

# NPFL099 Statistical Dialogue Systems

## 4. Training Neural Nets

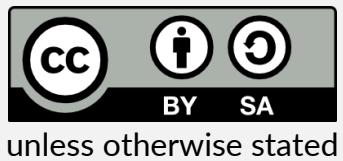
<http://ufal.cz/npfl099>

Ondřej Dušek, Vojtěch Hudeček & Zdeněk Kasner

24. 10. 2022



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

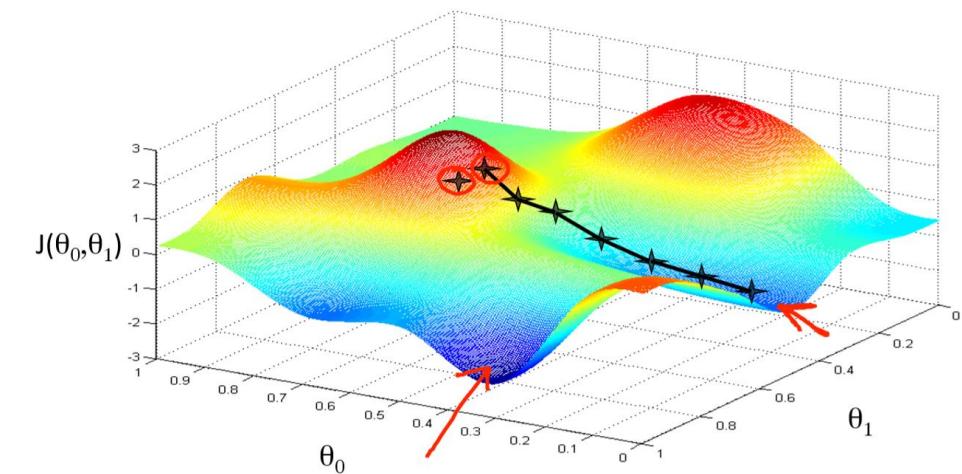


# Recap: Neural Nets

- **complex functions, composed of simple functions** (=layers)
  - linear, ReLU, tanh, sigmoid, softmax
- **fully differentiable**
- different arrangements:
  - feed forward / multi-layer perceptron
  - CNNs
  - RNNs (LSTM/GRU)
  - attention
  - Transformer
- input: binary, float, embedding
- tasks/problems: classification, regression, structured (sequences/ranking)

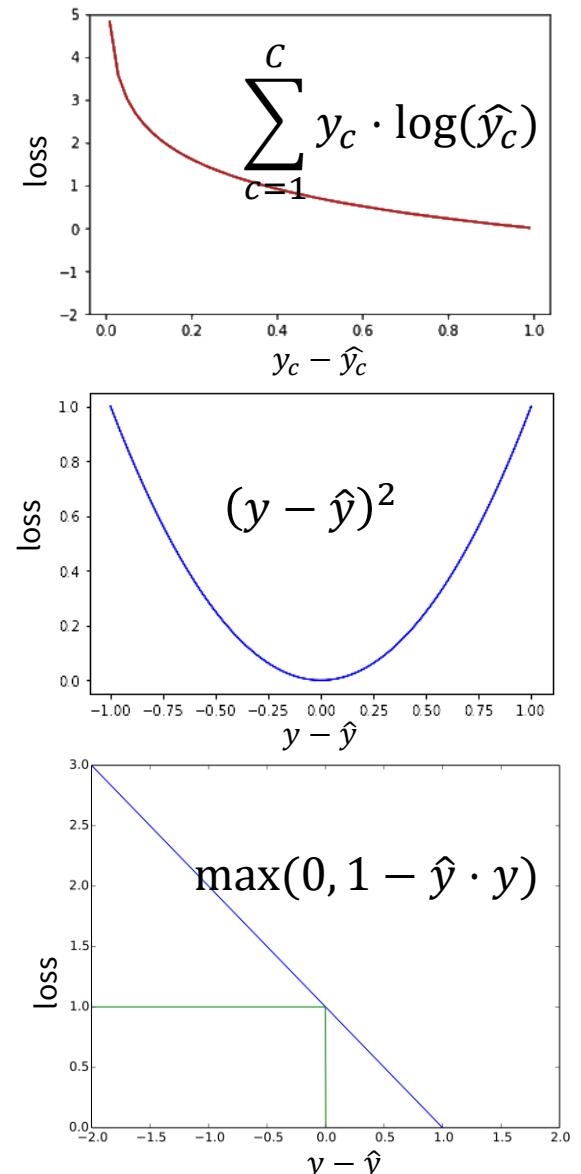
# Supervised Training: Gradient Descent

- 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 (=averaged) 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**
  - batches may be **accumulated** to fit into memory
    - e.g. your GPU only fits one instance  
→ compute gradients multiple times, then do 1 update



# 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  $\Rightarrow$  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>

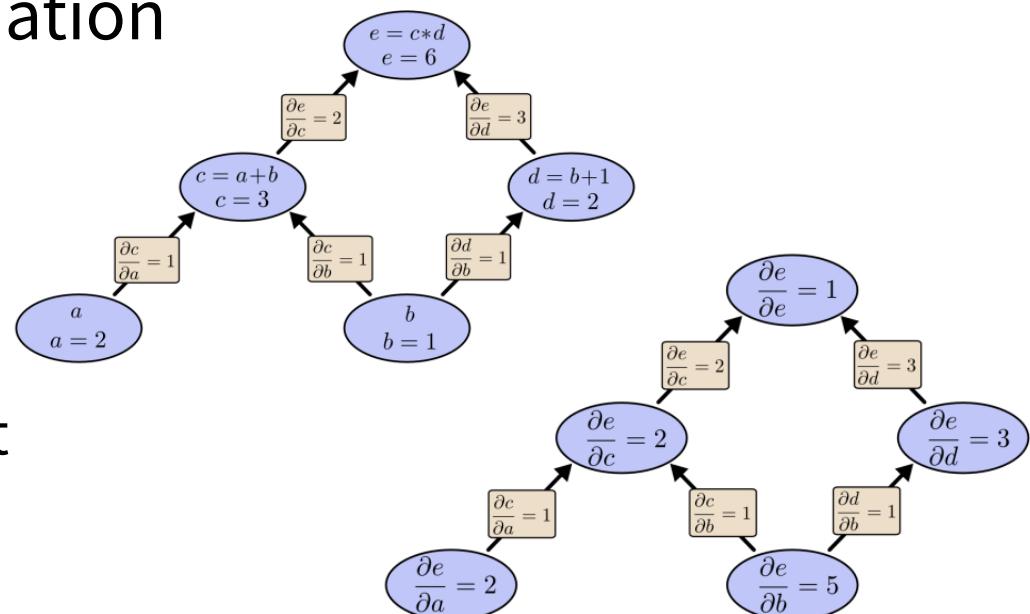
[https://en.wikipedia.org/wiki/Hinge\\_loss](https://en.wikipedia.org/wiki/Hinge_loss)

# Backpropagation

<https://www.mathsisfun.com/calculus/derivatives-rules.html>

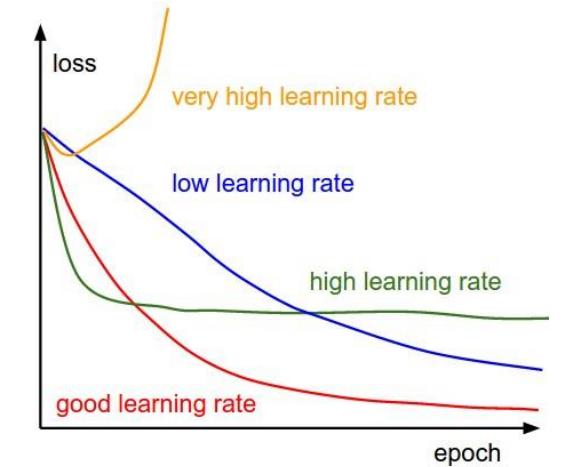
- network ~ computational graph
  - reflects function/layer composition
- composed function derivatives – simple rules
  - basically summing over different paths
  - factoring ~ merging paths at every node
- **backpropagation** = reverse-mode differentiation
  - going back from output to input
  - ~ how every node affects the output
  - your graph **output = cost function**
  - → derivatives of all parameters w. r. t. cost
  - one pass through the network only → easy & fast
  - NN frameworks do this automatically

Rules	Function	Derivative
Multiplication by constant	$cf$	$cf'$
<u>Power Rule</u>	$x^n$	$nx^{n-1}$
Sum Rule	$f + g$	$f' + g'$
Difference Rule	$f - g$	$f' - g'$
<u>Product Rule</u>	$fg$	$f'g + f'g$
Quotient Rule	$f/g$	$\frac{f'g - g'f}{g^2}$
Reciprocal Rule	$1/f$	$-f'/f^2$
Chain Rule (as " <a href="#">Composition of Functions</a> ")	$f \circ g$	$(f' \circ g) \times g'$

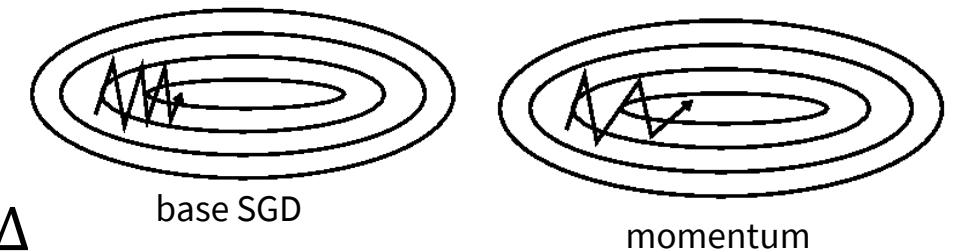


# Learning Rate ( $\alpha$ ) & Momentum

- $\alpha$ : most important parameter in (stochastic) gradient descent
- tricky to tune:
  - too high  $\alpha$  = may not find optimum
  - too low  $\alpha$  = may take forever
- Learning rate decay: start high, lower  $\alpha$  gradually
  - make bigger steps (to speed learning)
  - slow down when you're almost there (to avoid overshooting)
  - linear, stepwise, exponential
  - **reduce-on-plateau** – check every now and then if we're still improving, reduce LR if not
- Momentum: moving average of gradients
  - make learning less erratic
  - $m = \beta \cdot m + (1 - \beta) \cdot \Delta$ , update by  $m$  instead of  $\Delta$



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



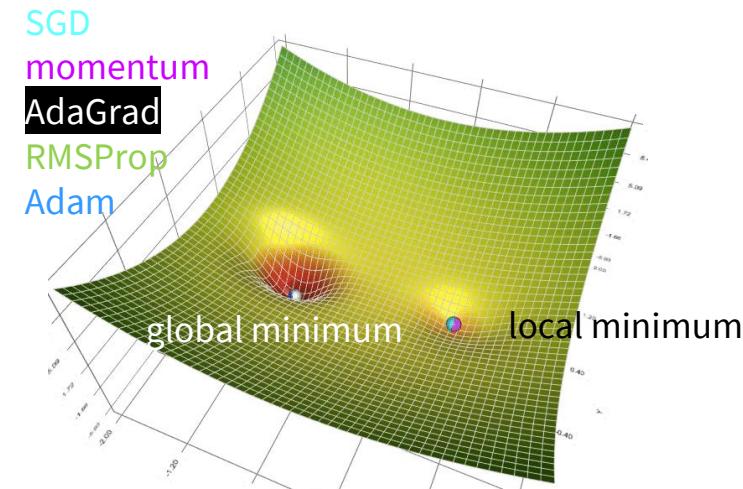
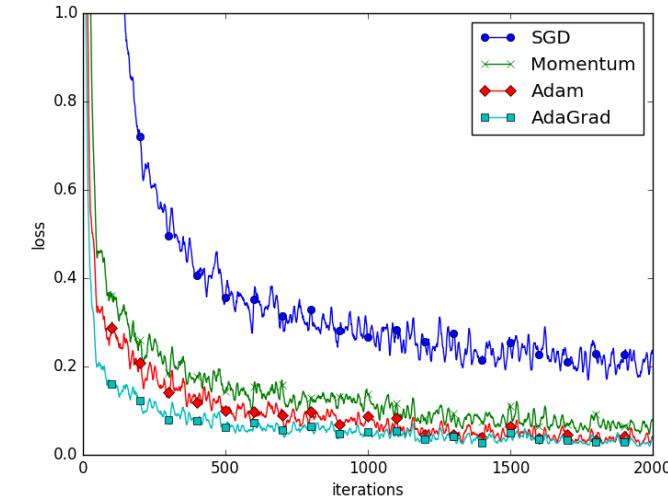
# Optimizers

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

- Better LR management
  - change LR based on gradients, less sensitive to settings
- **AdaGrad** – all history
  - remember sum of total gradients squared:  $\sum_t \Delta_t^2$
  - divide LR by  $\sqrt{\sum \Delta_t^2}$
  - variants: **Adadelta**, **RMSProp** – slower LR drop
- **Adam** – per-parameter momentum
  - moving averages for  $\Delta$  &  $\Delta^2$ :  
 $m = \beta_1 \cdot m + (1 - \beta_1)\Delta, v = \beta_2 \cdot v + (1 - \beta_2)\Delta^2$
  - use  $m$  instead of  $\Delta$ , divide LR by  $\sqrt{v}$
  - used as default in most applications
  - variant: **AdamW** – better regularization
    - not much difference though

(Kingma & Ba, 2015)  
<https://arxiv.org/abs/1412.6980>

(Loshchilov & Hutter, 2019)  
<https://arxiv.org/abs/1711.05101>



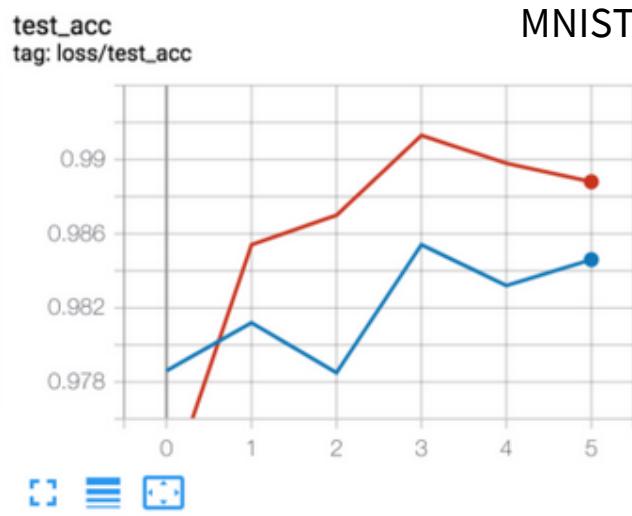
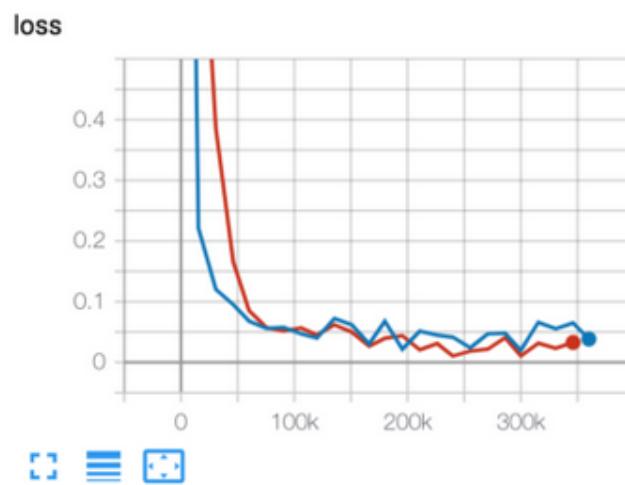
<https://ruder.io/optimizing-gradient-descent/>

<https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c>

# Optimizers

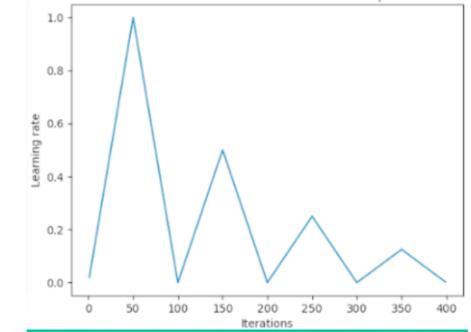
(You et al., 2020)  
<http://arxiv.org/abs/1904.00962>

- **LAMB** – Layer-wise Adaptive Moments for Batches
  - for larger batches & allowing to use larger LR (~unstable otherwise)
- **LARS** layer-wise adaptive rate scaling
  - layer-wise LRs, always multiplied by a **trust ratio**:
$$\alpha^l = \alpha \cdot \frac{\|w^l\|}{\|\Delta^l\|}$$
 – norm of weights/ norm of gradients
  - higher trust ratio = faster updates
  - start of training:  
low  $w$ , high  $\Delta \rightarrow$  slow **warm up**
  - towards convergence:  
higher  $w$ , low  $\Delta \rightarrow$  faster training
- $\text{LAMB} \approx \text{LARS} + \text{AdamW}$

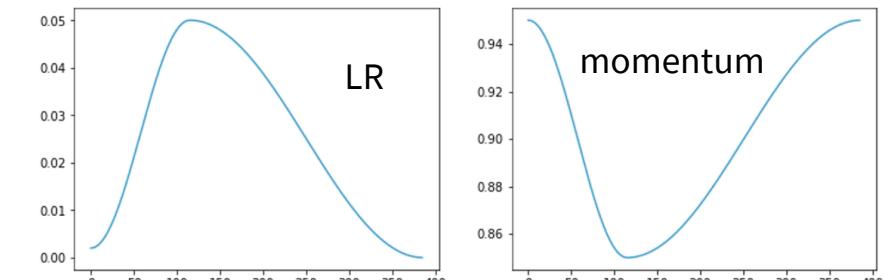
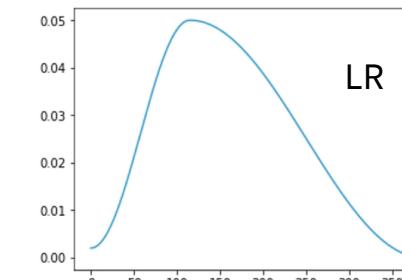


# Schedulers

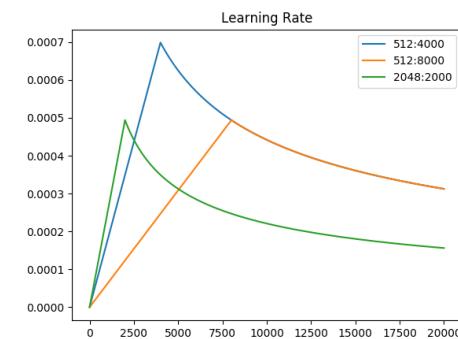
- more fiddling with LR – **warm-ups**
  - start learning slowly, then increase LR, then reduce again
  - may be repeated (**warm restarts**), with lowered maximum LR
    - allow to diverge slightly – work around local minima
- multiple options:
  - cyclical (=warm restarts) – linear, cosine annealing
  - **one cycle** – same, just don't restart
  - **Noam scheduler** – linear warm-up, decay by  $\sqrt{\text{steps}}$
- combine with base SGD or Adam/Adadelta etc.
  - momentum updated inversely to LR
  - may have less effect with optimizers
    - trade-off: speed vs. sensitivity to parameter settings



cyclical scheduler (warm restarts)



one cycle with cosine annealing

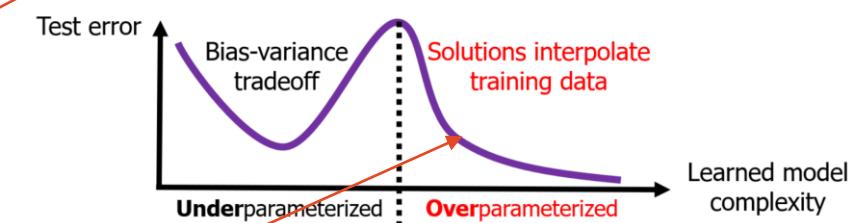
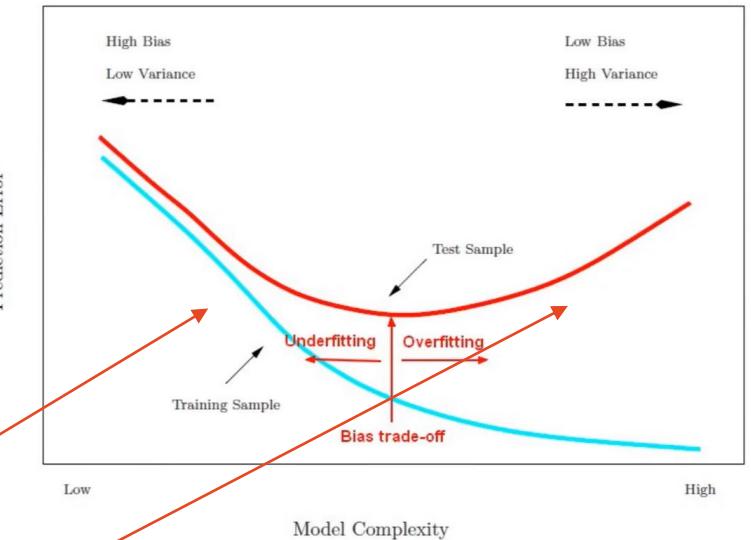


Noam scheduler with different parameters

# When to stop training

<https://www.andreaperlato.com/theorypost/bias-variance-trade-off/>

- generally, when cost stops going down
  - despite all the LR fiddling
- problem: **overfitting**
  - cost low on training set, high on validation set
  - network essentially memorized the training set
  - → **check on validation set** after each epoch (pass through data)
  - stop when cost goes up on validation set
  - regularization (see →) helps delay overfitting
- **bias-variance** trade-off:
  - smaller models may underfit (high bias, low variance = not flexible enough)
  - larger models likely to overfit (too flexible, memorize data)
  - XXL models: overfit sooo much they actually interpolate data → good (🤔?)

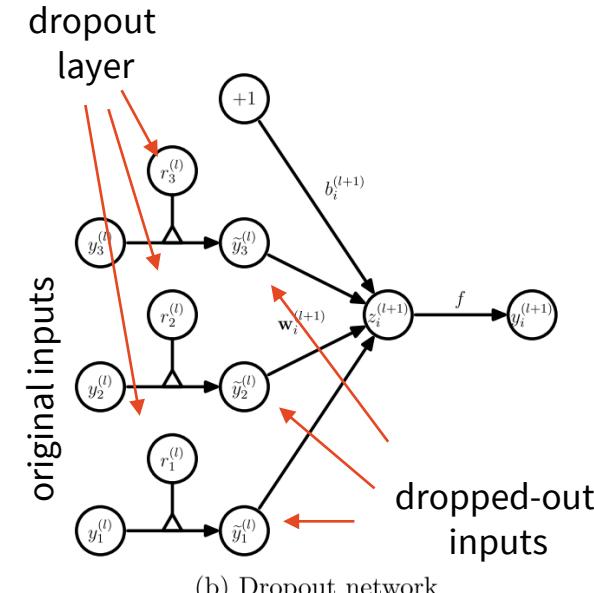
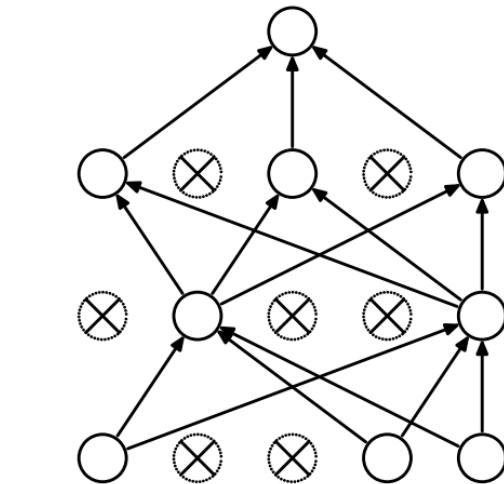


(Dar et al., 2021) <https://arxiv.org/abs/2109.02355>

# Regularization: Dropout

- regularization: preventing overfitting
  - making it harder for the network to learn, adding noise
- **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

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

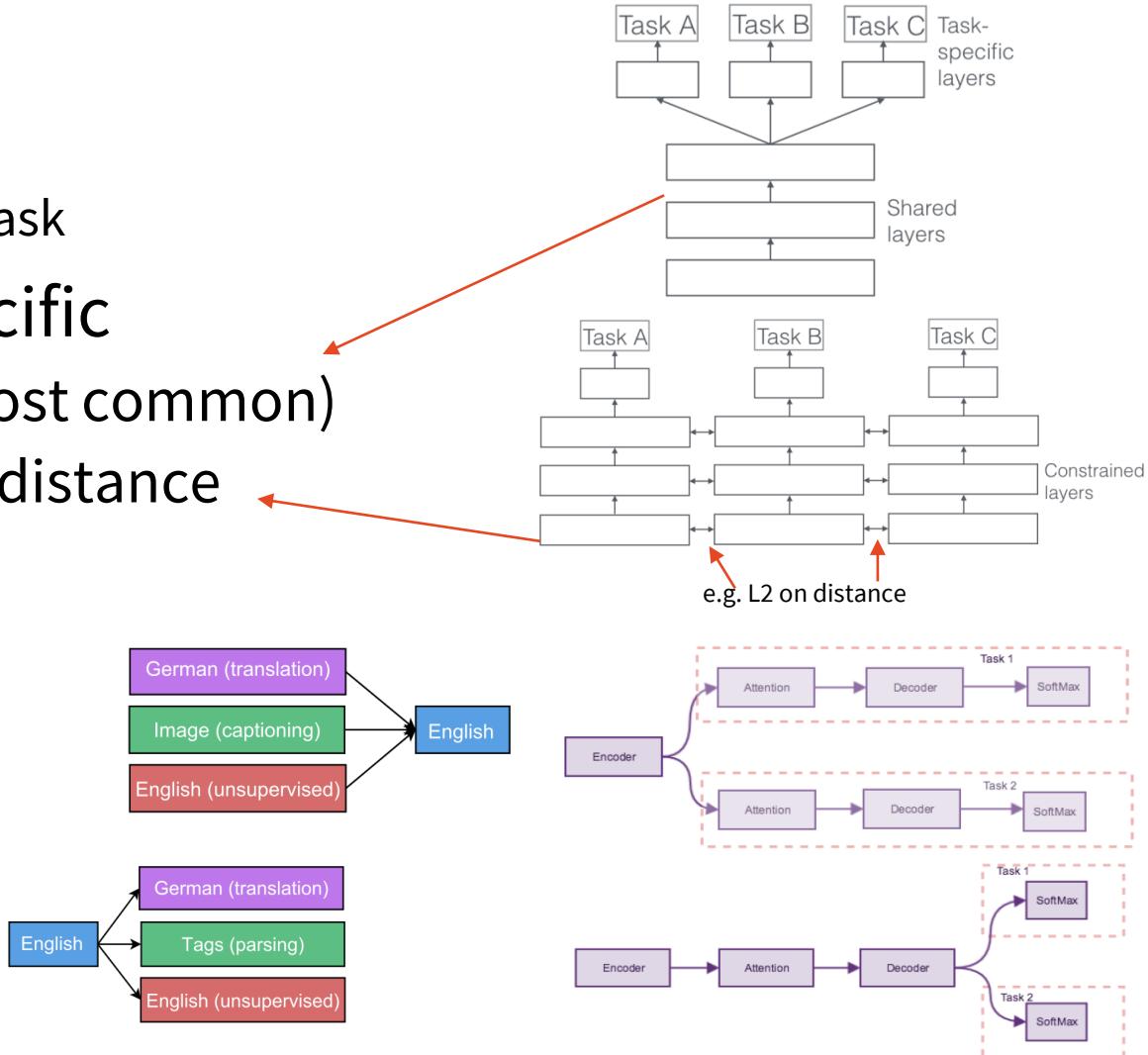


(b) Dropout network

# Regularization: Multi-task Learning

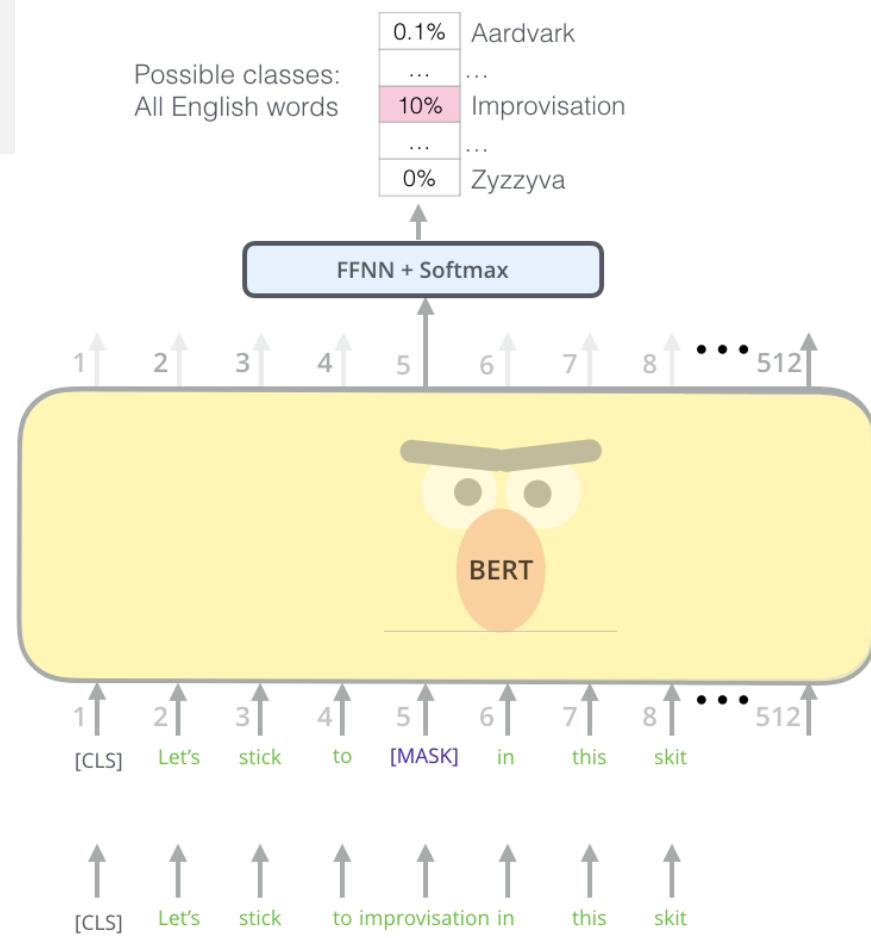
(Ruder, 2017)  
<http://arxiv.org/abs/1706.05098>  
(Fan et al., 2017)  
<http://arxiv.org/abs/1706.04326>  
(Luong et al., 2016)  
<http://arxiv.org/abs/1511.06114>

- 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"



# Self-supervised training

- train supervised, but **don't provide labels**
  - use naturally occurring labels
  - create labels automatically somehow
    - corrupt data & learn to fix them
    - learn from rule-based annotation (not ideal!)
  - use specific tasks that don't require manual labels
- good to train on huge amounts of data
  - language modelling
    - next-word prediction
    - MLM – masked word prediction (~like word2vec)
  - **autoencoding**: predict your own input (see →)
- good to **pretrain** the network for a final task
- unsupervised, but with supervised approaches



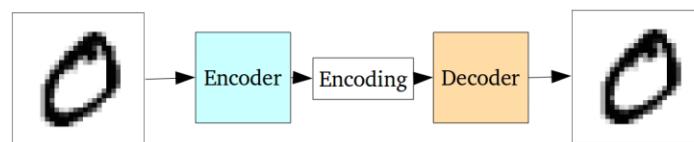
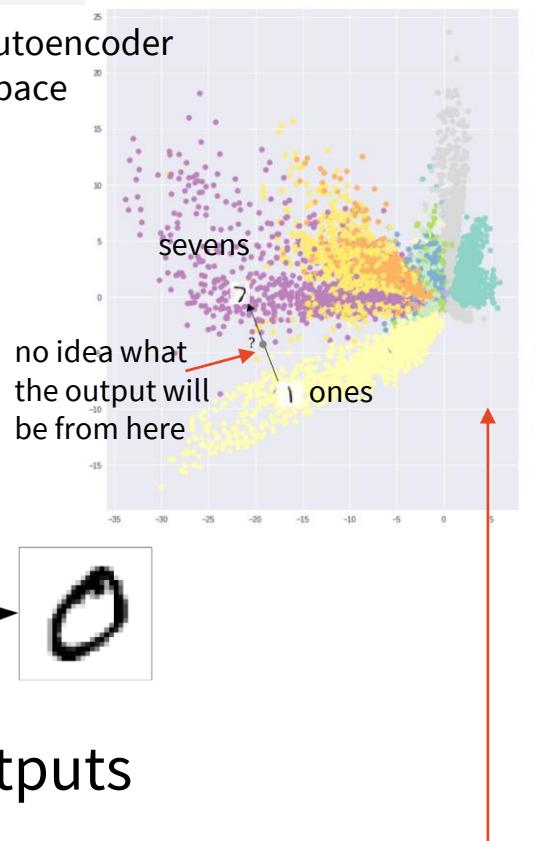
<http://jalammar.github.io/illustrated-bert/>

<https://ai.stackexchange.com/questions/10623/what-is-self-supervised-learning-in-machine-learning>

# Autoencoders

- 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
- **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
- extension – **denoising autoencoder:**
  - add noise to inputs, train to generate clean outputs
  - use in multi-task learning, representations for use in downstream tasks

MNIST digits autoencoder  
latent space

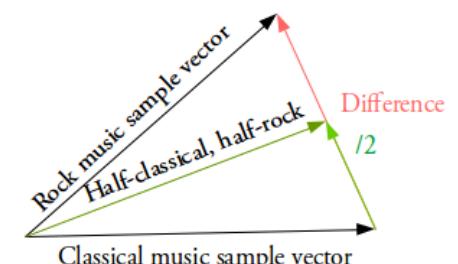
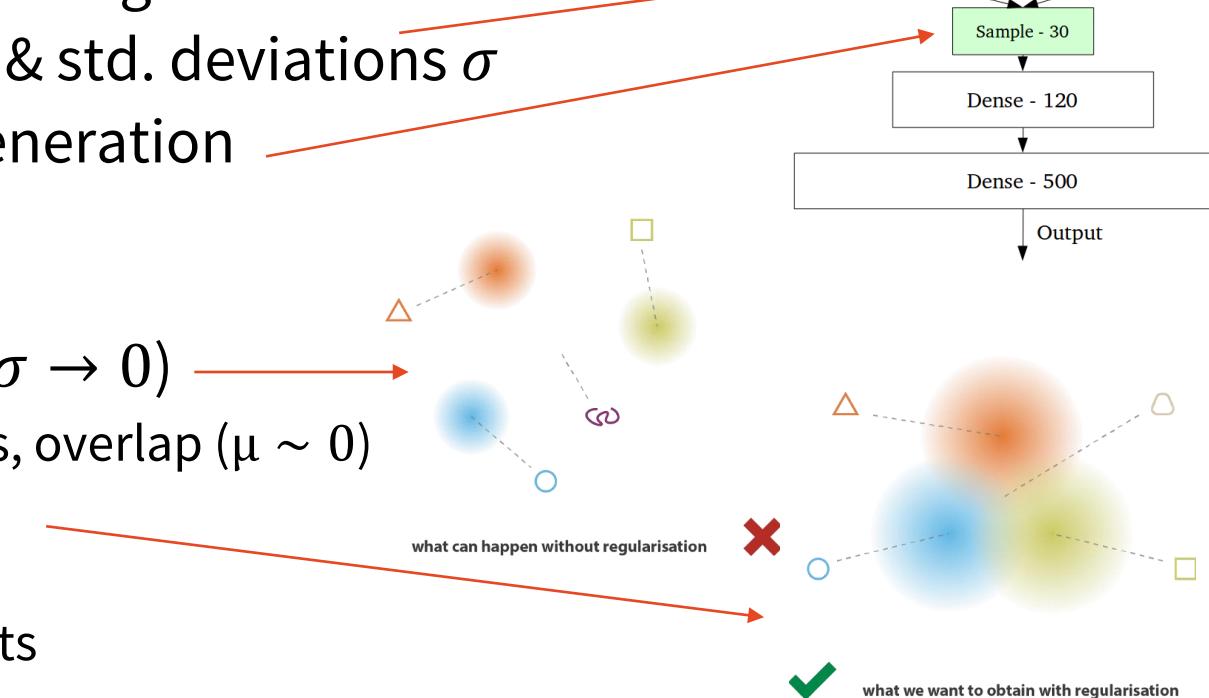
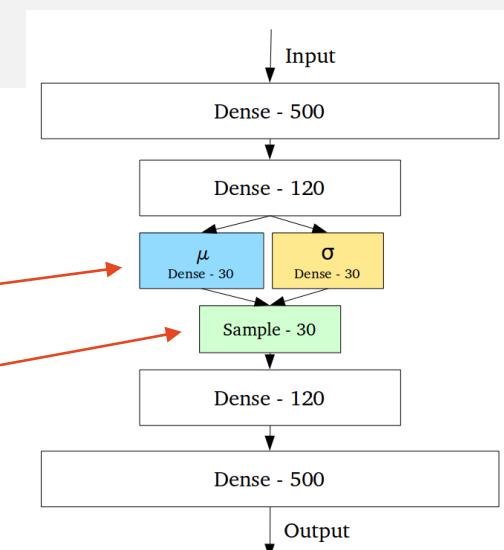


# Variational Autoencoders

- Making the encoding latent space more useful

- using **Gaussians** – continuous space by design
- encoding input into vectors of means  $\mu$  & std. deviations  $\sigma$
- 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  $\sigma$ , smoothness, overlap ( $\mu \sim 0$ )
  - 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



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

# VAE details

- VAE objective:

- “AE” {
  - **reconstruction loss** (maximizing  $p(x|z)$  in the decoder), MLE as per usual
- “V” {
  - **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:

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

↑  
“evidence”  
(i.e. data)      ELBO

Normal noise

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.  $\mu$  and  $\sigma$ 
  - differentiating w. r. t.  $\mu$  &  $\sigma$  still works, no hard sampling on that path

# Discrete VAE: Gumbel-Softmax

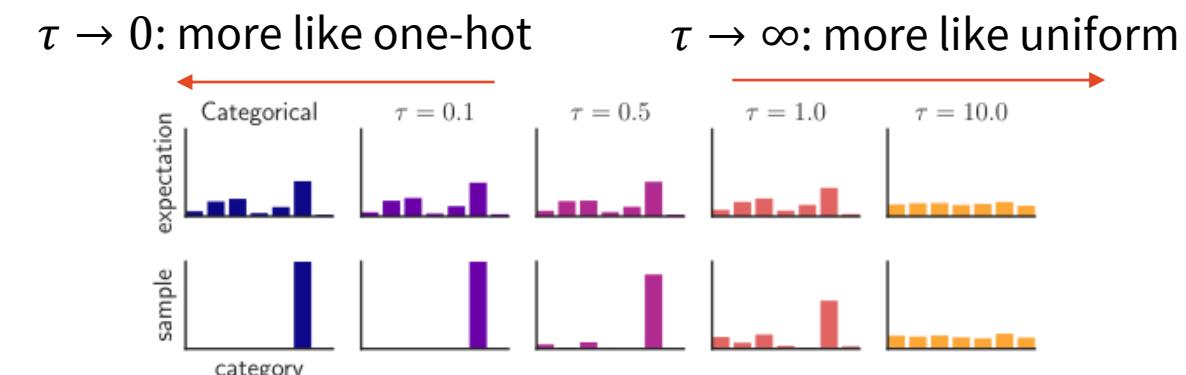
(Jang et al., 2017)  
<https://arxiv.org/abs/1611.01144>

- “reparameterization trick for discrete distributions”
  - same idea, just with a **discrete/categorial distribution**
  - this makes the latent space better interpretable
- **Gumbel-max trick:**
  - categorial distribution  $\pi$  with probabilities  $\pi_i$
  - sampling from  $\pi$ :  $z = \text{onehot}(\arg \max_i (\log \pi_i + g_i))$
- Swap argmax for softmax with temperature  $\tau$ 
  - differs from  $\pi$  if  $\tau > 0$ , but may be close
  - approx. sample of the true distribution
  - fully differentiable
  - $g_i$  bypassed in differentiation, same as  $\mathcal{N}(0,1)$  in Gaussian sampling

Gumbel noise:

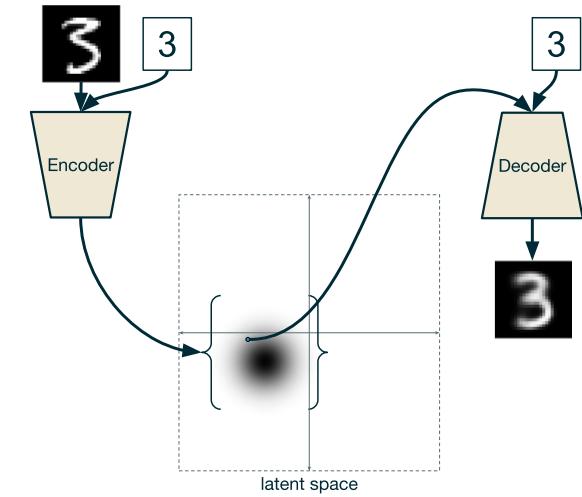
$$g_i = -\log(-\log(\text{Uniform}(0,1)))$$

$$y_i = \frac{\exp\left(\frac{\log(\pi_i) + g_i}{\tau}\right)}{\sum_{j=1}^N \exp\left(\frac{\log(\pi_j) + g_j}{\tau}\right)}$$



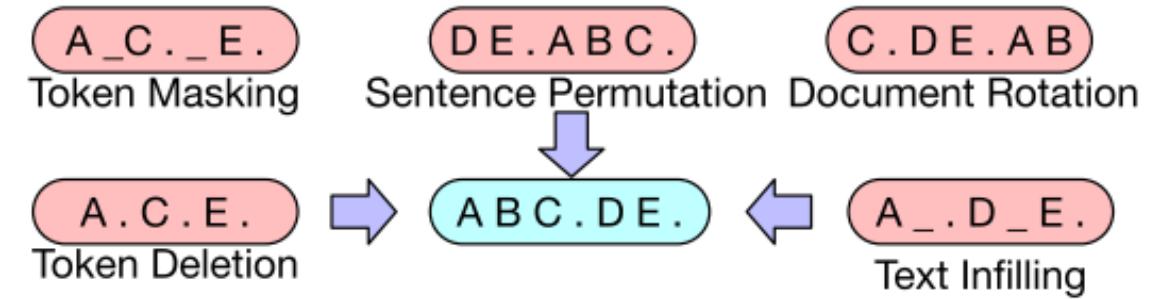
# Conditional Variational Autoencoders

- 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



# Pretraining & Finetuning

- 2-step training:
  1. **Pretrain** a model on a huge dataset (**self-supervised**, language-based tasks)
  2. **Fine-tune** for your own task on your smaller data (**supervised**)
- ~pretrained embeddings, many variants
  - mostly Transformer architecture
  - pretraining tasks vary and make a difference
- typical tasks:
  - masked language modelling  
(masked words/spans)
  - next-word prediction
  - denoising scrambled texts



(Lewis et al., 2020) <https://aclanthology.org/2020.acl-main.703/>

# Pretrained Language Models

- **BERT + RoBERTa** (optimized): Transformer encoder
  - masked word prediction, sentence order
- **GPT(-2/-3)**: Transformer decoder only
  - next-word prediction (=language modeling)
- **BART** – encoder-decoder  $\approx$  Transformer denoising autoencoder
  - masking, word removal...  $\rightarrow$  generate original sentence
- **T5**: generalization of  $\uparrow$  (multi-task)
  - prompts for different tasks
- multilingual: **XLM-RoBERTa, mBART, mT5**
- **ByT5**: byte-level (larger encoder)
- a lot of pretrained models released plug-and-play
  - you only need to finetune (and sometimes, not even that)

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

(Liu et al., 2019) <http://arxiv.org/abs/1907.11692>  
(Rogers et al., 2020) <http://arxiv.org/abs/2002.12327>

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

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

(Lewis et al., 2020) <https://aclanthology.org/2020.acl-main.703/>

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

(Conneau et al., 2020) <https://www.aclweb.org/anthology/2020.acl-main.747>  
(Liu et al., 2020) <http://arxiv.org/abs/2001.08210>  
<https://aclanthology.org/2021.nacl-main.41>

(Xue et al., 2022) <https://arxiv.org/abs/2105.13626>

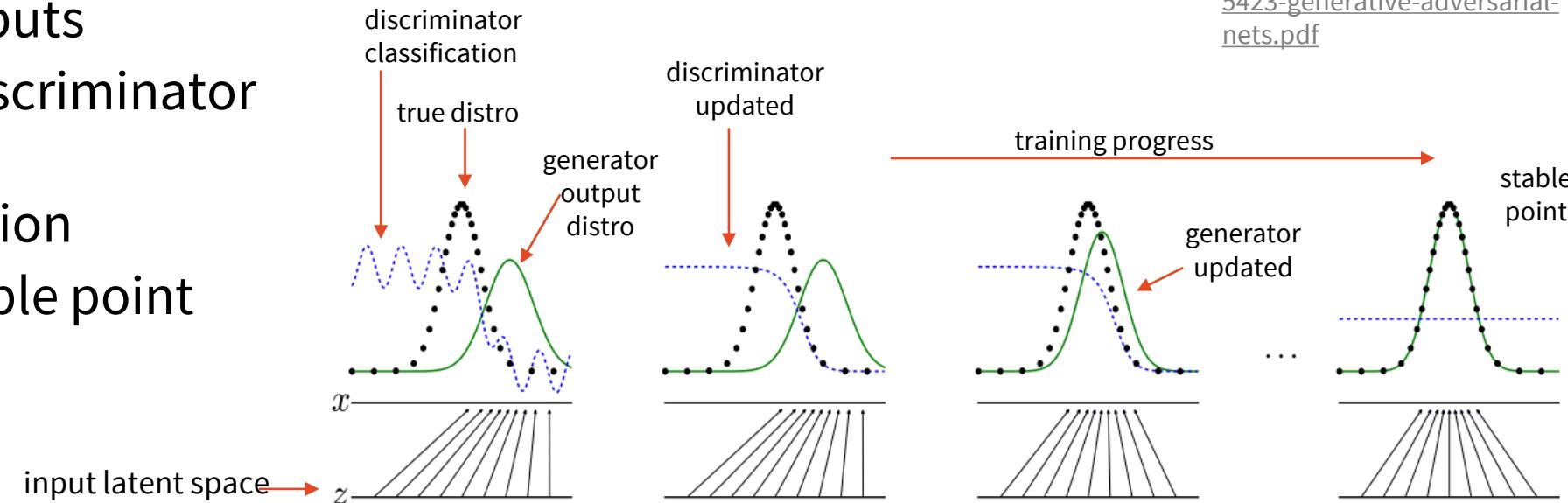


<https://github.com/huggingface/transformers>

# 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>

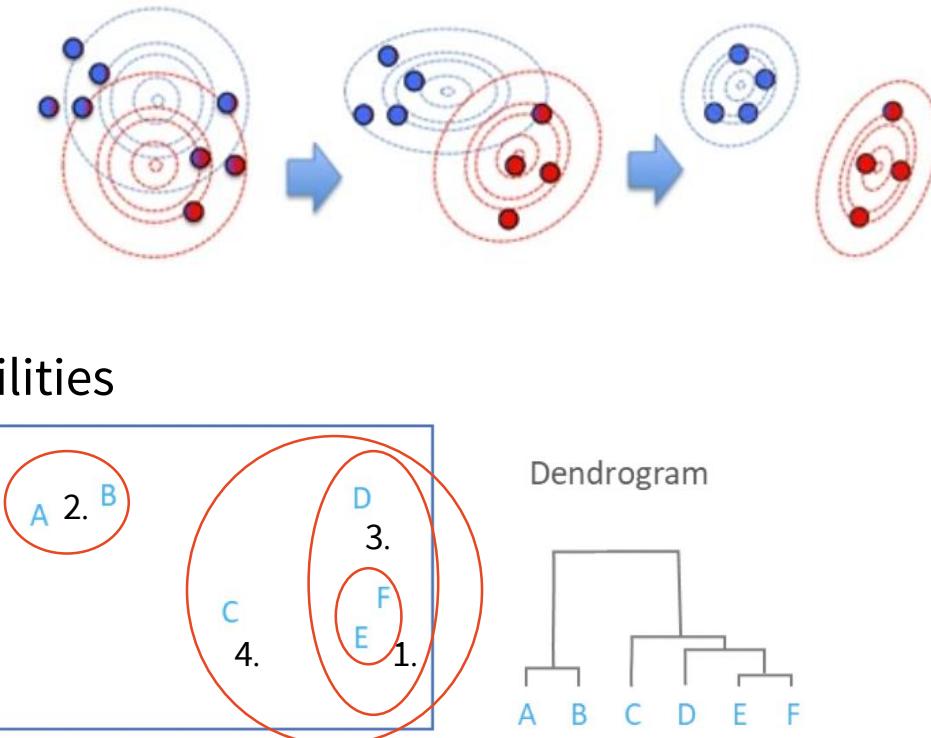


# Clustering

[https://en.wikipedia.org/wiki/K-means\\_clustering](https://en.wikipedia.org/wiki/K-means_clustering)  
<https://www.displayr.com/what-is-hierarchical-clustering/>  
<https://towardsdatascience.com/gaussian-mixture-models-d13a5e915c8e>

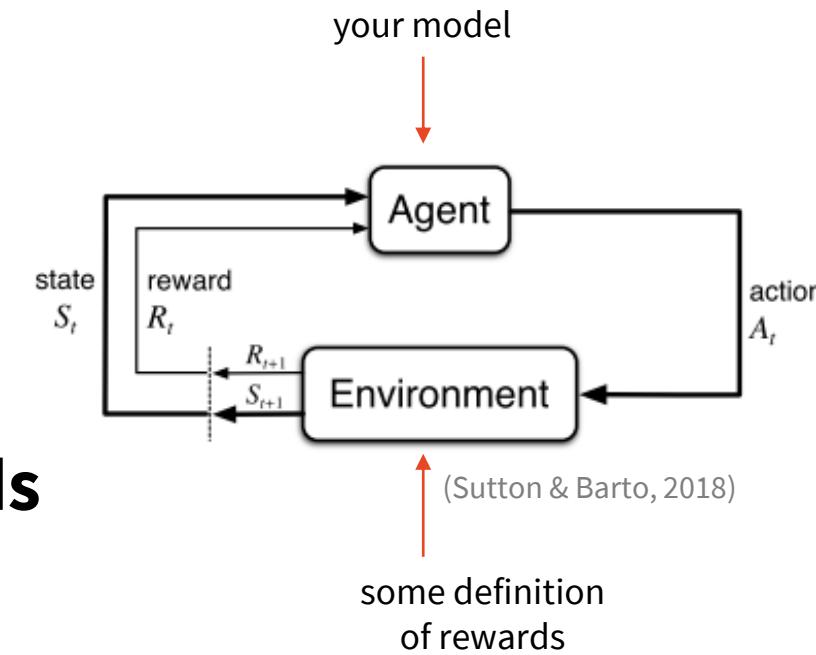
- Unsupervised, finding similarities in data
- basic algorithms
  - **k-means**: assign into  $k$  clusters randomly, iterate:
    - compute means (centroids)
    - reassign to nearest centroid
  - **Gaussian mixture**: similar, but soft & variance
    - clusters = multivariate Gaussian distributions
    - 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  $\rightarrow$ )
  - distance metrics & features decide what ends up together

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



# 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



# Summary

- Supervised training
  - cost function
  - stochastic **gradient descent** – minibatches
  - backpropagation
  - **learning rate** tricks – optimizers (Adam), schedulers
  - regularization: dropout, multi-task training
- Self-supervised learning (~kinda unsupervised)
  - autoencoders, denoising, variational autoencoders
  - (masked) language models
- Unsupervised
  - generative adversarial nets
  - clustering
- Reinforcement learning (more to come later)

# Thanks

## Contact us:

<https://ufaldsg.slack.com/>

{odusek,hudecek,kasner}@ufal.mff.cuni.cz

Zoom/Skype/Troja

**Labs in 10 mins**  
**Next Monday 12:20**

## 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/>