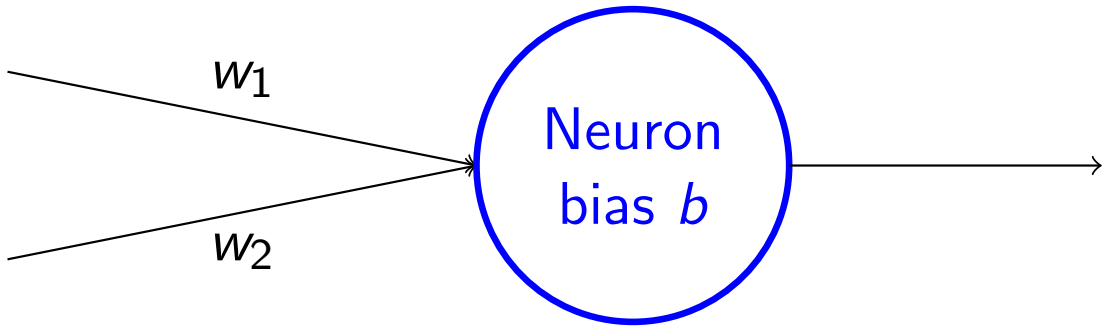
3. NN-based Classifiers

Neurons

- A neural network consists of neurons with internal weights that determine how its inputs are transformed into its outputs
- A simple linear neuron may look like:



- Its output o based on inputs x_1 and x_2 would be

$$o = w_1 x_1 + w_2 x_2 + b$$

- Usually a non-linear neuron activation function σ is used, such as sigmoid:

$$o = 1/(1 + e^{-(w_1 x_1 + w_2 x_2 + b)})$$

Neurons

- How do we choose w_1, w_2 and b?
- As usual, we want to optimize some objective, e.g. minimize cross-entropy loss $\mathcal{L}$. We find the gradient and take a step of a certain size in that direction.
- To decrement $\mathcal{L}$ by a certain size, we need to compute:

$$\frac{\partial \mathcal{L}}{\partial w_1} \qquad \frac{\partial \mathcal{L}}{\partial w_2} \qquad \frac{\partial \mathcal{L}}{\partial b}$$

- We need to do this efficiently for all neurons in all layers
  - The last layer is known as the **output layer**, which consists of neurons usually connected to an output-aggregating function such as softmax
  - The input layer has no neurons
  - The layers in between are known as **hidden layers**

Backpropagation

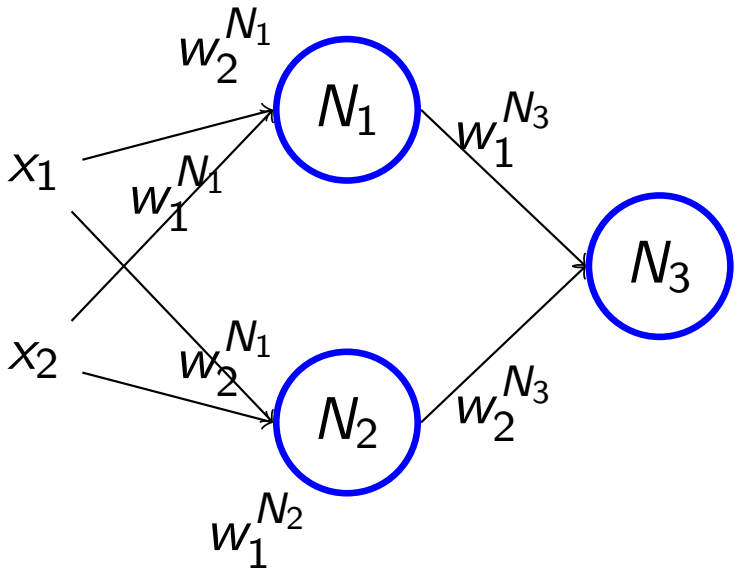
- **Backpropagation** is an efficient heuristic for finding derivatives on a neural network (or any complex computational graph)
- By computing derivatives backward from the output, we avoid any duplicate calculations with minimal storage
- Backpropagation is simply repeated application of the chain rule:

$$\frac{df(y(x))}{dx} = \frac{df}{dy} \cdot \frac{dy}{dx}$$

- What happens if a function is not differentiable? (eg. step function in original perceptrons)

Backpropagation

Working example: fully connected network with two layers, sizes 2 and 1 respectively, with activation σ, MSE loss; input has two variables x_1, x_2



Backpropagation

For N_3:

$$
\begin{aligned}
m &= w_1^{N_3} y^{N_1} + w_2^{N_3} y^{N_2} \\
y^{N_3} &= \sigma(m) \\
\mathcal{L} &= (C(x) - y^{N_3})^2
\end{aligned}
$$

With direct substitutions:

$$
\begin{aligned}
\frac{\partial L}{\partial w_1^{N_3}} &= \frac{\partial}{\partial w_1^{N_3}} (C(x) - y^{N_3})^2 \\
&= 2(C(x) - y^{N_3}) \frac{\partial}{\partial w_1^{N_3}} (C(x) - y^{N_3}) \\
&= 2(C(x) - y^{N_3}) \frac{\partial}{\partial w_1^{N_3}} (C(x) - \sigma(m)) \\
&= -2(C(x) - y^{N_3}) \sigma'(m) \frac{\partial m}{\partial w_1^{N_3}} \\
&= -2(C(x) - y^{N_3}) \sigma'(m) y^{N_1}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial w_2^{N_3}} &= \frac{\partial}{\partial w_2^{N_3}} (C(x) - y^{N_3})^2 \\
&= 2(C(x) - y^{N_3}) \frac{\partial}{\partial w_2^{N_3}} (C(x) - y^{N_3}) \\
&= 2(C(x) - y^{N_3}) \frac{\partial}{\partial w_2^{N_3}} (C(x) - \sigma(m)) \\
&= -2(C(x) - y^{N_3}) \sigma'(m) \frac{\partial m}{\partial w_2^{N_3}} \\
&= -2(C(x) - y^{N_3}) \sigma'(m) y^{N_2}
\end{aligned}
$$

Backpropagation

For N_3:

$$
\begin{aligned}
m &= w_1^{N_3} y^{N_1} + w_2^{N_3} y^{N_2} \\
y^{N_3} &= \sigma(m) \\
\mathcal{L} &= (C(x) - y^{N_3})^2
\end{aligned}
$$

Using backpropagation:

$$
\begin{aligned}
\frac{\partial L}{\partial y^{N_3}} &= -2(C(x) - y^{N_3}) \\
\frac{\partial L}{\partial m} &= \frac{\partial L}{\partial y^{N_3}} \sigma'(m) \\
\frac{\partial L}{\partial w_1^{N_3}} &= \frac{\partial L}{\partial m} y^{N_1} \\
\frac{\partial L}{\partial w_2^{N_3}} &= \frac{\partial L}{\partial m} y^{N_2}
\end{aligned}
$$

By just storing the intermediaries, we can reduce duplicate computations significantly!

Backpropagation

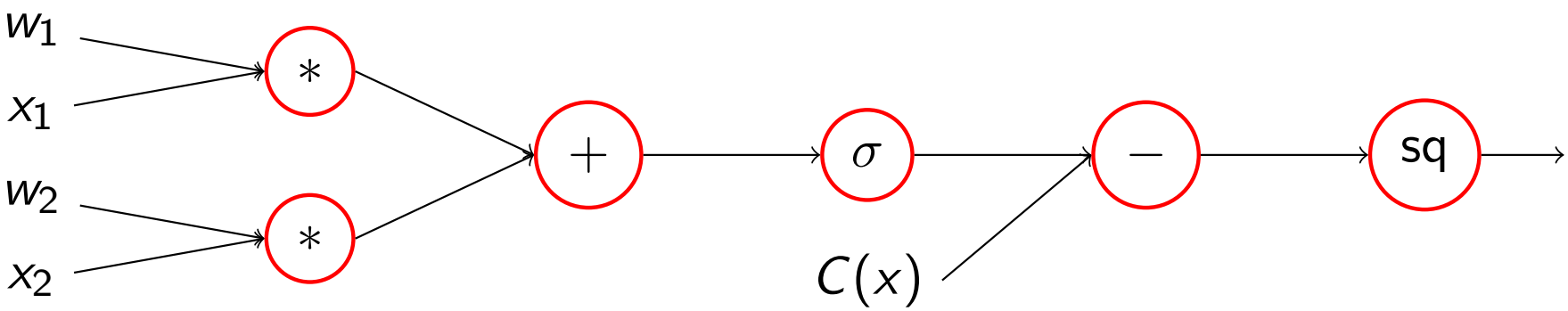
Extending it backwards to N_1:

$$\frac{\partial L}{\partial y^{N_1}} = \frac{\partial L}{\partial m} w_1^{N_3}$$
$$\frac{\partial L}{\partial m_1} = \frac{\partial L}{\partial y^{N_1}} \sigma'(m_1)$$
$$\frac{\partial L}{\partial w_1^{N_1}} = \frac{\partial L}{\partial m_1} x_1$$

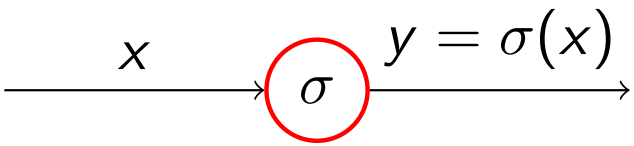
Extensions to other weights work the same way

Backpropagation

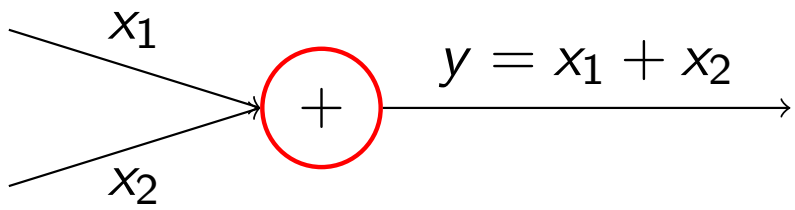
The above could be expressed graphically with a computation graph (eg N_3):



This makes it easy to understand how a derivative travels across the network by expressing it on each node, eg:



$$\frac{\partial L}{\partial x} = \sigma'(x)\frac{\partial L}{\partial y}$$



$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial y}$$

Backpropagation

- Intuitive understanding of backpropagation helps you design good neural networks
  - Add gates duplicate derivatives
  - Max gates send the derivative to the higher value and 0 to the other
- We want to avoid **vanishing/exploding gradients**: gradients become too small to allow for learning, or becoming so large that they stop other nodes from learning
  - Specifically, gradients can vanish as they travel backwards to the first layer
- Recommended: Andrej Karpathy's video "CS231n Winter 2016: Lecture 4: Backpropagation, Neural Networks 1"

Vanishing gradients

Example

Consider the sigmoid function, which has derivative

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

- The sigmoid function outputs $\sigma(x)$ from 0 to 1, so the maximum value of the above is 0.25
- Any gradient that passes backwards through a sigmoid gate will be reduced to at most $1/4$ of its original value
- Layers of sigmoid-activated neurons will cause gradients to vanish:

$$\frac{\partial L}{\partial x_1} = \sigma'(x)\frac{\partial L}{\partial x_2} \implies \frac{\partial L}{\partial x_1} = (\sigma'(x))^n \frac{\partial L}{\partial x_n}$$

- Similarly, gradients > 1 will explode

Activation functions

- Sigmoid activation was dominant until ReLU (around 2013):

$$ReLU(x) = \begin{cases} x \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

- No vanishing gradients
- But ReLU causes the **Dying ReLU problem**: it stops transferring derivatives when inputs becomes less than 0, so its inputs are less likely to change, and it cannot escape
  - High learning rates exacerbate the problem
- Alternative activation functions include:
  - Leaky ReLU: Instead of 0, outputs αx for tiny α when $x < 0$
  - Swish: $x \cdot sigmoid(x)$ (eg. GPT-4)
  - GELU (eg. Llama)

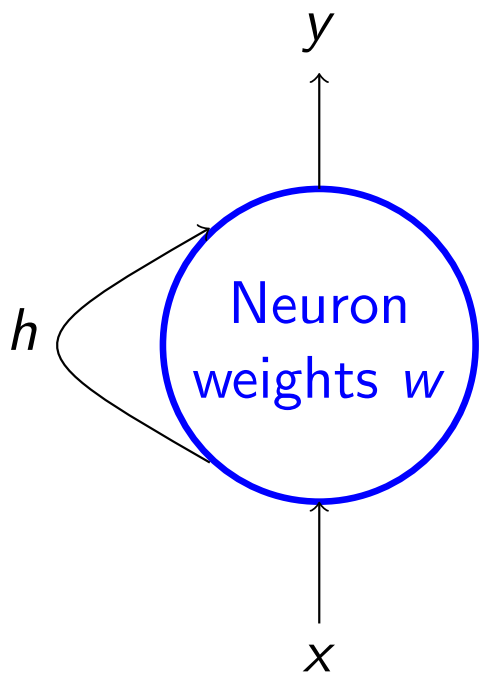
Training

- Training proceeds in **epochs**: in one epoch, every training element is used once
  - We usually shuffle the training data at the start of each epoch to prevent cycles
- We update the gradients after examining a number of training elements equal to the **batch size** (SGD: batch size $= 1$)
- After each epoch, we perform **validation** against the validation set and save the trained model, and revert back to the best model if performance starts to decrease

Recurrent Neural Networks

- Basic multi-layer perception has the (severe) limitation that the input and output size must be predictable (size of input/output layers)
- In addition, one output is made per input, which is not suitable for text (consider e.g. translation, speech-to-text)
- Recurrent neural networks create outputs and adjust hidden state (a sort of memory) as each input token is passed

Recurrent Neural Networks

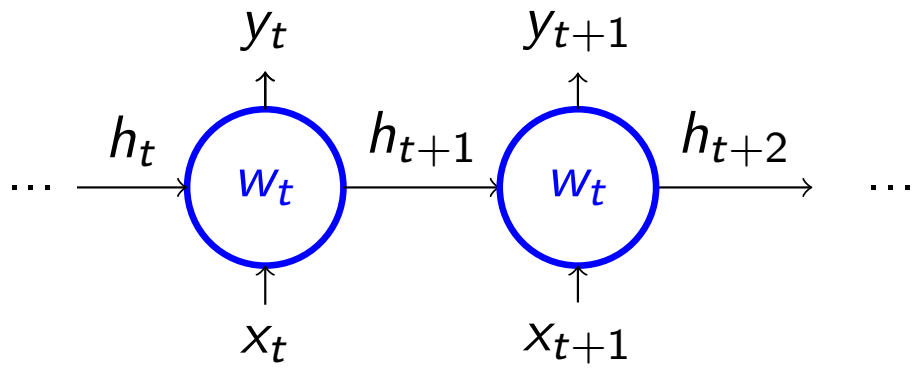


- At time t, input x_t is used to update hidden state h_t and output y_t as follows:

$$h_t = \sigma_h(W_h \cdot (h_{t-1}||x_t))$$
$$y_t = \sigma_y(W_y \cdot (h_{t-1}||x_t))$$

Recurrent Neural Networks

- We can unwrap a RNN through time so that it can be conceived as a layered neural network:



- We can see that backpropagation can still be done **through time (BPTT)**
- And similarly to before, we suffer vanishing or exploding gradients through time so that the first inputs are forgotten or dominant

$$\frac{\partial L}{\partial h_{t-1}} = \sigma_h'(W_h \cdot (h_{t-1}||x_t)) \cdot W_h \cdot \frac{\partial L}{\partial h_t}$$

Constant Error Carousel

We want a RNN unit with state c_t that backpropagates gradients with a multiplier of 1, i.e.

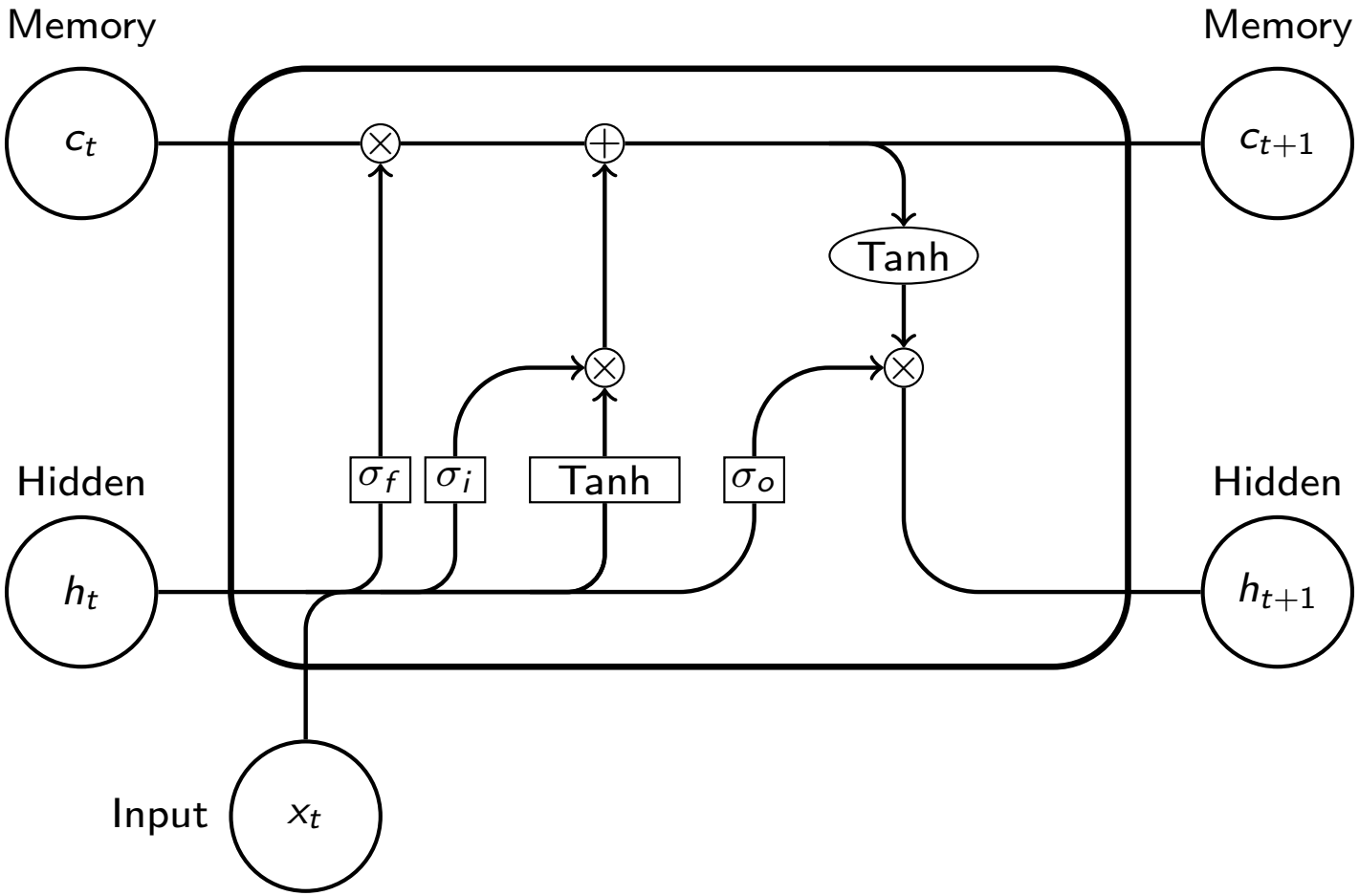
$$\frac{\partial L}{\partial c_{t-1}} = \frac{\partial L}{\partial c_t}$$

This requires

$$\sigma_c(W_c \cdot (c_{t-1}||x_t)) = \frac{W_c \cdot (c_{t-1}||x_t)}{W_c}$$

This can be achieved with identity function σ_c with $W_c = 1$

Long Short-Term Memory [LSTM97]



Tikz Credit: J Leon V on StackOverflow

Long Short-Term Memory

The σ functions are chosen to be the same, but marked differently for explanation:

- σ_f is the **forget** gate. It reads the hidden and input values to decide what part of memory c_t to retain. For example, if a value is 0, then that element of the memory is wiped.
- σ_i is the **input** gate. It decides what elements to add to the memory and also affects the hidden state.
- σ_o is the **output** gate, which directly affects the output.

Note that the memory unit is a constant error carousel

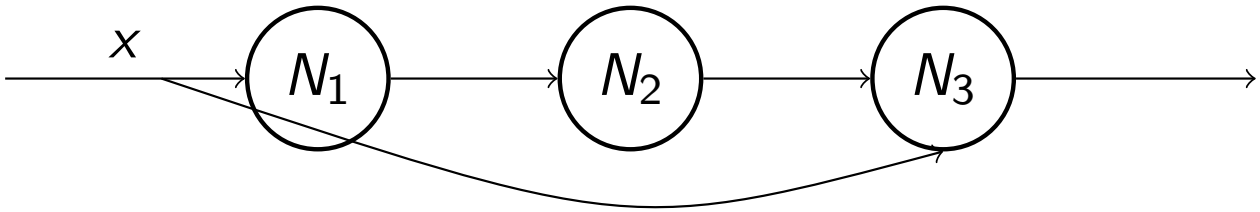
Long Short-Term Memory

Limitations:

- The total memory of a cell is the length of its memory state c_t, but it may be too small (insufficient to store information) or too large (hard to train)
  - In particular, very long sequences interact poorly with LSTMs
- There is no way to directly model long-distance relationships

Skip connections

Also known as residual connections:



$$N_3 = w_{2,3} N_2 + x$$

- Presented as neurons for simplicity above, but residual connections would skip layers
- This can also propagate constant gradients
- First used for CNNs [ResNet16]; similar concept in RNNs with a parametrized skip connection in 2015
  - ImageNet had rising error past 20 layers, but ResNet still had decreasing error at 152 layers

Batch normalization [Batch15]

A batch normalization layer takes in d inputs x_i and outputs d values y_i with normalized mean and variance:

$$\mu = \sum x_i/d \qquad \sigma^2 = \sum (x_i - \mu)^2/d$$

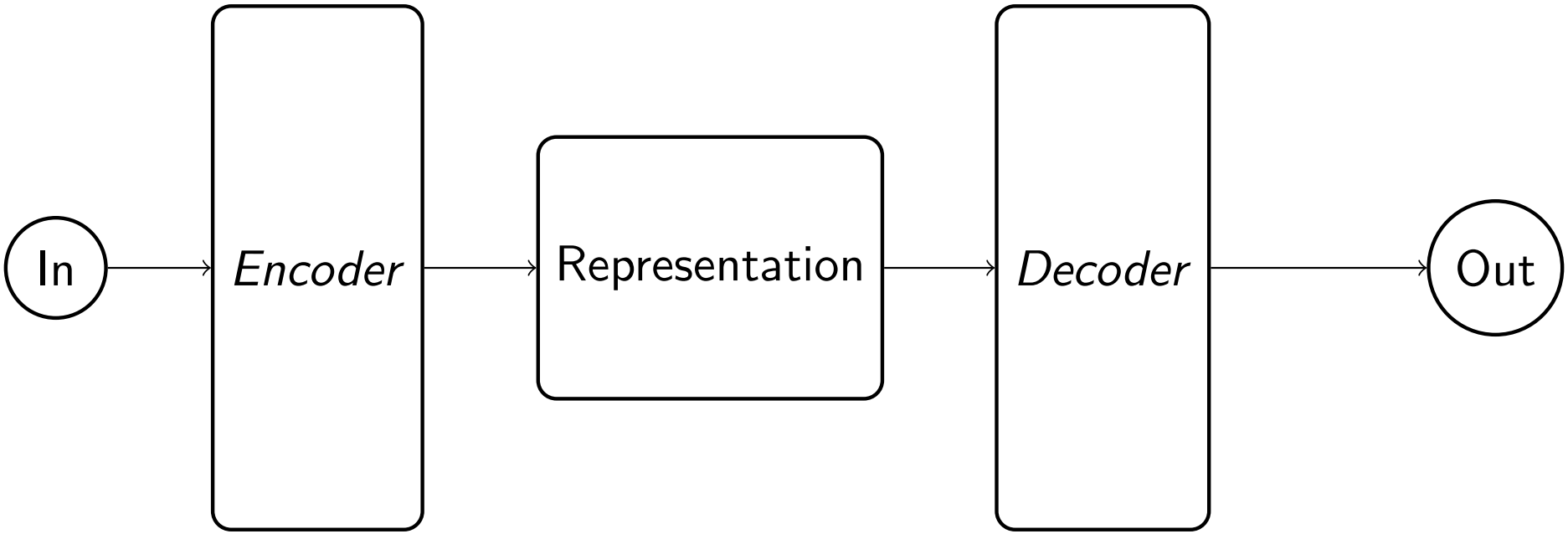
$$y_i = (x_i - \mu)/\sigma$$

Original proposal suggested that it would reduce the **internal covariate shift**:

- Recall that covariate shift describes the change in probability distribution of features between two different data sets
- Internal covariate shift describes the change in distribution of values for a layer interrupts learning
- For example, if input values to a layer continuously increase during learning; if a layer is thought to be learning a distribution, it will be off continuously

Encoder-decoder architecture

We can think of translating as: 1) encoding text in language A into a byte-representation of meaning, and 2) decoding meaning into text in language B



- Encoder and decoder are layers of neurons
- Originally called seq2seq for sequence processing
- Extended far beyond translation with attention mechanisms

Attention

- Original motivation: in different languages, words may be ordered differently; can we train a translator to understand alignment?
  - the red book $\rightarrow$ le livre rouge
- We can explicitly construct an alignment model to be jointly trained [Bahd2014]:
  - An alignment matrix is defined between all input and output words

	le	livre	rouge
the			
red			
book			

  - Combined with bidirectional RNNs (on the encoder) to capture forward and backward hidden states

Attention

- In Bahdanau, decoder is trained to predict a word y_i given the context c_i, the previous predicted word y_{i-1}, and the current hidden state of the decoder s_i
- The context c_i is different for each output word and unrelated to c_{i-1}; in the encoder-decoder architecture, c is a vector of the encoder hidden states $\{h_1, ..., h_{T_x}\}$
- The context is computed as:

$$c_i = \sum_{j=1}^{h_{T_x}} \alpha_{ij} h_j$$

where weight α_{ij} is computed with a jointly trained alignment model, describing how well input i relates to output position j.

Attention

Alternative view: Attention maps queries and key-value pairs to an output

$$Attention(q, k, v) = \sum_j a(q, k_j) v_j$$

For example, in the Bahdanau attention mechanism:

- The **query** is the current decoder output state, asking for the next word
- The **keys** are the encoder hidden states at j, capturing information around it
- $a(q, k_j)$ describes how well the encoder output state k_j matches the current decoder output state q
- The **values** are also the encoder hidden states, conveying that information to the decoder based on the weight a

Attention

The Transformer architecture, behind foundational models, uses **self-attention** with multi-headed attention

- Unlike Bahdanau, where decoders attend to encoders, in self-attention each input token to an attention layer attends to all other tokens (and itself)
- In multi-headed attention, each input token is duplicated across d dimensions so that each duplicate head can learn a different aspect of it
- Transformers do not use recurrent units; but they also have skip connections, softmax and layer normalization

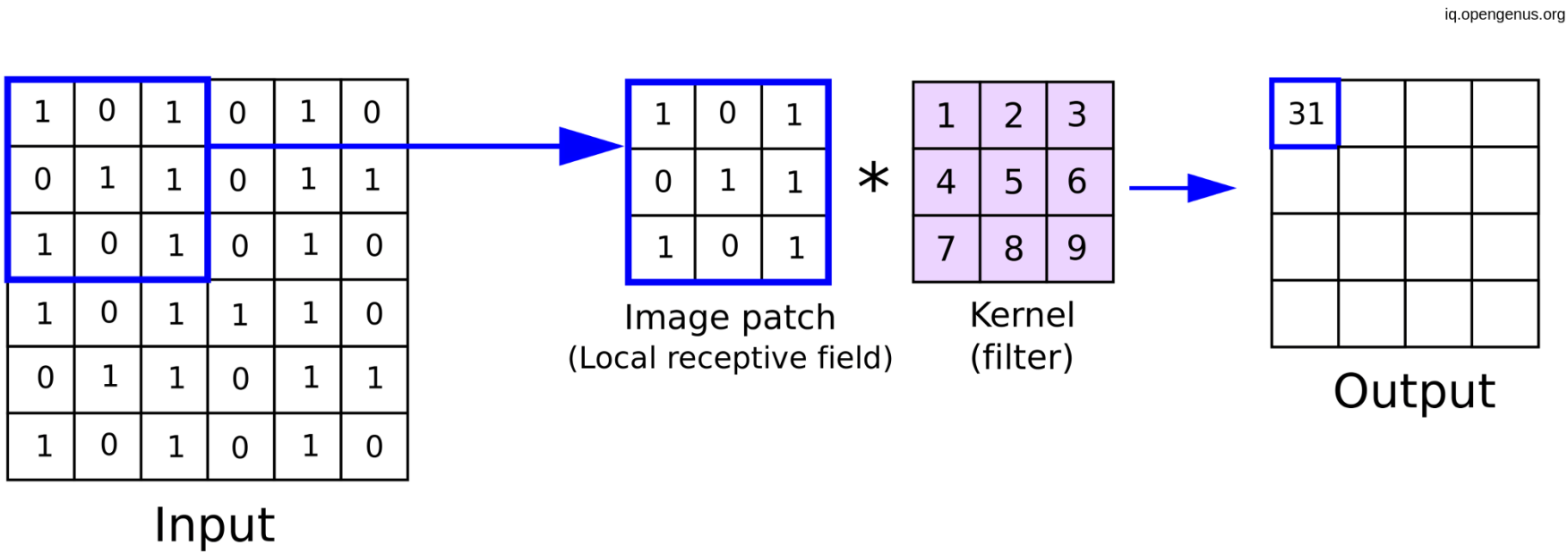
Generative Adversarial Networks

Generative model especially useful for unlabeled mimicry

- Consists of two components, a generator and a discriminator, and a data set of real samples
- The generator generates samples that are similar to the real data set
- The discriminator determines if samples are synthetic (generated by the generator) or real
- Each side is trained to beat the other
  - Samples on which the discriminator did well are used to move the gradients of the generator; samples on which the discriminator did poorly are used to move the gradients of the discriminator

Convolutional Neural Networks

- Neural network that uses **convolution units** with a kernel:



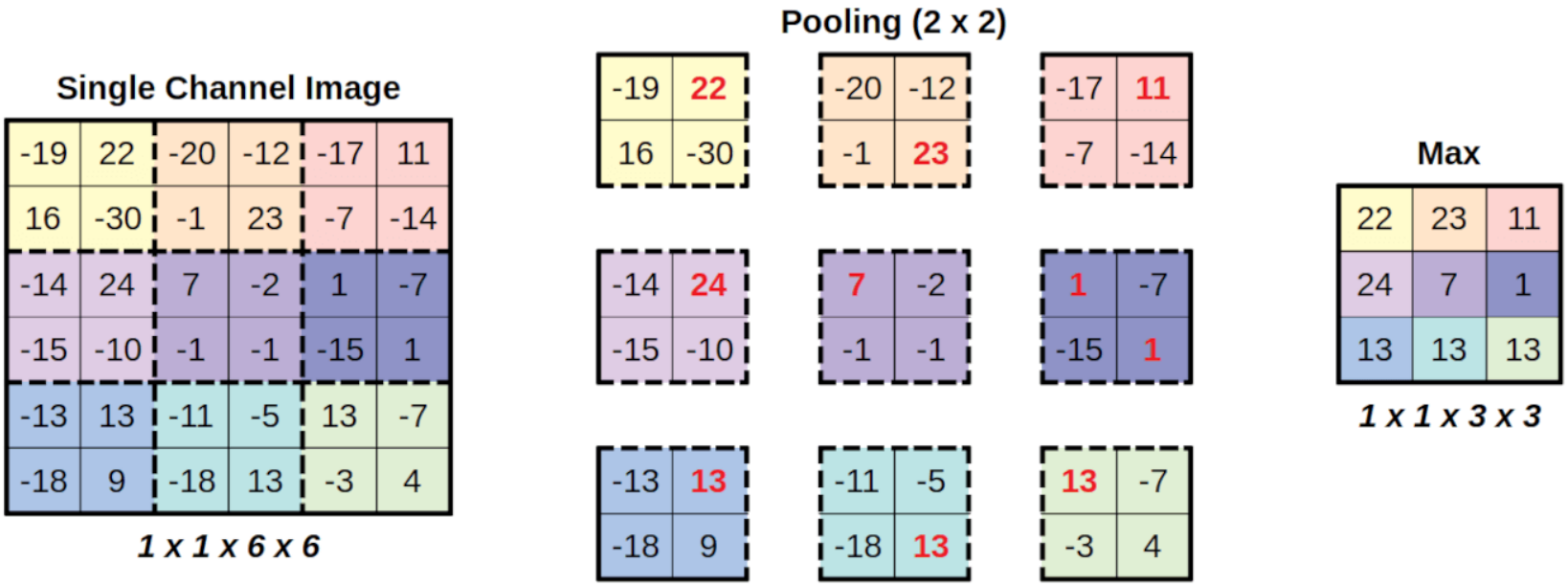
- Originally designed for image classification
- Sometimes dominant even for text/sequence problems

Convolutional Neural Networks

- A **kernel** is a 2D matrix of a specified size that is applied to the input with element-by-element multiplication then sum to produce an output
  - Kernel weights are learned through backpropagation
- After each computation, the kernel is moved rightwards/downwards by a number of pixels equal to the **stride**
- Since we lose pixels on the edges after each convolutional layers, we can pad the image to restore pixels (of value 0) to the edges
- Image data can be spread through multiple channels of the same size; a kernel takes any number of channels as input but always outputs to one channel
  - If we want k output channels, we have k different kernels

Convolutional Neural Networks

- Excessive sensitivity to certain pixels impedes training
- A **pooling** layer of size $x \times y$ reduces the dimensions by x and y times respectively
  - Max pooling takes the max value of each pool and discards the reset



Reading a neural network

Let us compare AlexNet (2012) and LeNet (1989)