Deep Learning and Natural Language Processing - Introduction

Vincenzo Scotti,
Ph.D. student
vincenzo.scotti@polimi.it

arcslab
adaptable, relational and cognitive software environments

N.L.P. – A.Y. 20-21

Contents

This symbol means that the formulae in the slide aren't required to be known

MACHINE LEARNING

Machine learning

- **Machine learning** is a sub-field of **Artificial Intelligence** (AI)
- Proposes techniques and frameworks to extract knowledge from data
- Data-driven techniques, in opposition to model- or expert-driven
- **Deep learning** is a particular approach combining **artificial neural networks** (a machine learning framework) with **representation learning**

Learning paradigms

Three learning paradigms:

- **Supervised** Learning: given a set of inputs $\mathbf{x}_i$ and the corresponding targets $\mathbf{t}_i$, learn to produce the correct output on new inputs
- **Unsupervised** Learning: given a set of inputs $\mathbf{x}_i$, learn to exploit regularities in the inputs to build a new representation of them
- **Reinforcement** Learning: given a set of actions $\mathbf{a}_i$ to perform in an environment and the corresponding rewards $\mathbf{r}_i$, learn how to maximize cumulative reward through the actions

The focus will be on *supervised* learning and (partially) *unsupervised*

Supervised learning problems

- Classification:
  - Given an input sample $\mathbf{x}$ (described by some features x_i so that $\mathbf{x} = < x_1, \ldots, x_i, \ldots, x_n >$
  - Given a finite set of k classes $\mathscr{C}$
  - Assert to which of the k classes $\mathbf{x}$ belongs to
- Regression:
  - Given an input sample $\mathbf{x}$ (described by some features x_i so that $\mathbf{x} = < x_1, \ldots, x_i, \ldots, x_n >$
  - Predict a continuous value $\mathbf{y} \in \mathbb{R}^d$

NEURAL NETWORKS

Motivations

- Most current machine learning works well because of human-designed representations and input features
  - Machine learning becomes just optimizing weights to best make a final prediction
- The representation learning approach attempts to automatically learn good features (representations)
  - Neural Networks can be used to perform feature learning
  - Learn a representation of their input at the hidden layer(s)

History

- 1943: Warren Mc Cullog and Walter Harry Pitts opened the subject by creating a computational model for neural networks
- 1958: Frank Rosenblatt created the Perceptron (classification model)
- 1960: Bernard Widrow and Ted Hoff created an ADAptive LInear NEuron or ADALINE (regression model).

History

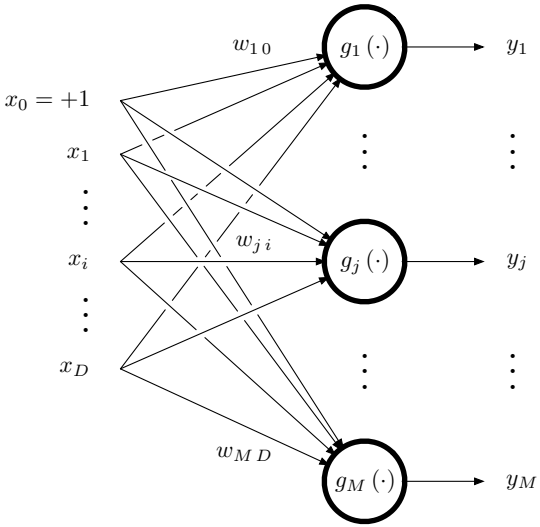
- 1969: Marvin Minsky and Seymour Papert proved that basic perceptron fails in front of XOR problem
- 1982: Hopfield proposes a solution to generalize Multi-Layer Perceptron
- 1986: David Rumelhart, Hinton and Williams propose the gradient descent algorithm

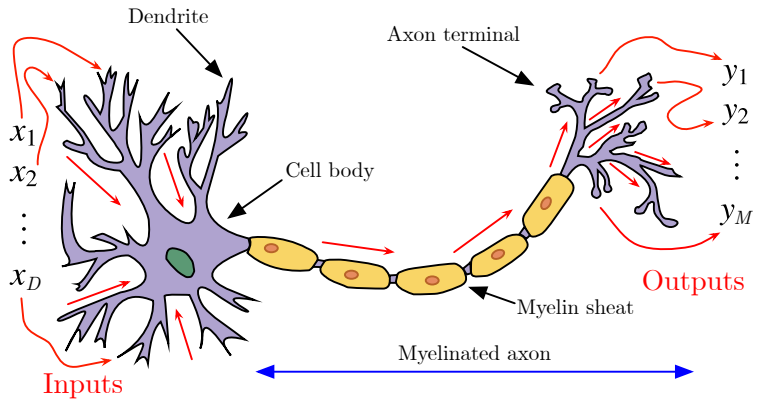
Artificial neuron

Vaguely inspired by the biological neuron

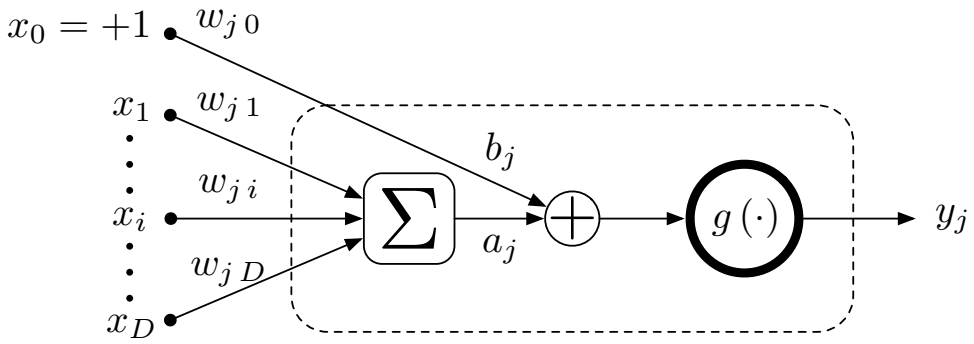
Information is transmitted through chemical mechanisms:

- Dendrites collect **inhibitory** and **excitatory** charges from synapses
- Accumulated charge is released if a threshold is passed (i.e. the **neuron fires**)





Artificial neuron (structure)



Neuron j

- Inputs x_i (with $i = 1,\ldots,d$)
- Output y_j

- w_{ji} (with $i = 0,\ldots,d$) are the **synaptic weights** (or weights)
- a_j is the **activation value** (or propagation function)
- b_j is the activation threshold (or **bias**)
- $g(\cdot)$ is the **activation function**

Artificial neuron (formulae)

- Activation value: $a_j = \sum_{i=1}^{d} w_{ji} \cdot x_i$
- Bias: $b_j = w_{j0} \cdot x_0$
- Output: $y_j = g\left(a_j + b_j\right) = g\left(\sum_{i=1}^{d} w_{ji}x_i + w_{j0}x_0\right) = g\left(\sum_{i=0}^{d} w_{ji}x_i\right)$
- $\mathbf{x} = \begin{bmatrix} +1 & x_1 & \dots & x_d \end{bmatrix}^\top$ column vector of inputs with $\mathbf{x} \in \mathbb{R}^{d+1}$
- $\mathbf{w}_j = \begin{bmatrix} w_0 & w_{j1} & \cdots & w_{jd} \end{bmatrix}^\top$ column vector of weights with $\mathbf{w} \in \mathbb{R}^{d+1}$
- $y_j = \mathbf{w}_j^\top \cdot \mathbf{x}$

Activation functions

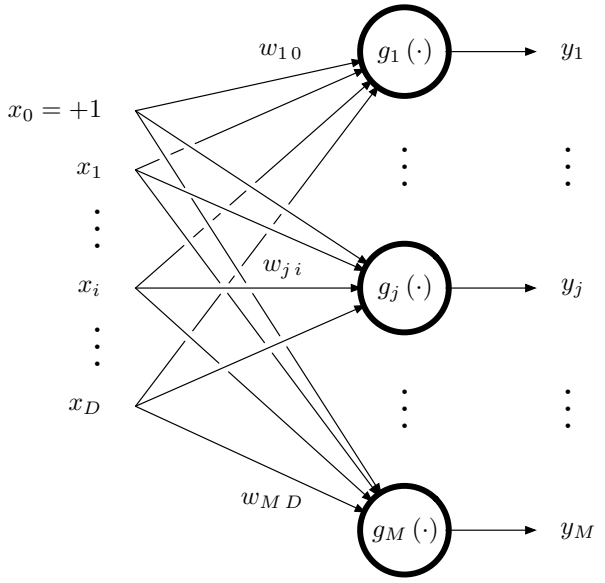
- Linear: $g(x) = x$
- Sign: $g(x) = \begin{cases} +1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$
- Sigmoid: $g(x) = \sigma(x) = \dfrac{1}{1 + \exp(-x)}$
- Hyperbolic tangent: $g(x) = \tanh(x) = \dfrac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$

Note that $\exp(x) = e^x$

Artificial neurons layer

- Also called **dense** or **fully-connected** layer
- Each output is computed as for the single output case
- The input is the same, what changes are the weights

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} g\left(\mathbf{w}_1^\mathsf{T} \cdot \mathbf{x}\right) \\ g\left(\mathbf{w}_2^\mathsf{T} \cdot \mathbf{x}\right) \\ \vdots \\ g\left(\mathbf{w}_m^\mathsf{T} \cdot \mathbf{x}\right) \end{bmatrix}$$



Artificial neurons layer (formulae)

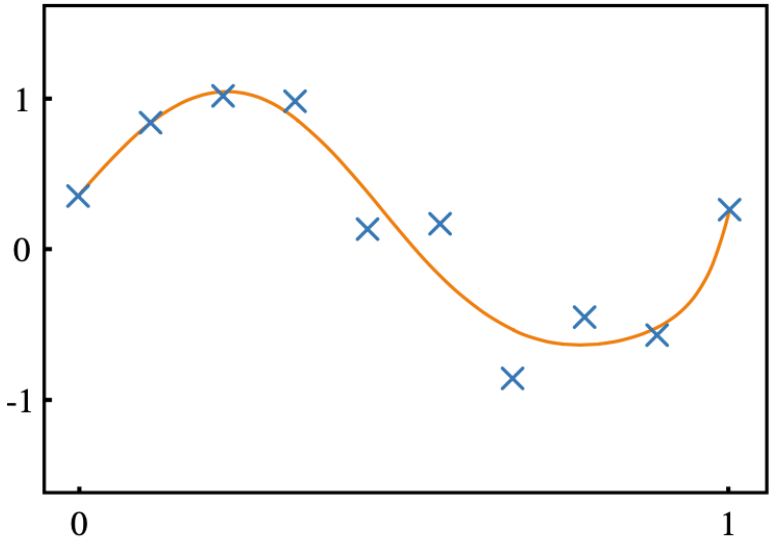
- Matrix notation:
- $\mathbf{x}$ column vector of inputs with $\mathbf{x} \in \mathbb{R}^{d+1}$
  $\mathbf{x} = \begin{bmatrix} +1 & x_1 & \dots & x_d \end{bmatrix}^\top$
- $\mathbf{W}$ matrix of weights with $\mathbf{W} \in \mathbb{R}^{m \times (d+1)}$

$$\mathbf{W} = [\mathbf{w}_1 \quad \cdots \quad \mathbf{w}_m] = \begin{bmatrix} w_{10} & w_{20} & \cdots & w_{m0} \\ w_{11} & w_{21} & \cdots & w_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1d} & w_{2d} & \cdots & w_{md} \end{bmatrix}$$

- $g(\cdot)$ is applied element-wise
  $\mathbf{y} = g\left(\mathbf{W}^\top \cdot \mathbf{x}\right)$

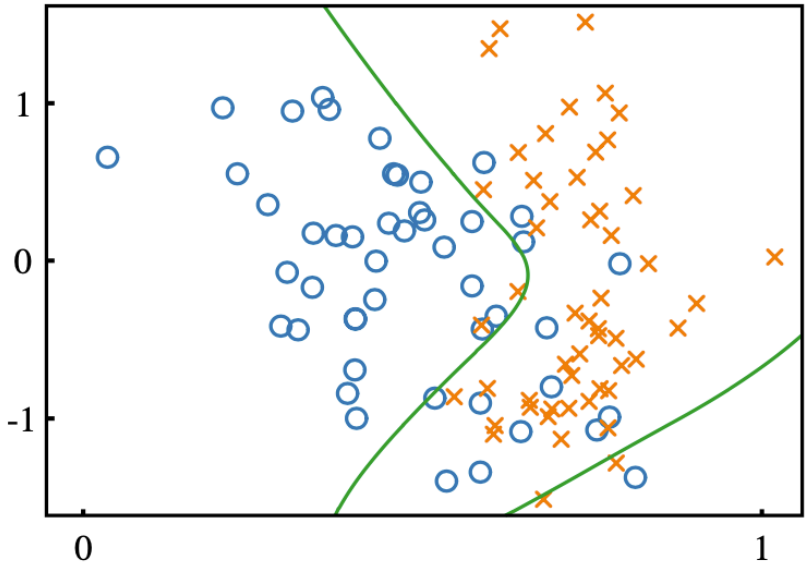
Artificial neural networks for regression

- The regression case is simpler with respect to the classification one
- The output activation is simply a linear function
- The network produces a continuous output over $\mathbb{R}^d$



Artificial neural networks for classification

- Neural networks can be used as a classifier
- They can learn to discriminate among $K \geq 2$ classes



Artificial neural networks for classification

- To classify, they are trained to compute the posterior probabilities through Bayes' theorem and assign the sample to the most probable class:

$$p\left(\mathscr{C}_k|\mathbf{x}\right) = \frac{p\left(\mathbf{x}|\mathscr{C}_k\right)\cdot p\left(\mathbf{x}\right)}{\sum_{k'=1}^{K} p\left(\mathbf{x}|\mathscr{C}_{k'}\right)\cdot p\left(\mathbf{x}\right)} = \frac{\exp(\alpha_k)}{\sum_{k'=1}^{K}\exp(\alpha_{k'})}$$

$$\alpha_k = \ln p\left(\mathbf{x}|\mathscr{C}_k\right)\cdot p\left(\mathbf{x}\right)$$

- This activation function is called **normalized exponential** or **softmax** function
  - It is built so that:

$$\sum_{k=1}^{K} p\left(\mathscr{C}_k\middle|\mathbf{x}\right) = 1 \;\; \forall k \in \left[1,K\right] \subseteq \mathbb{N} \rightarrow p\left(\mathscr{C}_k\middle|\mathbf{x}\right) \in \left[0,1\right] \subseteq \mathbb{R}$$

Artificial neural networks for classification

- Consider a neural network with m outputs so that $m = K$;
- The target output is modeled through one-hot encoding
  - An m-dimensional vector with all values set to 0 except a 1 on the k-th dimension if the input belongs to the k-th class
- The networks uses the softmax$(\cdot)$ activation function to output the probabilities

$$y_k = p\left(\mathscr{C}_k \middle| \mathbf{x}\right) = \frac{e^{\alpha_k}}{\sum_{k'=1}^{K} e^{\alpha_{k'}}} = \frac{e^{\mathbf{w}_k^\top \cdot \mathbf{x}}}{\sum_{k'=1}^{K} e^{\mathbf{w}_{k'}^\top \cdot \mathbf{x}}}$$

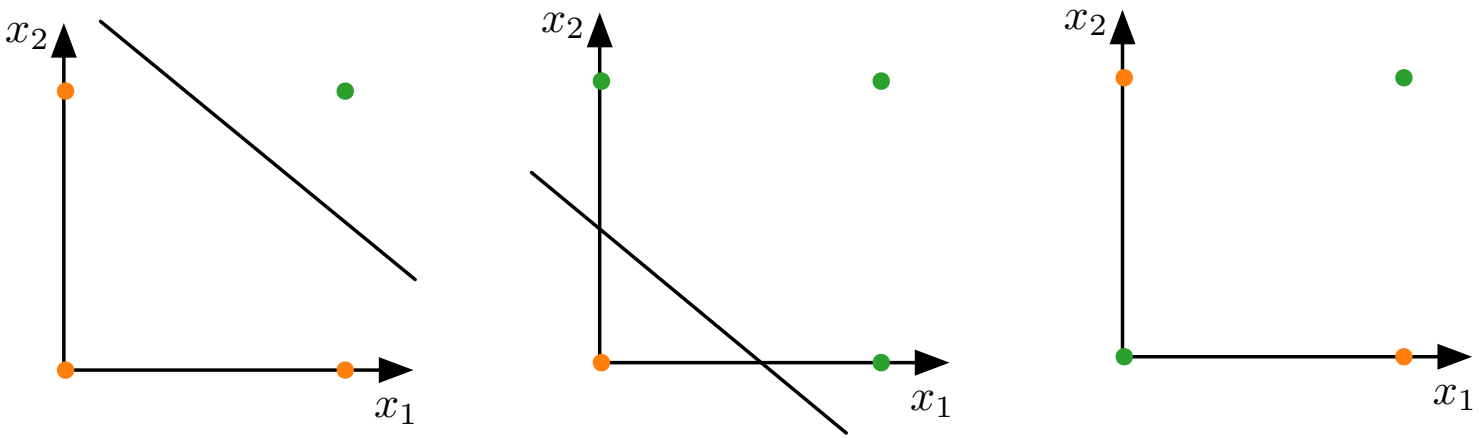
- In this case the values $\mathbf{w}_k^\top \cdot \mathbf{x}$ are referred to as the **logits**

Artificial neural networks for classification

- In the output each dimension represents the probability of the input to belong to the corresponding class
- If $K = 2$, problem can be simplified to a single output case
  - The network only needs one output and uses the sigmoid activation
    $$p\left(\mathscr{C}_1 \middle| \mathbf{x}\right) = 1 - p\left(\mathscr{C}_2 \middle| \mathbf{x}\right)$$
- Else if $K > 2$ the softmax($\cdot$) activation is used

Artificial neural networks limitations (single layer)

- From a geometric perspective, artificial neural network classifiers can be seen as linear classifiers with a decision boundary given by the hyperplane $w_{j0} + w_{j1}x_1 + \ldots + w_{jd}x_d = 0$
- Such classifier works well as long as the inputs are linearly separable in the input feature space



Basis functions

- Straightforward solution is to use a **basis function**
  - Project input into a different space
  - E.g. take the polynomial combination up to the n-th grade

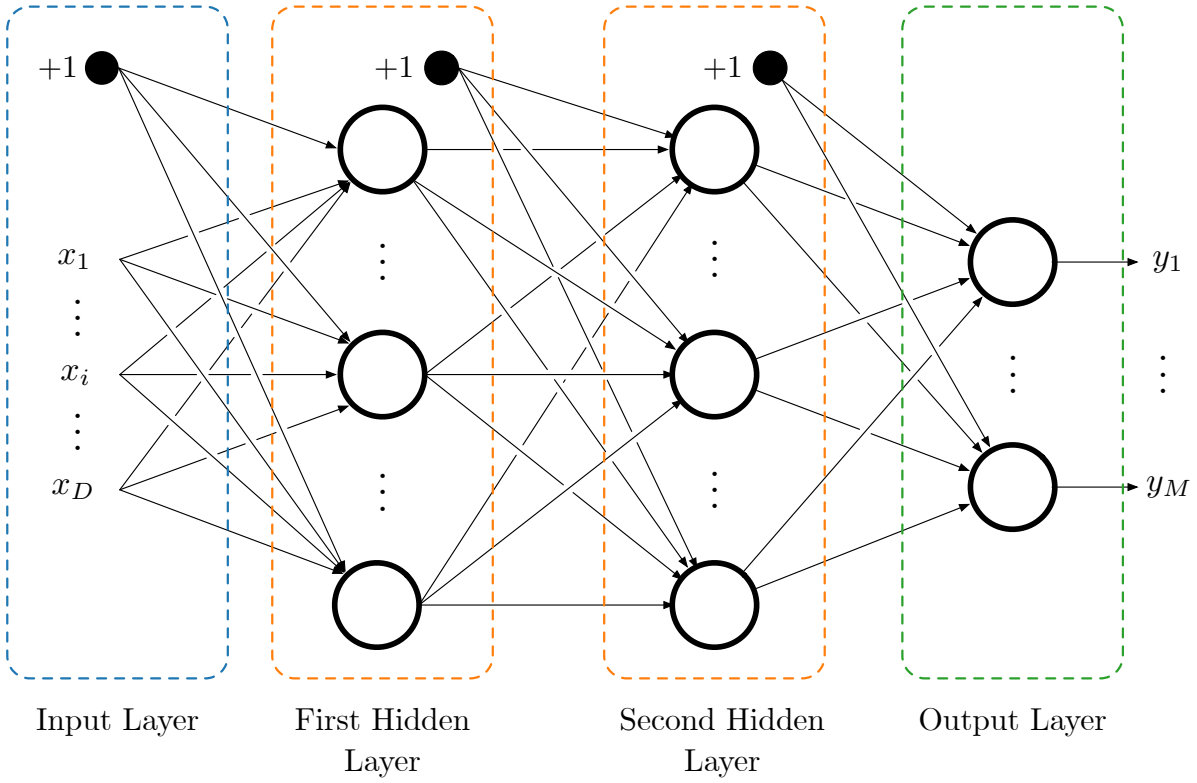
    $\mathbf{x} = < x_1, x_2 >$

    $\phi(\mathbf{x}) = < x_1, x_2, x_1^2, x_2^2, x_1 x_2 >$
- Use $\phi(\mathbf{x})$ as input to the network
- Model will stay linear in the parameters but the output will be non linear in the input
- This can be approximated through hidden layers

Feed forward neural networks

- Sometimes referred to as **Multi-Layer Perceptron** (MLP)
- One or more intermediate (**hidden**) layers are added, with non-linear activation function, between the input and the output
- From the **universal approximation theorem** (Kurt Hornik, 1991):
  - "A single hidden layer feedforward neural network with "S" shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set "
  - Actually, another kind of activation function is used instead of the "S" shaped ones
  - The point is to have a non-linear activation function

Feed forward neural networks



Deep neural networks

- De facto they're feed forward networks
- Huge number of hidden layers
- Trained on data sets with a huge number of samples
- The stack of hidden layers produces a hierarchy of transformations
- The intermediate values produced by the last hidden layers are very useful and transferrable features

TRAINING NEURAL NETWORKS

Training neural networks (supervised learning)

- Training a neural network means to find the weights for the layers
- In supervised learning settings, for every machine learning framework a training set is necessary
  - Couples of inputs-target output pairs
    $\mathscr{D} = \left\{ \left(\mathbf{x}_1, \mathbf{t}_1\right), \ldots, \left(\mathbf{x}_n, \mathbf{t}_n\right) \right\}$
- A cost (error, loss) function $E(\,\cdot\,)$ is used to measure goodness of the network
  - Weights are selected to minimize this error

Training neural networks (supervised learning)

- Training a neural network means to solve and optimization problem
  - "Find the weights that minimize the error of the network"
- The cost function measures the distance between the output of the network $\mathbf{y}_i$ given an input $\mathbf{x}_i$ and target output $\mathbf{t}_i$
  - The network is expected to produce an output $\mathbf{y}_i = \mathbf{t}_i$
  - The output of the network depend on its parameters ϑ
    $\mathbf{y}_i = g(\mathbf{x}_i \,|\, \vartheta)$
  - Thus the error is function of the network parameters ϑ
    $E(\mathbf{t}_i, \mathbf{y}_i) = E(\mathbf{t}_i, g(\mathbf{x}_i \,|\, \vartheta))$
- To find the optimal weights minimize the cost function

Training neural networks (supervised learning)

- To find the minimum of the error function it is "simply" necessary to compute the partial derivatives (in function of the parameters) and set them to 0
  - For linear problems it is possible to have a closed form solution

$$\nabla E(\vartheta) = \frac{\partial E(\vartheta)}{\partial \vartheta} = 0$$

- If the error function is non-convex (like when the function you're trying to learn is non-linear) closed form solutions are not available (not feasible)
  - Use iterative solutions
    - Start from random initialization of parameters
    - Iterate and update rule until convergence

Training neural networks (supervised learning)

- Depending on the problem, different error functions are used
- Regression problems usually employ the **sum of squared error**

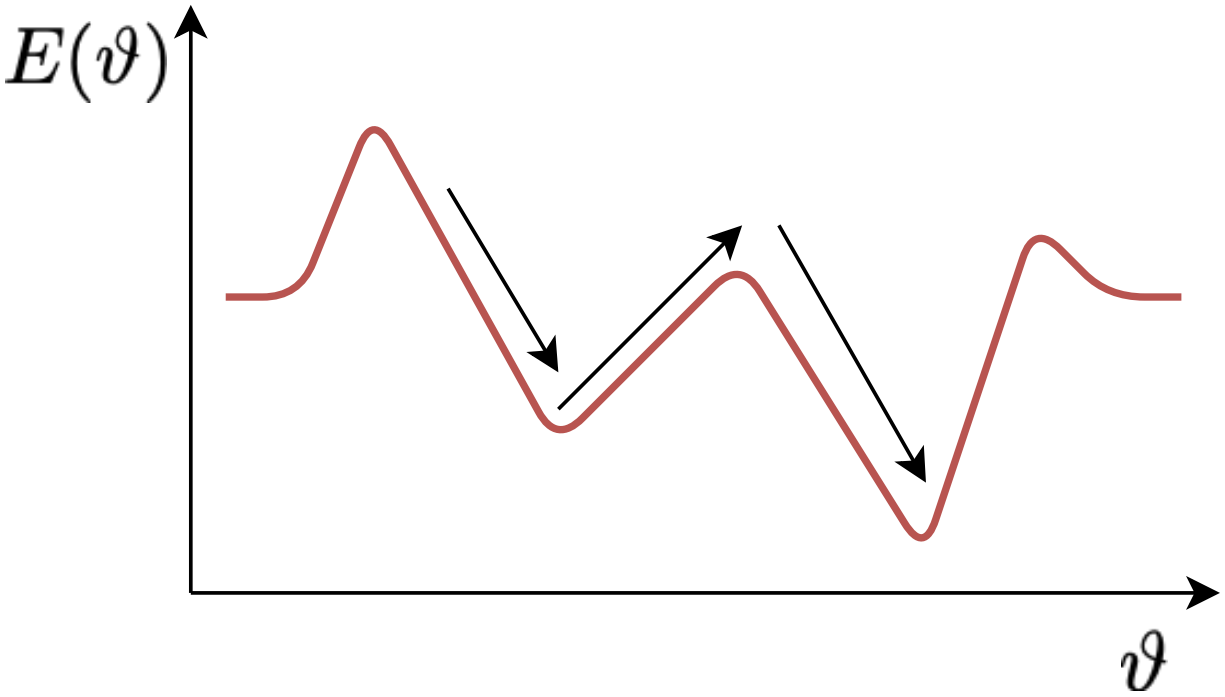
$$E(\vartheta) = \sum_{i=1}^{n} \left(\mathbf{t}_i - g(\mathbf{x}_i \,|\, \vartheta)\right)^2$$

- Classification problems usually employ the **cross-entropy**

$$E(\vartheta) = -\sum_{i=1}^{n} \mathbf{t}_i \log(g(\mathbf{x}_i \,|\, \vartheta))$$

- In general any error function can be used as long as is *differentiable*

Training neural networks (supervised learning)



Training neural networks (supervised learning)

- Neural networks are trained trough an optimization algorithm called **gradient descent** (or one of its variations)
  - At step (**epoch**) $\tau = 0$ the network is initialized with random parameters $\vartheta^{(0)}$
  - At each step $\tau \geq 1$ each weight $w \in \vartheta$ is updated according to the following rule

    $$w^{\tau} = w^{\tau-1} - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w=w^{(\tau-1)}}$$

  - Sometimes momentum is added to avoid local minima

    $$w^{\tau} = w^{\tau-1} - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w=w^{(\tau-1)}} - \alpha \frac{\partial E(w)}{\partial w}\bigg|_{w=w^{(\tau-1)}}$$

- η is called learning rate, is a parameter to control the update of the weights
- α is the momentum

Training neural networks (supervised learning)

- There are some variations of the gradient descent algorithm
  - **Batch** gradient descent: computes update on all the data set
  - **Stochastic** gradient descent (SGD): computes update on a single sample (unbiased but introduces high variance)
  - **Mini-batch** gradient descent: computes update on a subset of the data set
- There are some variations that adapt learning rate during training
  - Magnitude of gradient varies across layers and is lower for the first layers (*vanishing gradient*)
  - Solutions include RMSProp, AdaGrad, Adam, ...

Training neural networks (computing partial derivatives)

- The derivatives are computed through an algorithm called **back-propagation**
  - The computation is split in two steps: forward pass and backward pass
  - It takes advantage of the *chain-rule* to compute the derivatives of a composed function

$$z = f(y) = f(g(x))$$

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y}\frac{\delta y}{\delta x} = f'(y)g'(x) = f'(g(x))g'(x)$$

Assessing performances

- Data set is split into two subsets
  - **Train set**: the samples in this set are used to train the neural network
  - **Test set**: the samples in this set are used only to score the model, and never for training
- Sometimes an additional **validation set** is held out from the train set
  - These samples are used to test during training, they compute an estimate of the *generalization error*
- There is no absolute rule for splitting (usually 80-20%)

Assessing performances

- To have a more robust estimate training and testing can be repeated on different splits on the data set
- We talk of **cross-validation**
  - Usually k-fold cross-validation is performed
    - Data set is split into k parts (folds)
    - For each of the k folds, use it to test the network and train on the rest
    - Average the test results among the folds
  - Useful to perform hyper-parameters tuning and model selection in general

Model complexity

- Depending on the size of the model some problems can occur during training
- Too simple models are not able to learn
  - We talk of **underfitting**
- Too complex model do not generalize
  - We talk of **overfitting**

Limiting overfitting

- Overfitting can be limited
- Use **weights regularization** to shrink weights of less important features and help generalize by adding a regularization term
  - **Lasso** regularization changes the error function to $E_{Lasso}(\vartheta) = E(\vartheta) + \lambda \parallel \vartheta \parallel_1 = E(\vartheta) + \lambda \sum_{w \in \vartheta} |w|$ (these models are more sparse since weights are zeroed-out)
  - **Ridge** regularization (**weight decay**) changes the error function to $E_{Ridge}(\vartheta) = E(\vartheta) + \lambda \parallel \vartheta \parallel_2 = E(\vartheta) + \lambda \sum_{w \in \vartheta} w^2$
  - λ is the regularization parameters (should be chosen by cross-validation)

Limiting overfitting

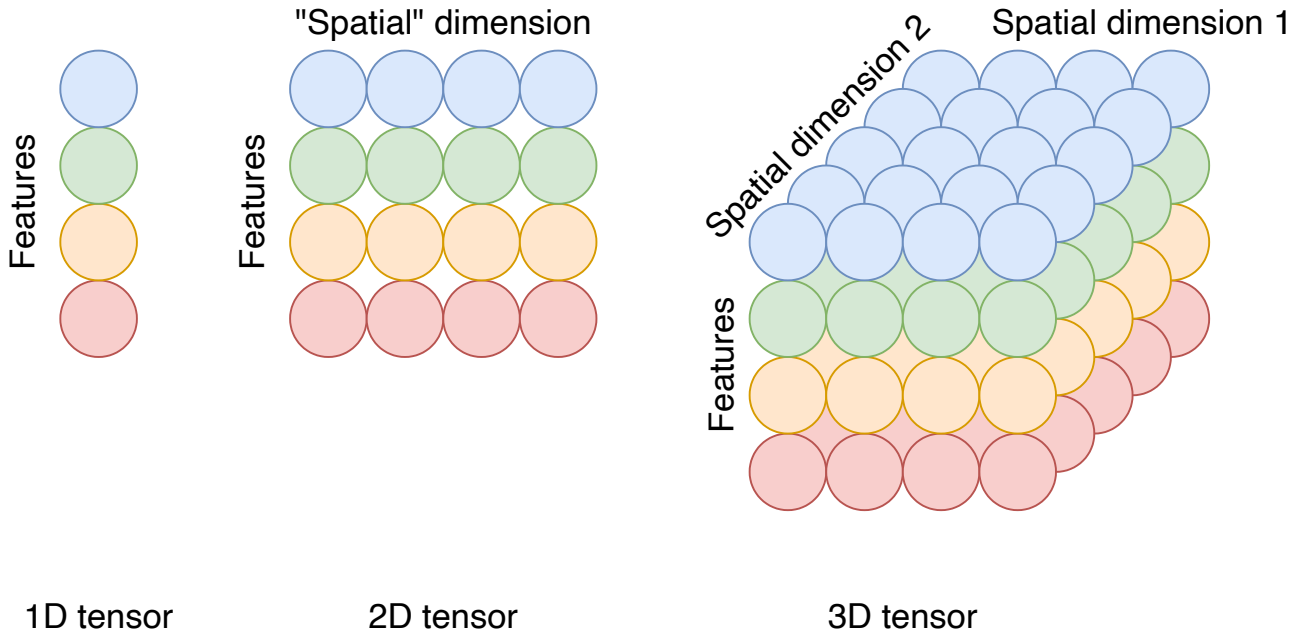
- Use **dropout** for stochastic regularization
  - Randomly zero-out the activations of some neurons during training
  - Forces neurons to learn independent features (hidden units cannot rely on each other)
  - Usually just $d\%$ of the neurons are shut down
  - d is the dropout rate (should be chosen by cross-validation)
- Use **early stopping** checking the error on the validation set at each epoch
  - Eventually the error on the train set will shrunk towards zero
  - The validation set can be used to estimate the generalization error
  - When the error on the validation set stop improving stop training
  - It is important not to use the validation set for training

LAYERS FOR NEURAL NETWORKS

Before starting

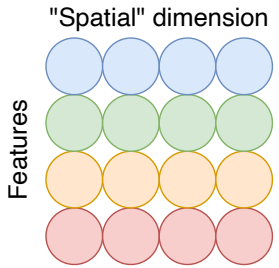
- **Tensors** are generalized matrices
  - They are characterized by a shape $\left(d_1, \ldots, d_n\right)$
  - Each element is the dimensionality along one of the n axes
  - The shape represents the dimensions of the tensor $\mathbf{X} \in \mathbb{R}^{d_1 \times \ldots \times d_n}$
- From now on we'll be working with tensors
  - They will be used to represent input, hidden values and sometimes also the output

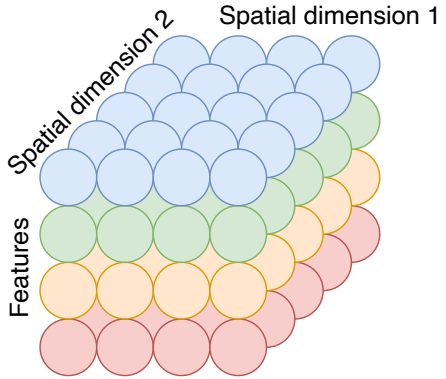
Tensors

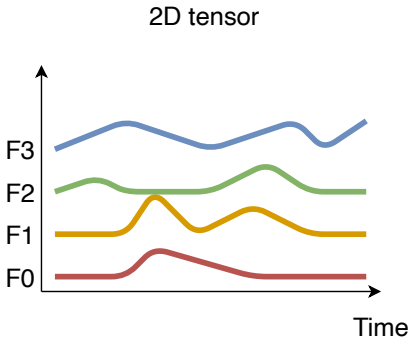


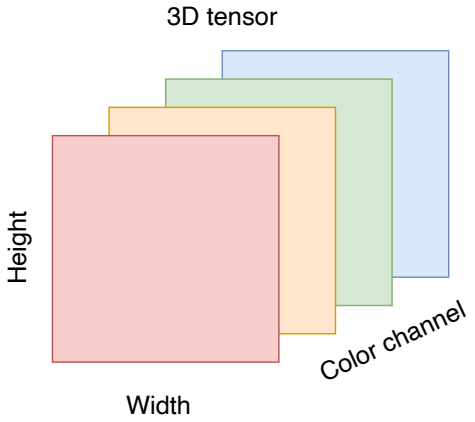
1D tensor 2D tensor 3D tensor

Tensors


"Spatial" dimension
Features


Spatial dimension 2
Spatial dimension 1
Features


2D tensor
F3
F2
F1
F0
Time


3D tensor
Height
Color channel
Width

Before starting

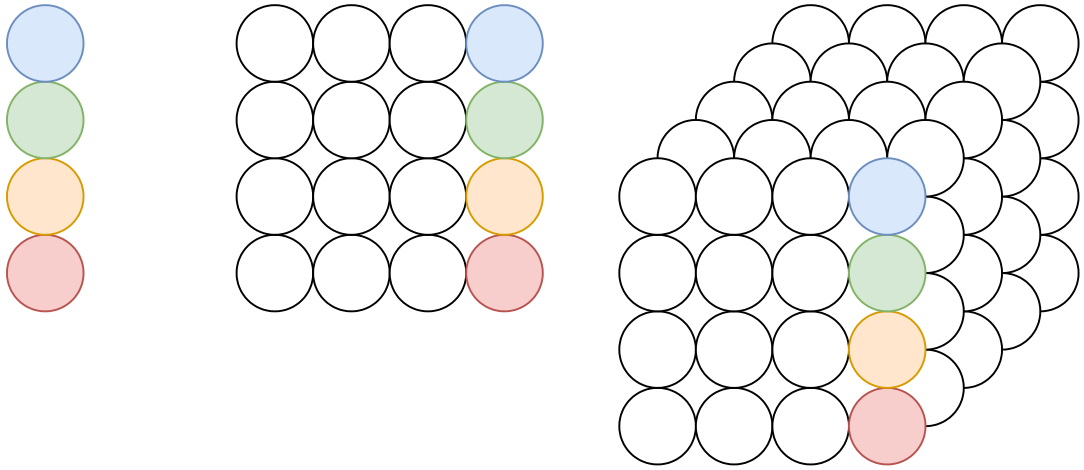
- Tensors flowing through the network's layers are sometimes referred to as **features maps**
  - The last axis is referred to as the **channel-axis** or **feature-axis**
  - The other axis represent the spatial dimensions of the input
    - E.g. a spectrogram is a tensor with 2 axes: the first is the "spatial" axis (time) and the last identifies the features (frequency bins, or considered formants)
    - E.g. an image is a tensor with 3 axes: the first two represent, respectively, height and width, the last one is the color channel axis
- A feature map with n axes is said to be $n-1$ dimensional ($n-1$D)

Before starting

- A single element of a feature map is often called **feature vector**
  - Feature vectors are identified within the tensor by the first $n - 1$ dimensions
  - Their dimensionality matches that of the last axis of the tensor
    - I.e. the number of features

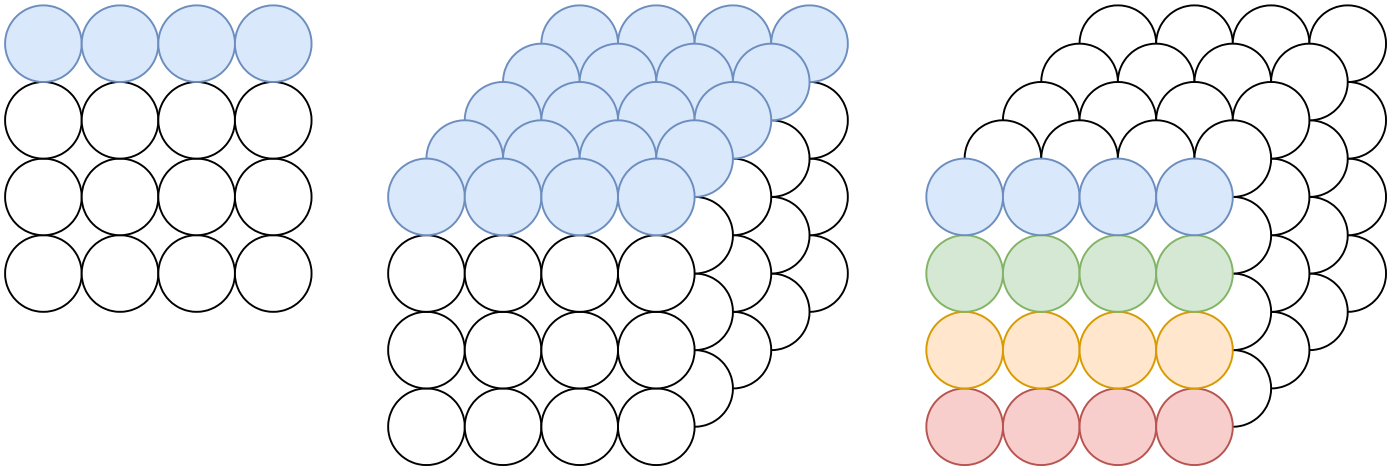
Tensors

- Feature vectors



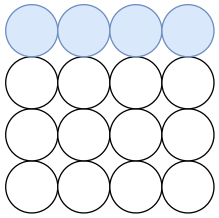
Tensors

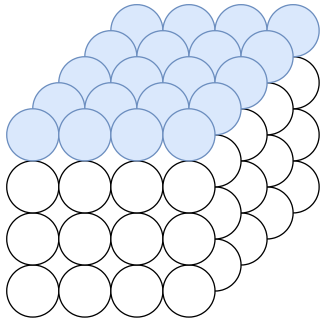
- Tensor slices

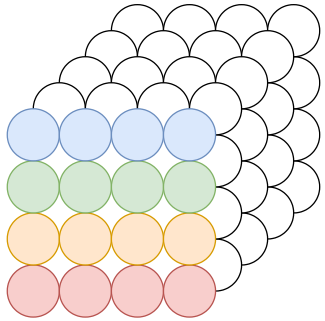


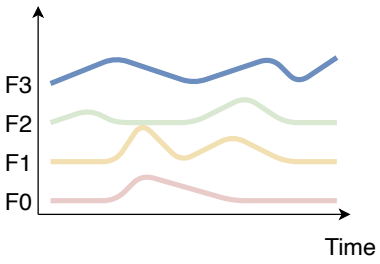
Tensors

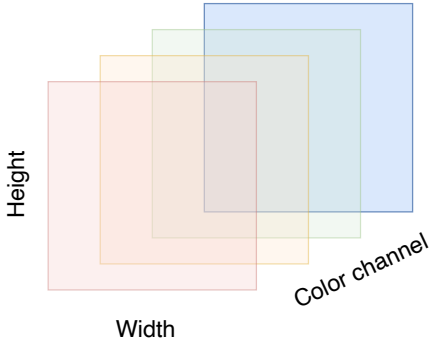
- Tensor slices

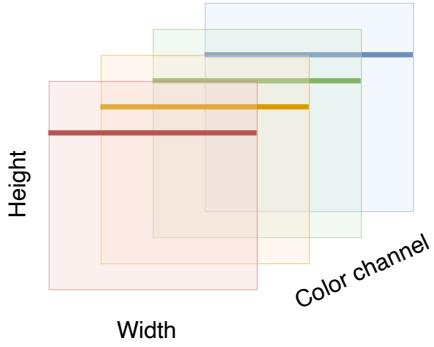












Fully connected layers

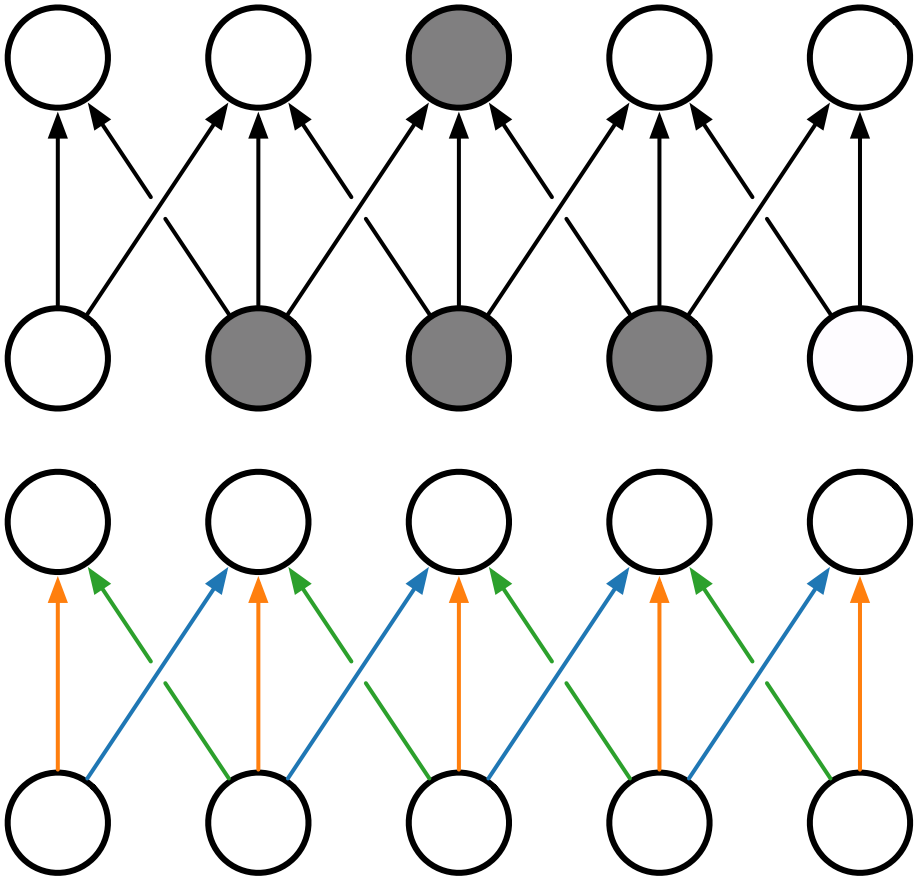
- **Fully connected** layers (or **dense** layers) are the simplest layers
- They take a $d_{in}-$dimensional vector of features as input and output a $d_{out}-$dimensional vector
  - The dimension of the output space depends on the number of output neurons
- Their weights are represented through a matrix $\mathbf{W}$ such that $\mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}$
- The output $\mathbf{y}$ given the input $\mathbf{x}$ is computed as $\mathbf{y} = g\left(\mathbf{W}^{\top} \cdot \mathbf{x}\right)$

Convolutional and pooling layers

- Basic constituents of Convolutional Neural Networks (CNNs)
- Take advantage of spatial organization of the input feature map (first $n-1$ axes)
- They learn a hierarchy of translation invariant features that can be used for classification and regression

Convolutional and pooling layers

- Take advantage of spatial organization
- Few weights



Convolutional layers

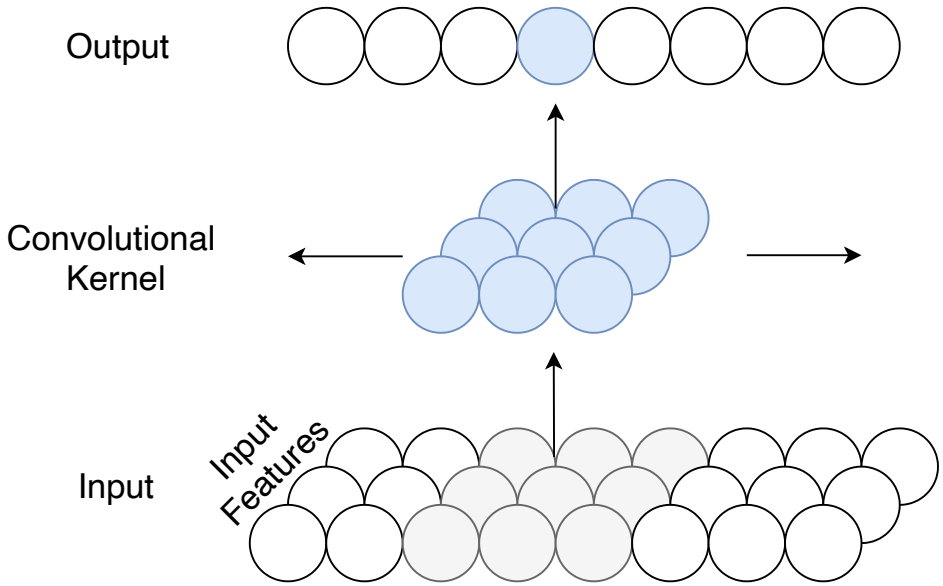
- They compute their output through a linear transformation called convolution
  - Output feature map is obtained convolving learnable filters over the input feature map
  - Filters are tensors with the same number of axes as the input, their channel axis matches the dimensionality of that of the input
  - These filters are slid over the entire input volume
  - Each filter yields a different slice of the output
    - A slice is a different level of the channel axis

Convolutional layers

- Convolution in a point of the input (identified by coordinates on the first $n-1$ axes) is computed by
  - Overlapping a kernel with the input in the given point
  - Computing the element-wise product between the filter and the overlapped portion of the feature map
  - Summing the values into a single scalar value
- The result is an element of the corresponding output feature vector
  - Sliding over the input with the same kernel computes a slice of the output
  - Changing kernel among those in the layer computes different dimensions of the output feature vector

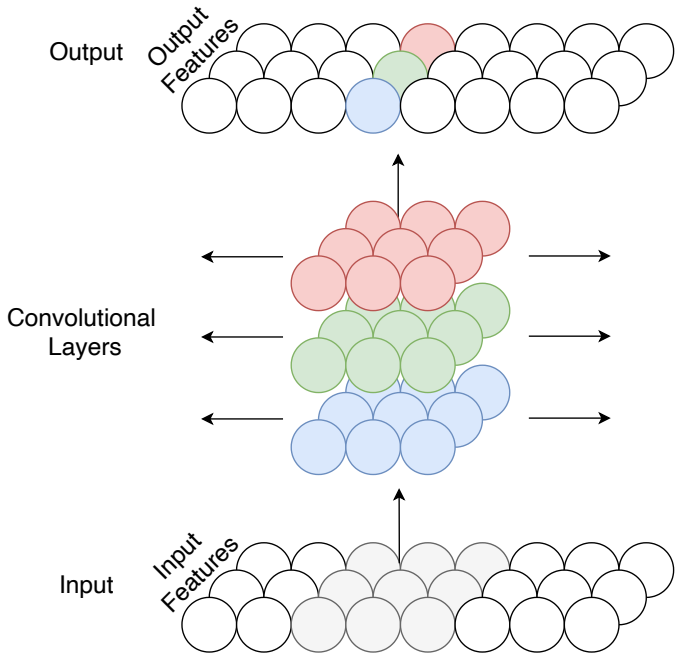
Convolutional layers (1D case)

- Single convolutional kernel



Convolutional layers (1D case)

- Many convolutional kernels



Convolutional layers (1-D case)

- Suppose to have as input a 1-D feature map (i.e. a 2-D tensor) $\mathbf{X} \in \mathbb{R}^{t \times n}$ so that
  - $\mathbf{x}_{(r)} = \mathbf{X}[r]$ is the r-th column of of the input tensor, a feature vector
  - $x_{(r,c)} = \mathbf{x}_{(r)}[c] = \mathbf{X}[r, c]$ is the c-th element of the r-th feature vector
- Suppose to have a convolutional filter of weights$\mathbf{W} \in \mathbb{R}^{s \times n}$
  - So that $w_{(r,c)} = \mathbf{W}[r, c]$
  - And
- The output of a convolutional layer with d filters $\mathbf{Y}$ is computed element wise as

$$\mathbf{Y}[r, k] = \mathbf{W}_k \star \mathbf{X}([r, c]) = \sum_{i=-\frac{s}{2}}^{\frac{s}{2}} \mathbf{w}_{k,(i+\frac{s}{2})}^{\top} \cdot \mathbf{x}_{(r+i)} = \sum_{i=-\frac{s}{2}}^{\frac{s}{2}} \sum_{j=1}^{n} w_{k,(i+\frac{s}{2},j)} \cdot x_{(r+i,j)}$$

  - $\mathbf{W}_k$ is the filter of the k-th of the d output channels

Convolutional layers (characteristics)

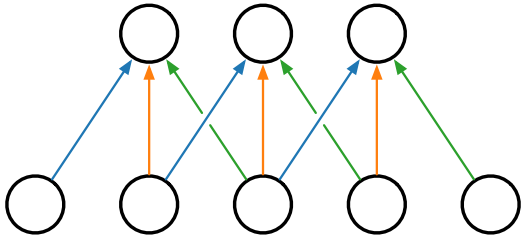
- **Kernel**: Tensor of weights slid over the input during the convolution operation (it has the same axes of the input feature map n)
- **Kernel size** (or shape): dimensions on the first $n-1$ axes of the convolutional kernel (last axis dimension is implicitly the same of the input)
- **Number of kernels**: identifies the number of convolutional kernels within a layer. This value controls the number of features in the output space

Convolutional layers (characteristics)

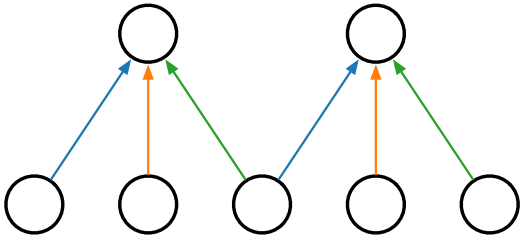
- **Stride**: size of the step (over the first $n-1$ axes) made by the kernel while sliding over the input
- **Dilation rate**: distance between two consecutive feature vectors considered by a convolution kernel while overlapping

Convolutional layers (characteristics)

- Stride:

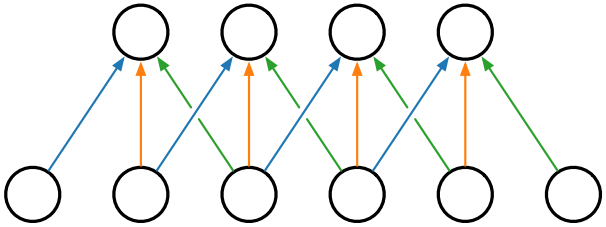


Kernel size = 3
Stride = 1

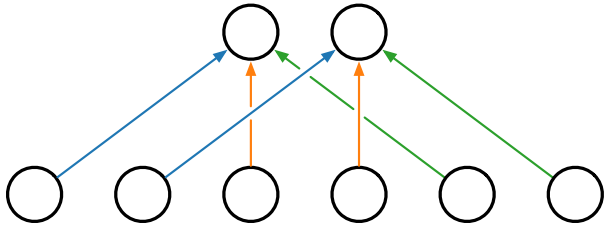


Kernel size = 3
Stride = 2

- Dilation rate:



Kernel size = 3
Dilation = 1



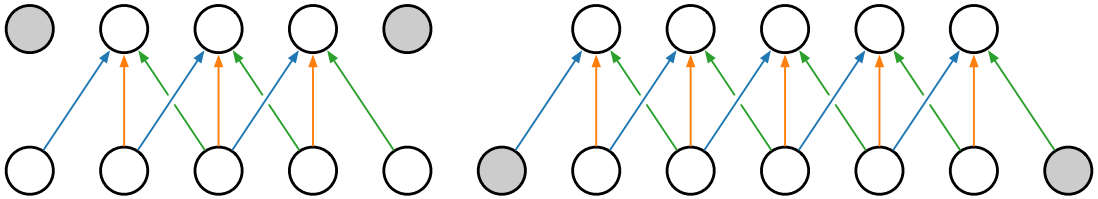
Kernel size = 3
Dilation = 2

Convolutional layers (characteristics)

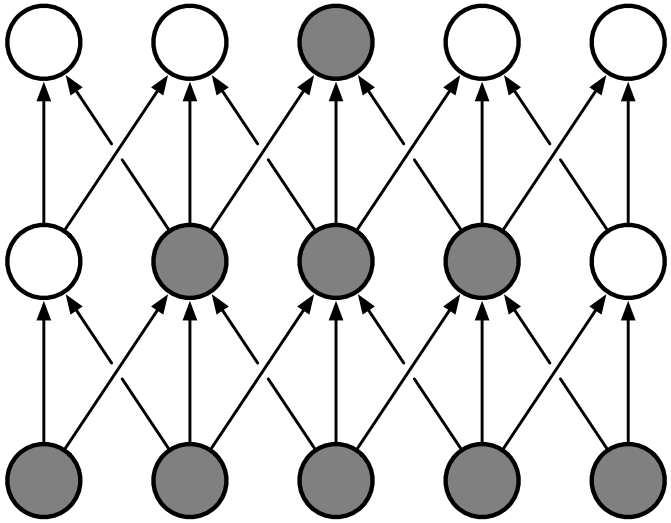
- **Padding**: policy to handle border. Two approaches:
  - Policy "same": dummy values are added at the borders of the first $n-1$ dimensions of the feature maps to have the same dimensions of the input in the output
  - Policy "valid": only portions of the input actually overlapping with the kernel are considered, thus output is smaller on the first $n-1$ axes (the last one depends on the number of kernels)
- **Receptive field**: it's the size of the context (on the first $n-1$ axes) seen by a stack of convolutions

Convolutional layers (characteristics)

- Padding:



- Receptive field:

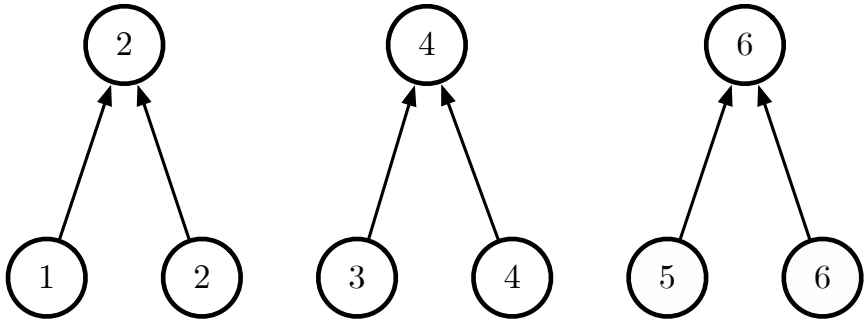


Pooling layers

- Pooling layers are used to downsample the spatial dimensions of the input feature map (first $n-1$ axes of the input tensor)
- They slide a window over the input as the convolutional layers
- They operate separately on each slice of the input map
  - They apply an operation on the values in the overlapped window
  - Usually they employ the $\mathbf{max()}$ or $\mathbf{average()}$ function
- Channel axis is preserved this time
  - Also in the meaning

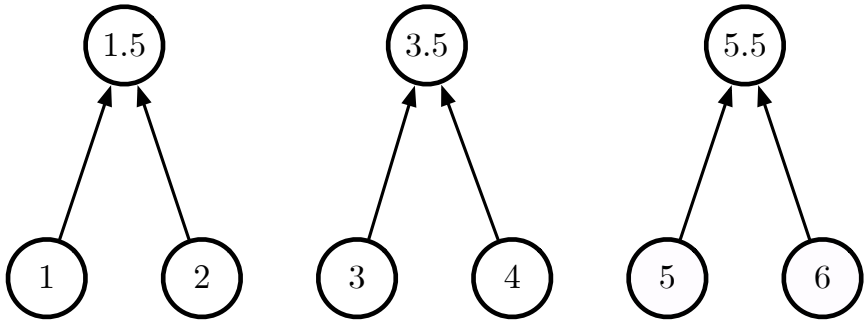
Pooling layers

- Max pooling:



Max-pooling
Kernel size = 2
Stride = 2

- Average pooling:



Average-pooling
Kernel size = 2
Stride = 2

Pooling layers (characteristics)

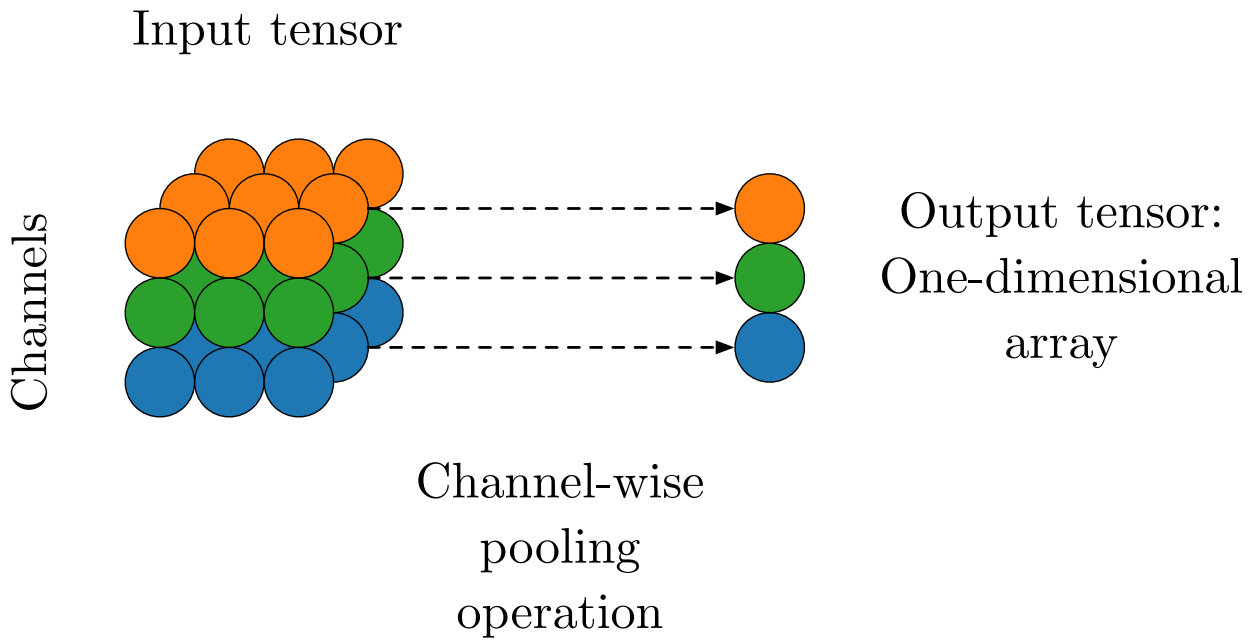
- Size and stride are defined as for the convolutions
  - Note that stride must be > 1 in order to reduce spatial dimensionality
- They to expand the receptive field of the stack of convolutions

Pooling layers (flattening)

- Usually the final operation of a neural net is fully connected one
  - Fully connected layers work only with features vectors (1-D tensors)
  - Necessary to **flatten** input feature map if it has different dimensions
- Previously the problem was solved by unrolling the feature map into a vector
  - This forces the input to a fixed dimension, otherwise shapes for the output won't match
- Use **Global Pooling layers**
  - Compute pooling operation on the entire slice of the feature map for each channel
  - Output is a vector with as many elements as the channels of the input feature map
  - Usually they employ the $\max()$ or $\text{average}()$ function
  - Helps regularization

Pooling layers (flattening)

- Global pooling layers:



Sequence modelling

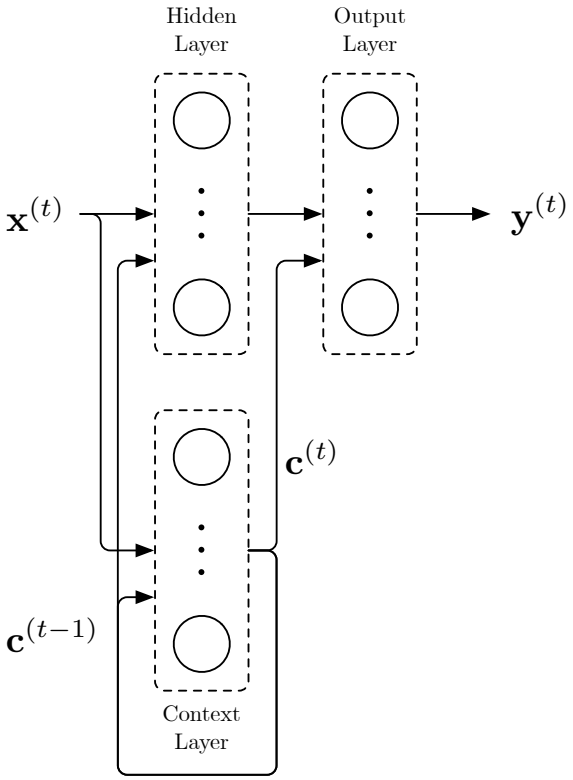
- Sometimes is necessary to deal with dynamic systems
  - Data evolving through time
- Memoryless solutions:
  - work by predicting the next output using delayed inputs
  - Go back only a fixed amount of time steps
  - Fail if the time window is not sufficient
- Models with memory
  - They can consider a virtually infinite context
  - Goes back in time indefinitely

Sequence modelling

- Memoryless models:
  - Autoregressive models
    - Like CNNs
  - Feed forward neural networks with delayed input
- Models with memory
  - Hidden Markov Models (HMMs)
    - Stochastic systems that use real-valued hidden state to represent the past, it cannot be observed directly but can be estimated with the Viterbi algorithm
  - **Recurrent Neural Networks** (RNNs)
    - Deterministic system, use a context vector to store the state, the non linear dynamics allows complex hidden state updates

Recurrent Neural Networks

- Memory is introduced through a context vector $\mathbf{c}^{(t)}$
  - Context vector is managed through recurrent transformations
- The output depends on both current input and previous context
- Proved to be Turing-complete (Siegelmann, 1995)
  - Provided a sufficiently large context vector

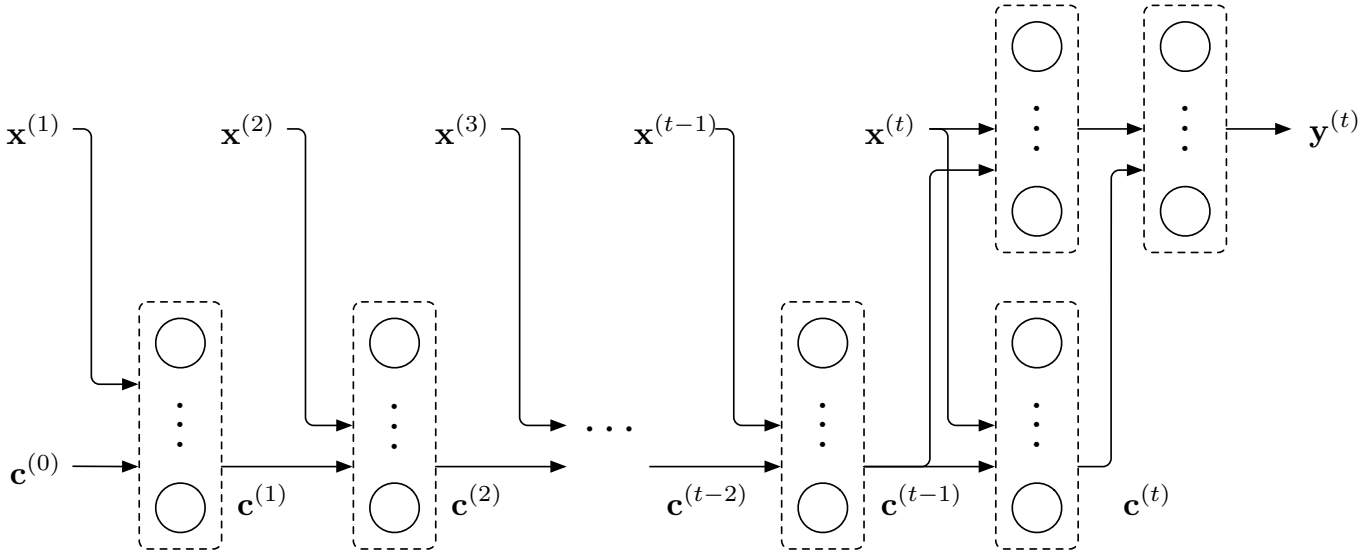


Recurrent neural networks

- The output depends on current input $\mathbf{x}^{(t)}$ and context vector $\mathbf{c}^{(t-1)}$
- **Output**
  $\mathbf{y}^{(t)} = g\left(\mathbf{W}^{\mathsf{T}} \cdot \mathbf{h}^{(t)} + \mathbf{V}^{\mathsf{T}} \cdot \mathbf{c}^{(t)}\right)$
- **Hidden** vector
  $\mathbf{h}^{(t)} = h\left(\mathbf{W}_h^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{V}_h^{\mathsf{T}} \cdot \mathbf{c}^{(t-1)}\right)$
- **Context** vector
  $\mathbf{c}^{(t)} = c\left(\mathbf{W}_c^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{V}_c^{\mathsf{T}} \cdot \mathbf{c}^{(t-1)}\right)$

Recurrent neural networks (unfolding)

- Back-propagation is performed through time to train the network
- The RNN is said to be **unfolded**
- Weights are updated with the average of the gradient



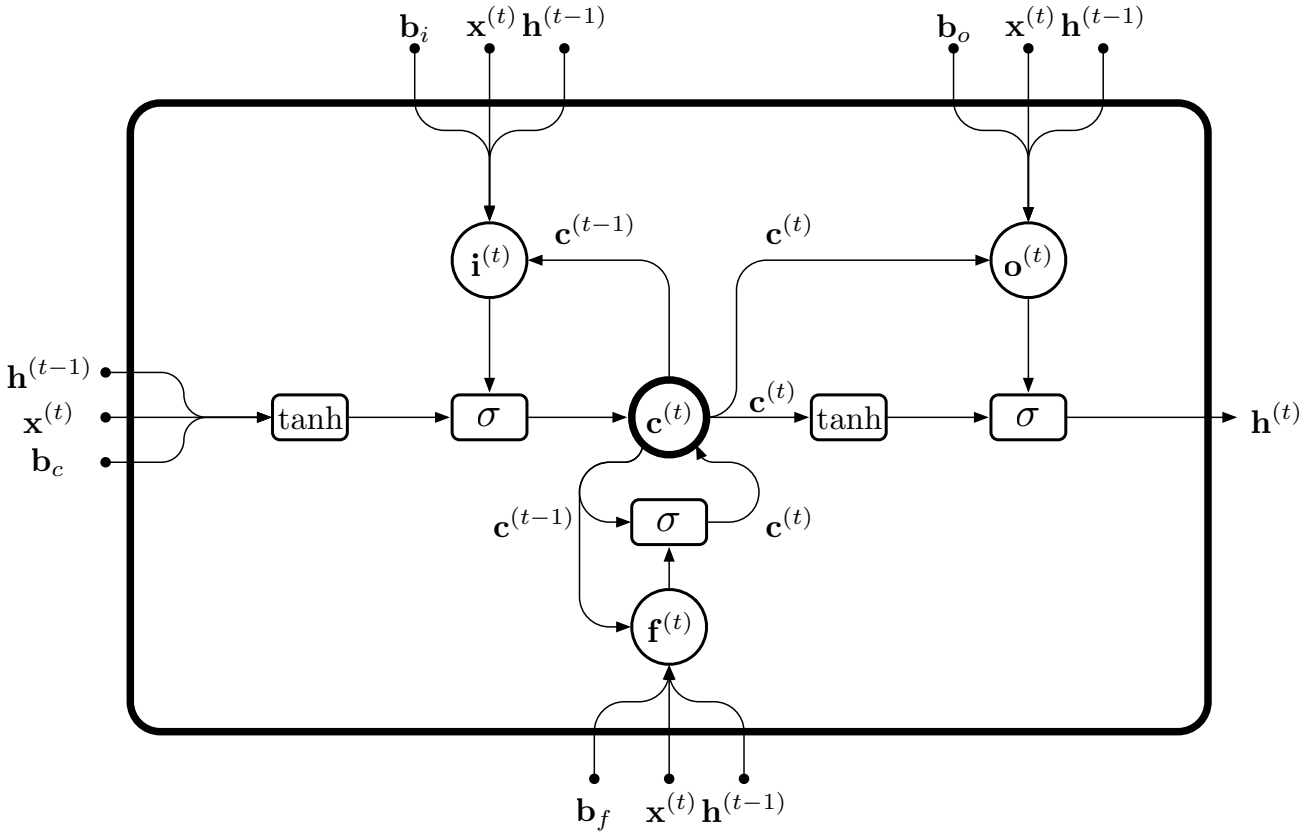
Recurrent neural networks (limitations)

- Vanilla RNNs cannot go back indefinitely in time
  - Vanishing or exploding gradient during back propagation
  - Not robust to noise
- Solution: use gating mechanism
  - **Long Short-Term Memory** (LSTM)(Hochreiter and Schmidhuber, 1997)
  - **Gated Recurrent Units** (GRU)

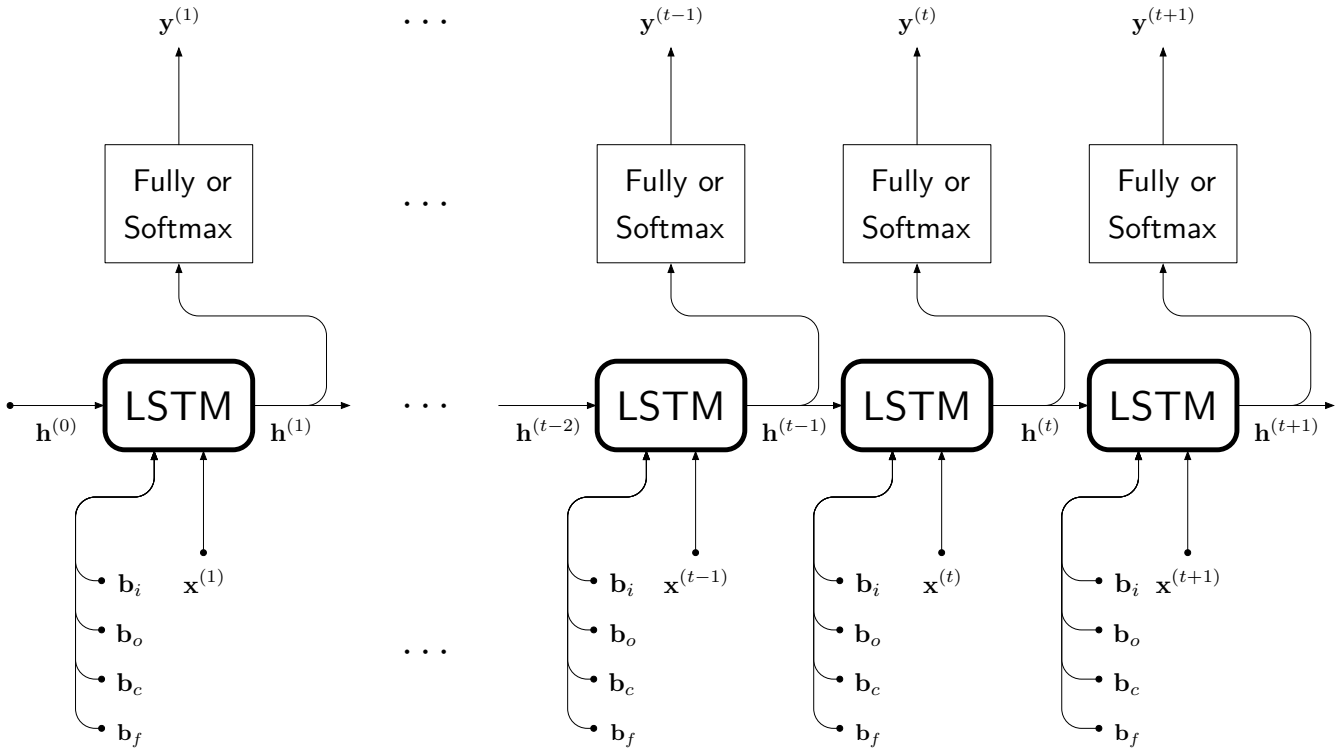
Long short term memory

- LSTMs solved the problem of vanishing gradient;
- Characterized by:
  - **Input** vector $\mathbf{x}^{(t)}$
  - **Output** (or hidden) $\mathbf{h}^{(t)}$
  - **State** (**memory**) vector $\mathbf{c}^{(t)}$
- Internal gates control the flow of information:
  - **Input gate** $\mathbf{i}^{(t)}$: controls the amount of new information to set in the cell state
  - **Forget gate** $\mathbf{f}^{(t)}$: controls the amount of information discarded from the cell state
  - **Output gate** $\mathbf{o}^{(t)}$: controls the amount of information to set in the output

Long short term memory



Long short term memory



Long short term memory

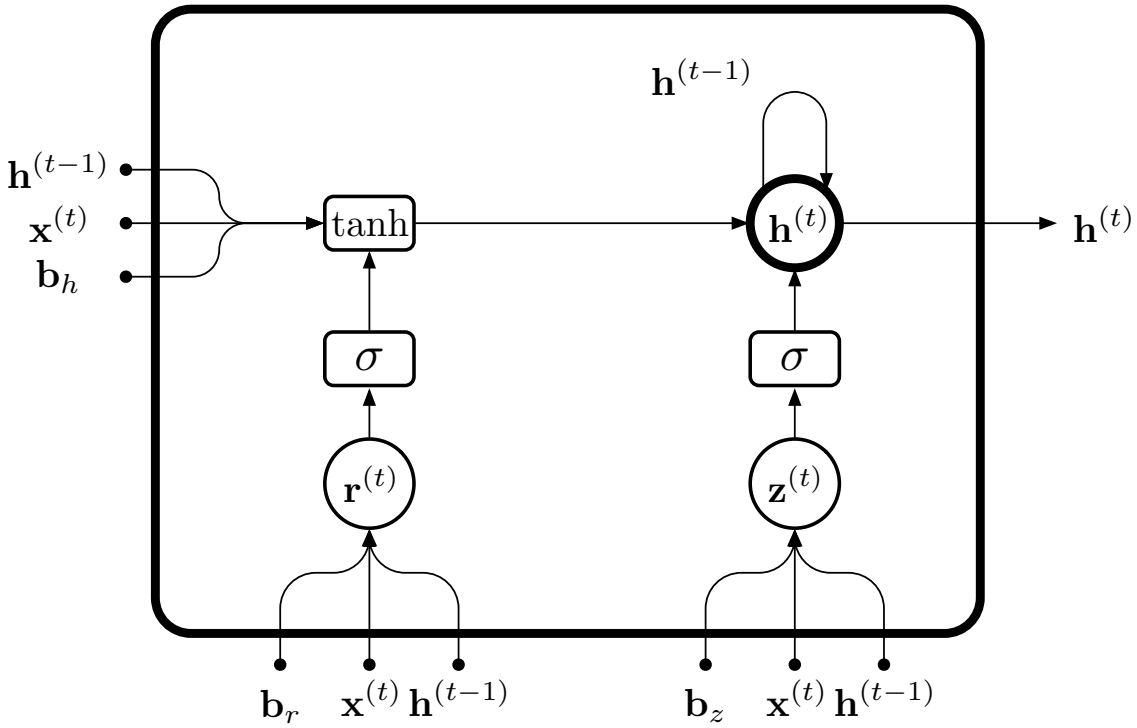
- **Input** gate
  $$\mathbf{i}^{(t)} = \sigma\left(\mathbf{W}_{xi}^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{W}_{hi}^{\mathsf{T}} \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_{ci}^{\mathsf{T}} \cdot \mathbf{c}^{(t-1)} + \mathbf{b}_i\right)$$
- **Forget** gate
  $$\mathbf{f}^{(t)} = \sigma\left(\mathbf{W}_{xf}^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{W}_{hf}^{\mathsf{T}} \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_{cf}^{\mathsf{T}} \cdot \mathbf{c}^{(t-1)} + \mathbf{b}_f\right)$$
- **Output** gate
  $$\mathbf{o}^{(t)} = \sigma\left(\mathbf{W}_{xo}^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{W}_{ho}^{\mathsf{T}} \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_{co}^{\mathsf{T}} \cdot \mathbf{c}^{(t)} + \mathbf{b}_o\right)$$
  $$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \circ \tanh\left(\mathbf{c}^{(t)}\right)$$
- State update (**memory** gate)
  $$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \circ \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \circ \tanh\left(\mathbf{W}_{xc}^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{W}_{hc}^{\mathsf{T}} \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_c\right)$$

- Note that $\circ$ is the element-wise product

Gated recurrent units

- GRUs are a variation of the LSTMs
  - Lightweight implementation
  - Perform better on smaller data sets
- Combine input and forget gate into **update gate** $\mathbf{z}^{(t)}$
- Output gate becomes **reset gate** $\mathbf{r}^{(t)}$
- Merge cell state and **hidden state** $\mathbf{h}^{(t)}$

Gated recurrent units



Gated Recurrent Units

- **Update** gate

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}_{xz}^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{W}_{hz}^{\mathsf{T}} \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_z\right)$$

- **Reset** gate

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}_{xr}^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{W}_{hr}^{\mathsf{T}} \cdot \mathbf{h}^{(t-1)} + \mathbf{b}_r\right)$$

- **Output** gate

$$\mathbf{h}^{(t)} = \left(1 - \mathbf{z}^{(t)}\right) \circ \mathbf{h}^{(t-1)} + \mathbf{z}^{(t)} \circ \tanh\left(\mathbf{W}_{xh}^{\mathsf{T}} \cdot \mathbf{x}^{(t)} + \mathbf{W}_{rh}^{\mathsf{T}} \cdot \left(\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}\right) + \mathbf{b}_h\right)$$

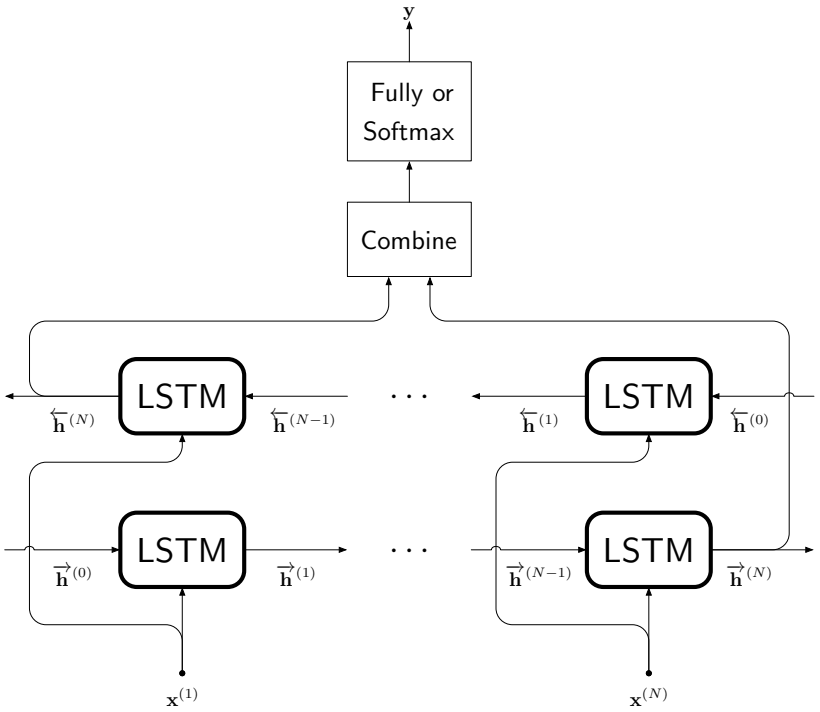
Bi-directional recurrent layers

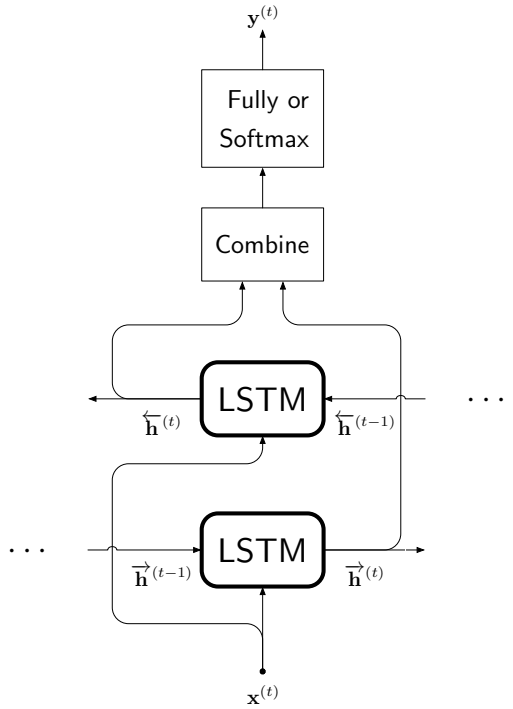
- Recurrent layers can scan 1D input features maps in both directions
- Leverage information from both sides combining the intermediate representation
  - Obtain richer representations

    $\mathbf{h}^{(t)} = \overrightarrow{\mathbf{h}}^{(t)} \oplus \overleftarrow{\mathbf{h}}^{(t)}$

    where $\oplus$ is a generic operation combining $\overrightarrow{\mathbf{h}}^{(t)}$ and $\overleftarrow{\mathbf{h}}^{(t)}$
    - Concatenation
    - Element wise sum
    - ...
- This approach is compatible with any kind of recurrent layer (RNN, LSTM, GRU)

Bi-directional recurrent layers





Attention mechanism

- Attention mechanism is used to produce a **weighted sum** combining all the content of a given input
  - The output at each point is a combination of all the points in the input
  - For each output point the weights in the combination are different
- Attention provides a direct route for the information to flow from input to output

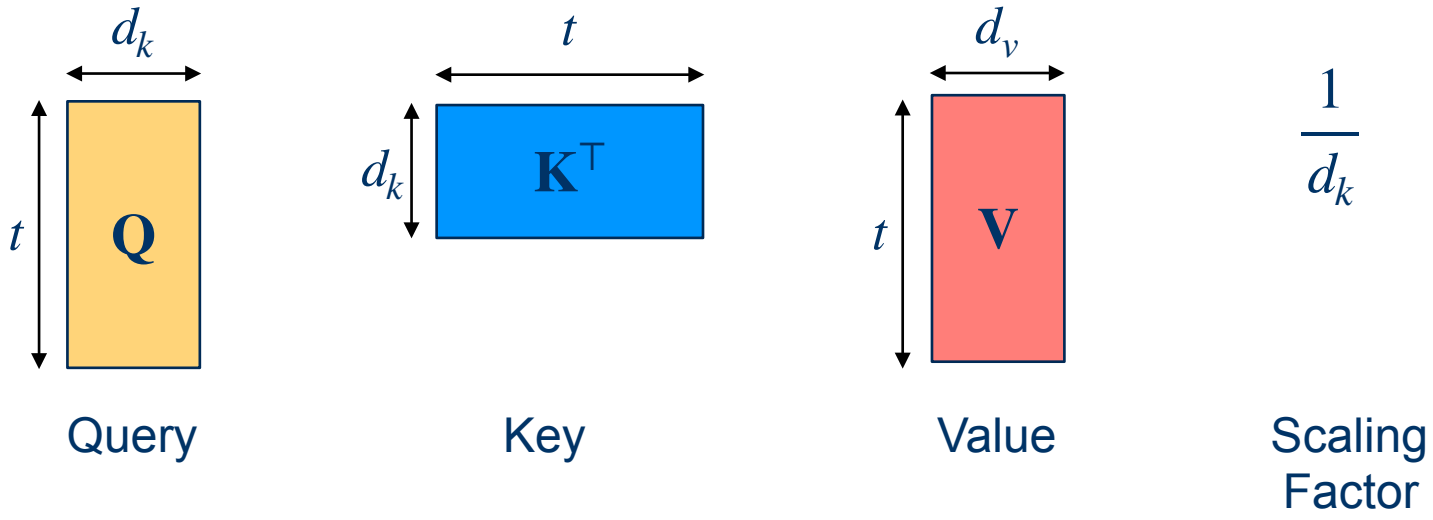
Attention mechanism (1D case)

- Most implementations use **scaled dot-product attention**

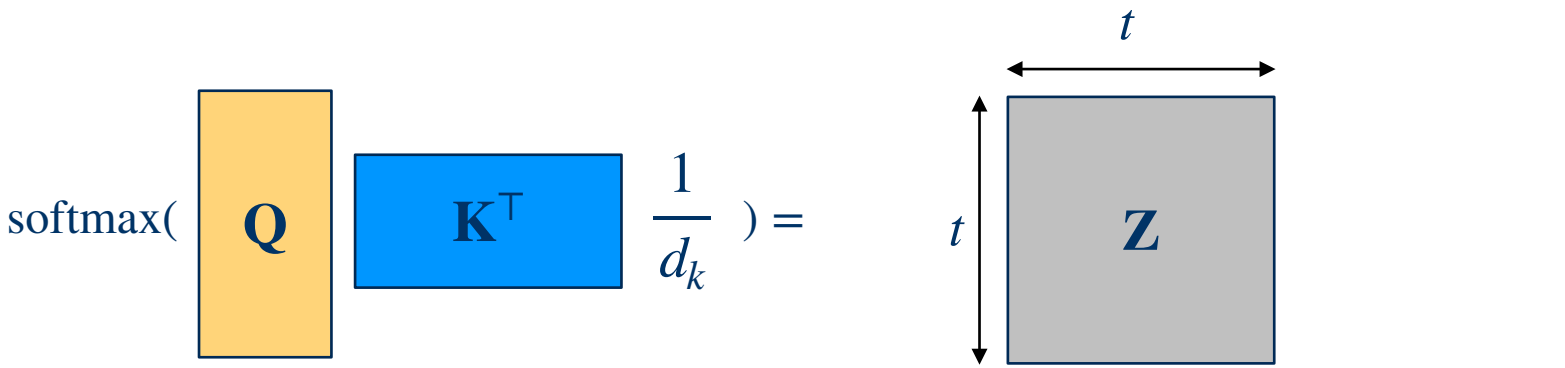
$$f_{\text{attention}}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right) = \text{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^{\top}}{\sqrt{d_k}}\right) \cdot \mathbf{V}$$

- Three input values:
  - $\mathbf{Q}$ is the query, a feature map $\mathbf{Q} \in \mathbb{R}^{t \times d_k}$
  - $\mathbf{K}$ are the keys, a feature map $\mathbf{K} \in \mathbb{R}^{t \times d_k}$
  - $\mathbf{V}$ are the values corresponding to the keys, a feature map $\mathbf{V} \in \mathbb{R}^{t \times d_v}$
- The output is a feature map $\mathbf{H}$ such that $\mathbf{H} \in \mathbb{R}^{t \times d_v}$
- Note: this mechanism can be generalized to any kind on input shape (i.e. n-dimensional)

Attention mechanism (visualized)



Attention mechanism (visualized)



Attention coefficients

Scaled dot product between query and key produces the attention weights, which are scaled in each row of the output through a $\mathrm{softmax}(\cdot)$

Each row of $\mathbf{Z}$ corresponds to a feature vector of the values, each value in the row is the weight to assign to the corresponding feature vector of the value before summing

Attention mechanism (visualized)

The weighted sum of the elements of $\mathbf{V}$ is done implicitly by the dot product.

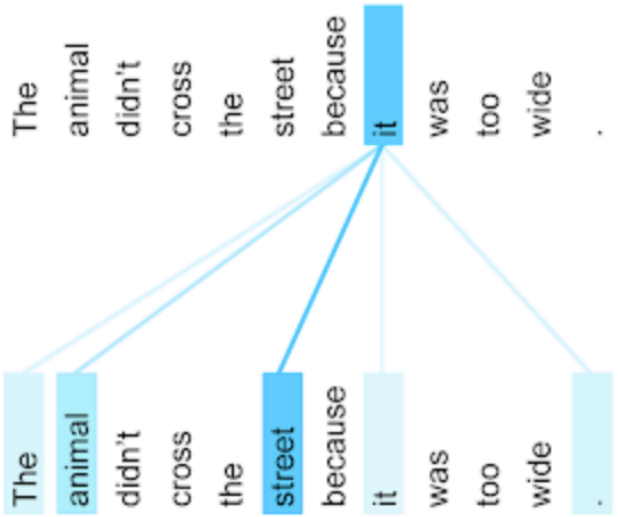
Each row of $\mathbf{V}$ is scaled by the weight corresponding to the current output

Output feature map

Each feature vector of the output feature map is a weighted compilation of the features vectors in the values feature map. Notice the correspondence in the spatial dimension

Attention mechanism (visualized)

- Example on text of self attention
  - The attention gives more weight to the related words



Attention mechanism (explained)

- Each query vector is matched against each key vector
  - Matching is done through the (**scaled**) **dot-product**
  - In practice the correlation between each **query-key pair** is computed
    - For each pair a scalar value is produced
  - The correlation is scaled into **attention scores** (through the $\mathrm{softmax}(\cdot)$)
    - The sum of all the attention scores (weights) between each single query and all the keys is 1

Attention mechanism (explained)

- All the weights produced by each query are used to combine the values
  - For each query all the values are combined through a weighted sum
  - The attention score of a **query-value pair** says how much of the value put in the combination
    - This is done query-wise

Multi-head attention

- Different attentions computed in parallel
- The operations are the same
- All outputs are concatenated and contented through a fully connected transformation $\mathbf{W}_{out}$ (with $\mathbf{W}_{out} \in \mathbb{R}^{(d_v h) \times d_{out}}$)

$$f_{\text{Multi-Head Attention}}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right) = f_{\text{Concatenate}}(\mathbf{H}_1, \ldots, \mathbf{H}_h) \cdot \mathbf{W}_{out}$$

$$\mathbf{H}_i = f_{\text{Attention}}\left(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i\right)$$

Self-attention

- Current hidden feature map $\mathbf{X} \in \mathbb{R}^{t \times d_x}$ is used as query, key and value
- Usually a linear (fully connected) transformation is performed to have separate values for the three feature maps
  - $\mathbf{Q} = \mathbf{W}_q^\top \cdot \mathbf{X}$, with $\mathbf{W}_q^\top \in \mathbb{R}^{d_x \times d_k}$
  - $\mathbf{K} = \mathbf{W}_k^\top \cdot \mathbf{X}$, with $\mathbf{W}_k^\top \in \mathbb{R}^{d_x \times d_k}$
  - $\mathbf{V} = \mathbf{W}_v^\top \cdot \mathbf{X}$, with $\mathbf{W}_v^\top \in \mathbb{R}^{d_x \times d_v}$

Attention (1D case)

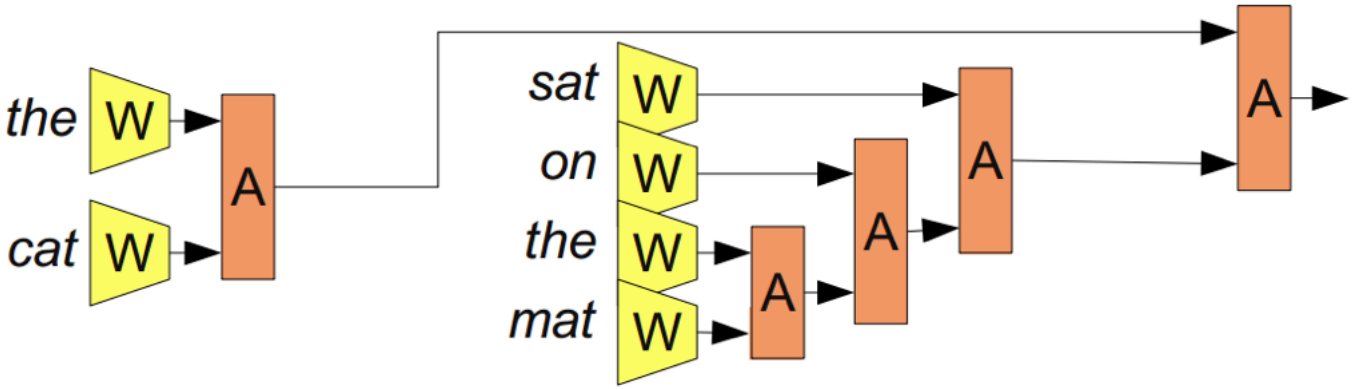
- Attention mechanism was introduced for sequential analysis
- Combined with recurrent networks at first
- Slightly substituted the recurrent layers nowadays
  - Recurrent layers are slower as they require sequential computation of each element
  - Attention mechanism is faster as its output can be computed “all at once” in parallel
    - However they lack of positional information during the combination (more on this later)

Recursive neural networks

- Sometimes called **tree-structured neural networks**
- Special kind of network, serves the purpose of association network
  - It's used to merge the components of the input
  - Normal networks merge the input in a linear way, but sometimes it doesn't necessarily make sense
- Output is fed back to the input (this is why they're called recursive)
  - Build a new representation merging the components recursively

Recursive neural network

- At each step the function is always the same
- What changes are the input representations that are merged together at each step



Miscellanea

- In the following are list some other informations to build neural networks

- Some layers have variants
  - Like separable convolution
- To help deep neural network converge is necessary to have gradient reach the first layers
  - Use skip connections
- How layers should be selected

Separable convolutions

- Separable convolutional layers are a special kind of convolutional
- They break the operation in two steps
  - **Depth wise convolution**
  - **Point wise convolution**
- They can be seen as a factorized convolution
  - Can perform the same operation with fewer parameters
  - If no *intermediate non-linearity* is inserted it is possible to transform the separable convolution into a normal convolution
    - However it is not always possible the vice-versa

Separable convolutions (1D case)

- Depth wise convolution:
  - Given a 1D feature map $\mathbf{X}$ with n channels
  - This layers performs n distinct convolutions (using as many kernels)
  - The peculiarity is that convolutions are performed channels wise
    - Each kernel works on a single slice of input and output
- Point wise convolution:
  - Is a classic convolution but all the spatial dimensions of the shape of the filters (first $n-1$ dimensions of the tensor) are set to 1
  - In practice this part takes care of combining the input features into new features

Skip connections

- Also called **residual connections** or **identity shortcut connections**
- In practice they introduce an identity transformation of the input feature map that *skips* one or more layer (transformations)
- The output is given by the sum of the feature map extracted by the new transformation on the input feature map
  $\mathbf{H} = f(\mathbf{X}) + \mathbf{X}$
- Helpful to train deep networks
  - Help to cope with the vanishing gradient problem
- Designed for CNNs they're actually compatible with any kind of layer

Building neural networks

- Neural Networks can be built combining layers in different manners
- The choice of the layer depends on the problem
  - I.e. the shape of the input data
- There is no best solution
  - Perform model selection to find the most appropriate one

Building neural networks

- NLP deals mostly with sequences
- Inputs are 2D tensors (1D feature maps)
- Compatible layers are
  - 1D convolutional and pooling layers
  - Recurrent layers
  - Attention layers
- However other kind of layers can be used

LEARNING APPROACHES

Representation learning

- Learn hidden features useful for other task
  - Find rich and informative representation of input
- Usually done training the network on a generic task on raw input
  - E.g. predict next word
- In some cases we talk of **self-supervised** or **unsupervised pre-training**

Transfer learning and fine-tuning

- Learnt representations or intermediate features from deep networks can be used to feed other models
- **Transfer learning**: reuse bottom layers of a pre-trained network to generate input features
  - Just remove fully connected on top used for the original classification/regression problem
  - Applies when input domain is the same
  - Pre-trained weights are "frozen"
- **Fine-tuning**: reuse bottom layers of a pre-trained network as weights initialization for the input features
  - Just remove fully connected on top used for the original classification/regression problem
  - Applies when input domain is different
  - Pre-trained weights are used as initialization and then updated

Adversarial learning

- Approach used to train **generative** models
  - Up to now we've seen discriminative ones
- Learn to sample the target space distribution
- Train two networks:
  - **Generator** creating samples from scratch
  - **Discriminator** to discriminate synthetic sample from real ones
- Generator is rewarded in training for fooling the discriminator and discriminator is rewarded for detecting the generator
- At convergence synthetic samples should not be distinguishable from real ones
- Helpful when there no appropriate loss to perform a task

Curriculum learning

- Iteratively increase task complexity to help the network generalize step by step
- Useful when only few data are tagged for the specific task
- Useful to help convergence

Knowledge distillation

- Sometimes deep networks are too computationally demanding to be deployed in everyday scenarios
- There are techniques to compress or factorize them
- An alternative developed in the latest years is **knowledge distillation**
- Train two distinct neural networks:
  - **Teacher**: very deep neural network trained on a huge data set
  - **Student**: smaller network trained to reproduce teacher behavior
    - Learns from the "weak" labels given by the teacher instead (or besides) of the original data sets labels
    - Easier to train smaller networks on same huge data set
    - Similar performances but still some degradation

REFERENCES

References

- McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133.
- Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain". Psychological Review. 65 (6): 386–408.
- B. Widrow, ``An Adaptive `Adaline' Neuron Using Chemical `Memistors','' Stanford Electronics Laboratories Technical Report 1553-2, October 1960.
- Minsky, Marvin; Papert, Seymour (1969). Perceptrons: An Introduction to Computational Geometry. MIT Press.
- D. E. Rumelhart, G. E. Hinton, and R. J. Williams. 1986. Learning internal representations by error propagation. In Parallel distributed processing: explorations in the microstructure of cognition, vol. 1, David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group (Eds.). MIT Press, Cambridge, MA, USA 318-362.
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, november 1998

References

- Krizhevsky, Alex & Sutskever, Ilya & Hinton, Geoffrey. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Neural Information Processing Systems. 25. 10.1145/3065386.
- Kurt Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks, 4(2), 251–257.
- John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research, 12(Jul):2121–2159, 2011.
- Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. 2010. Why Does Unsupervised Pre-training Help Deep Learning?. J. Mach. Learn. Res. 11 (March 2010), 625-660.
- Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013).
- Siegelmann, H.T. (1995). "Computation Beyond the Turing Limit". Science. 238 (28): 632–637
- http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/