# Lecture 9. Backpropagation, Training deep networks

COMP90051 Statistical Machine Learning

Semester 1, 2021
Lecturer: Trevor Cohn


THE UNIVERSITY OF MELBOURNE

# This lecture

- Deep learning
  - Representation capacity
  - Deep models and representation learning
- Backpropagation
  - Step-by-step derivation
- Training
  - Optimisation methods
  - Regularisation

# Deep Learning and Representation Learning

Hidden layers viewed as
feature space transformation

# Representational capacity

- ANNs with a single hidden layer are universal approximators
- For example, such ANNs can represent any Boolean function

$$OR(x_1, x_2) \qquad u = g(x_1 + x_2 - 0.5)$$

$$AND(x_1, x_2) \qquad u = g(x_1 + x_2 - 1.5)$$

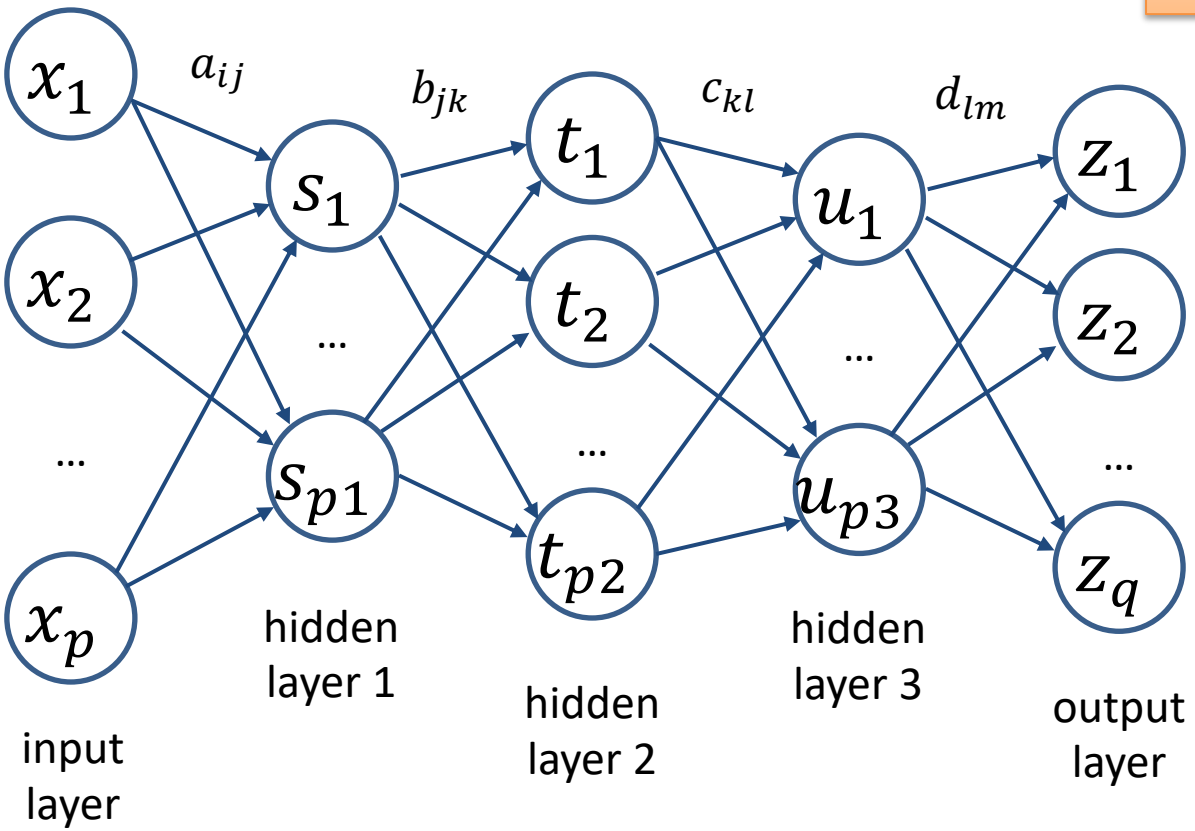
$$NOT(x_1) \qquad u = g(-x_1)$$

$g(r) = 1$ if $r \geq 0$ and $g(r) = 0$ otherwise

- Any Boolean function over $m$ variables can be implemented using a hidden layer with up to $2^m$ elements
- More *efficient* to stack several hidden layers

# Deep networks

"Depth" refers to number of hidden layers




$$\boldsymbol{s} = \tanh(\boldsymbol{A}'\boldsymbol{x}) \quad \boldsymbol{t} = \tanh(\boldsymbol{B}'\boldsymbol{s}) \quad \boldsymbol{u} = \tanh(\boldsymbol{C}'\boldsymbol{t}) \quad \boldsymbol{z} = \tanh(\boldsymbol{D}'\boldsymbol{u})$$

# Deep ANNs as representation learning

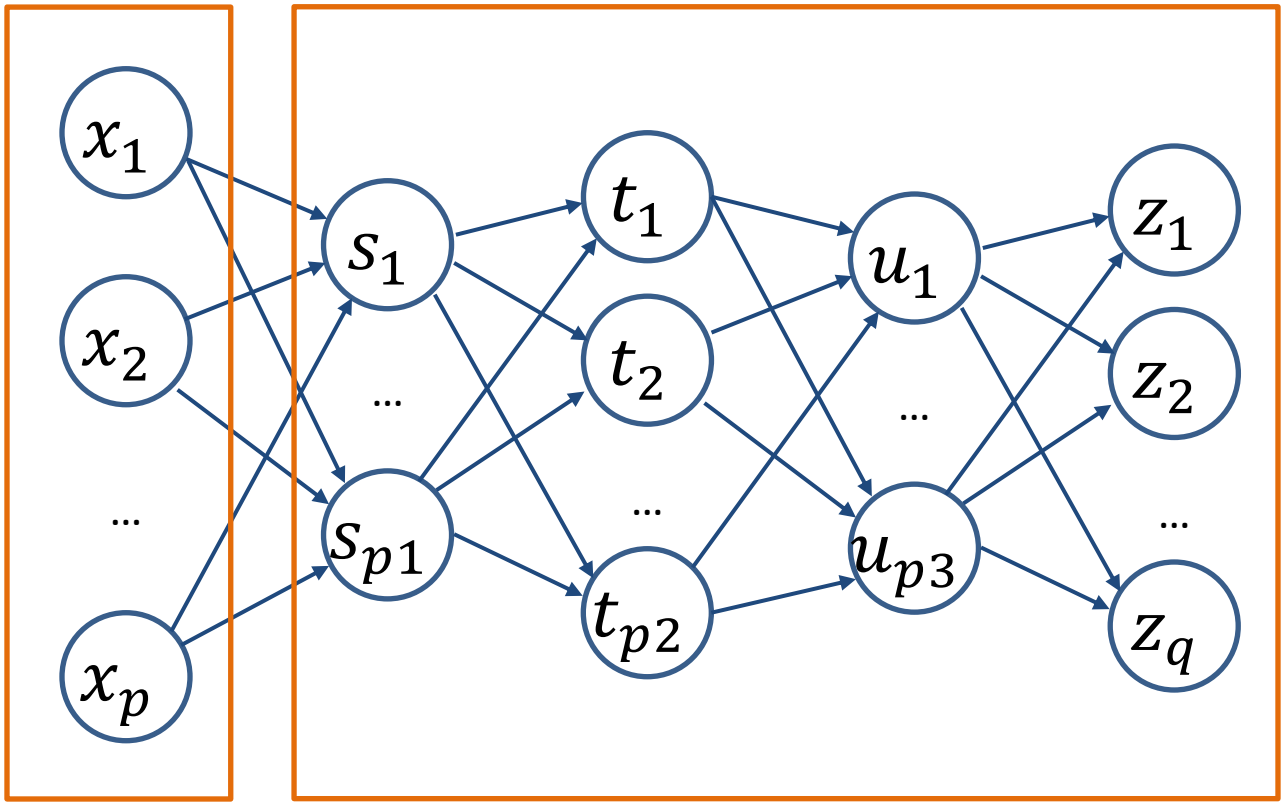
- Consecutive layers form *representations* of the input of increasing complexity
- An ANN can have a simple *linear* output layer, but using complex *non-linear* representation

$$\boldsymbol{z} = \tanh\left(\boldsymbol{D}'\left(\tanh\left(\boldsymbol{C}'\left(\tanh\left(\boldsymbol{B}'\left(\tanh\left(\boldsymbol{A}'\boldsymbol{x}\right)\right)\right)\right)\right)\right)\right)$$

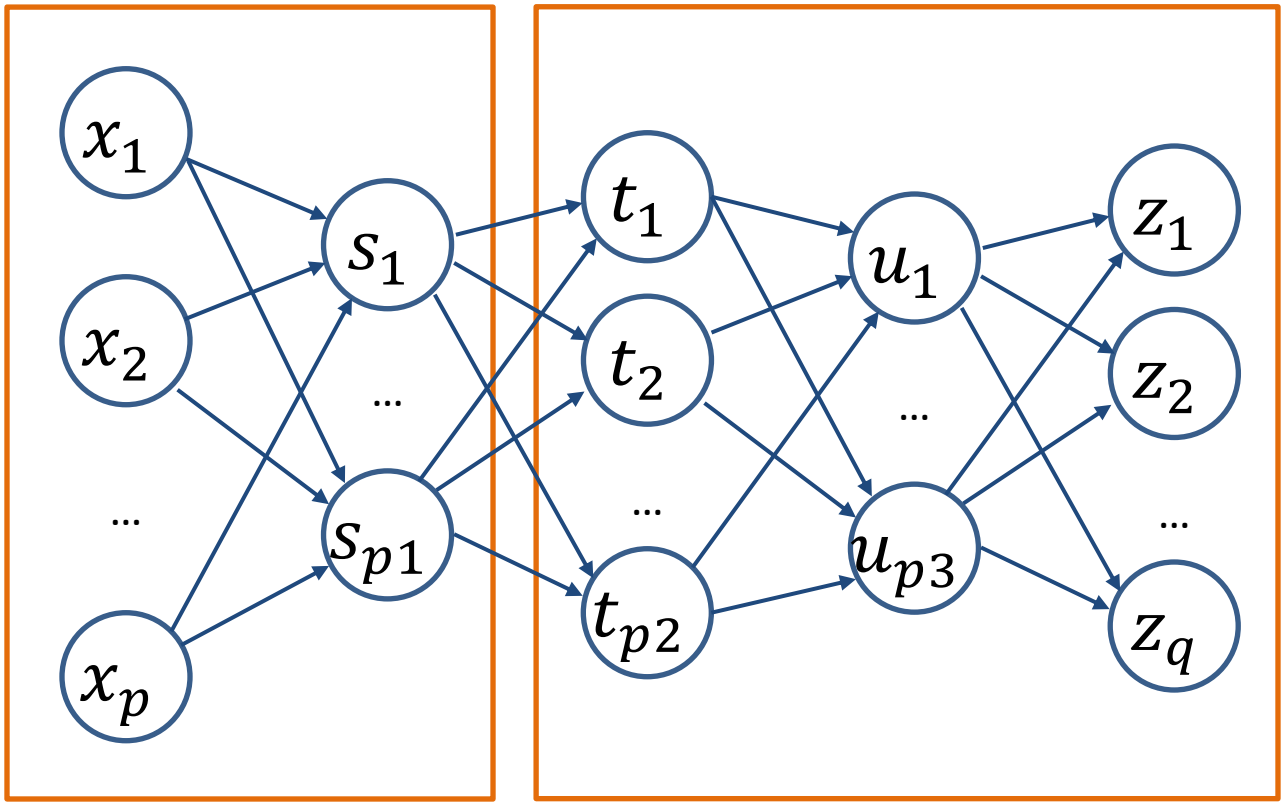
- Equivalently, a hidden layer can be thought of as the transformed feature space, e.g., $\boldsymbol{u} = \varphi(\boldsymbol{x})$
- Parameters of such a transformation are learned from data

Bias terms are omitted for simplicity

# ANN layers as data transformation



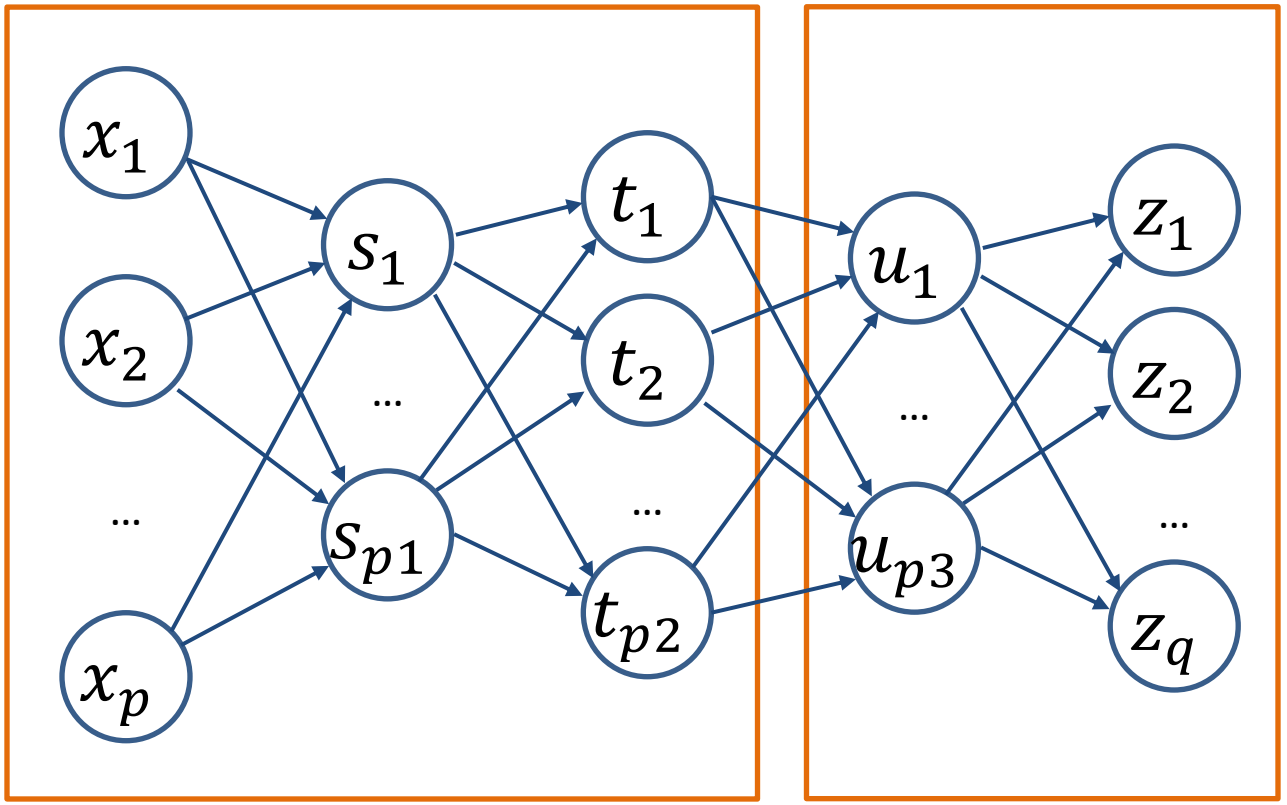
# ANN layers as data transformation



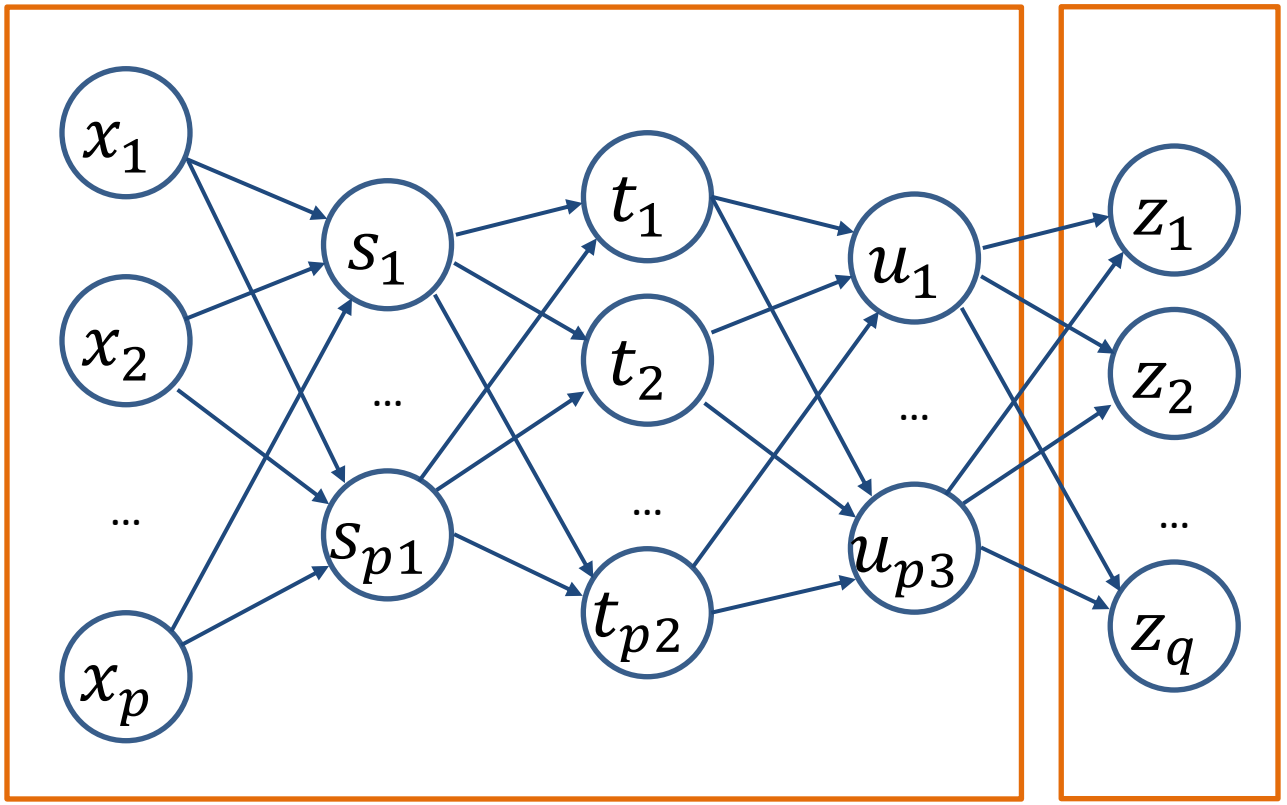

pre-processed data

the model

# ANN layers as data transformation



# ANN layers as data transformation



# Depth vs width

- A single infinitely wide layer in theory gives a universal approximator
- However (empirically) depth yields more accurate models

  Biological inspiration from the eye:
  - first detect small edges and color patches;
  - compose these into smaller shapes;
  - building to more complex detectors, of e.g. textures, faces, etc.
- Seek to mimic layered complexity in a network
- However *vanishing gradient problem* affects learning with very deep models

# Vs manual feature representation

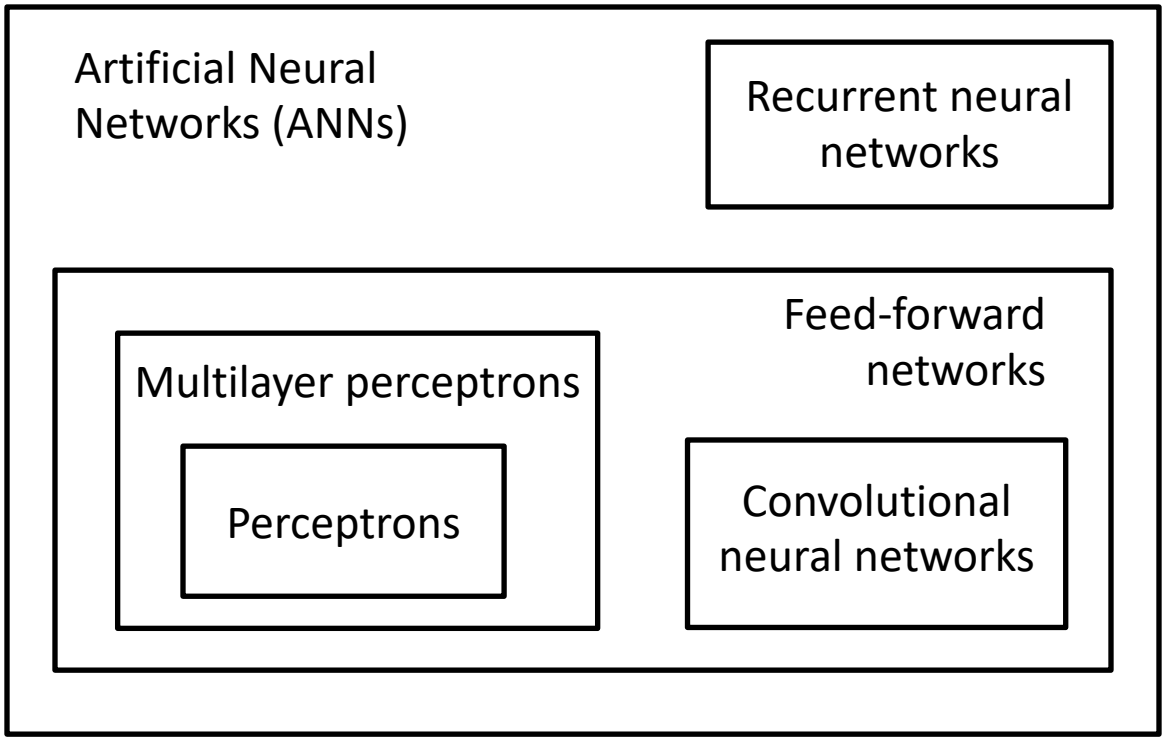
- Standard pipeline
  - input → feature engineering → classification algorithm
- Deep learning automates feature engineering
  - no need for expert analysis

| | | | |
|---|---|---|---|
| forehead_color | **black** | **black** | **black** |
| breast_pattern | **solid** | **solid** | **solid** |
| breast_color | **white** | **white** | **white** |
| head_pattern | **plain** | **capped** | plain |
| back_color | **white** | white | **black** |
| wing_color | grey/white | **grey** | **white** |
| leg_color | **orange** | orange | **orange** |
| size | **medium** | large | **medium** |
| bill_shape | needle | dagger | **dagger** |
| wing_shape | **pointed** | **tapered** | **long** |
| ... | ... | ... | ... |
| primary_color | white | white | white |

→ class = *Artic Tern*


https://vision.cornell.edu/se3/wp-content/uploads/2014/09/WelinderEtal10_CUB-200.pdf

# Animals in the zoo




art: OpenClipartVectors at pixabay.com (CC0)

- An autoencoder is an ANN trained in a specific way.
  - E.g., a multilayer perceptron can be trained as an autoencoder, or a recurrent neural network can be trained as an autoencoder.

# Backpropagation

= “backward propagation of errors”

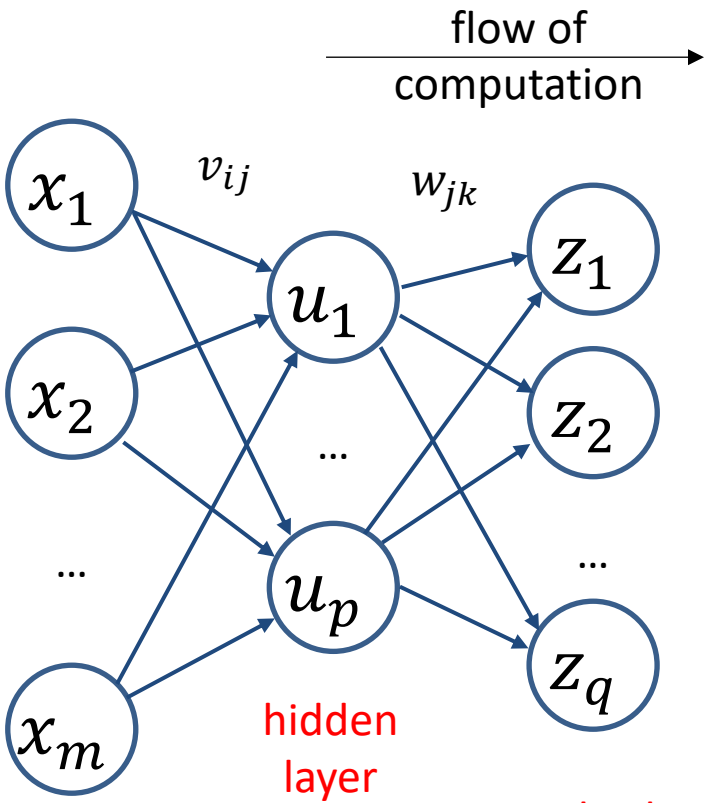
Calculating the gradient
of loss of a composition

# Recap: Feed-forward Artificial Neural Network

$x_i$ are inputs, i.e., attributes

note: here $x_i$ are components of a single training instance $\boldsymbol{x}$

a training dataset is a set of instances



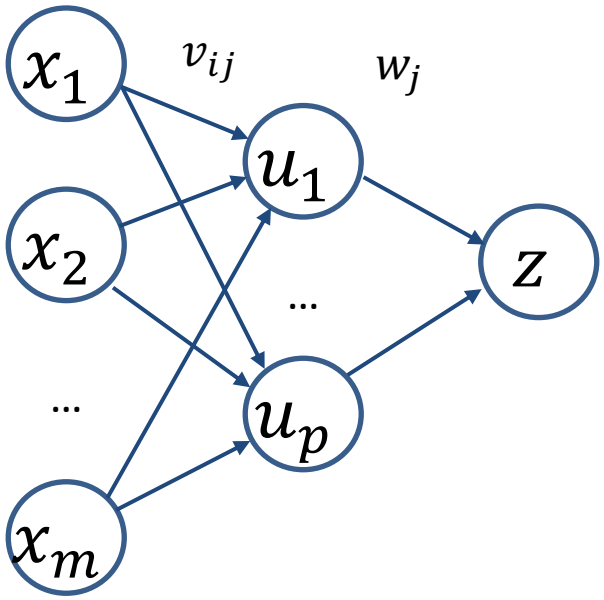

(ANNs can have more than one hidden layer)

$z_i$ are outputs, i.e., predicted labels

Loss function L used to compare outputs with gold, gradient used for training **v**, **w**

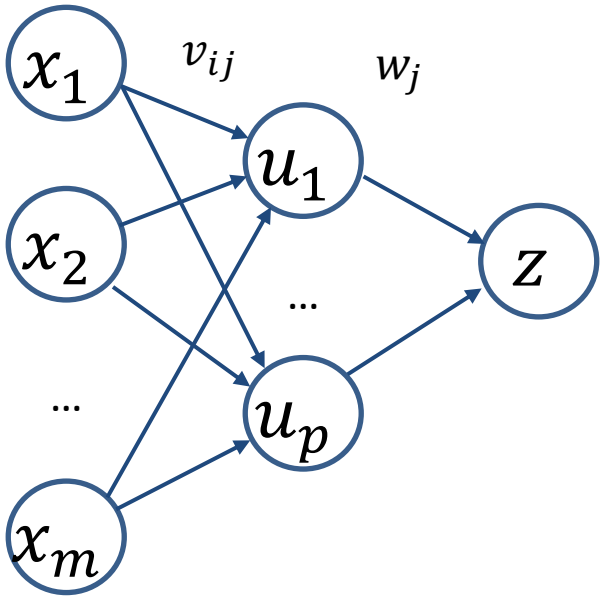
# Backpropagation: start with the chain rule

- Recall that the output $z$ of an ANN is a function composition, and hence $L(z)$ is also a composition
  - $L = 0.5(z-y)^2 = 0.5(h(s)-y)^2 = 0.5(s-y)^2$
  - $= 0.5\left(\sum_{j=0}^{p} u_j w_j - y\right)^2 = 0.5\left(\sum_{j=0}^{p} g(r_j) w_j - y\right)^2 = \cdots$
- Backpropagation makes use of this fact by applying the chain rule for derivatives
- $\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial s}\frac{\partial s}{\partial w_j}$
- $\frac{\partial L}{\partial v_{ij}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial s}\frac{\partial s}{\partial u_j}\frac{\partial u_j}{\partial r_j}\frac{\partial r_j}{\partial v_{ij}}$



# Backpropagation: intermediate step

- Apply the chain rule
- $\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial s}\frac{\partial s}{\partial w_j}$
- $\frac{\partial L}{\partial v_{ij}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial s}\frac{\partial s}{\partial u_j}\frac{\partial u_j}{\partial r_j}\frac{\partial r_j}{\partial v_{ij}}$




- Now define

$$\delta \equiv \frac{\partial L}{\partial s} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial s}$$

$$\varepsilon_j \equiv \frac{\partial L}{\partial r_j} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial s}\frac{\partial s}{\partial u_j}\frac{\partial u_j}{\partial r_j}$$

- Here $L = 0.5(z-y)^2$ and $z = s$

  Thus $\delta = (z-y)$

- Here $s = \sum_{j=0}^{p} u_j w_j$ and $u_j = g(r_j)$

  Thus $\varepsilon_j = \delta w_j g'(r_j)$

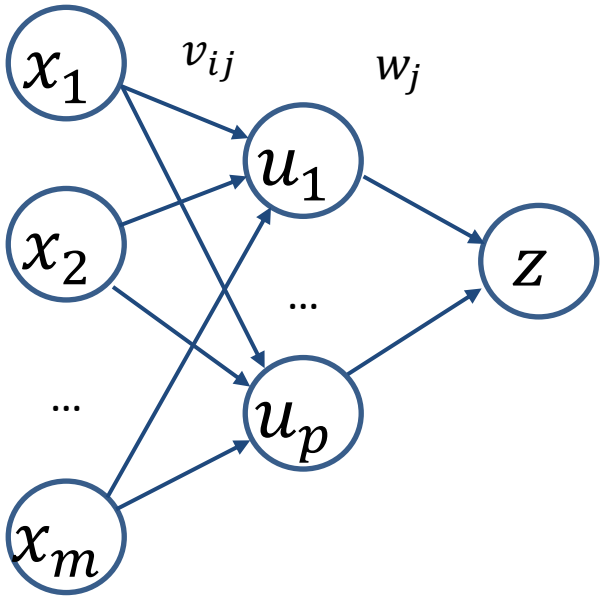
# Backpropagation equations

- We have
  - $\frac{\partial L}{\partial w_j} = \delta \frac{\partial s}{\partial w_j}$
  - $\frac{\partial L}{\partial v_{ij}} = \varepsilon_j \frac{\partial r_j}{\partial v_{ij}}$

… where

- $\delta = \frac{\partial L}{\partial s} = (z - y)$
- $\varepsilon_j = \frac{\partial L}{\partial r_j} = \delta w_j g'(r_j)$

- Recall that
  - $s = \sum_{j=0}^{p} u_j w_j$
  - $r_j = \sum_{i=0}^{m} x_i v_{ij}$



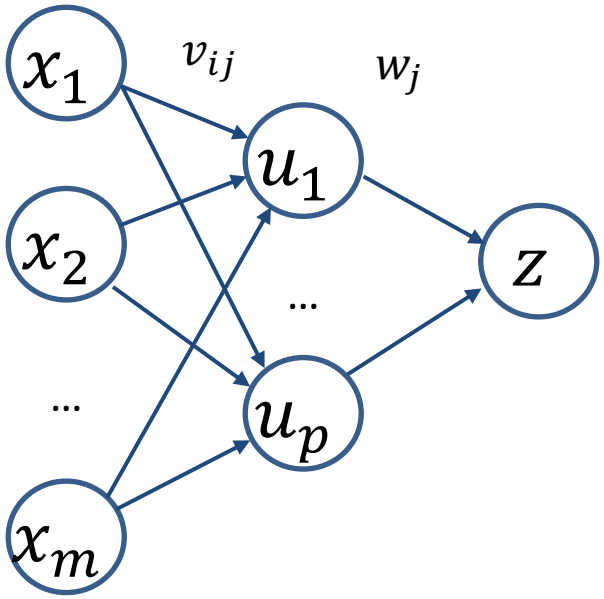

- So $\frac{\partial s}{\partial w_j} = u_j$ and $\frac{\partial r_j}{\partial v_{ij}} = x_i$

- We have
  - $\frac{\partial L}{\partial w_j} = \delta u_j = (z - y) u_j$
  - $\frac{\partial L}{\partial v_{ij}} = \varepsilon_j x_i = \delta w_j g'(r_j) x_i$

# Forward propagation

- Use current estimates of $v_{ij}$ and $w_j$

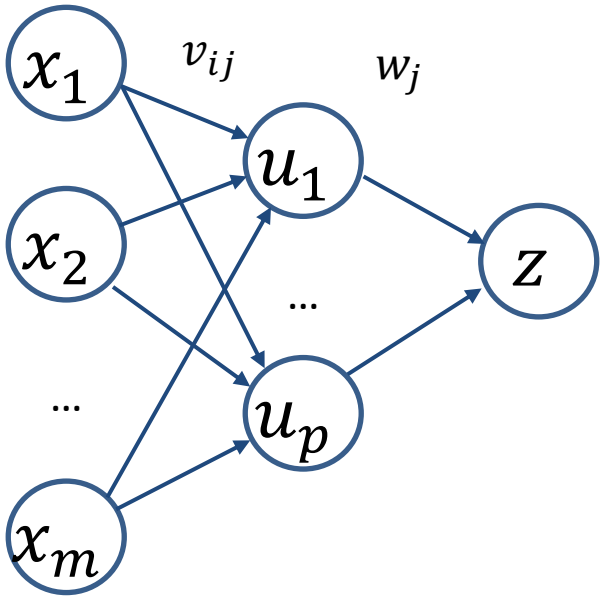
- Calculate $r_j$, $u_j$, $s$ and $z$




- Backpropagation equations
  - $\frac{\partial L}{\partial w_j} = \delta u_j = (z - y)u_j$
  - $\frac{\partial L}{\partial v_{ij}} = \varepsilon_j x_i = \delta w_j g'(r_j) x_i$

# Backward propagation of errors

$$\frac{\partial L}{\partial v_{ij}} = \varepsilon_j x_i \quad \Leftarrow \quad \varepsilon_j = \delta w_j g'(r_j) \quad \Leftarrow \quad \frac{\partial L}{\partial w_j} = \delta u_j \quad \Leftarrow \quad \delta = (z - y)$$




- Backpropagation equations
  - $\frac{\partial L}{\partial w_j} = \delta u_j = (z - y) u_j$
  - $\frac{\partial L}{\partial v_{ij}} = \varepsilon_j x_i = \delta w_j g'(r_j) x_i$

# Interim Summary

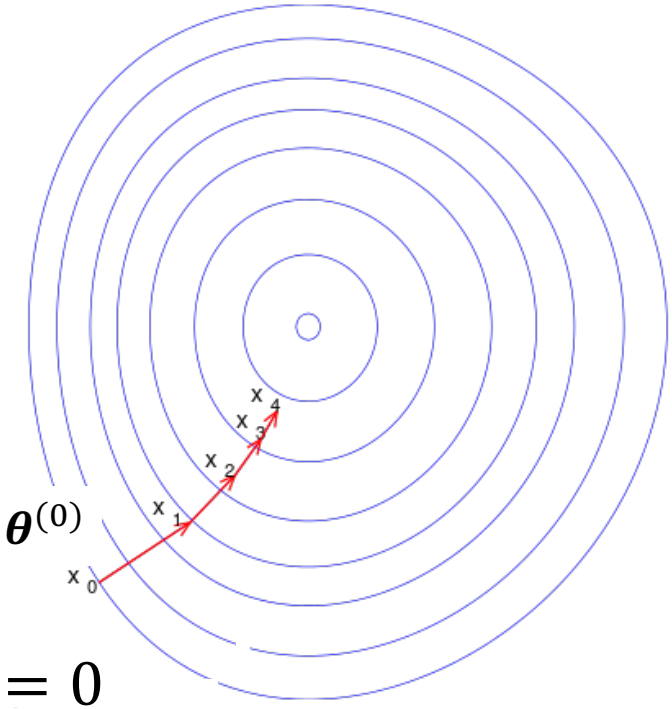
- Motivation for “deep” networks
- Backpropagation of loss
  - derivative chain rule
  - algorithm to compute gradients in backwards pass over network graph

# Training deep networks

Techniques specific to non-convex objective, largely based on gradient descent.

# Recap: Gradient descent vs SGD

1. Choose $\boldsymbol{\theta}^{(0)}$ and some $T$
2. For $i$ from 0 to $T-1$
   1. $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \nabla L(\boldsymbol{\theta}^{(i)})$
3. Return $\widehat{\boldsymbol{\theta}} \approx \boldsymbol{\theta}^{(T)}$



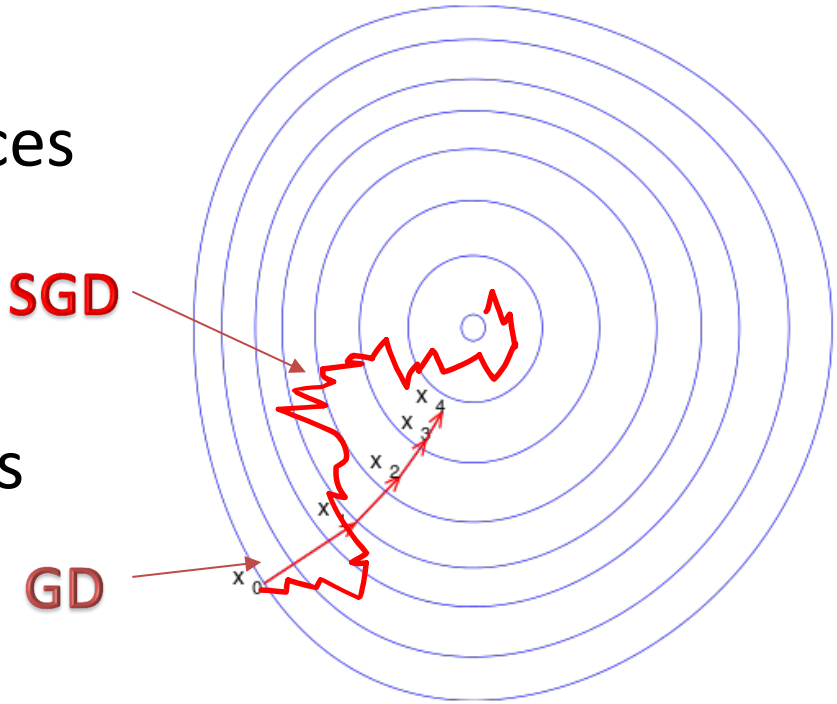

## Stochastic G.D.

1. Choose $\boldsymbol{\theta}^{(0)}$ and some $T, k = 0$
2. For $i$ from 1 to $T$
   1. For j from 1 to N (in random order)
      1. $\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla L(y_j, \boldsymbol{x}_j; \boldsymbol{\theta}^{(k)})$
      2. $k$++
3. Return $\widehat{\boldsymbol{\theta}} \approx \boldsymbol{\theta}^{(k)}$

Wikimedia Commons. Authors: Olegalexandrov, Zerodamage

# Mini-batch SGD

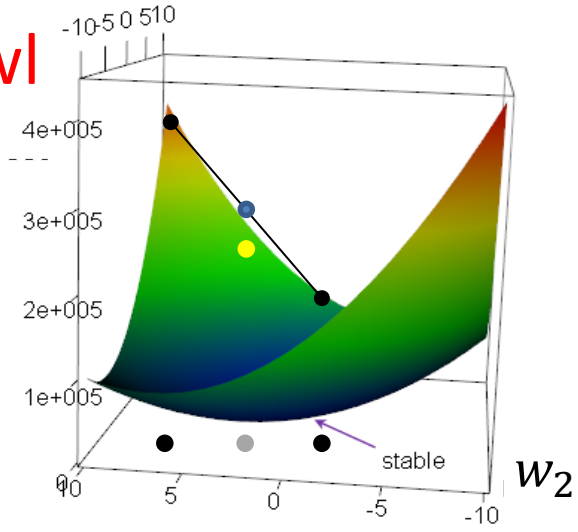
- SGD works on single instances
  - high variance in gradients
  - many, quick, updates
- GD works on whole datasets
  - stable update, but slow
  - computationally expensive



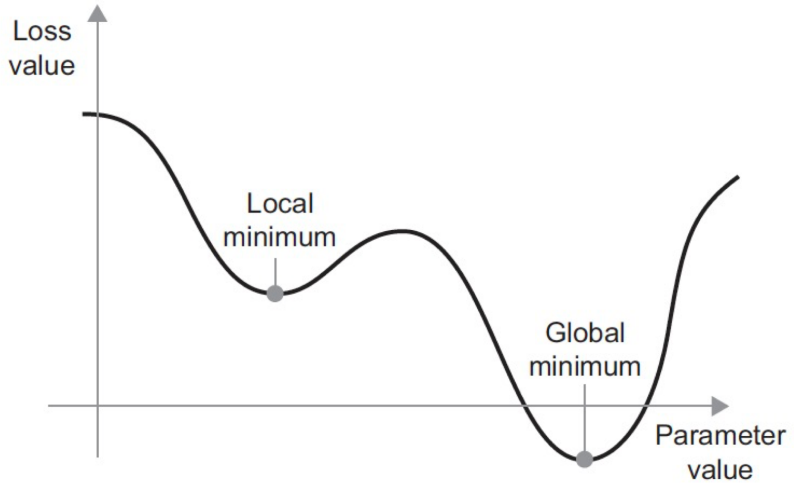

- Compromise: mini-batch *(often just called "SGD")*
  - process batches of size $1 < b < N$, e.g., $b = 100$
  - balances computation and stability

# (non-)Convex objective functions

- Recall logistic regression, convex 'Bowl shaped' objective
  - gradient descent finds a **global** optimum

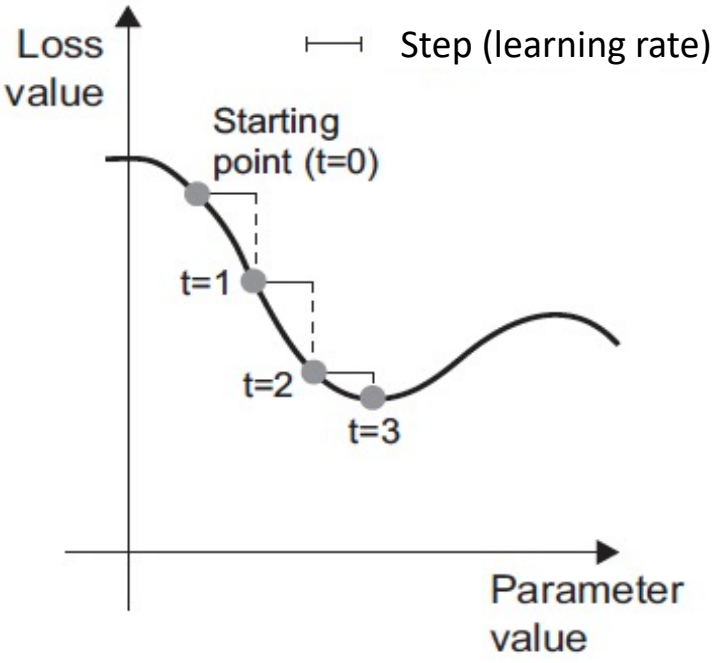



- In contrast, most ANN objective are not convex
  - gradient methods get trapped in **local** optima or **saddle points**



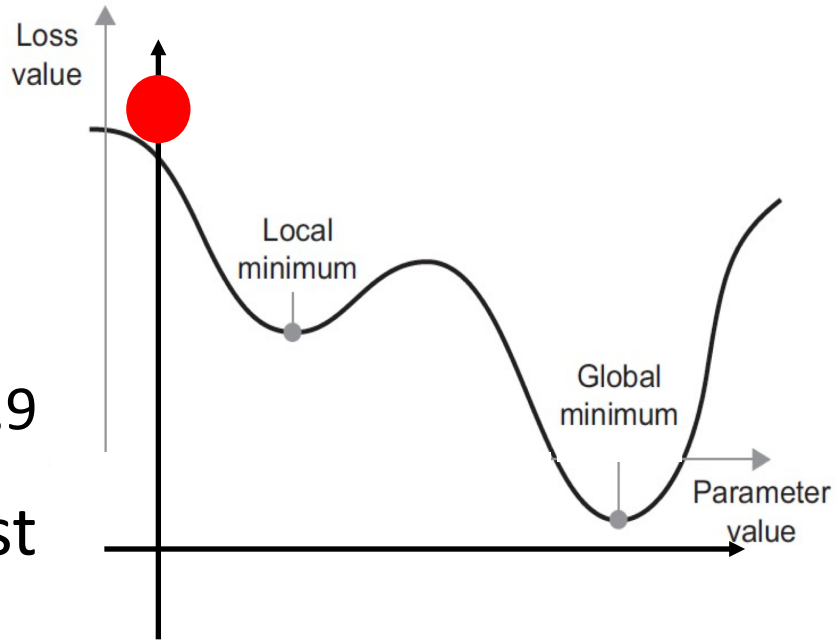
# Importance of learning rate

- Choice of $\eta$ has big effect on quality of final parameters
- Each SGD step:
  - $\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} - \eta \nabla L(\boldsymbol{\theta}^{(i-1)})$
- Choosing $\eta$:
  - Large $\eta$ fluctuate around optima, even diverge
  - Small $\eta$ barely moves, stuck at local optima



# Momentum as a solution

- Consider a ball with some mass rolling down the objective surface
  - velocity increases as it rolls downwards
  - momentum can carry it past local optima
- Mathematically, SGD update becomes
  - $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \boldsymbol{v}^{(t)}$
  - $\boldsymbol{v}^{(t)} = \alpha \boldsymbol{v}^{(t-1)} + \eta \Delta L(\boldsymbol{\theta}^{(t)})$
  - $\alpha$ decays the velocity, e.g., 0.9
- Less oscillation, more robust



# Adagrad: Adaptive learning rates

- Why one learning rate applied to all params?
  - some features (parameters) are used more frequently than others → smaller updates for common features cf rare
- Adagrad tracks the sum of squared gradient per-parameter, i.e., for parameter $i$
  - $g_i^{(t)} = g_i^{(t-1)} + \Delta L\left(\boldsymbol{\theta}^{(t)}\right)_i^2$
  - $\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t)} + \epsilon}} \Delta L\left(\boldsymbol{\theta}^{(t)}\right)_i$

Typically
$\epsilon = 10^{-8}$
$\eta = 0.01$

- Removes need to tune learning rate; but can be too conservative (learning rates shrink quickly!)

# Adam

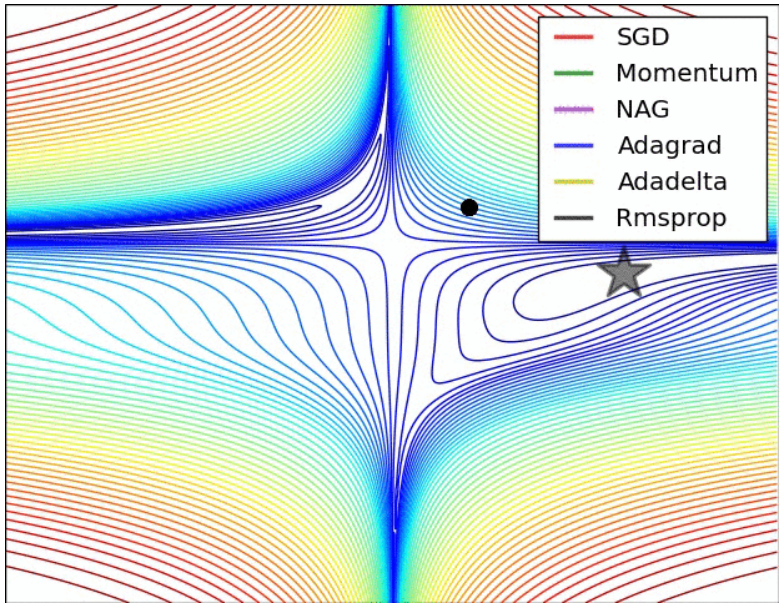
- Combining elements of momentum and adaptive learning rates
  - $\boldsymbol{m}^{(t)} = \beta_1 \boldsymbol{m}^{(t-1)} + (1-\beta_1)\Delta L\left(\boldsymbol{\theta}^{(t)}\right)$
  - $\boldsymbol{v}^{(t)} = \beta_2 \boldsymbol{v}^{(t-1)} + (1-\beta_2)\Delta L\left(\boldsymbol{\theta}^{(t)}\right)^2$
  - $\theta^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{\sqrt{\boldsymbol{v}^{(t)}/_{1-\beta_2}} + \epsilon} \boldsymbol{m}^{(t)}/_{1-\beta_1}$
  - $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

element-wise operations

- Good work-horse method, current technique of choice for deep learning

# Zoo of optimisation algorithms

- Suite of batch-style algorithms, e.g., BFGS, L-BFGS, Conjugate Gradient, …
- And SGD style:
  - Nesterov acc. grad.
  - Adadelta
  - AdaMax
  - RMSprop
  - AMSGrad
  - Nadam
  - AdamW
  - QHAdam…




- Lots of choice, and rapidly changing as deep learning matures

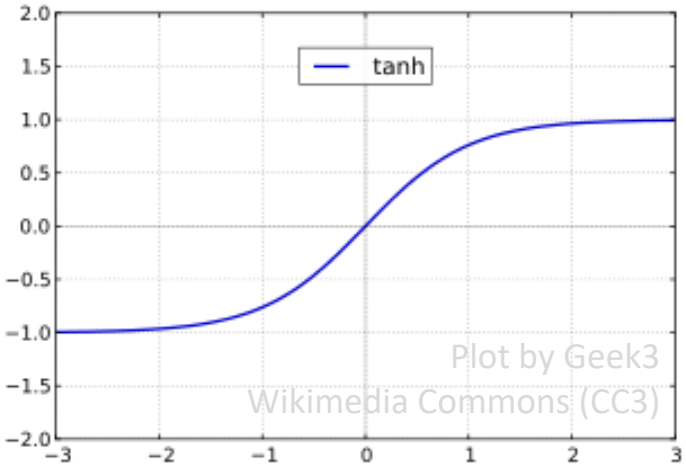
https://ruder.io/optimizing-gradient-descent/index.html

# Regularising deep networks

Best practices in preventing overfitting, a big problem for such high capacity and complex models.

# Some further notes on ANN training

- ANN's are flexible (recall universal approximation theorem), but the flipside is over-parameterisation, hence tendency to overfitting
- Starting weights usually random distributed about zero
- Implicit regularisation: early stopping
  - With some activation functions, this shrinks the ANN towards a linear model (why?)

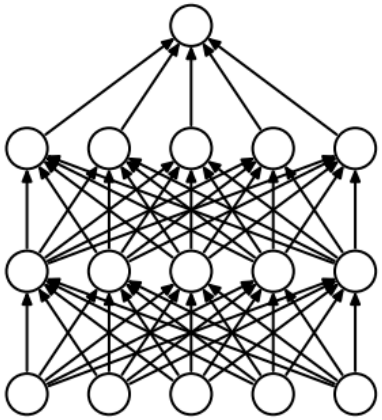


# Explicit regularisation

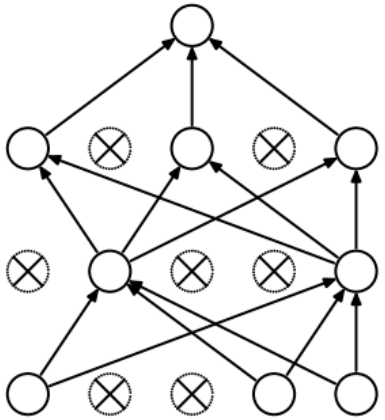
- Alternatively, an explicit regularisation can be used, much like in ridge regression
- Instead of minimising the loss $L$, minimise regularised function $L + \lambda\left(\sum_{i=0}^{m}\sum_{j=1}^{p} v_{ij}^2 + \sum_{j=0}^{p} w_j^2\right)$
- This will simply add $2\lambda v_{ij}$ and $2\lambda w_j$ terms to the partial derivatives
- With some activation functions this also shrinks the ANN towards a linear model

# Dropout

- Randomly mask fraction of units during training
  - different masking each presentation
  - promotes redundancy in network hidden representation
  - a form of ensemble of exponential space
  - no masking at testing (requires weight adjustment)



(a) Standard Neural Net



(b) After applying dropout.

- Results in smaller weights, and less overfitting
- Used in most SOTA deep learning systems

# This lecture

- Deep learning
- Backpropagation
- Training deep networks
  - Optimisation methods
  - Regularisation techniques
- Next lecture: CNNs