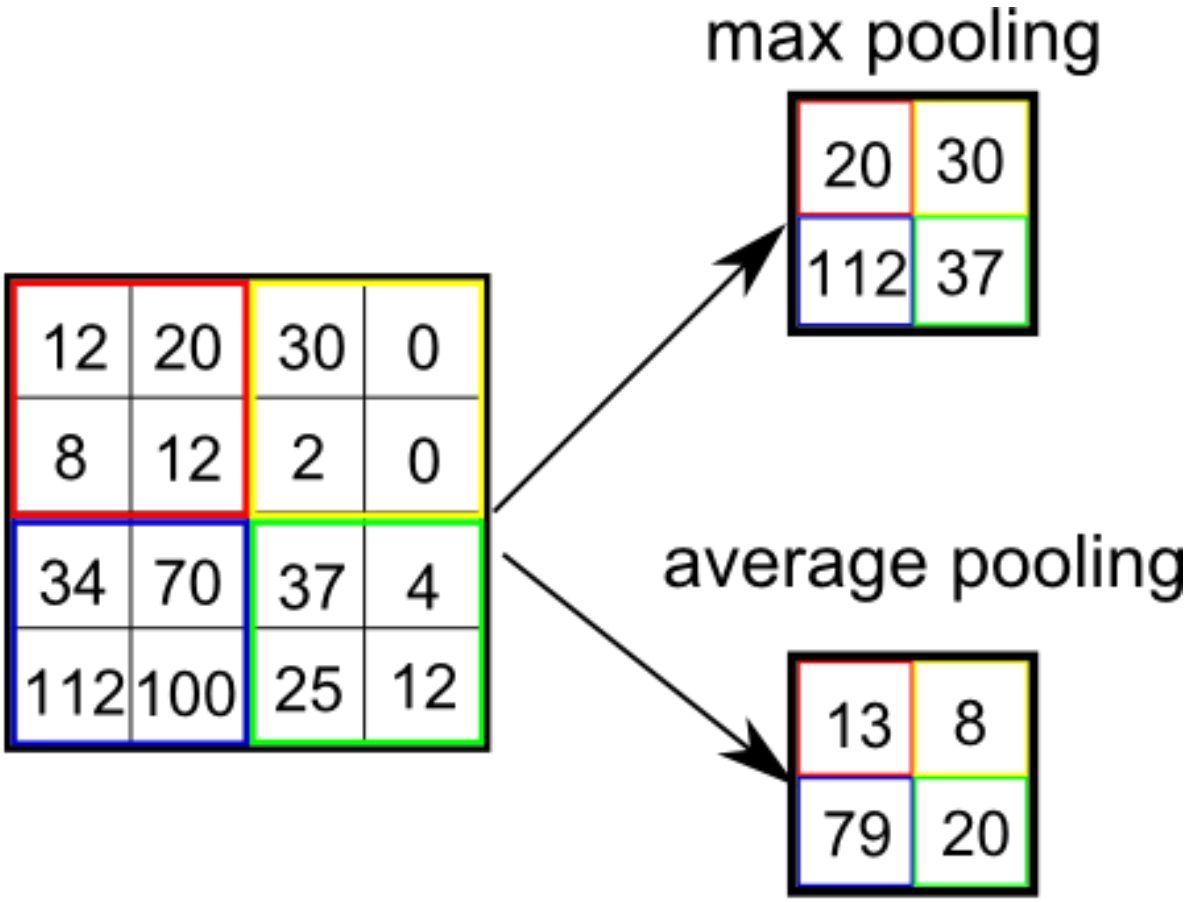
- The ”step size” as the kernel moves across the image.
- Can be different for vertical and horizontal steps (but usually is the same value).
- When stride is greater than 1, it scales down the output dimension.

A convolutional layer

A CNN is a neural network with some convolutional layers (and some other layers). A convolutional layer has a number of filters that does convolutional operation.

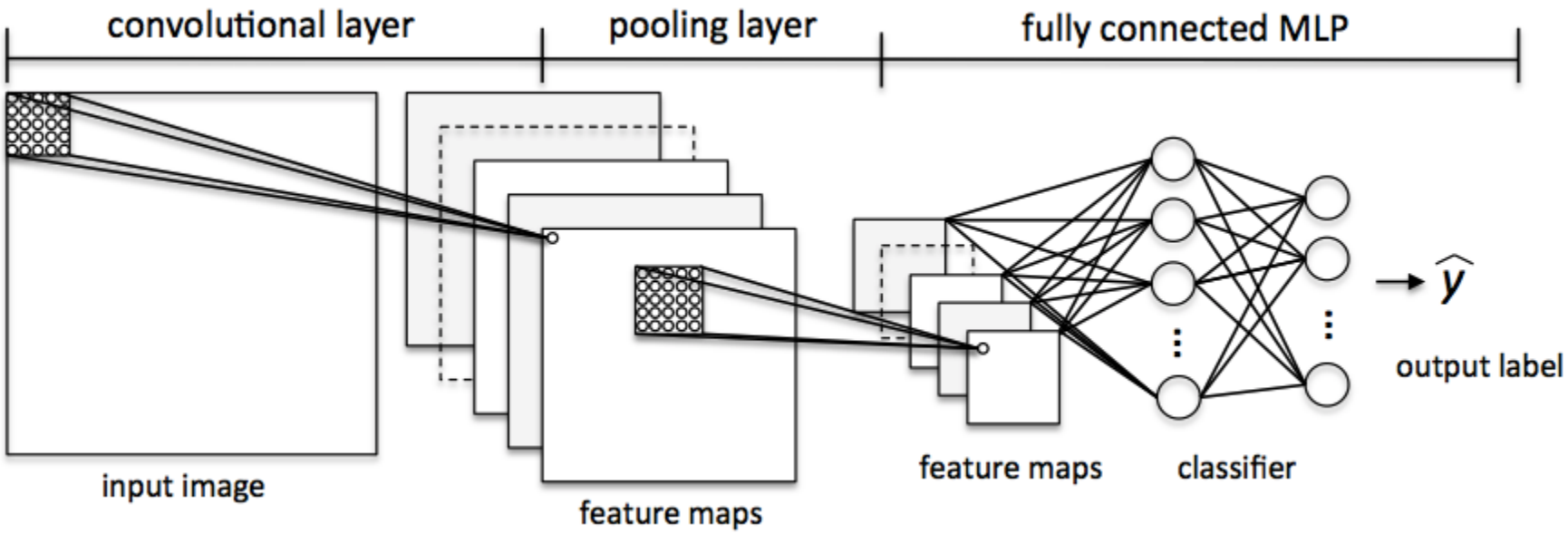


The Pooling Layer



Pooling layer

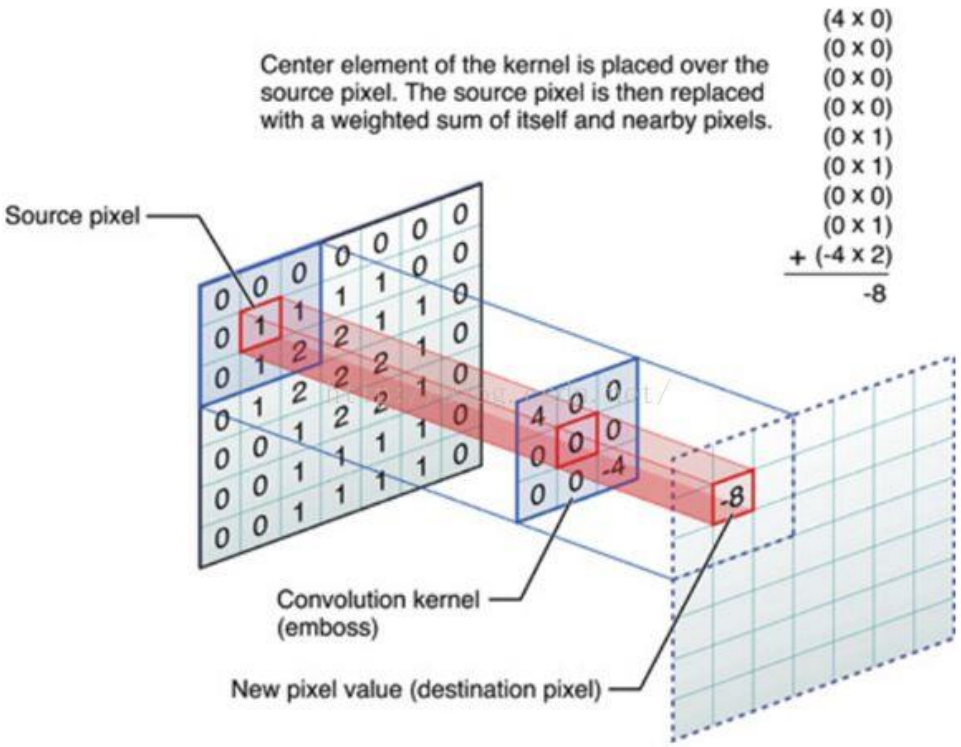
- In CNNs, a convolutional layer is followed by a **pooling layer** (sometimes also called **sub-sampling**).
- In pooling, we summarize neighboring feature detectors to reduce the number of features for the next layer.
- Pooling can be understood as a simple method of feature extraction where we take the average or maximum value of a patch of neighboring features and pass it on to the next layer.
- To create a deep convolutional neural network, we stack multiple layers—alternating between convolutional and pooling layers—before we connect it to a multi-layer perceptron for classification.
- This is shown in the following figure:


convolutional layer
pooling layer
fully connected MLP
input image
feature maps
feature maps
classifier
$\hat{y}$
output label

Does it contain a cat?

- So typically the units of our final layer should be sensitive to the entire input.
- By gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing.

Convolutional kernel



Padding on the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input

Kernels as feature detectors

Kernels as feature detectors

Can think of kernels as a "local feature detectors".

Vertical Line Detector

-1	1	-1
-1	1	-1
-1	1	-1

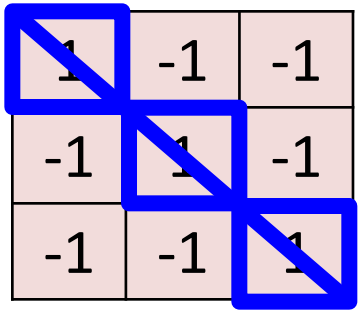
Horizontal Line Detector

-1	-1	-1
1	1	1
-1	-1	-1

Corner Detector

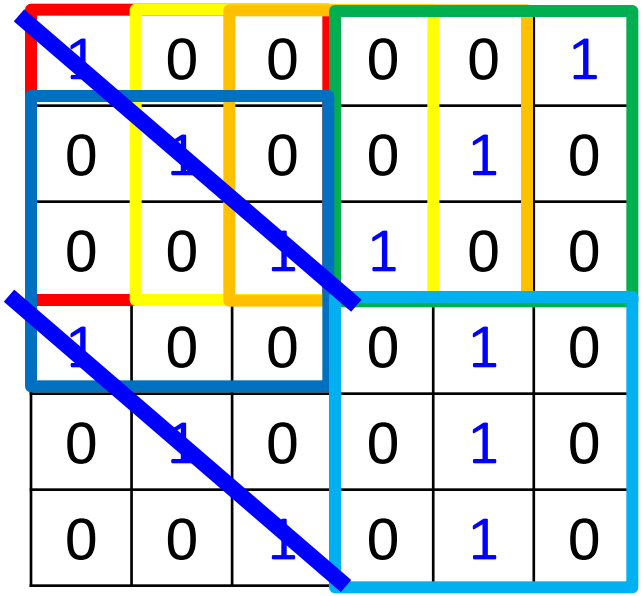
-1	-1	-1
-1	1	1
-1	1	1

Convolution

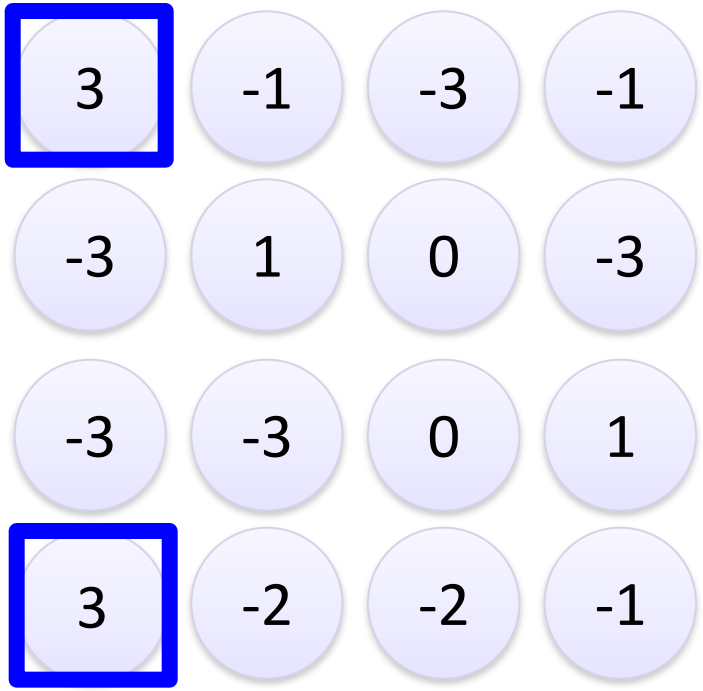


Filter 1

stride=1



6 x 6 image



Convolution

-1	1	-1
-1	1	-1
-1	1	-1

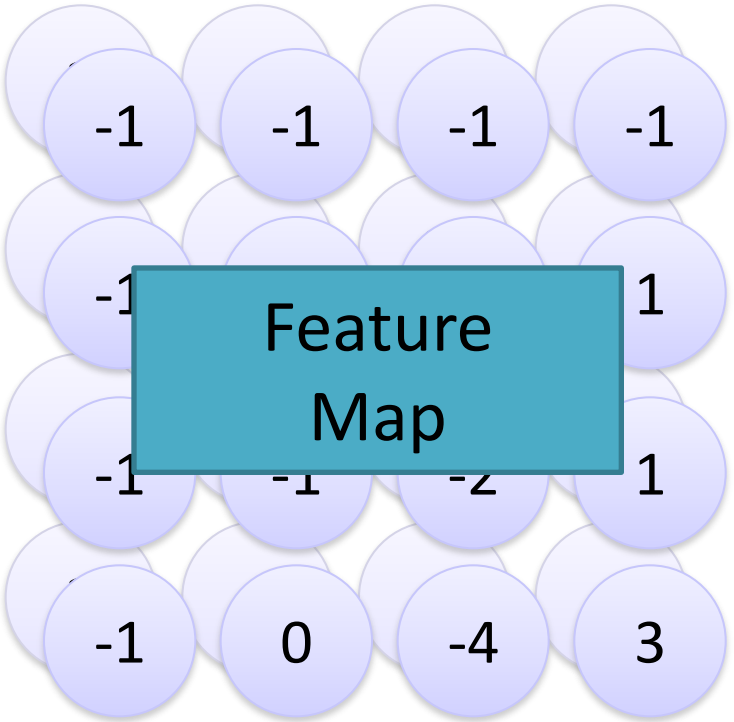
Filter 2

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Repeat this for each filter



Two 4 x 4 images
Forming 2 x 4 x 4 matrix

Multiple kernels to detect different features

These are the network parameters to be learned.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

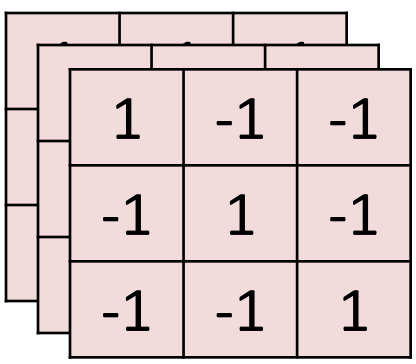
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

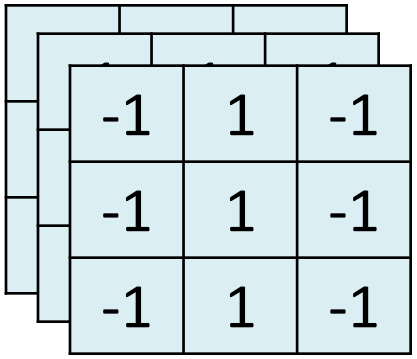
⋮ ⋮

Each filter detects a small pattern (3 x 3).

Color image: RGB 3 channels

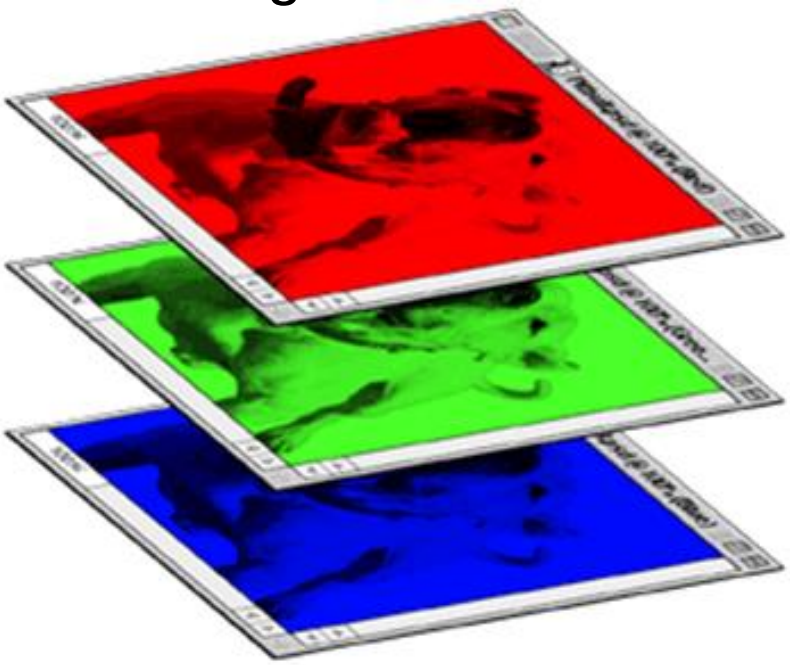


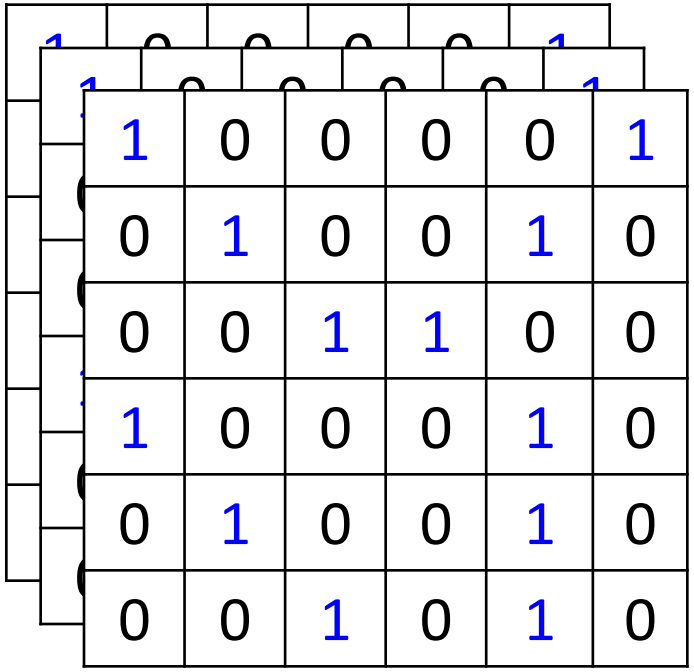
Filter 1



Filter 2

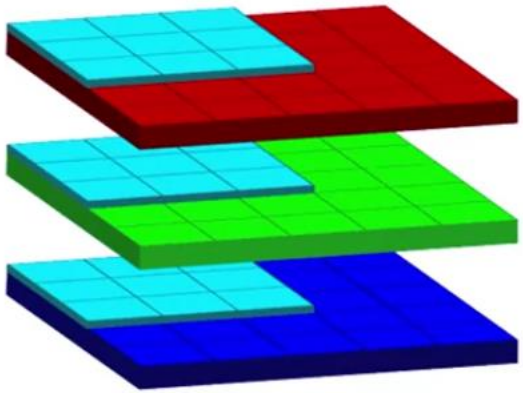
Color image

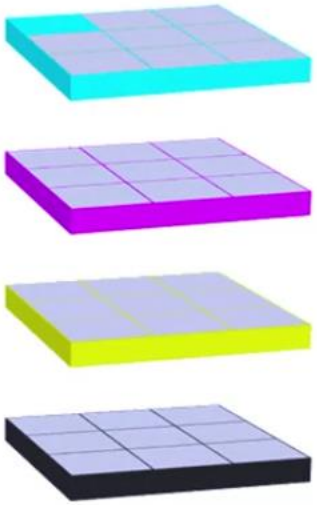




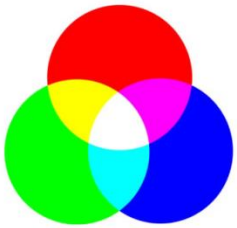
For coloured images
RGB|CYAN

Convolutions

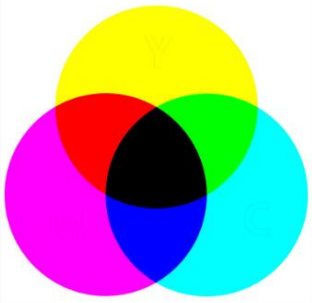




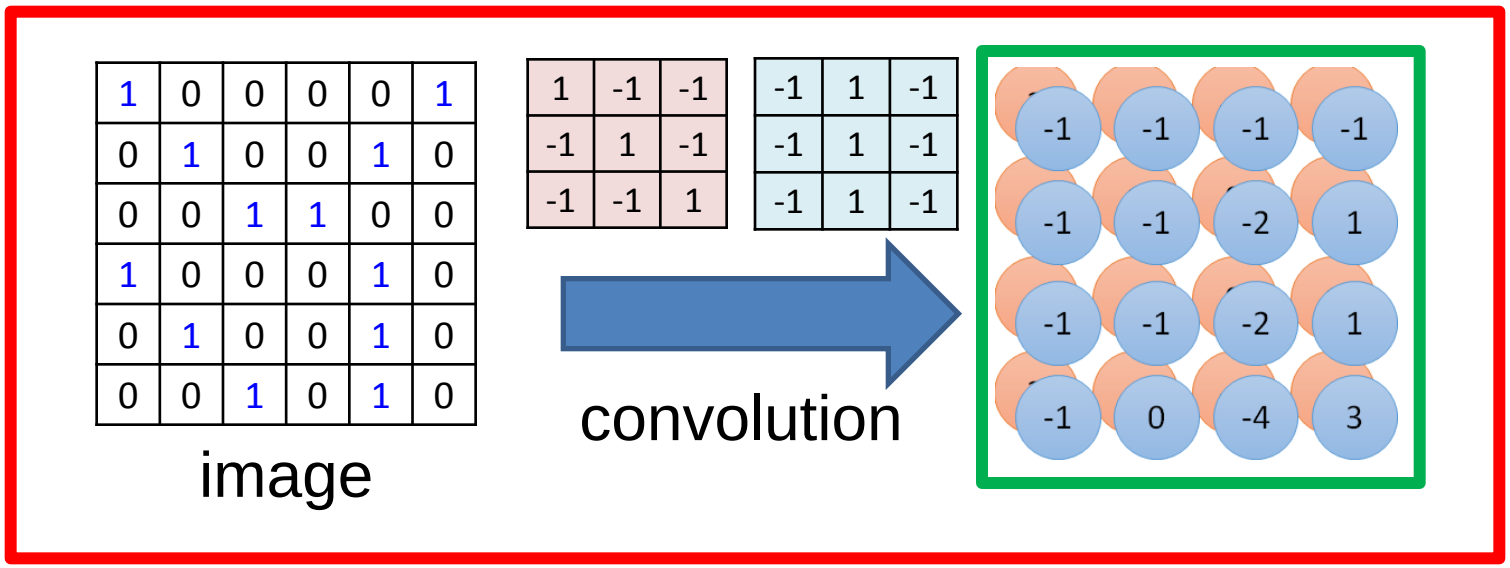
RGB - Red Green Blue



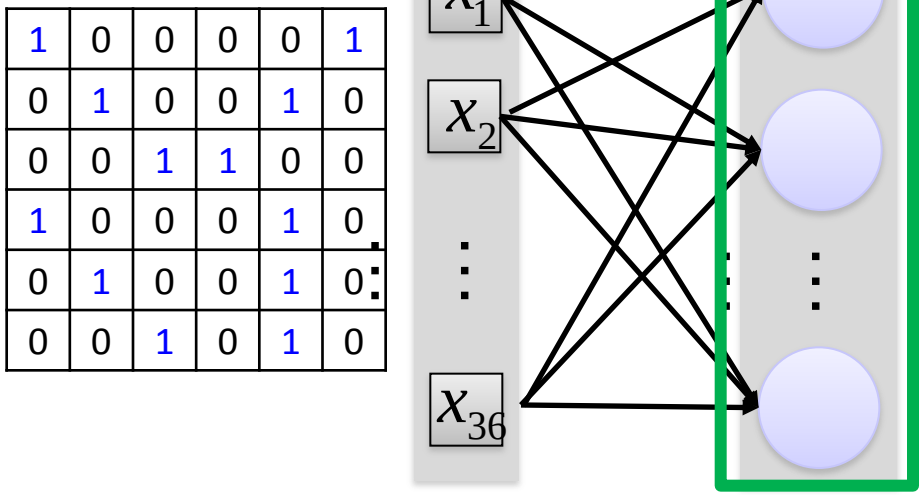
CMYK - Cyan Magenta Yellow Key(Black)

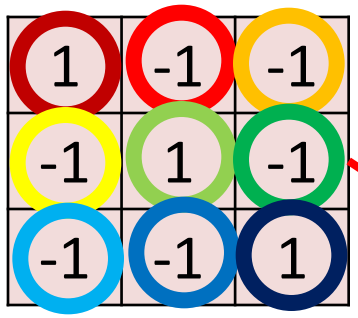


Convolution v.s. Fully Connected

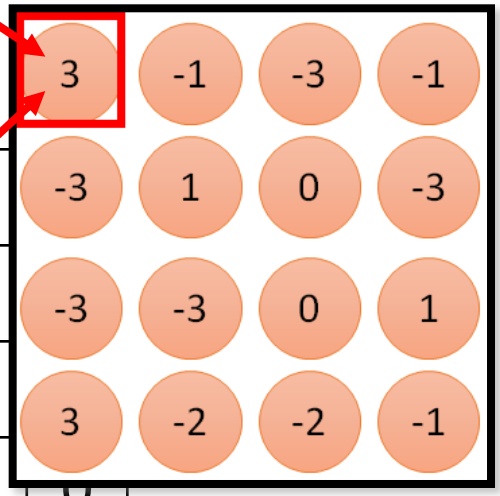


Fully-connected





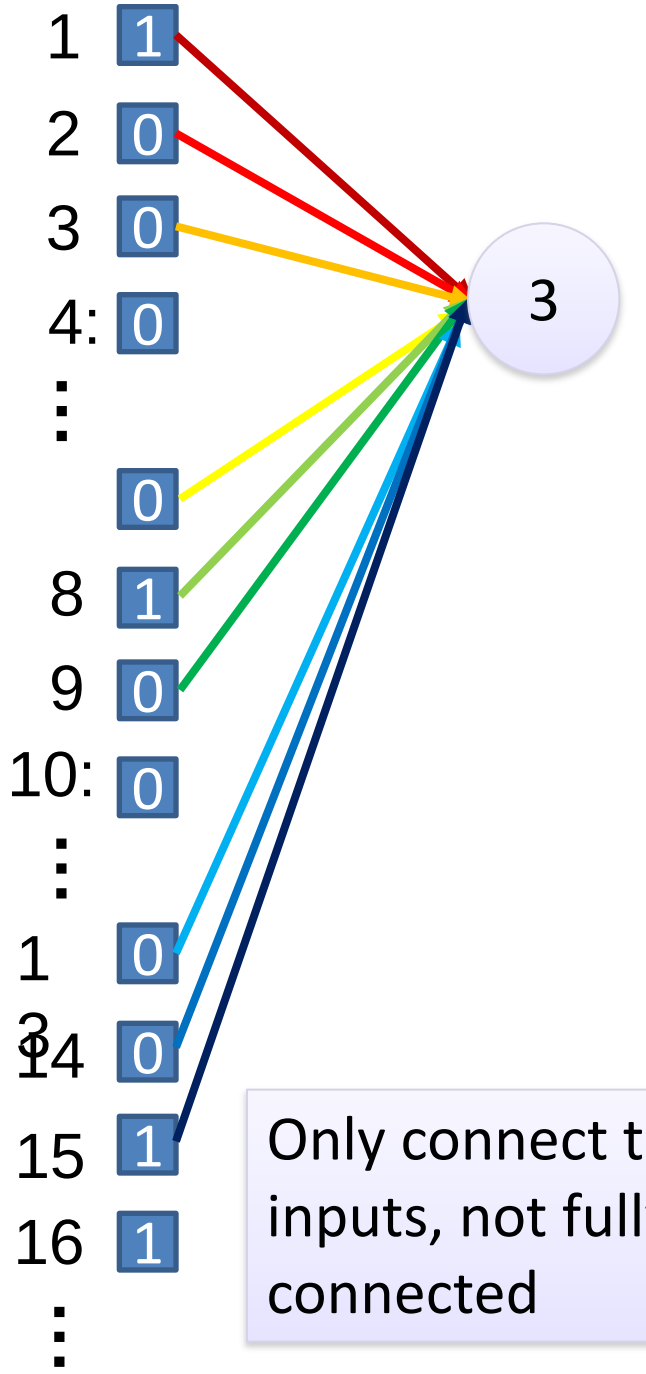
Filter 1



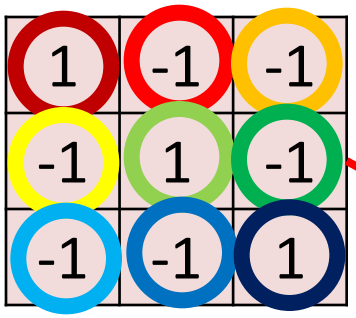
1	0	0	0	0	
0	1	0	0	1	
0	0	1	1	0	
1	0	0	0	1	
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

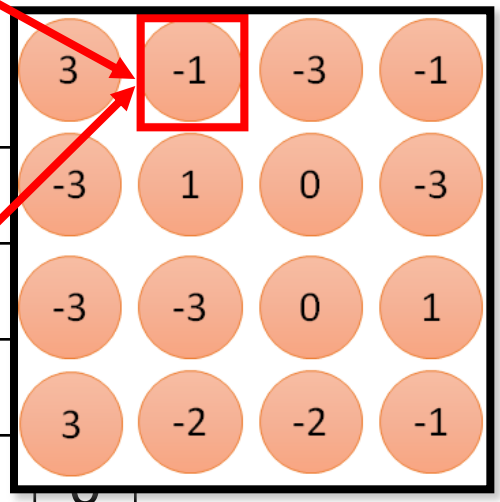
fewer parameters!



Only connect to 9 inputs, not fully connected



Filter 1

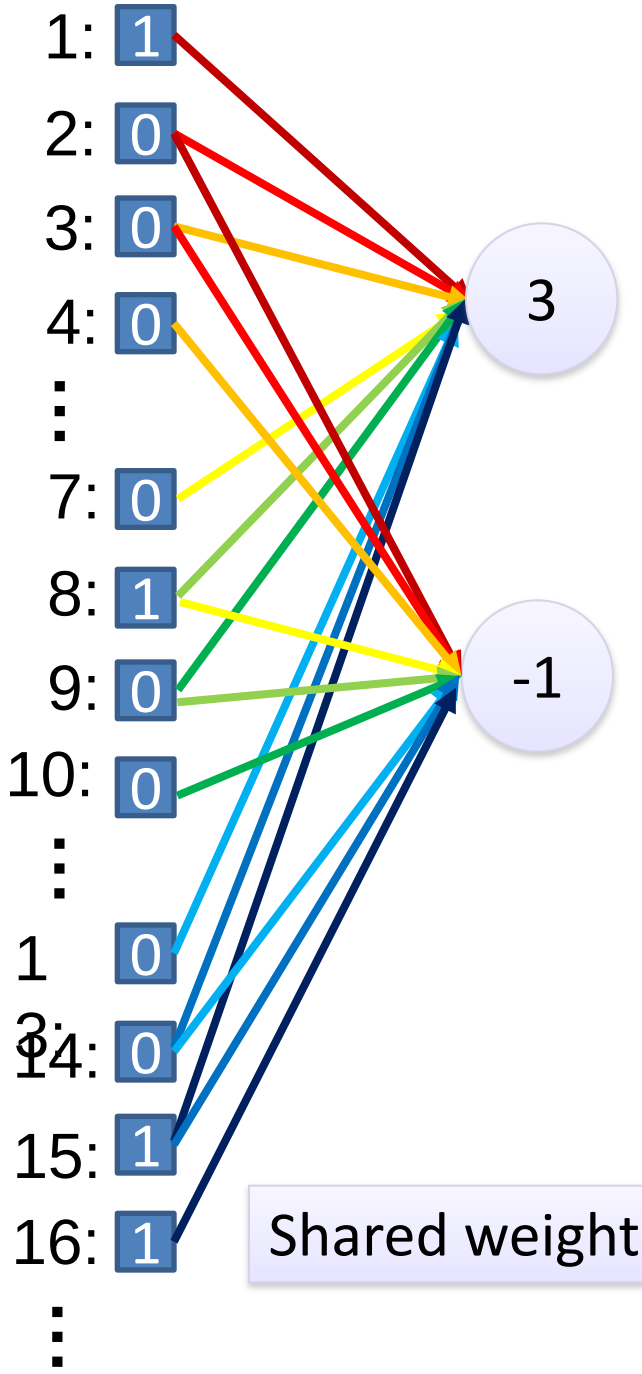


1	0	0	0	0	
0	1	0	0	1	
0	0	1	1	0	
1	0	0	0	1	
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Fewer parameters

Even fewer parameters



Shared weights

Max Pooling

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

3	-1
-3	1

-3	-1
0	-3

-3	-3
3	-2

0	1
-2	-1

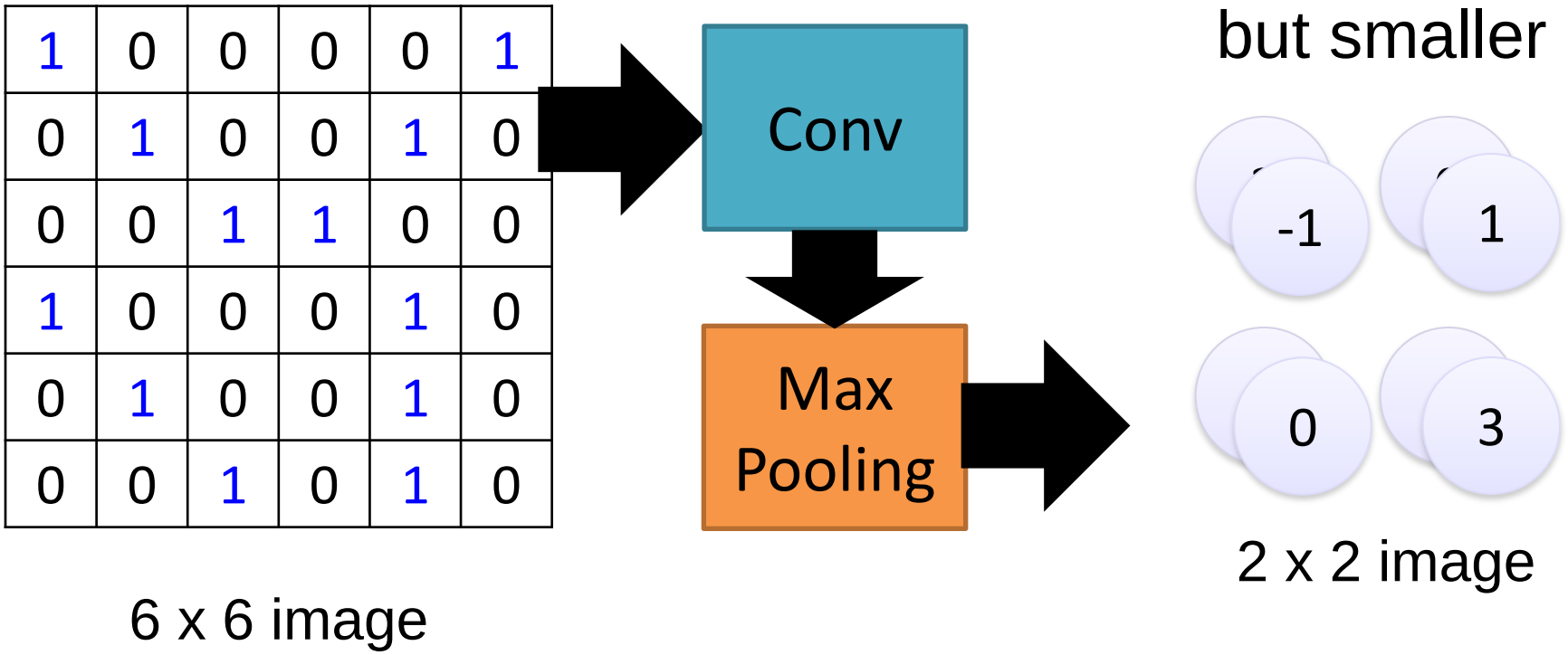
-1	-1
-1	-1

-1	-1
-2	1

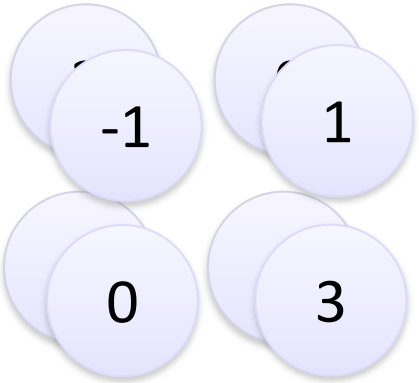
-1	-1
-1	0

-2	1
-4	3

Max Pooling


1 0 0 0 0 1
0 1 0 0 1 0
0 0 1 1 0 0
1 0 0 0 1 0
0 1 0 0 1 0
0 0 1 0 1 0
Conv
Max Pooling
New image but smaller
-1
1
0
3
6 x 6 image
2 x 2 image

Repeat



A new image

Convolution

Max Pooling

Convolution

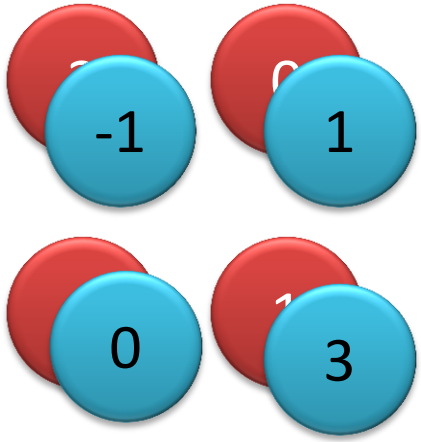
Max Pooling

Can repeat many times

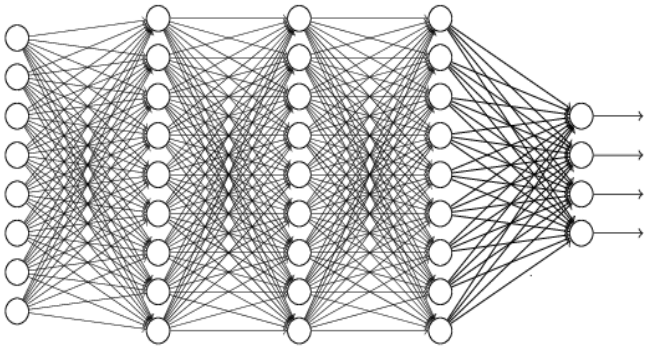
Smaller than the original image

The number of channels is the number of filters

Flattening



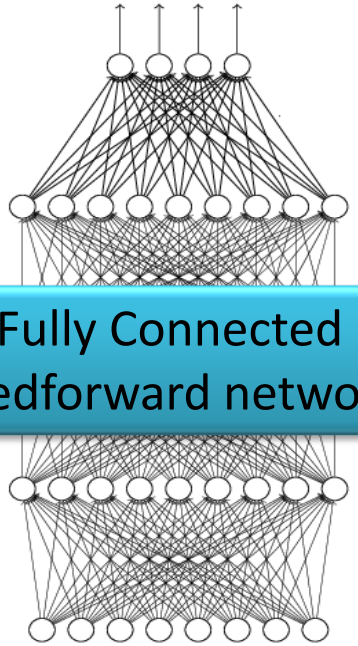
Flattened



Fully Connected
Feedforward network

The whole

cat dog ……



Fully Connected Feedforward network

Convolution

Max Pooling

Convolution

Max Pooling

Can repeat many times

Flattened

Convolutional Neural Networks

- **Convolutional Neural Networks** (**CNNs** or **ConvNets**) gained popularity in computer vision due to their extraordinary good performance on image classification tasks.
- As of today, CNNs are one of the most popular neural network architectures in deep learning.
- The key idea behind convolutional neural networks is to build many layers of **feature detectors** to take the spatial arrangement of pixels in an input image into account.
- Note that there exist many different variants of CNNs