NPFL099 Statistical Dialogue Systems
2. Machine Learning Toolkit

http://ufal.cz/npfl099

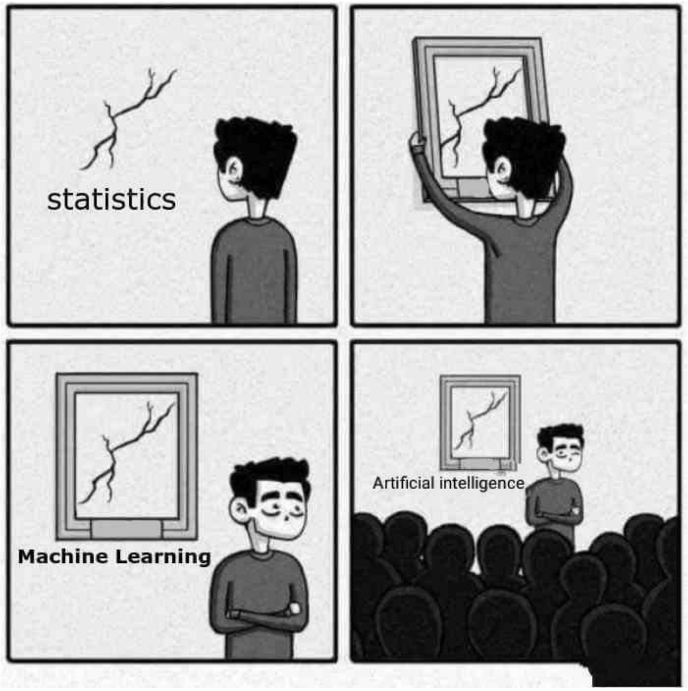
Ondřej Dušek & Vojtěch Hudeček

6. 10. 2020

Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Machine Learning

- ML is basically function approximation
- function: data (**features**)→ **labels**
  - discrete labels = **classification**
  - continuous labels = **regression**
- function shape
  - this is where different algorithms differ
  - neural nets: complex functions, composed of simple building blocks (linear, sigmoid, tanh…)
- training/learning = adjusting function parameters to minimize error
  - **supervised learning** = based on data + labels given in advance
  - **reinforcement learning** = based on exploration & rewards given online



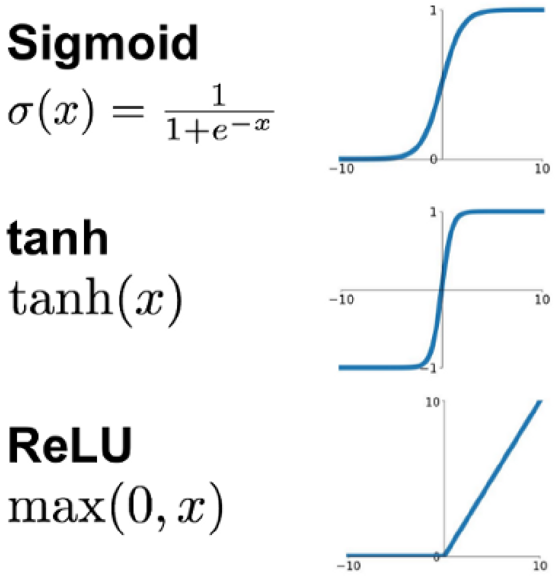
https://towardsdatascience.com/no-machine-learning-is-not-just-glorified-statistics-26d3952234e3

Neural networks

- Can be used for both classification & sequence models
- **Non-linear functions**, composed of basic building blocks
  - stacked into **layers**
- Layers are made of **activation functions**:
  - linear functions
  - nonlinearities – sigmoid, tanh, ReLU
  - softmax – probability estimates:

$$\mathrm{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{|\mathbf{x}|} \exp(x_j)}$$

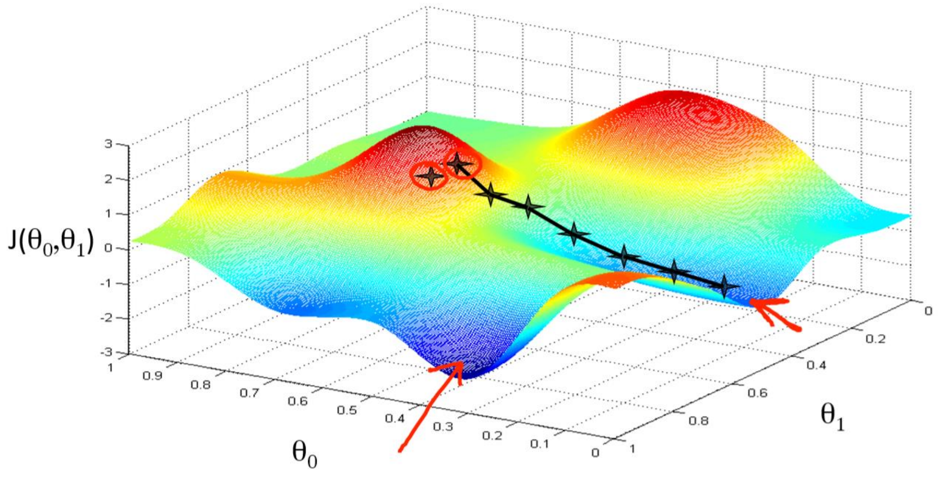
- Fully differentiable – training by **gradient descent**
  - network output incurs loss/cost
  - gradients **backpropagated** from loss to all parameters (composite function differentiation)



https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

Gradient Descent

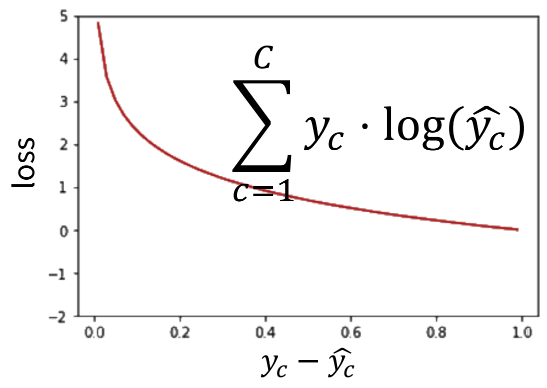
- supervised training– **gradient descent** methods
  - minimizing a **cost/loss function**
    (notion of error – given system output, how far off are we?)
  - calculus: derivative = steepness/slope
  - follow the slope to find the minimum – derivative gives the direction
  - **learning rate** = how fast we go (needs to be tuned)
- gradient typically computed over **mini-batches**
  - random bunches of a few training instances
  - not as erratic as using just 1 instance,
    not as slow as computing over whole data
  - **stochastic gradient descent**

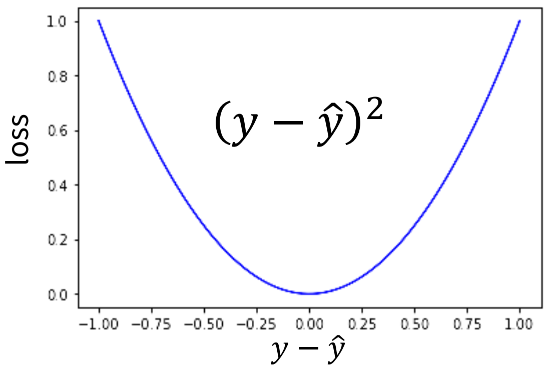


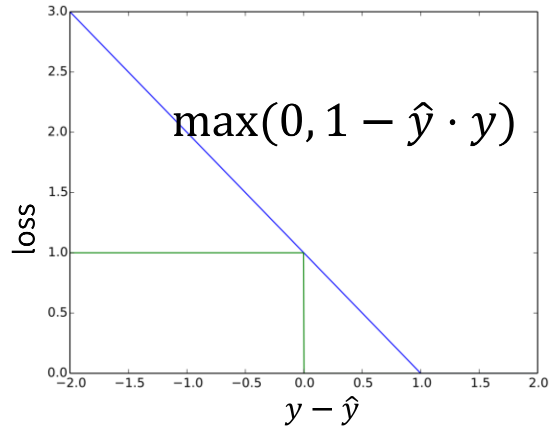
https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

Cost/Loss Functions

- differ based on what we're trying to predict
- **logistic / log loss** ("cross entropy")
  - for classification / softmax – including **word prediction**
    - classes from the whole dictionary
  - pretty stupid for sequences, but works
    - sequence shifted by 1 ⇒ everything wrong
- **squared error loss** – for regression
  - forcing the predicted float value to be close to actual one
- **hinge loss** – for binary classification (SVMs), ranking
  - forcing the correct sign
- many others, variants



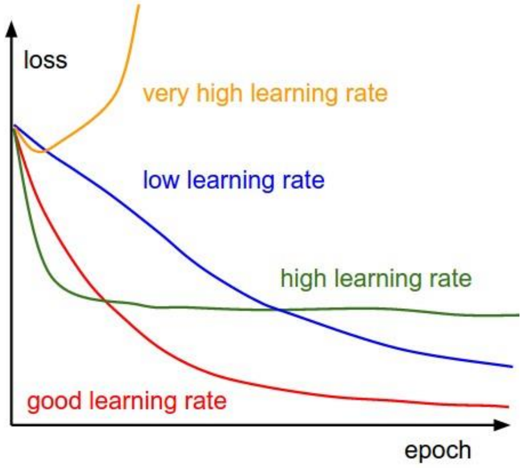




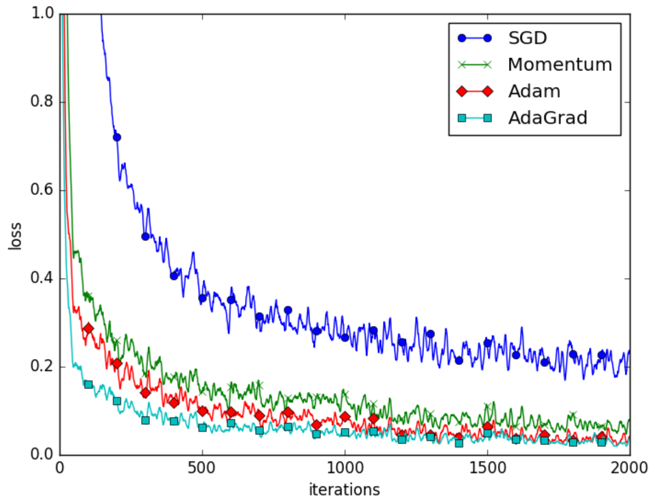
https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/
https://medium.com/@risingdeveloper/visualization-of-some-loss-functions-for-deep-learning-with-tensorflow-9f60be9d09f9
https://en.wikipedia.org/wiki/Hinge_loss

Gradient Descent: Learning Rate

- Learning rate (α) is tricky
  - too high α = may not find optimum
  - too low α = may take forever
- **Learning rate decay**: start high, lower α gradually
- **Momentum**: moving average
  - $m = \beta \cdot m + (1 - \beta) \cdot \Delta$, update by m instead of Δ
- Better options – per-parameter
  - look at how often each single weight gets updated
  - **AdaGrad** – all history
    - remember sum of total gradients squared: $\sum_t \Delta_t^2$
    - divide learning rate by $\sqrt{(\sum \Delta_t^2)}$
  - **Adam** – per-parameter momentum
    - moving averages for Δ & Δ^2: $m = \beta_1 \cdot m + (1 - \beta_1)\Delta$, $v = \beta_2 \cdot v + (1 - \beta_2)\Delta^2$
    - use m instead of Δ, divide learning rate by $\sqrt{(v)}$



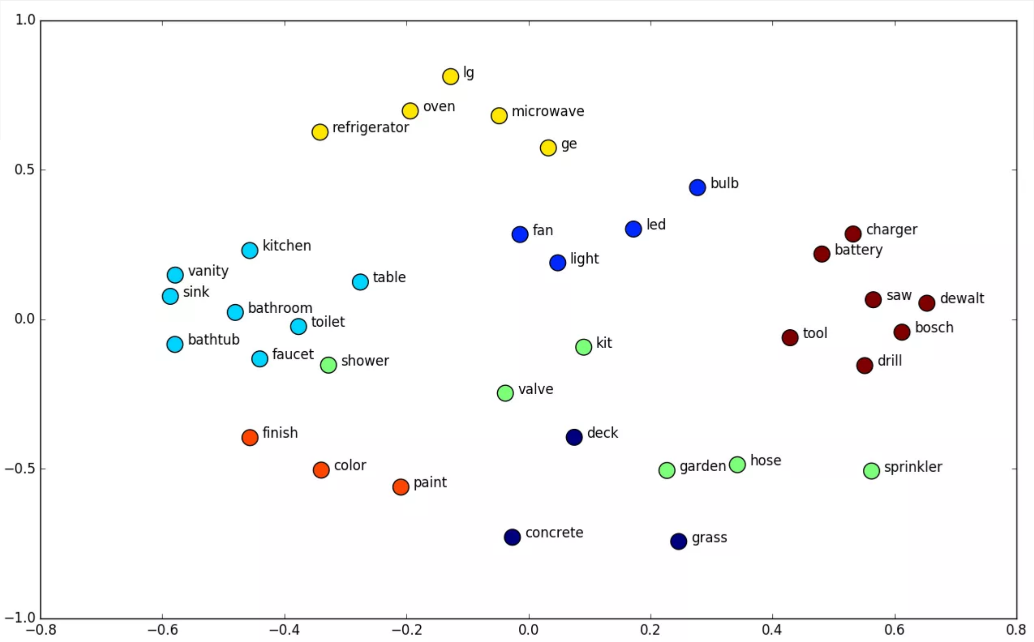
http://cs231n.github.io/neural-networks-3/



http://kaeken.hatenablog.com/entry/2016/11/10/203151

Word Embeddings

- let the network learn features by itself
  - input is just words (vocabulary is numbered)
- distributed word representation
  - **each word = a vector of floats**
- part of network parameters – trained
  a) random initialization
  b) pretraining
- the network learns which words are used similarly
  - they end up having close embedding values
  - different embeddings for different tasks

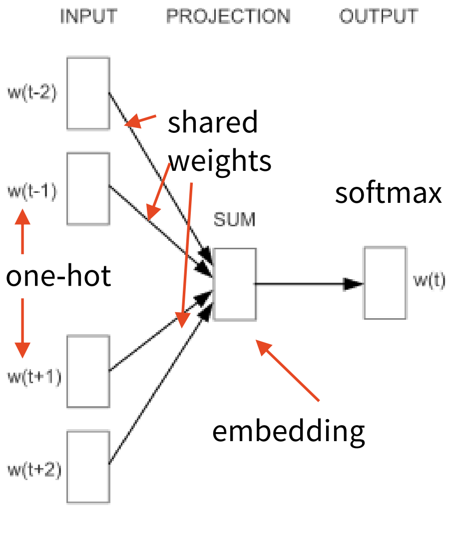


http://blog.kaggle.com/2016/05/18/home-depot-product-search-relevance-winners-interview-1st-place-alex-andreas-nurlan/

http://ruder.io/word-embeddings-2017/

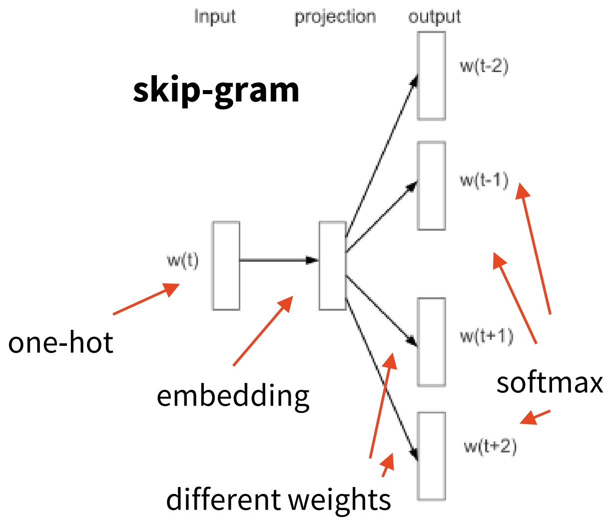
Pretrained Word Embeddings

- **Word2Vec**
  - Continuous Bag-of-Words
    - predict a word, given $\pm k$ words window
    - disregarding word order within the window
  - Skip-gram: reverse
    - given a word, predict its $\pm k$ word window
    - closer words = higher weight in training



CBOW

(Mikolov et al., 2013)
http://arxiv.org/abs/1301.3781



skip-gram

- **GloVe**
  - optimized directly from corpus co-occurrences (= w_1 close to w_2)
  - target: $e_1 \cdot e_2 = \log(\#\text{co-occurrences})$
    - number weighted by distance, weighted down for low totals
  - trained by minimizing reconstruction loss on a co-occurrence matrix

(Pennington et al., 2014)
http://aclweb.org/anthology/D14-1162

https://geekyisawesome.blogspot.com/2017/03/word-embeddings-how-word2vec-and-glove.html
https://machinelearninginterview.com/topics/natural-language-processing/what-is-the-difference-between-word2vec-and-glove/

Word Embeddings

- Vocabulary is unlimited, embedding matrix isn't
  - + the bigger the embedding matrix, the slower your models
- Special **out-of-vocabulary token** *<unk>*
  - "default" / older option
  - all words not found in vocabulary are assigned this entry
  - can be trained using some rare words in the data
  - problem for generation – you don't want these on the output
- Using limited sets
  - **characters** – very small set
    - works, but makes for very long sequences
  - **subwords** – decided e.g. by byte-pair encoding
    - start from individual characters
    - iteratively merge most frequent bigram, until you get desired # of subwords
    - *sub@@ word* – the *@@* marks "no space after"

(Sennrich et al., 2016)
https://www.aclweb.org/anthology/P16-1162/

Convolutional Networks

- Designed for computer vision – inspired by human vision
  - works for language in 1D, too!
- Use less parameters than fully connected – **filter/kernel**
- Apply filter repeatedly over the input
  - element-wise multiply window of input x filter
  - sum + apply non-linearity (ReLU) to result
  - => produce 1 element of output
- **Stride** – how many steps to skip
  - less overlap, reducing output dimension
- **Pooling** – no filter, pre-set operation
  - **maximum**/average on each window
  - typical CNN architecture alternates convolution & pooling

input

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

filter

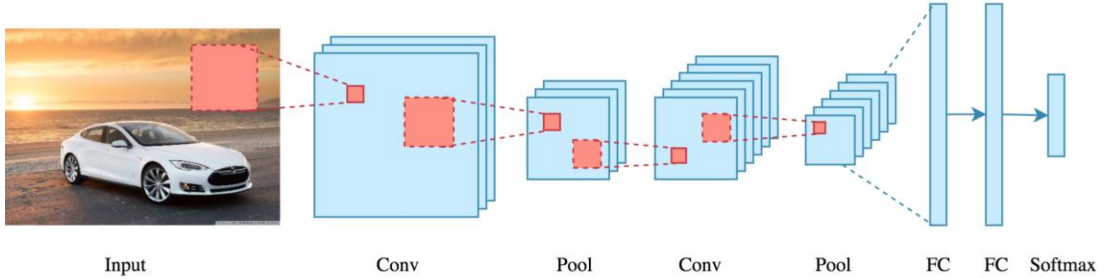
1	0	1
0	1	0
1	0	1

input x filter

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		

output

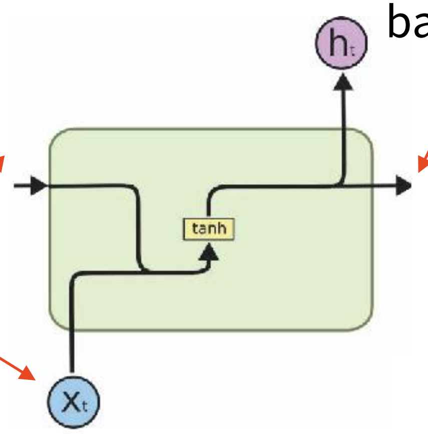


https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

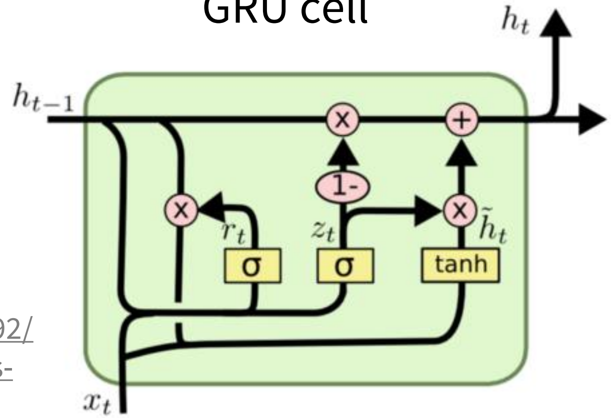
Recurrent Neural Networks

- Many identical layers with shared parameters (**cells**)
  - ~ the same layer is applied multiple times, taking its own outputs as input
    - ~ same number of layers as there are tokens
    - output = **hidden state** – fed to the next step
  - additional input – next token features
- Cell types
  - **basic RNN**: linear + tanh
    - problem: vanishing gradients
    - can't hold long recurrences
  - **GRU, LSTM**: more complex, to make backpropagation work better
    - "gates" to keep old values

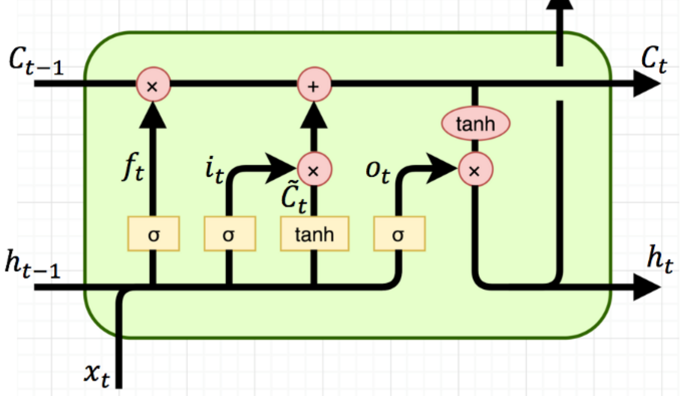
basic RNN cell



GRU cell



LSTM cell



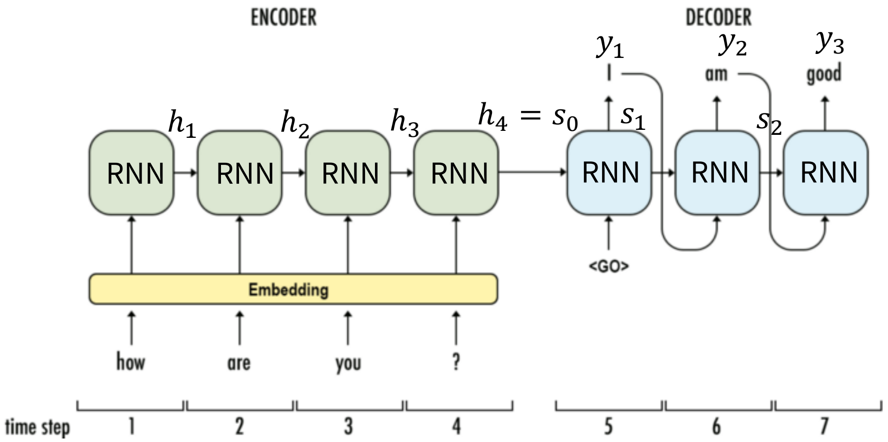
https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57

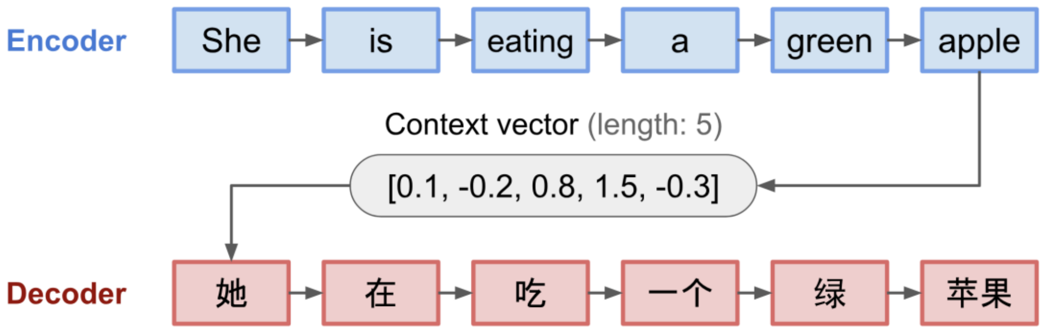
Encoder-Decoder Networks (Sequence-to-sequence)

- Default RNN paradigm for sequences/structure prediction
  - **encoder** RNN: encodes the input token-by-token into **hidden states** h_t
    - next step: last hidden state + next token as input
  - **decoder RNN**: constructs the output token-by-token
    - initialized by last encoder hidden state
    - output: hidden state & softmax over output vocabulary + argmax
    - next step: last hidden state + last generated token as input
  - LSTM/GRU cells over vectors of ~ embedding size
  - used in MT, dialogue, parsing…
    - more complex structures linearized to sequences

$$\boldsymbol{h}_0 = \boldsymbol{0}$$
$$\boldsymbol{h}_t = \mathrm{cell}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$$

$$\boldsymbol{s}_0 = \boldsymbol{h}_T$$
$$p(y_t | y_1, \dots y_{t-1}, \mathbf{x}) = \mathrm{softmax}(\boldsymbol{s}_t)$$
$$\boldsymbol{s}_t = \mathrm{cell}(\boldsymbol{y}_{t-1}, \boldsymbol{s}_{t-1})$$



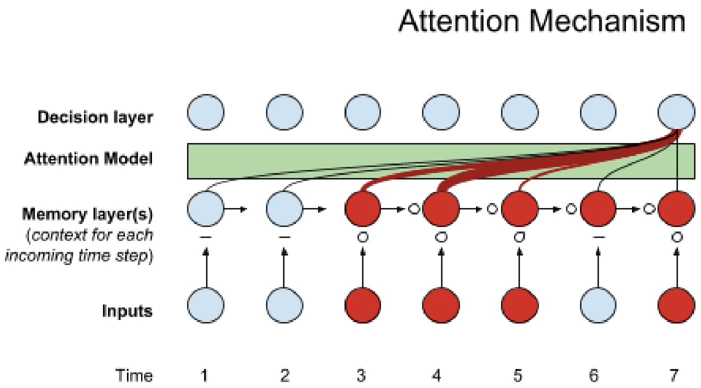


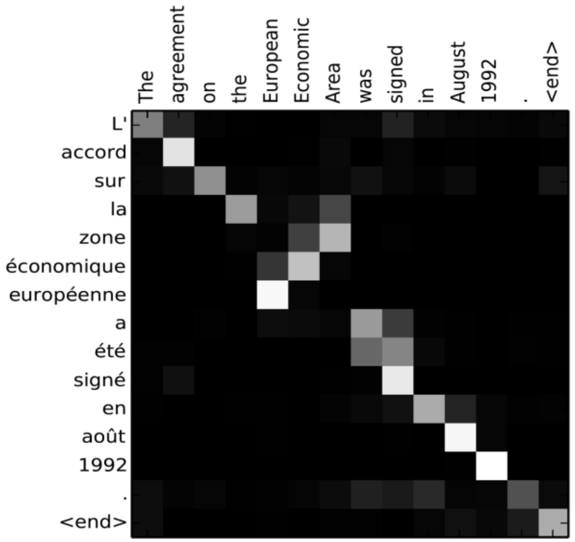
https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

https://medium.com/syncedreview/a-brief-overview-of-attention-mechanism-13c578ba9129

Attention

- Encoder-decoder is too crude for complex sequences
  - the whole input is crammed into a fixed-size vector (last hidden state)
- **Attention** = “memory” of **all encoder** hidden states
  - weighted combination, re-weighted for every decoder step
    → can focus on currently important part of input
  - fed into decoder inputs + decoder softmax layer
- **Self-attention** – over **previous decoder steps**
  - increases consistency when generating long sequences



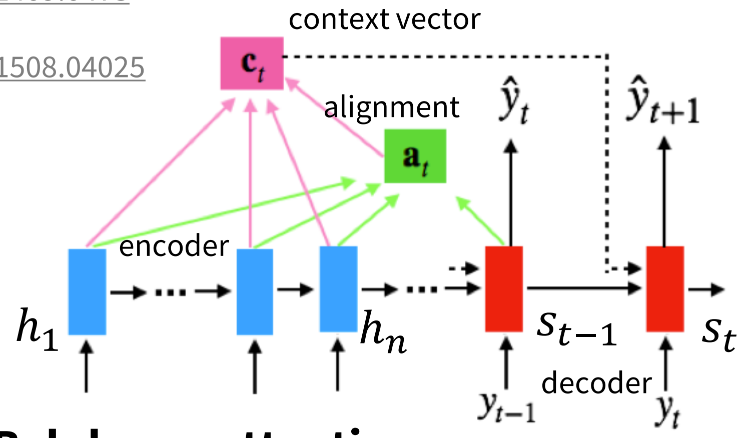


https://skymind.ai/wiki/attention-mechanism-memory-network

Bahdanau & Luong Attention

- different combination with decoder state
  - Bahdanau: use on input to decoder cell
  - Luong: modify final decoder state
- different weights computation
- both work well – exact formula not important

(Bahdanau et al., 2015)
http://arxiv.org/abs/1409.0473
(Luong et al., 2015)
http://arxiv.org/abs/1508.04025



Bahdanau attention

attention weights = alignment model

decoder state

Bahdanau: trained parameters

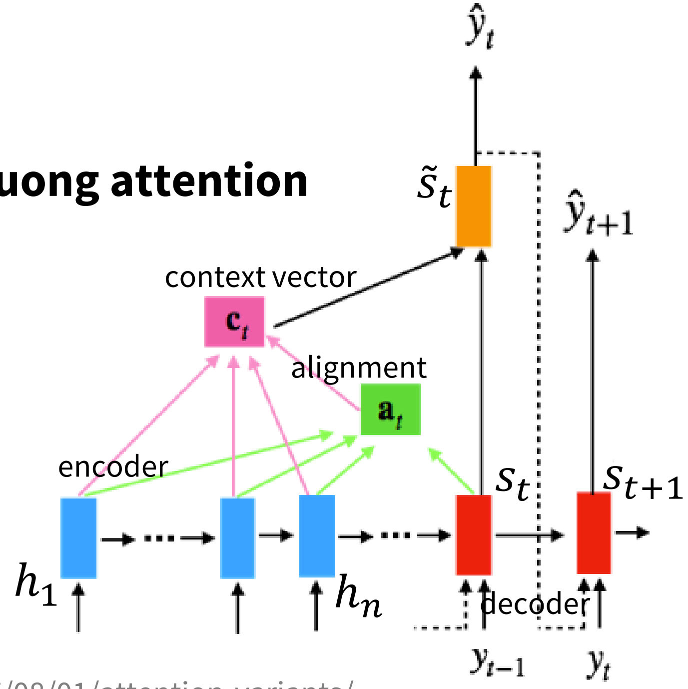
$$\alpha_{ti} = \mathrm{softmax}(\boldsymbol{v}_\alpha \cdot \tanh(\mathbf{W}_\alpha \cdot \boldsymbol{s}_{t-1} + \mathbf{U}_\alpha \cdot \boldsymbol{h}_i))$$

encoder hidden state

Luong: $\alpha_{ti} = \mathrm{softmax}(\boldsymbol{h}_i^\top \cdot \boldsymbol{s}_t)$ decoder state

encoder hidden state

Luong attention



attention value = context vector

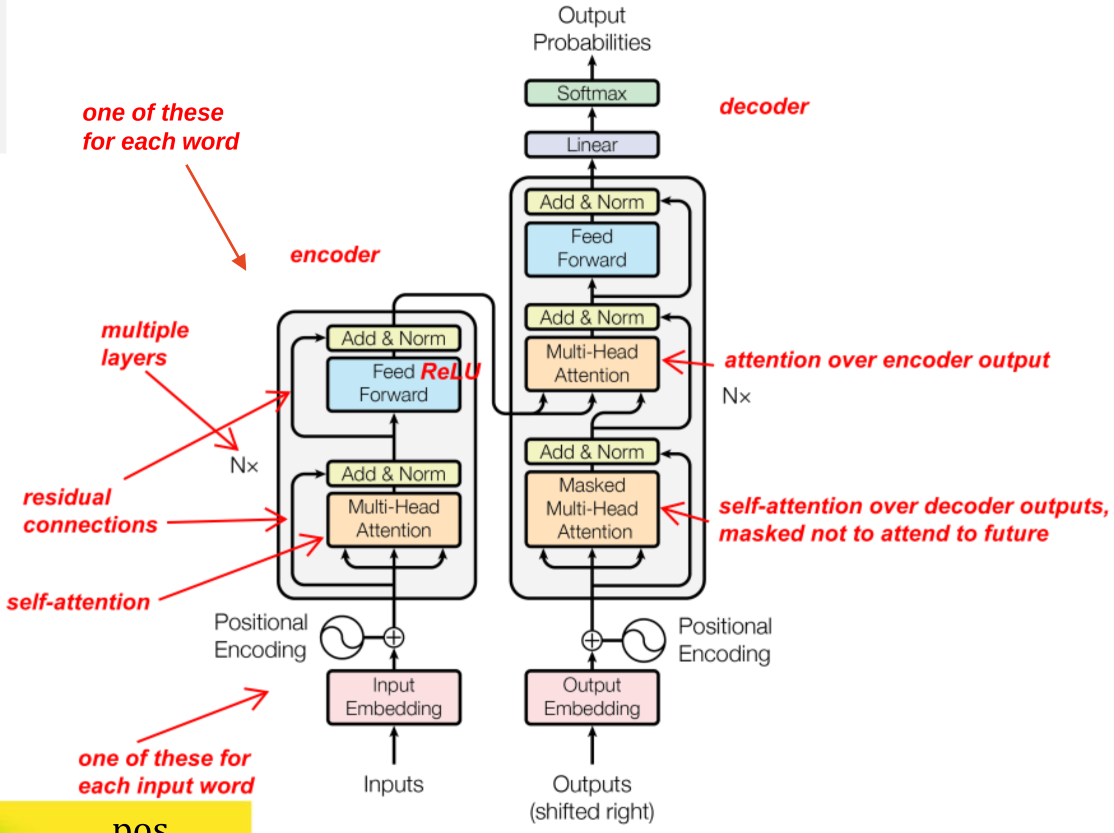
same for both – sum encoder hidden states weighted by α_{ti}

$$\boldsymbol{c}_t = \sum_{i=1}^{n} \alpha_{ti} \boldsymbol{h}_i$$

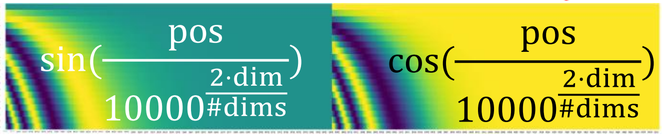
http://cnyah.com/2017/08/01/attention-variants/

Transformer

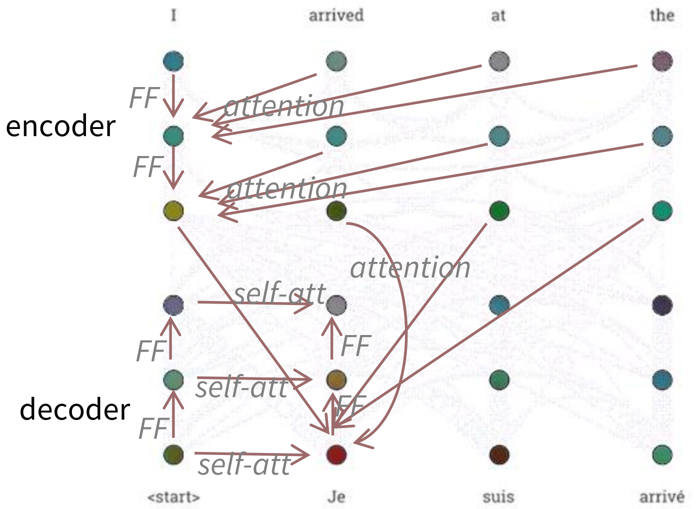
(Waswani et al., 2017)
https://arxiv.org/abs/1706.03762



- getting rid of (encoder) recurrences
  - making it faster to train, allowing bigger nets
  - replace everything with attention + feed-forward networks
  - ⇒ needs more layers
  - ⇒ needs to encode positions
- positional encoding
  - adding position-dependent patterns to the input



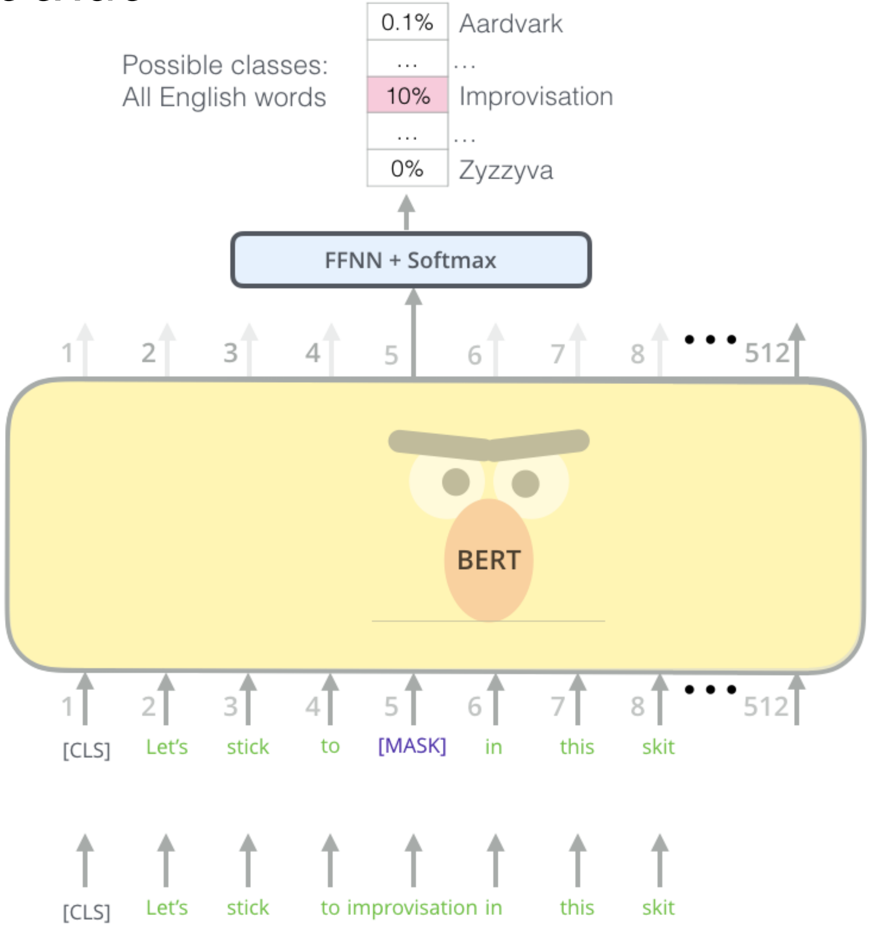
- attention – dot-product (Luong style)
  - scaled by $\frac{1}{\sqrt{\#\text{dims}}}$ (so values don't get too big)
  - **more heads** (attentions in parallel) – focus on multiple inputs



http://jalammar.github.io/illustrated-transformer/ https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

Contextual Word Embeddings

- Beyond pretrained word embeddings
  - words have different meanings based on context
  - static word embeddings (word2vec/GloVe) don't reflect that
- **ELMo**
  - LSTMs trained for language modelling
  - ELMo embeddings = weighted sum of input static embeddings & LSTM outputs
    - the weights are trained for a specific downstream task
- **BERT**
  - huge Transformer encoder trained for:
    - masked word prediction
    - adjacent sentences detection (does B come right after A?)
  - BERT embeddings = any combination of the Transformer layers



http://jalammar.github.io/illustrated-bert/

Pretrained Language Models (~ Contextual Word Embeddings)

- Basically a newer name/perspective for the same idea
  1. **Pretrain** a model on a huge dataset and some meaningful language-related task
  2. **Fine-tune** for your own task on your (smaller) data
- There are many variants of the pretrained models
  - mostly based on the Transformer architecture
  - pretraining tasks vary and make a difference

(Devlin et al., 2019)
https://www.aclweb.org/anthology/N19-1423
https://github.com/google-research/bert

(Rogers et al., 2020) http://arxiv.org/abs/2002.12327

(Liu et al., 2019) http://arxiv.org/abs/1907.11692

- **BERT** + variants: multilingual, **RoBERTa** (optimized)

(Radford et al., 2019)
https://openai.com/blog/better-language-models/

- **GPT**(**-2**/**-3**): Transformer decoder only, next-word prediction

(Brown et al., 2020)
http://arxiv.org/abs/2005.14165

- **BART**: BERT as denoising autoencoder (more below)

(Lewis et al., 2019) http://arxiv.org/abs/1910.13461

- **T5**: generalization, many variants

(Raffel et al., 2019) http://arxiv.org/abs/1910.10683

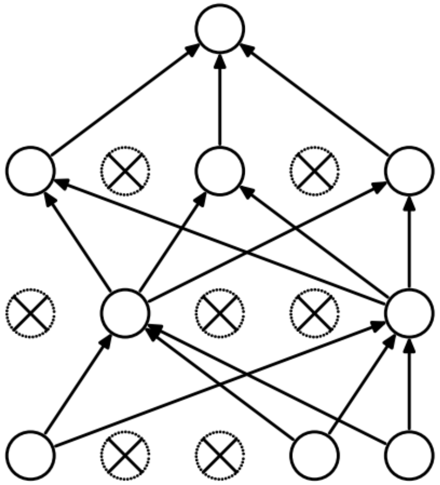
- a lot of this is released plug-and-play
  - you only need to finetune (and sometimes, not even that)

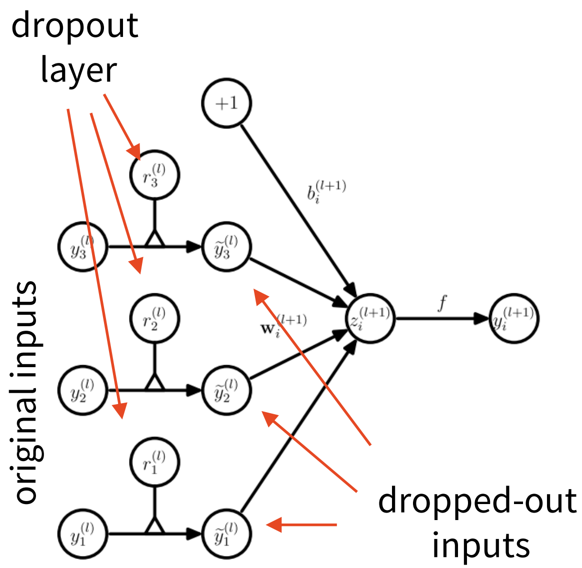
https://github.com/huggingface/transformers

Dropout

(Srivastava et al., 2014)
http://jmlr.org/papers/v15/srivastava14a.html

- overfitting to training data is a problem for NNs
    - too many parameters
- **Dropout** – simple regularization technique
    - more effective than e.g. weight decay (L2)
    - **zero out some neurons/connections** in the network at random
    - technically: multiply by dropout layer
        - 0/1 with some probability (typically 0.5–0.8)
    - at training time only – full network for prediction
    - weights scaled down after training
        - they end up larger than normal because there’s fewer nodes
        - done by libraries automatically
    - may need larger networks to compensate



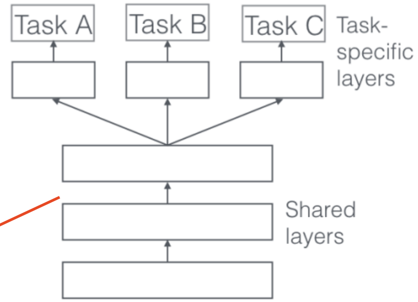


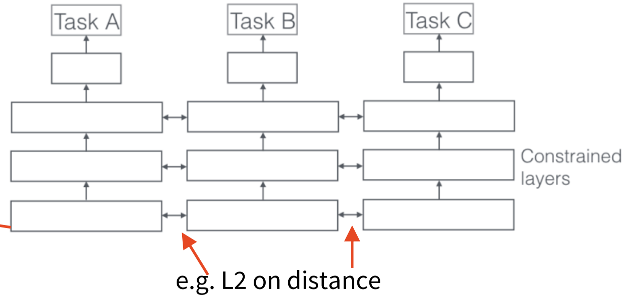
(b) Dropout network

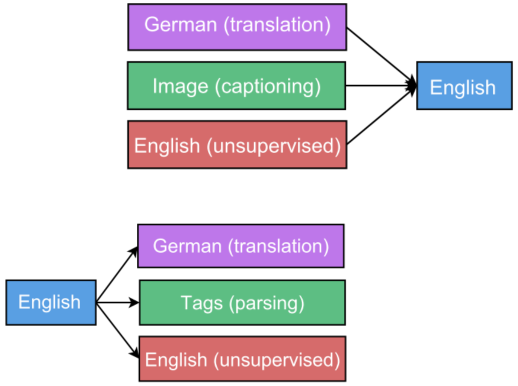
Multi-task Learning

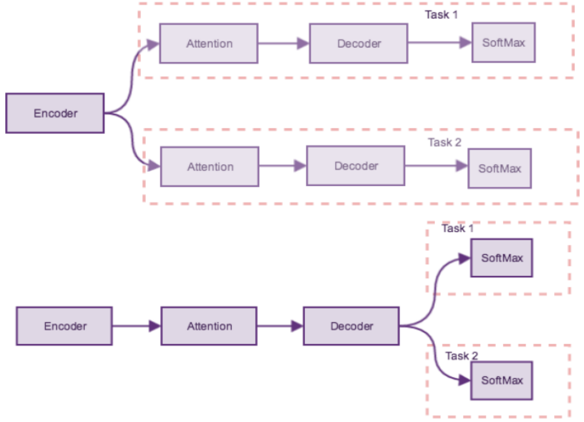
(Ruder, 2017)
http://arxiv.org/abs/1706.05098
(Fan et al., 2017)
http://arxiv.org/abs/1706.04326
(Luong et al., 2016)
http://arxiv.org/abs/1511.06114

- achieve better generalization by learning more things at once
  - a form of regularization
  - implicit data augmentation
  - biasing/focusing the model
    - e.g. by explicitly training for an important subtask
- parts of network shared, parts task-specific
  - hard sharing = parameters truly shared (most common)
  - soft sharing = regularization by parameter distance
  - different approaches w. r. t. what to share
- training – alternating between tasks
  - so the network doesn't "forget"



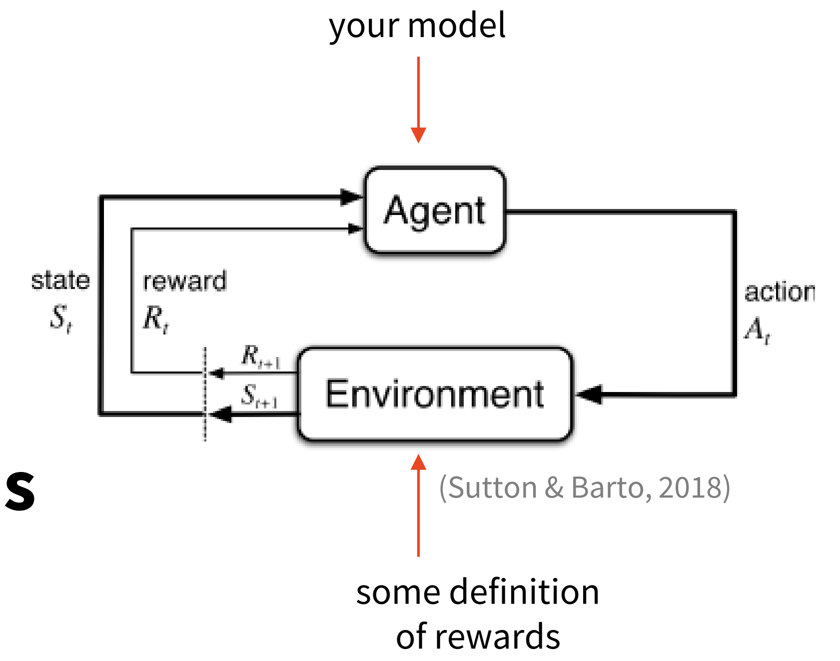






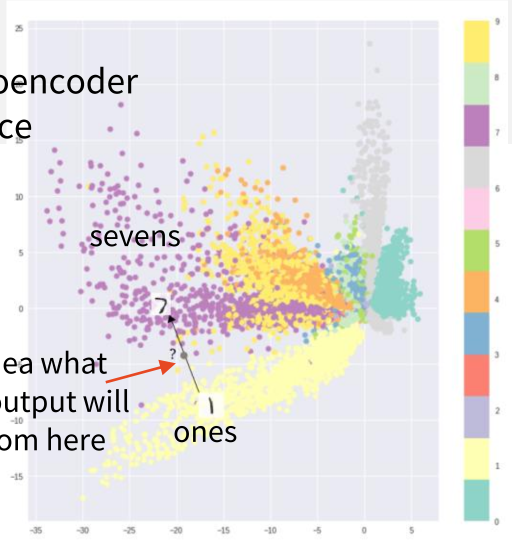
Reinforcement Learning

- Learning from **weaker supervision**
  - only get feedback once in a while, not for every output
  - good for globally optimizing sequence generation
    - you know if the whole sequence is good
    - you don't know if step X is good
  - sequence = e.g. sentence, dialogue
- Framing the problem as **states & actions & rewards**
  - "robot moving in space", but works for dialogue too
  - state = generation so far (sentence, dialogue state)
  - action = one generation output (word, system dialogue act)
  - defining rewards might be an issue
- Training: **maximizing long-term reward**
  - via state/action values (Q function)
  - directly – optimizing policy

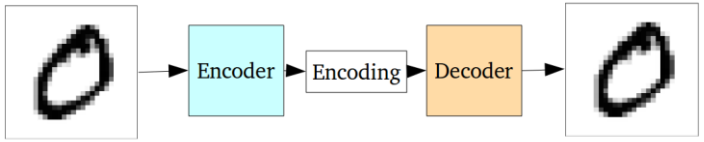


Autoencoders

https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf

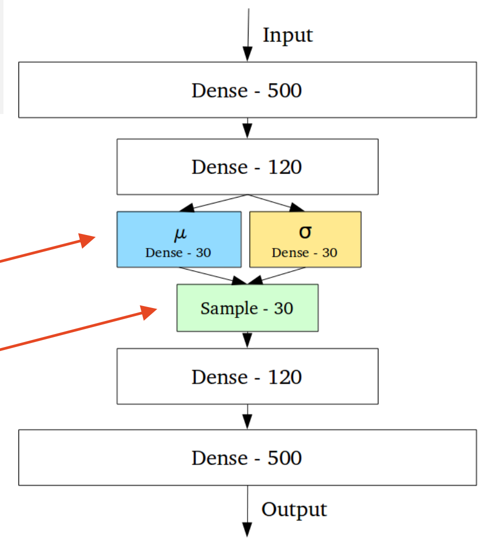


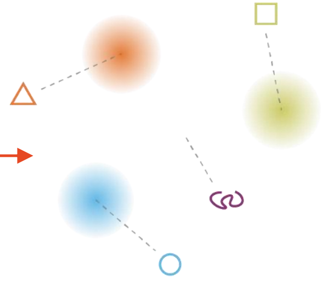
- Using NNs as **generative models**
  - more than just classification – modelling the whole distribution
    - (of e.g. possible texts, images)
  - generate new instances that look similar to training data
  - considered **unsupervised learning**
- **Autoencoder**: input → encoding → input
  - encoding ~ "embedding" in latent space (i.e. some vector)
  - trained by reconstruction loss
  - problem: can't easily get valid embeddings for generating new outputs
    - parts of embedding space might be unused – will generate weird stuff
    - no easy interpretation of embeddings – no idea what the model will generate
  - still has uses:
    - **denoising autoencoder**: add noise to inputs, train to generate clean outputs
    - multi-task learning, representations for use in downstream tasks



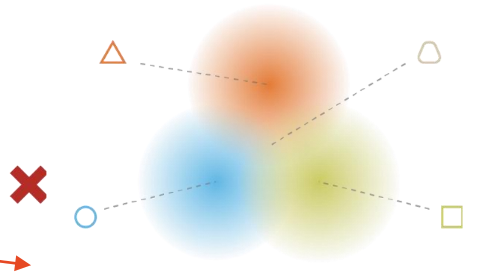
Variational Autoencoders

- Making the encoding latent space more useful
  - using **Gaussians** – continuous space by design
  - encoding input into vectors of means μ & std. deviations σ
  - sampling encodings from $N(\mu, \sigma)$ for generation
    - samples vary a bit even for the same input
    - decoder learns to be more robust
  - model can degenerate into normal AE ($\sigma \rightarrow 0$)
    - we need to encourage some σ, smoothness, overlap (μ ~ 0)
    - add **2nd loss: KL divergence** from $N(0,1)$
    - VAE learns a trade-off between
      using unit Gaussians & reconstructing inputs
- Problem: still not too much control of the embeddings
  - we can only guess what kind of output the model will generate

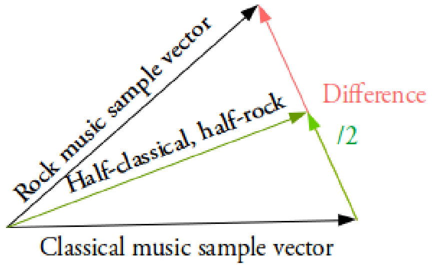




what can happen without regularisation



what we want to obtain with regularisation



https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf
https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73
http://kvfrans.com/variational-autoencoders-explained/

VAE details

- VAE objective:
  - **reconstruction loss** (maximizing $p(x|z)$ in the decoder), MLE as per usual
  - **latent loss** (KL-divergence from ideal $p(z) \sim \mathcal{N}(0,1)$ in the encoder)

$$\mathcal{L} = -\mathbb{E}_q[\log p(x|z)] + KL[q(z|x)||p(z)]$$

- This is equivalent to maximizing true $\log p(x)$ with some error
  - i.e. maximizing **evidence lower bound** (ELBO) / variational lower bound:

$$\mathbb{E}_q[\log p(x|z)] - KL[q(z|x)||p(z)] = \underbrace{\underset{\substack{\uparrow \\ \text{“evidence”} \\ \text{(i.e. data)}}}{\log p(x)} - KL[q(z|x)||p(z|x)]}_{\text{ELBO}}$$

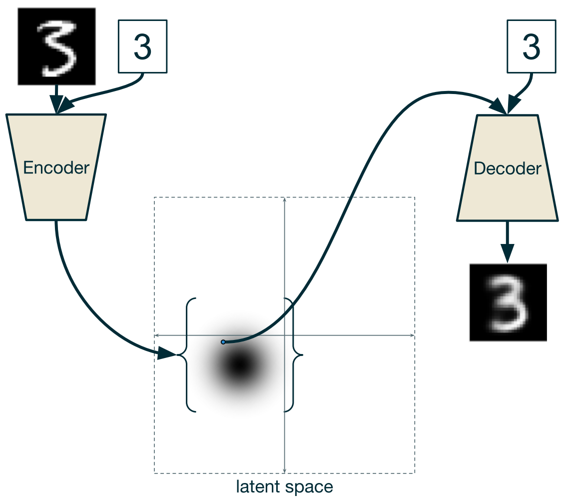
error incurred by using q instead of true distribution p

- Sidestepping sampling – **reparameterization trick**
  - $z \sim \mu + \sigma \cdot \mathcal{N}(0,1)$, then differentiate w. r. t. μ and σ

https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/

Conditional Variational Autoencoders

- Direct control over types of things to generate
- Additional conditioning on a given label/type/class c
  - c can be anything (discrete, continuous…)
    - image class: MNIST digit
    - sentiment
    - “is this a good reply?”
    - coherence level
  - just concatenate to input
  - given to both encoder & decoder at training time
- Generation – need to provide c
  - CVAE will generate a sample of type c
  - Latent space is partitioned by c
    - same latent input with different c will give different results

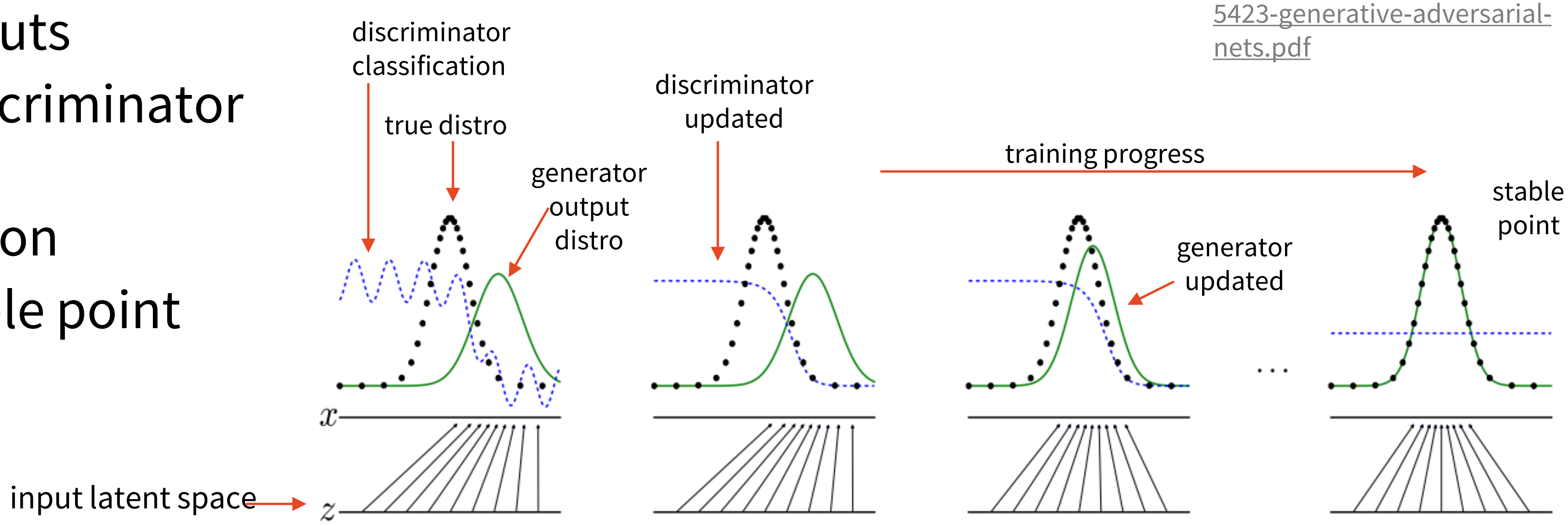


https://ijdykeman.github.io/ml/2016/12/21/cvae.html

Generative Adversarial Nets

- Training generative models to generate **believable** outputs
  - to do so, they necessarily get a better grasp on the distribution
- Getting loss from a 2nd model:
  - **discriminator *D*** – "adversary" classifying real vs. generated samples
  - **generator *G*** – trained to fool the discriminator
    - the best chance to fool the discriminator is to generate likely outputs
- Training iteratively (EM style)
  - generate some outputs
  - classify + update discriminator
  - update generator
    based on classification
  - this will reach a stable point

(Goodfellow et al, 2014)
http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

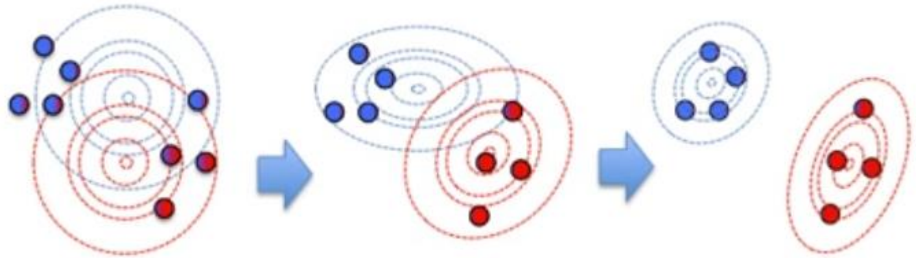


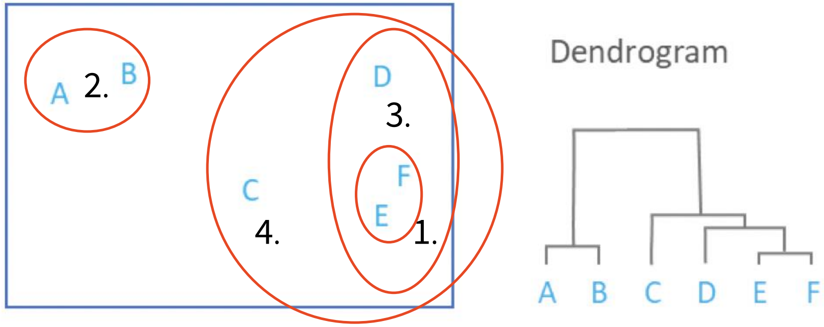
Clustering

https://en.wikipedia.org/wiki/K-means_clustering
https://www.displayr.com/what-is-hierarchical-clustering/
https://towardsdatascience.com/gaussian-mixture-models-d13a5e915c8e

- Unsupervised, finding similarities in data
- basic algorithms
  - **k-means**: assign into k clusters randomly, iterate:
    - compute means (centroids)
    - reassign to nearest centroid
  - **Gaussian mixture**: similar, but soft & variance
    - clusters = multivariate Gaussian distributions
    - estimating probabilities of belonging to each cluster
    - cluster mean/variance based on data weighted by probabilities
  - **hierarchical** (bottom up):
    start with one cluster per instance, iterate:
    - merge 2 closest clusters
    - end when you have k clusters / distance is too big
  - hierarchical top-down (reversed ↵)
- distance metrics & features decide what ends up together

https://www.youtube.com/watch?v=9YA2t78Ha68





Summary

- ML as a function mapping in → out
- Neural networks (function shapes)
  - CNNs, RNNs, encoder-decoder (seq2seq), attention, Transformer
  - input representation: embeddings (+ pretrained, + contextual/LMs: BERT et al.)
- Supervised training
  - cost function
  - gradient descent + learning rate tricks
  - dropout
- Reinforcement learning (more to come later)
- Unsupervised learning
  - autoencoders, variational autoencoders
  - generative adversarial nets
  - clustering

Thanks

Contact us:

https://ufaldsg.slack.com/
{odusek,hudecek}@ufal.mff.cuni.cz
Troja N231/N233 (by agreement)

Labs in 10 mins
Next Tuesday 9:50am

Get the slides here:

http://ufal.cz/npfl099

References/Further:

Goodfellow et al. (2016): Deep Learning, MIT Press (http://www.deeplearningbook.org)
Kim et al. (2018): Tutorial on Deep Latent Variable Models of Natural Language (http://arxiv.org/abs/1812.06834)
Milan Straka's Deep Learning slides: http://ufal.mff.cuni.cz/courses/npfl114/1819-summer

Neural nets tutorials:

- https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0
- https://minitorch.github.io/index.html
- https://objax.readthedocs.io/en/latest/