Deep Learning

Wrap-Up Lecture

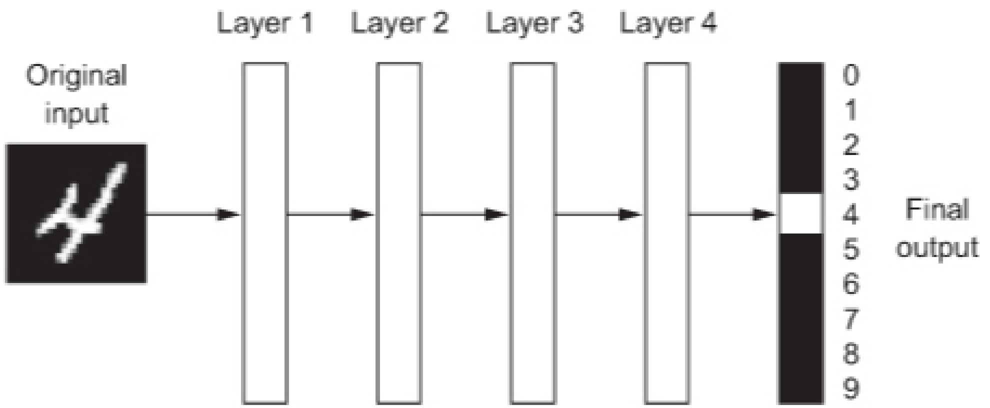
4th February 2020

Dr. Nicholas Cummins

- **Machine Learning**
  - **Discovering rules to execute a data-processing task**
  - A machine-learning system is trained rather than explicitly programmed.
    - It’s presented with many examples relevant to a task
    - It identifies statistical structure in these examples
    - These structure eventually allows the system to determine rules for automating the task
  - Unlike optimisation and conventional statistical analysis we want to learn rules that are generalisable to new data instances

What is Deep Learning?

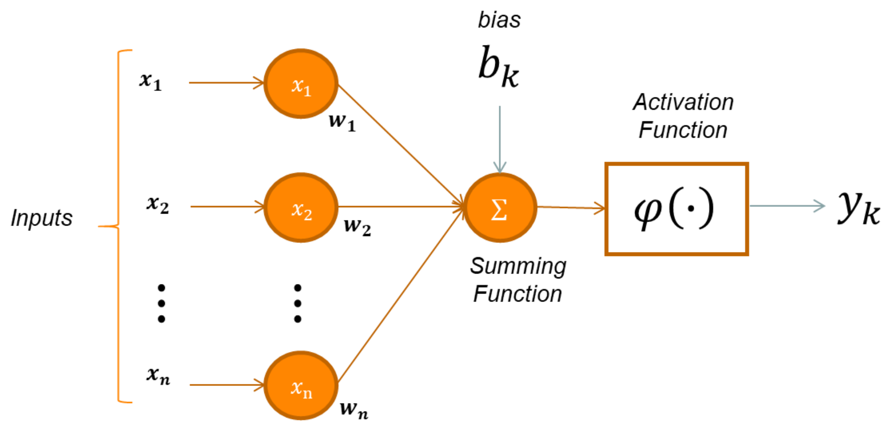
- **Deep learning is a specific subfield of machine learning**
  - Algorithms that put specific emphasis on learning successive layers of meaning full representations
  - The term deep represents this idea of successive layers of representations

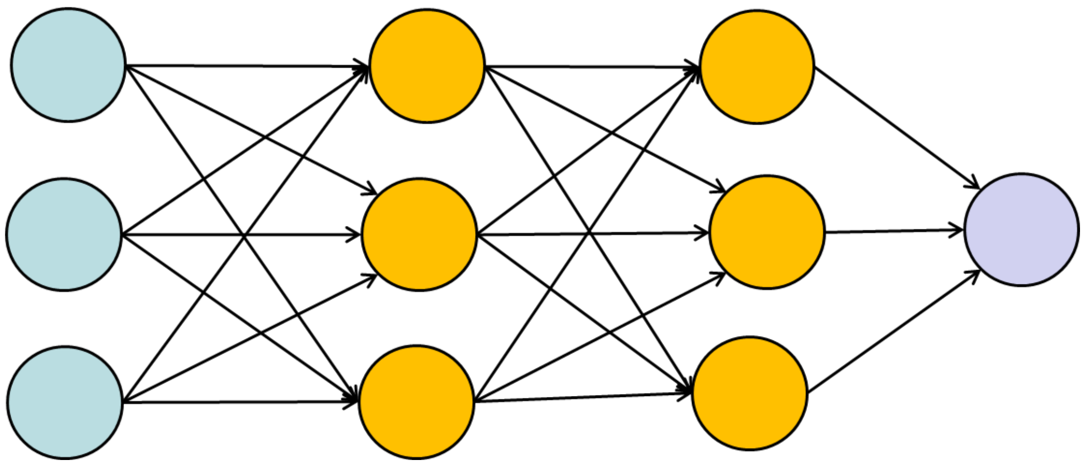


How many layers contribute to a model of the data is called the *depth* of the model

What is Deep Learning?

- **Neural networks**
  - In deep learning, the layered representations are (almost always) learned via models called neural networks structured in literal layers stacked on top of each other

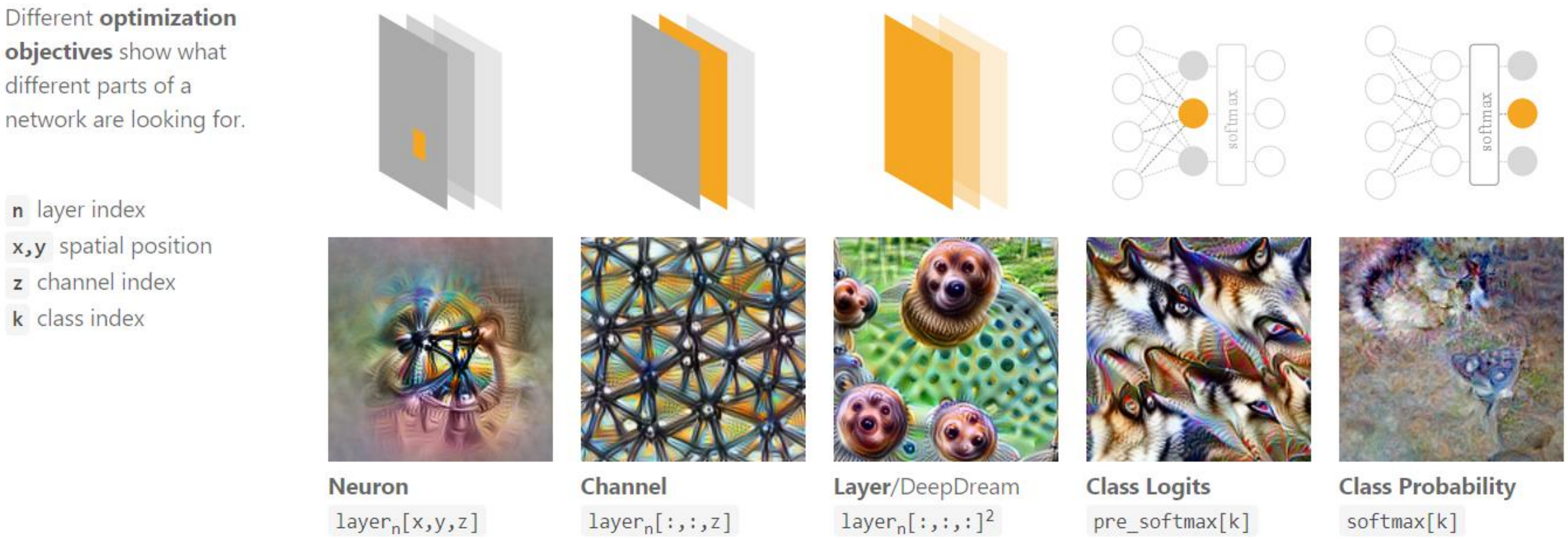




$$\varphi\left(\begin{pmatrix} w_1 & w_2 & \dots & b \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ 1 \end{pmatrix}\right) = \varphi(w_1 x_1 + w_2 x_2 + \dots + b\,) = y_k$$

Image Source:
https://distill.pub/2017/feature-visualization/

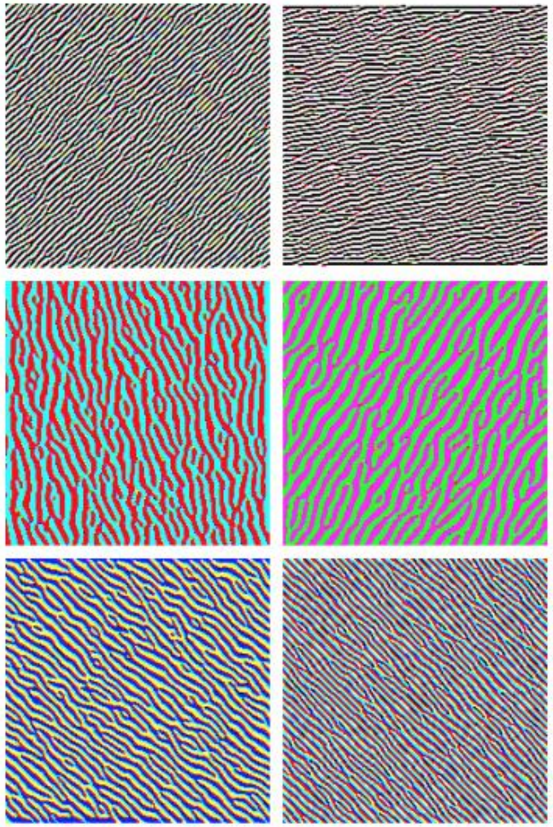
- **Deep learning is a set of multistage techniques for learning successive data representations**
  - A DNN transforms input data into a set of representations that are increasingly informative about the final result



Overview

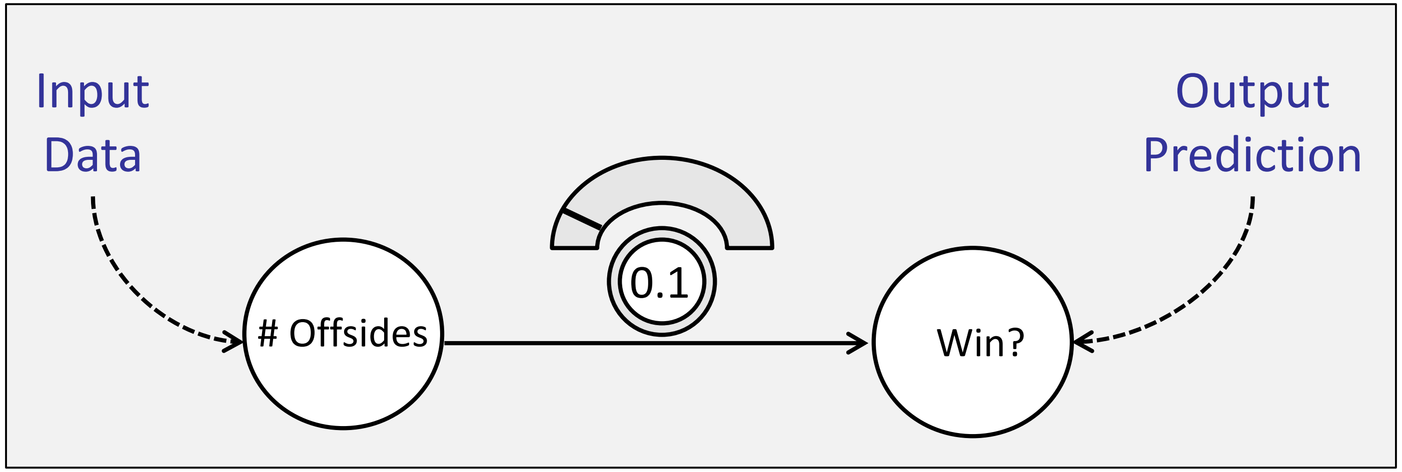
Image Source: https://distill.pub/2017/feature-visualization/

- Introduction
- **Feed Forward Networks**
- Convolutional Neural Networks
- Recurrent Neural Networks
- Sequence to Sequence
- Regularisation
- Explainable AI



Edges (layer conv2d0)

- **Simple predictions**
  - One input data point, one output prediction
  - Build a network with one single knob (*the weight*), to learn a mapping to one single output

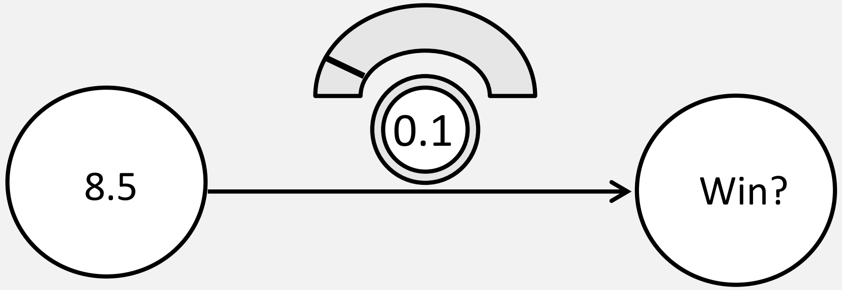


Prediction with a single network

- **What does a neural network do?**
  - It scales an input by a particular amount
  - It uses *knowledge* captured in the weights to *interpret* the input data to *predict* a certain outcome
    - This premise rings true, not may how complicated the network!

Multiplying input by weight

$$8.5 \times 0.1 = 0.85$$



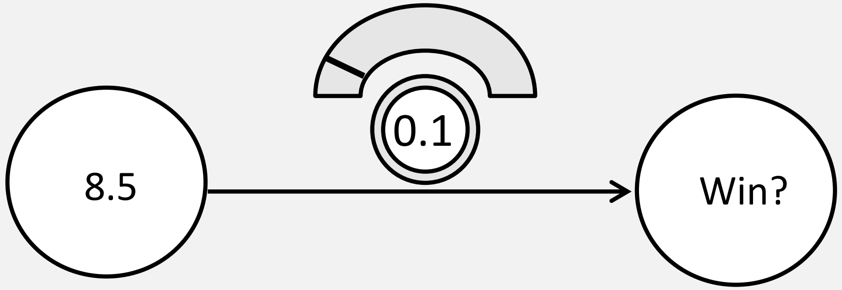

```python
weight = 0.1
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

Prediction with a single network

- **Weights represent knowledge**
  - It is a measure of sensitivity between the input data of the network and its prediction
    - If weight is high, tiny inputs can create large predictions
    - If weight is low, large inputs will make small predictions

Multiplying input by weight

$8.5 \times 0.1 = 0.85$




```python
weight = 0.1
def neural_network(input, weight):
    prediction = input * weight
    return prediction
```

- **Making accurate predictions:**
  - **Compare**
    - Evaluate how well the network performed

```
error = ((input * weight)- goal_pred) ** 2
```

  - **Learn**
    - Adjusting each weight to reduce the error
    - **Gradient Descent Algorithm**
      - Using the derivate of `weight` and `error` to adjust the weights

```
weight = weight - (alpha*derivative)
```

Prediction with a single network

- **Making accurate predictions:**
  - **Compare**
    - Evaluate how well the network performed

```
error = ((input * weight)- goal_pred) ** 2
```

  - **Learn**
    - Adjusting each weight to reduce the error
    - **Gradient Descent Algorithm**
      - Using the derivate of `weight` and `error` to adjust the weights

```
weight = weight - (alpha*derivative)
```

Prediction with a single network

- **Why measure error?**
  - Tuning weights to predict the target is actually a more complicated task than tuning weights to set error to zero
    - Therefore tune network such that `Error == 0`
- **Why squaring the error**
  - Help the network learn more effectively
    - Big errors become bigger
    - Small errors become smaller
  - We also want positive errors so they don’t cancel each other out when they are averaged

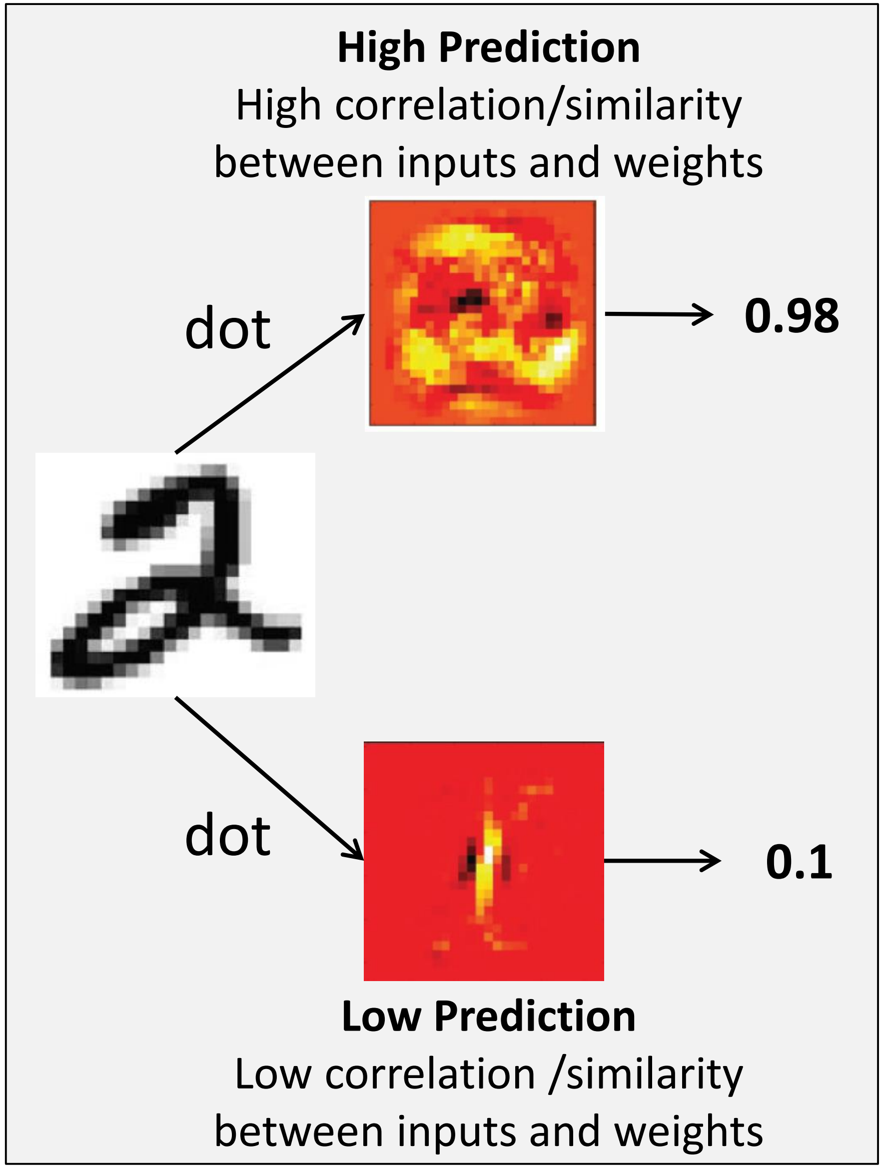
Prediction with a single network

- **Gradient Descent for neural learning**

```
pred = input * weight
error = (pred - goal_pred) ** 2
derivative = input * (pred - goal_pred)
weight = weight - (alpha * derivative)
```

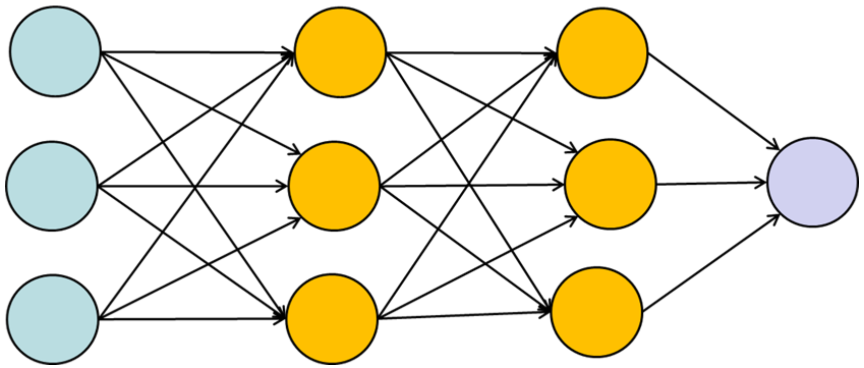
  - `error` is a measure of how much the network missed by
    - We define `error` to be always positive
  - `derivative` is the derivate of `weight` and `error`
    - Predicts both direction and amount to adjust the weights
  - `Alpha` scales the weight update
    - Helps minimise divergence effects when the input is large

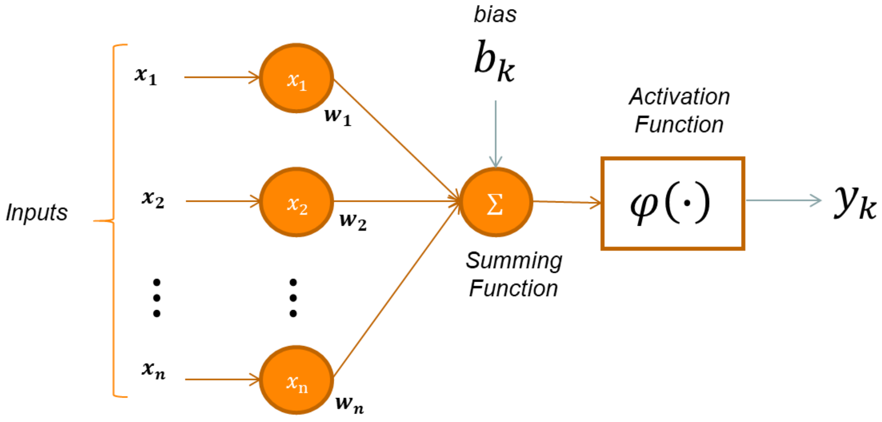
- **What are the weights learning?**
  - **Correlations between input and output**
    - If a weight is high, it means the model believes there's a high degree of correlation between that input and the prediction.
    - If the number is very low (negative), then the network believes there is a very low correlation (perhaps even negative correlation) between that input and the prediction
  - **Why is this?**
    - Weights are found via dot products



Role of Hidden Layers

- **Creating Correlation**
  - To learn when there is no correlation, just use more networks
  - Hidden layer(s) can be thought of as creating intermediate dataset(s) that has correlation with the output
  - Stacking linear neural networks does not give and more power
    - A more computationally expensive version of a single weighted sum
  - Use non-linear activation functions to induce correlation between layer



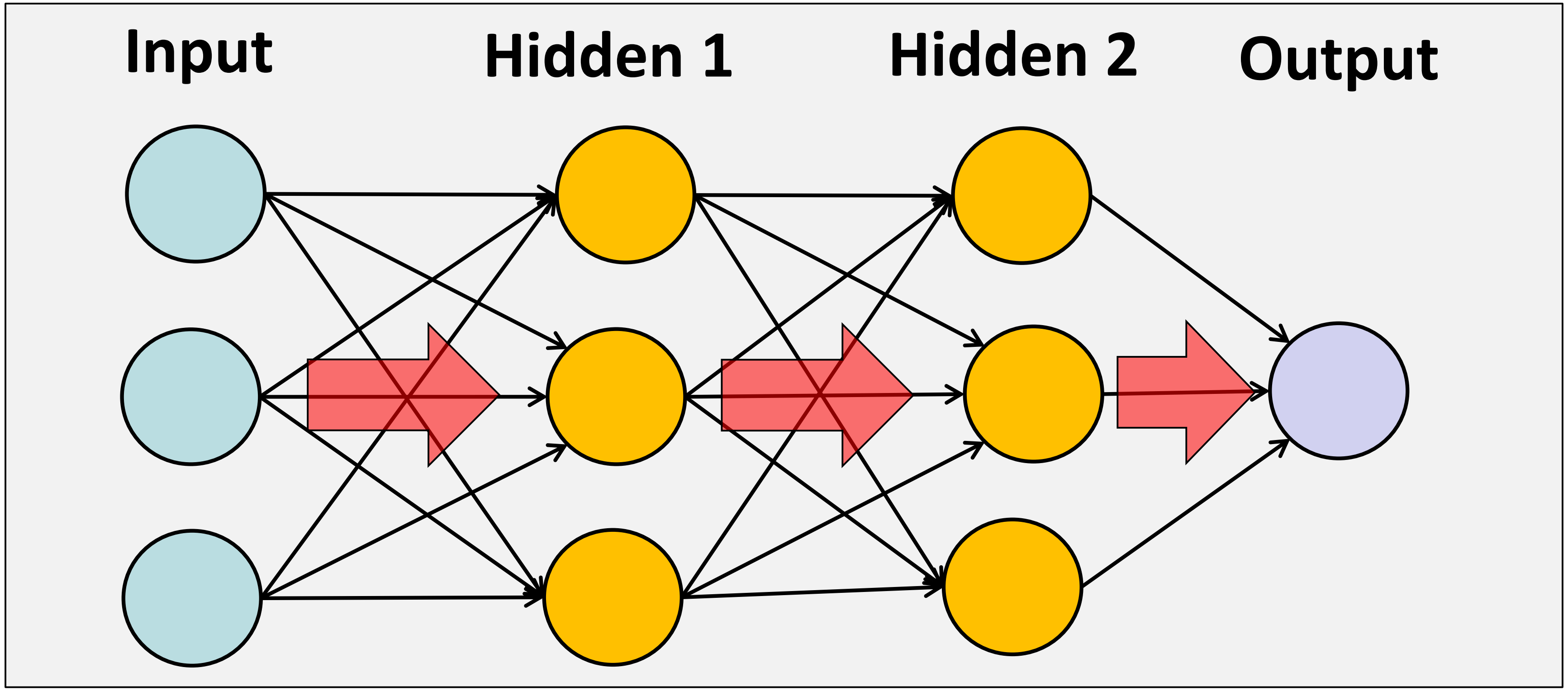


- **Role of Activation Functions**
  - **Good activation functions are nonlinear**
    - Allow for selective correlation: increase or decrease how correlated the neuron is to all the other incoming signals
  - **Other core properties**
    - The function must be continuous and infinite
    - The function should be monotonic
      - I.e., no two input values of have the same output value
    - The function and its derivative should be easily computable
      - Enable efficiency when training and deploying the network

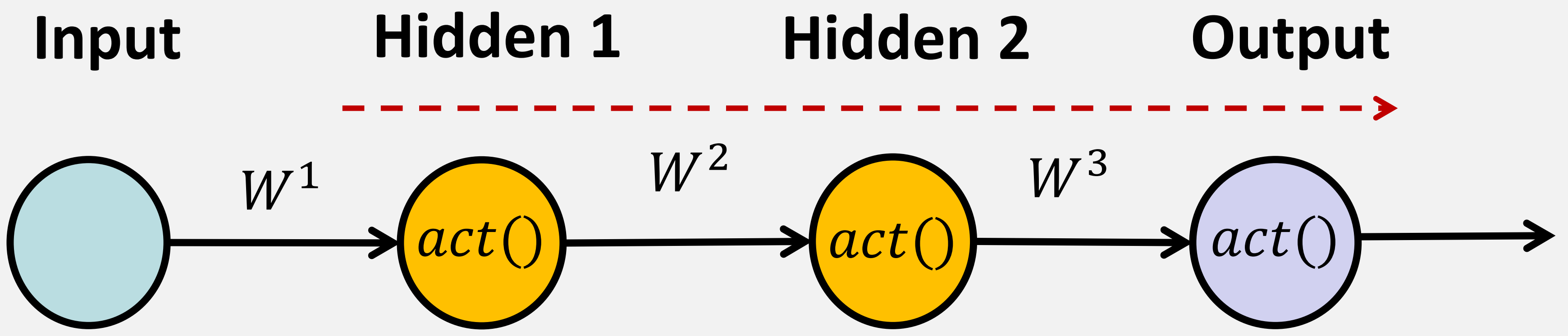
Activation Functions

- **Linear**
  - $f(x) = ax$
  - Range $-\infty$ to ∞
- **Sigmoid**
  - $\sigma(x) = \frac{1}{1+e^{x}}$
  - Range: 0 to 1
- **Hyperbolic Tangent**
  - $tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
  - Range: -1 to 1
- **Rectified Linear Unit**
  - $ReLU(x) = \max(0, x)$
  - Range 0 to ∞
- **Leaky Rectified Linear Unit**
  - $LReLU(x) = \begin{cases} x \; for \; x \geq 0 \\ 0.01x \; for \; x < 0 \end{cases}$
  - Range $-\infty$ to ∞
- **Softmax**
  - $softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(j)}$
  - Range: 0 to 1

- **Forward Propagation**
  - Information flows from input to output to make a predication



- **Forward Propagation**
  - Each neuron is a function of the previous one connected to it
    - Output is a composite function of the weights, inputs, and activations
      - Change any one of these and ultimately the output well change

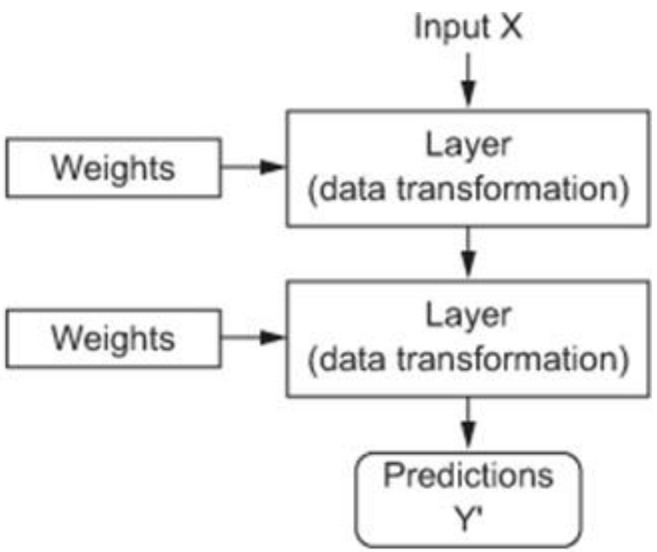


$$output = act\left(W^3 \times act\left(W^2 \times act(W^1 \times input)\right)\right)$$

- **Weights Learnt via Gradient Descent**
  - Update weights to minimise loss function
  - This is achieved by taking the gradient of loss function with respect to the weights

$$W += W + \alpha \frac{\partial j}{\partial w}$$

  - Not a trivial process as neural networks are structured as a series of layers
  - A single network can contain many millions of weights, and modifying the value of one weight will affect the behaviour of all the others

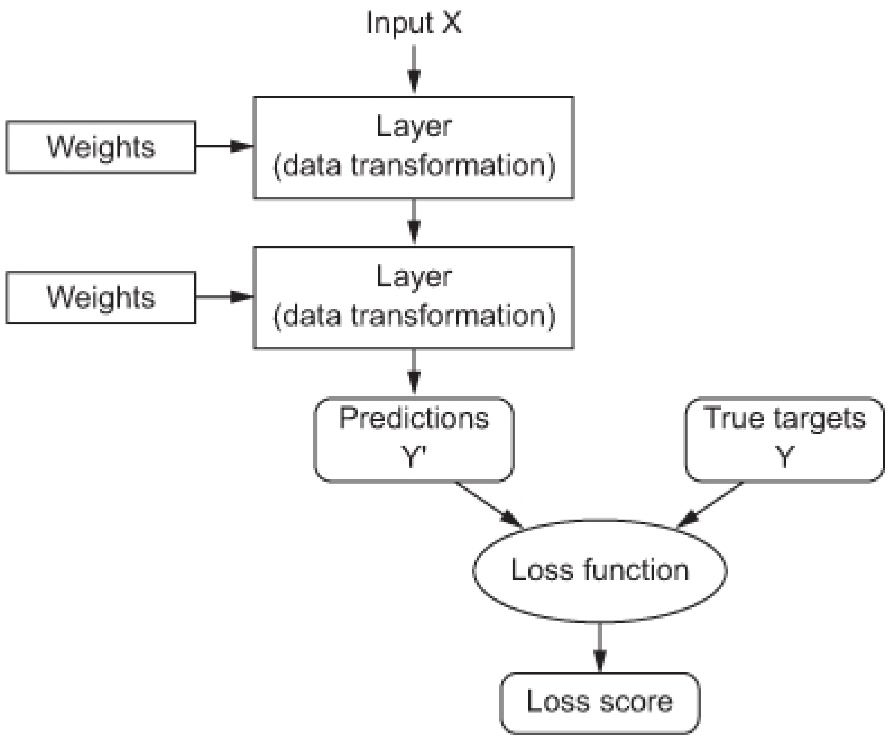


Training a Deep Neural Network

- **Learning in Deep Neural Networks**
  - To control weight updates in neural networks we use a loss function to how far an output prediction is from what we expected
    - The loss function computes a single scalar value relating to network performance
    - Measures the difference between what we have predicted, $\tilde{y}$, with the what it should predicted y.

$$\mathcal{L}(\tilde{y}, y)$$

    - We can then use this information to update the network



Loss Functions

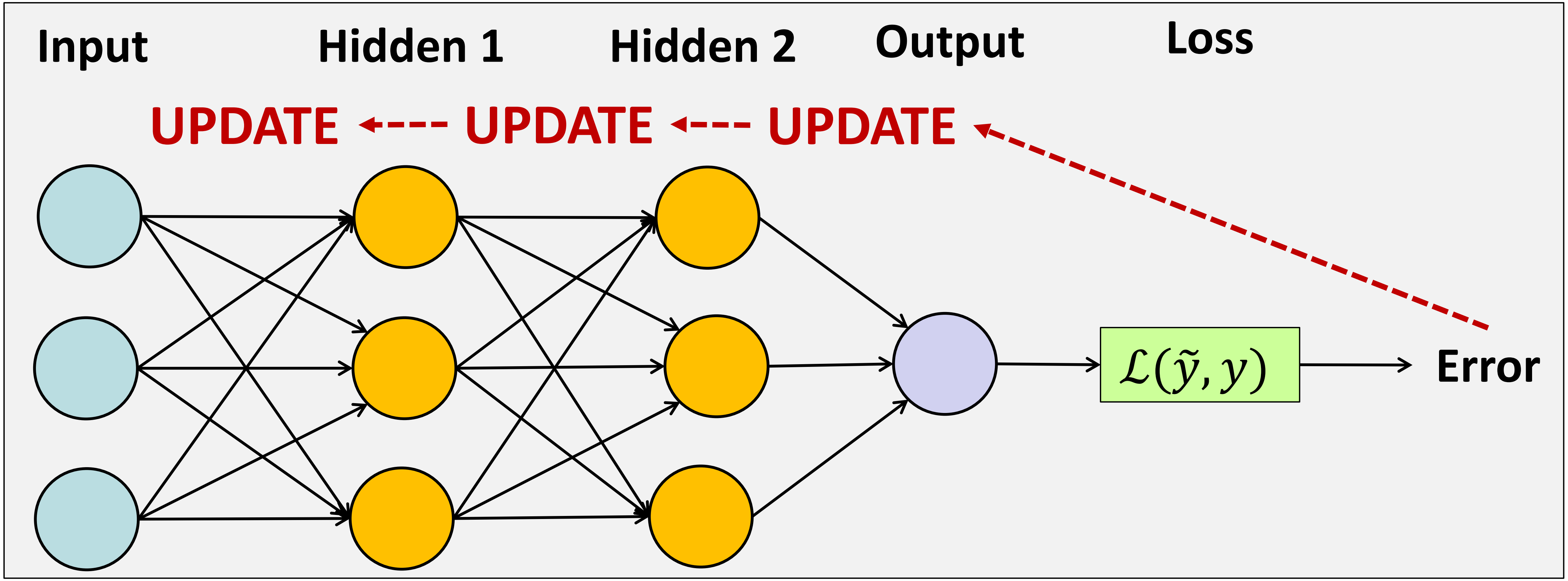
- **Regression**
  - Predicting a single numerical value
  - Final activation – **Linear**
  - Loss function – **Mean Squared Error**

- **Binary outcome**
  - Data is or isn't a class
  - Final Activation function – **Sigmoid**
  - Loss function – **Binary Cross Entropy**

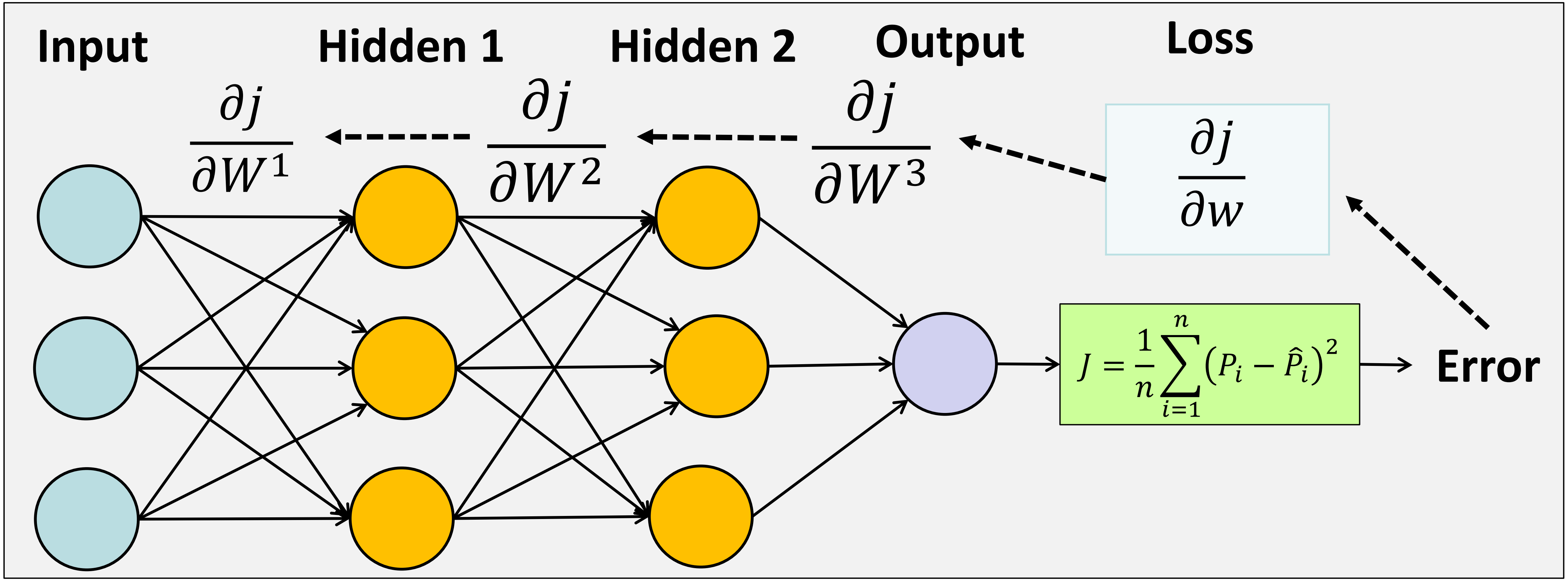
- **Single label from multiple classes**
  - Multiple classes which are exclusive
  - Final Activation function – **Softmax**
  - Loss function – **Cross Entropy**

- **Multiple labels from multiple classes**
  - If there are multiple labels in your data
  - Final Activation function – **Sigmoid**
  - Loss function – **Binary Cross Entropy**

- **Perform Gradient descent**
  – Output value effected by weights at all layers

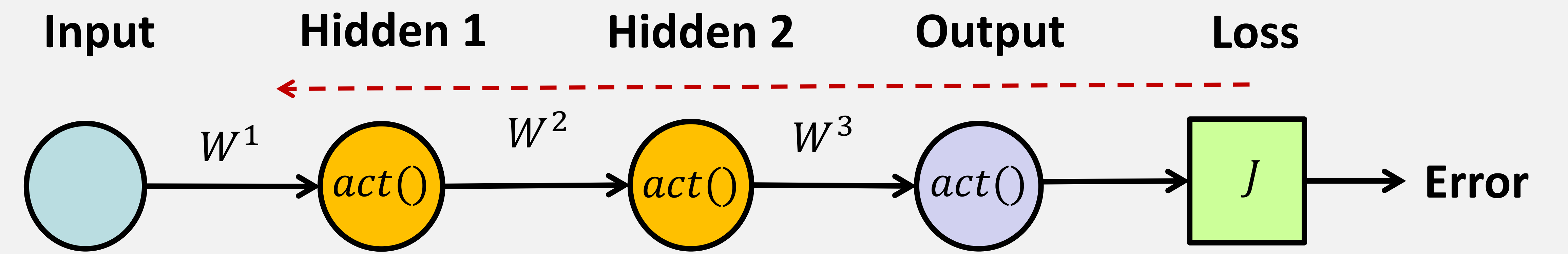


- **Backpropagation**
  - Tool to calculate the gradient of the loss function for each weight



Backpropagation

- **Calculating gradient for arbitrary weight**
  - Iteratively apply the chain rule
  - Note: Error is now a function of the output and hence a function of the input, weights, and activation functions



$$\frac{\partial error}{\partial W^1} = \frac{\partial error}{\partial output} \times \frac{\partial output}{\partial hidden2} \times \frac{\partial hidden2}{\partial hidden1} \times \frac{\partial hidden1}{\partial W^1}$$

- **Weights**

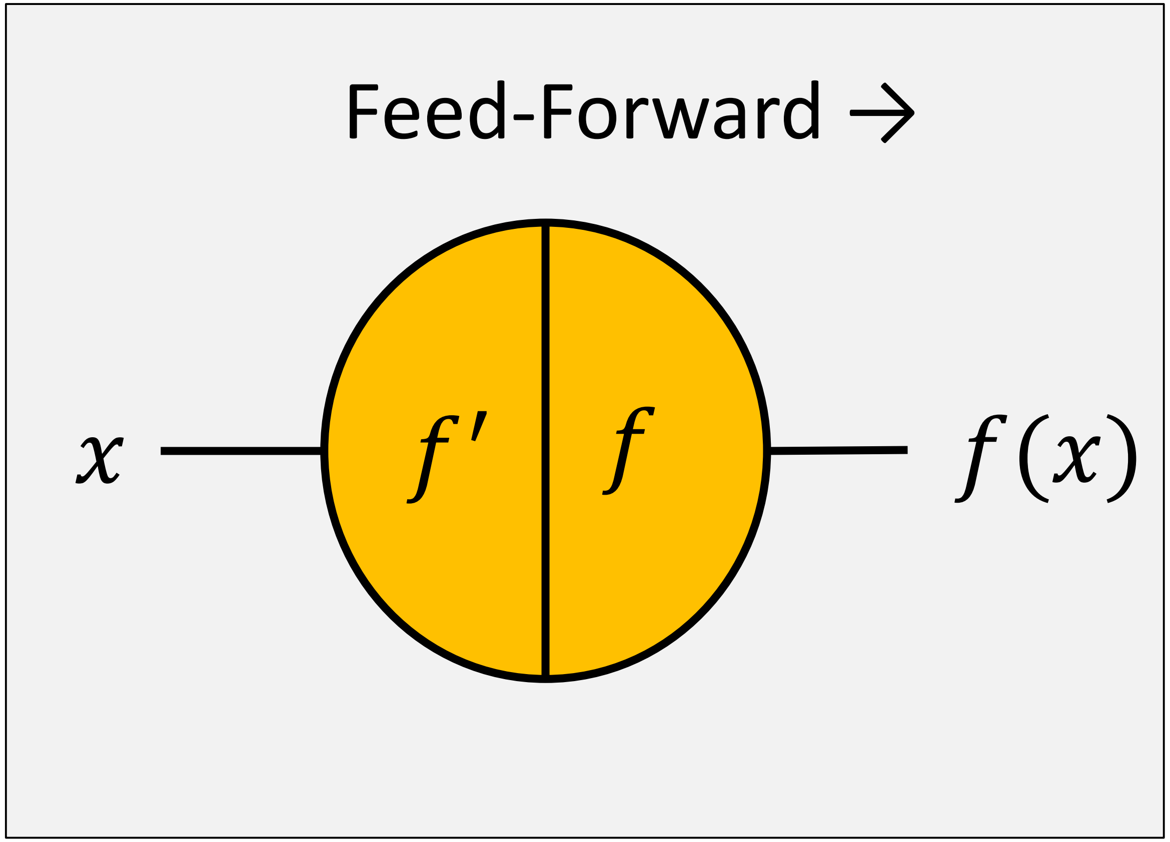
Feed-Forward →

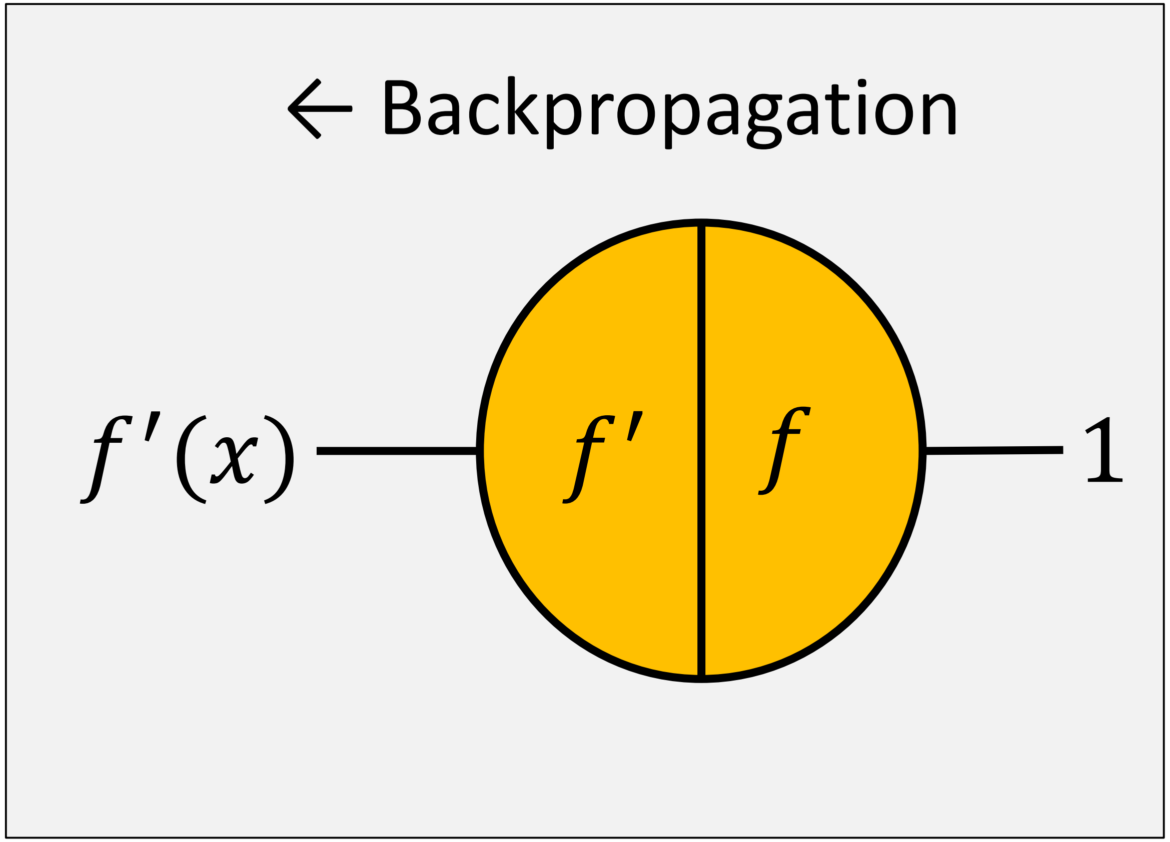
$$x \xrightarrow{\quad w \quad} wx$$

← Backpropagation

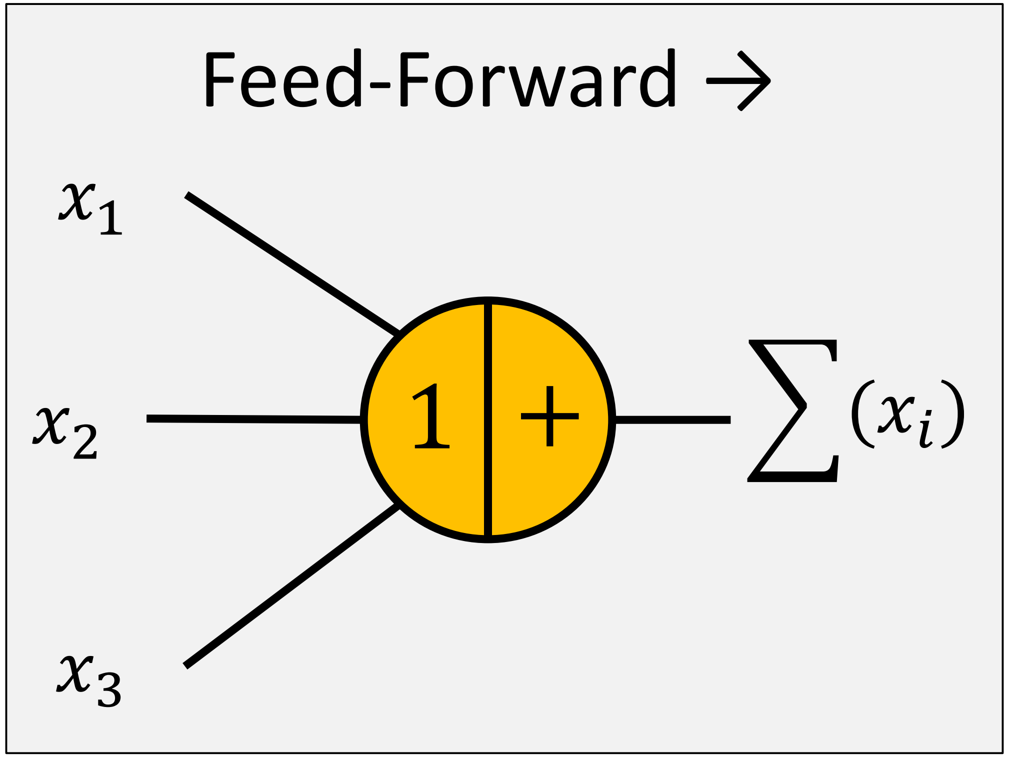
$$w \xleftarrow{\quad w \quad} 1$$

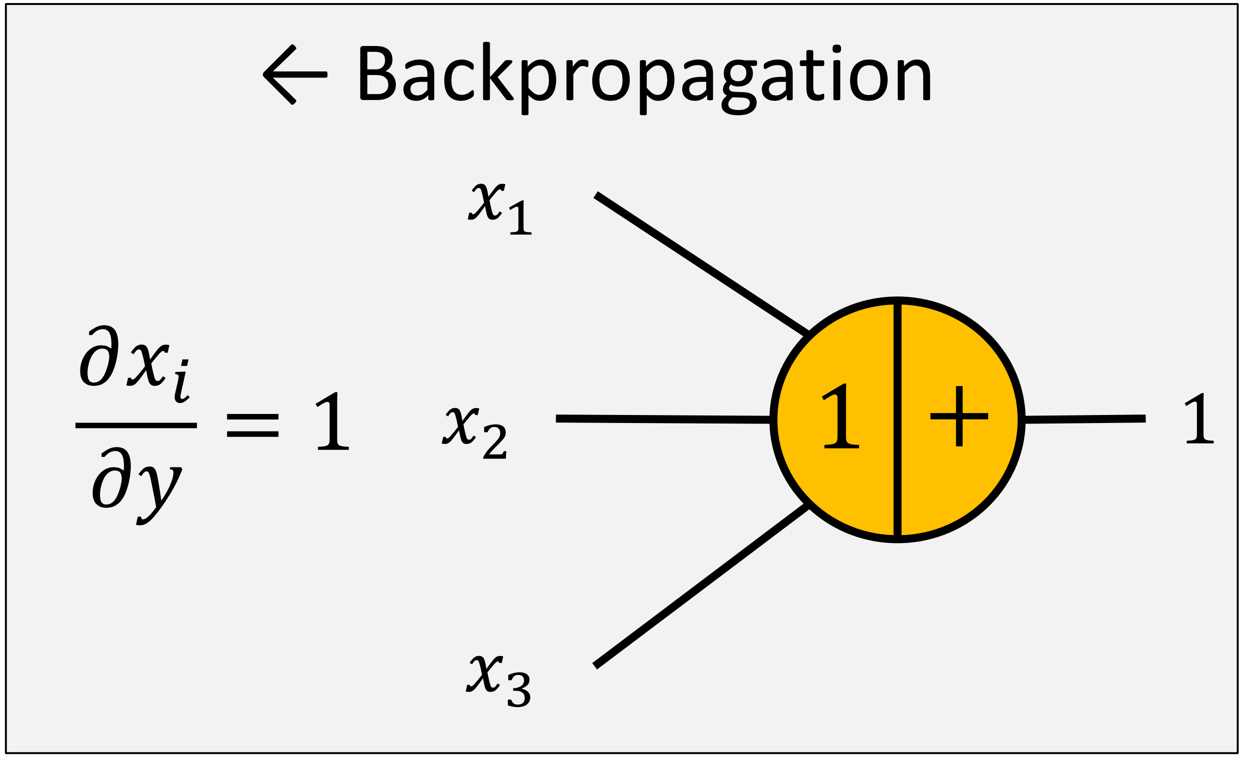
- **Activation function**





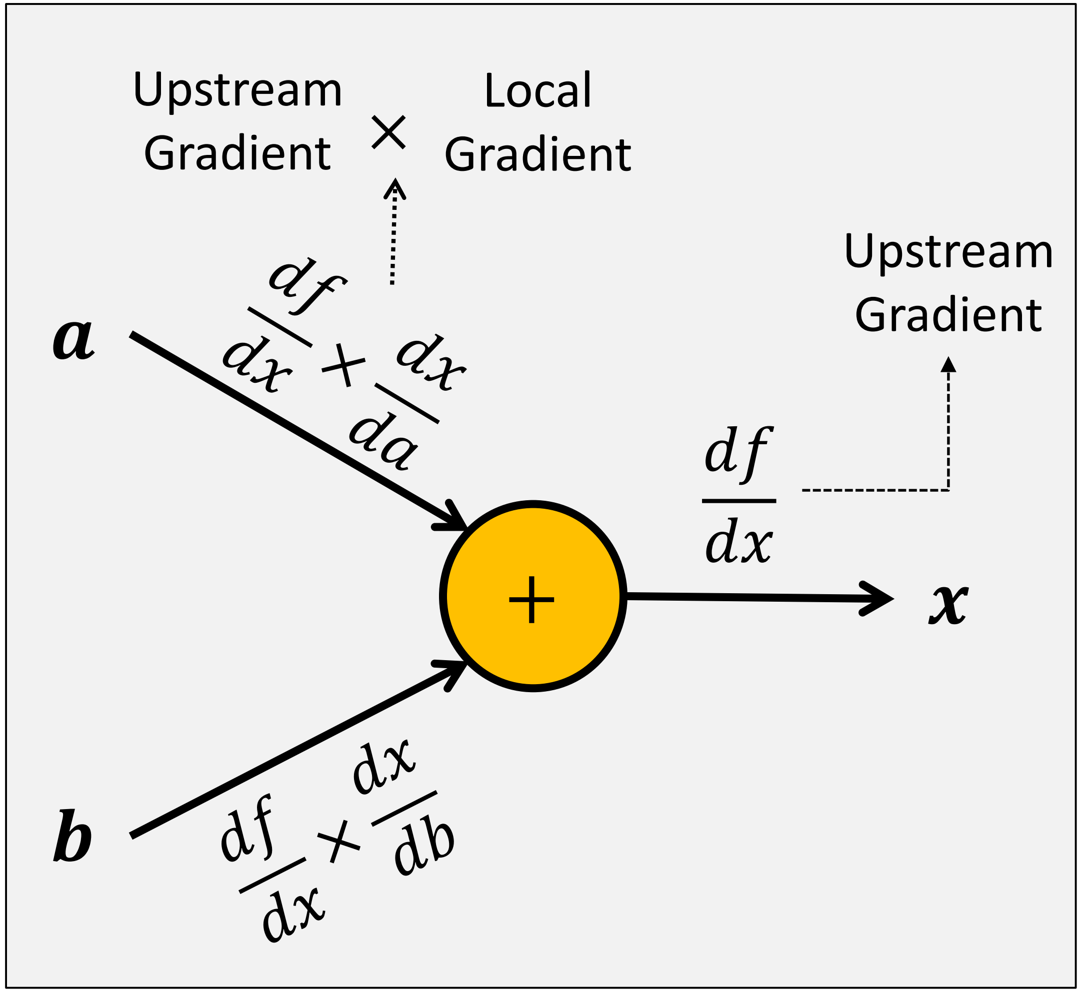
- **Summation Function**





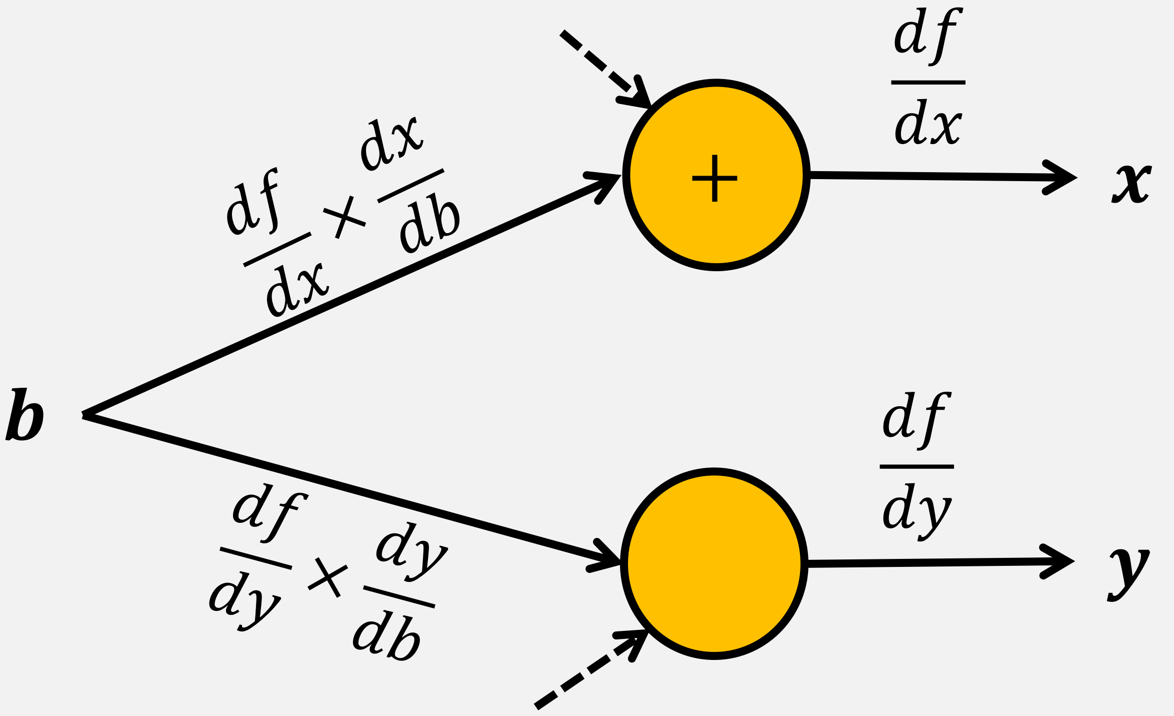
Key Backpropagation Rules

- **Calculating gradient via backpropagation**
  - Local gradients can be calculated before starting the backpropagation process
  - Iteratively apply the chain rule
    - The derivative of the output of the network with respect to a local variable is found by multiplying the local gradient with the upstream gradient

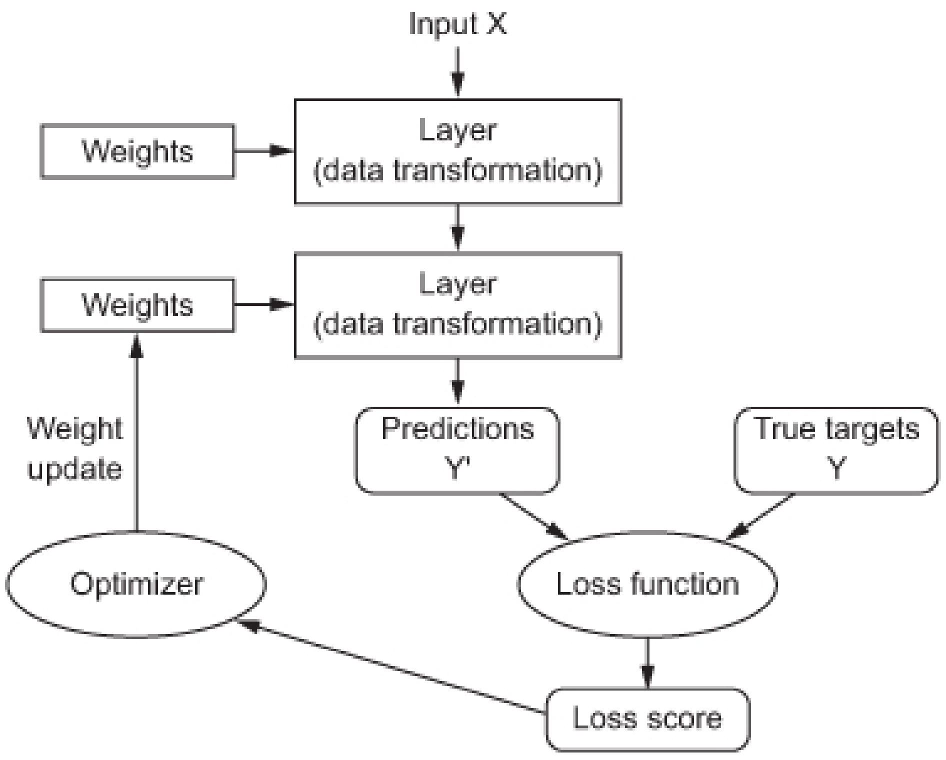


- **Calculating gradient via backpropagation**
  - Multivariate chain rule: Gradients add at branches

$$\frac{df}{db} = \frac{df}{dx} \times \frac{dx}{db} + \frac{df}{dy} \times \frac{dy}{db}$$

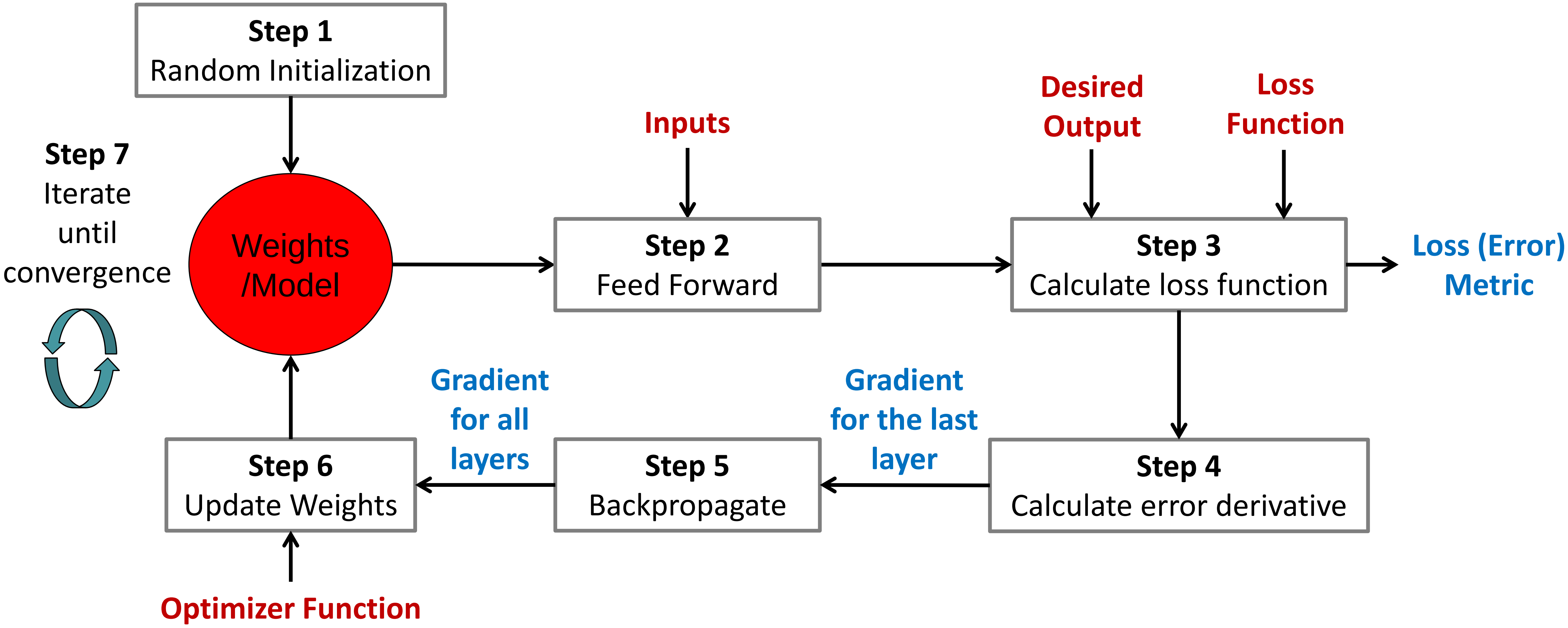


- **Learning in Deep Neural Networks**
  - The loss function provides a feedback signal to adjust the weights by a small amount, in a particular direction that will lower the score
  - This adjustment is performed by an *optimizer*, which implements the *Backpropagation* algorithm
    - Error attribution: figuring out how much each weight contributed to the final error by propagating the error back through the network



Comparison of different Optimisers

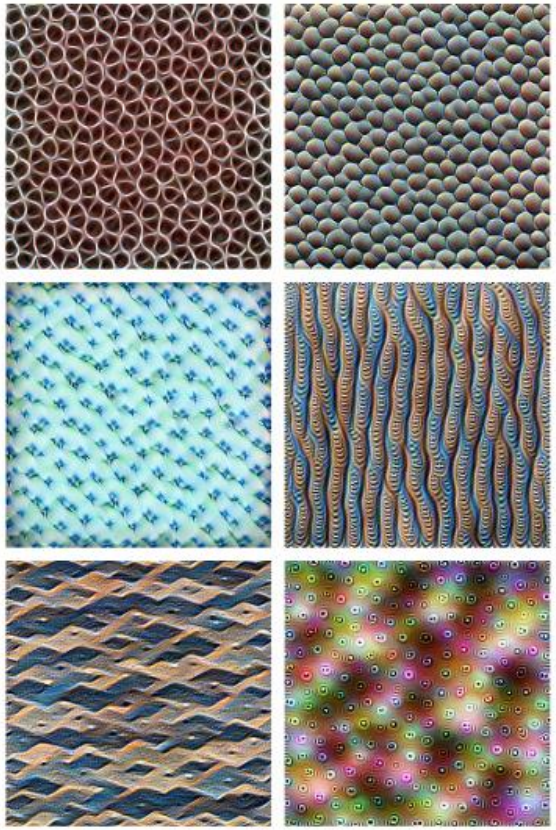
- Gradient Descent
  - Slow to converge, need to heuristically set learning rate
- Momentum
  - Uses gradient of previous time step to accelerate convergence
- Nesterov Accelerated Gradient (NAG)
  - Calculates gradient with respect to the future step
- Adagrad Adaptive Gradient Algorithm
  - An adaptive learning rate method
- Adadelta and RMSProp
  - Adaptive approach which restrict the window size of accumulated past gradients
- Adam Adaptive Moment Estimation
  - Calculates adaptive learning rate from first and second moments of the gradients


Step 1
Random Initialization
Step 7
Iterate
until
convergence
Weights
/Model
Inputs
Step 2
Feed Forward
Desired
Output
Loss
Function
Step 3
Calculate loss function
Loss (Error)
Metric
Gradient
for all
layers
Gradient
for the last
layer
Step 6
Update Weights
Step 5
Backpropagate
Step 4
Calculate error derivative
Optimizer Function

Overview

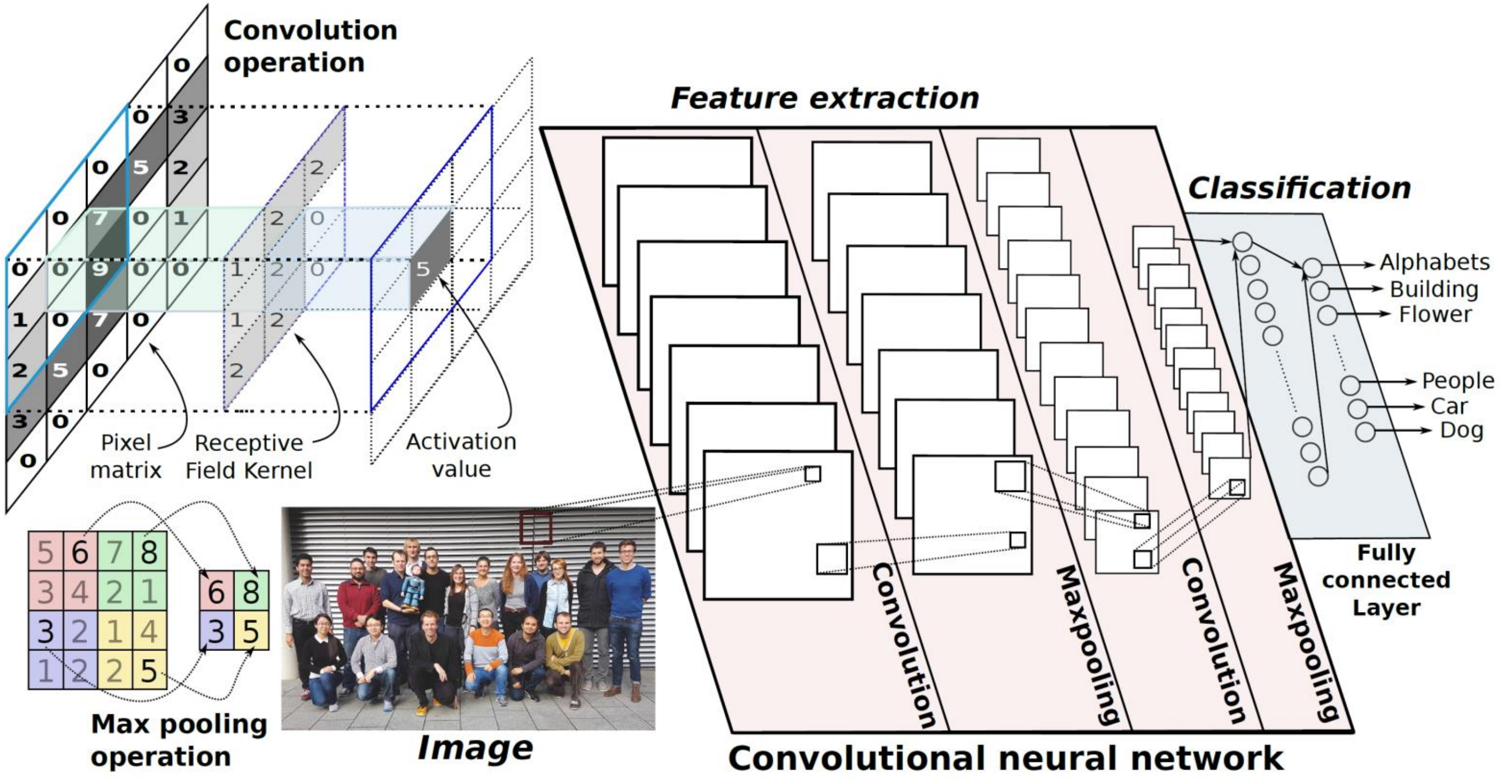
- Introduction
- Feed Forward Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Sequence to Sequence
- Regularisation
- Explainable AI

Image Source: https://distill.pub/2017/feature-visualization/



Textures (layer mixed3a)

- **Convolutional Neural Network (CNN)**
  - Convolutional kernels perform feature extraction



CNN – Motivation

Can repeat many times

Convolution

Max Pooling

Convolution

Max Pooling → Flatten → Fully Connected Feedforward Network → Prediction

Property 1: Some patterns are much smaller than the whole image

Property 2 : The same patterns appear in different regions

Property 3 : Downsampling the pixels does not change the object

Convolutional Layers

Apply small filers to detect small patterns

Each filter has a size of **3 x 3**

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	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

. . . . .

Note: Only the size of the filters is specified; the weights are initialised to arbitrary values before the start of training.

The weights of the filters are learned through the CNN training process

Convolutional Layers

stride = 1

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Filter 1

3 -3

Compute the dot product between the filter and a small 3 x 3 chunk of the image

Convolutional Layers

stride = 1

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Filter 1

3	-3	-3	3
1	-2	-2	1
1	-2	-2	1
-1	-1	-1	-1

4 x 4 image

stride = 1

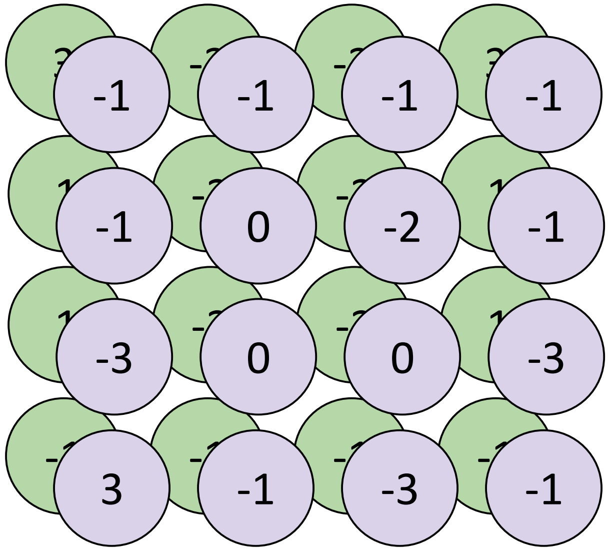
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

1	-1	-1
-1	**1**	-1
-1	-1	**1**

Filter 2

Feature Maps



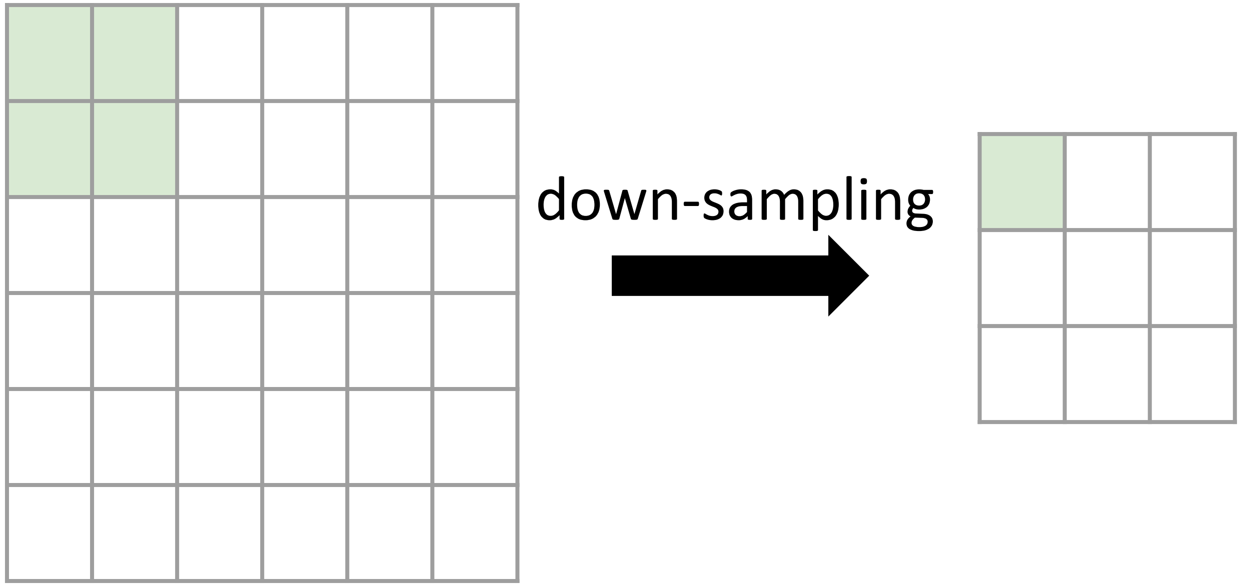
4 x 4 x (#filters)

Do the same process for every filter

Max Pooling

Pooling layers are usually present after a convolutional layer.

They provide a down-sampled version of the convolution output.



In this example, a 2x2 region is used as input of the pooling.

There are different types of pooling, the most used is max pooling.

Max Pooling

Filter 1

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Pooling size = 2 x 2

Stride = 2

Filter 2

1	-1	-1
-1	**1**	-1
-1	-1	**1**

3	-3	-3	3
1	-2	-2	1
1	-2	-2	1
-1	-1	-1	-1

Feature map 1

-1	-1	-1	-1
-1	0	-2	-1
-3	0	0	-3
3	-1	-3	-1

Feature map 2

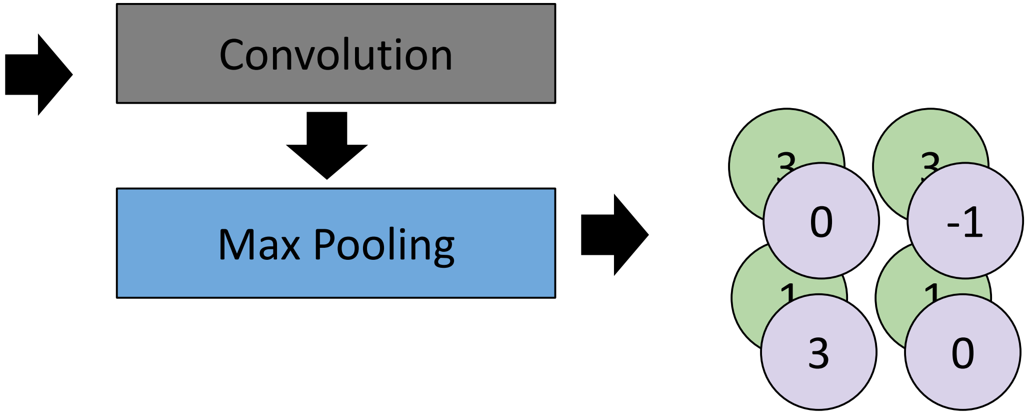
Operates over each feature map independently

Invariant to small differences in the input

Max Pooling

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image



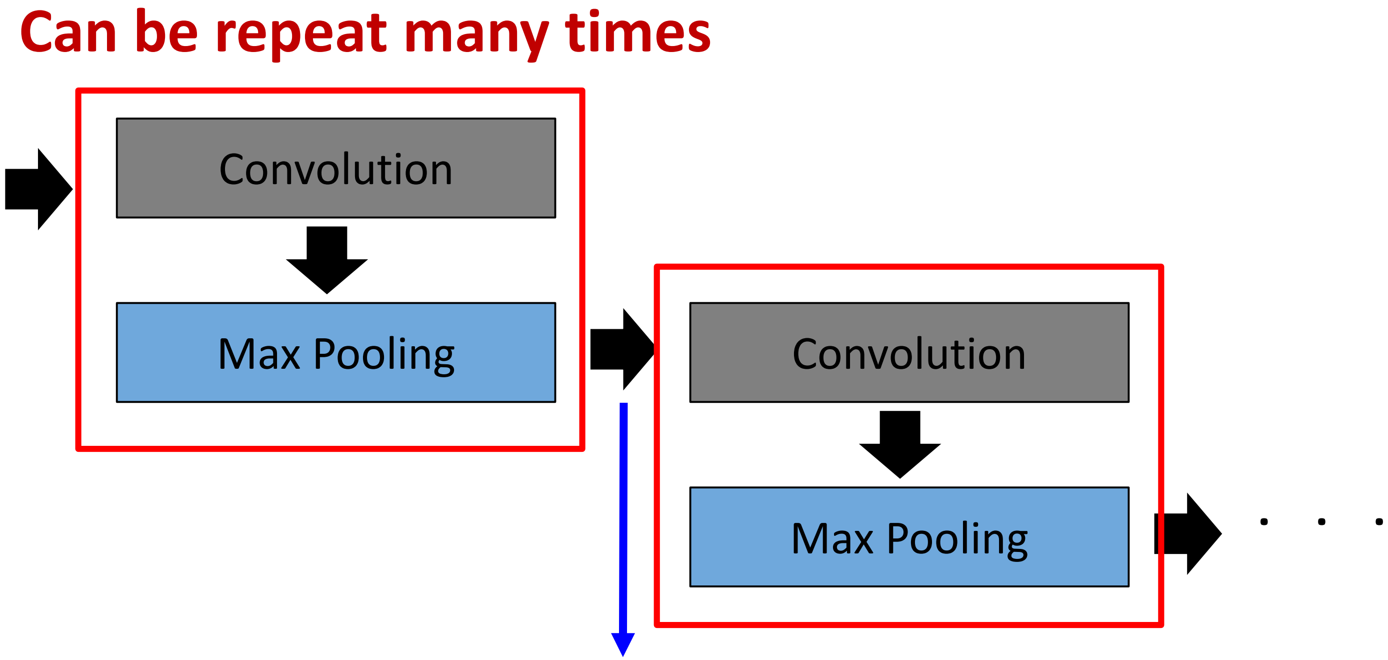
each filter is a channel

2 feature maps

each of size 2 x 2

Smaller and more manageable

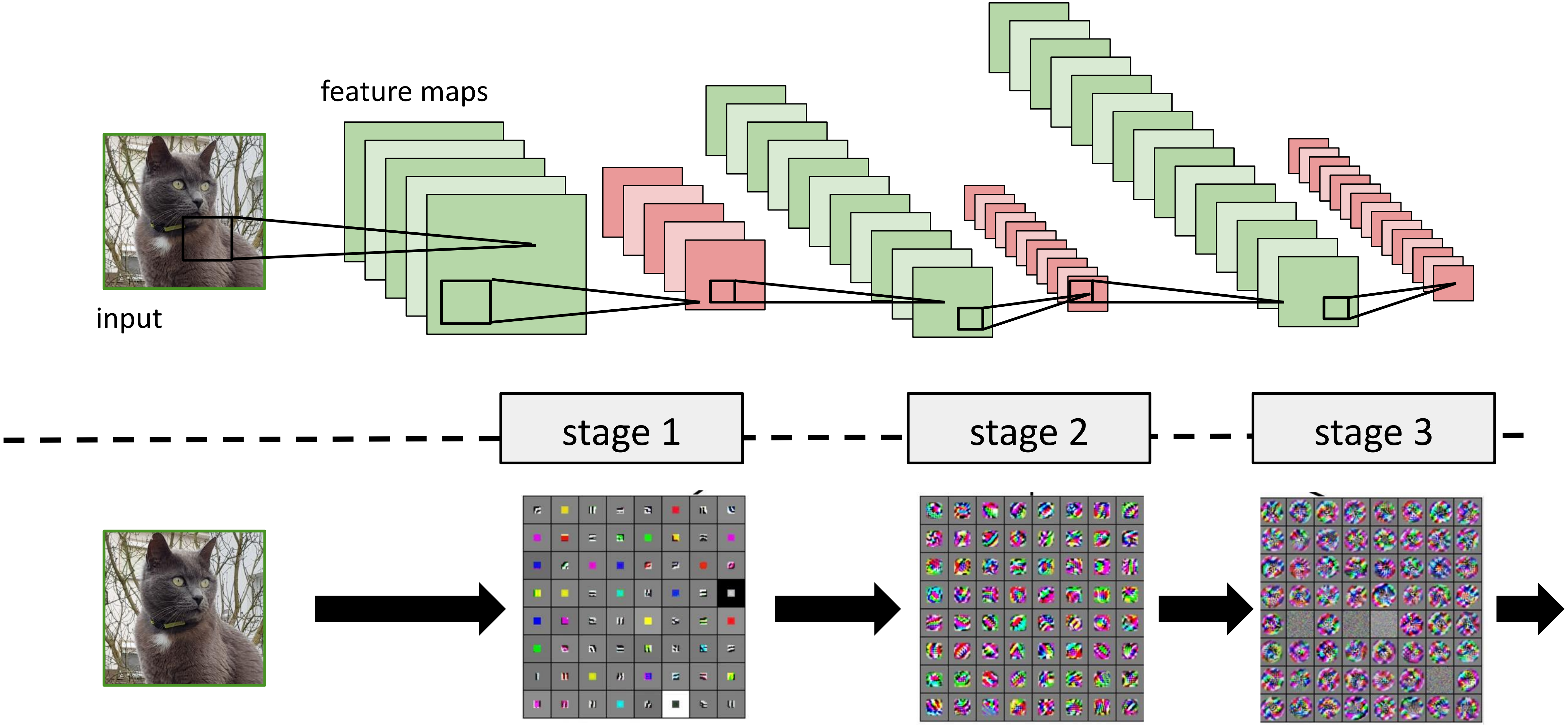
Convolve, Pool, Repeat



Output can be regarded as new images:

- Smaller than the original images
- The depth of new images is the number of filters

Representation Learning with CNNs



Overview

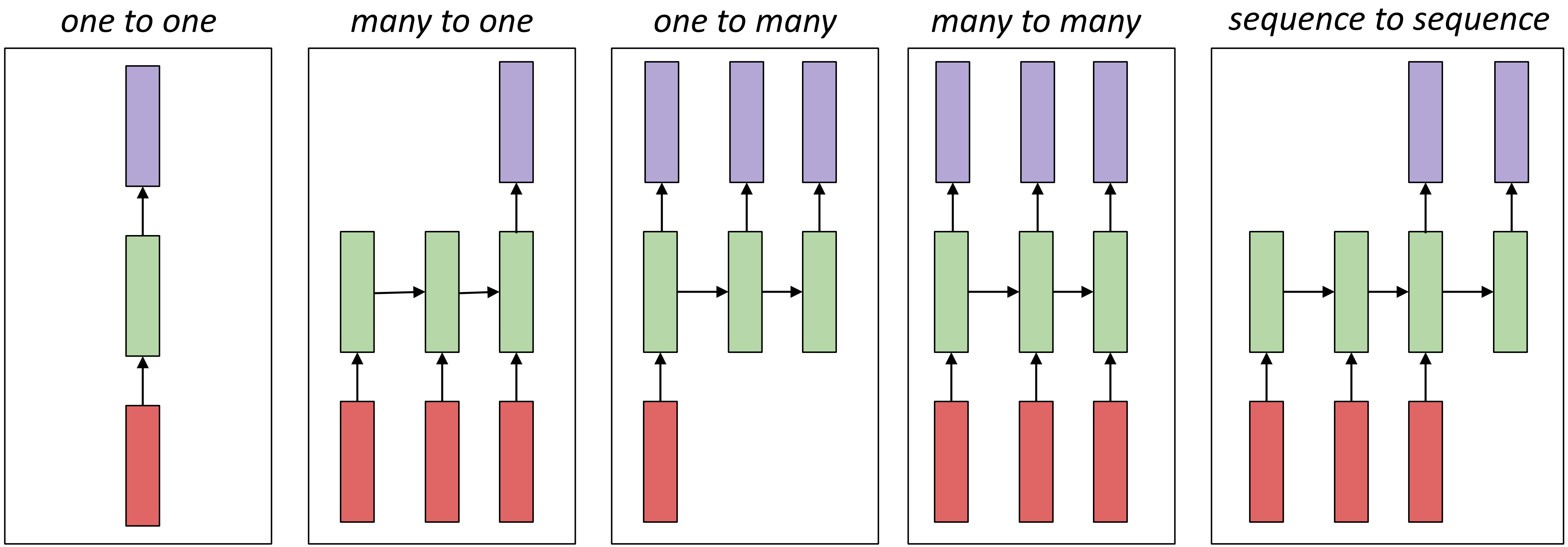
- Introduction
- Feed Forward Networks
- Convolutional Neural Networks
- **Recurrent Neural Networks**
- Sequence to Sequence
- Regularisation
- Explainable AI

Image Source: https://distill.pub/2017/feature-visualization/

Patterns (layer mixed4a)

- **Processing sequential inputs/outputs**
  - Requires information/knowledge from **previous inputs**

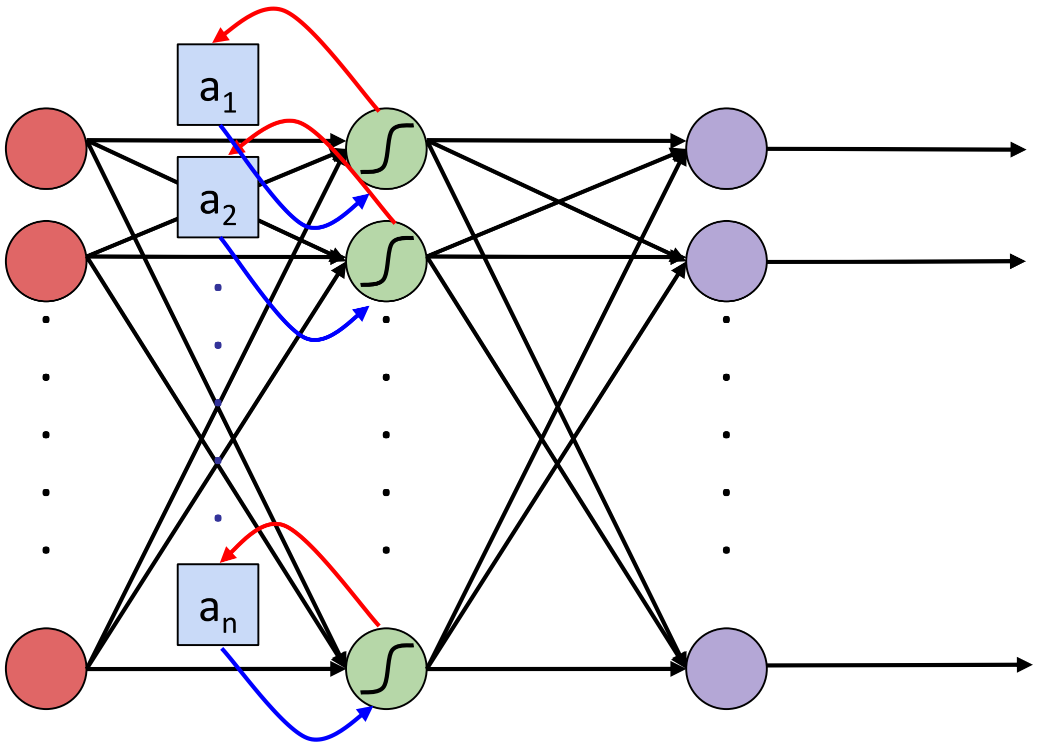
Neural Network needs **memory**!



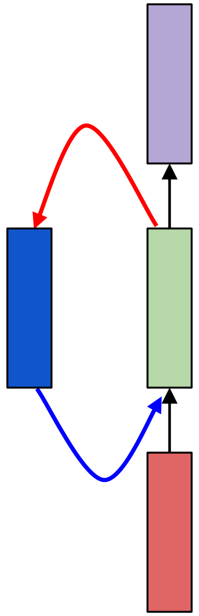
Inclusion of feedback into the network structure

Output of hidden layer are **stored** in the memory

Values in the memory are considered as **additional input** in the next time step



Inclusion of feedback into the network structure



memory to store the hidden states

Process a sequence of vectors $\boldsymbol{x}$ by applying a *recurrence formula* at every time step:

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

current state

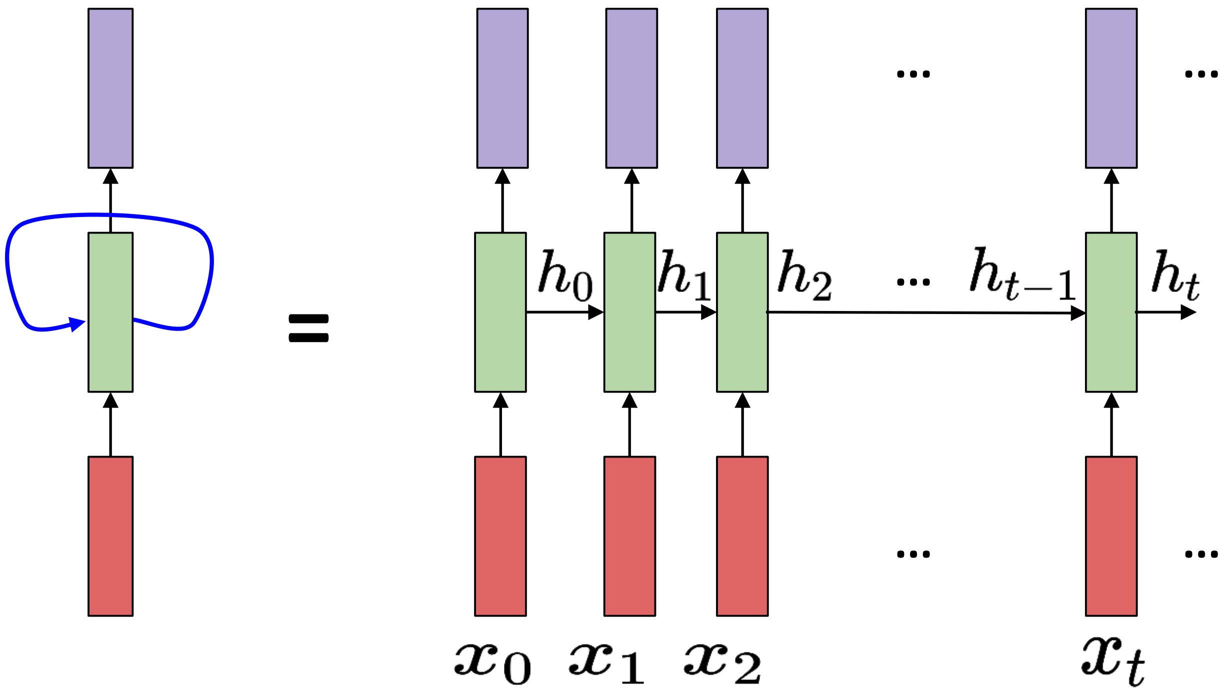
function with parameters W

state in memory (previous state)

current input vector

The same function and the same parameters are used at every time step.

Unrolled RNN



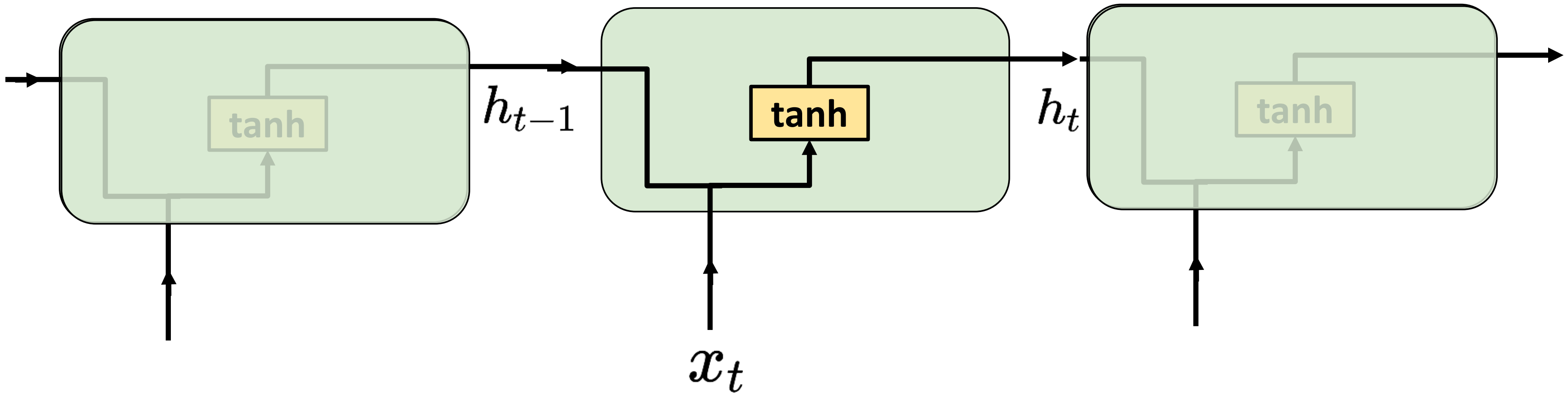
- Reuse the same weight matrix at every time step
- Makes the network easier to train

Image source: https://colah.github.io/

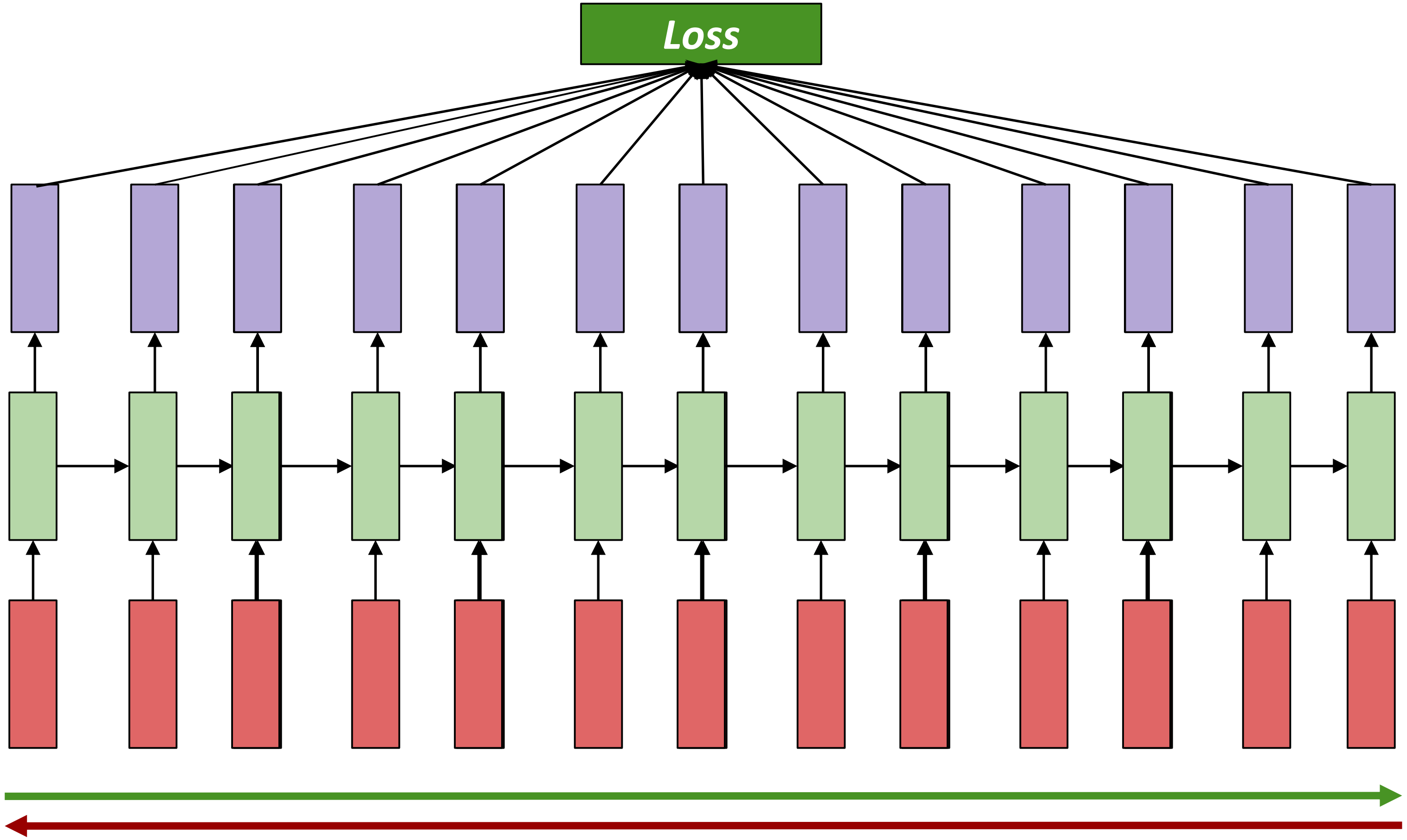
- **Simple RNN unwrapped over time**

$$h_t = tahn(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$y_t = W_{hy} h_t$$



Backpropagation through time (BPTT)



Forward

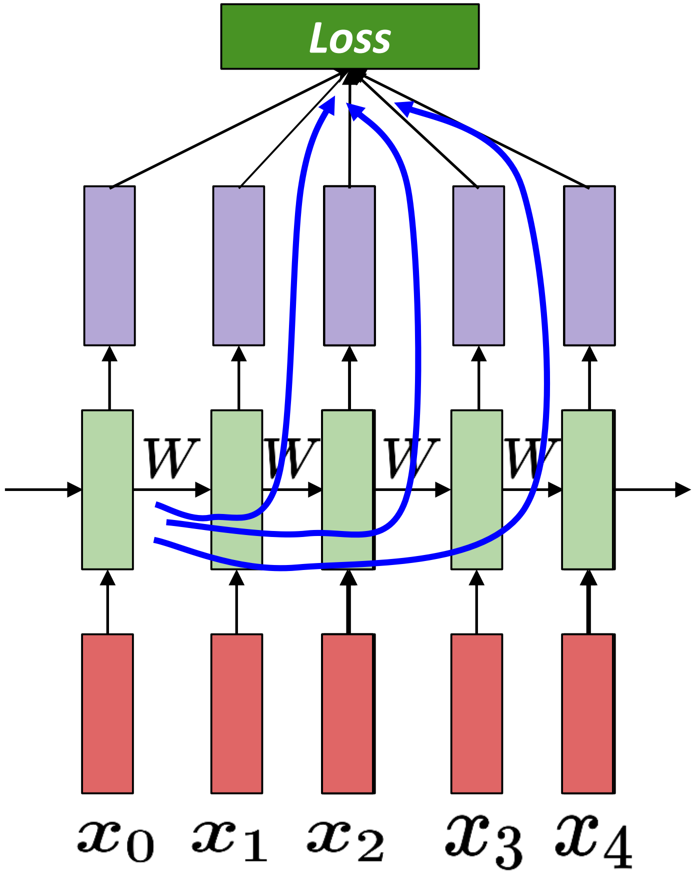
Run through entire sequence to compute the ***loss***

Backward

Run through entire sequence to compute the **gradient** and update the weight matrix:

$$w \leftarrow w - \eta \partial L / \partial w$$

Gradient flow in simple RNNs



$$w \leftarrow w - \eta \partial L / \partial w$$

Issue: **W** occurs each timestep

Every path from **W** to ***Loss*** is one dependency

All paths from **W** to ***Loss*** need to be involved

$$\frac{\partial L}{\partial w} = \sum_{j=0}^{T} \frac{\partial L_j}{\partial w}$$

$$\frac{\partial L}{\partial w} = \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \boxed{\left(\prod_{t=k+1}^{j} \frac{\partial h_t}{\partial h_{t-1}} \right)} \frac{\partial h_k}{\partial w}$$

Repeated matrix multiplications leads to ***vanishing*** and ***exploding*** gradients.

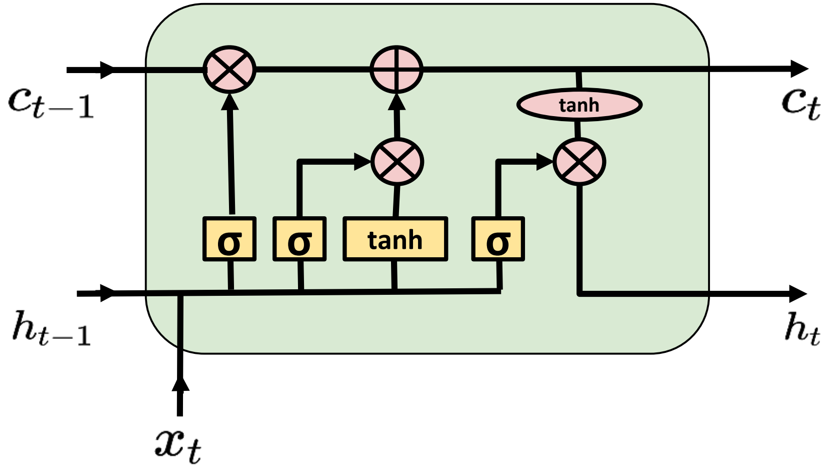
Highly effective sequence models

- Gated RNNs are based on the idea of creating paths that have derivatives that neither vanish nor explode
- Gated RNNs have connection weights that may change at each time step
- Gated RNNs also allow a network to forget an old state
- Instead of manually deciding when to clear the state, the network to learn to decide when to do it.

Image source: https://colah.github.io/

LSTM network

- There are four interacting networks: cell state, input gate, forget gate, output gate



$$g_t = tanh\left(W_g \cdot [h_{t-1}, x_t] + b_g\right)$$

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

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

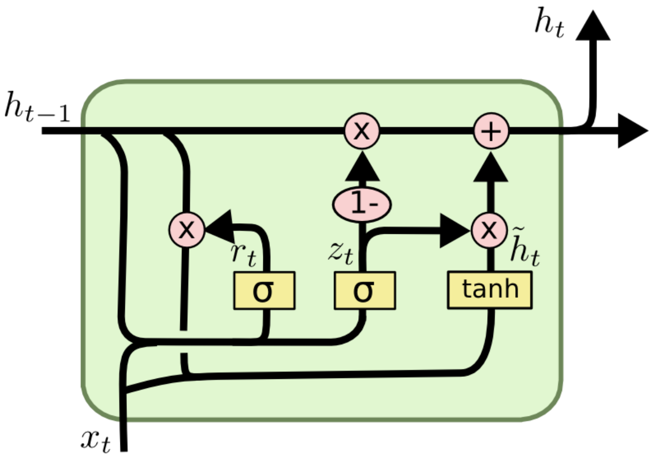
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

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

$$h_t = o_t \odot tanh(c_t)$$

Gated Recurrent Unit

- Single gating unit simultaneously controls the forgetting factor and the decision to update the state unit



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right)$$

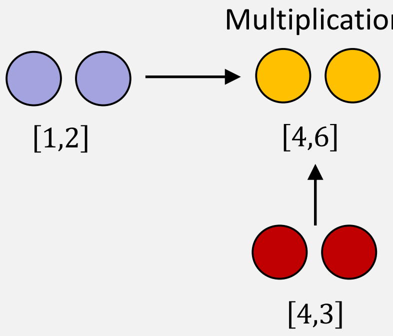
$$g_t = tanh\left(W_g \cdot [r_t \odot h_{t-1}, x_t] + b_g\right)$$

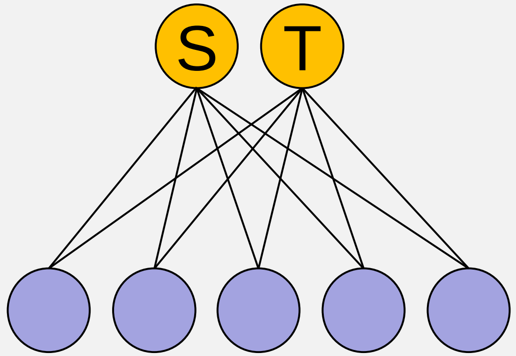
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot g_t$$

Image source: https://colah.github.io/

Guide to upcoming illustrations

Element-Wise Multiplication

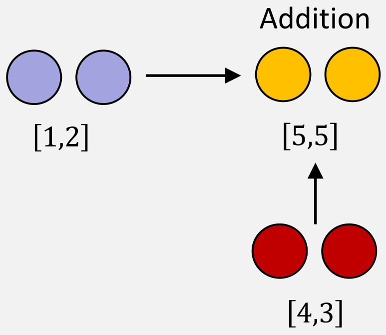




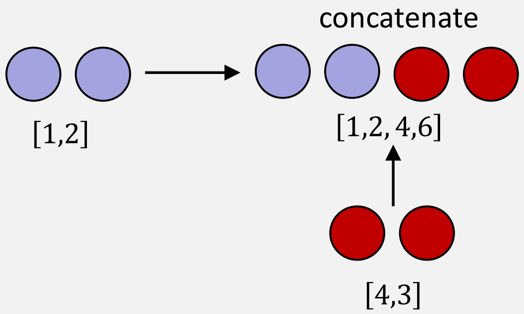
S is a weighted sum of from the previous layer and activated with a **sigmoid** function

T is a weighted sum of from the previous layer and activated with a **tanh** function

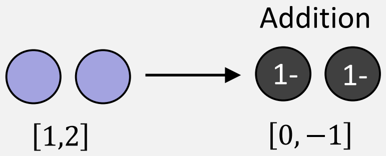
Element-Wise Addition



Concatenation



Subtract from 1



- **Sigmoid Activation Function**

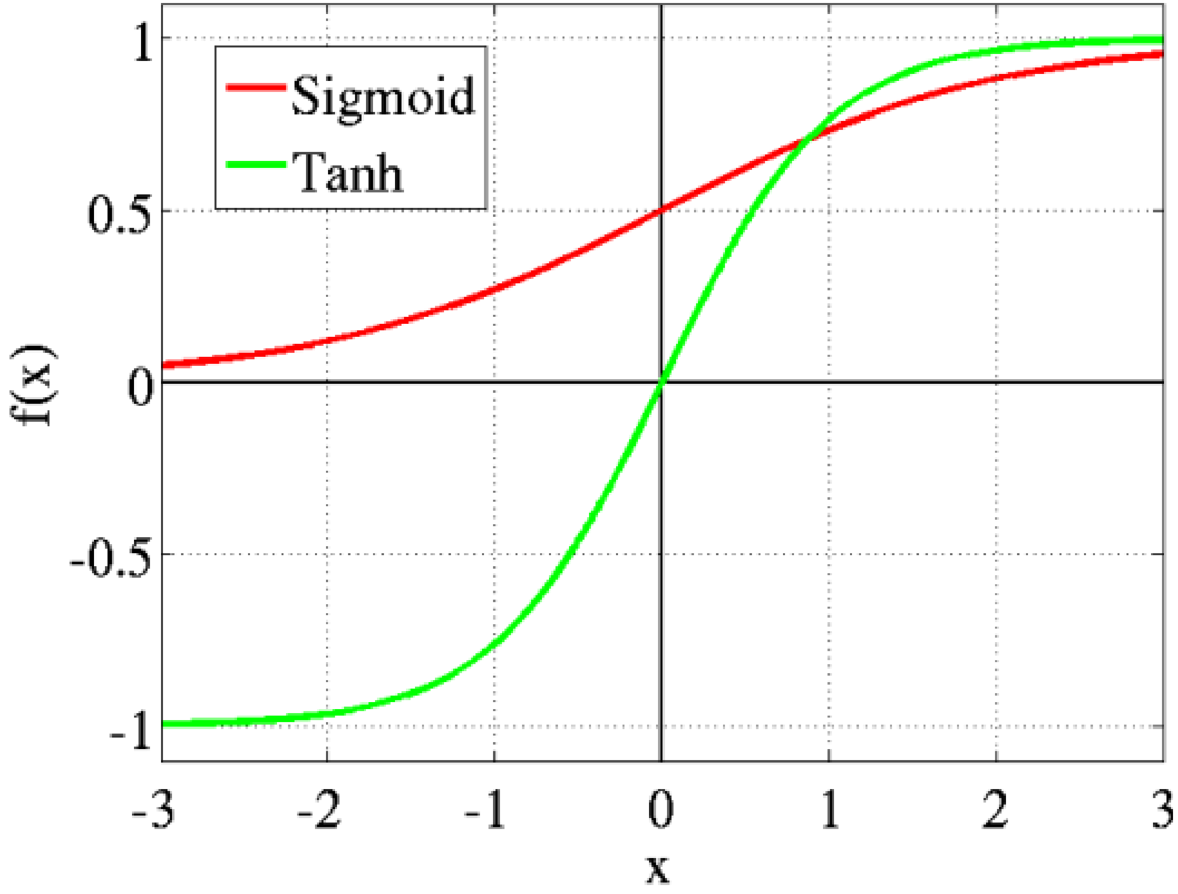
$$\sigma(x) = \frac{1}{1 + e^{x}}$$

Range: 0 to 1

- **Tanh Activation Function**

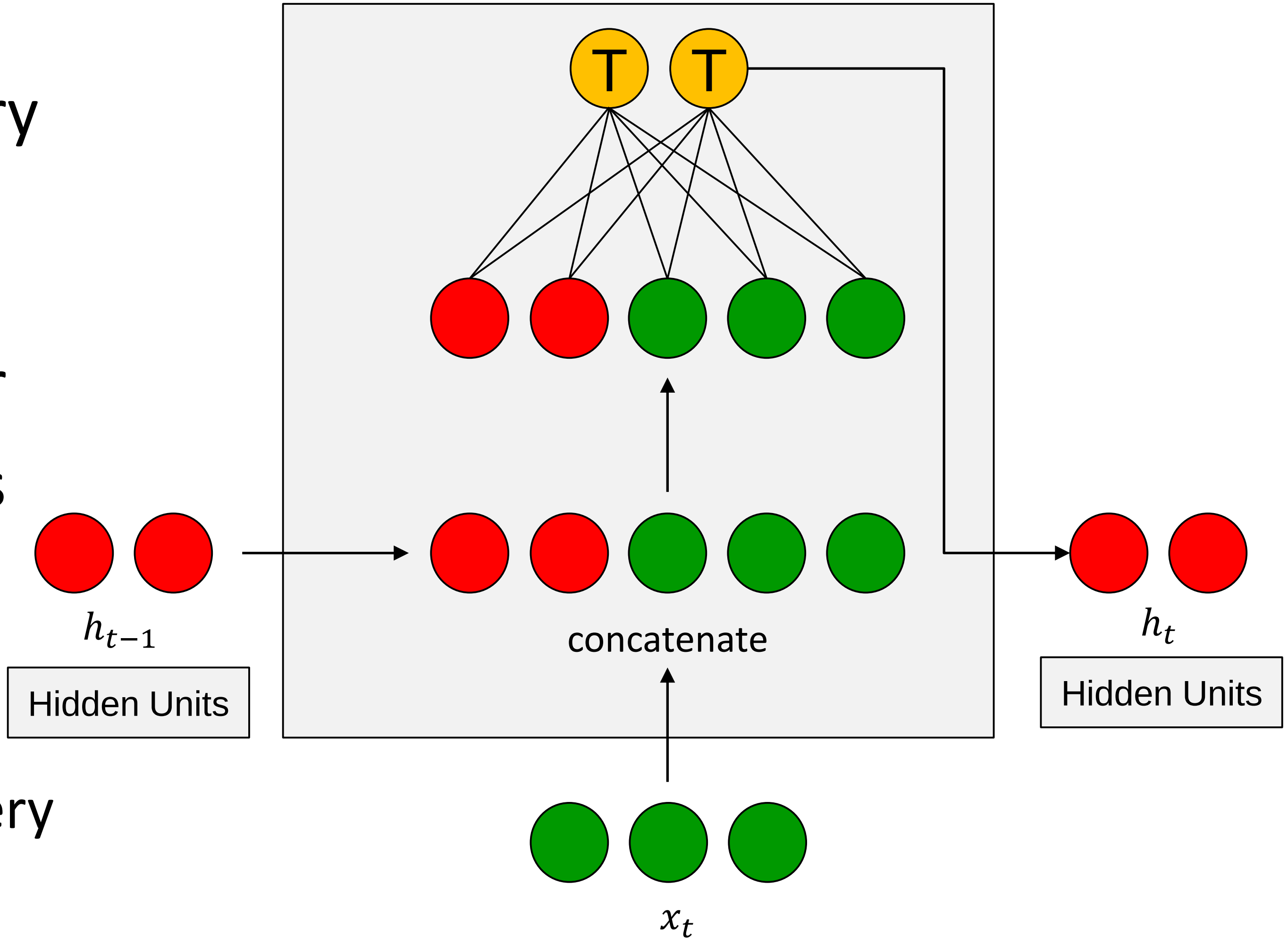
$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Range: -1 to 1



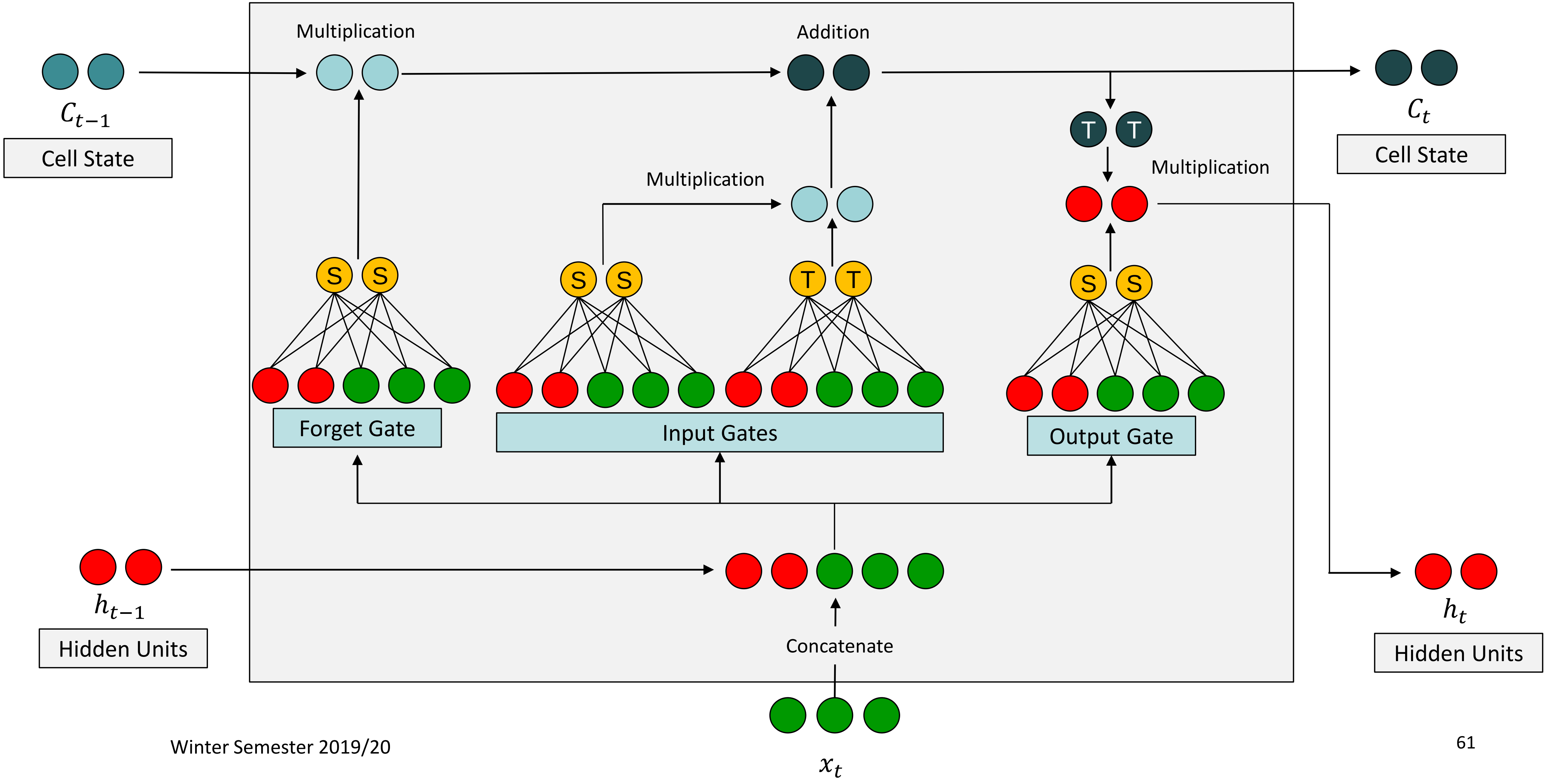
Universität Augsburg University

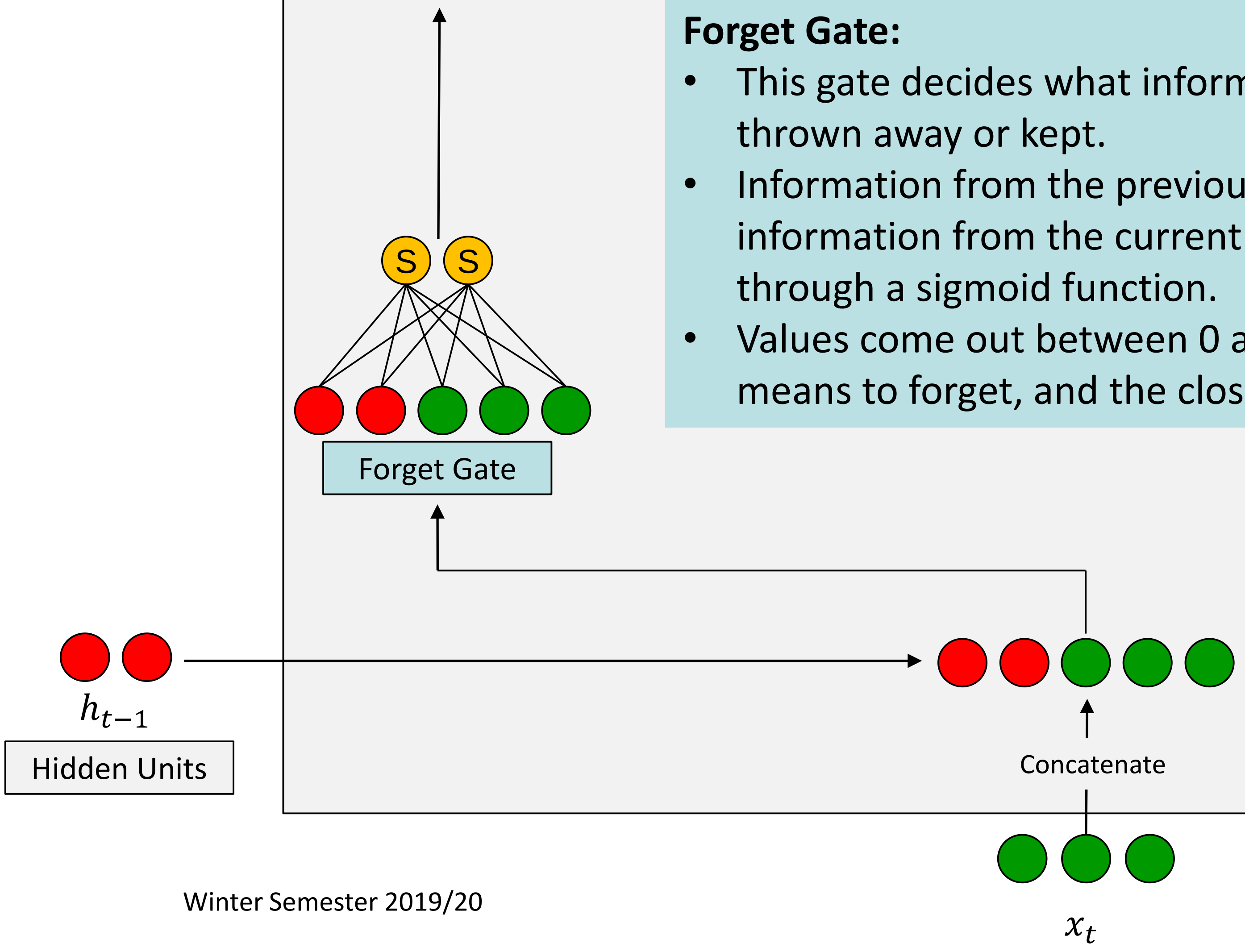
- **Vanilla RNNs**
  - Suffer from short-term memory
  - If a sequence is long enough they cannot carry information from earlier time steps to later
  - Vanishing/Exploding Gradients
    - If a gradient value becomes extremely small, it doesn't contribute too much learning
    - Large error gradients result in very large updates during training.



- **The LSTM Recurrent Unit**
  - The internals structure of an LSTM unit allows is to keep or forget information over time
  - Cell State: the memory of the network, carries relevant information throughout the processing of the sequence
  - Information get’s added or removed to the cell state via gates
    - Forget gate: what is relevant to keep from prior step
    - Input gate: what information is relevant to add from current step
    - Output gate: what the next hidden state should be
  - The gates learns to determine what is relevant during training

Long Short-term Memory (LSTM)



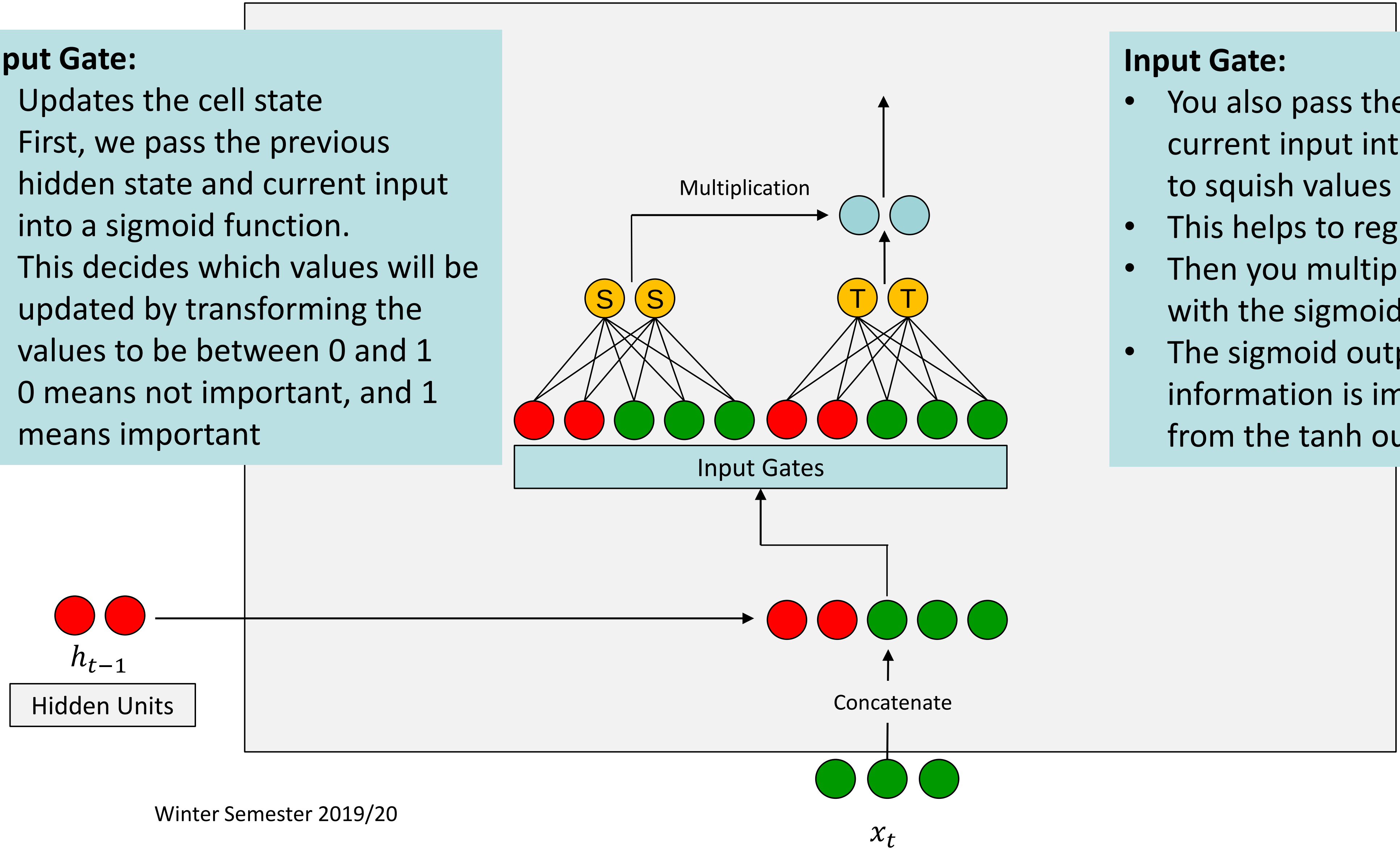


Forget Gate:

- This gate decides what information should be thrown away or kept.
- Information from the previous hidden state and information from the current input is passed through a sigmoid function.
- Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep

Input Gate:

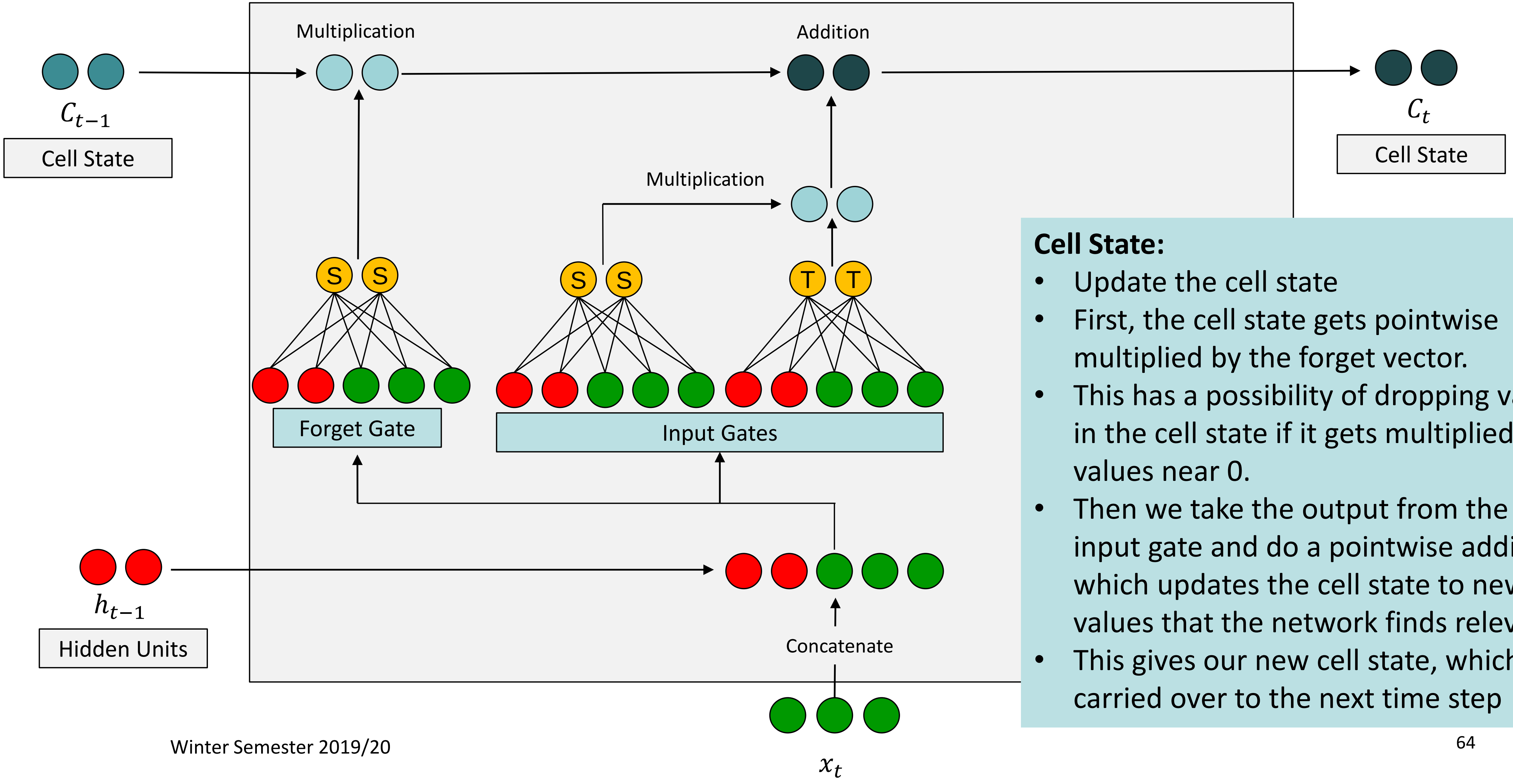
- Updates the cell state
- First, we pass the previous hidden state and current input into a sigmoid function.
- This decides which values will be updated by transforming the values to be between 0 and 1
- 0 means not important, and 1 means important



Input Gate:

- You also pass the hidden state and current input into the tanh function to squish values between -1 and 1
- This helps to regulate the network.
- Then you multiply the tanh output with the sigmoid output.
- The sigmoid output will decide which information is important to keep from the tanh output

Long Short-term Memory (LSTM)



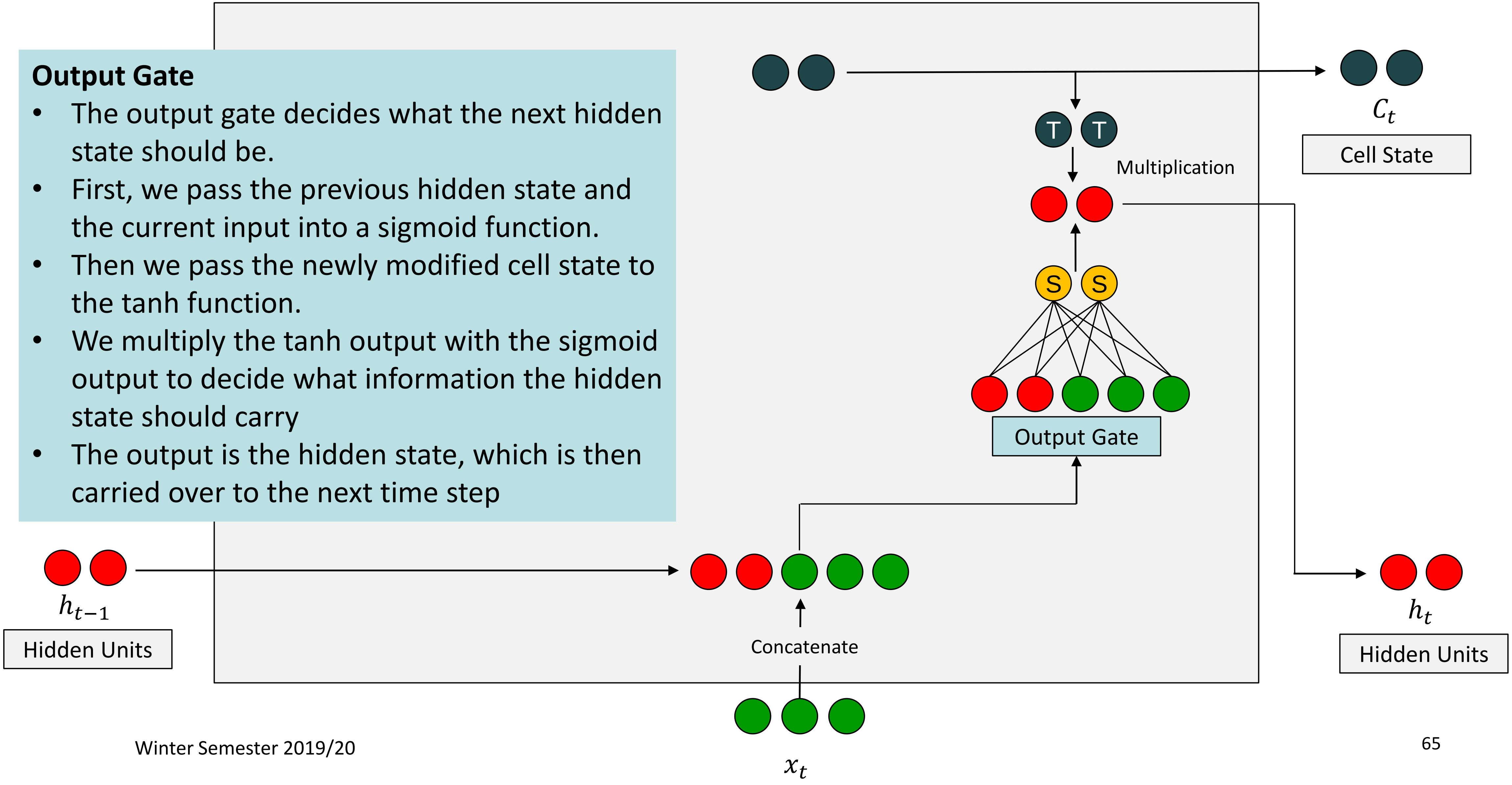
Cell State:

- Update the cell state
- First, the cell state gets pointwise multiplied by the forget vector.
- This has a possibility of dropping values in the cell state if it gets multiplied by values near 0.
- Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the network finds relevant
- This gives our new cell state, which is carried over to the next time step

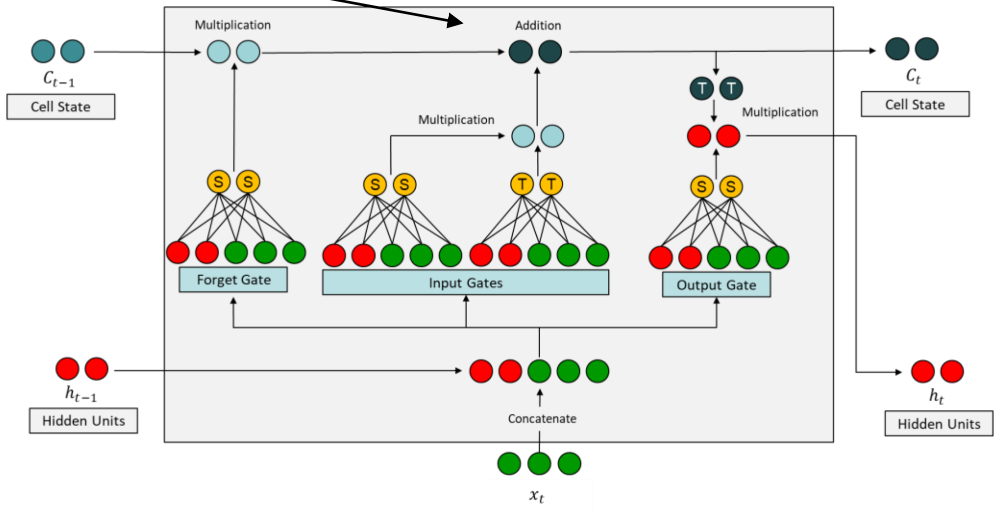
Long Short-term Memory (LSTM)

Output Gate

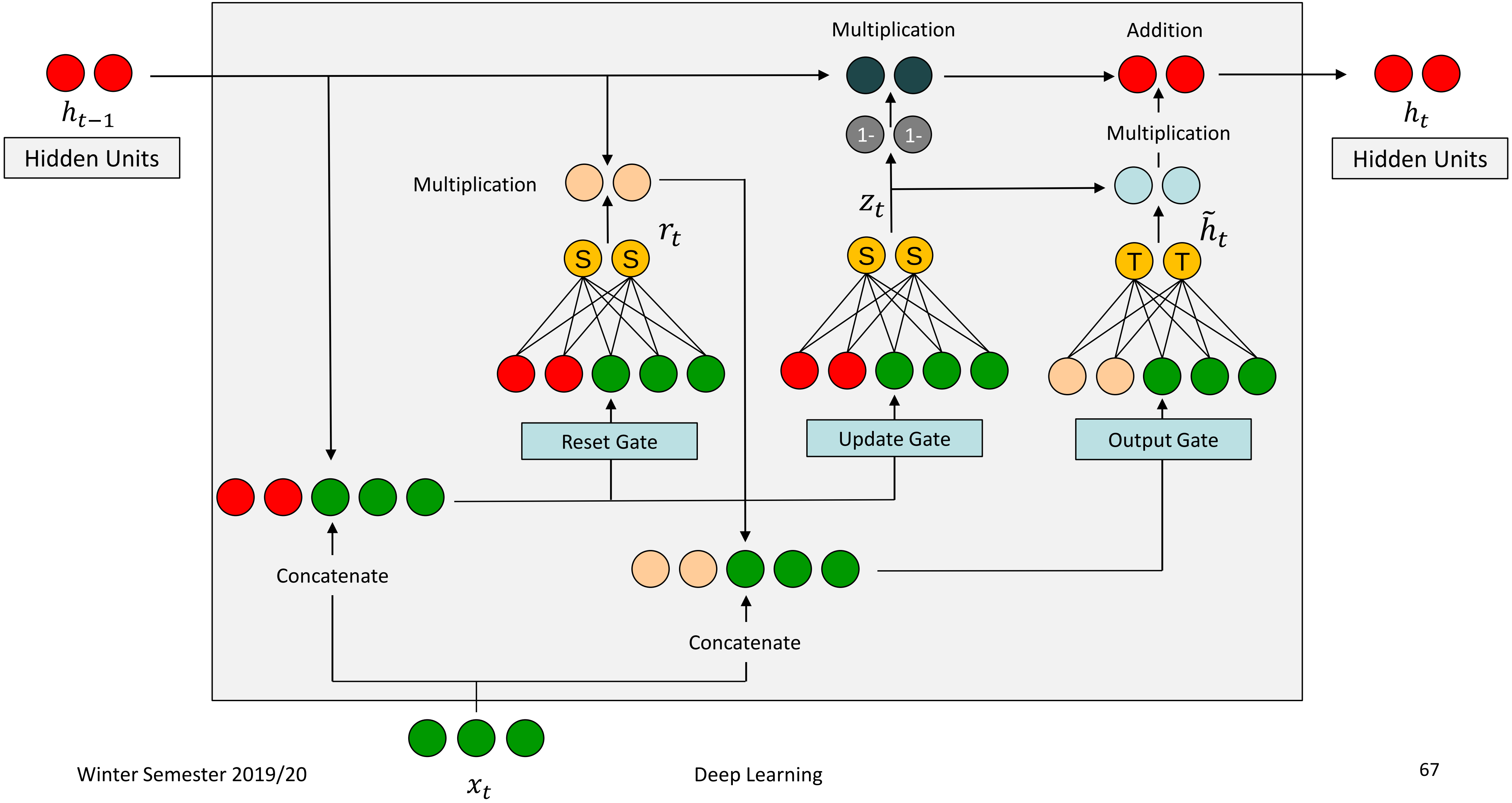
- The output gate decides what the next hidden state should be.
- First, we pass the previous hidden state and the current input into a sigmoid function.
- Then we pass the newly modified cell state to the tanh function.
- We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry
- The output is the hidden state, which is then carried over to the next time step



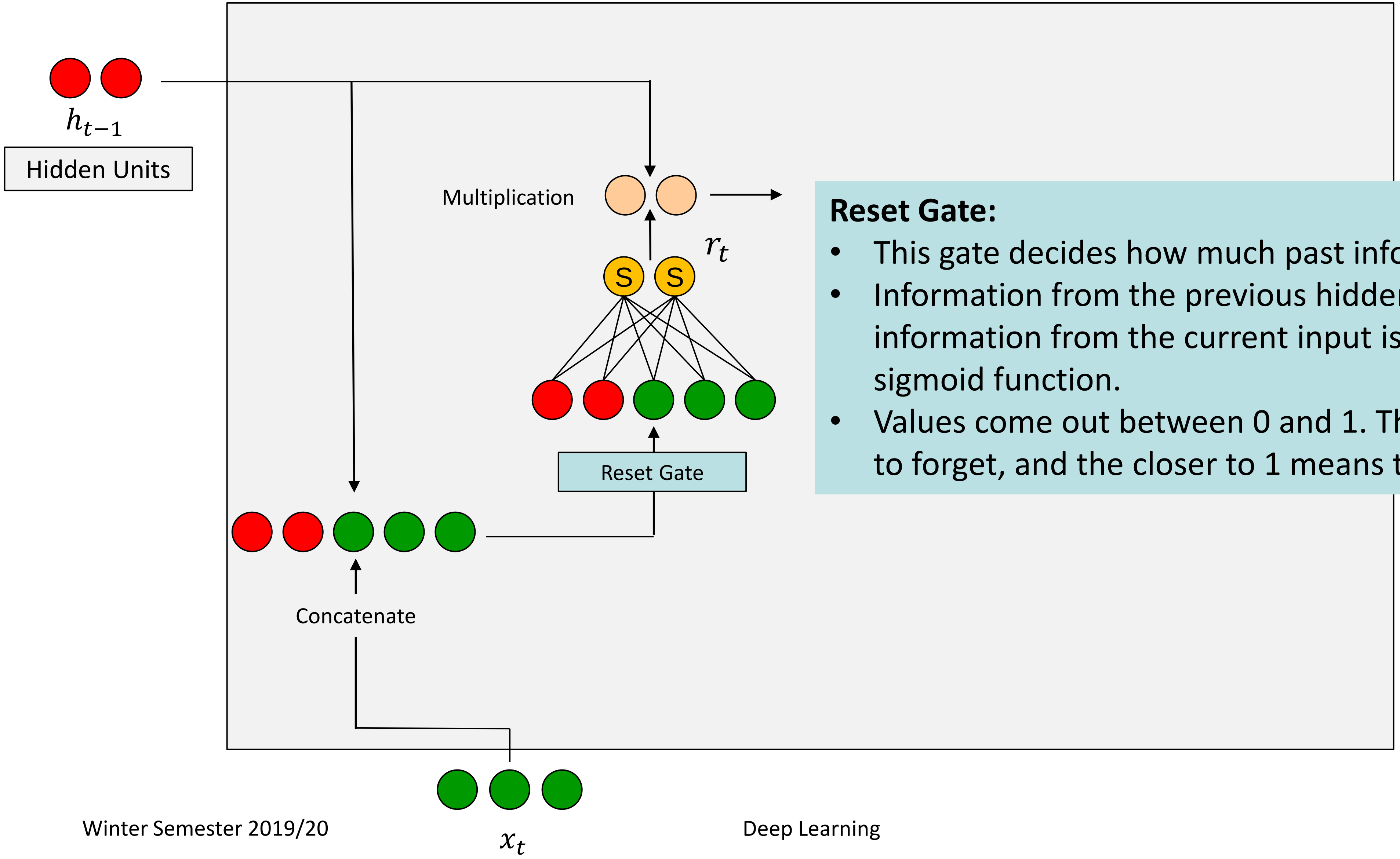
- **LSTM can deal with vanishing gradients:**
  - Previous cell state and input are added together
  - The influence of the previous state never completely disappears unless the forget gate is closed
  - Gradient flow is therefore improved
  - Information is stored/retrieved over longer time periods



Gated Reccurent Unit



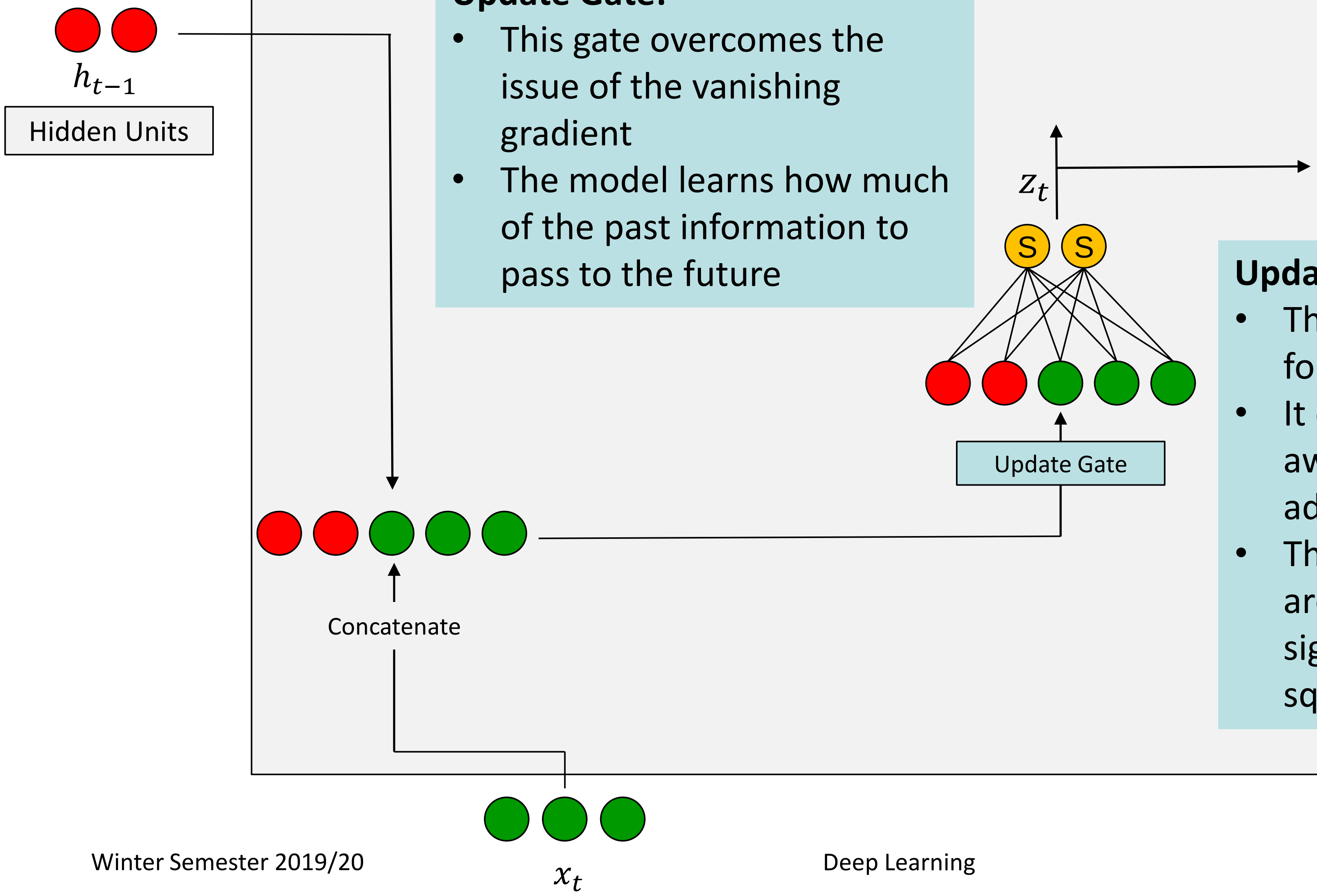
Gated Reccurent Unit



Reset Gate:

- This gate decides how much past information to forget
- Information from the previous hidden state and information from the current input is passed through a sigmoid function.
- Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep

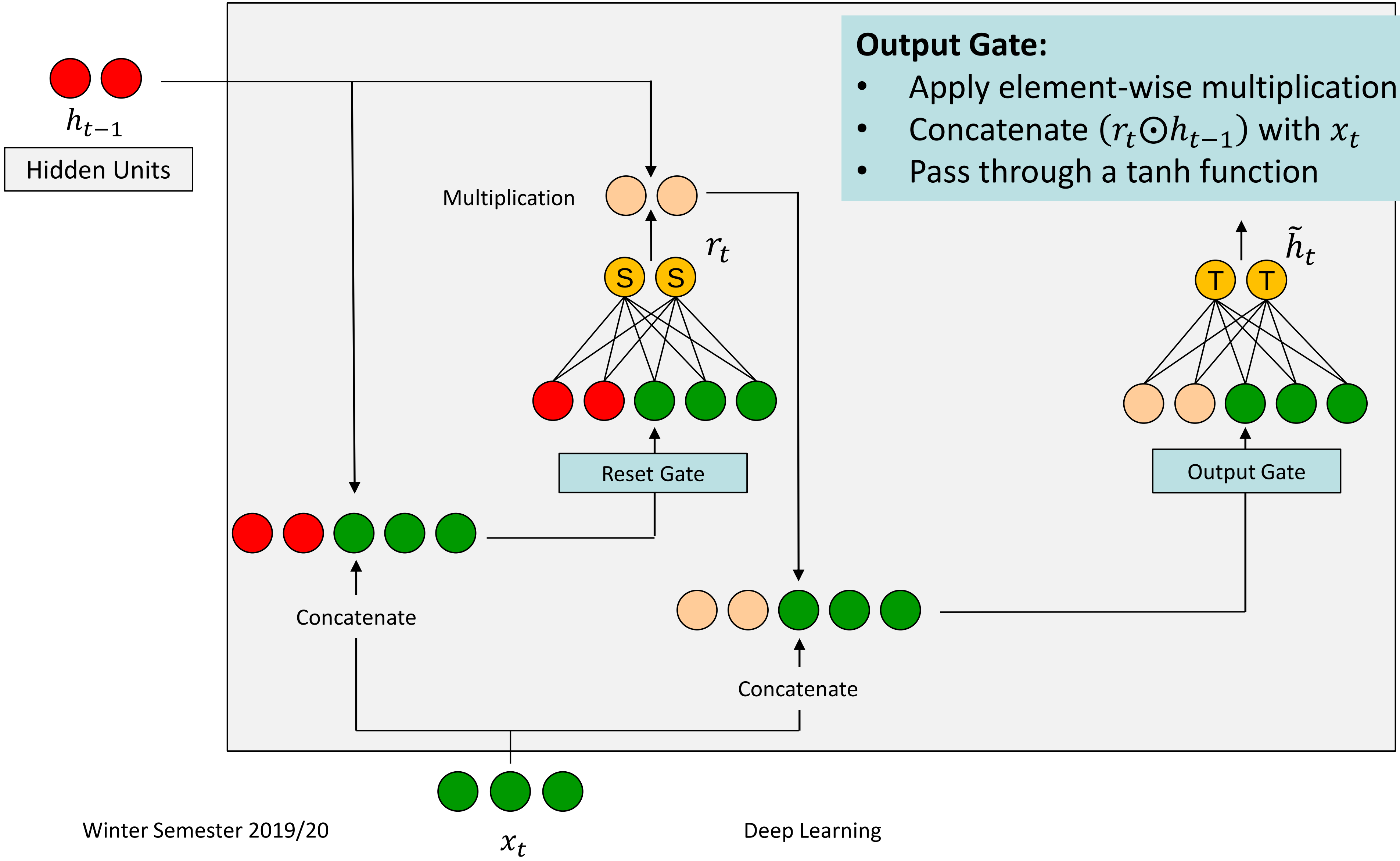
Gated Reccurent Unit



Update Gate:
- This gate overcomes the issue of the vanishing gradient
- The model learns how much of the past information to pass to the future

Update Gate:
- The update gate acts similar to the forget and input gate of an LSTM.
- It decides what information to throw away and what new information to add
- The previous state and current input are concentrated together and a sigmoid function is applies to squeeze the output between 0 and 1

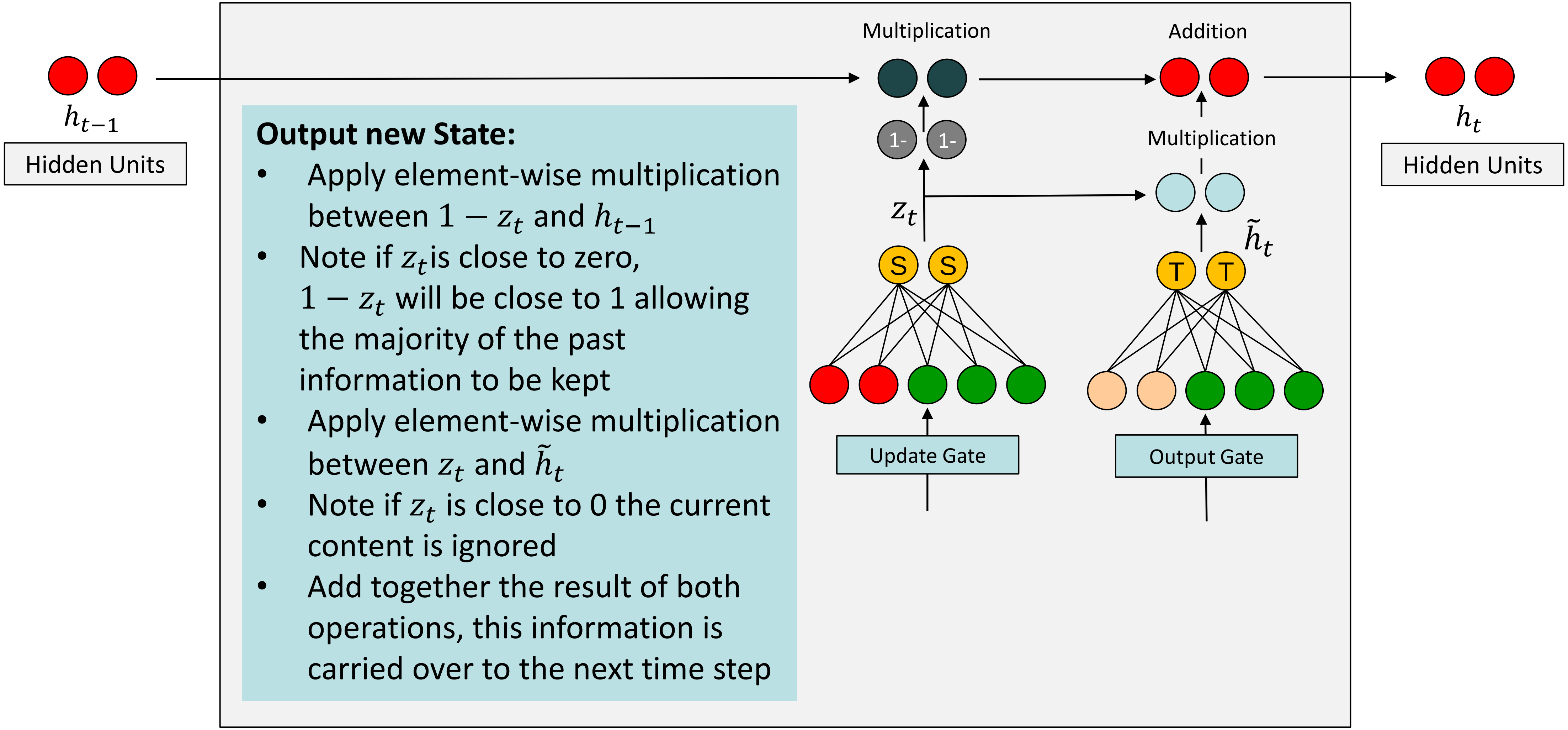
Output Gate:
- Apply element-wise multiplication between r_t and h_{t-1}
- Concatenate $(r_t \odot h_{t-1})$ with x_t
- Pass through a tanh function



Gated Reccurent Unit

Output new State:

- Apply element-wise multiplication between $1 - z_t$ and h_{t-1}
- Note if z_t is close to zero, $1 - z_t$ will be close to 1 allowing the majority of the past information to be kept
- Apply element-wise multiplication between z_t and $\tilde{h}_t$
- Note if z_t is close to 0 the current content is ignored
- Add together the result of both operations, this information is carried over to the next time step



Overview

Image Source:
https://distill.pub/2017/feature-visualization/

- Introduction
- Feed Forward Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- **Sequence to Sequence**
- Regularisation
- Explainable AI

Parts (layers mixed4b & mixed4c)

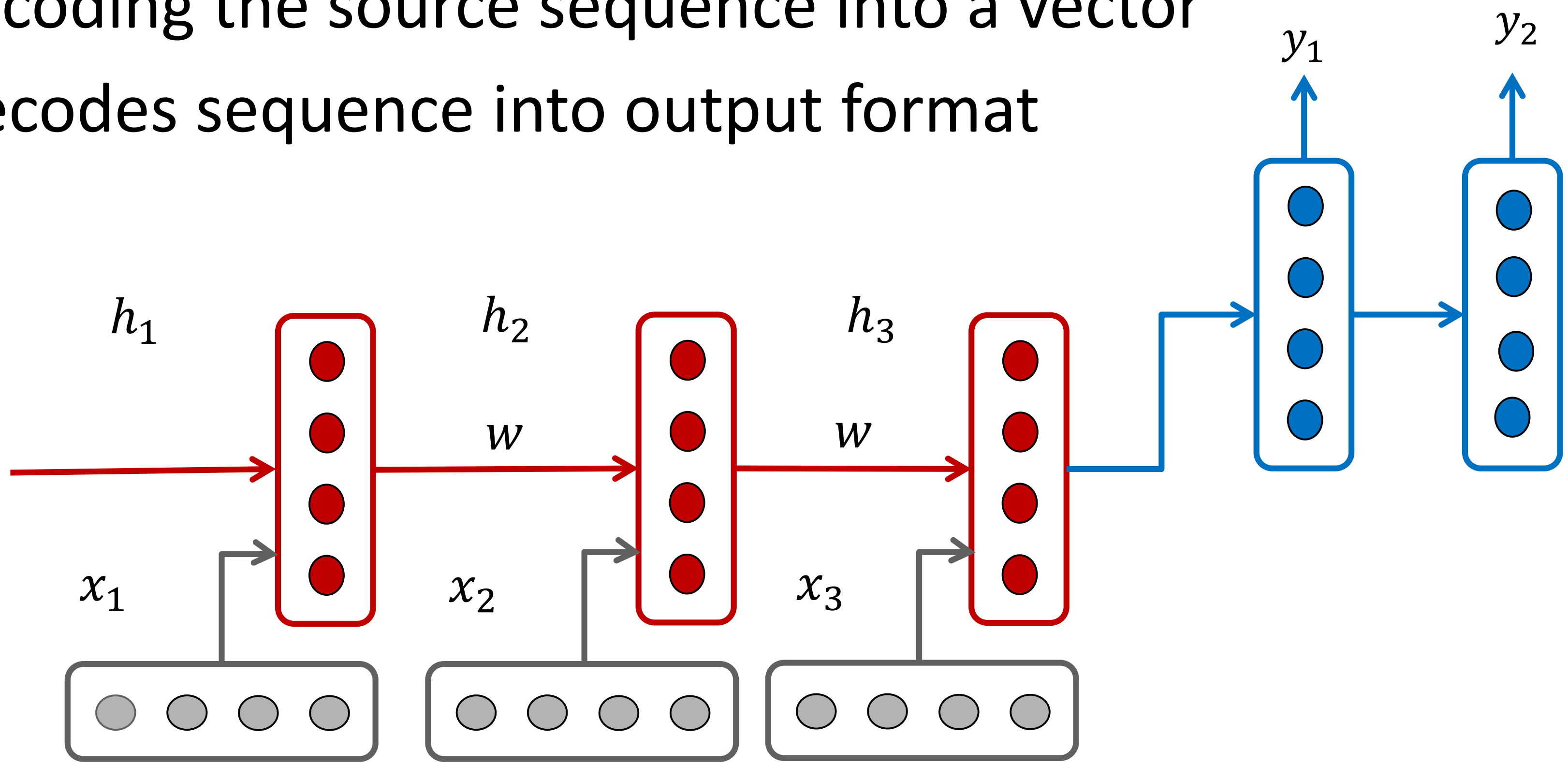
Aim:

- Build and train a single, large neural network that reads an input sequence and outputs an alternate sequence
  - E.g., for language translation applications

Core Idea:

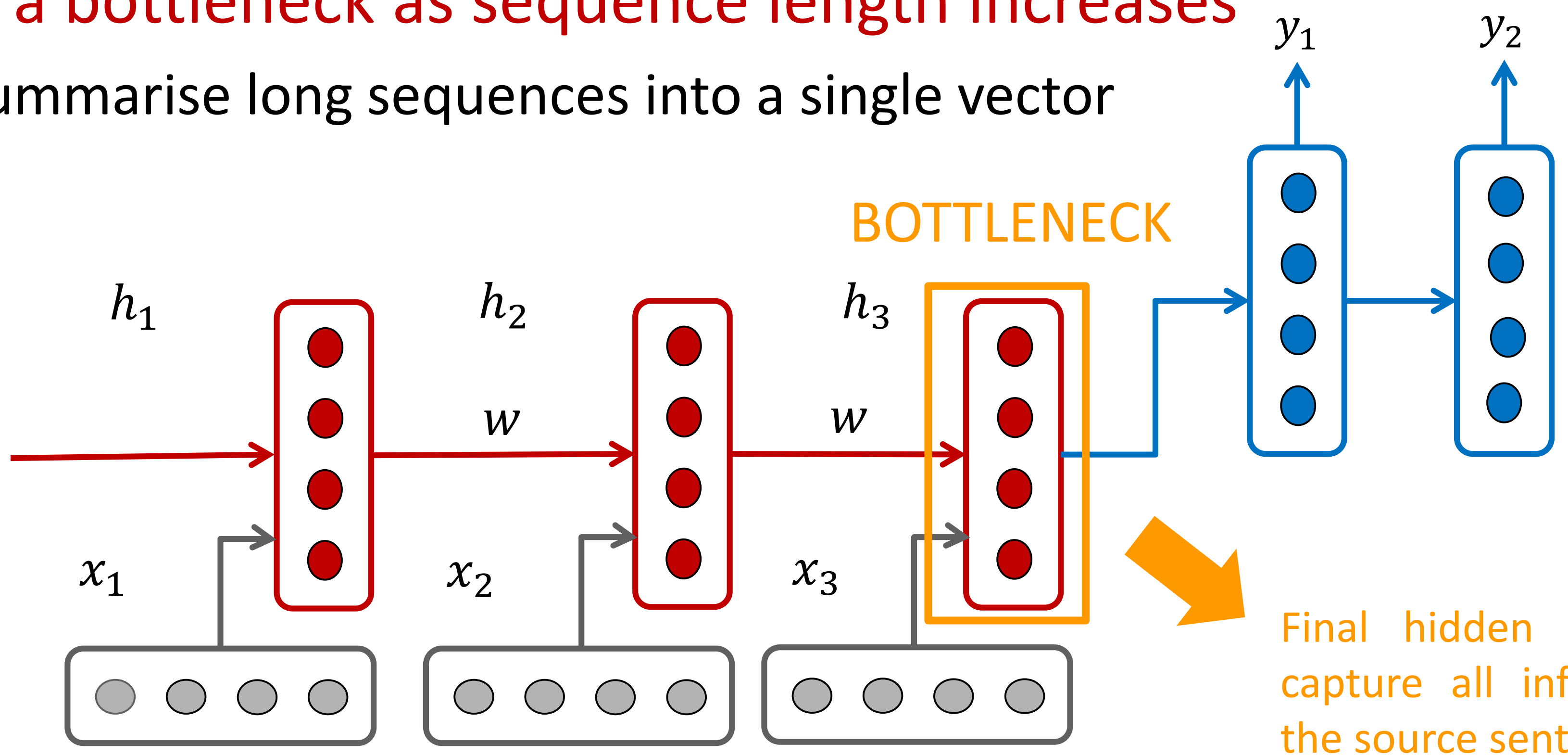
- Use two RNNs in an encoder-decoder architecture
  - Encoder: models the input sequence to obtain a vector representation of a fixed dimensionality
  - Decoder: uses the output of the encoder as an input and extract the output sequence using another RNN

- **Sequence to sequence model**
  - Encoding the source sequence into a vector
  - Decodes sequence into output format



Sequence modelling bottleneck

- Only the final encoder state is used to initialise the decoder
- This becomes a bottleneck as sequence length increases
  - Difficult to summarise long sequences into a single vector



Final hidden state needs to capture all information about the source sentence

- **Attention Mechanisms**
  - **Do not discard intermediate encoder states**, instead utilise all states in order to construct an new context vector
    - Probability distribution mapping each input to the output state that the decoder wants to generate
  - Use this context vector when decoding the output sequence
  - This means the decoder captures global information rather than solely making inferences based on a single hidden state
  - During training the new network learns which inputs are important for the task, hence the name attention

Core Idea

- Attention places different focus on different parts of the input sequence assigning each input with a score.
- Then the encoder hidden states are aggregated using a weighted sum to produce a context vector which is also supplied to the decoder

Key steps

1. Obtain a score for every encoder hidden state
2. Run all the scores through a softmax layer
3. Multiply each encoder hidden state by its softmaxed score
4. Sum up the resulting vectors
5. Feed the context vector into the decoder

Attention Mechanisms

Image Source: https://towardsdatascience.com/

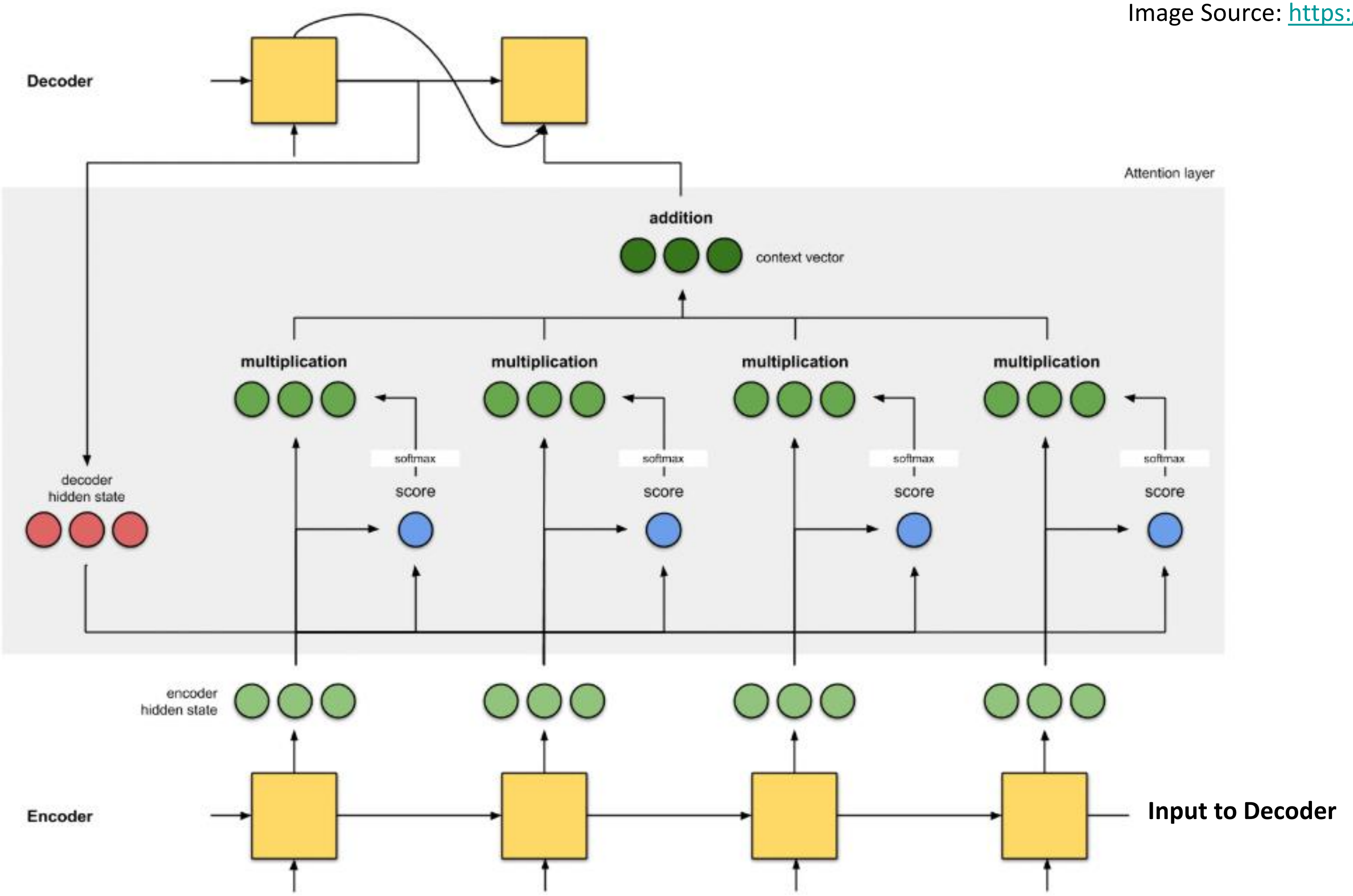
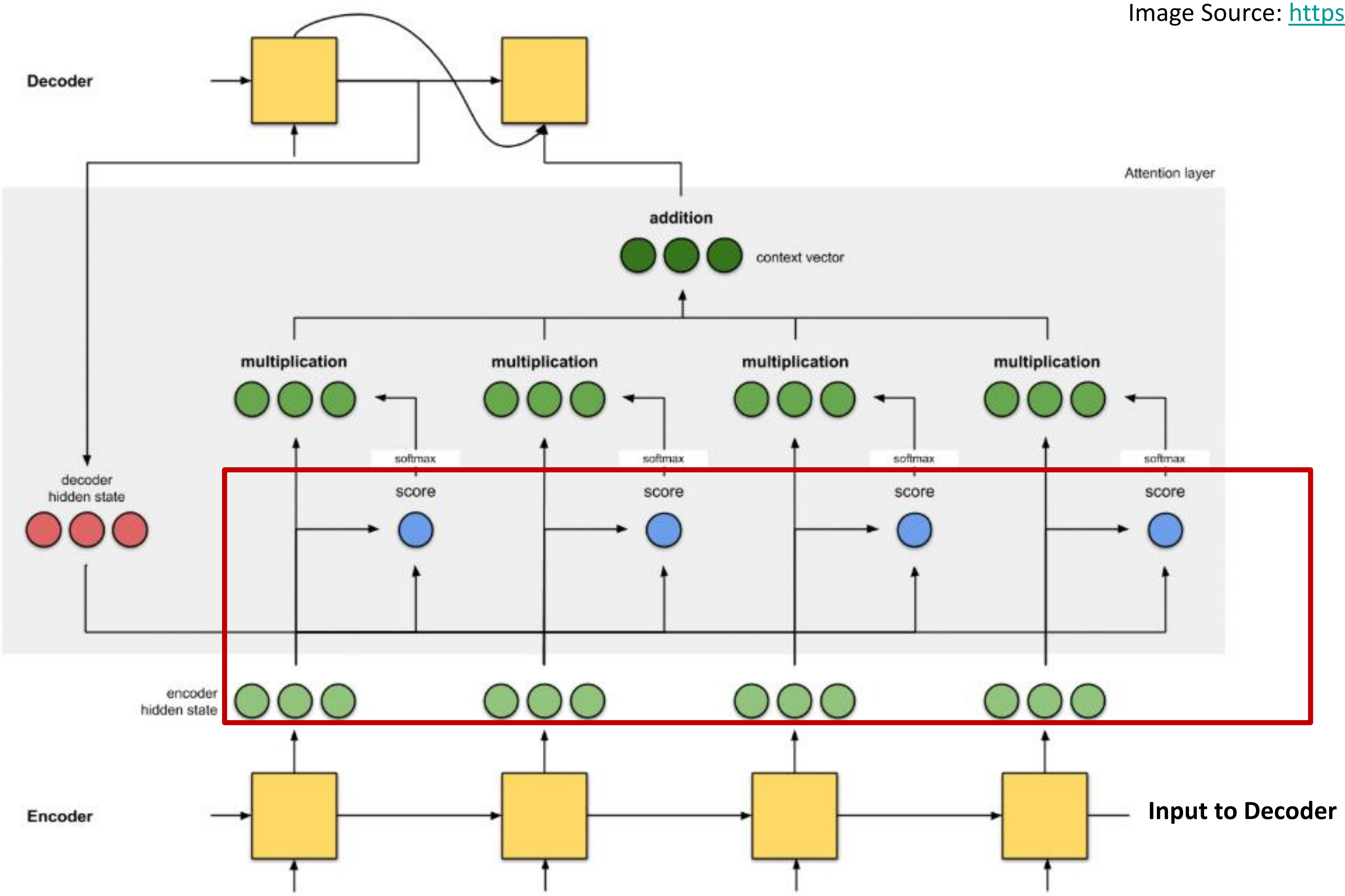


Image Source: https://towardsdatascience.com/



1. Obtain a score for every encoder hidden state

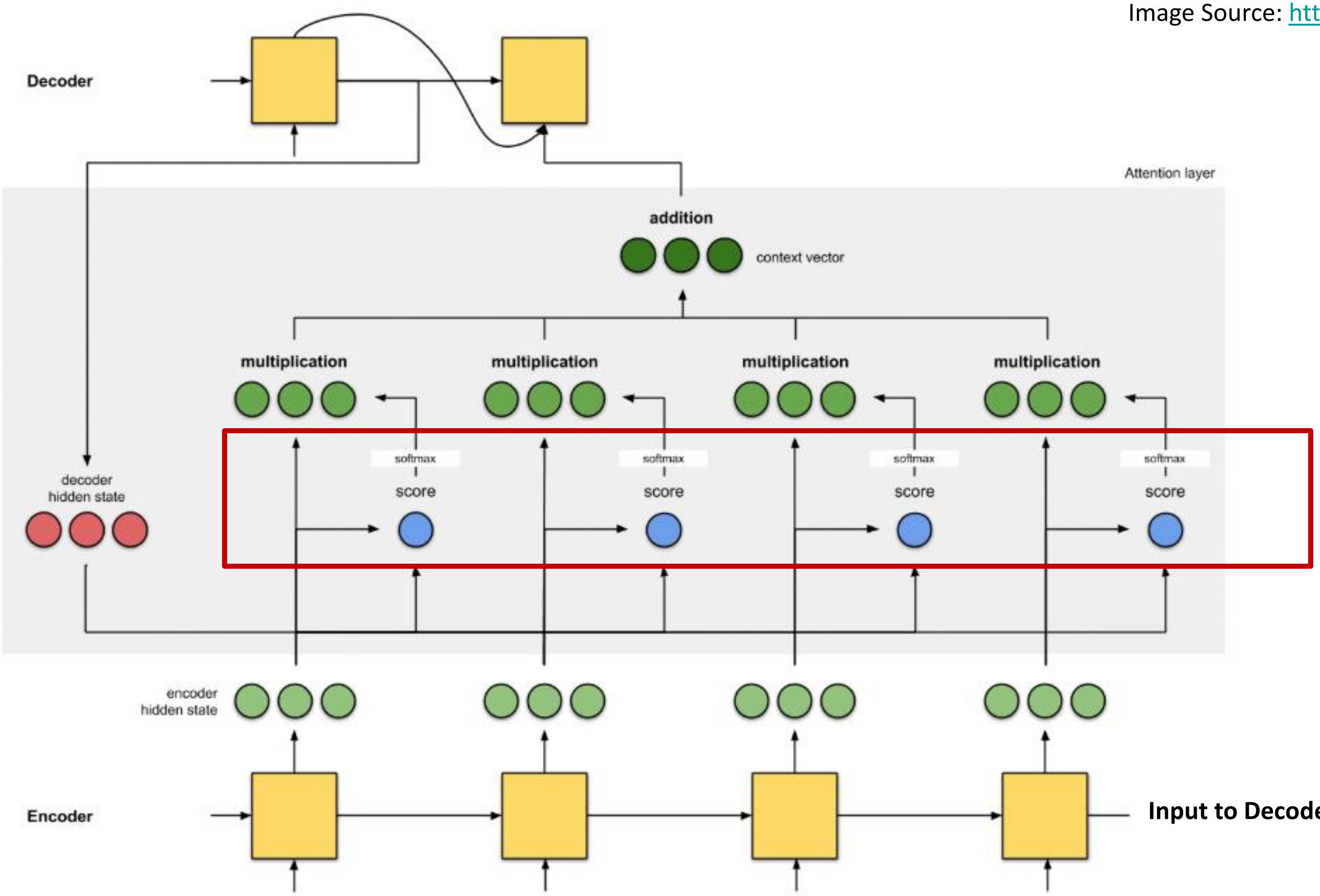
Step 1:

- A (scalar) score is obtained by a score function
- Typically the score function is a dot product between the decoder and encoder hidden states.
- E.g.,

```
decoder_hidden = [10, 5, 10]

encoder_hidden  score
---------------------
     [0, 1, 1]     15 (= 10×0 + 5×1 + 10×1, the dot product)
     [5, 0, 1]     60
     [1, 1, 0]     15
     [0, 5, 1]     35
```

Image Source: https://towardsdatascience.com/



Input to Decoder

2. Run all the scores through a softmax layer

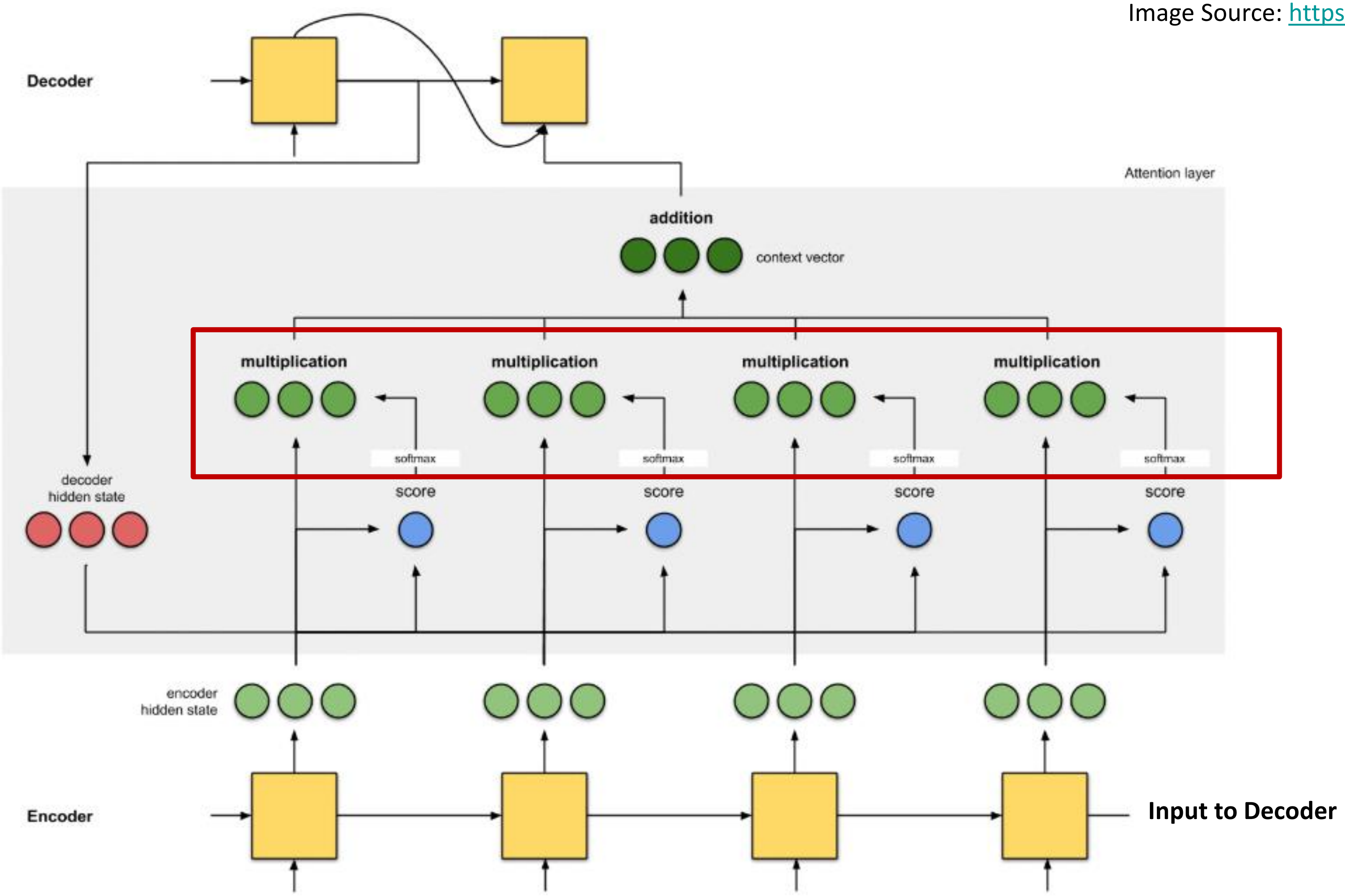
Step 2:

- Put the scores through a softmax layer so that the softmaxed scores add up to 1.
- These softmaxed scores represent the *attention distribution*
- E.g.,

```
encoder_hidden  score  score^
-----------------------------
     [0, 1, 1]     15       0
     [5, 0, 1]     60       1
     [1, 1, 0]     15       0
     [0, 5, 1]     35       0
```

Image Source: https://towardsdatascience.com/

3. Multiply each encoder hidden state by its softmaxed score



Step 3:

- Multiplying each encoder hidden state with its softmaxed score (scalar), to obtain the *alignment vectors*
- E.g.,

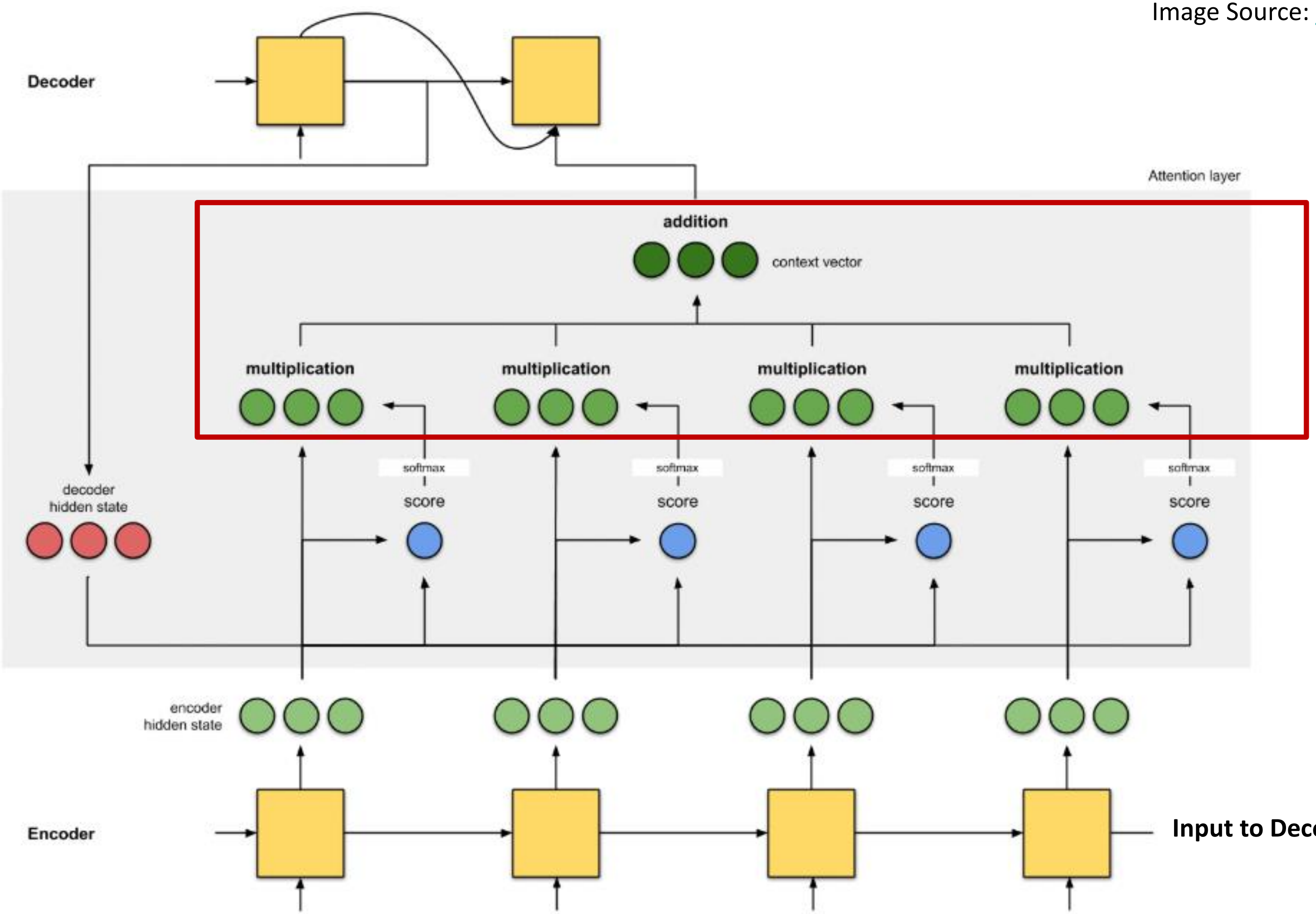
```
encoder   score  score^  alignment
---------------------------------
[0, 1, 1]   15       0   [0, 0, 0]
[5, 0, 1]   60       1   [5, 0, 1]
[1, 1, 0]   15       0   [0, 0, 0]
[0, 5, 1]   35       0   [0, 0, 0]
```

Attention Mechanisms

Universität Augsburg University

Image Source: https://towardsdatascience.com/

4. Sum up the resulting vectors



Input to Decoder

Step 4:

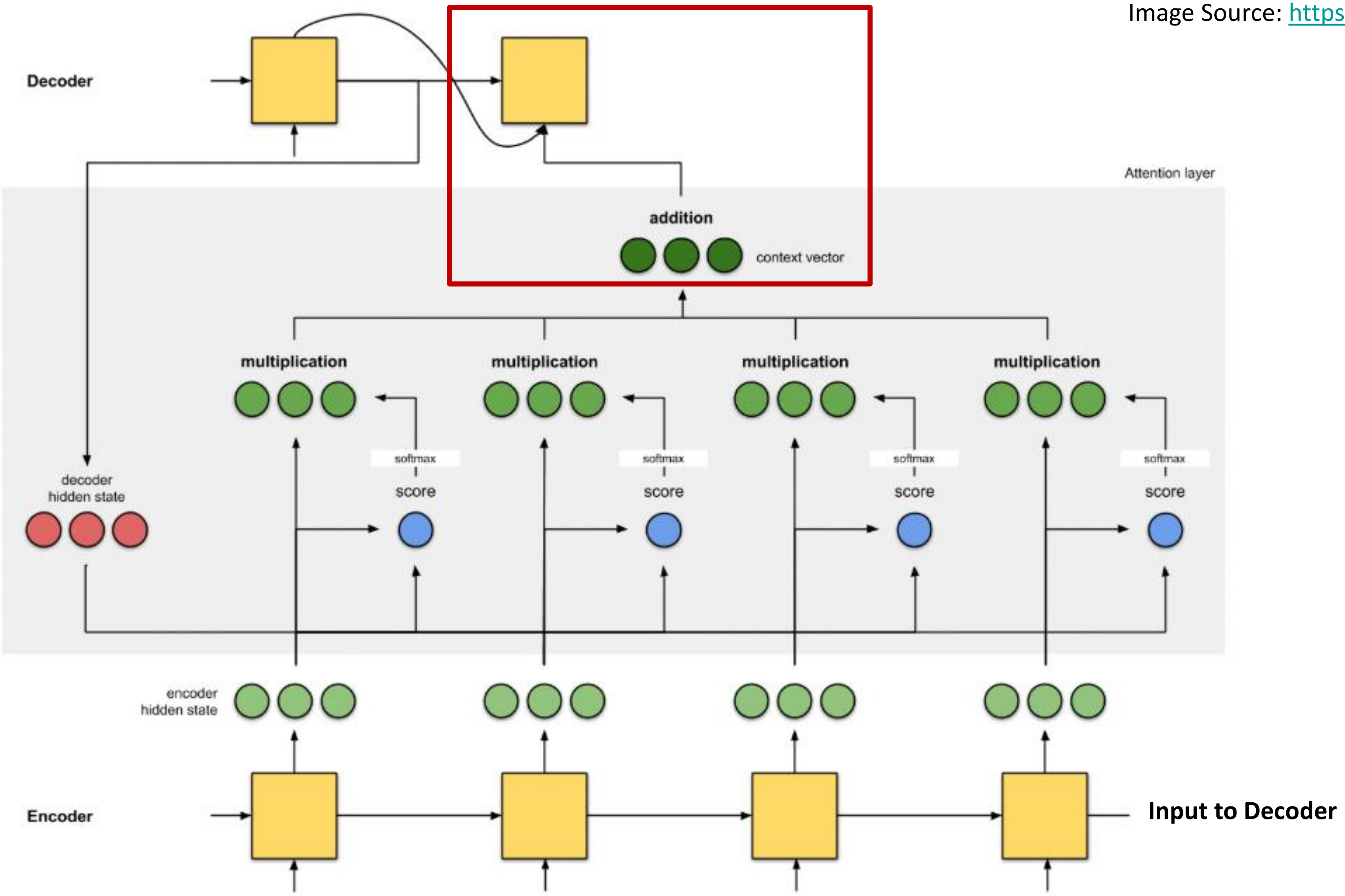
- The alignment vectors are summed up to produce the context vector
- A context vector is an aggregated information of the alignment vectors from the previous step
- E.g.,

```
encoder   score   score^   alignment
---------------------------------
[0, 1, 1]   15      0   [0, 0, 0]
[5, 0, 1]   60      1   [5, 0, 1]
[1, 1, 0]   15      0   [0, 0, 0]
[0, 5, 1]   35      0   [0, 0, 0]


context = [0+5+0+0, 0+0+0+0, 0+1+0+0] = [5, 0, 1]
```

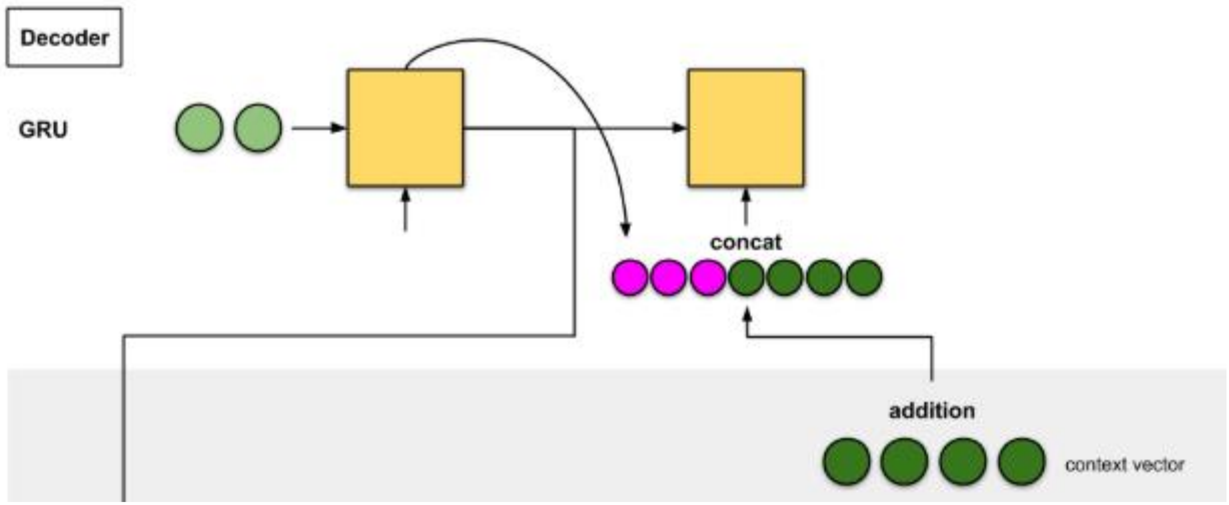
Image Source: https://towardsdatascience.com/

5. Feed the context vector into the decoder



Step 5:

- The manner this is done depends on the architecture design
  - Normal some form of concatenation with a decoder state or output
  - E.g.: Concatenation between the generated output from the previous decoder time step and context vector from the current time step



Overview

Image Source: https://distill.pub/2017/feature-visualization/

- Introduction
- Feed Forward Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Attention Mechanisms
- **Regularisation**
- Explainable AI

Objects (layers mixed4d & mixed4e)

Goal

- Learn a *robust* predictive function $f(\cdot)$
- A mapping from the feature space $\mathcal{X}$ to the label space $\mathcal{Y}$

$$\mathcal{X} \xrightarrow{f(\cdot)} \mathcal{Y}$$

- Given a test sample (unknown label), the learnt function maps the test feature vector $\boldsymbol{x}_*$ into a specific label $\boldsymbol{y}_*$

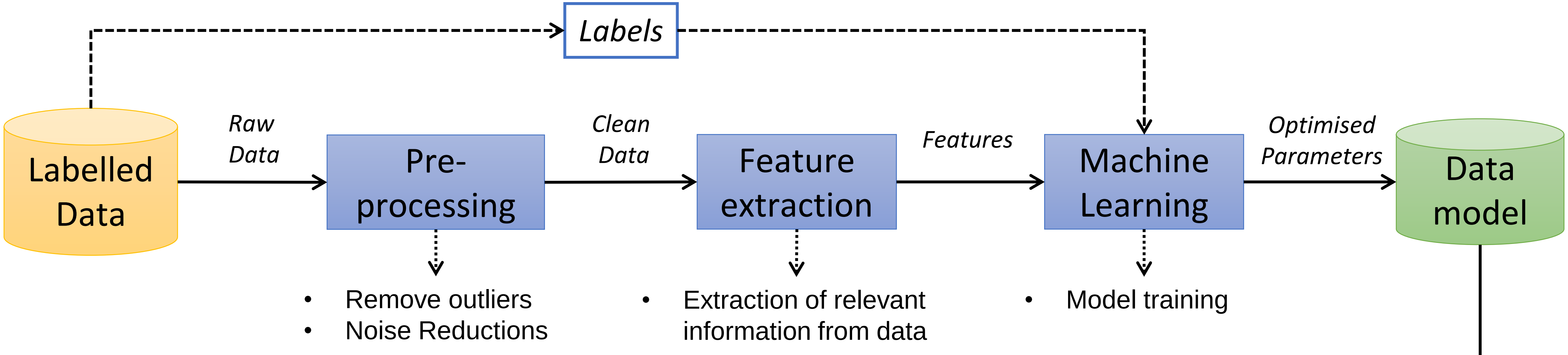
$$\boldsymbol{y}_* = f(\boldsymbol{x}_*)$$

- **Generalisation of a supervised model**
  - In machine learning we want to minimise both the **training error** and the **generalisation error**
  - We only ever have a finite number of training samples
  - There is a need to ensure the **generalisability** of a model
    - The model's ability to adequately label new test data samples
    - Data **not used** during the training/optimisation phase
  - **Generalization Error**: The error on these new data instances

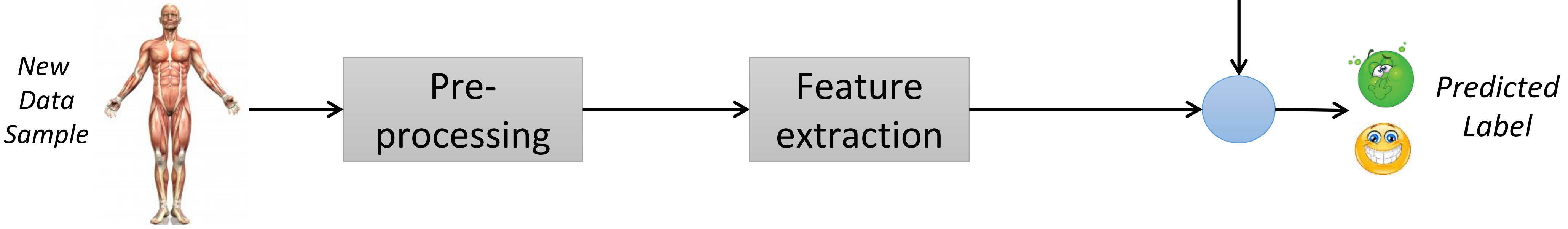
Training a machine learning model to generalise well to new (test) data is a challenging problem

Generalisation

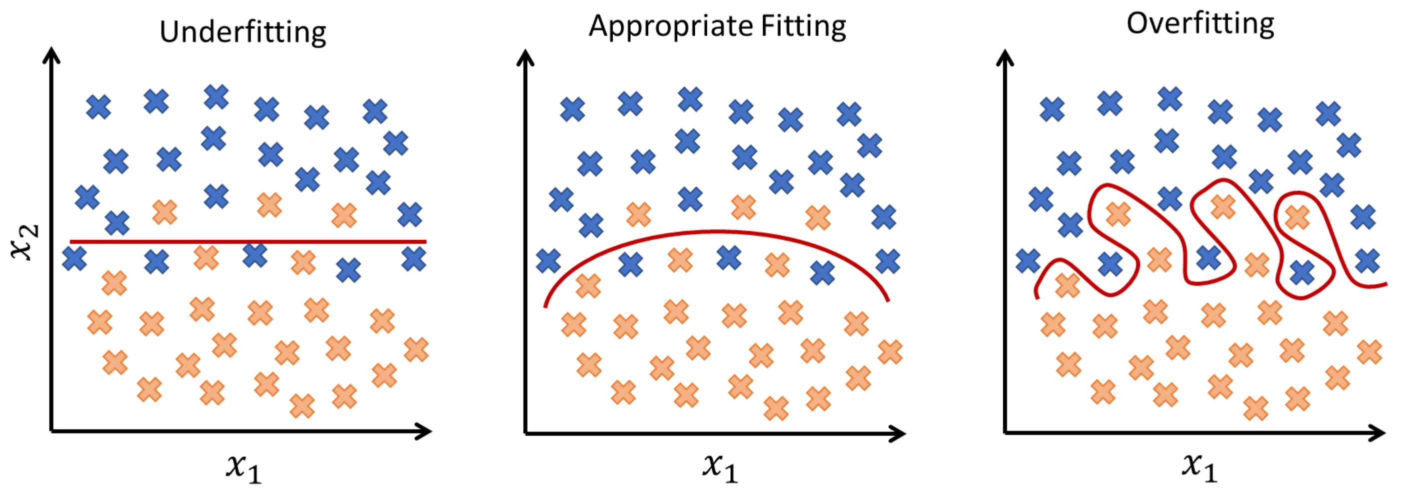
- **Training phase** – *minimise training errors*



- **Test phase** – *minimise generalisation errors*

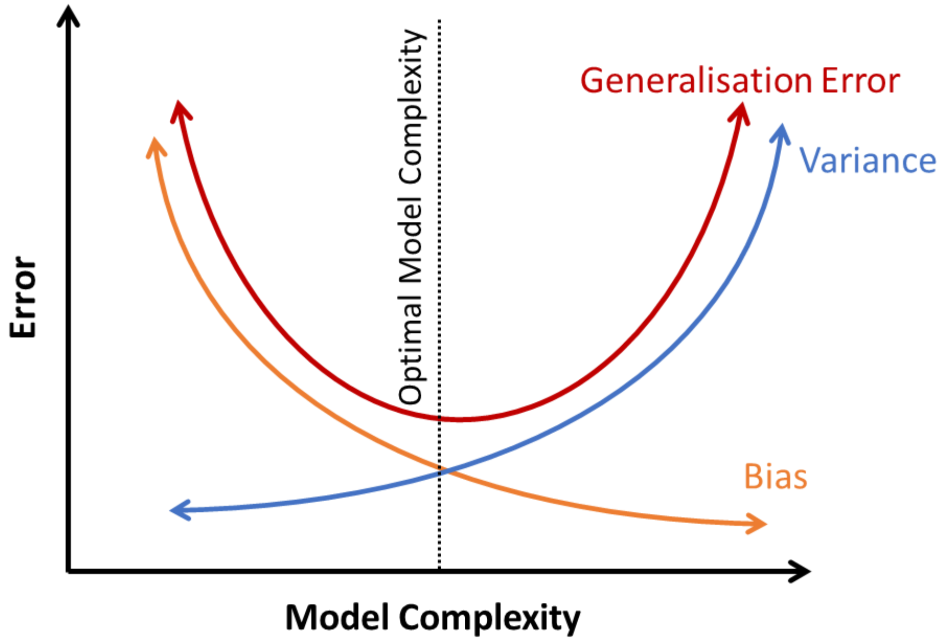


- **Generalisation Errors**
  - **Underfitting** – the model is too simple
    - The model lacks sensitivity to the variation in data
  - **Overfitting** – the model is too complex
    - Model attempts to account for all the variation in the training data



Universität Augsburg University

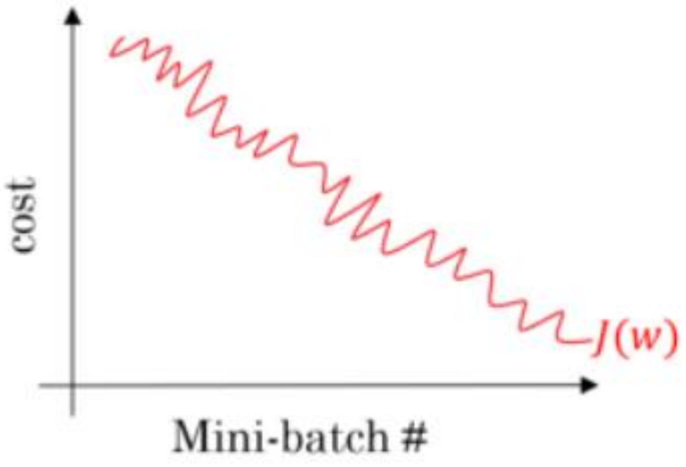
- **Minimising Generalisation Errors**
  - A trade-off of between bias and variance errors and the effect of model complexity
    - Increase in model complexity results in an initial decreases in generalisation error due to a decrease in model bias
    - As model becomes more complex generalisation errors increases due an increase in model variance



- **Mini-Batch Learning**
  - Performs an update for every batch of n training examples
  - Reduces noise in variance of weight updates

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J(\theta^{(t)}; x^{(i:i+n)}; y^{(i:i+n)})$$

  - Advantages
    - Stable convergence
    - Good learning speed
    - Good approximation minima location
  - Disadvantages
    - Total loss not accumulated



- **Weight Penalties**
  - Addition of a weight penalty term to the cost function

$$\tilde{J}(\theta; x, y) = J(\theta; x, y) + \alpha\Omega(\theta)$$

  - Large weights make networks unstable
    - Minor variation or statistical noise on the expected inputs will result in large differences in the output
  - Aim of penalty term is to encourage the model to map the inputs to the outputs of the training dataset in such a way that the weights of the model are kept small

- **Common penalty terms**
  - **L1 Norm**
    - The sum of the absolute values of the weights
    - L1 encourages weights to be zero if possible
    - Resulting in more sparse weights
      - Weights with more zeros values

$$\alpha\Omega(\theta) = ||\theta||_1$$
$$= \sum_n |\theta_n|$$

  - **L2 Norm**
    - The sum of the squared values of the weights
    - Penalizes larger weights

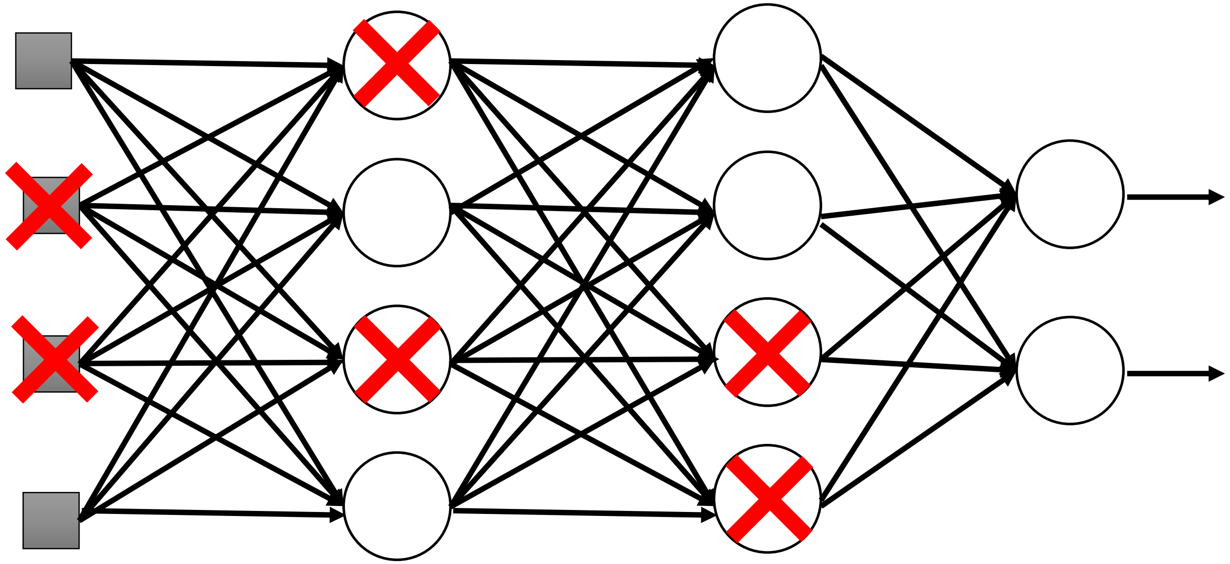
$$\alpha\Omega(\theta) = \frac{1}{2}||\theta||_2^2$$
$$= \sqrt{\sum_n |\theta_n|^2}$$

- **Data Augmentation**
  - The best way to make a machine learning model generalise better is to train it on more data
  - One way to get around this problem is to create fake data and add it to the training set
    - Trivial with images: shifts, flips, zooms, rotations
    - Adding noise is a form of augmentation
  - One must be careful not to apply transformations that would change the correct class.
    - Flips and rotations not useful when recognising the difference between “b” and “d” or the difference between “6” and “9”,

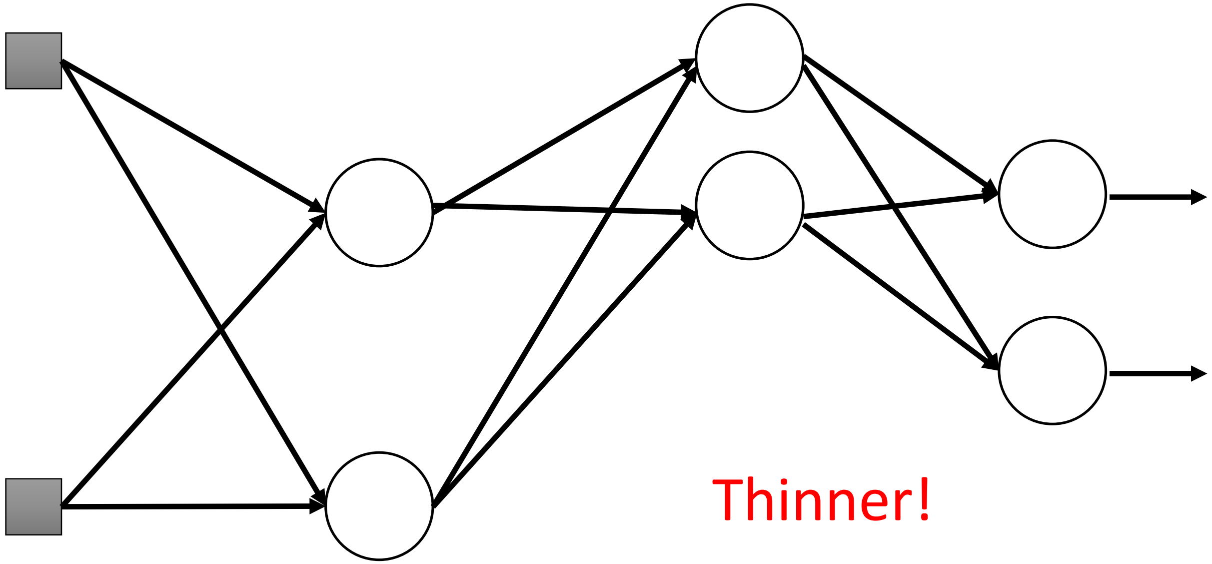
- **Training with noise**
  - **Adding noise means that the network is less able to memorise training samples**
    - Input data is changing all of the time
  - Results in smaller network weights and a more robust network that has lower generalisation error
  - Typical to add noise to input data
    - White Gaussian noise with mean of 0 and a standard deviation of 1
    - Generated as needed using a pseudorandom number generator.
    - We can also add noise activations, to weights, to the gradients and to the to the outputs

- **Dropout**
  - Method for training a ensemble of slightly different networks and averaging them
  - **Underlying concept:**
    - Although it's likely that large, unregularized neural networks will overfit to noise, it's unlikely they will overfit to the *same* noise
      - I.e. they will make slightly different mistakes
    - Averaging will cancel out the *differing* mistakes revealing what they all learned in common: *the signal* properties
  - The ensemble of subnetworks is formed by randomly removing nonoutput units during training

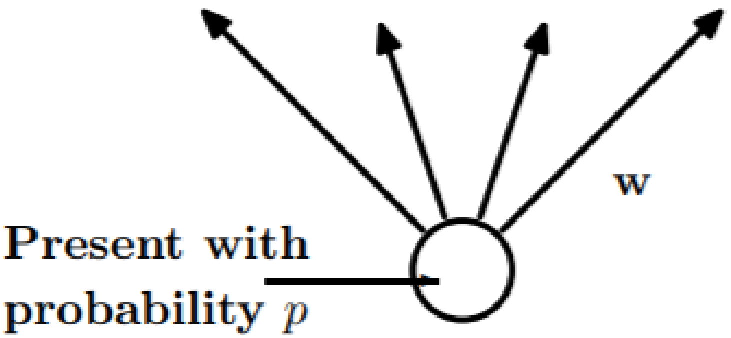
- **Training with Dropout**
  - Random neurons before updating the parameters
  - Each neuron has $p\%$ to dropout
    - p is a hyperparameter chosen before training



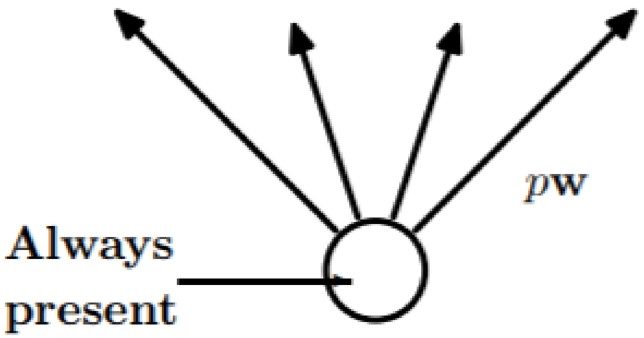
- **Training with Dropout**
  - **The structure of the network is changed**
    - Continue training with this new network
  - **For each mini-batch, we resample the dropout neurons**



- **Testing with Dropout**
  - Use a single neural without dropout. The weights of this network are scaled-down versions of the trained weights.
  - If a neuron is retained with probability p during training, the outgoing weights are multiplied by p at test time



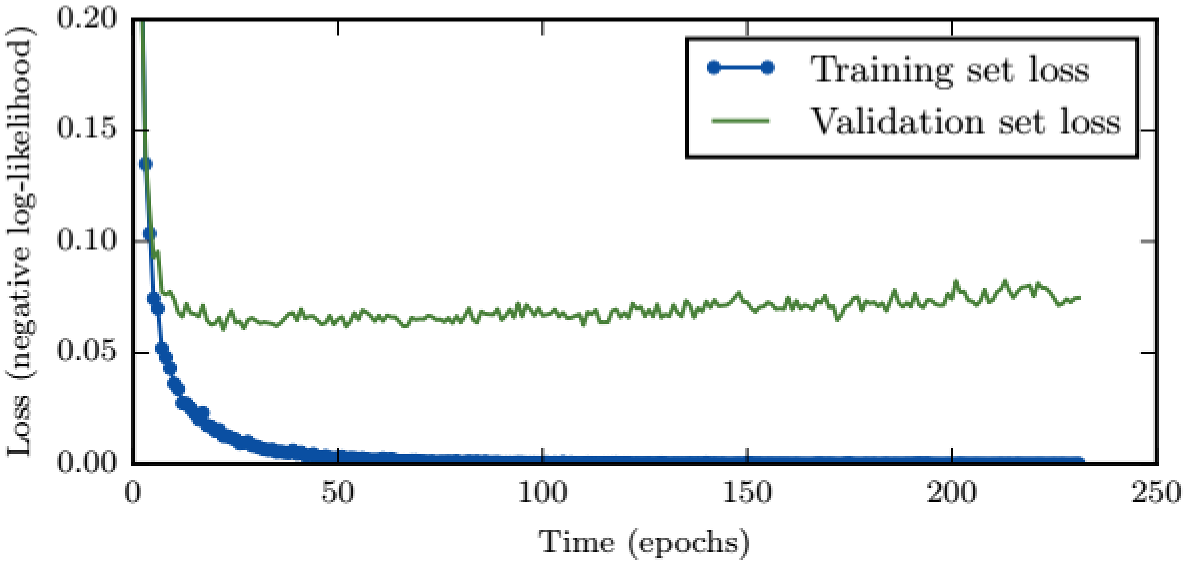
(a) At training time



(b) At test time

- **Early Stopping**
  - Neural networks can get worse if you train them too much
    - When training a large network, there will be a point during training when the model will stop learning the signal and start learning the statistical noise in the training dataset
  - Training a neural network long enough to learn the mapping, but not so long that it overfits the training data.
  - Trivial to monitor performance on a holdout validation dataset can be monitored during training
  - Stop training when generalization error increases

- **Early Stopping**
  - During training, the model is evaluated on a holdout validation dataset after each epoch
  - If the performance of the model on the validation dataset starts to degrade then the training process is stop

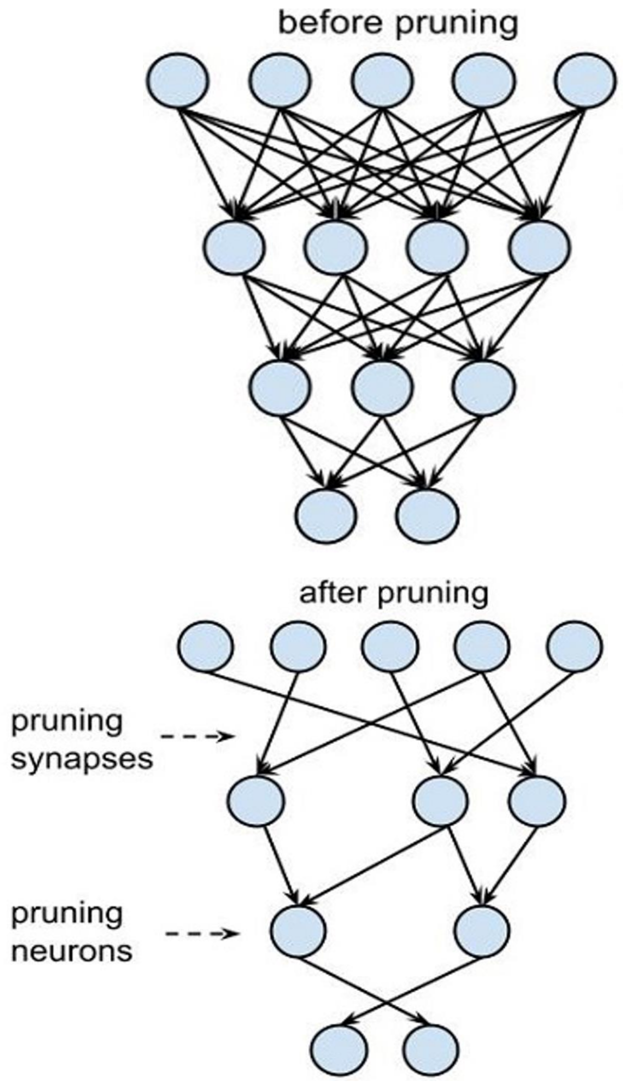


Pruning

- Induce sparsity in a network's connection matrices
- Possible to remove 75-90% of neurons from a network without significantly affecting performance

Core idea:

- Rank neurons according to how much they contribute
  - E.g., the L1/L2 norm of neuron weights
- Remove the low ranking neurons from the network
- This results a smaller and faster network



Quantization

- Reducing the number of bits that represent a number
  - Predominant numerical format used for deep learning is the 32-bit floating point
  - Weights and activations can be represented using 8-bit integers without incurring significant loss in accuracy
  - Possible to reduce precision further
    - Binary Neural Networks {-1,1}
    - Ternary neural networks {-1,0,1}
    - Quantize to powers of Two

-0.38	1.74	1.93
2.56	1.27	3.71
-0.95	-7.67	-0.86

2^0	2^1	2^1
2^1	2^1	2^2
-2^0	-2^3	-2^0

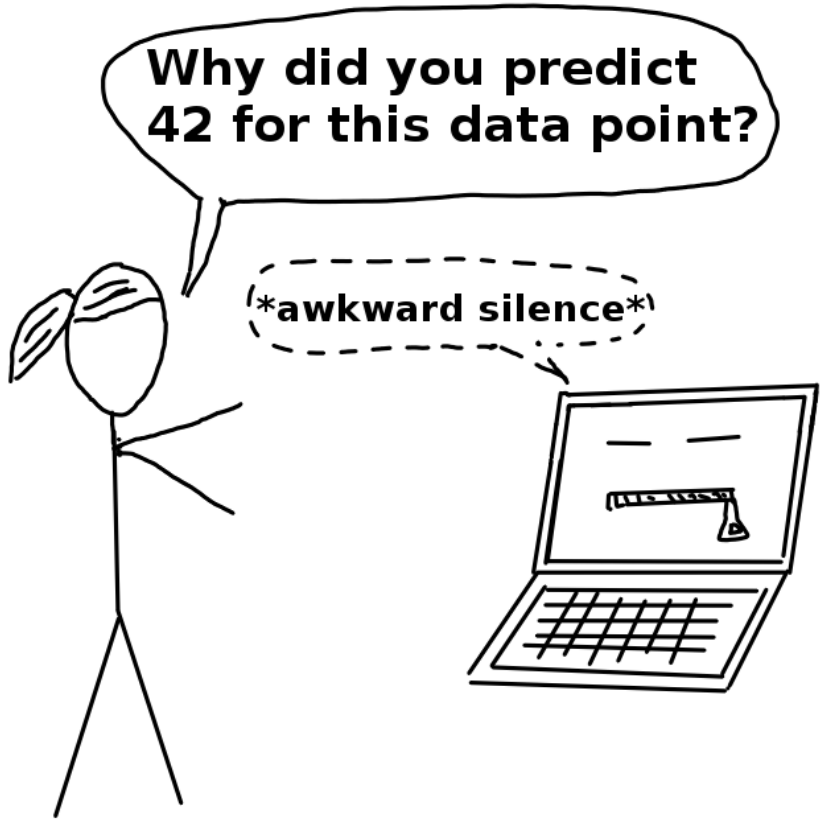
Overview

Image Source: https://distill.pub/2017/feature-visualization/

- Introduction
- Feed Forward Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Attention Mechanisms
- **Regularisation**
- Explainable AI

Objects (layers mixed4d & mixed4e)

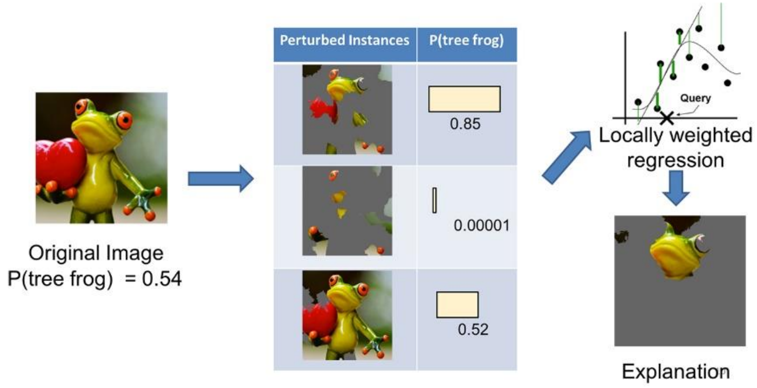
- **Interpretable Machine Learning**
  - Methods and models that make the behaviour and predictions of machine learning systems understandable to humans
- **Black-Box Systems**
  - Models that cannot be understood by looking at their parameters
  - These models are not interpretable

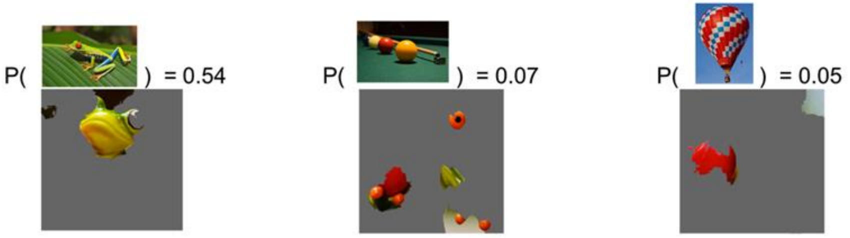


- **Importance of Interpretability**
  - **Provide complete descriptions**
    - A single-value evaluation metric is an incomplete description
  - **Aid problem formalisation**
    - Model must also explain how it came to the prediction
  - **Gain Knowledge**
    - The model becomes the source of knowledge as well as the data
  - **Easier Debugging and Auditing**
    - Important for health and safety, detect inherent biases in data

- **Taxonomy of Interpretability Methods**
  - **Intrinsic or post hoc?**
    - Restricting the complexity of the machine learning model (intrinsic)
    - Applying methods that analyse the model after training (post hoc)
  - **Model-specific or model-agnostic?**
    - Model-specific tools are limited to specific model classes.
    - Model-agnostic tools can be used on any machine learning model and are applied after the model has been trained
  - **Local or global?**
    - Does the interpretation method explain an individual prediction (local) or the entire model behaviour (global)

- **Local Interpretable Model-Agnostic Explanations (LIME)**
  - Trains local surrogate models to approximate the predictions of the underlying black box model
  - Key Steps
    - Trains your (black-box) model
    - Select instance to explain
    - Create perturbed dataset
    - Train a weighted, interpretable model, on perturbed dataset variations
    - Explain the prediction using model

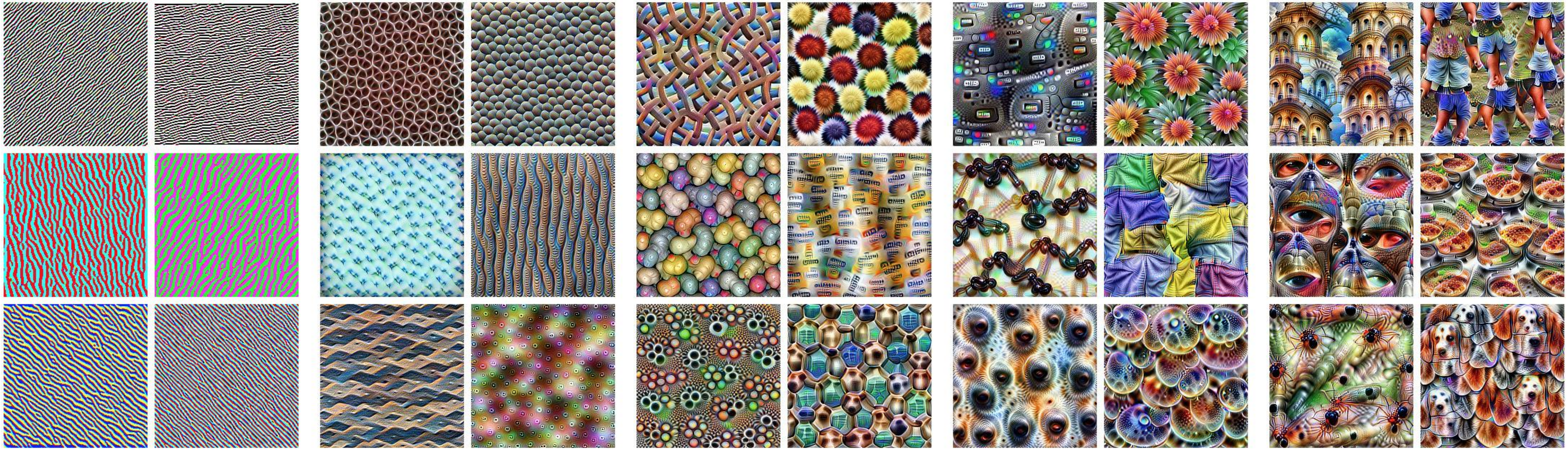




- **Deep Learning system are complex**
  - A single prediction can involve millions of mathematical operations, depending on the network architecture
  - Interpretation is virtually impossible
    - We would have to consider millions of weights that interact in a complex way to understand a prediction by a neural network
  - Standard explanation methods can work, however specific neural-network based approaches are advantageous
    - Access learnt information in hidden layers
    - Utilise the gradient of the network

Image Source: https://distill.pub/2017/feature-visualization/

- **Feature Visualization**
  - Making the inner works of neural networks interpretable



Universität Augsburg University

- **Feature Visualisation**
  - Approaches of making the learned features explicit
    - Finding the input that maximizes the activation of a unit
      - Unit: neurons, entire feature maps, entire (convolutional) layers
  - Feature visualisation is an optimisation problem
    - Assume that the weights of the neural network are fixed
      - I.e., the network is trained.
    - We are looking for a new image that maximizes the (mean) activation of a unit:

$$img^* = \arg\max_{img} \sum_{x,y} h_{n,x,y,z}(img)$$

Equation identifies mean activation of an entire channel z in layer n

- **Feature Visualisation**
  - Instead of maximizing the activation, you can also minimize the activation

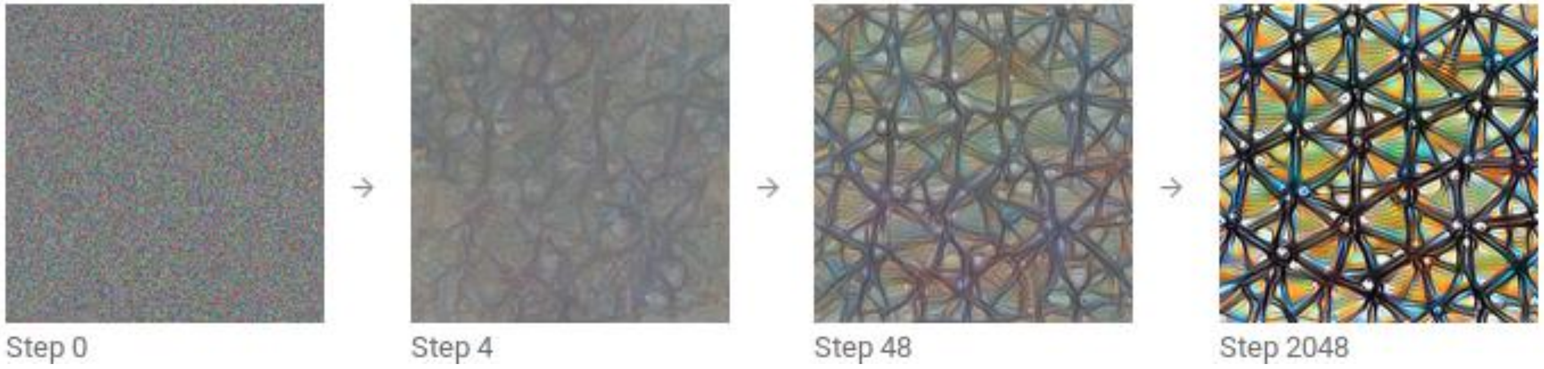
$$img^* = \arg\max_{img} \sum_{x,y} h_{n,x,y,z}(img)$$

$$img^* = \arg\min_{img} \sum_{x,y} h_{n,x,y,z}(img)$$

Activations from Inception V1 neuron 484 from layer mixed4d pre Relu

Image Source: https://distill.pub/2017/feature-visualization/

- **Key steps in Feature Visualisation**
  - – Start from random noise
  - – Place constraints on the update
    - Ensure that only small changes are allowed
  - – Apply steps to reduce noise in updates
    - Jittering, rotation or scaling to the image



- **Advantages of Visualisation**
  - Unique insights into the learning process of neural networks
  - Can be used to explain which pixels were important for the classification
- **Disadvantages of Visualisation**
  - Many feature visualization images are simply not interpretable
  - Illusion of explainability
    - They offer no real insight into the working of the neural network
    - The neurons interact in a highly complex manner, we still cannot infer these interactions from observing when certain neurons activate