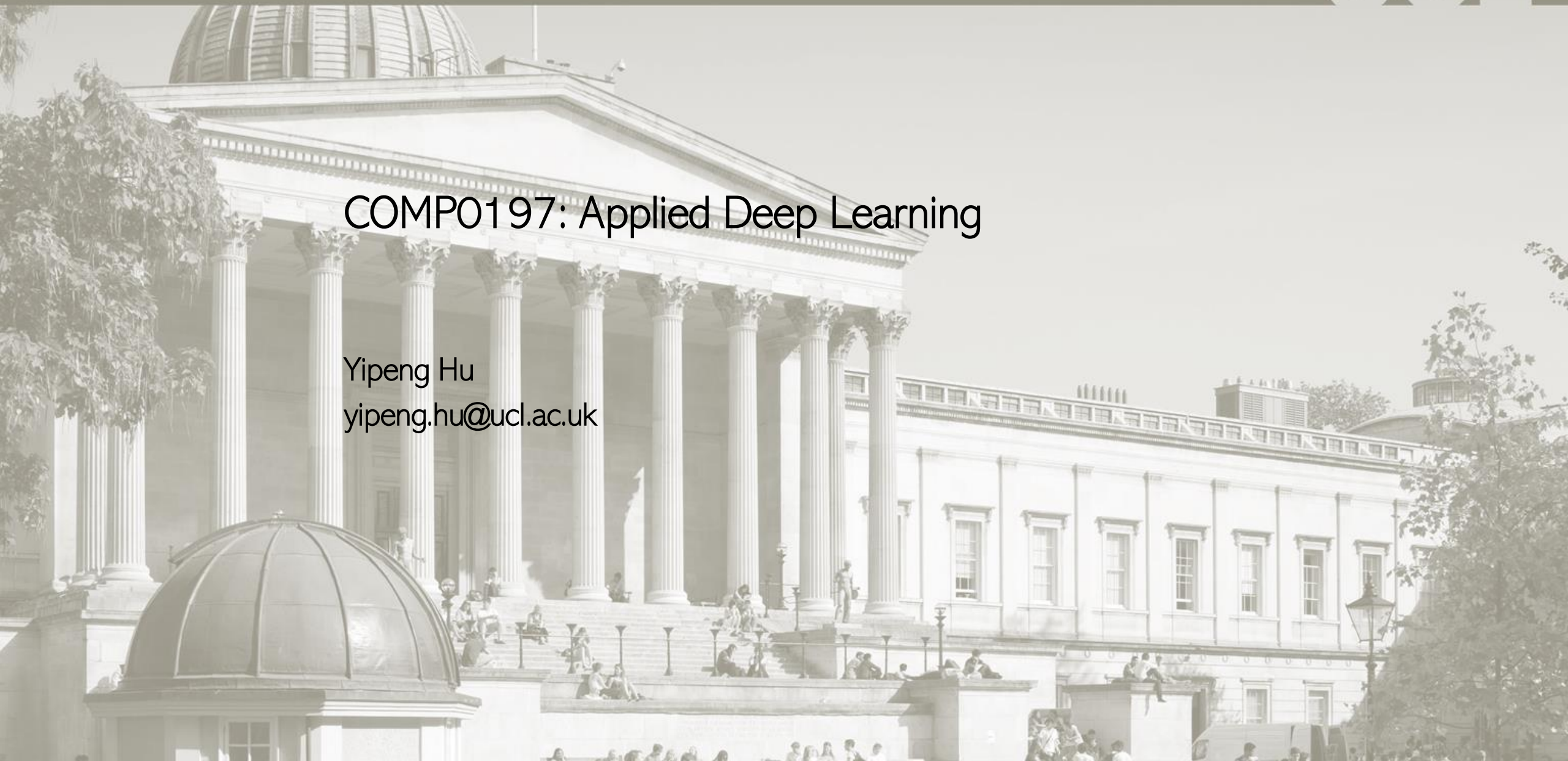
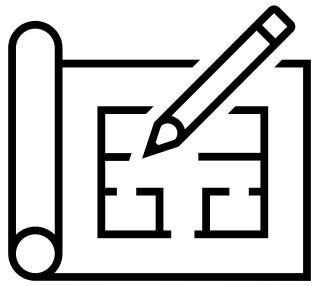


COMP0197: Applied Deep Learning

Yipeng Hu
yipeng.hu@ucl.ac.uk

- Multilayer Perceptron (MLP)
- Anatomy of A Layer
- Activation Functions
- Architecture

  Width and Depth | Branching and Joining | Skipping | Sampling | “Ignoring”
- Architecture Search



- Training
- Loss Functions
- Stochastic Gradient Descent

  Minibatch | Epoch | Automatic Differentiation
- Backpropagation

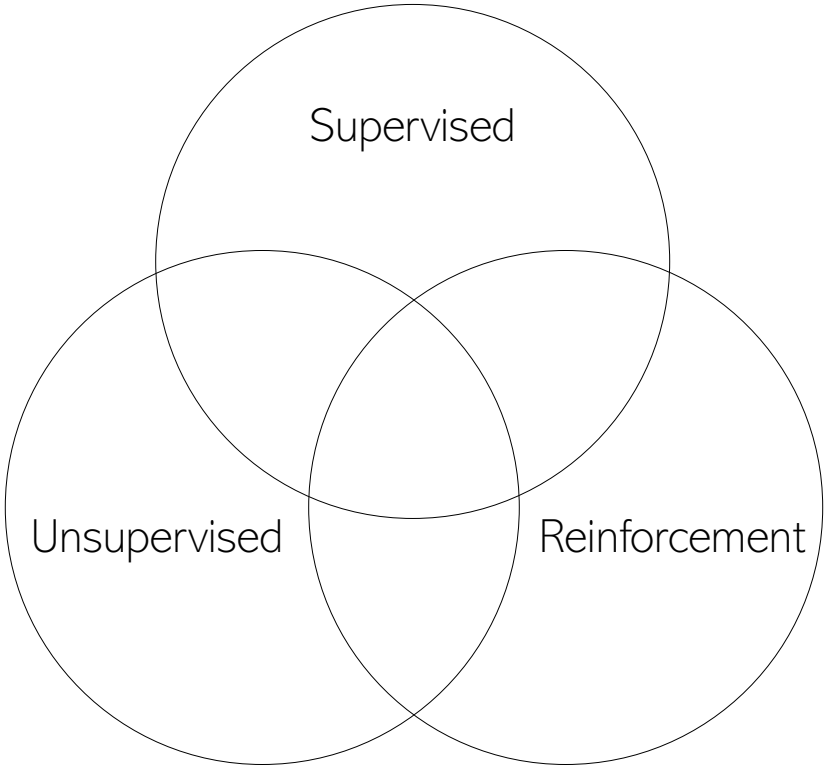
  Computational Graph | Weight Initialisation | Adaptive Learning Rates
- Distributed Computing

Deep Feedforward Neural Networks | Training

Training Objectives

- Supervised learning
- Unsupervised learning
- Reinforcement learning

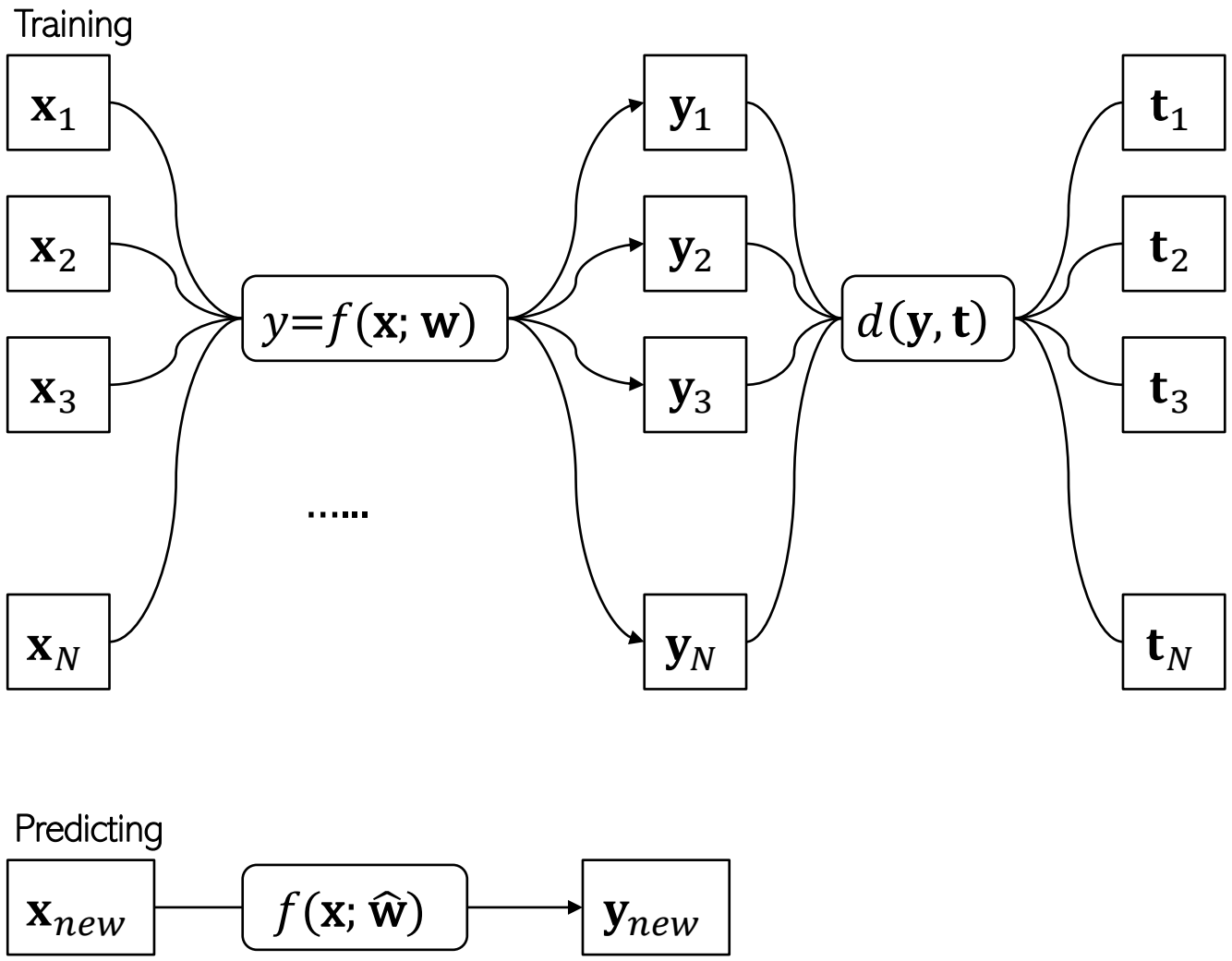
$$y = f(\mathbf{x}; \mathbf{w})$$



Training Objectives

- Optimising parameters/weights
- Testing/predicting/inference
- Generalisation
- Parameter estimation

  e.g. maximum likelihood principle



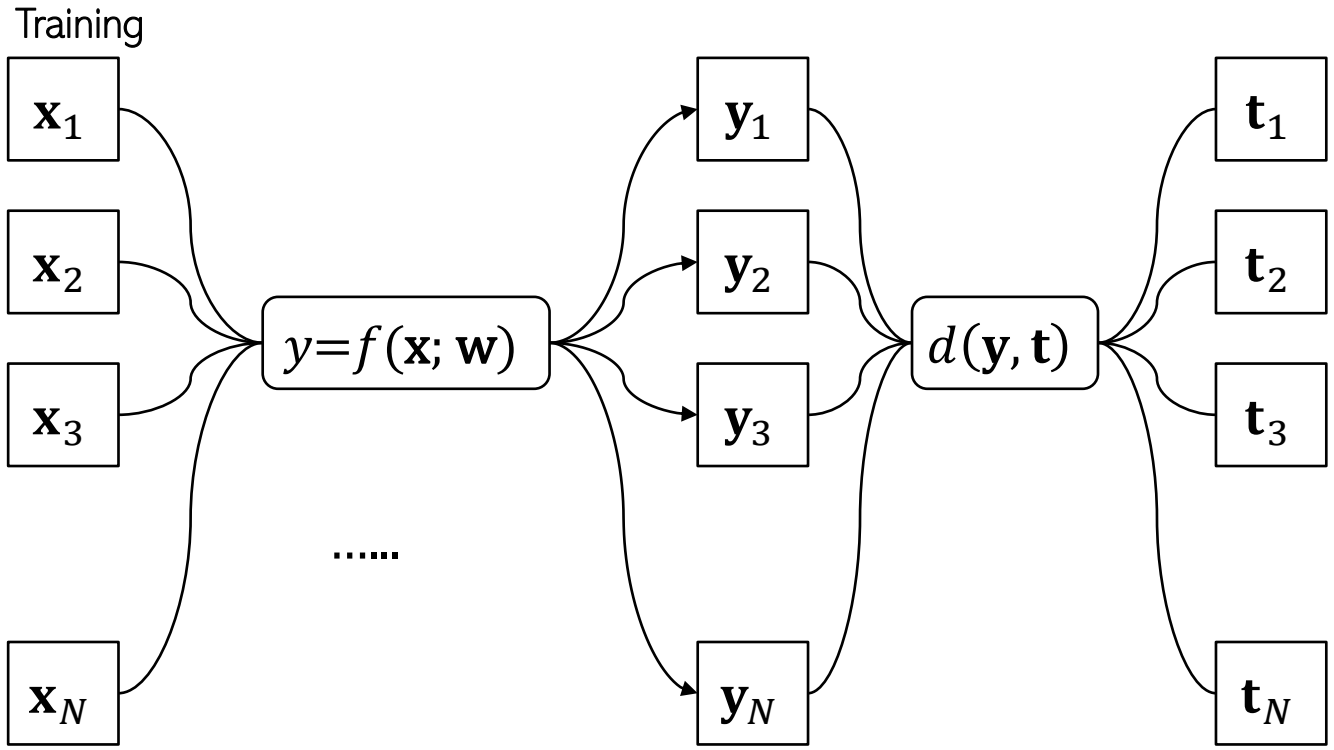
Training Objectives

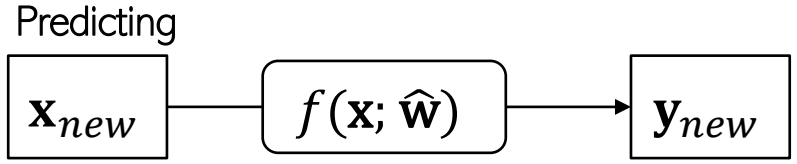
– Supervised learning

$$\ell_{\mathbf{w}} = \frac{1}{N}\sum_{n=1}^{N} d(\mathbf{y}_n, \mathbf{t}_n)$$

$$= \frac{1}{N}\sum_{n=1}^{N} d(f(\mathbf{x}_n;\mathbf{w}), \mathbf{t}_n)$$

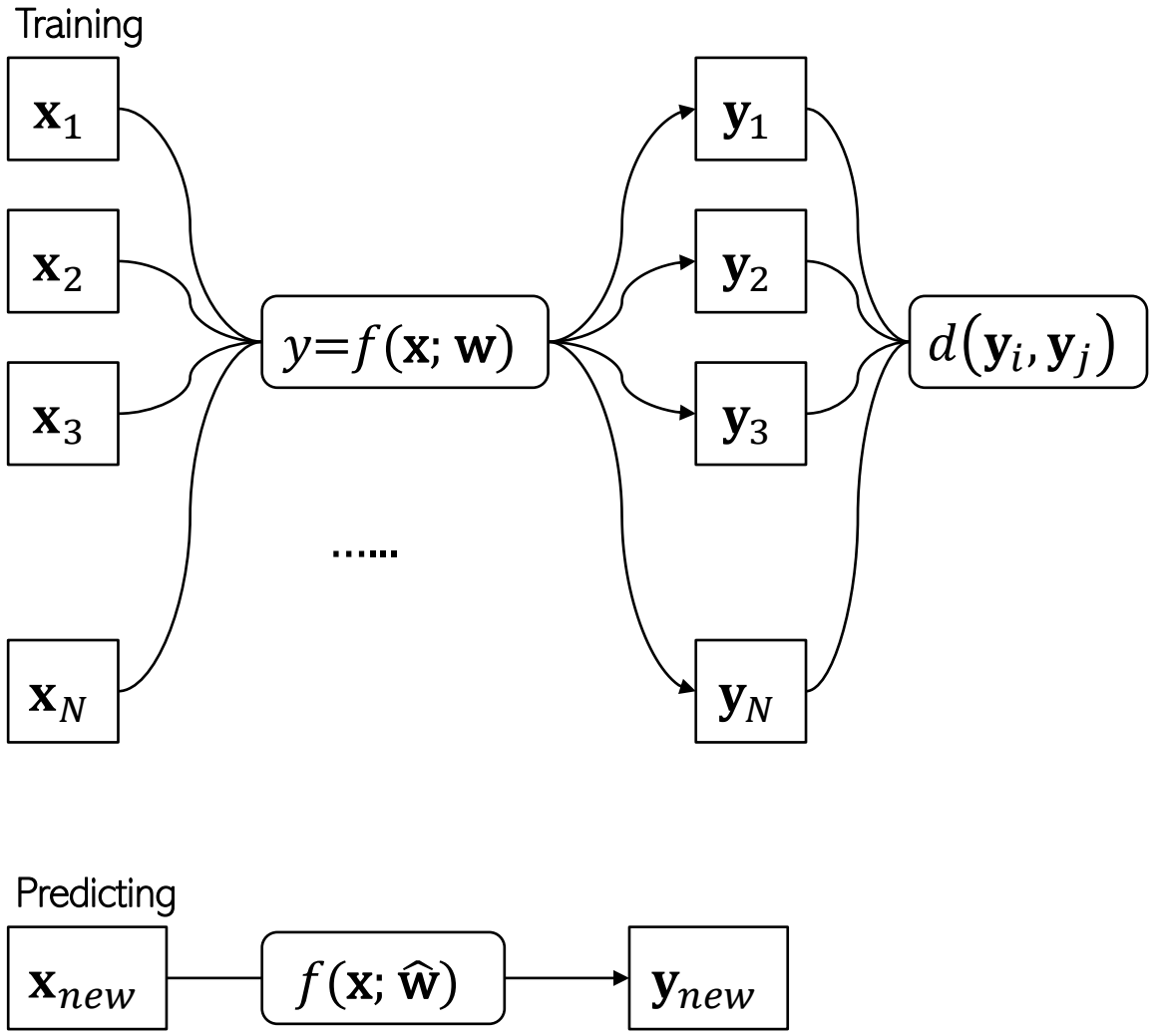
Why average ?





Training Objectives

- Unsupervised learning
- Reinforcement learning



Deep Feedforward Neural Networks | Loss Functions

Supervised regression loss

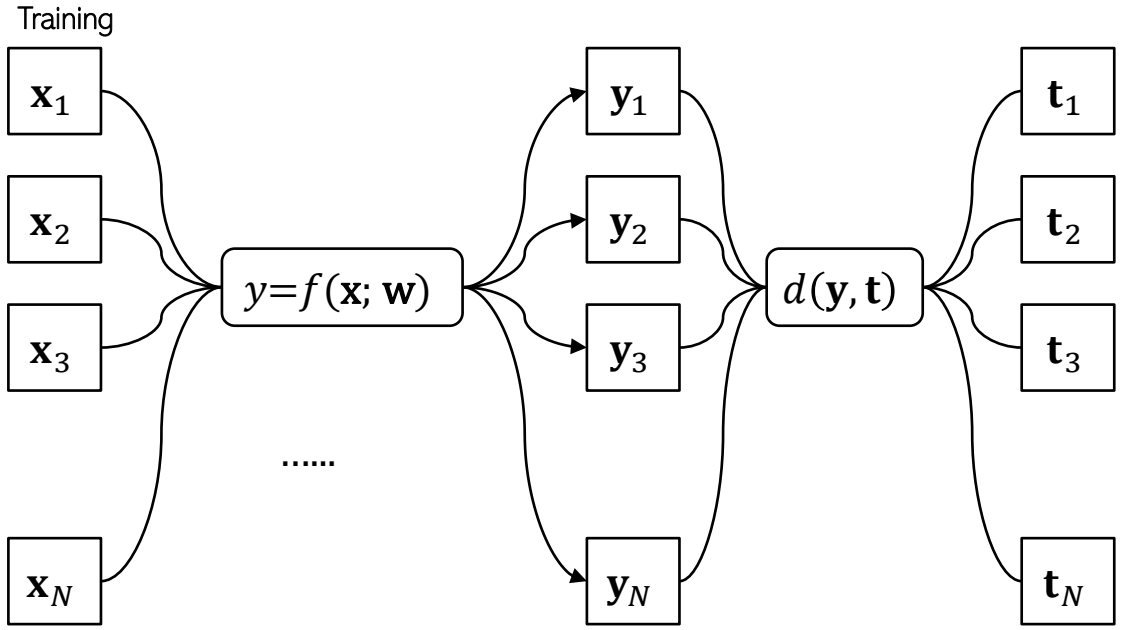
$$\ell_{\mathbf{w}} = \frac{1}{N}\sum_{n=1}^{N} d(\mathbf{y}_n, \mathbf{t}_n) = \frac{1}{N}\sum_{n=1}^{N} d(f(\mathbf{x}_n;\mathbf{w}), \mathbf{t}_n)$$

- Mean-squared-error (squared L^2-norm)

$$\ell_{\mathbf{w}} = \frac{1}{N}\sum_{n=1}^{N} (y_n - t_n)^2$$

- L^1-norm

$$\ell_{\mathbf{w}} = \frac{1}{N}\sum_{n=1}^{N} |y_n - t_n|$$



Supervised classification loss

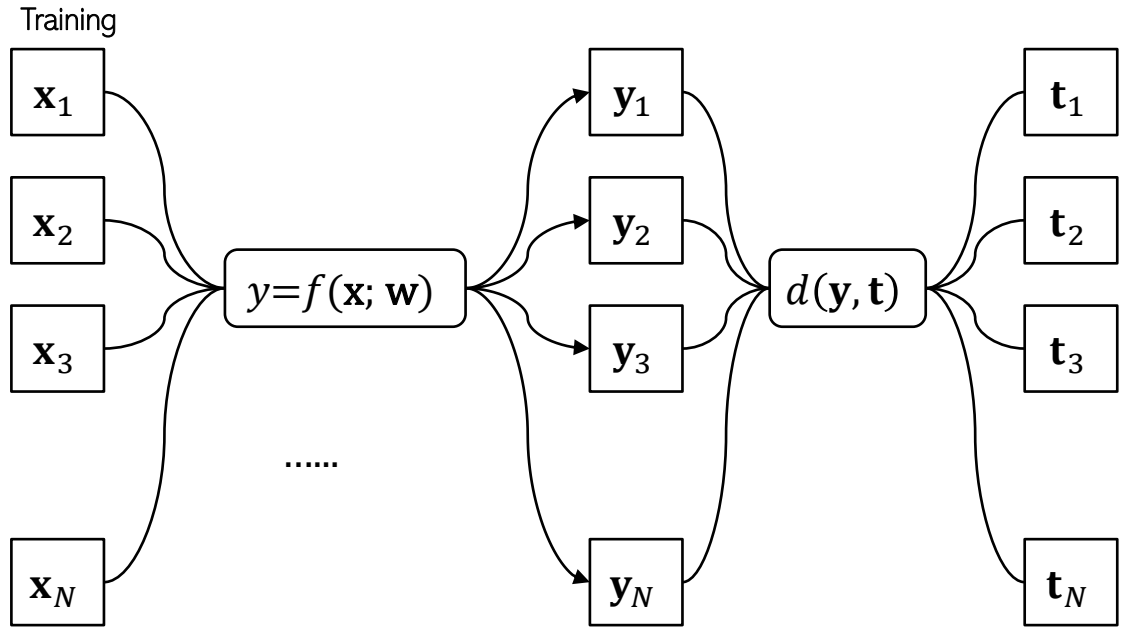
$$\ell_{\mathbf{w}} = \frac{1}{N}\sum_{n=1}^{N} d(\mathbf{y}_n, \mathbf{t}_n) = \frac{1}{N}\sum_{n=1}^{N} d(f(\mathbf{x}_n;\mathbf{w}), \mathbf{t}_n)$$

- Binary cross-entropy

$$\ell_{\mathbf{w}} = -\frac{1}{N}\sum_{n=1}^{N} [t_n log(y_n) + (1-t_n) log(1-y_n)]$$

- Multi-class cross-entropy

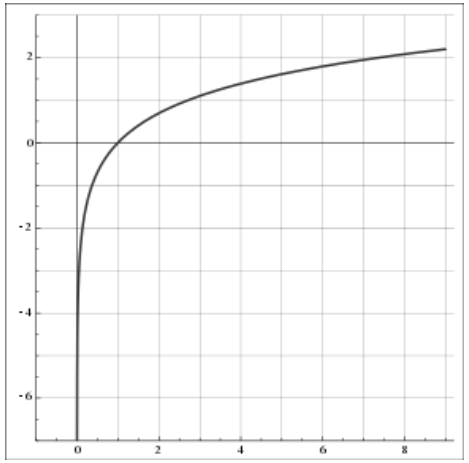
$$\ell_{\mathbf{w}} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} [t_{k,n} log(y_{k,n})]$$

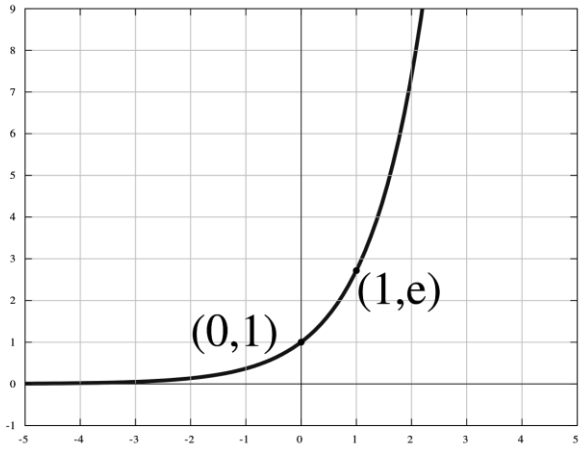


Cross-entropy loss with logits

- Binary cross-entropy

$$\ell_{\mathbf{w}} = -\frac{1}{N}\sum_{n=1}^{N}[t_n ln(y_n) + (1-t_n)ln(1-y_n)]$$





- Logistic sigmoid activation

$y = g(z) = \frac{1}{1+e^{-z}}$

Numerical stability*

e.g.: $t \cdot ln(y) + (1-t)ln(1-y) = t \cdot ln(1+e^{-z}) + (1-t)\cdot[z + ln(1+e^{-z})] =$

$$\begin{cases} ln(1+e^{-z}) + z - z\cdot t, & if\ z \geq 0 \\ ln(1+e^{-z}) + z - z\cdot t, & if\ z < 0 \end{cases} = \begin{cases} ln(1+e^{-z}) + z - z\cdot t, & if\ z \geq 0 \\ ln(1+e^{-z}) + ln(e^{z}) - z\cdot t, & if\ z < 0 \end{cases} =$$

$$\begin{cases} ln(1+e^{-z}) + z - z\cdot t, & if\ z \geq 0 \\ \quad ln(1+e^{z}) - z\cdot t, & if\ z < 0 \end{cases}$$

Piecewise functions

- Label smoothing

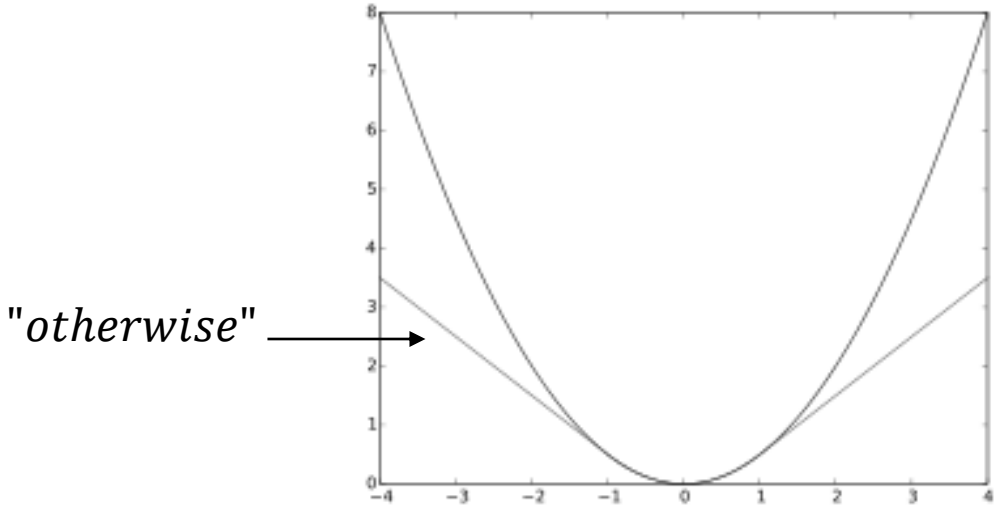
$$t_n^*(k) = \begin{cases} k, if\ t_n < k \\ 1-k, if\ t_n > 1-k \end{cases}$$

- Huber loss

$$d(y_n, t_n;\ \delta) = \begin{cases} \frac{1}{2}(y_n - t_n)^2, if\ |y_n - t_n| < \delta \\ \delta \cdot |y_n - t_n| - \frac{1}{2}\delta^2, \quad otherwise \end{cases}$$

- Hinge loss

$$d(y_n, t_n;\ \beta) = \begin{cases} \frac{1}{2\beta}(y_n - t_n)^2, if\ |y_n - t_n| < \beta \\ |y_n - t_n|, \quad otherwise \end{cases}$$



Structured output

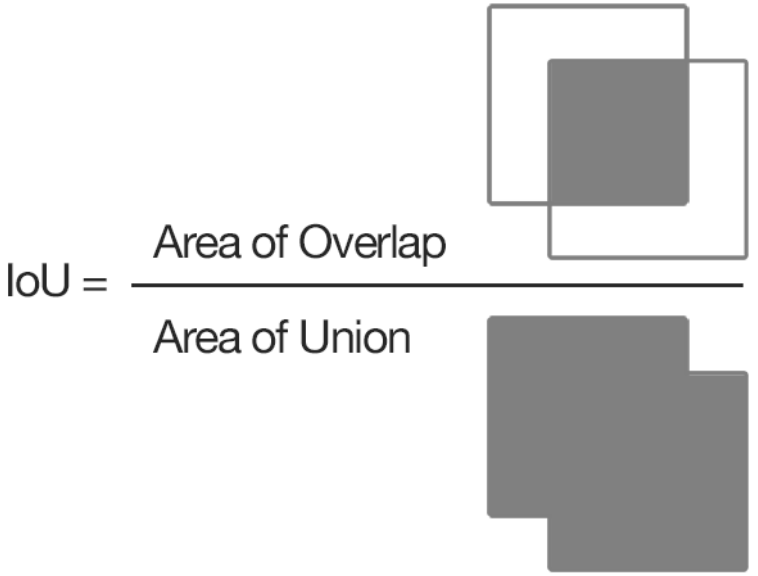
- Mean/sum "reduction" over dimensions/classes

$$d(\mathbf{y}_n, \mathbf{t}_n) = \sum_{k=1}^{K} d(y_{k,n}, t_{k,n})$$

- Set theory

e.g. Intersection-of-union (IoU), Jaccard and Dice

$$d(\mathbf{y}_n, \mathbf{t}_n) = -\frac{\sum_{k=1}^{K} y_{k,n} \cdot t_{k,n}}{\sum_{k=1}^{K} y_{k,n} + \sum_{k=1}^{K} t_{k,n}}$$



- Similarity measures

e.g. Cosine, MMD, KL, EMD, and other divergence

$$d(\mathbf{y}_n, \mathbf{t}_n) = -\frac{\mathbf{y}_n \cdot \mathbf{t}_n}{\|\mathbf{y}_n\| \cdot \|\mathbf{t}_n\|}$$

Cost sensitivity

- (Re-)weighting positives and negatives

$$d(y_n, t_n; \alpha_{pos}) = -\alpha_{pos} \cdot t_n log(y_n) - (1 - \alpha_{pos}) \cdot (1 - t_n) log(1 - y_n)$$

	Predicted +ve	Predicted -ve
Positive		£?
Negative	£?	

- Class imbalance

  i.e. Over- and under- data sampling with pre-defined frequencies

$$\ell_{\mathbf{w}}(\mathbf{y}_n, \mathbf{t}_n; \omega_n) = \frac{1}{N} \sum_{n=1}^{N} \omega_n d(\mathbf{y}_n, \mathbf{t}_n)$$

- Weighting difficulties / predicted class probabilities

e.g. Focal loss*

$$d(y_n, t_n; \alpha_t, \gamma) = -\alpha_t \cdot (1 - y_n)^{\gamma} log(y_n), \qquad for\ t_n = 1$$

Unsupervised loss

- Representation learning

  e.g. self-reconstruction autoencoder

$$\ell_{\mathbf{w}} = \frac{1}{N}\sum_{n=1}^{N}(y_n - x_n)^2$$

- Adversarial training*

  e.g. generator loss

- Regularisation*

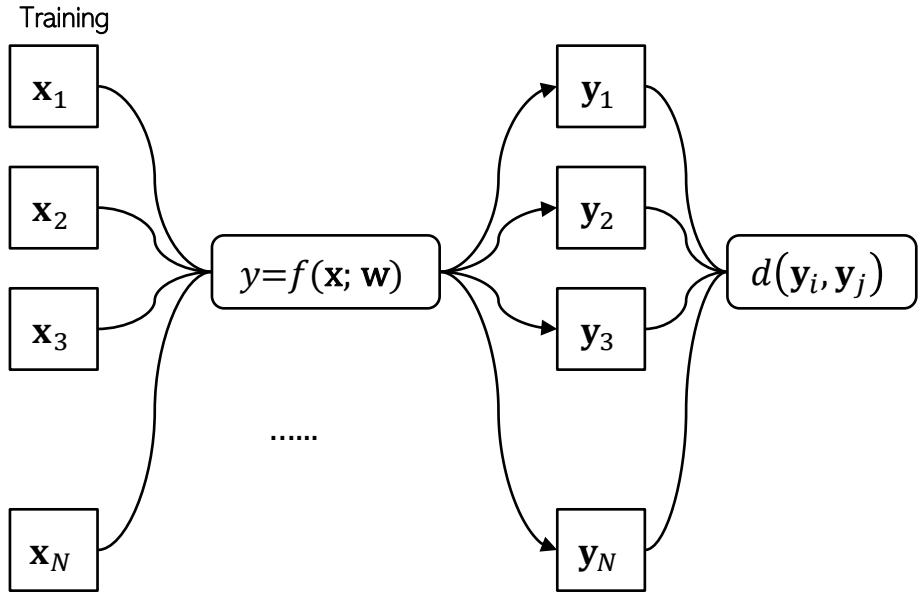
  e.g. semi-supervised, multi-task, self- and cycle consistency

- Pre-training and fine-tuning*

- Application-specific goodness-of-prediction measures*

  e.g. optical flow, registration, word embeddings

- Reward*



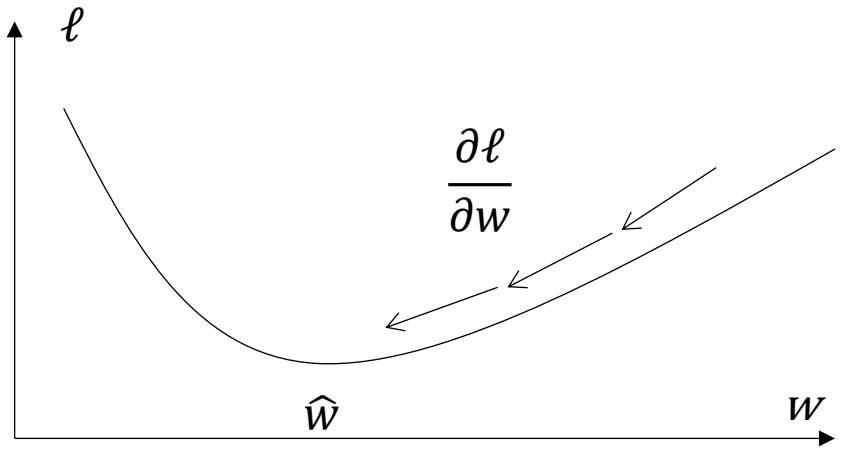
Deep Feedforward Neural Networks | Stochastic Gradient Descent

Batch gradient descent

$$\ell(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} d(f(\mathbf{x}_n; \mathbf{w}), \mathbf{t}_n) = \mathbb{E}_{(\mathbf{x}, \mathbf{t})\sim\hat{p}_{data}}[\ell(f(\mathbf{x}; \mathbf{w}), \mathbf{t})]$$
$$\hat{\mathbf{w}} = \min_{\mathbf{w}} \ell(\mathbf{w})$$

- using the entire training batch of N: $\left\{\mathbf{x}_m^{(\tau)}, \mathbf{t}_m^{(\tau)}\right\}$ in τ^{th} iteration

$$w^{(\tau+1)} = w^{(\tau)} - \eta \frac{\partial \ell}{\partial w}\left(\mathbf{w}^{(\tau)}\right)$$
$$\frac{\partial \ell}{\partial w}\left(\mathbf{w}^{(\tau)}\right) = \frac{\partial \mathbb{E}_{(\mathbf{x}, \mathbf{t})\sim\hat{p}_{data}}}{\partial w}\left(\mathbf{w}^{(\tau)}\right)$$



Stochastic gradient descent

- sampling one: $(\mathbf{x}^{(\tau)}; \mathbf{t}^{(\tau)})$

$$\mathbb{E}_{(\mathbf{x}, \mathbf{t})\sim\hat{p}_{data}} = d\left(f(\mathbf{x}^{(\tau)}; \mathbf{w}^{(\tau)}), \mathbf{t}^{(\tau)}\right)$$

Minibatch gradient descent

- sampling a minibatch with a size of M: $\left\{\mathbf{x}_m^{(\tau)}, \mathbf{t}_m^{(\tau)}\right\}$

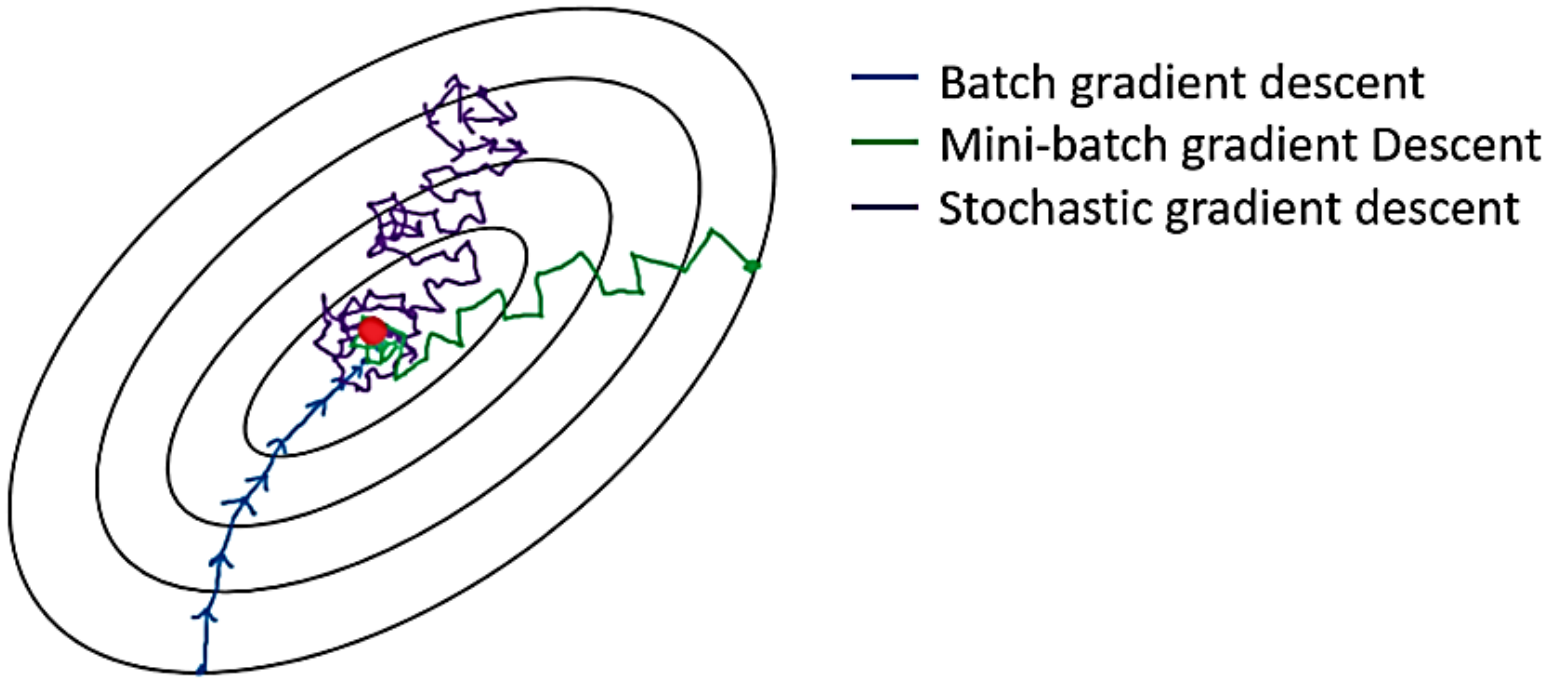
$$\mathbb{E}_{(\mathbf{x}, \mathbf{t})\sim\hat{p}_{data}} = \frac{1}{M}\sum_{m=1}^{M} d\left(f(\mathbf{x}_m^{(\tau)}; \mathbf{w}^{(\tau)}), \mathbf{t}_m^{(\tau)}\right)$$

Stochastic (minibatch) gradient descent

Efficiency

Memory requirement

Generalisation



Epoch - training data sampling and batching

Practical things to consider:

- Hardware, e.g. 2^n, memory constraint
- Training time vs. gradient accuracy
- Software, e.g. within minibatch parallelisation
- Regularising effect
- Batch normalisation* and data normalisation
- With- or without replacement? (faster convergence, maximising generalisation)
- Bias with multiple epochs?

How to estimate partial derivatives?

$$w^{(\tau+1)} = w^{(\tau)} - \eta \frac{\partial \ell(\mathbf{w}^{(\tau)})}{\partial w}$$

- Manual differentiation

  Challenging, prone to error, the non-differentiable

- Symbolic differentiation

  Complex and cryptic expression, the non-differentiable, computational complexity can be > $O(2^n)$

- Numerical differentiation

  Round-off/truncated error, computational complexity $O(n)$

- Automatic differentiation

Baydin et al "Automatic Differentiation in Machine Learning: a Survey"

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial x}$$

$$y = f(x) = f\left(h_2\left(h_1(x)\right)\right), where \begin{cases} z_1 = h_1(x) \\ z_2 = h_2(z_1) \end{cases}$$

Baydin et al "Automatic Differentiation in Machine Learning: a Survey"

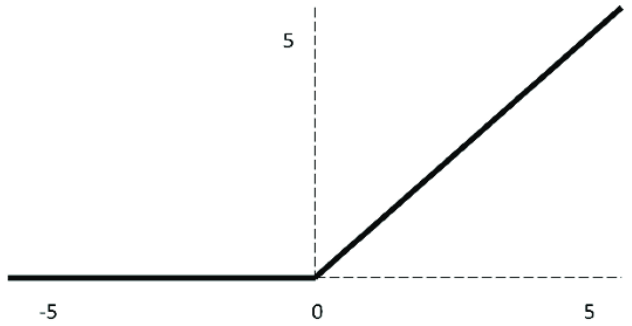
Automatic differentiation

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial x}$$

$$y = f(x) = f\left(h_2\left(h_1(x)\right)\right), where \begin{cases} z_1 = h_1(x) \\ z_2 = h_2(z_1) \end{cases}$$

"*All numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known (Verma, 2000; Griewank and Walther, 2008), and combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition.*"

- Control flow, e.g. branching, loops, recursion
- Approximating non-/piecewise-differentiable functions?
- Reverse accumulation algorithm
  - Step 1 forward evaluating
  - Step 2 reverse derivatives

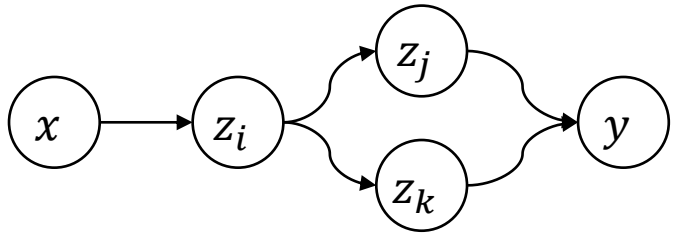


Baydin et al "Automatic Differentiation in Machine Learning: a Survey"

Automatic differentiation

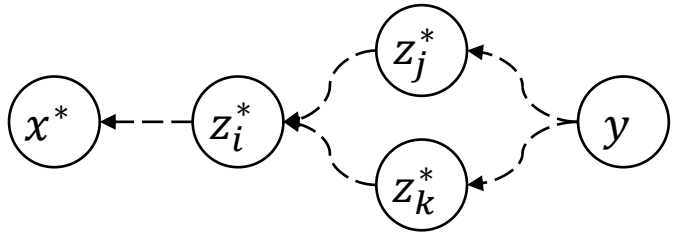
- Reverse accumulation algorithm
  - Step 1 forward evaluating all intermediate variables and book-keeping the path – computational graph

$$z_i = f_i(x)$$
$$z_j = f_j(z_i)$$
$$z_k = f_k(z_i)$$
$$y = f_y(z_i, z_k)$$



  - Step 2 reverse derivatives accumulating the stored[?] intermediate derivatives $(*)$ – using the chain rule

$$z_j^* = \frac{\partial y}{\partial z_j}$$
$$z_k^* = \frac{\partial y}{\partial z_k}$$
$$z_i^* = \frac{\partial y}{\partial z_i} = \frac{\partial y}{\partial z_j}\frac{\partial z_j}{\partial z_i} + \frac{\partial y}{\partial z_k}\frac{\partial z_k}{\partial z_i} = z_j^*\frac{\partial z_j}{\partial z_i} + z_k^*\frac{\partial z_k}{\partial z_i}$$
$$x^* = \frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_i}\frac{\partial z_i}{\partial x}$$

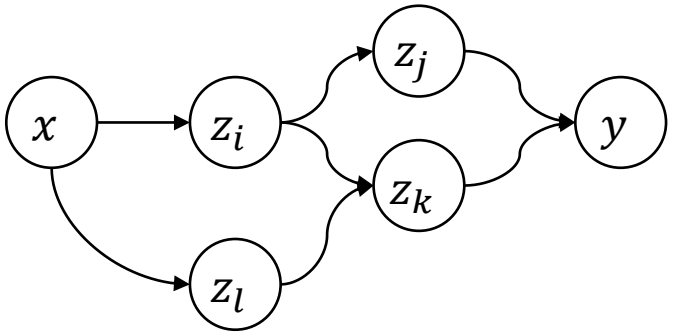


Automatic differentiation

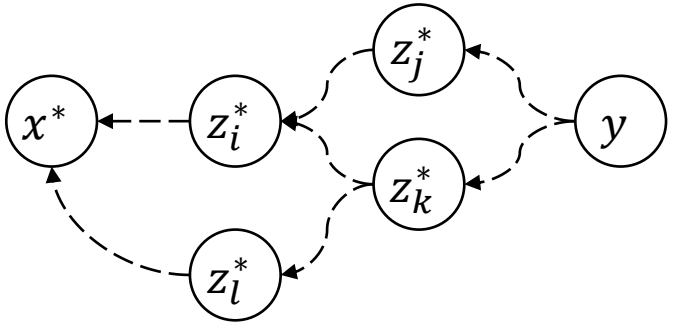
- Store the intermediate derivatives

$$z_i^* = \frac{\partial y}{\partial z_i} = \frac{\partial y}{\partial z_j}\frac{\partial z_j}{\partial z_i} + \frac{\partial y}{\partial z_k}\frac{\partial z_k}{\partial z_i} = z_j^*\frac{\partial z_j}{\partial z_i} + z_k^*\frac{\partial z_k}{\partial z_i}$$

$$z_l^* = \frac{\partial y}{\partial z_l} = \frac{\partial y}{\partial z_k}\frac{\partial z_k}{\partial z_l} = z_k^*\frac{\partial z_k}{\partial z_l}$$



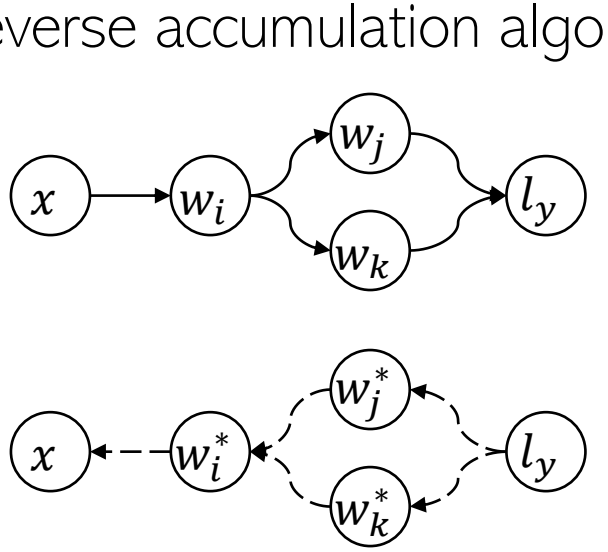
- Store vs. re-compute: memory vs. speed

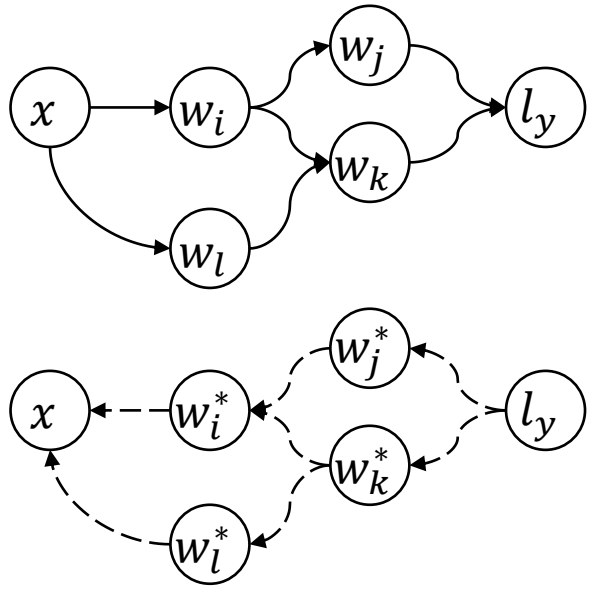


Deep Feedforward Neural Networks | Backpropagation

Computational graph

- Reverse accumulation algorithm



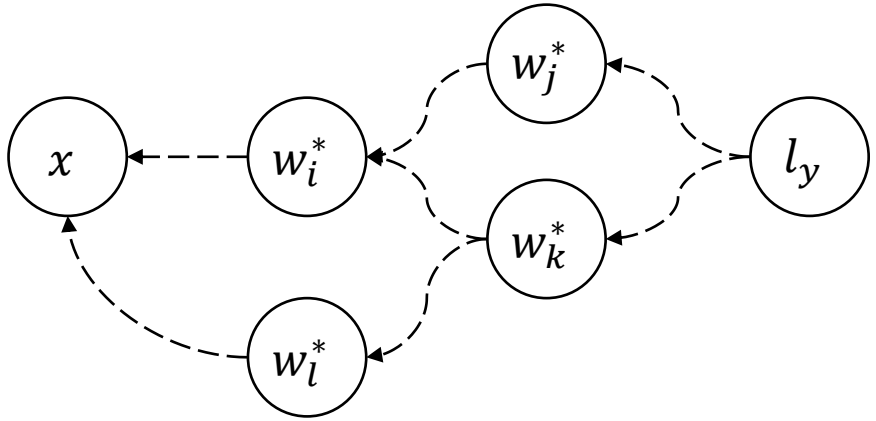


- Operations (Ops), directed edge: allowable simple functions
- Nodes: scalar, vector, matrix or tensor variables
- Output node: sigmoid / softmax / linear node

Backpropagation implementation consideration

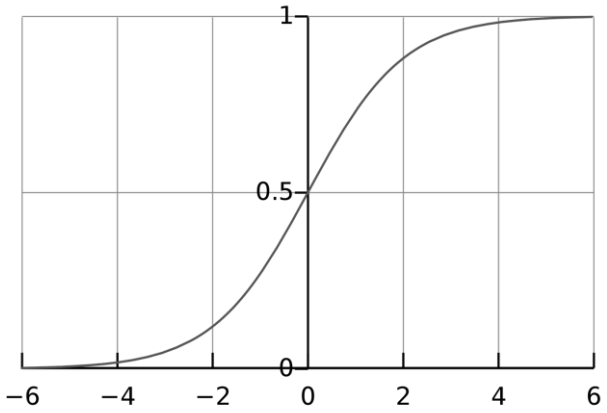
Practical things to consider

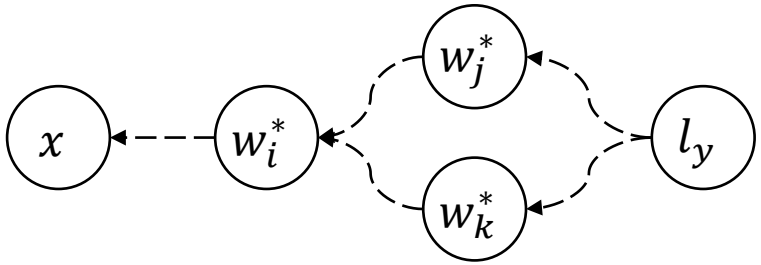
- What derivatives to store, weights, memory vs. speed
- Data type casting and numerical precision
- Differentiation, analytical ops, approximation
- Tensor-valued nodes, high dimensional
- Higher-order derivatives?
- TensorFlow and PyTorch, reference-quality implementations



Weight initialisation

- Saturation
- Symmetry
- Hierarchy

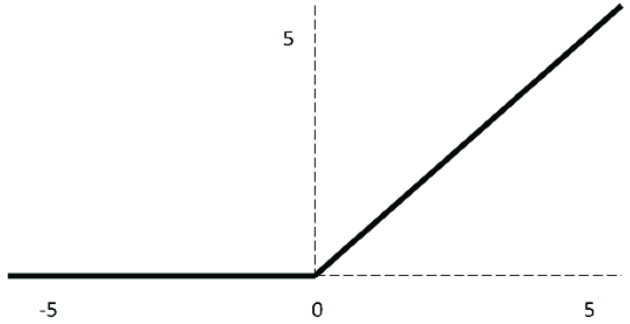




$$\frac{\partial l_y}{\partial w_0} = \frac{\partial l_y}{\partial w_2} \cdot \frac{\partial w_2}{\partial w_1} \cdot \frac{\partial w_1}{\partial w_0}$$

Weight initialisation

- Saturation
- Symmetry
- Hierarchy



- Constant initialisation, saturation with relu
- Random initialisation, Gaussian, uniform
- Xavier Glorot initialiser: $w_{i,j} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], n = J$ (size of the previous layer)
- Other initial values, e.g. precision, identity $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- Consider the evolving roles of these *de facto* techniques

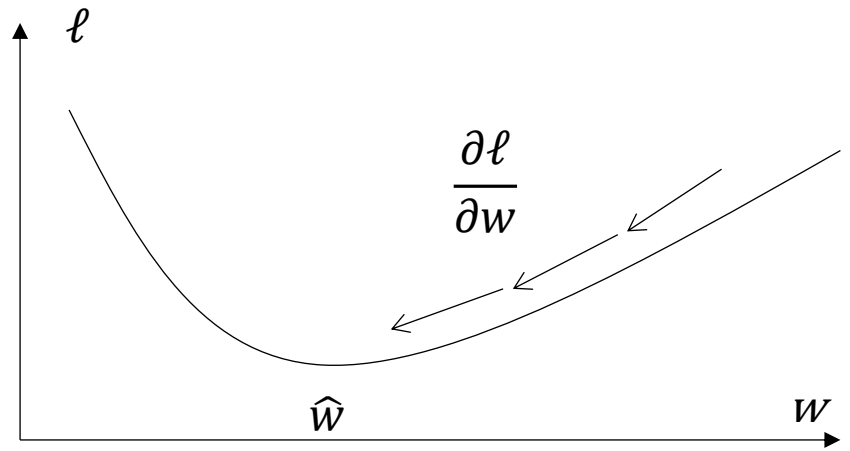
Adaptive Learning Rates

- Small vs. big learning rate / step size
- Higher-order
- Momentum (second-derivative approximation)

$$w^{(\tau+1)} = w^{(\tau)} - \eta \frac{\partial \ell\left(\mathbf{w}^{(\tau)}\right)}{\partial w}$$

$$w^{(\tau+1)} = w^{(\tau)} + v^{(\tau)}$$

$$v^{(\tau)} = \alpha v^{(\tau-1)} - \eta \frac{\partial \ell\left(\mathbf{w}^{(\tau)}\right)}{\partial w}$$



Adaptive Learning Rates

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Adaptive Learning Rates

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

Adaptive Learning Rates

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Deep Feedforward Neural Networks | Distributed Computing

Hardware

- General-purpose computing on graphics processing units (GPGPU)
- Multi-core CPU
- Multiple GPU
- TPU

Software

- (Compute Unified Device Architecture) CUDA (C/C++, Fortran)
- JAX (automatic differentiation)
- TensorFlow ecosystem
  Keras, TensorBoard, TF Addons, TF Graphics, TF Probability, TF I/O, TFX, TRFL, TensorLayer, NiftyNet…
- PyTorch ecosystem
  Torchvision, torchaudio, torchtext, torchserve, PyTorch-NLP, PyTorch Geometric Ignite, Lightning, MONAI…

- Training
- Loss Functions
- Stochastic Gradient Descent

  Minibatch | Epoch | Automatic Differentiation
- Backpropagation

  Computational Graph | Weight Initialisation | Adaptive Learning Rates
- Distributed Computing