# Neural Networks

Jan Chorowski
Instytut Informatyki
Wydział Matematyki i Informatyki Uniwersytet Wrocławski

2018

# Final exam

Please don’t forget about it:

- **Tuesday 5.2.2019 9:30 in 119**

It should last for about 60-90 minutes (but we will have time until 12:00)

# Projects

- We can grant extension until **14.2.2018**. You need to ask me or mr Smajek for it.
- Please consult with me or mr Smajek if you have problems with them ASAP.
- To submit projects:
  - for ICLR repro challenge, please follow the rules at https://reproducibility-challenge.github.io/iclr_2019/ but send us the reports first for approval
  - For other projects, please provide a similar report
  - In both cases, please consider submiting a PR with the report against the nn_assignments github for archiving

# What to do after Neural Nets?

- Artificial Intelligence course by P. Rychlikowski
- Eksploracja Danych by P. Lipiński
- My seminars:
  - Statistics and Neural Networks
  - Probabilistic Graphical Models
- And a good summer school (if you get accepted I'll try to find you money from the University for it): https://www.eeml.eu/
- I do have research topics, that can morph into diplomas and theses, if you are interested please reach out!
- My group (Pracownia Inteligencji Obliczeniowej, PIO) meets weekly to discuss papers and research ideas – let me know if you want to be notified about them.

# Learning materials

Most lectures have accompanying Notebooks with explanations. Additional meterials:

- For Linear Models, Learning Theory, SVMs, K-Means, EM and PCA you can consult Stanford's CS229 handouts by A. Ng: https://see.stanford.edu/Course/CS229
- For Deep Neural Nets and Convnets you can consult lecture notes for Stanford's CS231 http://cs231n.stanford.edu/
- For more on NLP topics: http://cs224d.stanford.edu/
- For more info on LSTMs you can consult Chris Olah's blog and distill.pub: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ https://distill.pub/2016/augmented-rnns/
- Last but not least – the Goodfellow and Bengio Deep Learning book: http://www.deeplearningbook.org/

# Topic 1 - Learning

- We speak about learning when we want to automatically determine the relations present in the data.
- Thus learning starts with **DATA**
  - Implementation of an algorithm is not learning
  - Choosing the parameters of a program to match the data is learning
- The other part of learning is choosing a **family of functions** (hypotheses) from which we will choose the one matching the data
  - The larger the hypothesis space, the more data we need to have to choose the correct hypothesis
  - We need to restrict the set of hypotheses (introduce bias based on our knowledge about the problem) – reliably learning a function from the set of all functions is impossible!

# Learning – hypotheses

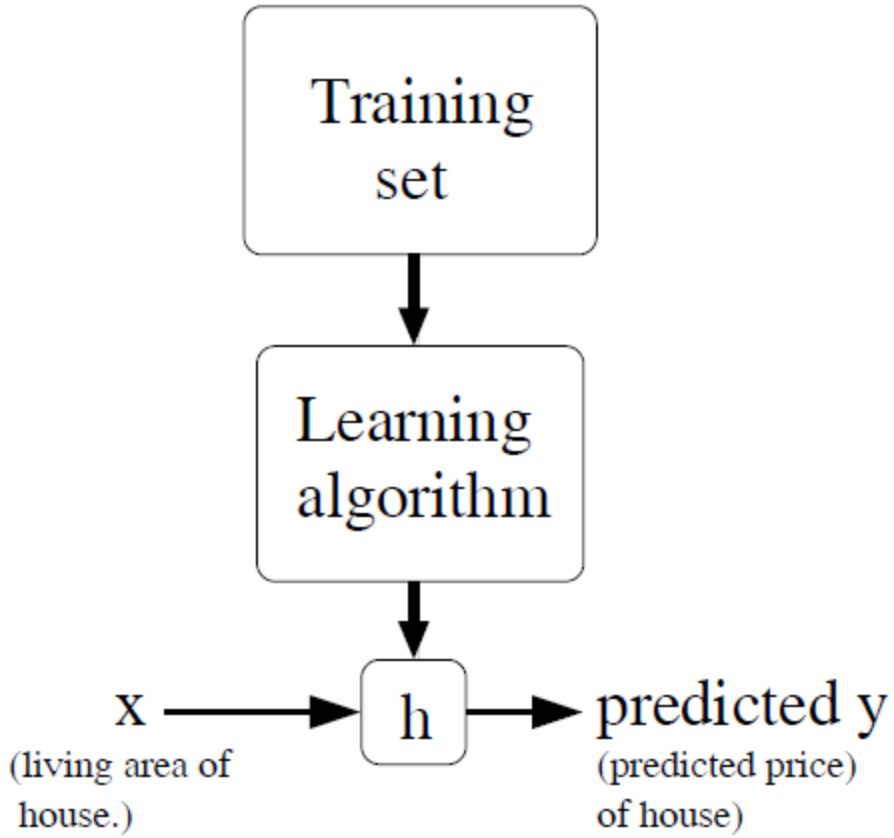
- During learning we choose a function (a model) from a family (the hypothesis space) based on a dataset.
- We choose it using **TRAINING DATA**, but we really want it to work on **UNSEEN TEST DATA**
- 2 sources of error:
  - **BIAS**: There is no function in the hypothesis space that faithfully represents the data
  - **VARIANCE**: The hypothesis space is so large and the data so scarce that we can't distinguish using the data a good function from a bad one.

# Intuitive example: fitting polynomials.

# Please remember that…

- Learning from data is:
  - Choosing a hypothesis space (e.g. neural nets)
  - Choosing a hypothesis goodness criterion (e.g. log-likelihood)
  - Choosing the best hypothesis (i.e. optimization, e.g. SGD)
- Two major problems:
  - Mismatch between data and hypothesis space
  - Too large hypothesis space
- Learning Theory (PAC and Statistical Learning Theory):
  - Tells us a bound on the **error rate on unseen (test) data** that depends on the **error rate on the training data**, the **size of the hypothesis space**, and the **amount of training data**.
  - In other words: When one has sufficiently many data and a sufficiently small hypothesis space, the TRAINING and TESTING error will be similar

# Learning


Training set
Learning algorithm
x
(living area of house.)
h
predicted y
(predicted price) of house)

# Types of learning

- Supervised:
  - The desired outputs (**labels)** are given
  - Data are **(input, output)** pairs
  - Goal is to learn the **input-output relation**
  - Examples:
    - Classification (discrete targets)
    - Regression (real-valued targets)
- Unsupervised:
  - No labels, just data points
  - Goal is to **describe** the data
  - Examples:
    - Clustering (find groups of closely related samples)
    - Dimensionality reduction
    - Find good latent codes for data
- Reinforcement (basics are in the AI course):
  - Feedback is given after a set of actions
  - E.g. learn to play a game based on its outcome only
  - Credit assignment problem: which actions were good, which were bad

# Learners we know

- Least squares regression:
  - Supervised learning
  - Data are $\{(\boldsymbol{x}^{(j)}, y^{(j)}), \in \mathbb{R}^n \times \mathbb{R}, j = 1..m\}$
  - Hypothesis space:

$$\boldsymbol{\Theta} \in \mathbb{R}^n \text{ are the parameters}$$

$$y \approx f(\boldsymbol{x}, \boldsymbol{\Theta}) = \sum_{i=1}^{n} \Theta_i x_i = \boldsymbol{\Theta}^T \boldsymbol{x}$$

  - Training criterion (which hypothesis is the best):

$$\sum_{j=1}^{m} \left(f\left(\boldsymbol{x}^{(j)}, \boldsymbol{\Theta}\right) - y^{(j)}\right)^2$$

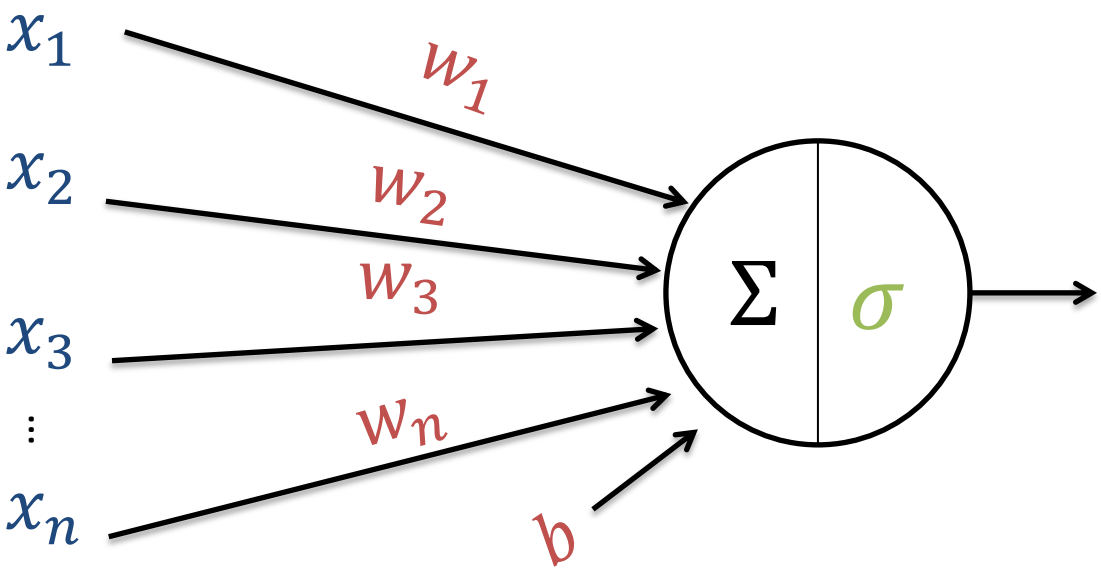
  - Learning algorithm (how to choose the best hypothesis): mathematical optimization:

$$\boldsymbol{\Theta}^* = \underset{\Theta}{\operatorname{argmin}} \sum_{j=1}^{m} \left(f\left(\boldsymbol{x}^{(j)}, \boldsymbol{\Theta}\right) - y^{(j)}\right)^2$$

# Artificial Neural networks

- Are a family of functions that take real-valued vectors as inputs and produce real-valued vectors as outputs
- Are pictured as a **NETWORK** (directed graph) of simple computing nodes (the **NEURONS**)
- The function of the NN is stored in:
  - The architecture (which neurons are connected)
  - Weights (how strong the connections are)

# The artificial neuron (perceptron)




Output:

$$y = \sigma(\sum w_i x_i + b)$$
$$= \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$$

- $x_i$ are the inputs
- $w_i$ are the weights and $b$ the bias
- Σ denotes the summation
- $\sigma$ is a (possibly nonlinear) activation function

**$w_i$, b are TUNABLE!!**

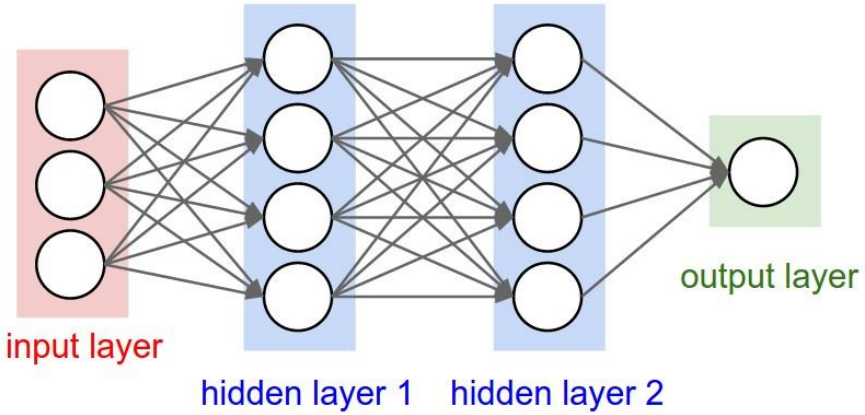
# The Artificial Neural Network

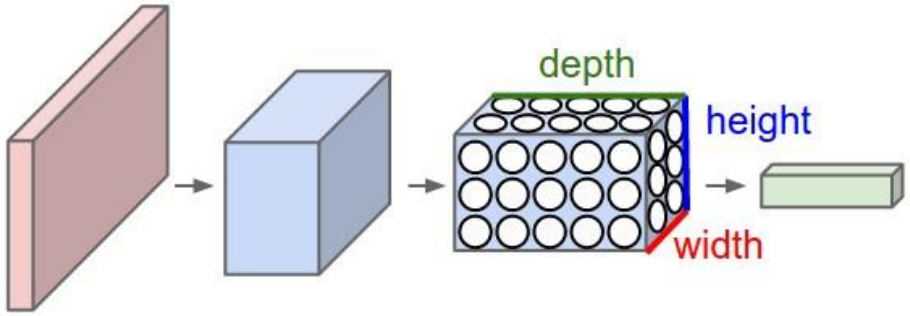
# Sharing neurons - convolutions

Note: material from http://cs231n.github.io/convolutional-networks/

In a conv net we use a different connection pattern between layers:

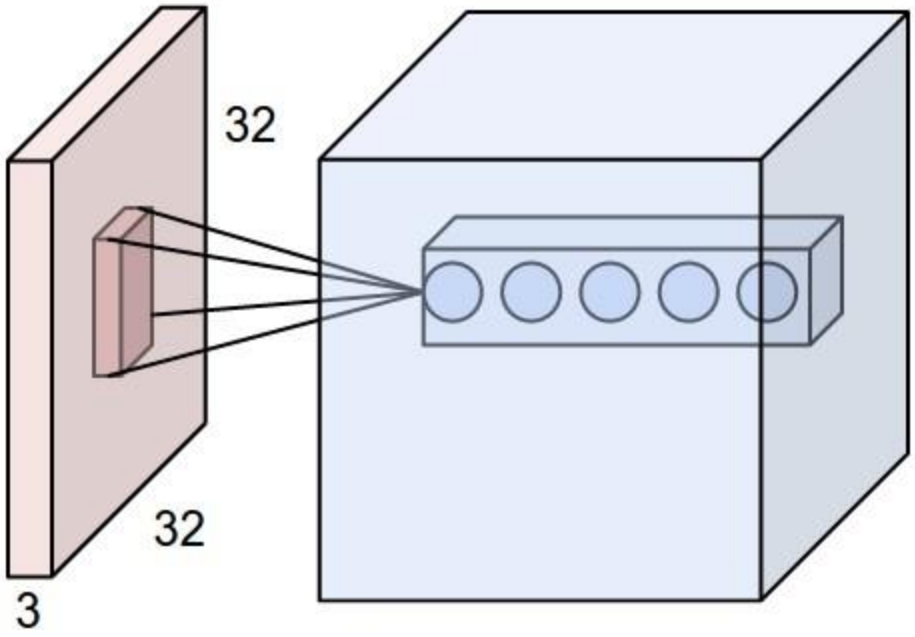
- Typically we use an all-to-all scheme
- In a conv-net we use local connectivity!



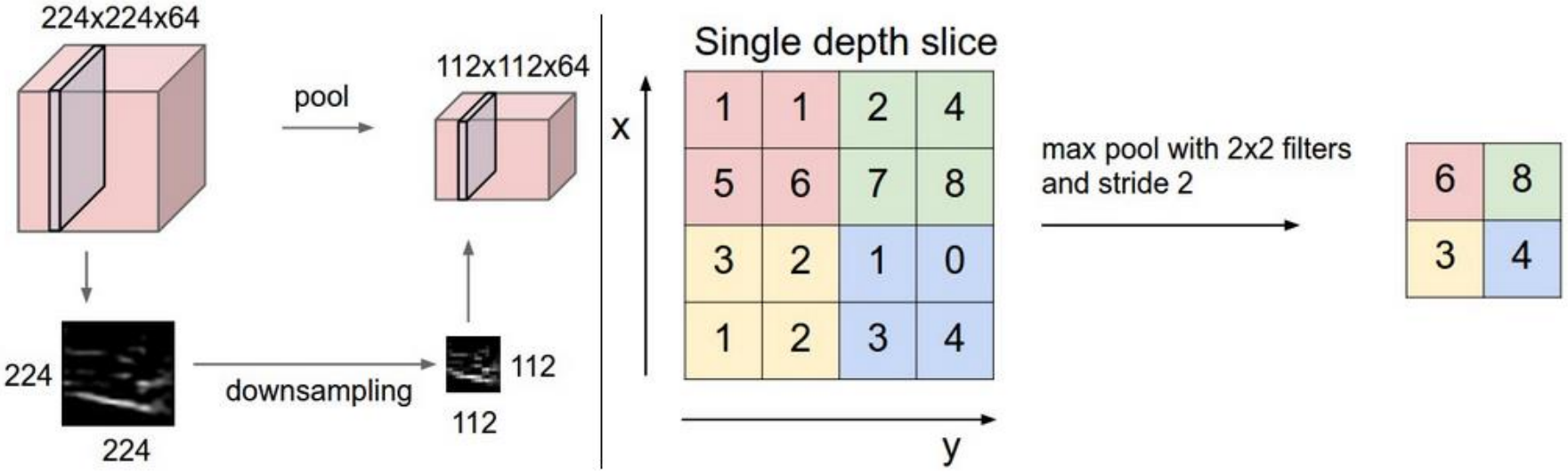



# 2D conv layer

- http://cs231n.github.io/convolutional-networks/

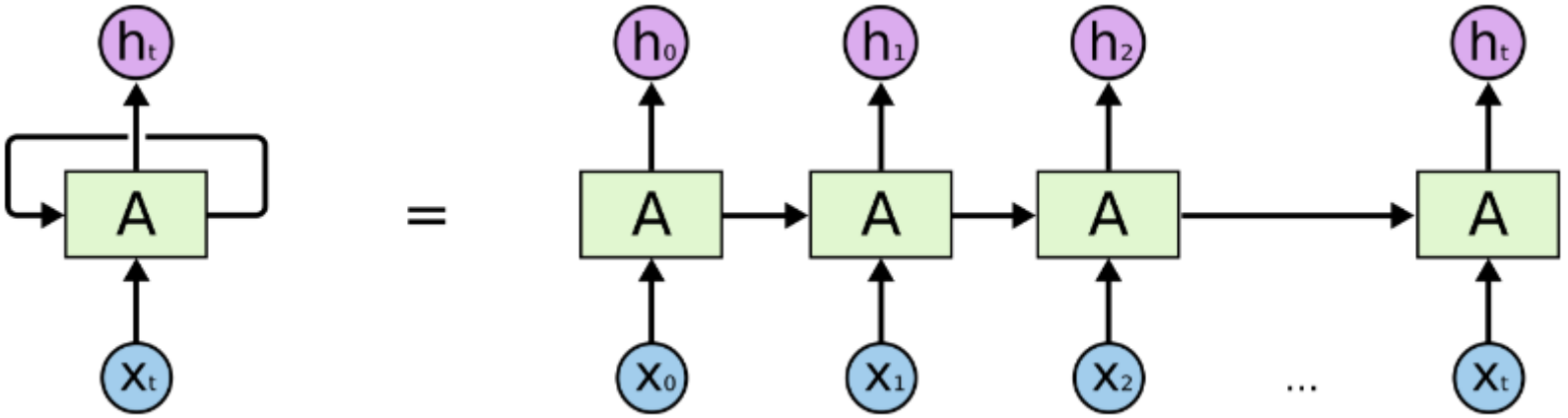


# Pooling



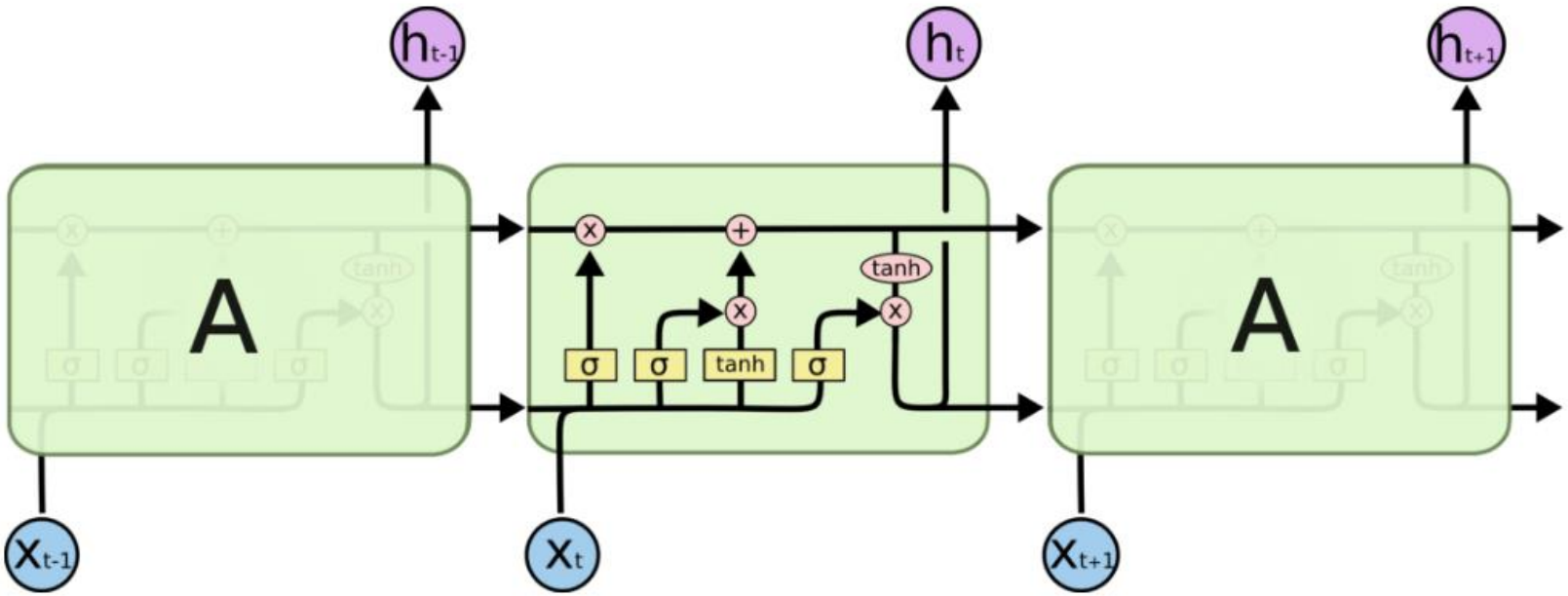

Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

# RNNs




Images from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

# LSTMs




Images from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

# Neural Net Uses

Please know about:

- Neural networks in image recognition
- Neural networks in language processing (language models, word vectors)
- Translation

# How to train a net?

- Assume a certain architecture (#inputs, #outputs, connections, transfer functions)
- Then the network is fully specified by the weights
- Define a loss function – usually the negative of the logarithm of the likelihood (neg log-likelihod)
- Minimize the loss with respect to the weights
  - Initialize the weights to small random values – why?
  - Use gradient information to iteratively change weights – why?
    - Know how network architectural decisions impact gradient computations (e.g. activation function choice).
  - Backpropagation is a structured algorithm to compute the derivative of the loss wrt. weights. It is a direct consequence of the chain rule for differentiation.

# Batch vs stochastic grad descent

- In batch gradient descent we compute the weight update on all (or a large subset) of available data:
  - Pros: the direction is reliable, can use second order methods and make large steps
  - Cons: many computations
- In on-line (stochastic) grad descent we compute the update using few (often just one) sample
  - Pros: very fast computations, good on large data sets
  - Cons: the weight update is „noisy” – must do small steps
  - Tricks:
    - Proper learning rate schedules are a must, impact both overfitting and underfitting
    - Momentum – $\Delta\Theta_t = \alpha\nabla_\Theta(Loss) + \beta\Delta\Theta_{t-1}$

# Practical aspects

- Neural Networks implement functions $\mathbb{R}^n \rightarrow \mathbb{R}^k$
- Need to encode inputs and outputs:
  - Discrete data is usually encoded using 1-of-N
    e.g. Opt1 -> 100, Opt2 -> 010, Opt3 -> 001
    This gives each option its own embedding
    In NLP, this is a very useful representation of words
  - Need to normalize inputs:
    - Zero mean, unit variance
    - Ideally decorrelate them (i.e. apply PCA or ZCA)
  - For classification – apply a sigmoid/softmax to limit the range of outputs, then treat the outputs as probabilities assigned by the net to a class
- For more see LeCun „Efficient Backprop”

# Negative log likelihood

- Typically, we assume that the outputs of our model are probabilities of observing a data sample
  - E.g. $P(y|x;\Theta) = \mathcal{N}(\mu = \Theta^T x, \sigma = 1)$

  Then, under the assumption that samples are iid:

$$P(Y|X;\Theta) = \prod_{j=1}^{m} P\left(y^{(j)}\middle|x^{(j)};\Theta\right)$$

$$\ell(\Theta;Y,X) = -\sum_{j=1}^{m} \log\left(P\left(y^{(j)}\middle|x^{(j)};\Theta\right)\right)$$

  Training minimizes $\ell(\Theta;Y,X)$ over $\Theta$

# Regularization

- As we have seen, too „flexible” models are prone to overtraining.
- We need to prefer some hypotheses over others
  - Examples:
    - Linear models are simpler than polynomial
    - Small neural net is simpler than a large one
- Regularization serves to express our preferences about model simplicity
- Typically, we assign a **prior probability** to our models:

$$P(\boldsymbol{\Theta}) = \prod_{i=1}^{n} \mathcal{N}(\boldsymbol{\Theta}_i; \mu = 0, \sigma = \lambda)$$

# Bayes theorem

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

Interpretation: how our estimate of $A$ changes after seeing $B$.

Why?

$$p(A, B) = p(A|B)p(B) = p(B|A)p(A)$$

Then divide by $p(B)$

# Bayesian approach to ML

- What is the model probability after seeing the samples S?

$$p(\Theta|S) = \frac{p(S|\Theta)p(\Theta)}{p(S)}$$

How to make predictions? Integrate over all models:

$$p(y|x,S) = \int_{\Theta} p(y|x,\Theta)p(\Theta|S)d\Theta$$

Then

$$E[y|x,S] = \int_{y} yp(y|x,S)dy$$

But computing $p(y|x,S)$ is often intractable :(

# Maximum-a-posteriori

- Instead of integrating over all $\Theta$
- Use the maximally probable $\Theta$:

$$\Theta_{MAP} = \arg\max_{\Theta} p(\Theta|S)$$

$$= \arg\max_{\Theta} \left( \prod_{i=1}^{m} p\left(y^{(i)}\middle|x^{(i)}, \Theta\right) \right) p(\Theta)$$

- It's like Max. Likelihood with the extra term (which is the regularization).

# Gaussian model MAP

$$\arg\max_{\Theta} \prod_{i=1}^{m} p\left(y^{(i)} \middle| x^{(i)}, \Theta\right) p(\Theta) =$$

$$\arg\max_{\Theta} \sum_{i=1}^{m} \log p\left(y^{(i)} \middle| x^{(i)}, \Theta\right) + \log\left(p(\Theta)\right)$$

Now if $\Theta_j$ are Gaussian with zero-mean,

$$\log\left(p(\Theta)\right) \propto \sum_{j=1}^{n} \left(\Theta_j\right)^2$$

Thus our minimization criterion gets an extra term, whose derivative is:

$$\nabla_{\Theta} \log\left(p(\Theta)\right) \propto \Theta$$

# Putting it all together

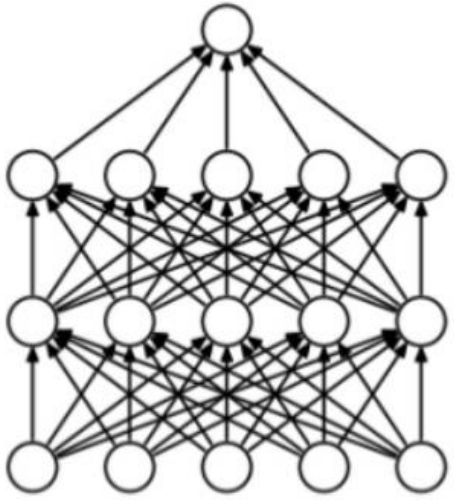
- MAP learning results in two terms:

  $\sum_{i=1}^{m} \log p\left(y^{(i)} \middle| x^{(i)}, \Theta\right) + \lambda \log\left(p(\Theta)\right)$

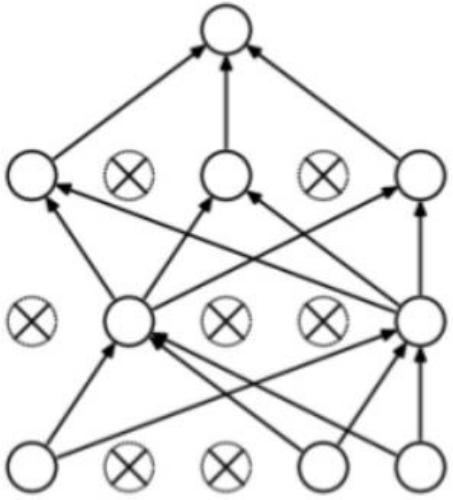
- The two terms (with their constants) allow us to balance MODEL COMPLEXITY and TRAINING LOSS
- Constants $C, \lambda$ and other model parameters, such as number of neurons, type of kernel function, are set via CROSS-VALIDATION

# Other regularization methods

- You can average many models – this nearly always boosts accuracy at the cost of making more computations.
- For neural networks try dropout:
  - For each sample remove some neurons (typically ½)
  - This is like we were sampling a new net for each sample. However, all these networks share weights.
  - During testing use all neurons (need to divide their activations)
  - Net should overfit less



(a) Standard Neural Net



(b) After applying dropout.

# Honest estimates: Hold-out set

- Split the training data into two parts:

- Train only on training, then test on testing.
- Often we do a three-way split:

- Then:
  - Train many models on training (different algos, parameters)
  - Use validation to choose best model
  - Test on testing

# Cross-validation

- Hold-out set makes inefficient data use
- Idea:
  - Divide the data into $k$ sets (~5,10)

  For i=1..k

  Train on all but the i-th set

  may further split to choose the model…

  Test on the i-th set

  Finally:

  take the answers on the testing sets and use them to compute the performance measures
- Extreme case: leave-one-out (jackknife) – always use all but one sample to train!
- We also used the bootstrap – repeated sampling with replacement from the training set.

# Approximations we take

- We want: accuracy on UNKNOWN TEST DATA
- Approximation: Cross-Validation, hold-out set
- But we can't directly optimize accuracy (non-differentiable, NP-hard…)
- Thus optimize a loss function as a proxy for accuracy
- This is often impossible to do exactly – usually use some greedy algorithm (e.g. gradient descent) started randomly
- When doing a ML project: the problem is defined by the accuracy metric and a training set!!!

# Errors can come at all stages

- Data:
  - Is it representative of the problem
  - Does it cover all possible variations (e.g. in France “z” is )
  - Can you get more of it? Generate? Transform?
- Prior beliefs:
  - Does the architecture you choose match the problem?
  - Maybe you know something (e.g. invariants, predominating probability distribution…)
- Loss function:
  - Does it make sense? Is it for classification/regression? Do smaller loss correspond to better performance?
- Training algorithm:
  - Do you reach the minimum of what you optimize?
  - Intentionally? How about early stopping?
- Performance measures:
  - do you separate train from test data?
  - How do train and test errors compare?

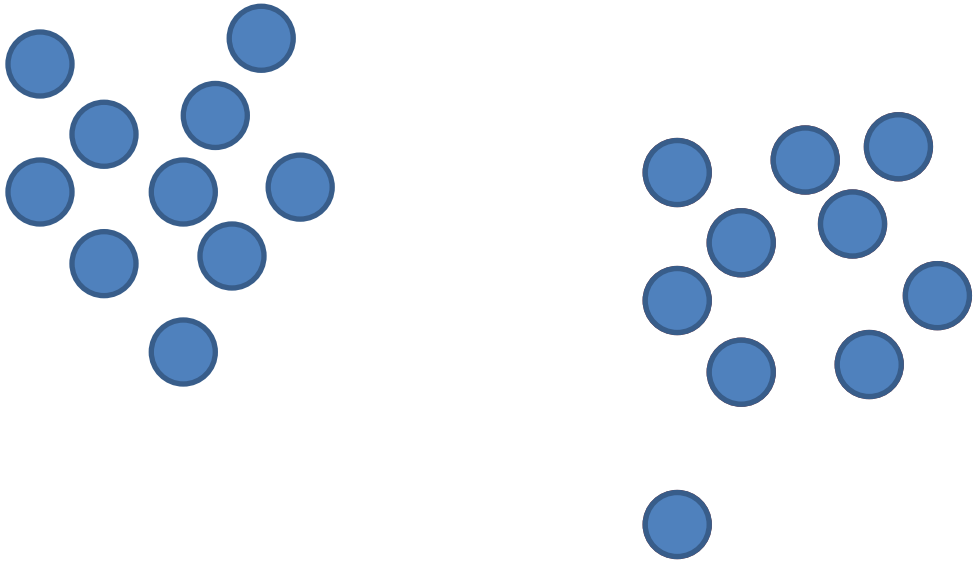
# Example

Linear classifier makes 10% errors

Neural net with 1 hidden layer makes 20% ☹

- Use the same loss – e.g. cross-entropy,
  which one has the lowest?
  (don't change training, just loss computation):
  - Linear classifier -> do you train the net correctly??
    - Maybe use a second order method or SGD?
    - Maybe the net is too small/too regularized?
  - Network -> how is your train and test error, do you over-fit?
    - If they are trained using a different loss, can you try the net with the loss of the linear classifier?
    - Try a smaller network
    - Do you use regularization? Early-stopping?
    - Can you get more training data?
    - Maybe the linear classifier is also over-fitting?

# Unsupervised learning



In supervised learning we have labels
In unsupervised we don't have them!

Describe the data!:

- Find clusters (distinct groups of similar points)
- Reduce the dimensionality
- Find good features that describe the data
- Find and fit a probabilistic model that generated the data

# K-Means – a basic algorithm

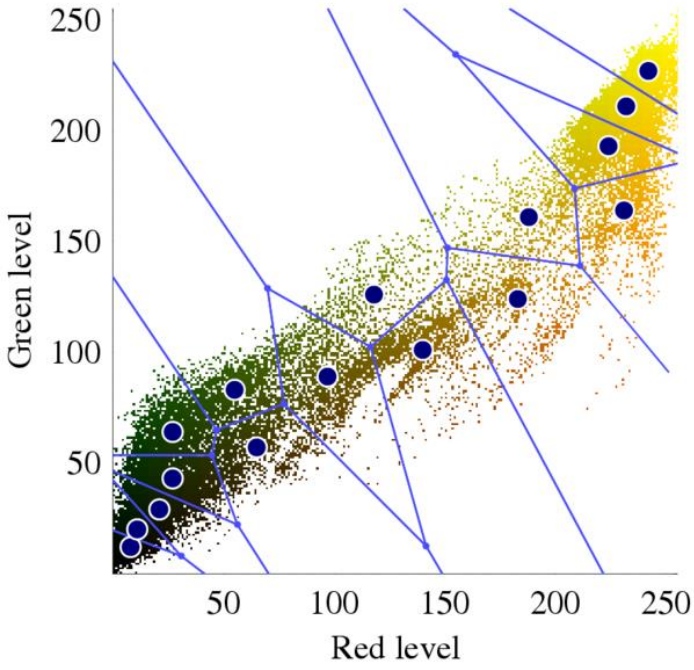
Divides the data into globular clusters according to some distance measure (typ. Euclidean)

Input: $m$ input patterns $x^{(i)}$

1. Initialize K cluster centers $\mu_1 \ldots \mu_k$
   randomly, to some input patterns…
2. Loop until convergence:
   1. For all $i$: set $c^i := \arg\min_j \left\| x^{(i)} - \mu_j \right\|^2$
   2. For all $j$: set $\mu_j := \frac{\sum_i [c^{(i)} = j] x^{(i)}}{\sum_i [c^{(i)} = j]}$

# The K-Means optimization problem

- $J(c, \mu) = \sum_i \left\| x^{(i)} - \mu_{c^{(i)}} \right\|^2$
- The K-means algorithm repeatedly minimizes this over $c$, then over $\mu$ mu etc.
- Initialization:
  - Random
  - To some data samples
  - Bisecting:
    - Start with two clusters
    - Then divide them
    - Then repeat



# Kohonen maps

K-means with topology:

- Assume a topology of units
- Iterate over data samples $x^i$
  - Find the Best Matching Unit
    $$bmu = \arg\min_u \left\| x^i - w^u \right\|$$
  - Move the weights of the BMU and its neighbors in the chosen topology towards $x^i$:
    $$\Delta w^j = \alpha N(j, bmu) x^i$$
- In a Kohonen map units close in the chosen topology point to similar data-space regions.

# Gaussian mixtures and EM

- Assume the data comes from a mixture of Gaussian distributions.

- Probabilistic model for data:
  - First pick a cluster id $p\left(z^{(i)} = j\right) = \phi_j$
  - Then sample from the cluster
$$p\left(x^{(i)} \middle| z^{(i)} = j\right) = \mathcal{N}\left(x; \mu_j, \Sigma_j\right)$$
- Thus the log-likelihood is:

$$\ell(\phi, \mu, \Sigma) = \sum_i \log\left(\sum_j p(x^i | z^i = j) p(z^i = j)\right)$$

# EM algorithm

Initialize randomly or from K-means

Iterate between:

- Estimate probability of $w_j^{(i)} = p\left(z^{(i)} = j\right)$
  - From the Bayes rule

$$w_j^{(i)} = p\left(z^{(i)} = j\right) = \frac{p\left(x^{(i)} \middle| z^{(i)} = j\right) p(z^{(i)} = j)}{\sum_l p(x^{(i)} | z^{(i)} = l) p(z^{(i)} = l)}$$

- Maximize log-likelihood:

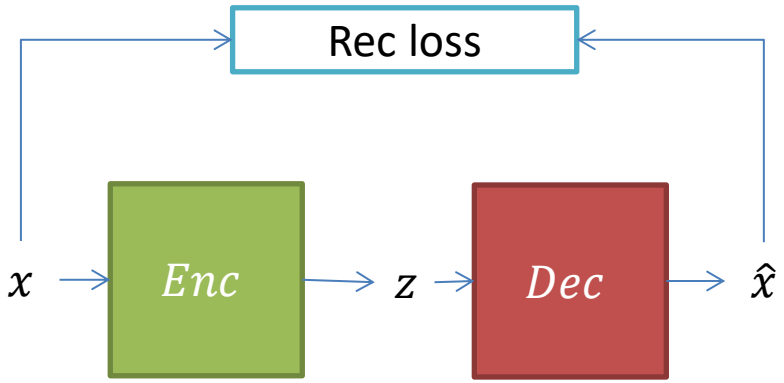
$$\phi_j = \frac{1}{\#samples} \sum_i w_j^{(i)}$$

$$\mu_j = \frac{\sum_i w_j^{(i)} x^{(i)}}{\sum_i w_j^{(i)}}$$

$$\Sigma_j = \frac{\sum_i w_j^{(i)} \left(x^{(i)} - \mu_j\right)\left(x^{(i)} - \mu_j\right)^T}{\sum_i w_j^{(i)}}$$

# Autoencoders

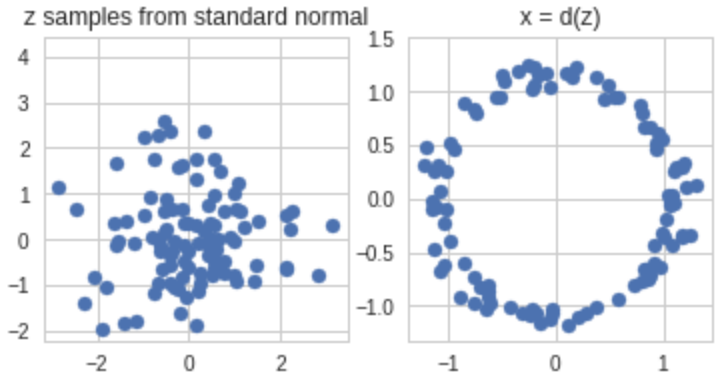
- Two models (neural networks, but can also be affine projections or other):
  - Encodes maps data points into latent codes $z$
  - Decoder reconstructs $x$
- Train for good reconstructions
- Constrain $z$ to learn useful codes (reduce dimensionality, enforce sparsity,…)



# Powerful generative Neural Nets

- Core idea: generate a complicated probability distribution from a simple one
  - E.g.

$$z \sim \mathcal{N}(0,1)$$
$$x = f(z) = \frac{z}{10} + \frac{z}{|z|}$$




- What we really want:
  seed from a normal distribution, generate images

$$z \sim \mathcal{N}(\mu = 0, \sigma = 1)$$
$$x \sim \mathcal{N}(\mu = g(z), \sigma_x)$$

  where $g(z)$ is a powerful transformation, implemented using a neural network

# Generative Adversarial Nets

$$z \sim \mathcal{N}(\mu = 0, \sigma = 1)$$
$$x = g(z)$$

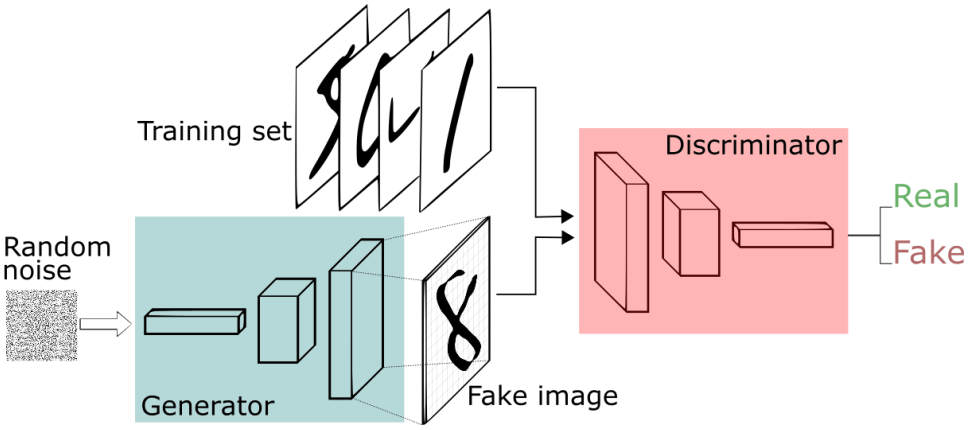
We want $x$ to look like images!

How to train?? What loss to use??

Use a neural net to tell real data from the generated one!

https://arxiv.org/abs/1406.2661

# GAN




Two neural networks: generator $g()$ and discriminator $d()$

$g$ takes a noise sample and returns fake images

$d$ tells them apart

Training:

- Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:
- Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

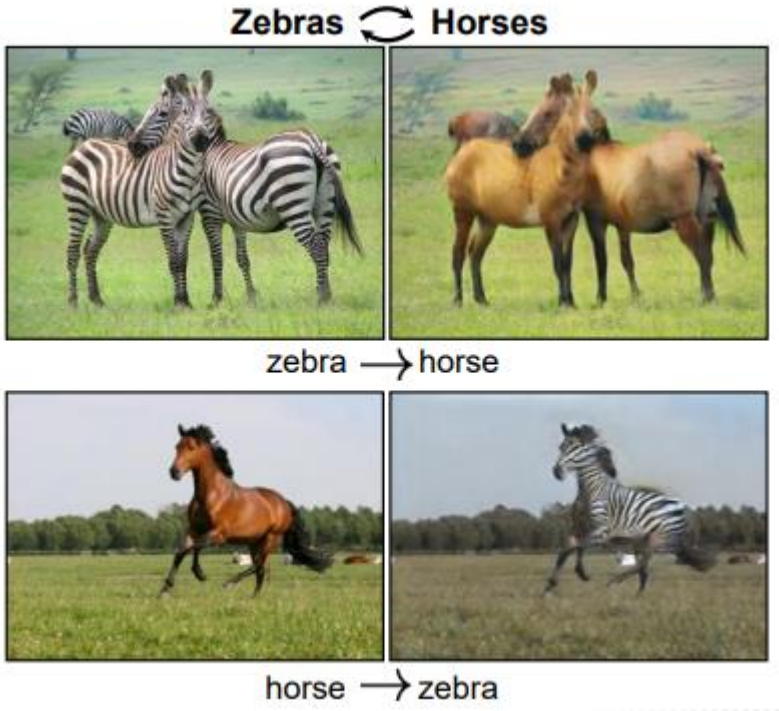
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

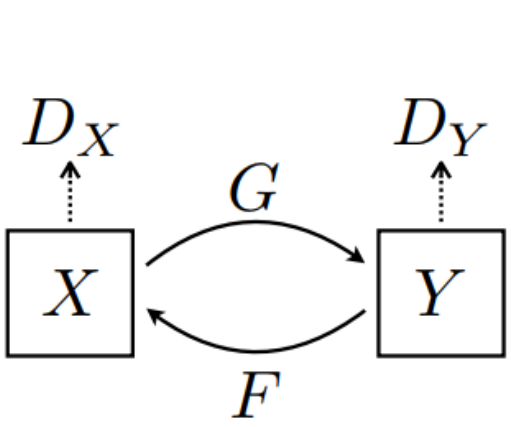
In practice: many tricks needed, but results are excellent

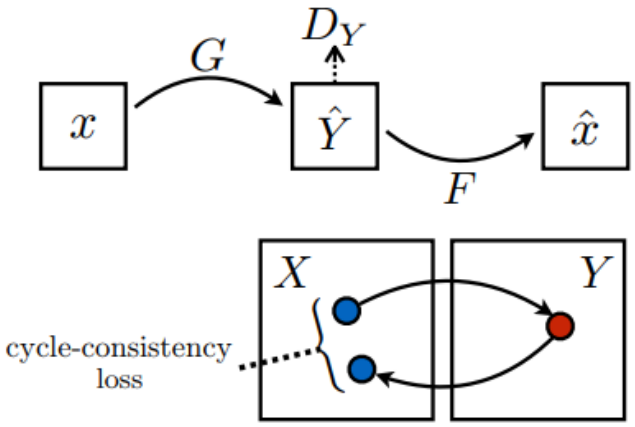
Image from https://medium.com/ml-everything/generating-letters-using-generative-adversarial-networks-gans-161b0be3c229

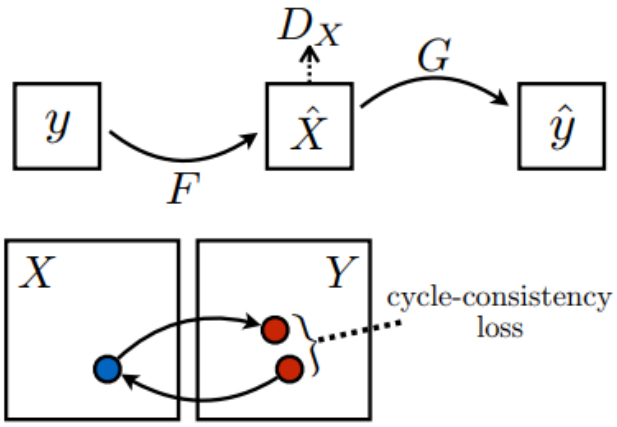
# CycleGAN

Train a mapping between two domains
Make sure a „round-trip” conversion is identity
Use domain discriminators to ensure conversion.



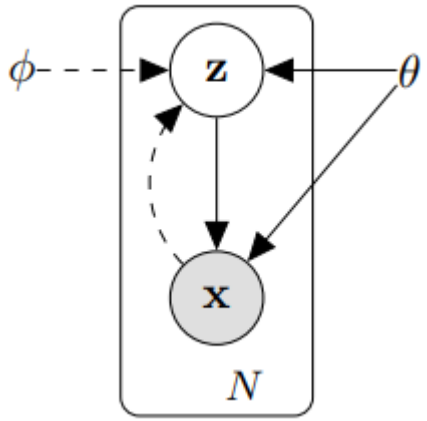




https://arxiv.org/pdf/1703.10593.pdf

# VAE




A probabilistic model:

$$z \sim \mathcal{N}(0,1)$$

$$x \sim \mathcal{N}(\mu = g(z), \sigma_x)$$

$g(z)$ is implemented using a neural network.

We want to train using max likelihood

But $p(x) = \int p(x|z)p(z)dz$ intractable

Also intractable: $p(z|x) = \frac{p(x|z)p(z)}{p(x)}$

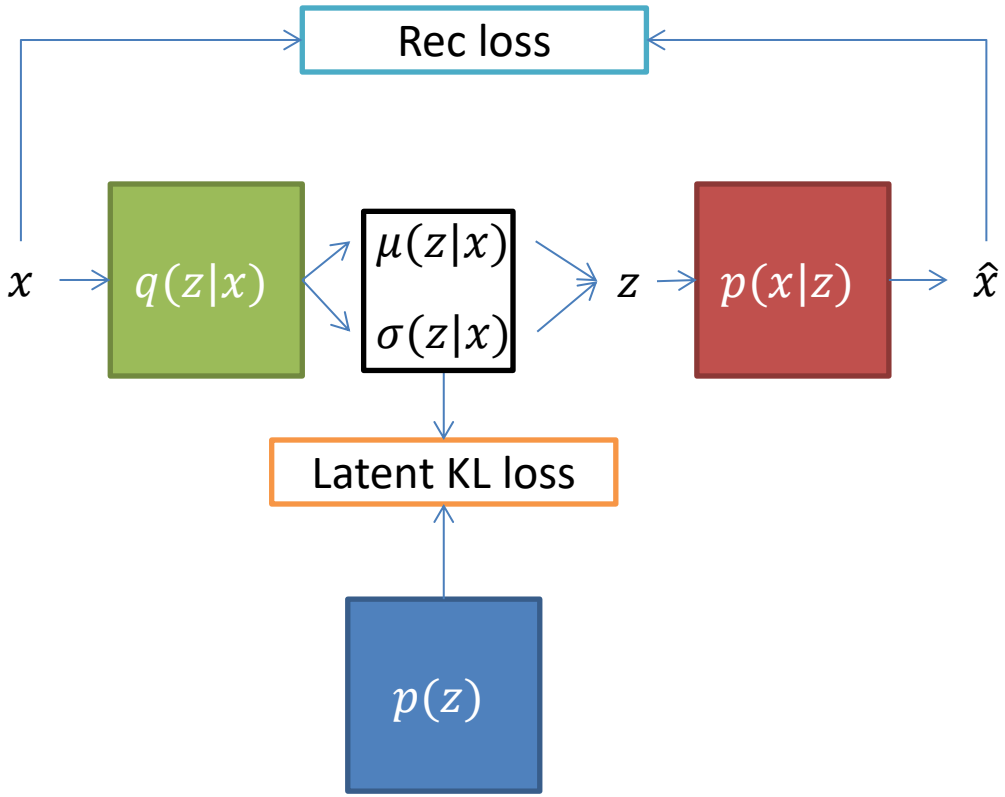
https://arxiv.org/pdf/1312.6114.pdf

# VAE

We can prove (derivation in notebook) that for any $q_\phi(z|x)$:

$$\log p_\Theta(x) \geq \mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\Theta(x|z)] - \mathrm{KL}(q_\phi(z|x)||p_\Theta(z))$$

This corresponds to training an auto-encoder!



# VAE

- Uses two networks:
  - $q$ – approximate inference, given $x$ predict $z$
  - $p$ – generation, given $z$ generate $x$
- Both trained at the same time
- The objective is to:
  - Make good reconstructions of x
  - But limit the amount of information about x transmitted in z

# PCA

- Idea: find a projection direction that will maximize the variance of the data
- $X$ – data matrix (each column is a sample)
- $v$ – projection direction
- $v^T X$ – projected data
- $\overline{v^T X}$ - projection mean $\overline{v^T X} = \frac{1}{N}\sum_{i=1}^{N} v^T x^{(i)}$
- $\frac{1}{N}\left(v^T X - \overline{v^T X}\right)\left(v^T X - \overline{v^T X}\right)^T$ - projection variance
- Goal: find $v$ maximizing variance such that $v^T v = 1$

# PCA - implementation

PCA looks for eigenvectors of data covariance matrix:

- Normalize data – subtract mean
- Compute covariance $\Sigma = XX^T$
- Find eigendecomposition:

$$XX^T = V\lambda V^T$$

- Select the eigenvectors corresponding to the largest eigenvalues

# PCA - interpretation

- PCA is a linear transformation that:
  - Maximizes the variation of the projection
    - Minimum amount of data variability lost, hopefully we loose only noise!
  - The projected data are:
    - Decorrelated
    - Normalized
- PCA is a good data preprocessing algorithm
  - It is quite common to do a PCA prior to training

# Matrix factorization

- Express the data matrix as a product of two low-rank matrices

$$X = UV$$

- K-Means computes such factorization, with one matrix constrained to 1-hot columns (indicating cluster ids)
- Other uses: text representation, rating prediction

# Important topics about learning

- Understand maximum log-likelihood and maximum a posteriori training rules
- Bayes theorem
- Be able to write the negative-log likelihood for a small model (e.g. finding a population's mean)
- Be able to tell the probabilistic interpretation of a model (what is the interpretation of SoftMax, least squares etc.)?

# Important topics for supervised learning

- Define the learning problem
- Linear classifiers:
  - Least-squares regression and logistic regression
  - Probabilistic interpretations
- Neural network – define, compute derivatives – chain rule, backprop algorithm , how to train
  - Batch vs on-line training
  - Regularization, weight decay, dropout
  - Data preparation
  - Why random initialization
- Honest estimates: cross-validation

# Important topics about neural networks

- Convolutional networks:
  - Know about convolution and pooling. What is their purpose?
  - Know for which data they are useful to use.
- Recurrent networks:
  - Be able to explain typical problems of gradient vanishing/exploding
  - Describe parts of LSTM cell, understand the operation.
- Applications
  - Which networks work for which problems
  - Networks used in image recognition and NLP

# Important topics – unsupervised learning

- K-Means
- EM (with derivation)
- Autoencoders – encoder and decoder networks, K-means seen from an autoencoder perspective
- GAN intuitions