Overview

I Artificial Intelligence

II Problem Solving

III Knowledge, Reasoning, Planning

IV Uncertain Knowledge and Reasoning

V Machine Learning

- 19. Learning from Examples
- 20. Learning Probabilistic Models
- 21. **Deep Learning**
- 22. Reinforcement Learning

VI Communicating, Perceiving, and Acting

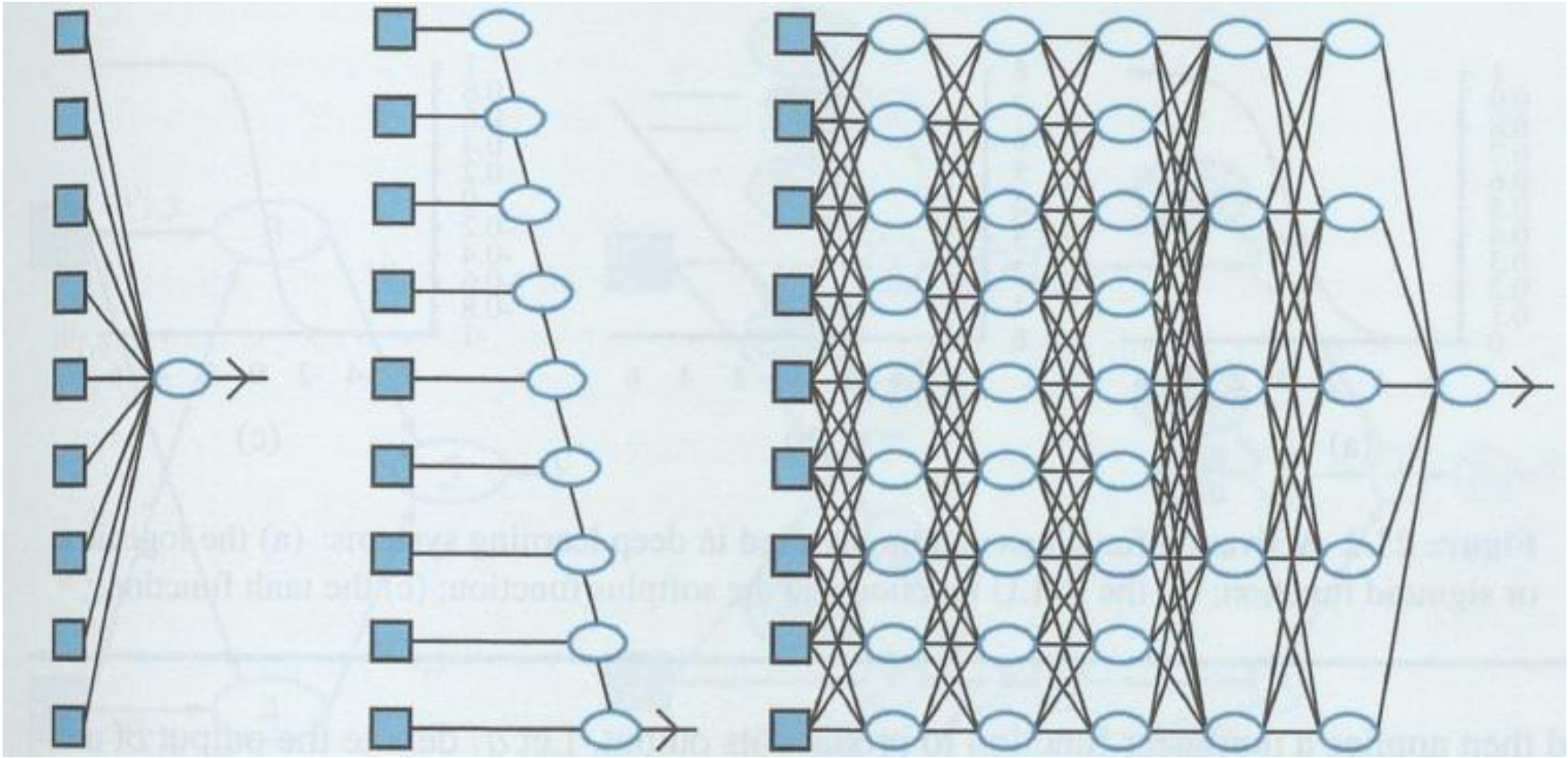
VII Conclusions

Deep Learning

- Simple Feedforward Networks
- Computation Graphs for Deep Learning
- Convolutional Neural Nets
- Learning Algorithms
- Generalization
- Recurrent Neural Networks
- Unsupervised Learning and Transfer Learning
- Applications

Expressiveness of Deep Learning Networks

- Linear or logistic regression or Naive Bayes can handle a large number of input variables, but they do not interact with each other (left)
- Decision lists or decision trees allow for long computation paths, but only for a small fraction of the possible input values (middle)
- Simple feedforward networks have long computations paths with high interactions (right)



Simple Feedforward Networks

- Each node within a network is called a unit (neuron) and has typically continuous values as input and output
- A unit calculates the weighted sum of its inputs and applies a nonlinear function to produce its output

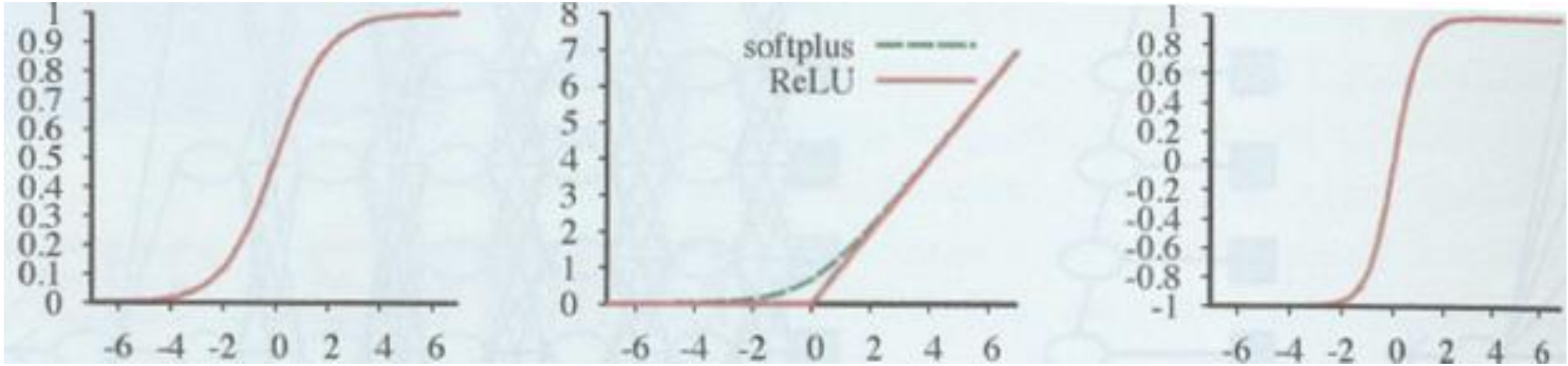
$$a_j = g_j(\sum_i w_{i,j} a_i) \equiv g_j(in_j)$$

  - a_j = output of unit j;
  - $w_{i,j}$ = weight of link from unit i to j
  - g_j = nonlinear activation function
  - in_j = is the weighted sum of the inputs to unit j
- In vector form: $a_j = g_j(\mathbf{w}^T\mathbf{x})$
  - **w** = vector of weights leading into unit j (including weight $w_{0,j}$auf a dummy unit 0 with the fixed value +1)
  - **x** = vector of inputs to unit j (including the +1)

Activation Functions

Activation functions must be **nonlinear**, otherwise network represents linear functions only

- **Universal approximation theorem**: A network with just two layers, one linear, one nonlinear, can represent arbitrary functions
  - Idea: Model continuous function piecewise with many „bumps“ similar to table-look-up
- Common activation functions:
  - **Logistic** or **sigmoid** function: $\sigma(x) = 1 / (1 + e^{-x})$ // left
  - **ReLU** function (rectified linear unit): $ReLU(x) = max(0,x)$ // middle
  - **Softplus** function, a smooth version of ReLU: $softplus(x) = log(1+e^{x})$ // middle
  - **Tanh** function: $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ with range (-1,+1); [tanh(x) = 2 σ(2x) - 1] // right



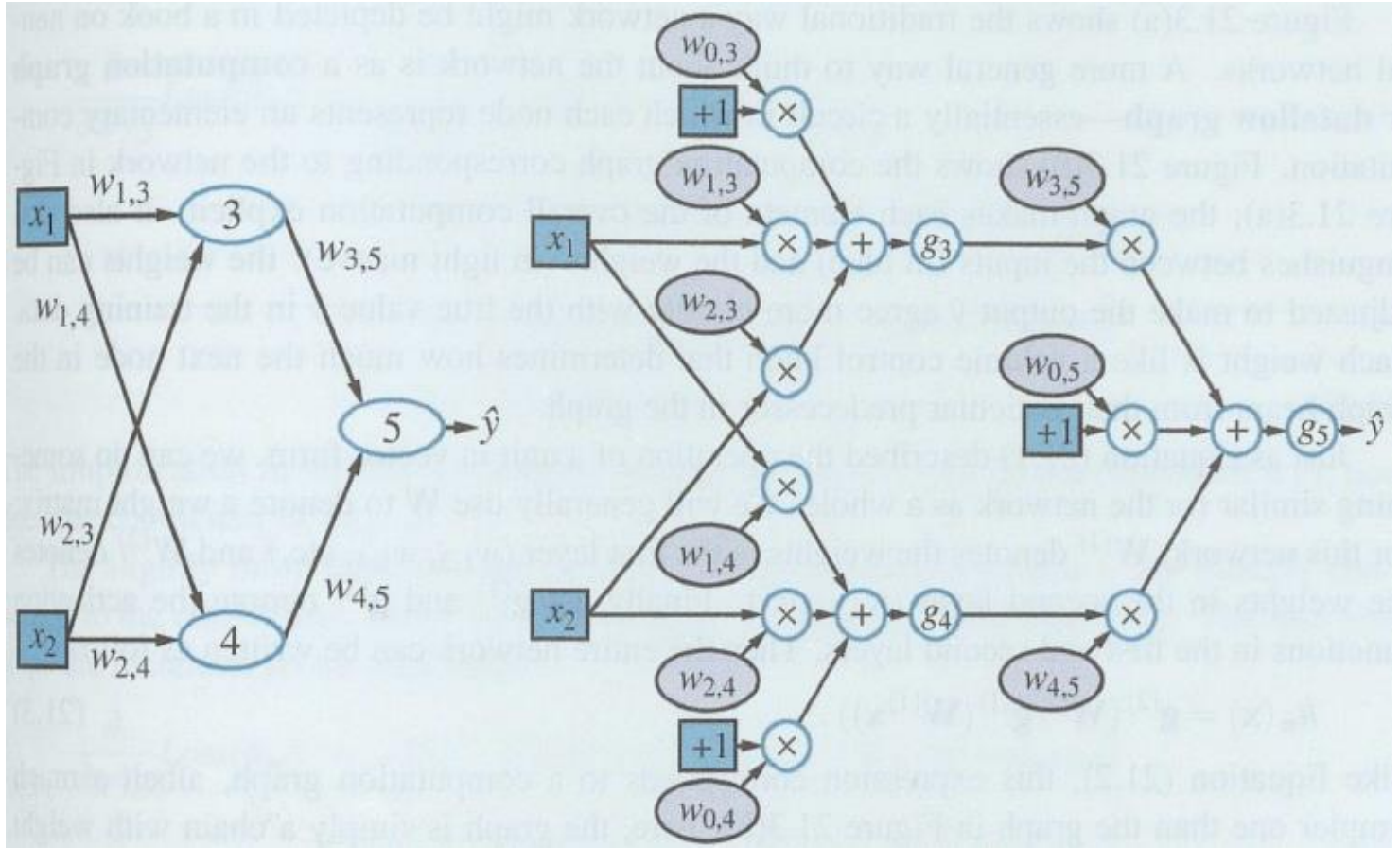
Properties of Activation Functions (2)

- Monotonically nondecreasing
- Have **derivatives g‘**
- Derivatives g‘ are nonnegative.

Networks

- Coupling multiple units together into a network creates a complex function $h_w(x) = \hat{y}$
- Composition of algebraic expressions represented by the individual units
- Example network for a function $\hat{y}$:
  - Above: Algebraic expression
  - Left: Usual graphical representation
  - Right: Unpacked **dataflow** graph (inputs x_i blue, weights $w_{i,j}$ gray)
  - Vector form: $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{g}^{(2)}(\mathbf{W}^{(2)}\,\mathbf{g}^{(1)}(\mathbf{W}^{(1)}\,\mathbf{x}))$

$$\begin{aligned}\hat{y} &= g_5(in_5) = g_5(w_{0,5} + w_{3,5}\,a_3 + w_{4,5}\,a_4)\\ &= g_5(w_{0,5} + w_{3,5}\,g_3(in_3) + w_{4,5}\,g_4(in_4))\\ &= g_5(w_{0,5} + w_{3,5}\,g_3(w_{0,3} + w_{1,3}\,x_1 + w_{2,3}\,x_2)\\ &\qquad + w_{4,5}\,g_4(w_{0,4} + w_{1,4}\,x_1 + w_{2,4}\,x_2))\end{aligned}$$



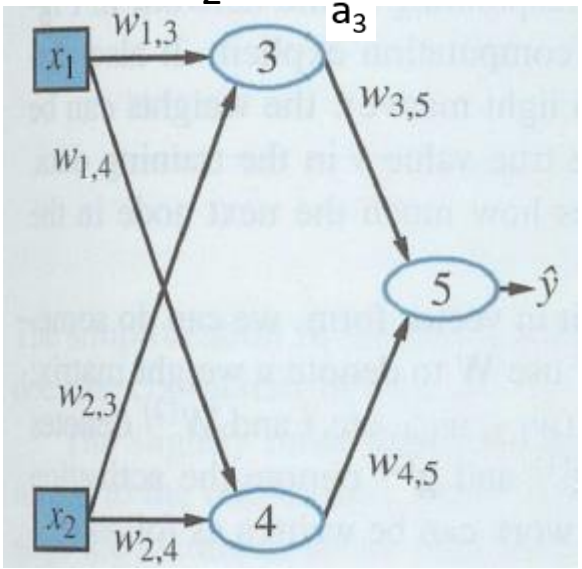
Gradients and Learning

- Linear classification with logistic regression used gradient descent for learning a hypothesis
- Same technique for Deep Learning:
  - Calculate the gradient of the loss function with respect to the weights and adjust the weights along the gradient direction to reduce the loss
  - Loss = Difference between observed and expected value, e.g. squared error L_2-loss
  - **L_2-loss**: Loss (h_w) = $L_2(y,hw(x)) = ||y-h_w(x)||^2 = (y-\hat{y})^2$
  - Gradient for a weight of the output layer, e.g. $w_{3,5}$:

$$\frac{\partial}{\partial w_{3,5}} \text{Loss}(h_w) = \frac{\partial}{\partial w_{3,5}} (y-\hat{y})^2 = -2(y-\hat{y})\, g'_5(in_5)\, a_3$$

  - Gradient for a weight of the hidden layer, e.g. $w_{1,3}$:

$$\frac{\partial}{\partial w_{1,3}} \text{Loss}(h_w) = -2(y-\hat{y})\, g'_5(in_5)\, w_{3,5}\, g'_3(in_3)\, x_1$$



➢ Output loss proportional to: **error** = $(y-\hat{y})$, **gradient** = $g'_5(in_5)$ and **unit value** = a3

➢ Hidden loss proportional to: **error**, **2 gradients** [$g'_5(in_5)$, $g'_3(in_3)$], **weight** ($w_{3,5}$), **unit value** (x1)

➢ **Vanishing gradient problem** for many hidden layers, if gradients are close to zero

Backpropagation

- The program propagates the error (loss) at the output units back to all hidden units
- To reduce the loss, the weights for an unit are changed, depending on the error, the gradient of the error with respect to the weight and the value of the unit with which the weight is multiplied (if this unit has e.g. a zero value, it makes no sense to change the weight)
- For hidden units, gradients are multiplied, causing the problem of vanishing gradients in deep networks with many layers
  - Therefore, deep networks need mechanisms to avoid this problem

Automatic Differentation

- All gradients within a neural net are computed by **automatic differentiation**
  - This method calculates gradients for any numeric program
  - Backpropagation applies a reverse mode differentiation
  - Applies the chain rule from the outside in and gains effiency by dynamic programming when the network has many inputs and relatively few outputs
- All major packages for deep learning provide automatic differentiation
  - ➢ Users can experiment with different network structures, activation functions, loss functions and forms for composition without having to do lots of calculus to derive a new learning method for each experiment
  - ➢ This has encouraged **end-to-end learning**
    - ➢ Even if a task (like machine translation) can be composed from several trainable subsystems, the entire system is trainend in an end-to-end fashion from input/output pairs

Computation Graphs for Deep Learning

- Basic idea of deep learning:
  - Represent hypotheses as computation graphs with tunable weights
  - Compute the gradient of the loss function with respect to those weights
  - Tune the weights to fit the training data
- Recommendations for the computation graph (next slides):
  - Input layers (input encoding)
  - Output layers and loss function
  - Hidden layers

Input Encoding

- Reflect attributes of training data
  - Boolean attributes: Logical values „false“ and „true“ are mapped to 0 and 1 or -1 and 1
  - Numeric attributes:
    - Used as is
    - Scaled to fit within a fixed range (e.g. 0/-1 and 1) with e.g. logarithmic transformation
  - Image data, e.g. RGB image of size X x Y pixels with integer range {0, … 255}
    - Pixel adjacency matters, therefore array-like internal structures are used
  - Categorical data with > 2 values (e.g. restaurant type: French, Italian, Thai, Burger)
    - One-hot encoding: each category is represented like a separate Boolean attribute
    - If values represent a scale (e.g. very low, low, medium, high, very high), a mapping on integers might be more appropriate

Output Layers and Loss Function

- Output encoding y of raw data values similar to input encoding
- Predictions $\hat{y}$ of the net are compared with the output y to compute the loss
  - Till now: L_2 loss: Squared error loss
  - Alternate: Interprete the prediction as probability und use the negative log likelihood as the loss function (like maximum likelihood learning):
    - Minimize the sum of negative log probabilities of the examples:
      - $w^* = \text{argmin}_w - \sum_{j=1}^{N} \log P_w(y_j|x_j)$
    - Equivalent to minimize **cross entropy loss** (common term in deep learning literature)
      - Cross entropy: H(P,Q): kind of dissimilarity between two distributions P and Q
        - P: true distributions over all training examples P*(x,y), approximated by a sample
        - Q: predictive hypothesis $P_w(y|x)$: represented as probability
        - Minimize the cross-entropy $H(P^*(x,y), P_w(y|x))$ by adjusting the weights w

Output as Probability

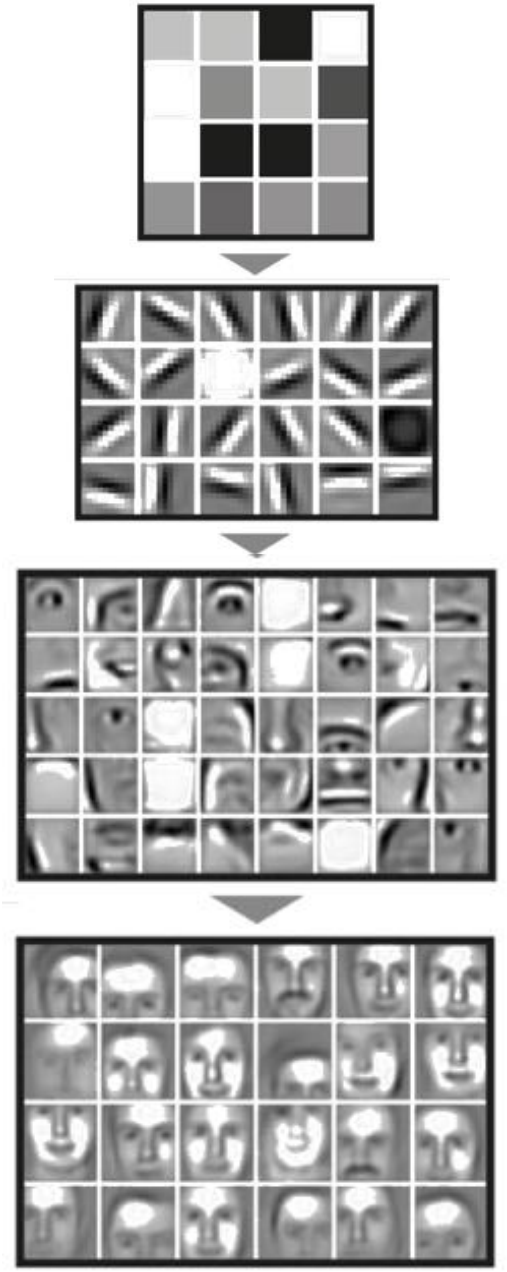
- For boolean classification problems, the sigmoid output layer can be directly interpreted as probability
- For multiclass classification problems, the sum of all predictions must add to 1
  - Achieved by **softmax layer**:
    - Input: d input values $\langle in_1, \ldots, in_d \rangle$
    - Output a vector of d numbers summing up to 1 $\langle in_1, \ldots, in_d \rangle$

      $$\text{softmax}(\mathbf{in})_k = e^{in_k} / \sum_{k'=1}^{d} e^{in_{k'}}$$
    - Example: Input: $\langle 5,2,0,-2 \rangle$; Output: $\langle 0.946, 0.047, 0.006, 0.001 \rangle$
    - Sigmoid function is equivalent to softmax of two values
- For regression problems, the target value is continuous, it is common to use a linear output layer $\hat{y}_j = in_j$ without any activation function (like classical linear regression)
- Other output layers are possibles

Hidden Layers

Each hidden layer in a neural network is a different representation of the input (the output layer is the final representation)

- Each layer tranforms the representation of the preceding layer
- Composition of all layers transforms the input in the (desired) output
- Each layer may make relatively simple transformations, which are fairly easy to learn by a local updating process
- Often the layers of deep networks discover **meaningful intermediate representations** (e.g. in images: edges, corners, ellipses, eyes, faces etc.)
  - Often intermediate representations are not meaningful for humans, but nevertheless reusable (e.g. word embeddings)
  - Can be reused in other networks for other tasks by exchanging just the last layers
- Often, deep and shallow networks learn better than wide ones
  - However, little theoretical understanding and much experimentation



Convolutional Neural Networks (CNN)

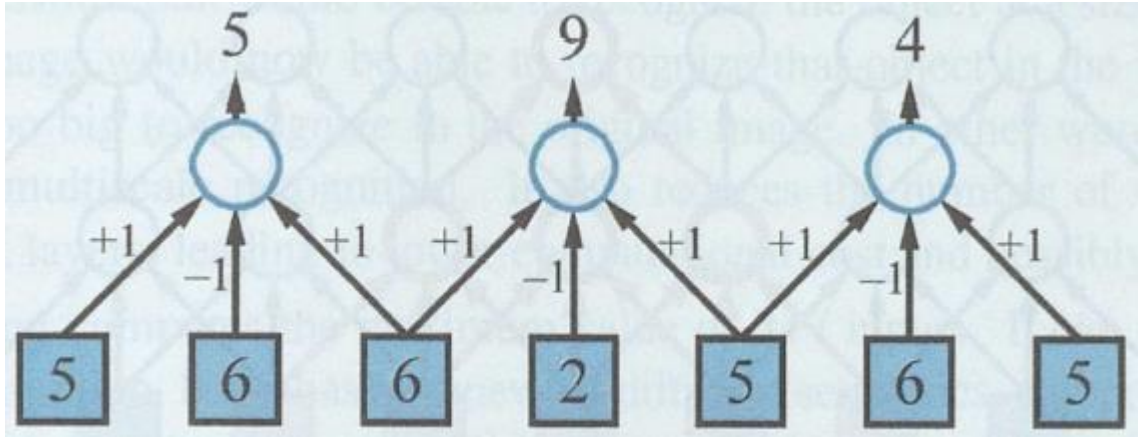
- Special features required for image interpretation
  - Image is represented as 2-dimensional array of pixels (with RGB-colors 3 dimensions)
  - **Adjacency of pixel** matters
    - Otherwise any pertubation of the pixel would have the same interpretation
    - Detection of local patterns
  - **Spatial invariance**
    - Local patterns have same interpretation in all regions of an image
- Solution: Convolutional neural networks
  - Contains spatially local connections (called **kernel)** replicated across multiple local regions
  - Application of the kernel to pixels of an image is called **convolution**

Convolution Operation

- Written as **z** = **x** * **k** (x: input, k: kernel, z: result)
- In one dimension:

$$z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}$$

- Example for one-dimensional convolution with kernel of size 3 [+1,-1,+1] and stride 2 applied to input vector [5,6,6,2,5,6,5] yields the output [5,9,4]:

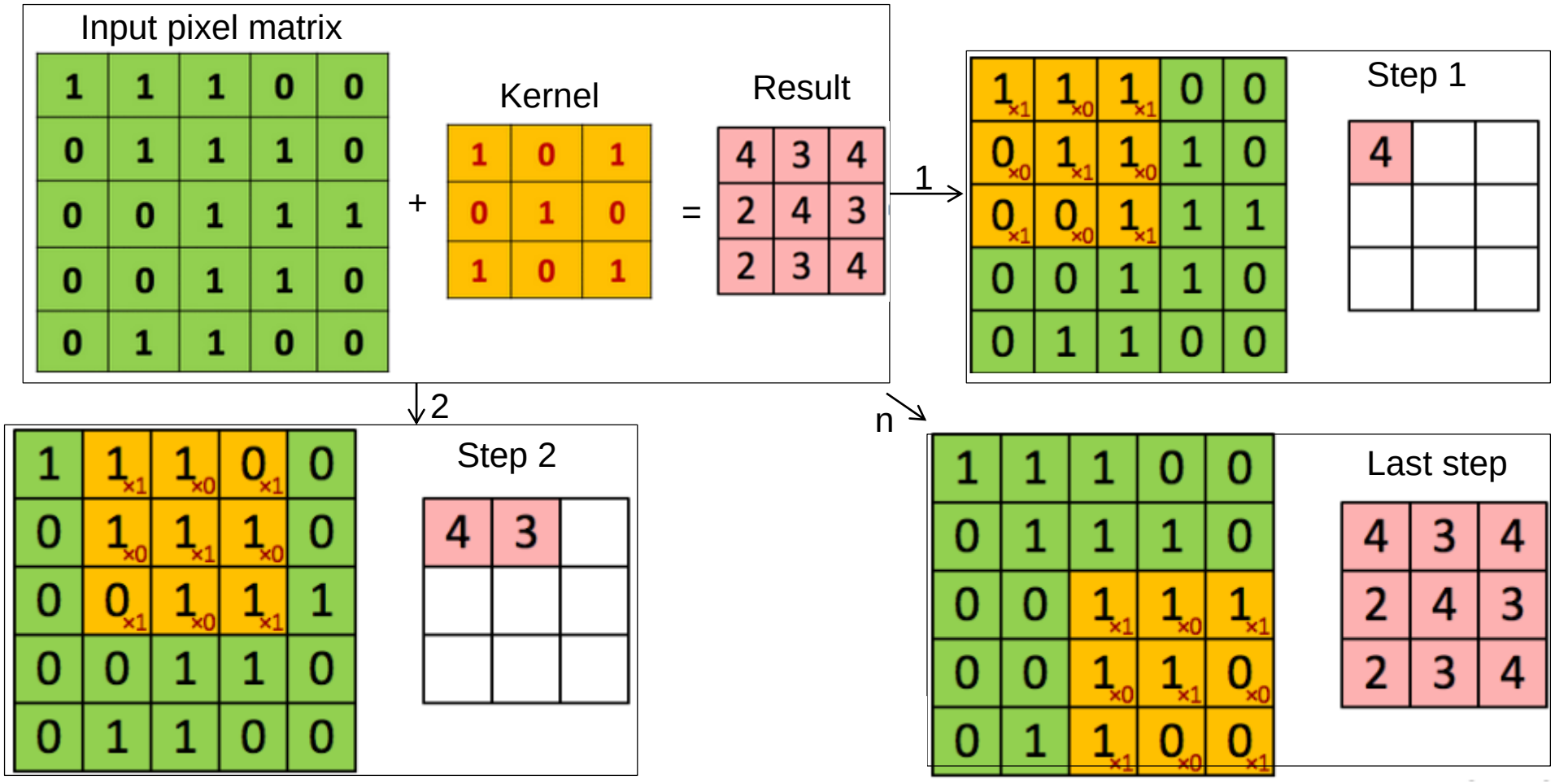


- Viewed (for illustration purposes only) as one matrix multiplication:

$$\begin{pmatrix} +1 & -1 & +1 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & -1 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & -1 & +1 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 6 \\ 2 \\ 5 \\ 6 \\ 5 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \\ 4 \end{pmatrix}$$

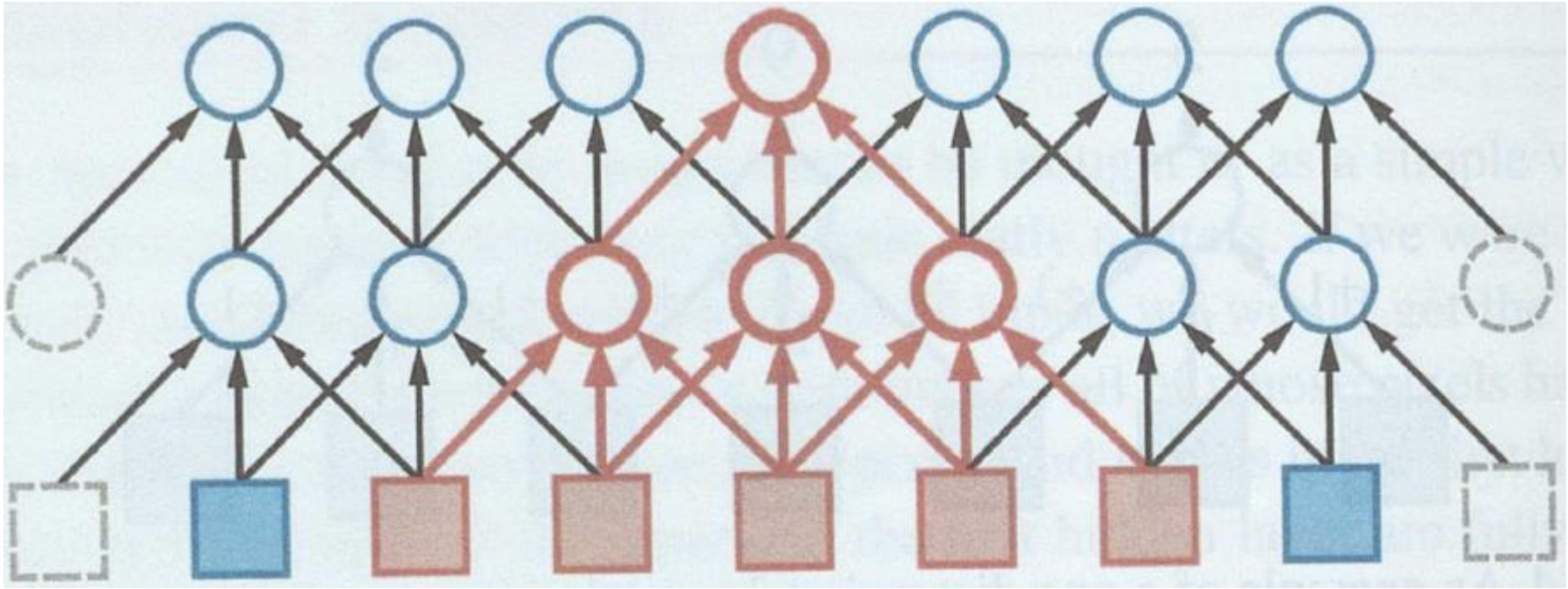
Example for Convolution Operator in Two Dimensions

- The **3x3 kernel** slides over the 5x5 input matrix and computes for each region an output (3x3)
- The pixels on which the kernel is centered have a **stride of 1**



Receptive Field

- The first two layers of a CNN for a 1D image with kernel size = and stride = 1
- Beyond the border, pixels can be filled with zeros (**padding**), so that the kernel remains applicable on the borders
- All input pixels contributing to a node in the hidden layer, are called its **receptive field** (in red)
  - Receptive field in first hidden layer in example is 3, in second hidden layer 5



Multiple Kernels and Pooling

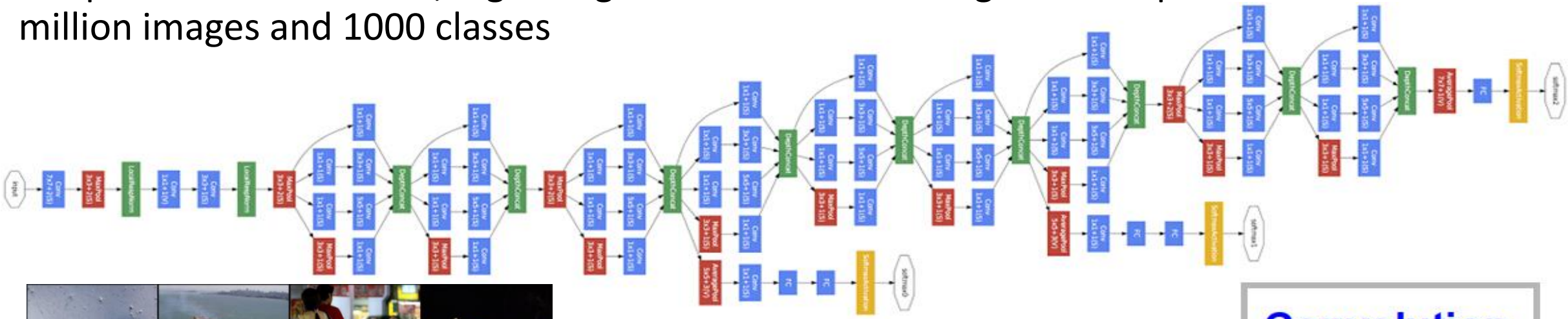
- With d kernels applied to the input, so the output will be d times larger
- Kernels are applied to the output of other kernels forming a hierarchy of kernels
  - Extracting local features of increasing complexity
- Usually, between two kernel operations a **pooling** operation is inserted
  - Pooling is similar, but simpler than a kernel and summarizes the values of its receptive field to just one value by (usually) taking the average or the maximum
    - Examples: Max-Pooling for 1-dimensional vector [5,9,4] = 9

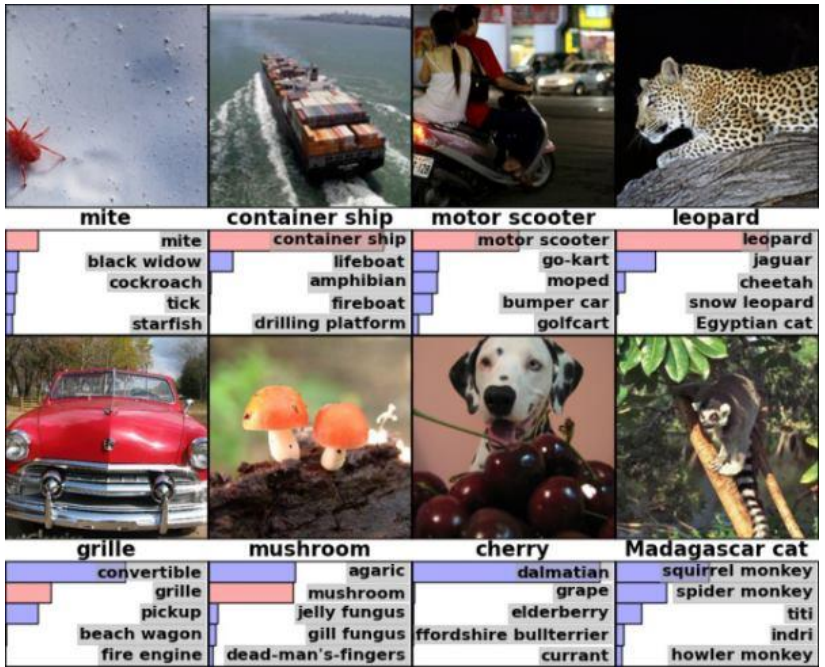
Max-Pooling for 2-dimensional vector $\begin{bmatrix} 4 & 0 \\ 3 & 7 \end{bmatrix}$ = 7

  - Pooling can be applied with different strides and reduces the dimension of a matrix

Softmax layer for Image Classification

- For classification, the final layer is usually a **softmax layer**, that assigns each class a probability
- Deep network structures, e.g. GoogLeNet: Winner of ImageNet competition 2014 with 1 million images and 1000 classes





Convolution
Pooling
Softmax
Other

Tensor Operation in CNNs

- The main operations in neural nets can be formulated elegantly as vector and matrix operations
  - Special cases of tensors (multidimensional array in any dimension)
  - For CNNs, tensors represent also adjacency of the data
  - Even multiple layers in a network can be reprented as tensor
- Special hardware for tensor operations
  - GPU (graphical processing unit)
  - TPU (tensor processing unit)
    - Google 3rd generation TPU pods have throughput equivalent to ca. 10 million laptops!
    - Can be exploited by high parallelism
      - e.g. process many image for training simultaneously (batch size)
  - Example: color images (RGB) with 256x256 pixel with batch size 64: 256x256x3x64 tensor
    - Application of 96 kernels of size 5x5x3 with stride 2: Output tensor: 128x128x96x64

Residual Networks

- Approach to building very deep networks that avoid the problem of vanishing gradients
  - In typical deep models a layer completely replace the information of the former layer:

    $$\mathbf{z}^{(i)} = f(\mathbf{z}^{(i-1)}) = \mathbf{g}^{(i)}(\mathbf{W}^{(i)}\mathbf{z}^{(i-1)})$$

  - If in each layer, some important information is lost, deep networks degenerate rapidly
- Key idea of residual networks: perturb the representation of a layer instead of replacing it:

  $$\mathbf{z}^{(i)} = \mathbf{g}_r^{(i)}(\mathbf{z}^{(i-1)} + f(\mathbf{z}^{(i-1)}))$$

  - g_r denotes the activation function of the residual layer
    - often consists of the ReLU activation function
- Function used to compute the residual is typically a neural network with one nonlinear layer combined with one linear layer (W and V are learned weight matrices):

  $$f(\mathbf{z}) = \mathbf{V}\,\mathbf{g}(\mathbf{W}\mathbf{z})$$

Learning Algorithms

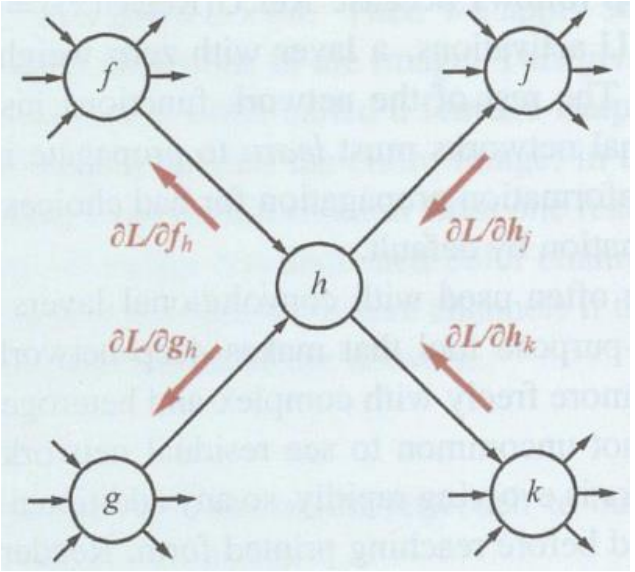
- Training a neural network consists of modifying the network‘s parameters (weights) to minimize the loss function on the training set
- Standard technique: Stochastic gradient descent (SGD) with learning rate α und Loss L(w) defined with respect to a minibatch of m examples chosen randomly at each step

$$w \leftarrow w - \alpha \nabla_w L(w)$$

- Improvements:
  - For large networks favor a small minibatch size m to reduce computational costs (and to escape small local minima)
    - Minibatch size often optimizes hardware parallelism in GPUs or TPUs
  - Learning rate α descreases over time
  - Near the global or a local optimum, the gradient has high variance. Solutions: (1) increase batch size as training proceeds; (2) use a **momentum** (running average of past gardients)
  - Avoid numerical instabilities due to overflow, underflow, rounding error
  - Terminate training process if error decreases very slowly

Computing Gradients in Computation Graph

- The error of an output node o (based an the error between expected and observed value) is back-propagated from o to a node h of the next hidden layer proportional to the quotient of weight w_{ho} and the sum of weights of all nodes in the hidden layer contributing to o
- If a node gets backpropagated errors from several nodes, they are simply summed up
- From these back-propagated errors the loss is computed for each node. This process stretches across all layers
- For each weight, a weight update is computed from the gradient of the loss function with respect to the weight
- If a weight is used for multiple purposes (weight sharing, e.g. in CNNs), each shared weight is treated as a single node with multiple outgoing arcs in the computation graph
- Illustration of back-propagation of gradient information in a network (forward path for computing the output from left to right; backward path for computing gradients from right to left:



Storage Requirements of Backpropagation

- For computing the gradients, the values of the nodes computed during forward propagation must be stored
  - The total memory costs for training the network is proportional to the number of units in the entire network
  - The run-time through a forward and backward path is linear in the size of the network

Batch Normalization

- Batch normalization improves the rate of convergence of stochastic gradient descent by replacing the values of a node z for m examples z_1 .. z_m in a minibatch by a normalized value of the z_i:

$$\hat{z}_i = \gamma \frac{z_i - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta$$

μ: mean value of z across the minibatch z_1 .. z_m ;

σ = standard deviation of z_1 .. z_m

ε = small constant to avoid dividing by zero

γ and β: learned parameters

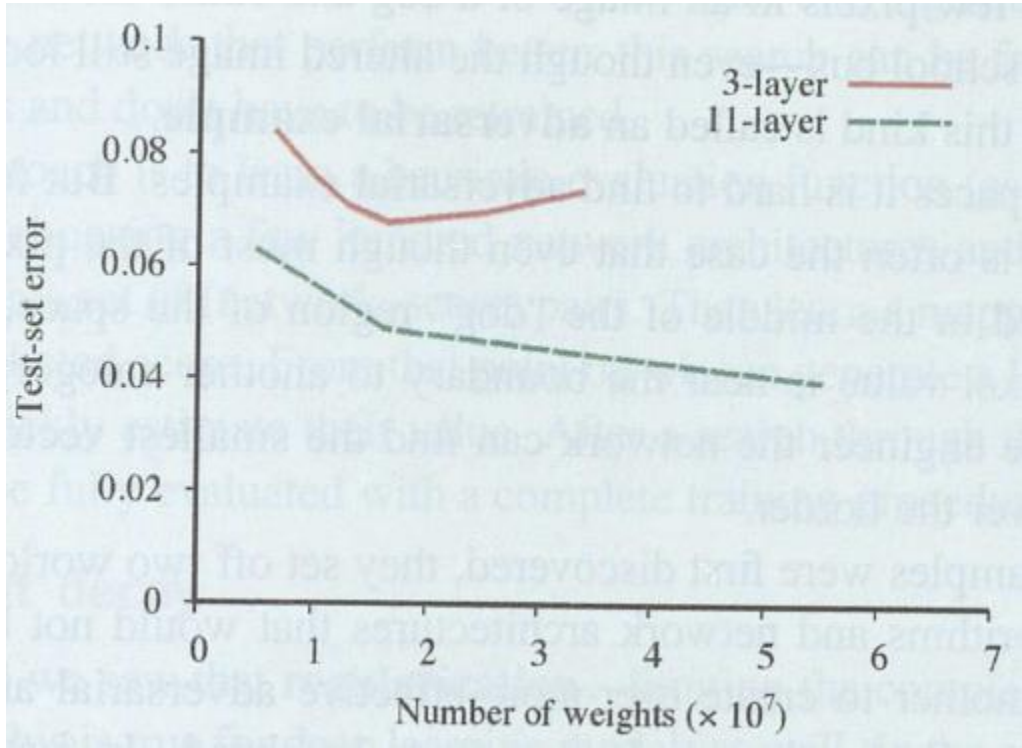
- Batch normalization is also useful for deep networks to prevent weights to become too small (similar to residual networks) and lessens the need for a clever initializiation of the network‘s weights
- The parameters γ and β are fixed after the training and may be node- or layer-specific

Generalization

- Optimization on training examples is prone to overfitting
- Techniques for generalization to new data:
  - Choosing the right network architecture
  - Penalizing large weights
  - Random pertubation of weights during training

Choosing a Network Architecture

- A great deal of effort in deep learning research goes into finding network architectures that generalize well
  - Different architectures for different tasks and data: Images, speech, text, videos etc.
    - Grid data (images): Convolutional neural nets
    - Sequential data (speech, text): Recurrent networks
  - Variations inside an architecture: number of layers, their connectivity and the types of nodes in each layer
  - Hyperparameter tuning (loss function, batch side, normalization, learning rate, …)
- Deeper networks are usually better than flat networks with similar number of weights:



Neural Architecture Search

- Trying all combinations of parameters usually impossible
  - Testing a neural architecture requires a long time, since a new neural net must be trained and tested
- Search strategies in the parameters space necessary
  - Genetic (evolutionary) algorithms popular with recombination of two networks and mutations (e.g. adding or removing a layer or changing a parameter value)
  - Hill Climbing
  - Other approaches: Reinforcement learning, Bayesian learning, Gradient descent, …
- Reducing the need to train a full network in each search step
  - Use part of the network, reduced versions, or a small number of batches
  - A* search with an evaluation function learned from a few hundred evaluated network architectures with (network score) pairs and test only the most promising architectures from A*

Success and Problems of Deep Learning

- Deep learning moduls outperform other models an high-dimensional tasks like image, video, speech
  - Very fast with a small number of computation steps (10 to 10^3) due to massive parallelism
  - Other learning approaches are good at low-dimensionsal tasks
- Unintuitive errors
  - Tend to produce discontinuous input-output mappings, i.e. small changes in input produce very different outputs
    - Changing a few pixels in an image may cause the network to classify a parrot as a bookshelf or a panda as a gibbon
  - Adversarial examples: Train neural networks to find slight pixel changes of an image to cause misclassifications by another network
    - Even possible without knowing the other network

Adversarial Examples (1)

Parrot (97,38%)

Bookshelf (99,12%)

Adversarial Examples (2)

$+0.007 \times$

$=$

Panda (57,7%)

Gibbon (99,3%)

Weight Decay

- To get a robust neural network, the output should not depend on just a few features with high weights
- **Weight decay** adds a penalty to the loss function proportional to the (squared) weight:

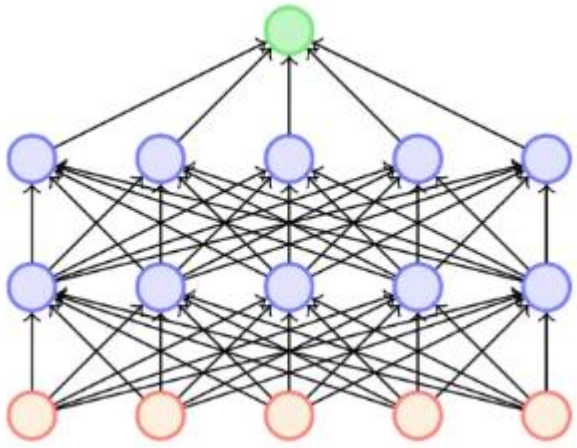
  $$\lambda \sum_{i,j} W_{i,j}^2$$

  with λ chosen between 0 (no weight decay) and a small value (e.g. 10^{-4})
- Weight decay decreases large weights, but does not prevent them
- Often beneficial to all kind of networks

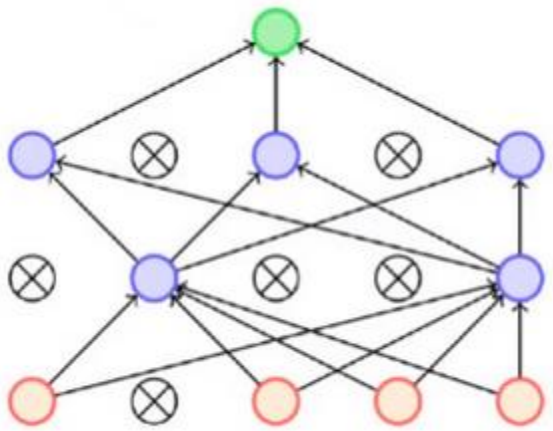
Dropout

- In each step of the training, a new version of the network is created by deactivating a randomly choosen subset of the neurons (see below)
  - Typical probability for a node in hidden layer is 50% to be dropped, for an input node 20%
- Each new version is trained with a different minibatch of examples
- The different versions share all weights, which are not dropped, but alter them during training
- All thinned networks can be viewed as an ensemble trained together in an efficient manner
- The final network uses all nodes and weigths, but weighted by the probability they appeared in the ensemble

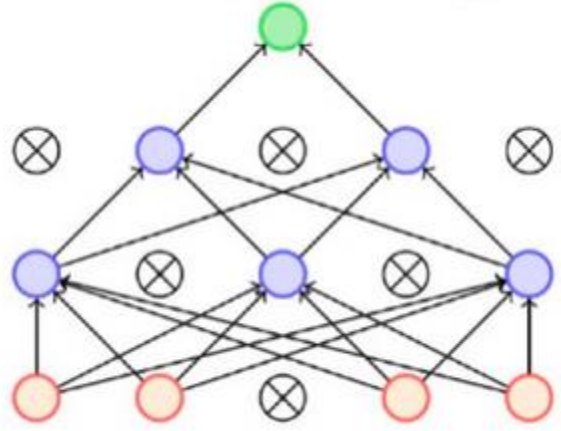
from:
https://prvnk10.medium.com/ensemble-methods-and-the-dropout-technique-95f36e4ae9be



Orginal architecture



Random drop out (1)



Random drop out (2)

Effectivity of Dropout

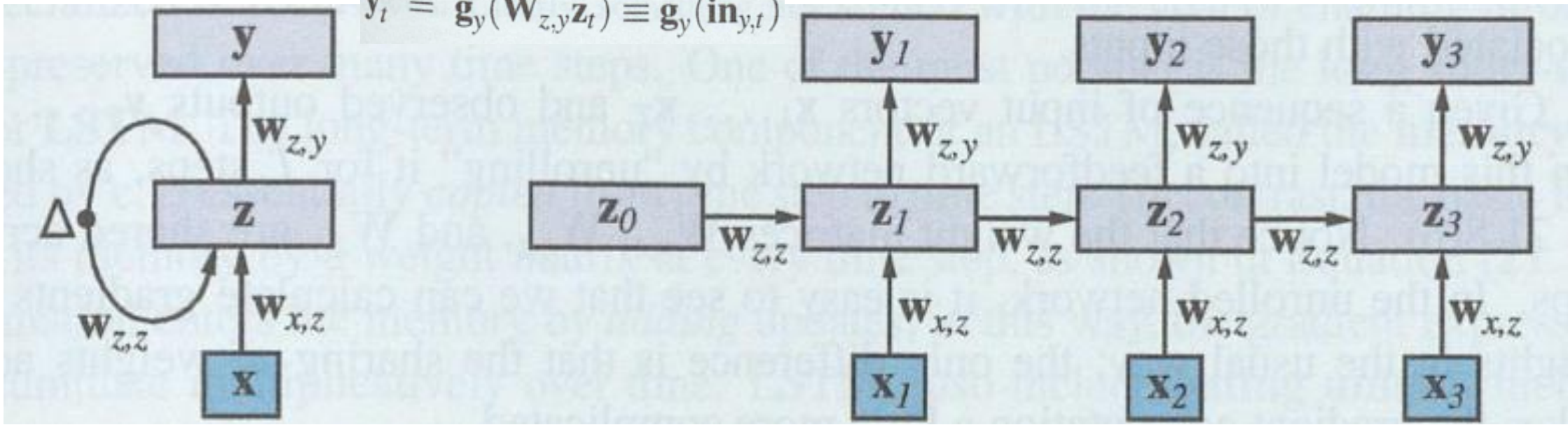
- By introducing noise, the model is forced to become robust to noise
- Dropout approximates the creation of a large ensemble of thinned networks
- Hidden units trained with dropout learn to be useful with different sets of other hidden units
  - e.g. for recognizing an eye, a hidden unit should recognize it from different low level feature (i.e. other units)
- If there is one dominant feature for classification of a picture (e.g. a special kind of nose), dropout forces the network to classifiy the picture also from other features
  - Dropout forces a model to learn multiple robust representations for each input
- Dropout makes it more difficult to fit the training set, thus a larger model and more training iterations are usually necessary
- Dropout is usually more effective than e.g. weight decay

Recurrent Neural Network (RNN)

- RNNs allow cycles in the computation graph (unlike feedforward networks)
  - Each cycle has a delay, taking as input the old value of the unit
  - RNN have a kind of internal state or memory
  - Similar to Markow assumption (hidden state depends on last hidden state and evidence)

$$\mathbf{z}_t = f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t) = \mathbf{g}_z(\mathbf{W}_{z,z}\mathbf{z}_{t-1} + \mathbf{W}_{x,z}\mathbf{x}_t) \equiv \mathbf{g}_z(\mathbf{in}_{z,t})$$

$$\hat{\mathbf{y}}_t = \mathbf{g}_y(\mathbf{W}_{z,y}\mathbf{z}_t) \equiv \mathbf{g}_y(\mathbf{in}_{y,t})$$



Basic Structure (z: State, x: Input, y: output)

Same network unrolled over 3 time steps to create a feedword net with weight sharing in all time steps

Training a Basic RNN

- Gradient expression for training is recursive (depending on the last time step)
  - Backpropagation through time (linear in the size of the network)
  - Suffers inherently from vanishing oder exploding gradient problem
    - Not really useful for practical purposes

Long Short-Term Memory RNN (LSTM)

- Very successful extension of RNN
- Memory cell is essentially copied from time step to time step
  - Instead of multiplying the memory cell by a weight matrix at every time step in RNNs
- New information enters the memory cell c by adding updates
- LSTMs include three gating units to control the flow of information in the memory cell:
  - **Forget gate f:** determines if each element of the memory cell is rememberd (copied) or forgotten (set to zero)
  - **Input gate i:** determines if each element of the memory cell is updated additively by new information form the input vector
  - **Output gate o:** determines if each element of the memory cell is transfered to the short-term memory z, from which the output of the network is computed (similar to RNN)

LSTM Cell

f: Forget gate

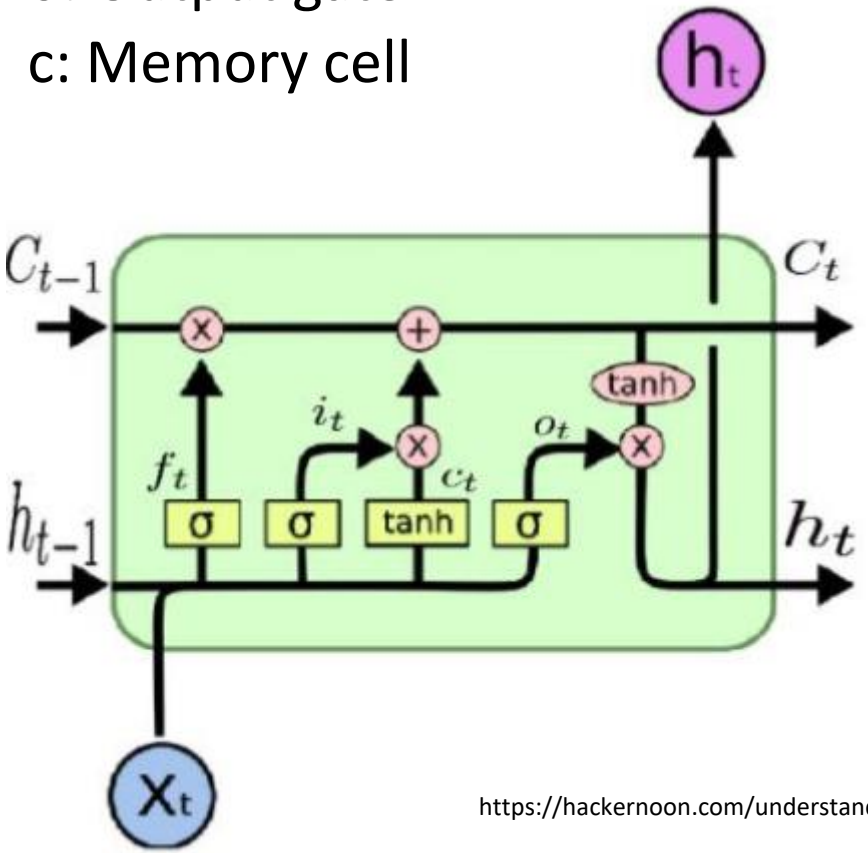
i: Input gate

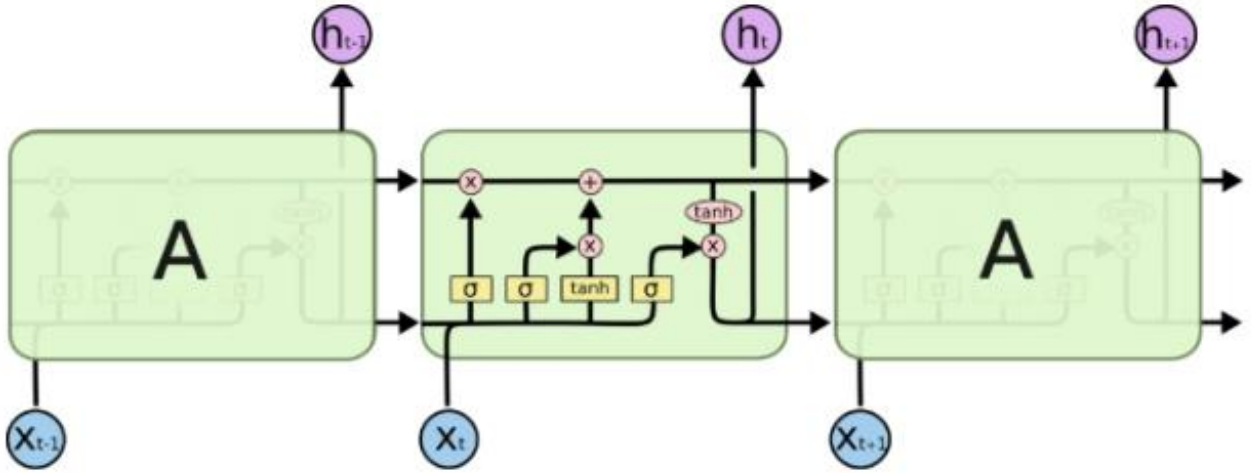
o: Output gate

c: Memory cell

x: Input

h: Output





https://hackernoon.com/understanding-architecture-of-lstm-cell-from-scratch-with-code-8da40f0b71f4

Unsupervised Learning and Transfer Learning

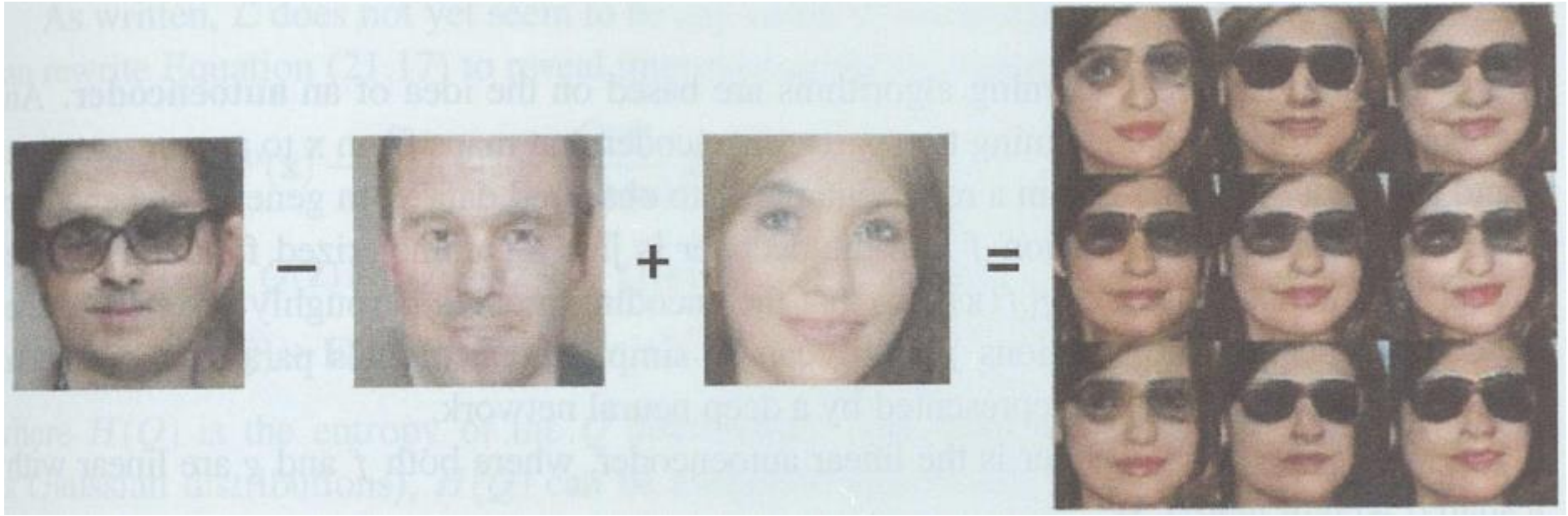
- Goal: Reducing the amount of training data necessary
  - **Unsupervised learning:** Learning from unlabeled data
    - Generative Models: Produces realistic text (GPT3) images, audio, video
  - **Transfer learning:** Reuse models for similar tasks
    - Adapt the model to a new task by a few new labeled training examples
  - **Semisupervised learning:** Require some labeled examples and improve by also studying unlabed examples

Unsupervised Learning

- Supervised Learning: Given training set of inputs x and outputs f(x) = y, a hypothesis h is learned, that approximates f
- What can be learning without labels y from the inputs x?
  - Outside of Deep Learning: Clustering Algorithms to group similar inputs to clusters
  - Within in Deep Learning:
    - Learn new representations (e.g. new features in images or new word representations)
    - Learn a generative model, typically in form of a probability distribution from which new samples can be generated

Learning Unobserved Variables

- Goal: Learn a joint model Pw(x,z)
  - z is a set of latent unobserved variables that represent the context of x in some way
  - z not predefined
    - Candidate variables for pictures of handwritten text: thicknes of pen strokes, ink color, background color etc.
    - Candidate variables for faces: Gender, glasses, etc.



Probabilistic PCA: A Simple Generative Model

- PCA (Principal Component Analysis):
  - Input: A collection of points in n dimensions
  - Output: A representation of the points in new linearly uncorrelated dimensions (principal components) ordered by their capability to retain the variance among the points
    - Typically used for dimension reduction by taking only the k best new dimensions
- Probabilistic PCA: Generalization of PCA
  - The new dimensions z are used to generate the data points (examples) x by applying a weight matrix W and adding some noise σ (N: normal distribution)
    - $P(z) = N(\mathbf{z:0,I})$
    - $P_w(\mathbf{x}|\mathbf{z}) = N(\mathbf{x}; \mathbf{Wz}, \sigma^2\mathbf{I})$
    - Find W maximizing the likelihood of the data: $P_w(\mathbf{x}) = \int P_w(\mathbf{x,z})d\mathbf{z} = N(\mathbf{x;0,WW}^T + \sigma^2\mathbf{I})$
  - The weights W can be learned by an EM-algorithm or by gradient methods

Autoencoders

- Model containing two parts:
  - Encoder that maps from observed data x to a representation $\hat{z}$
  - Decoder that maps from $\hat{z}$ back to x
- If encoder is a parametrized function f and decoder a parametrized function g, then $\mathbf{x} \approx g(f(\mathbf{x}))$
- **Linear autoencoder**, if f and g are linear, e.g. with a shared weight matrix **W**:
  - $\hat{\boldsymbol{z}} = f(\mathbf{x}) = \mathbf{W}\mathbf{x}$
  - $\mathbf{x} = g(\hat{\boldsymbol{z}}) = \mathbf{W}^T\hat{\boldsymbol{z}}$
  - Can be trained by minimizing the squared error $\sum_j ||\mathbf{x}_j - g(f(\mathbf{x}_j))||^2$
  - Choose a low-dimensional $\hat{\boldsymbol{z}}$ to reconstruct high dimensional data x
  - Very similar to PCA
- Variational autoencoder: Use more complex kinds for generative models

Deep Autoregressive Models

- **Autoregressive Models** predict each elemet x_i of the data vector **x** based on other elements of the vector
  - Most common application on time series data, i.e. predict x_t given x_{t-k},… x_{t-1}
    - Example n-gram model of letter or word sequences
  - Classical autoregressive models with real valued variables use a linear regression model finding a maximum likelihood solution
- **Deep autoregressive models** replace the linear model by an arbitrary deep network with a suitable output layer depending, whether x_i is discrete or continuous
  - Applications for speech generation (e.g. Wave net, 2016) and text generation (e.g. GPT3, 2020)

Generative Adversarial Networks (GAN)

- GAN: Pair of networks (generator, discriminator) that combine to form a generative system
  - **Generator**: Generates virtual examples z from a distribution $P_w(x)$
  - **Discriminator**: Is trained to classify inputs x as real (drawn from the training set) or fake (created by the generator)
- Both networks are trained simultaneously
  - Generator learns to fool the discriminator
  - Discriminator learns to accurately separate real from fake data
  - Competetion like in game theory
  - Idea: In the equilibrium, the generator reproduces the training distribution perfectly such that the discriminator cannot perform better than random guessing
- GANs very successful for image generation tasks:
  - e.g. creates photorealistic high resolution images of people who never existed
  - Also successful for generating adversarial examples (misclassified images)

Unsupervised Translation

- GANs can be used for translation
  - The generator learns a translation from (x,y) pairs of translations by professional human translators (e.g. from documents from international organizations)
  - The discriminator learns to classify a translation as human or machine generated

Transfer Learning

- **Transfer learning:** Experience with one learning task helps an agent better on another task
  - e.g. Tennis-players learn squash easier
  - Pilots can transfer their flying-capabilities to new airplanes
  - But mechanisms for human transfert learning unknown
- **Transfer learning in neural nets:**
  - Copy the weights learned for task A to a network that will be trained for task B
  - Adjust the weights with new training examples for task B as usual
  - Variants:
    - Use a smaller learning rate, if the two tasks are very similar
    - Reuse the first few layers in a deep network for a new task by freezing them
- Transfer learning for multiple tasks simultaneously

Successful Applications for Transfer Learning

- Image processing:
  - Low level features like edges are quite universal useful, therefore most image neural nets are pretrained
  - Usually pretraining with publicly available data sets like Imagenet or specialized data sets for a particular task (recognising people, medical image interpretation etc.)
- Natural language processing:
  - Pretrained language models for vocabulary (Word-embeddings) and syntax of every day language like BERT or RoBERTa etc.
    - Must be fine-tuned with specialized vocabulary for the new domain
- Autonomous car driving:
  - Pretrained model with data from traffic simulations (also for rarely occuring events)
  - Fine-tuning with real world data

Transfer Learning and Multitask Learning

- Multitask learning is a form of tranfer learning, in which we simultaneously train a model on multiple purposes to support generalisations (instead of specializing for one task)
  - Model might be better usuable for transfer learning for specialized domains
  - e.g. natural language system with multiple objectives like
    - part-of-speech tagging, document classification, language detection, word prediction, sentence difficulty modeling, plagiarism detection, sentence entailment, question answering etc.
  - Such a system is more likely to create a common representation for NLP tasks

Applications: Vision

- First break-through in the image net competion task (1 200 000 image in 1000 categories):
  - AlexNet deep learning system: top5 score with ca. 15% error compared to ca. 25% before
    - 5 Convolutional and Max Pooling layers, 3 fully connected layers, ReLU activation function, GPUs for training the 60 million weights
  - Currently: Top5 score (correct category in best 5 predictions) < 2% error rate (humans: 5%)

Applications: Natural Language Processing

- End-to-End deep learning system for **machine translation** much better than pipeline systems
  - 2016: error reduction of 60%
  - 2020: machine translation for similar language like French and English approaches human performance and usable for most language pairs
- Great success representing word embeddings with deep learning systems (e.g. BERT 2018)
- GPT3 (2020) for plausible text generation
  - Examples taken from OpenAI GPT3
  - https://machinelearningknowledge.ai/openai-gpt-3-demos- to-convince-you-that-ai-threat-is-real-or-is-it/

```
Q: What is your favorite animal?
A: My favorite animal is a dog.

Q: Why?
A: Because dogs are loyal and friendly.

Q: What are two reasons that a dog might be in a bad mood?
A: Two reasons that a dog might be in a bad mood are if it is hungry or if it is hot.

Q: How many eyes does a giraffe have?
A: A giraffe has two eyes.

Q: How many legs does a frog have?
A: A frog has four legs.

Q: Are there any animals with three legs?
A: No, there are no animals with three legs.

Q: Why don't animals have three legs?
A: Animals don't have three legs because they would fall over.
```

GPT-3 can impressively lower down the tone of an offensive sentence to a cordial tone:

- Original: "Stop fucking sending me emails. I hate it."
- Toned down: "Do not send me emails"

- Original: "I need to talk to you ASAP, call me"
- Toned down: "Kindly contact me when you read this, it's urgent."

- Original: "As you can read in my previous email, I already told you that we won't be able to make it."
- Toned down: "There might be a misunderstanding. Unfortunately, I don't think we'll be able to make it this time."

Applications: Reinforcement Learning

- Reinforcement Learning (RL): Agent learns from a sequence of reward signals to optimize future rewards
  - Agent might learn a value function, a Q-function, a policy etc.
  - These functions can be represented as deep neural net, e.g.:
    - Go: estimation of game position
    - Atari video games: Q-function that map a raw image with reward signal (game score)
- In majority of Atari Games, Go, chess and many other games, deep RL systems play at superhuman level
  - Training by self play and nearly unlimited training examples
- Nevertheless, it is difficult to get good performance and trained systems may behave very unpredictably
  - Rarely applied in commercial applications