Lecture 12. Training Deep Networks & Autoencoders

COMP90051 Statistical Machine Learning

Lecturer: Zahra Dasht Bozorgi

THE UNIVERSITY OF MELBOURNE

This lecture

- Training DNNs
  - SGD
  - Regularisation
- Autoencoders
  - Learning efficient coding
  - Use in pre-training pipelines

Training DNNs

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

How to train your ~~dragon~~ network?

- You know the drill: Define the loss function and find parameters that minimise the loss on training data

Adapted from Movie Poster from Flickr user jdxyw (CC BY-SA 2.0)

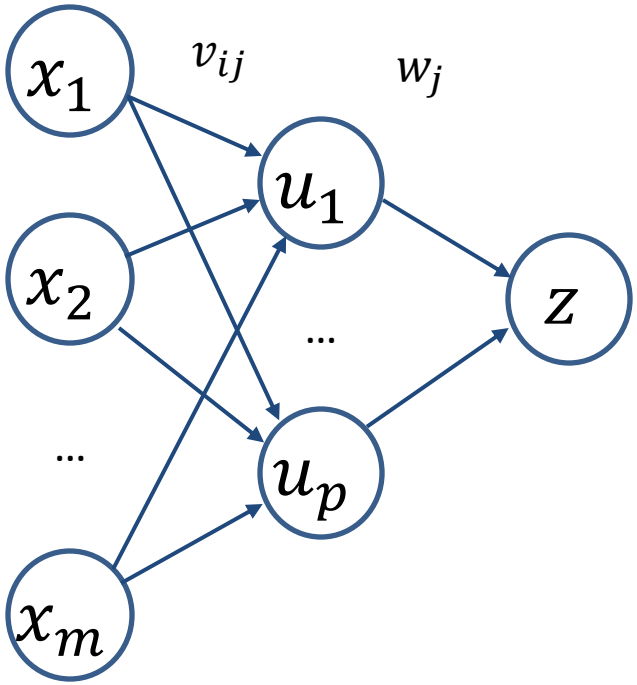
- In the following, we are going to use stochastic gradient descent with a batch size of one. That is, we will process training examples one by one

Example: univariate regression

- Consider regression
- Moreover, we'll use identity output activation function

$$z = h(s) = s = \sum_{j=0}^{p} u_j w_j$$

- This will simplify description of backpropagation. In other settings, the training procedure is similar



Loss function for NNet training

- Need loss between training example $\{\boldsymbol{x}, y\}$ & prediction $\hat{f}(\boldsymbol{x}, \boldsymbol{\theta}) = z$, where $\boldsymbol{\theta}$ is parameter vector of v_{ij} and w_j
- As regression, can use squared error

$$L = \frac{1}{2}\left(\hat{f}(\boldsymbol{x}, \boldsymbol{\theta}) - y\right)^2 = \frac{1}{2}(z - y)^2$$

(the constant is used for mathematical convenience, see later)

- Decision-theoretic training: minimise L w.r.t $\boldsymbol{\theta}$
  - Fortunately $L(\boldsymbol{\theta})$ is differentiable
  - Unfortunately no analytic solution in general

Stochastic gradient descent for NNet

Choose initial guess $\boldsymbol{\theta}^{(0)}$, $k = 0$

Here $\boldsymbol{\theta}$ is a set of all weights form all layers

For i from 1 to T (epochs)

For j from 1 to N (training examples – could shuffle)

Consider example $\{\boldsymbol{x}_j, y_j\}$

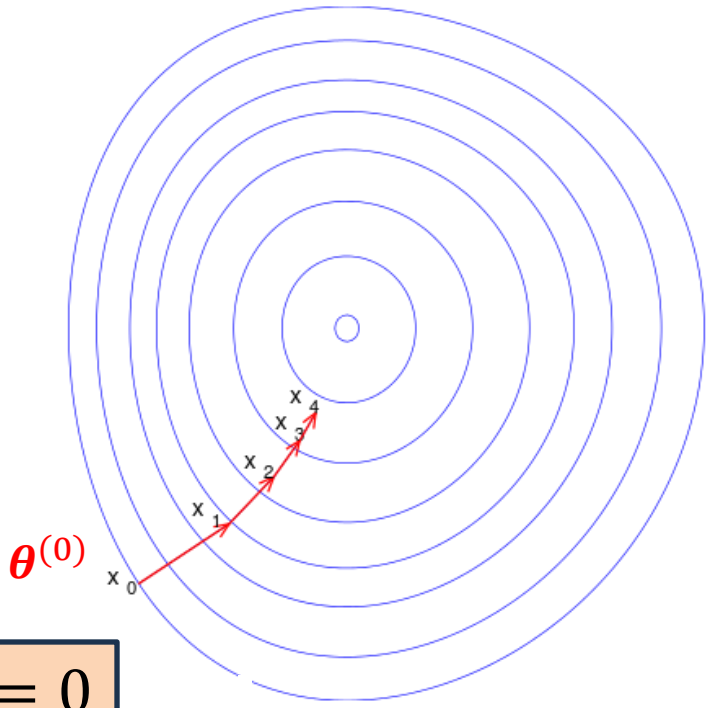
Update: $\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla L(\boldsymbol{\theta}^{(k)})$; $k \leftarrow k+1$

$$L = \frac{1}{2}(z_j - y_j)^2$$

Need to compute partial derivatives $\frac{\partial L}{\partial v_{ij}}$ and $\frac{\partial L}{\partial w_j}$

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 $\widehat{\boldsymbol{\theta}} \approx \boldsymbol{\theta}^{(T)}$



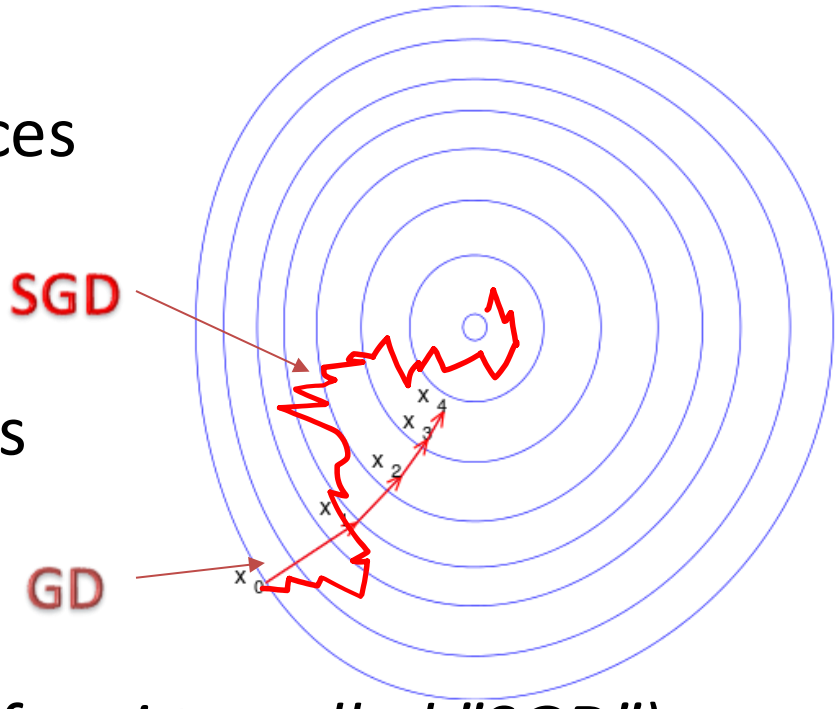
Stochastic G.D.

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

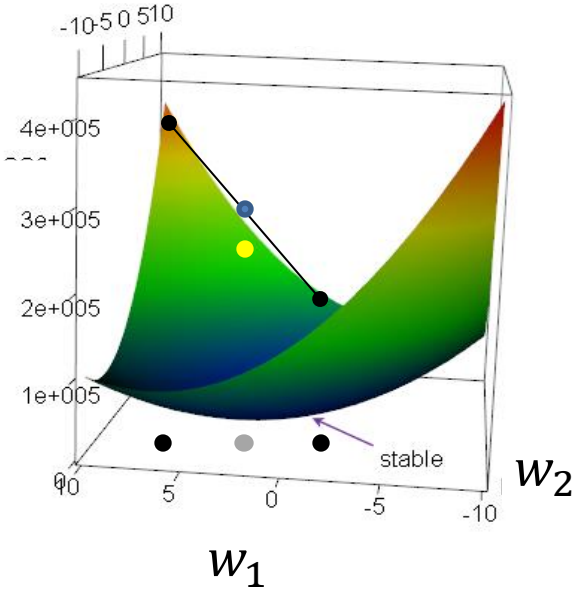
- SGD works on single instances
  - high variance in gradients
  - many, quick, updates
- GD works on whole datasets
  - stable update, but slow
  - computationally expensive



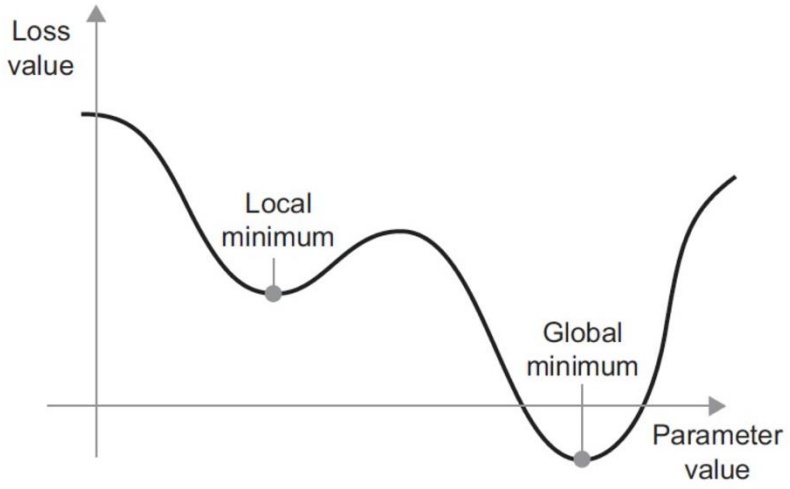
- Compromise: mini-batch *(often just called "SGD")*
  - process batches of size $1 < b < N$, e.g., $b = 100$
  - balances computation and stability
  - parallelise over cluster of GPUs (size batch for GPU)

(non-)Convex objective functions

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

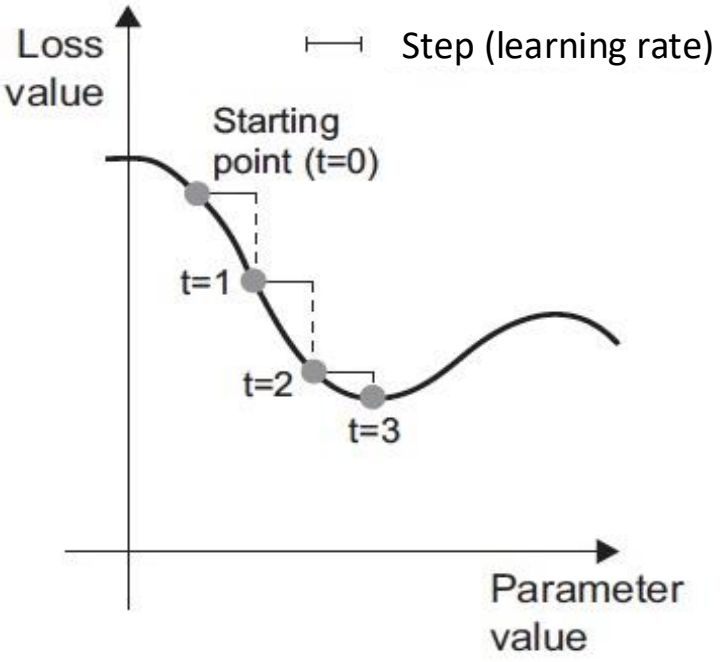


- In contrast, most DNN objectives are not convex
  - gradient methods get trapped in local optima or saddle points



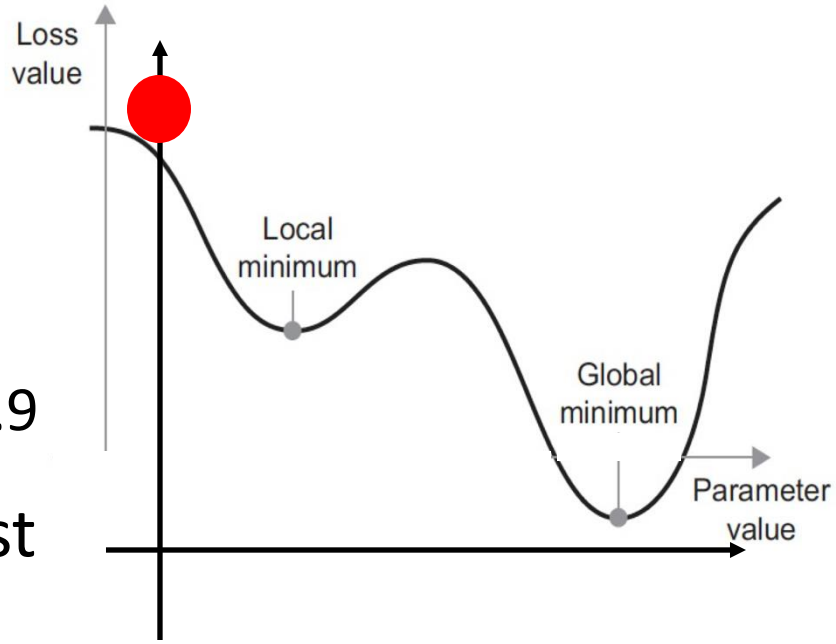
Importance of learning rate

- Choice of η 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 η:
  - Large η fluctuate around optima, even diverge
  - Small η 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
  - $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \boldsymbol{v}^{(t)}$
  - $\boldsymbol{v}^{(t)} = \alpha \boldsymbol{v}^{(t-1)} + \eta \nabla L(\boldsymbol{\theta}^{(t)})$
  - α decays the velocity, e.g., 0.9
- Less oscillation, more robust



Adagrad: Adaptive learning rates

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

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

- No need to tune learning rate! But can be conservative

Adam

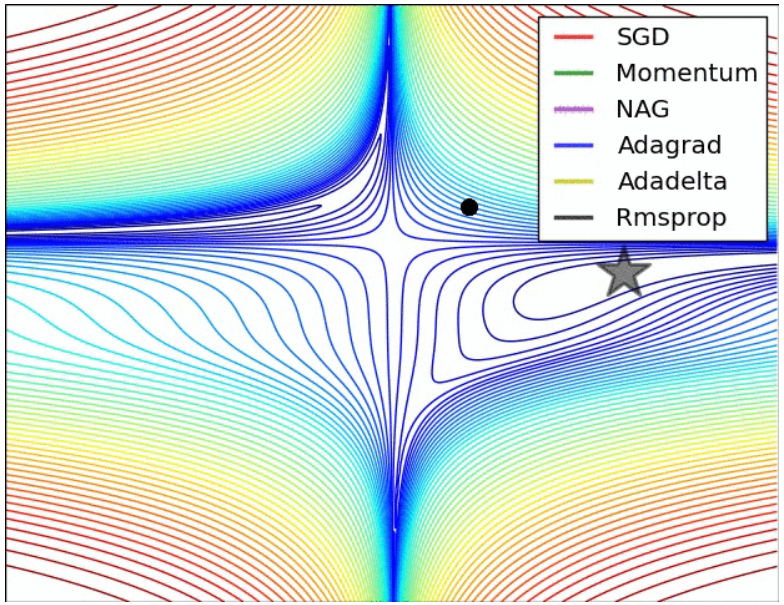
- Combining elements of momentum and adaptive learning rates
  - $\boldsymbol{m}^{(t)} = \beta_1 \boldsymbol{m}^{(t-1)} + (1-\beta_1)\nabla L\left(\boldsymbol{\theta}^{(t)}\right)$
  - $\boldsymbol{v}^{(t)} = \beta_2 \boldsymbol{v}^{(t-1)} + (1-\beta_2)\nabla L\left(\boldsymbol{\theta}^{(t)}\right)^2$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

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
  - Adam
  - …



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

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

Mini summary

- Training DNNs
  - SGD
  - Mini batch SGD
  - Momentum, Adagrad, Adam

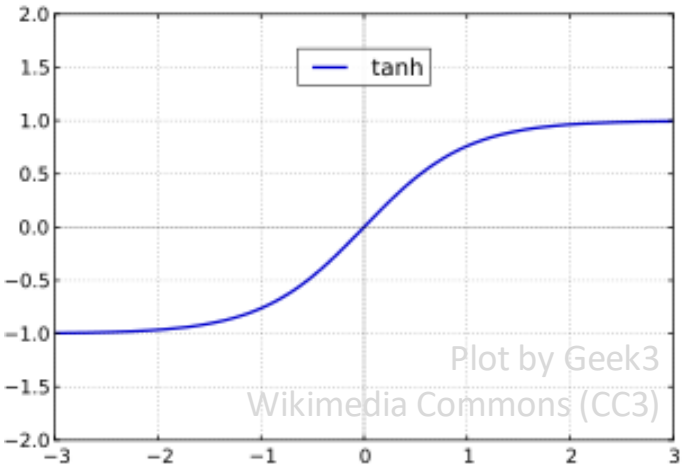
Next: Regularising DNNs

Regularising Deep Nets

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

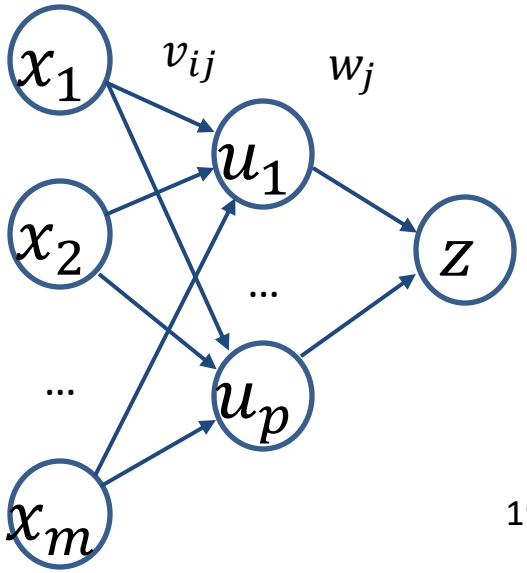
Some further notes on DNN training

- DNNs 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 DNN towards a linear model (why?)



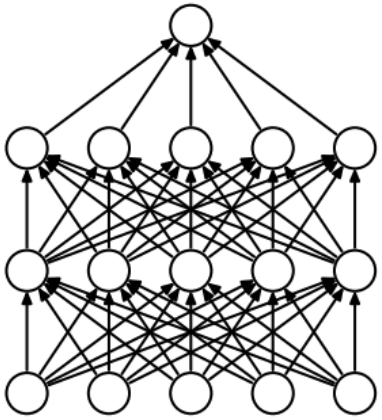
Explicit regularisation

- 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 (aka weight decay)
- With some activation functions (e.g. tanh / sigmoid) this also shrinks the DNN towards a linear model

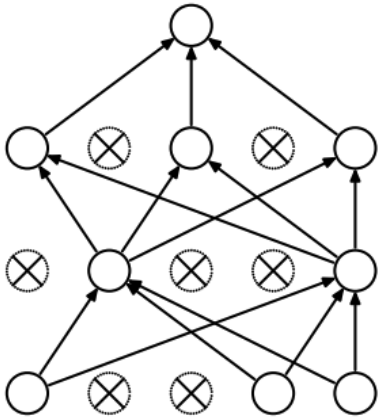


Dropout

- Randomly mask fraction of units during training
  - different masking each presentation
  - promotes redundancy in network hidden representation (a form of regularization)
  - 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

Mini summary

- Regularised training of (overparameterised) DNNs
  - Early stopping
  - Explicit L_2 regularisation / weight decay / shrinkage
  - Dropout

Next: Autoencoders

Autoencoders

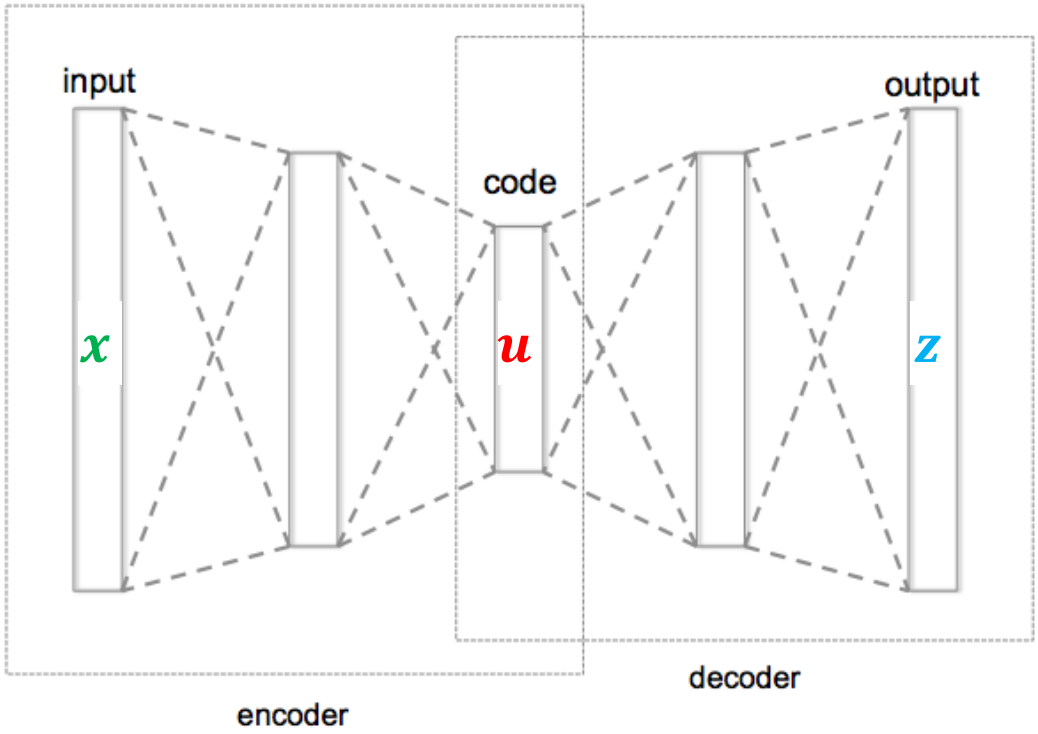
A DNN training setup that can be used for unsupervised learning, initialisation, or just efficient coding

Autoencoding idea

- Supervised learning:
  - Univariate regression: predict y from $\boldsymbol{x}$
  - Multivariate regression: predict $\boldsymbol{y}$ from $\boldsymbol{x}$
- Unsupervised learning: explore data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$
  - No response variable
- For each $\boldsymbol{x}_i$ set $\boldsymbol{y}_i \equiv \boldsymbol{x}_i$
- Train a NNet to predict $\boldsymbol{y}_i$ from $\boldsymbol{x}_i$ i.e., model $\mathrm{p}(\boldsymbol{x}|\boldsymbol{x})$
- Pointless?

Autoencoder topology

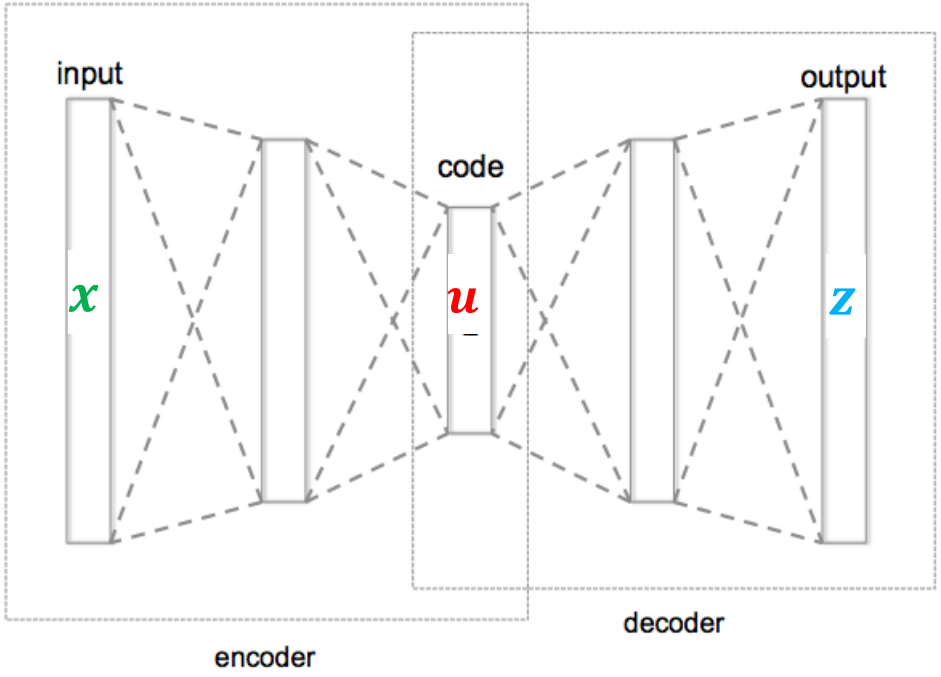
- Given data without labels $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, set $\boldsymbol{y}_i \equiv \boldsymbol{x}_i$ and train a DNN to predict $\boldsymbol{z}(\boldsymbol{x}_i) \approx \boldsymbol{x}_i$
- Set bottleneck layer $\boldsymbol{u}$ in middle "thinner" than input, and/or
  - corrupt input $\boldsymbol{x}$ with noise
  - regularise s.t. $\boldsymbol{u}$ is sparse
  - regularise to contract inputs



adapted from: Chervinskii at Wikimedia Commons (CC4)

Introducing the bottleneck

- Suppose you managed to train a network that gives a good restoration of the original signal $\boldsymbol{z}(\boldsymbol{x}_i) \approx \boldsymbol{x}_i$
- This means that the data structure can be effectively described (encoded) by a lower dimensional representation $\boldsymbol{u}$



adapted from: Chervinskii at Wikimedia Commons (CC4)

Under-/Over-completeness

- Manner of bottleneck gives rise to:
  - undercomplete: model with thinner bottleneck than input forced to generalise
  - overcomplete: wider bottleneck than input, can just "copy" input directly to output
- Even undercomplete models can learn trivial codes, given complex non-linear encoder and decoder
- Various methods to ensure learning

Dimensionality reduction

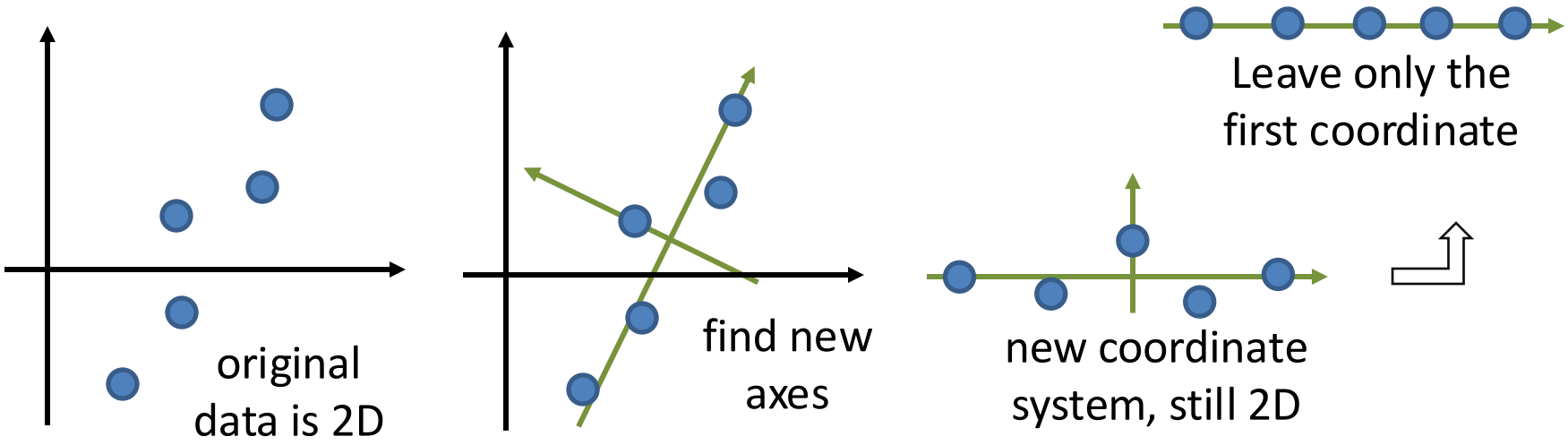
- Autoencoders can be used for
  - compression
  - dimensionality reduction
  - unsupervised pre-training
  - finding latent feature space

  …via a non-linear transformation

- Related to principal component analysis (PCA)…

Principal component analysis

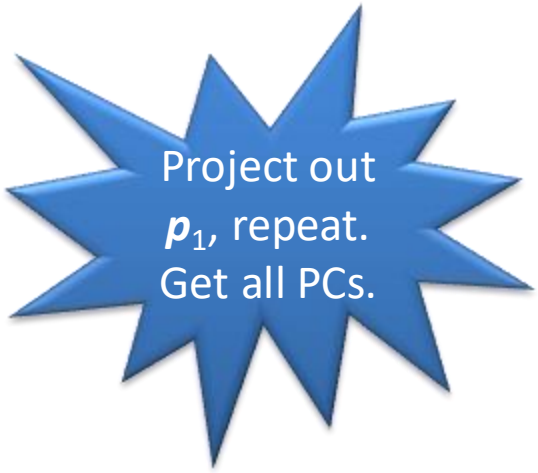
- Principal component analysis (PCA) is a popular method for dimensionality reduction and data analysis in general
- Given a dataset $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, $\boldsymbol{x}_i \in \boldsymbol{R}^m$, PCA aims to find a new coordinate system such that most of the variance is concentrated along the first coordinate, then most of the remaining variance along the second (orthogonal) coordinate, etc.
- Dimensionality reduction is then based on discarding coordinates except the first $l < m$. Coordinates = axes of data = principal components



PCA: Solving the optimisation

- PCA aims to find principal component $\boldsymbol{p}_1$ that maximises variance of data projected onto the PC, $\boldsymbol{p}_1' \boldsymbol{\Sigma}_X \boldsymbol{p}_1$,
  - Subject to $\|\boldsymbol{p}_1\| = \boldsymbol{p}_1' \boldsymbol{p}_1 = 1$
  - Have to first subtract the centre of the data from the data
- Constrained → Lagrange multipliers. Introduce multiplier λ_1; set derivatives of Lagrangian to zero, solve

- $L = \boldsymbol{p}_1' \boldsymbol{\Sigma}_X \boldsymbol{p}_1 - \lambda_1 (\boldsymbol{p}_1' \boldsymbol{p}_1 - 1)$
- $\frac{\partial L}{\partial \boldsymbol{p}_1} = 2\boldsymbol{\Sigma}_X \boldsymbol{p}_1 - 2\lambda_1 \boldsymbol{p}_1 = 0$
- $\boldsymbol{\Sigma}_X \boldsymbol{p}_1 = \lambda_1 \boldsymbol{p}_1$



- Precisely defines $\boldsymbol{p}_1$ as an eigenvector of covariance $\boldsymbol{\Sigma}_X$ with λ_1 being the corresponding eigenvalue

PCA vs Autoencoding

- If you use linear activation functions and only one hidden layer, then the setup becomes almost that of Principal Component Analysis (PCA)
  - PCA finds orthonormal basis where axes are aligned to capture maximum data variation
  - NNet might find a different solution, doesn't use eigenvalues (directly)

Uses of Autoencoders

- Data visualisation & clustering
  - Unsupervised first step towards understanding properties of the data
- As a feature representation
  - Allowing the use of off-the-shelf ML methods, applied to much smaller and informative representations of input
- Pre-training of deep models
  - Warm-starting training by initialising model weights with encoder parameters
  - In some fields like vision, mostly replaced with supervised pre-training on very large datasets

This lecture

- Training DNNs as optimisation
- Regularisation
- Autoencoders

Next: Convolutional neural networks