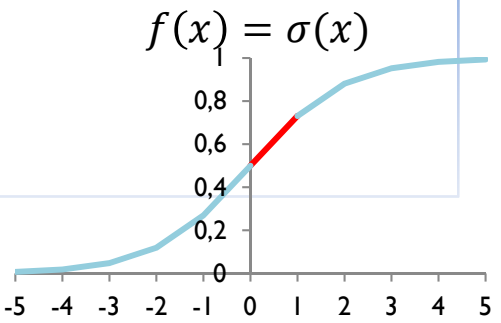
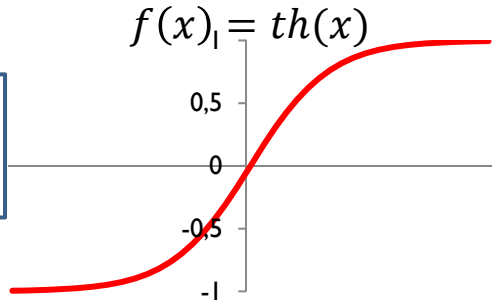
Multi-layer Perceptron

Multi-layer Perceptron (Hinton – Sejnowski – Williams 1986)

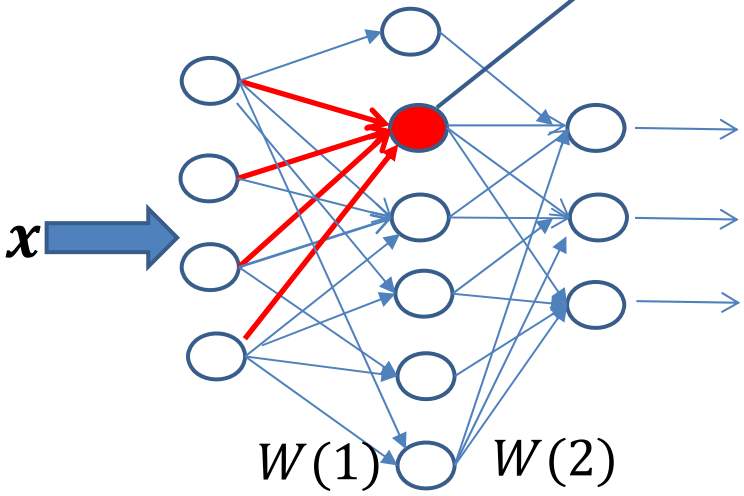
- Neurons arranged into layers
- Each neuron is a non linear unit, e.g.





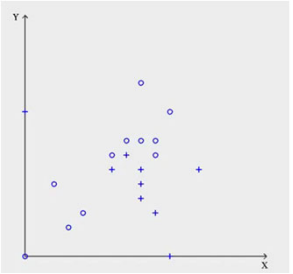
$f(\boldsymbol{w}.\boldsymbol{x})$

$\boldsymbol{w}$: **cell weight vector**



$$\hat{\boldsymbol{y}} = F_w(\boldsymbol{x}) = f\odot(W(2)f\odot(W(1)\boldsymbol{x}))$$

http://playground.tensorflow.org/



Note: $\odot$ *is a pointwise operator*, if $\boldsymbol{x} = (x_1, x_2)$, $f\odot\big((x_1, x_2)\big) = (f(x_1), f(x_2))$

Multi-layer Perceptron - Training

- **Stochastic Gradient Descent** - The algorithm is called **Back-Propagation**
  - Pick one example $(\boldsymbol{x}, \boldsymbol{y})$ or a **Mini Batch** $\{(\boldsymbol{x^i}, \boldsymbol{y^i})\}$ sampled from the training set
    - Here the algorithm is described for 1 example and for the sigmoid ($f(\quad) = \sigma(\quad)$) non linearity
  - Forward pass
    - $\hat{\boldsymbol{y}} = F_w(\boldsymbol{x}) = f\odot(W(2)f\odot(W(1)\boldsymbol{x}))$
  - Compute error
    - $c(\boldsymbol{y}, \hat{\boldsymbol{y}})$, e.g. mean square error or cross entropy
  - Backward pass
    - efficient implementation of chain rule
    - $w_{ij} = w_{ij} - \epsilon \frac{\partial c(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}}$

Note: $\odot$ *is a pointwise operator*, if $\boldsymbol{x} = (x_1, x_2)$, $f\odot((x_1, x_2)) = (f(x_1), f(x_2))$

Algorithmic differentiation

- Back-Propagation is an instance of **automatic differentiation / algorithmic differentiation - AD**
  - A mathematical expression can be written as a **computation graph**
    - i.e. graph decomposition of the expression into elementary computations
  - **AD** allows to **compute** efficiently the derivatives of every element in the graph w.r.t. any other element.
  - **AD** transforms a programs computing a numerical funtion into the program for computing the derivatives
  - All modern DL framework implement AD

Notations – matrix derivatives

$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \mathrm{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}, \alpha \in R, W\colon\ p \times q$

Vector by scalar

$$\frac{\partial x}{\partial \alpha} = \begin{pmatrix} \frac{\partial x_1}{\partial \alpha} \\ \vdots \\ \frac{\partial x_n}{\partial \alpha} \end{pmatrix}$$

Matrix by scalar

$$\frac{\partial W}{\partial \alpha} = \begin{pmatrix} \frac{\partial w_{11}}{\partial \alpha} & \cdots & \frac{\partial w_{1q}}{\partial \alpha} \\ \vdots & \ddots & \vdots \\ \frac{\partial w_{p1}}{\partial \alpha} & \cdots & \frac{\partial w_{pq}}{\partial \alpha} \end{pmatrix}$$

Scalar by vector

$$\frac{\partial \alpha}{\partial x} = \left(\frac{\partial \alpha}{\partial x_1}, \cdots, \frac{\partial \alpha}{\partial x_n} \right)$$

Scalar by matrix

$$\frac{\partial \alpha}{\partial W} = \begin{pmatrix} \frac{\partial \alpha}{\partial w_{11}} & \cdots & \frac{\partial \alpha}{\partial w_{p1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \alpha}{\partial w_{1q}} & \cdots & \frac{\partial \alpha}{\partial w_{pq}} \end{pmatrix}$$

Vector by vector

$$\frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Matrix cookbooks

http://www.cs.toronto.edu/~roweis/notes/matrixid.pdf –

http://www.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/ imm3274.pdf

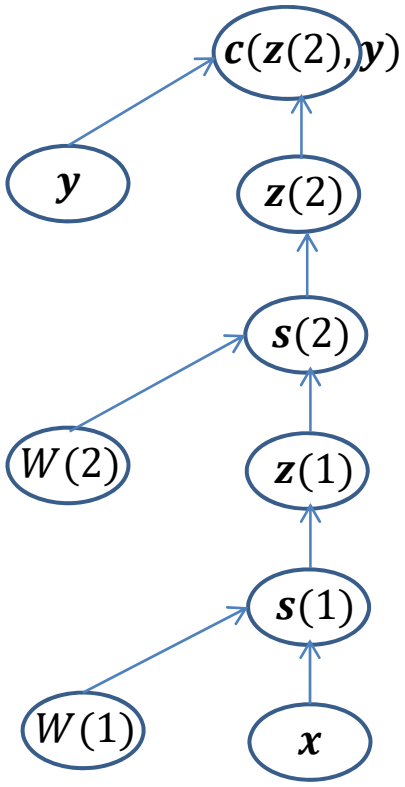
Multi-layer Perceptron - Training

- Computational graph

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

Here, $\boldsymbol{z}(2) = \hat{\boldsymbol{y}}$

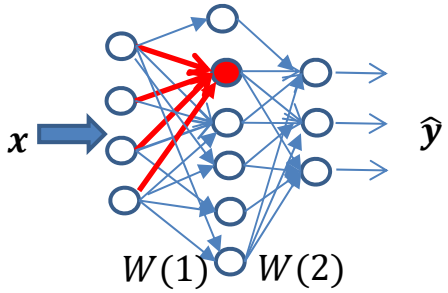
$\boldsymbol{y}$: target



Forward propagation:

$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$

$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$

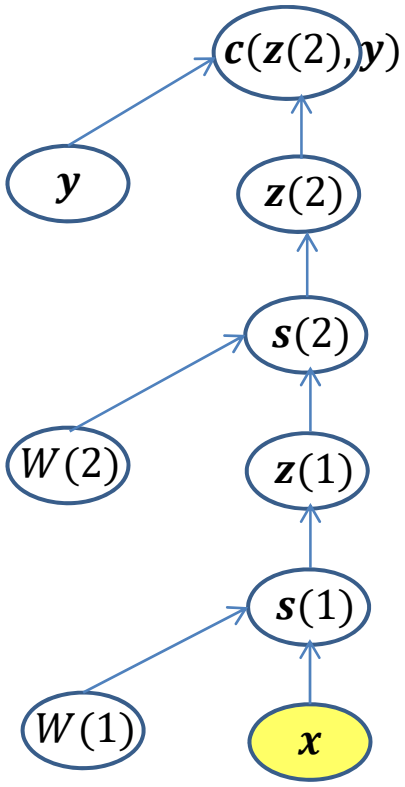


Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$

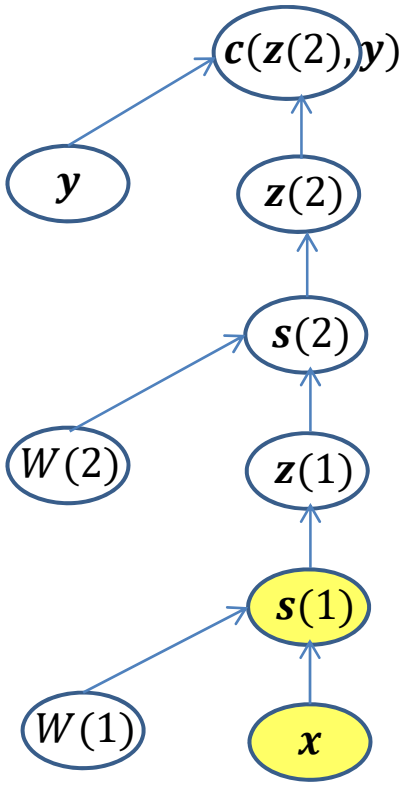
$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

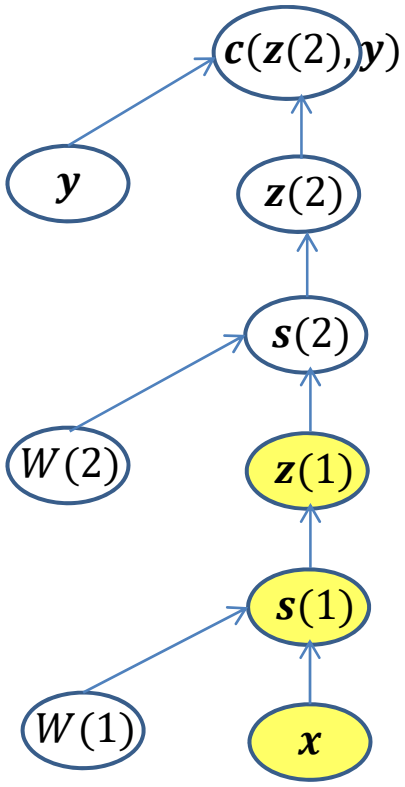
$$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$$
$$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$$

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

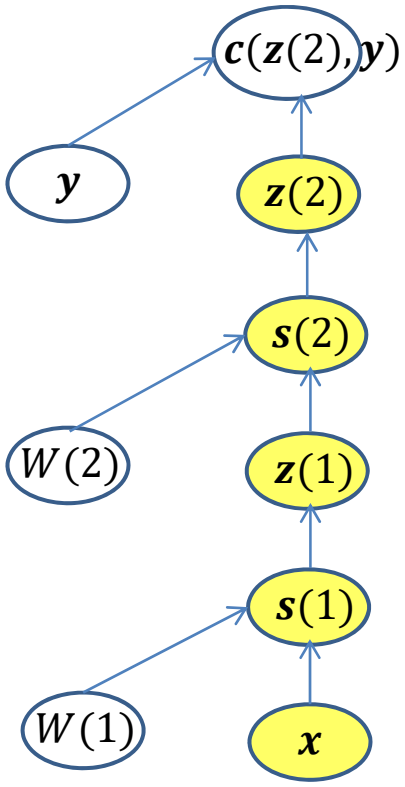
$$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$$
$$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$$

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$

$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$

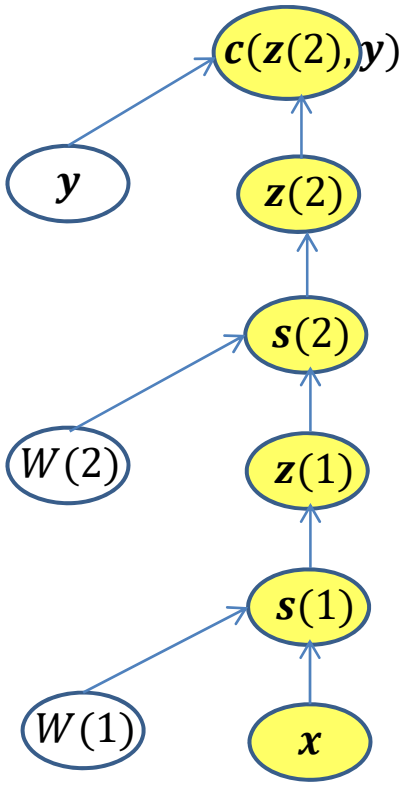
Note: $\boldsymbol{x}$, $\boldsymbol{s}(n)$, $\boldsymbol{x}(n)$ are vectors

Multi-layer Perceptron - Training

- Forward pass

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss

$\boldsymbol{y}$: target



Forward propagation:

$\boldsymbol{s}(n) = W(n)\boldsymbol{z}(n-1)$

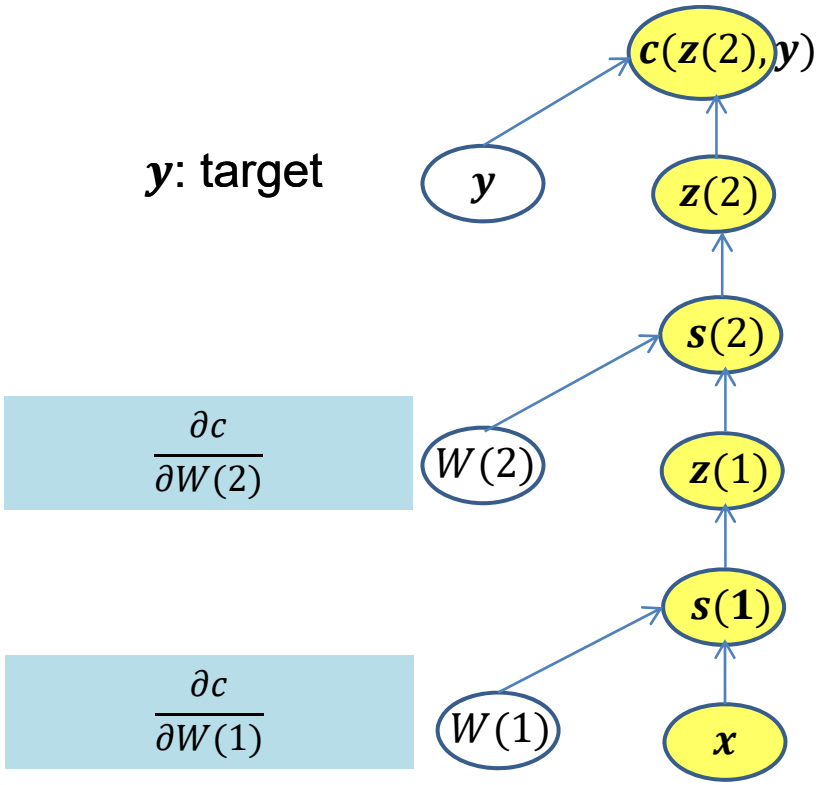
$\boldsymbol{z}(n) = \sigma(\boldsymbol{s}(n))$

Note: $\boldsymbol{x}$, $\boldsymbol{s}(n)$, $\boldsymbol{x}(n)$ are vectors

Multi-layer Perceptron - Training

- Back Propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



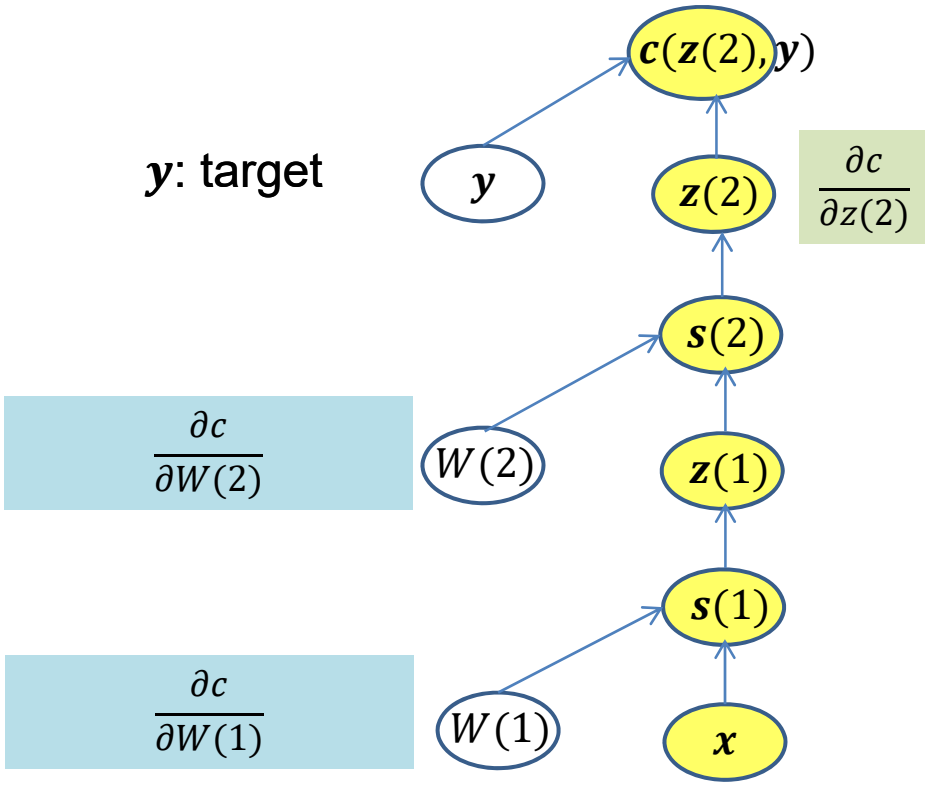
$$W = W - \epsilon \frac{\partial c}{\partial W}$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are row vectors of the appropriate size

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'\,(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1)\frac{\partial c}{\partial \boldsymbol{s}(n)}$$

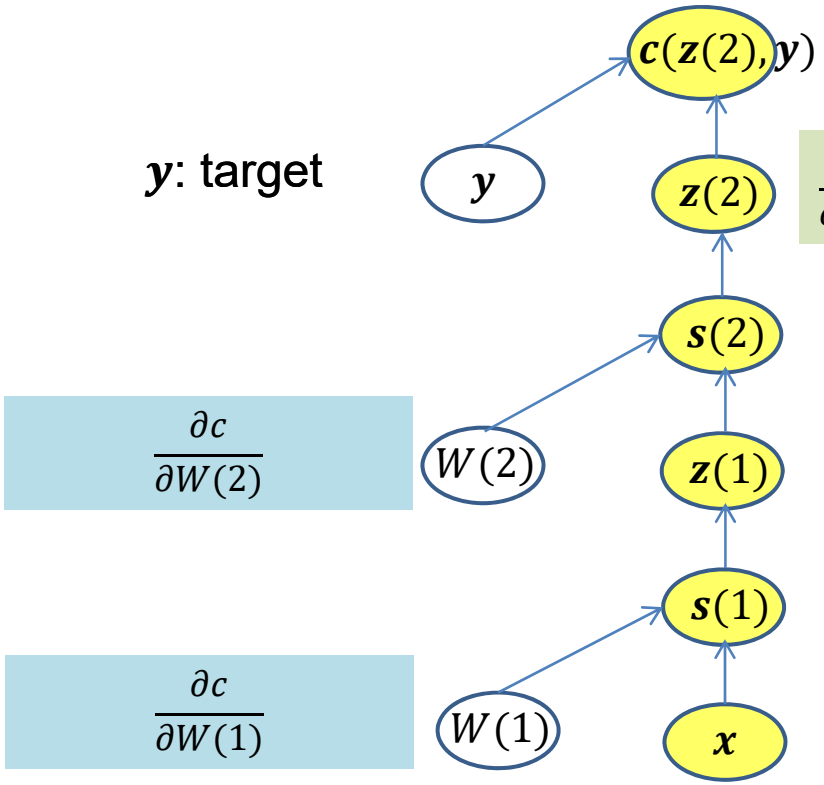
$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are row vectors of the appropriate size

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1)\frac{\partial c}{\partial \boldsymbol{s}(n)}$$

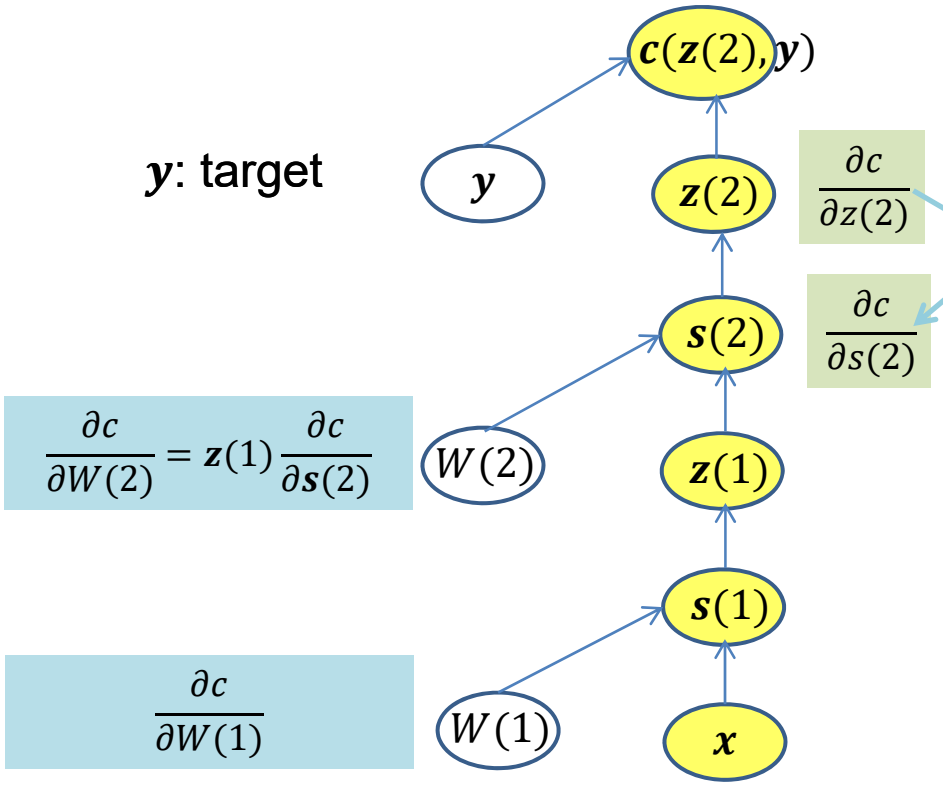
$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are row vectors of the appropriate size

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1)\frac{\partial c}{\partial \boldsymbol{s}(n)}$$

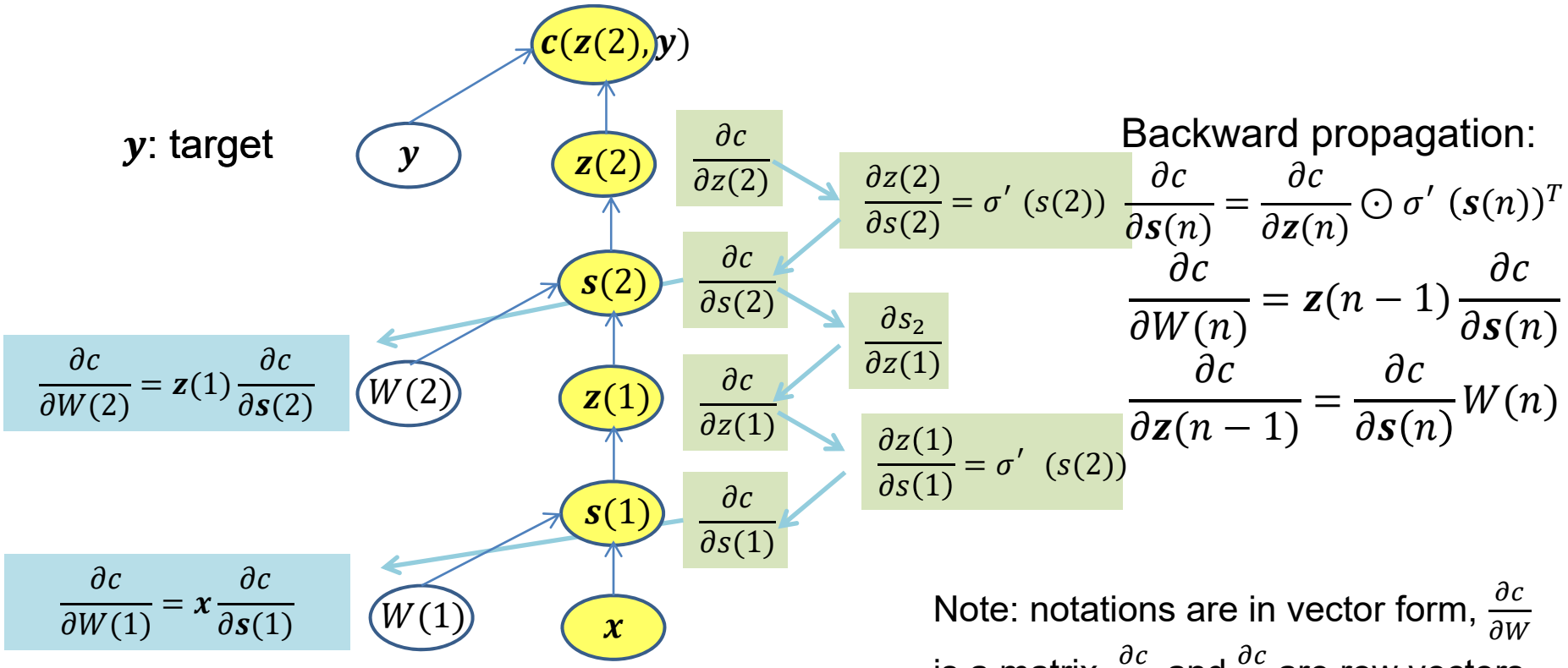
$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are row vectors of the appropriate size

Multi-layer Perceptron - Training

- Back propagation: Reverse Mode Differentiation

$\boldsymbol{c}(\boldsymbol{z}(2), \boldsymbol{y})$: loss



Backward propagation:

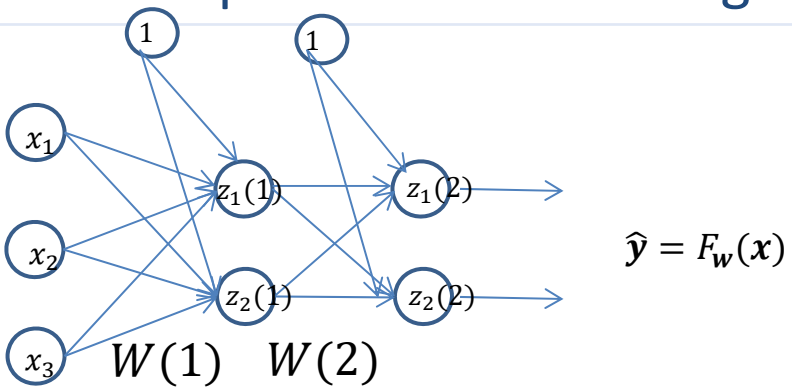
$$\frac{\partial c}{\partial \boldsymbol{s}(n)} = \frac{\partial c}{\partial \boldsymbol{z}(n)} \odot \sigma'(\boldsymbol{s}(n))^T$$

$$\frac{\partial c}{\partial W(n)} = \boldsymbol{z}(n-1) \frac{\partial c}{\partial \boldsymbol{s}(n)}$$

$$\frac{\partial c}{\partial \boldsymbol{z}(n-1)} = \frac{\partial c}{\partial \boldsymbol{s}(n)} W(n)$$

Note: notations are in vector form, $\frac{\partial c}{\partial W}$ is a matrix, $\frac{\partial c}{\partial \boldsymbol{z}}$ and $\frac{\partial c}{\partial \boldsymbol{s}}$ are row vectors of the appropriate size

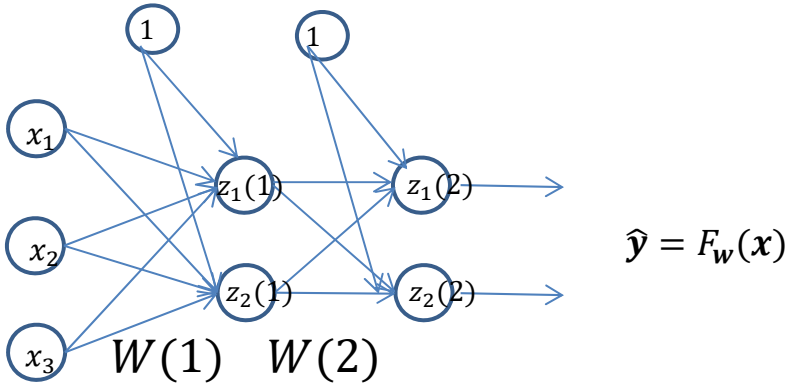
Multi-layer Perceptron – SGD Training – example - notations



$\hat{\boldsymbol{y}} = F_{\boldsymbol{w}}(\boldsymbol{x})$

- Notations
  - $\boldsymbol{z}(i)$ activation vector for layer i
  - $z_j(i)$ activation of neuron j in layer i
  - $W(i+1)$weight matrix from layer i to layer $i+1$, including bias weights $w_{jk}(i)$weight from cell k on layer i to cell j on layer $i+1$
  - $\hat{\boldsymbol{y}}$ computed output
  - $\hat{y}_1 = z_1(2) = g(w_{10}(2) + w_{11}(2)z_1^{(1)} + w_{12}(2)z_2\ (1))$
  - $z_1(1) = g(w_{10}(1) + w_{11}(1)x_1 + w_{12}(1)x_2 + w_{13}(1)x_3)$
  - $W(1) = \begin{pmatrix} w_{10}(1) & w_{11}(1) & w_{12}(1) & w_{13}(1) \\ w_{20}(1) & w_{21}(1) & w_{22}(1) & w_{23}(1) \end{pmatrix}$

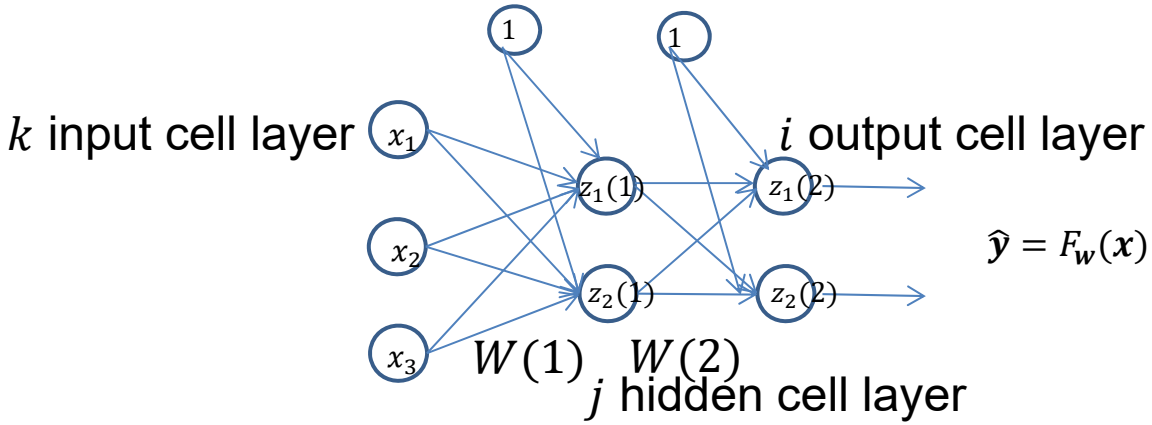
Multi-layer Perceptron – SGD Training – Detailed derivation for a 1 hidden layer network (MSE loss + sigmoid units) - forward pass



- For example x
  - The activations of all the neurons from layer 1 are computed in parallel
  - $\boldsymbol{s}(1) = W(1)\boldsymbol{x}$ then $\boldsymbol{z}(1) = g(\boldsymbol{s}(1))$
    - with $g(\boldsymbol{s}(1)) = (g(\boldsymbol{s}_1(1)), g(\boldsymbol{s}_2(1)))^{\mathrm{T}}$
  - The activations of cells on layer 1 are then used as inputs for layer 2. The activations of cells in layer 2 are computed in parallel.
  - $\boldsymbol{s}(2) = W(2)\boldsymbol{z}(1)$ then $\hat{\boldsymbol{y}} = \boldsymbol{z}(2) = g(\boldsymbol{s}(2))$
    -

Multi-layer Perceptron – SGD derivation
Detailed derivation for a 1 hidden layer network (MSE loss + sigmoid units)

- Forward pass
  - Indices used below for this detailed derivation: i output cell layer, j hidden cell layer, k input cell layer



  - $s_j\ (1) = \sum_k\ w_{jk}(1)x_k\ ,\ z_j(1)\ = g(s(1))$
  - $s_i(2) = \sum_j\ w_{ij}(2)z_j(1)\ ,\ z_i(2) = g(s_i(2))$
    - $s_i(2) = \sum_j\ w_{ij}(2)g(\sum_k\ w_{jk}(1)x_k)\ , z_i(2) = g(\sum_j\ w_{ij}(2)g(\sum_k\ w_{jk}(1)x_k)\ )$
- Loss
  - $c = \frac{1}{2}\sum_i\ (y_i - \hat{y}_i)^2 = \frac{1}{2}\sum_i\ \left(y_i - g(\sum_j\ w_{ij}(2)z_j(1))\right)^2$

Multi-layer Perceptron – SGD derivation
Detailed derivation for a 1 hidden layer network (MSE loss + sigmoid units)

- Backward (derivative) pass
  - Upgrade rule for weight w_{ij}, layer m: $w_{ij}(m) = w_{ij}(m) + \Delta w_{ij}(m)$
  - 2nd weight layer
    - $\Delta w_{ij}(2) = -\epsilon \frac{\partial C}{\partial w_{ij}(2)} = -\epsilon \frac{\partial C}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{w_{ij}(2)}$
    - $\Delta w_{ij}(2) = \epsilon (y_i - \hat{y}_i) \frac{\partial \hat{y}_i}{\partial s_i(2)} \frac{\partial s_i(2)}{\partial w_{ij}(2)}$
    - $\Delta w_{ij}(2) = \epsilon (y_i - \hat{y}_i) g'(s_i(2)) z_j(1)$
    - $\Delta w_{ij}(2) = \epsilon e_i(2) z_j(1)$, with $e_i(2) = (y_i - \hat{y}_i) g'(s_i(2))$
  - 1st weight layer
    - $\Delta w_{ij}(1) = -\epsilon \frac{\partial C}{\partial w_{ij}(1)} = -\epsilon \frac{\partial C}{\partial z_j(1)} \frac{\partial z_j(1)}{\partial w_{ij}(1)}$
      - $\frac{\partial C}{\partial z_j(1)} = \sum_{i\ parents\ of\ j} \frac{\partial C}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_j(1)} = -\sum_i (y_i - \hat{y}_i) \frac{\partial \hat{y}_i}{\partial s_i(2)} \frac{\partial s_i(2)}{\partial z_j(1)}$
      - $\frac{\partial C}{\partial z_j(1)} = -\sum_i (y_i - \hat{y}_i) g'(s_i(2)) w_{ij}(2)$

Multi-layer Perceptron – SGD derivation
Detailed derivation (MSE loss + sigmoid units)

- $\frac{\partial z_j(1)}{\partial w_{jk}(1)} = \frac{\partial z_j(1)}{\partial s_j(1)} \frac{\partial s_j(1)}{\partial w_{jk}(1)} = g'\big(s_j(1)\big) z_k$

- $\Delta w_{jk}(1) = \epsilon \sum_{i\ parents\ of\ j} (y_i - \hat{y}_i) g'(s_i(2)) w_{ij}(2)\ g'\big(s_j(1)\big) x_k$
- $\Delta w_{jk}(1) = \epsilon e_j(1) x_k$ with $e_j = g'\big(s_j(1)\big) \sum_{i\ parents\ of\ j} e_i w_{ij}(2)$

Back Propagation and Adjoint

- BP is an instance of a more general technique: the Adjoint method
- Adjoint method
  - has been designed for computing **efficiently** the sensitivity of a loss to the parameters of a function (e.g. weights, inputs or any cell value in a NN).
  - Can be used to solve different constrained optimization problems (including BP)
  - Is used in many fields like control, geosciences
  - Interesting to consider the link with the adjoint formulation since this opens the way to generalization of the BP technique to more general problems
    - e.g. continuous NNs (Neural ODE)

Back Propagation and Adjoint

- Learning problem
  - $Min_W c = \frac{1}{N}\sum_{k=1}^{N} c(F(x^k), y^k)$
  - With $F(x) = F_l \circ \cdots \circ F_1(x)$
- Rewritten as a constrained optimisation problem
  - $Min_W c = \frac{1}{N}\sum_{k=1}^{N} c(z^k(l), y^k)$
  - Subject to
$$\begin{cases} z^k(l) = F_l(z^k(l-1), W(l)) \\ z^k(l-1) = F_{l-1}(z^k(l-2), W(l-1)) \\ \cdots \\ z^k(1) = F_1\left(x^k, W(1)\right) \end{cases}$$
  - Note
    - z and W are vectors of the appropriate size
    - e.g. $z(i)$ is $n_z(i) \times 1$ and $W(i)$ is $n_W(i) \times 1$

Back Propagation and Adjoint

- For simplifying, one considers pure SGD, i.e. $N = 1$
  - So that we drop the index k
- The Lagrangian associated to the optimization problem is
  - $\mathcal{L}(x, W) = c(z(l), y) - \sum_{i=1}^{l} \lambda_i^T (z(i) - F_i(z(i-1), W(i))$
  - Unknowns to be estimated:
    - $z(i), W(i), \lambda_i, i = 1 \dots l,$

Back Propagation and Adjoint

- We want to solve for the Lagrangian
  - $\mathcal{L}(x, W) = c(z(l), y) - \sum_{i=1}^{l} \lambda_i^T (z(i) - F_i(z(i-1), W(i))$
  - with unknowns: $z(i), W(i), \lambda_i, i = 1, \dots, l$
- The partial derivatives of the Lagrangian are
  - $\frac{\partial \mathcal{L}}{\partial z(l)} = -\lambda_l^T + \frac{\partial c(z(l),y)}{\partial z(l)}$ for the last layer l
  - $\frac{\partial \mathcal{L}}{\partial z(i)} = -\lambda_i^T + \lambda_{i+1}^T \frac{\partial F_{i+1}(z(i),W(i+1))}{\partial z(i)}, \; i = 1, \dots, l-1$ for intermediate layer i
  - $\frac{\partial \mathcal{L}}{\partial W(i)} = \lambda_i^T \frac{\partial F_i(z(i-1),W(i))}{\partial W(i)}, \; \mathrm{i} = 1 \dots \mathrm{l}$
  - $\frac{\partial \mathcal{L}}{\partial \lambda_i} = z(i) - F_i(z(i-1), W(i)) \; , \; \mathrm{i} = 1 \dots \mathrm{l}$
  - Note
    - $\frac{\partial \mathcal{L}}{\partial z(i)}$ is $1 \times n_z(i)$, $\frac{\partial \mathcal{L}}{\partial W_i}$ is $1 \times n_W(i)$, $\frac{\partial \mathcal{L}}{\partial \lambda_i}$ is $1 \times n_\lambda(i)$, λ_i is $n_z(i) \times 1$, $\frac{\partial F_{i+1}(z(i),W(i+1))}{\partial z(i)}$ is $n_z(i+1) \times n_z(i)$, $\frac{\partial c(z(l),y)}{\partial z(l)}$ is $1 \times n_z(l)$, $\frac{\partial F_i(z(i-1),W(i))}{\partial W(i)}$ is $n_z(i) \times n_W(i)$

Back Propagation and Adjoint

- Forward equation
  - $\frac{\partial \mathcal{L}}{\partial \lambda_i} = z(i) - F_i(z(i-1), W(i))$, i = 1 ... l, represent the constraints
  - One wants $\frac{\partial \mathcal{L}}{\partial \lambda_i} = 0, \quad i = 1 \dots l$
  - Starting from $i = 1$ up to $i = l$, this is exactly the forward pass of BP
- Backward equation
  - Remember the Lagrangian
    - $\mathcal{L}(x, W) = c(z(l), y) - \sum_{i=1}^{l} \lambda_i^T (z(i) - F_i(z(i-1), W(i))$
  - Since one imposes $(z(i) - F_i\big(z(i-1), W(i)\big) = 0$ (forward pass), one can choose λ_i^T as we want
  - Let us choose the λs such that $\frac{\partial \mathcal{L}}{\partial z(i)} = 0, \forall\, i$
  - The λs can be computed backward Starting at $i = l$ down to to $i = 1$
    - $\lambda_l^T = \frac{\partial c(z(l), y)}{\partial z(l)}$
    - …
    - $\lambda_i^T = \lambda_{i+1}^T \frac{\partial F_{i+1}(z(i), w(i+1))}{\partial z(i)} = \lambda_{i+1}^T \frac{\partial z(i+1)}{\partial z(i)}$

Back Propagation and Adjoint

- Derivatives
  - All that remains is to compute the derivatives of $\mathcal{L}$ wrt the W_i
    - $\frac{\partial \mathcal{L}}{\partial W(i)} = \lambda_{i+1}^T \frac{\partial F_i(z(i-1),W(i))}{\partial W(i)}$, $\forall\, i$

      - $\frac{\partial F_i(z(i-1),W(i))}{\partial W(i)} = \frac{\partial z(i)}{\partial W(i)}$ easy to compute

Back Propagation and Adjoint – Algorithm Recap

- Recap, BP algorithm with Adjoint
- Forward
  - Solve forward $\frac{\partial \mathcal{L}}{\partial \lambda_i} = 0$
    - $z(1) = F_1(z(0), W(1))$
    - …
    - $z(i) = F_i(z(i-1), W(i))$
- Backward
  - Solve backward $\frac{\partial \mathcal{L}}{\partial z(i)} = 0$
    - $\lambda_l^T = \frac{\partial c(z(l), y)}{\partial z(l)}$
    - …
    - $\lambda_i^T = \lambda_{i+1}^T \frac{\partial F_{i+1}(z(i), w(i+1))}{\partial z(i)} = \lambda_{i+1}^T \frac{\partial z(i+1)}{\partial z(i)}$
- Derivatives
  - $\frac{\partial \mathcal{L}}{\partial W(i)} = \lambda_{i+1}^T \frac{\partial F_i(z(i-1), W(i))}{\partial W(i)}$, $\forall\, i$

Adjoint method – Adjoint equation

- Let us consider the Lagrangian written in a simplified form
  - $\mathcal{L}(x, W) = c(z(l), y) - \lambda^T g(z, W)$
    - z, W represent respectively all the variables of the NN and all the weights
    - z is a $1 \times n_z$ vector, and W is a $1 \times n_W$ vector
    - $g(z, W) = 0$ represents the constraints written in an implicit form
      - here the system $z(i) - F_{l-1}\big(z(i-1), W(i)\big) = 0, i = 1 \dots l$

The derivative of $\mathcal{L}(x, W)$ wrt W is

- $\frac{d\mathcal{L}(x,W)}{dW} = \frac{\partial c}{\partial z}\frac{\partial z}{\partial W} - \lambda^T\left(\frac{\partial g}{\partial z}\frac{\partial z}{\partial W} + \frac{\partial g}{\partial W}\right)$
- $= \left(\frac{\partial c}{\partial z} - \lambda^T \frac{\partial g}{\partial z}\right)\frac{\partial z}{\partial W} + \lambda^T \frac{\partial g}{\partial W}$
- In order to avoid computing $\frac{\partial z}{\partial W}$, choose λ such that
  - $\frac{\partial c}{\partial z} - \lambda^T \frac{\partial g}{\partial z} = 0$, rewriten as:

$$\boldsymbol{\frac{\partial g^T}{\partial z}\,\lambda = -\frac{\partial c}{\partial z}} \qquad \text{<<<<<<<<< Adjoint Equation}$$

Adjoint method

- λ is determined from the Adjoint equation
  - Different options for solving λ, depending on the problem
  - For MLPs, the hierarchical structure leads to the backward scheme

Multi-layer Perceptron – stochastic gradient

- Note
  - The algorithm has been detailed for « pure » SGD, i.e. one datum at a time
  - In practical applications, one uses mini-batch implementations
  - This accelerates GPU implementations
  - The algorithm holds for any differentiable loss/ model
  - Deep Learning on large architectures makes use of SGD variants, e.g. Adam

Loss functions

- Depending on the problem, and on model, different loss functions may be used
- Mean Square Error
  - For regression
- Classification, Hinge, logistic, cross entropy losses
  - Classification loss
    - Number of classification errors
    - Exemples
      - $\hat{\boldsymbol{y}} \in R^p, \boldsymbol{y} \in \{-1,1\}^p$
  - Hinge, logistic losses are used as proxies for the classification loss

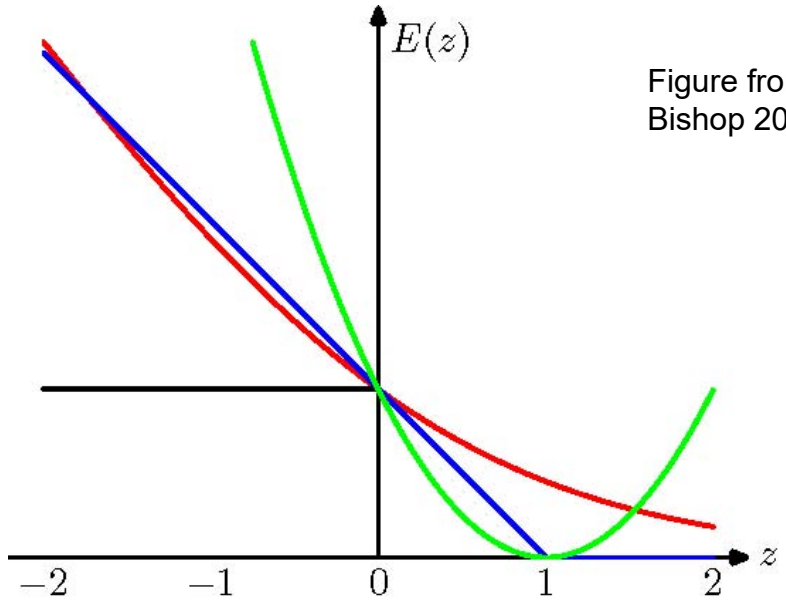


Figure from Bishop 2006

z coordinate: $z = \hat{\boldsymbol{y}}.\boldsymbol{y}$ (margin)

$$C_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2$$
$$C_{hinge}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = [1 - \hat{\boldsymbol{y}}.\boldsymbol{y}]_+ = \max(0, 1 - \hat{\boldsymbol{y}}.\boldsymbol{y})$$
$$C_{logistic}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \ln(1 + \exp(-\hat{\boldsymbol{y}}.\boldsymbol{y}))$$

Approximation properties of MLPs

- Results based on functional analysis
  - (Cybenko 1989)
    - Theorem 1 (regression): Let f be a continuous saturating function, then the space of functions $g(x) = \sum_{j=1}^{n} v_j f(\mathbf{w}_j . . \mathbf{x})$ is dense in the space of continuous functions on the unit cube $C(I)$. i.e. $\forall h \in C(I) et \; \forall \epsilon > 0, \exists \; g : |g(x) - h(x)| < \epsilon$ on I
    - Theorem 2 (classification): Let f be a continuous saturating function. Let F be a decision function defining a partition on I. Then $\forall \epsilon > 0$, there exists a function $g(x) = \sum_{j=1}^{n} v_j f(\mathbf{w}_j . . \mathbf{x})$ and a set $D \subset I$ such that $measure(D) = 1 - \epsilon(D)$ and $|g(x) - F(x)| < \epsilon$ on D
    - .
  - (Hornik et al., 1989)
    - Theorem 3 : For any increasing saturating function f, and any probability measure m on R^n , the space of functions $g(x) = \sum_{j=1}^{n} v_j f(\mathbf{w}_j . . \mathbf{x})$ is uniformely dense on the compact sets $C(R^n)$ - the space of continuous functions on R^n
  - Notes:
    - None of these result is constructive
    - Recent review of approximation properties of NN: Guhring et al., 2020, Expressivity of deep neural networks, arXiv:2007.04759

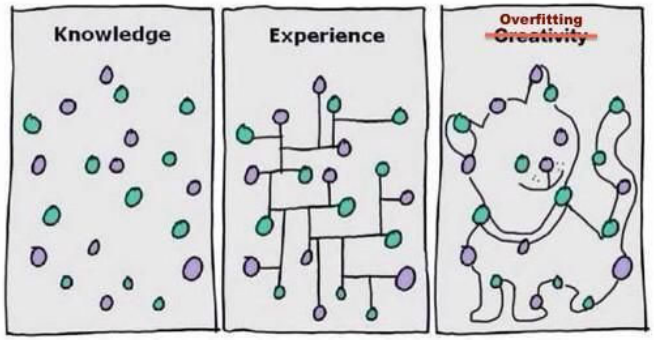
Complexity control

Bias – Variance

Overtraining and regularization

Generalization and Model Selection

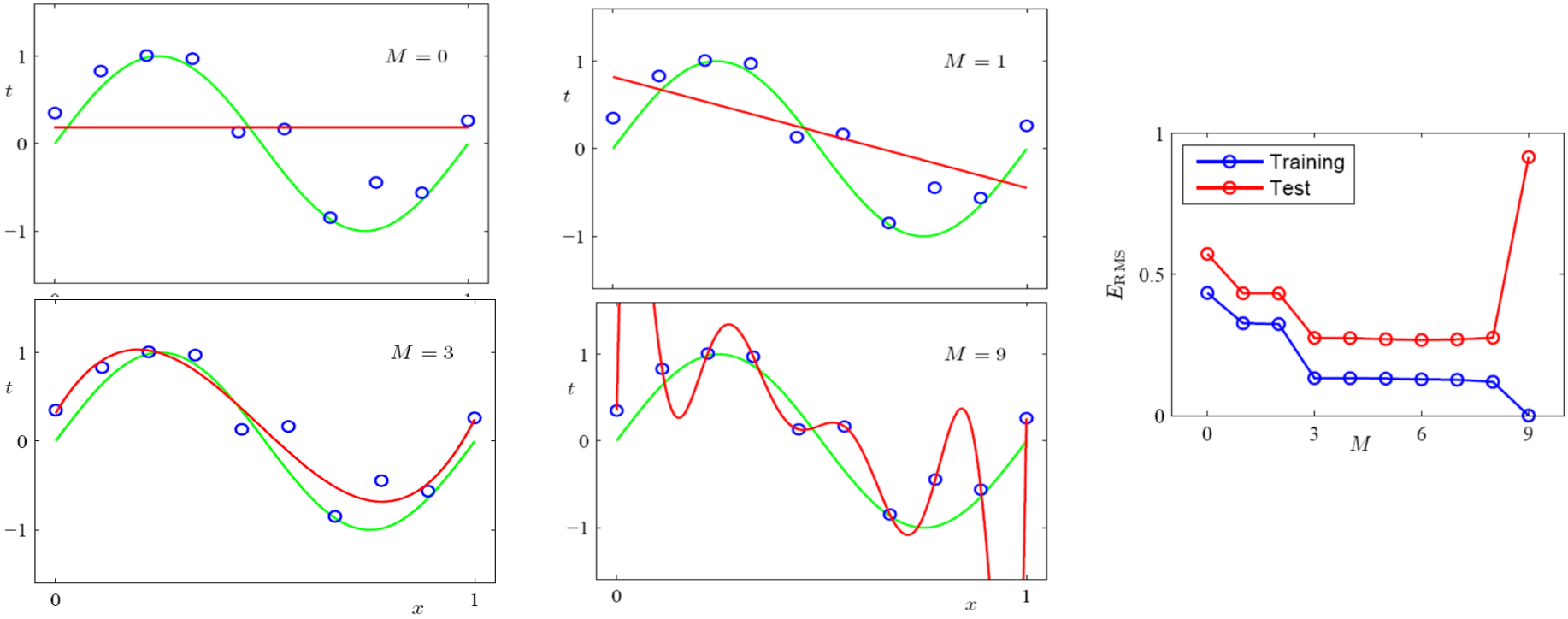
- Complex models sometimes perform worse than simple linear models
  - Overfitting/ generalization problem



- Empirical Risk Minimization is not sufficient
  - The model complexity should be adjusted both to the task and to the information brought by the examples
  - Both the model parameters and the model capacity should be learned
  - Lots of practical method and of theory has been devoted to this problem

Complexity control
Overtraining / generalization for regression

- **Example** (Bishop 06) fit of a sinusoid with polynomials of varying degrees



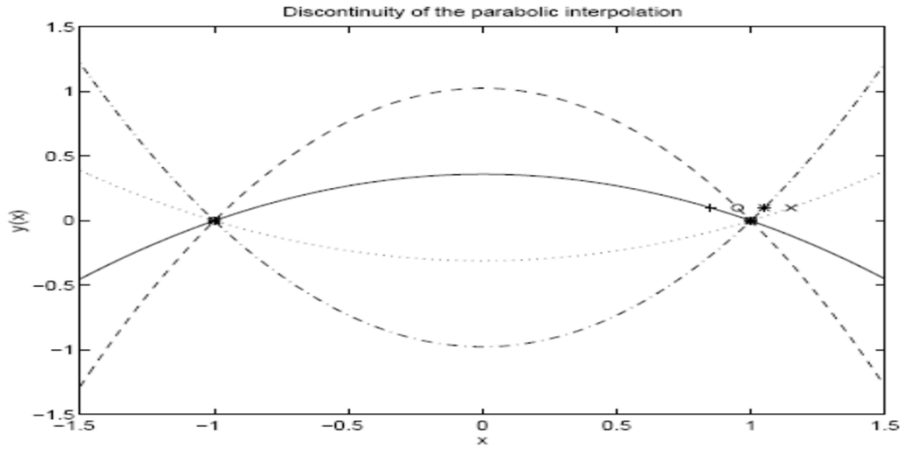
- Model complexity shall be controlled (learned) during training
  - How?

Complexity control

- One shall optimize the risk while controling the complexity
- Several methods
  - Régularisation (Hadamard …Tikhonov)
    - Theory of ill posed problems
  - Minimization of the structural risk (Vapnik)
  - Algebraic estimators of generalization error (AIC, BIC, LOO, etc)
  - Bayesian learning
    - Provides a statistical explanation of regularization
    - Regularization terms appear as priors on the parameter distribution
  - Ensemble methods
    - Boosting, bagging, etc
  - Many others especially in the Deep NN literature (seen later)

Regularisation

- Hadamard
  - A problem is well posed if
    - A solution exists
    - It is unique and stable
  - Example of ill posed problem (Goutte 1997)



- Tikhonov
  - Proposes methods pour transforming a ill posed problem into a "well" posed one

Bias-variance decomposition

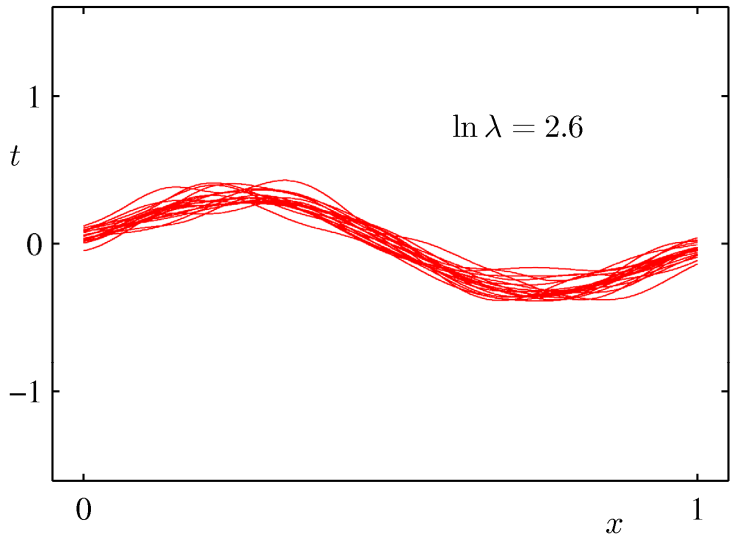
- Illustrates the problem of model selection, puts in evidence the influence of the complexity of the model
  - Remember: MSE risk decomposition
    - $E_{x,y}\left[\left(y - F_w(\boldsymbol{x})\right)^2\right] = E_{x,y}\left[\left(y - E_y[y|\boldsymbol{x}]\right)^2\right] + E_{x,y}\left[\left(E_y[y|\boldsymbol{x}] - F_w(\boldsymbol{x})\right)^2\right]$
    - Let $h^*(x) = E_y[y|\boldsymbol{x}]$ be the optimal solution for the minimization of this risk
  - In practice, the number of training data for estimating $E_y[y|\boldsymbol{x}]$ is limited
    - The estimation will depend on the training set D
    - Uncertainty due to the training set choice for this estimator can be measured as follows:
      - Sample a series of training sets, all of size N: $D_1, D_2, \ldots$
      - Learn $F_w(\boldsymbol{x}, D)$ for each of these datasets
      - Compute the mean of the empirical errors obtained on these different datasets

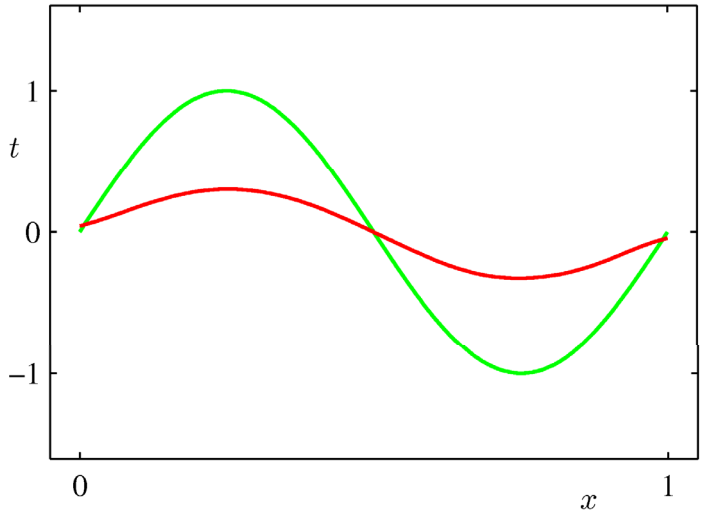
Bias-variance decomposition

- Let us consider the quadratic error $\left(F(x;D) - h^*(x)\right)^2$ for a datum x and for the solution $F_w(x;D)$ obtained with the training set D (in order to simplify, we consider a 1 dimensional real output, extension to multidimensional outputs is trivial)
  - Let $E_{D\sim p(D)}[F_w(x;D)]$ denote the expectation w.r.t. the distribution of $D, p(D)$
- $\left(F_w(x;D) - h^*(x)\right)^2$ decomposes as:
  - $\left(F_w(x;D) - h^*(x)\right)^2 = \left(F_w(x;D) - E_D[F_w(x;D)] + E_D[F_w(x;D)] - h^*(x)\right)^2$
  - $\left(F_w(x;D) - h^*(x)\right)^2 = \begin{array}{l}(F_w(x;D) - E_D[F_w(x;D)])^2 + \left(E_D[F_w(x;D)] - h^*(x)\right)^2 \\ + 2(F_w(x;D) - E_D[F_w(x;D)])\left(E_D[F_w(x;D)] - h^*(x)\right)\end{array}$
- Expectation w.r.t. D distribution decomposes as:
  - $E_D[\left(F_w(x;D) - h^*(x)\right)^2] = (E_D[F_w(x;D)] - h^*(x))^2 + E_D[(F_w(x;D) - E_D[F_w(x;D)])^2]$
  - $= \quad bias^2 \quad + \quad variance$
- Intuition
  - Choosing the right model requires a compromise between flexibility and simplicity
    - *Flexible model* : low bias – strong variance
    - *Simple model* : strong bias – low variance

The Bias-Variance Decomposition (Bishop PRML 2006)

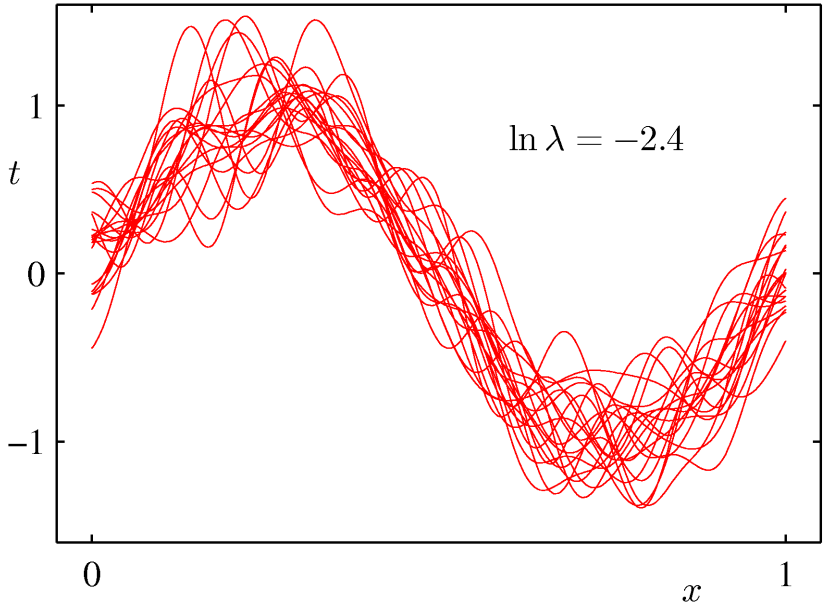
- Example: 100 data sets from the sinusoidal, varying the degree of regularization
  - Model: gaussian basis function, Learning set size = 25, λ is the regularization parameter
    - High values of λ correspond to simple models, low values to more complex models
  - Left 20 of the 100 models shown
  - Right : average of the 100 models (red), true sinusoid (green)
  - Figure illustrates high bias and low variance ($\lambda = 13$)

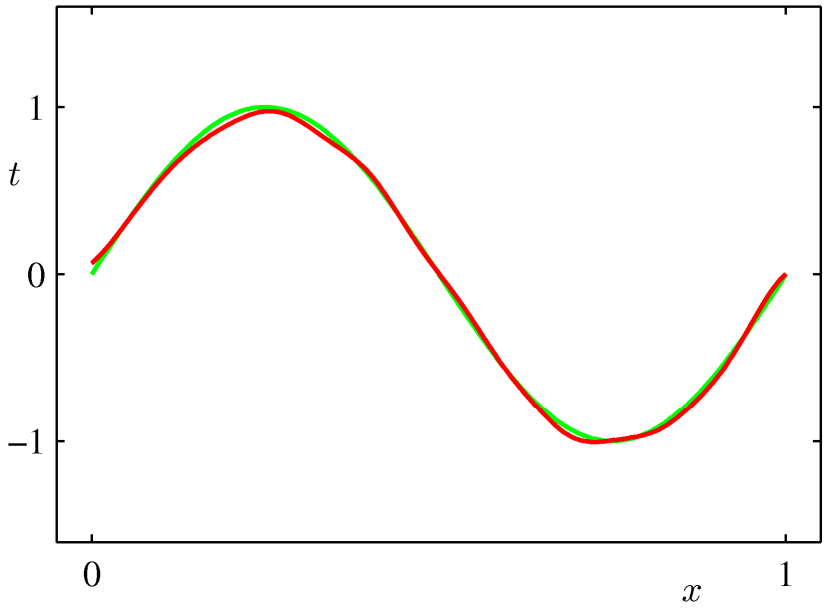




The Bias-Variance Decomposition (Bishop PRML 2006)

- Example: 100 data sets from the sinusoidal, varying the degree of regularization
  - Same setting as before
    - Figure illustrates low bias and high variance ($\lambda = 0.09$)

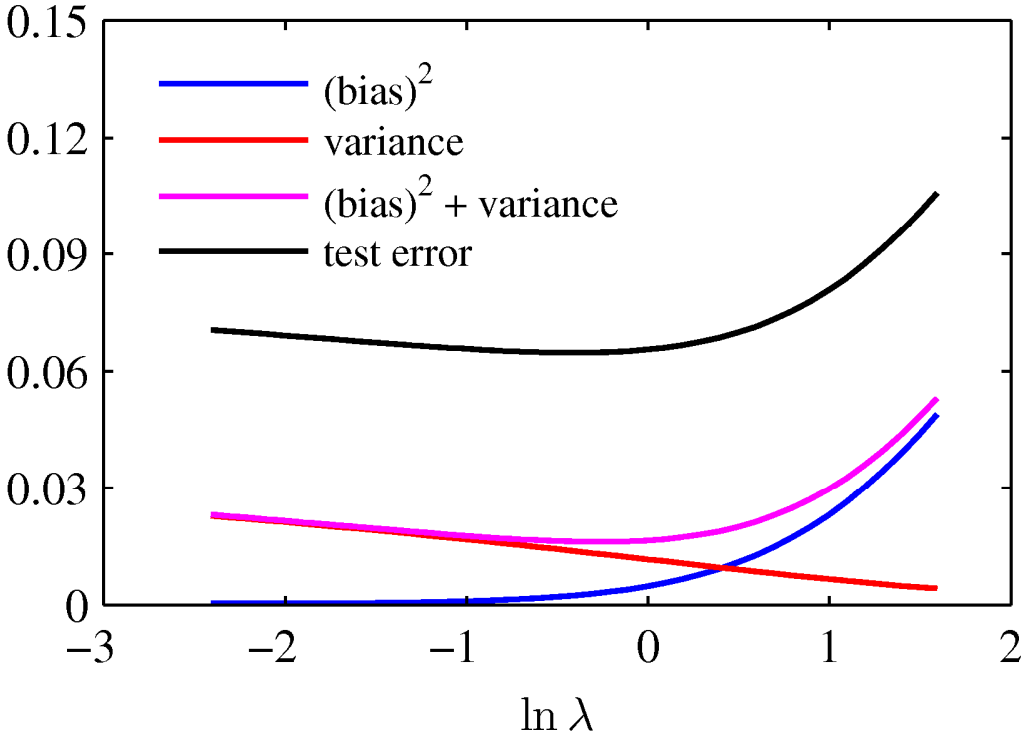




  - Remark
    - The mean of several complex models behaves well here (reduced variance)
      - $\rightarrow$ leads to ensemble methods

The Bias-Variance Decomposition (Bishop PRML 2006)

- From these plots, we note that an over-regularized model (large λ) will have a high bias, while an under-regularized model (small λ) will have a high variance.



Regularisation

- Principle: control the solution variance by constraining function F
  - Optimise $C = C_1 + \lambda C_2$
  - C is a compromise between
    - C_1 : reflects the objective e.g. MSE, Entropie, ...
    - C_2 : constraints on the solution (e.g. weight distribution)
  - λ : constraint weight
- Regularized mean squares
  - For the linear multivariate regression
  - $C = \frac{1}{N}\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.x^i\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\left|w_j\right|^q$
    - $q = 2$ regularization L_2, $q = 1$ regularization L_1 also known as « Lasso »

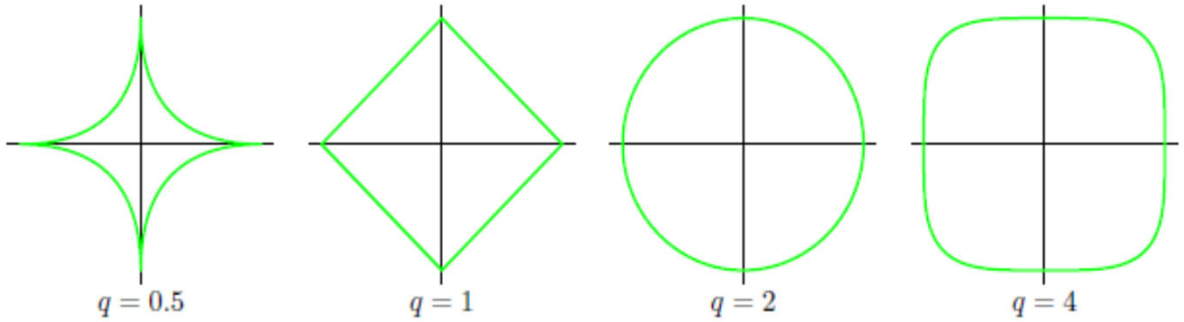


Fig. from Bishop 2006

Figure 3.3 Contours of the regularization term in (3.29) for various values of the parameter q.

Régularisation

- Solve
  - $Min_{\boldsymbol{w}}\, C = \frac{1}{N}\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.x^i\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\left|w_j\right|^q, \lambda > 0$
- Amounts at solving the following constrained optimization problem
  - $Min_{\boldsymbol{w}}\, C = \frac{1}{N}\sum_{i=1}^{N}\left(y^i - \boldsymbol{w}.x^i\right)^2$
  - Under constraint $\sum_{j=1}^{n}\left|w_j\right|^q \leq s$ for a given value of s
- Effect of this constraint

Figure 3.4 Plot of the contours of the unregularized error function (blue) along with the constraint region (3.30) for the quadratic regularizer $q = 2$ on the left and the lasso regularizer $q = 1$ on the right, in which the optimum value for the parameter vector $\mathbf{w}$ is denoted by $\mathbf{w}^*$. The lasso gives a sparse solution in which $w_1^* = 0$.

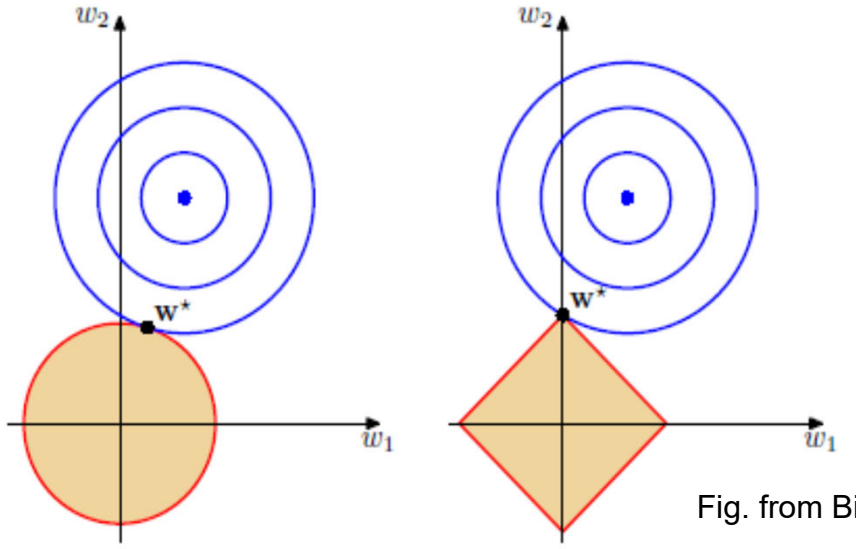


Fig. from Bishop 2006

Regularization

- Penalization L_2
  - Loss
    - $C = C_1 + \lambda \sum_{j=1}^{n} |w_j|^2$
  - Gradiant
    - $\nabla_{\boldsymbol{w}} C = \lambda \boldsymbol{w} + \nabla_{\boldsymbol{w}} C_1$
  - Update
    - $\boldsymbol{w} = \boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} C = (1 - \epsilon\lambda)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} C_1$
    - Penalization is proportional to $\boldsymbol{w}$
- Penalization L_1
  - Loss
    - $C = C_1 + \lambda \sum_{j=1}^{n} |w_j|^1$
  - Gradiant
    - $\nabla_{\boldsymbol{w}} C = \lambda sign(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} C_1$
    - $sign(\boldsymbol{w})$ is the sign of $\boldsymbol{w}$ applied to each component of $\boldsymbol{w}$
  - Update
    - $\boldsymbol{w} = \boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} C = \boldsymbol{w} - \epsilon\lambda sign(\boldsymbol{w}) - \epsilon \nabla_{\boldsymbol{w}} C_1$
    - Penalization is constant with sign $sign(\boldsymbol{w})$

Other ideas for improving generalization in NNs

- Several heuristics have been developed in order to force inductive biases for NNs – some
  - Gradient descent and stochastic gradient descent perform implicit regularization
  - Weights initialization
  - Early stopping
  - Data augmentation
    - By adding noise
      - with early work from Matsuoka 1992 ; Grandvallet and Canu 1994 ; Bishop 1994
      - and many new developments for Deep learning models
    - By generating new examples (synthetic, or any other way)
  - Note: Bayesian learning and regularization
    - Regularization parameters correspond to priors on these model variables
  - Ensembling
    - Model averaging
      - Average models outputs: reduces the variance
    - Functional ensembling (recently developed)
      - Average the network weights on the training trajectory
        - As for 2022: SOTA in classification (e.g. vision tasks)

Generalization in modern Deep Learning

- Deep Learning models often do not follow the common complexity / performance wisdom
  - Extremely large models / with no complexity control (like e.g. regularization or early stopping), may reach good performance, better than models trained with the usual complexity control ingredients
  - Observed in modern deep learning
    - High complexity models with zero train error may not overfit and lead to accurate predictions on unseen data
      - This observation questions the usual claim and the theoretical beliefs such as Bias – Variance dilemma
- Example
  - Double descent phenomenon
    - Based on (Belkin 2019) and (Nakkiran 2020)

Generalization in modern Deep Learning - Double Descent

- Observed by different authors but formalized as a general concept in (Belkin 2019)
- General message
  - Learning curves as a function of model capacity (complexity) exhibit a two regimes phenomenon coined as « double descent »
  - Classical regime corresponds to under-parameterized models and exhibits the classical U shaped curve corresponding to the bias-variance intruition
    - Models do not achieve perfect interpolation
    - The test risk first decreases and then increases when the model starts interpolating
  - Modern interpolation regime corresponds to over-parameterized models
    - Models may achieve near zero train error, i.e. near perfect interpolation
    - Test risk value may decrease below the level of the best classical regime risk value

Generalization in modern Deep Learning - Double Descent Intuition (Belkin 2019)

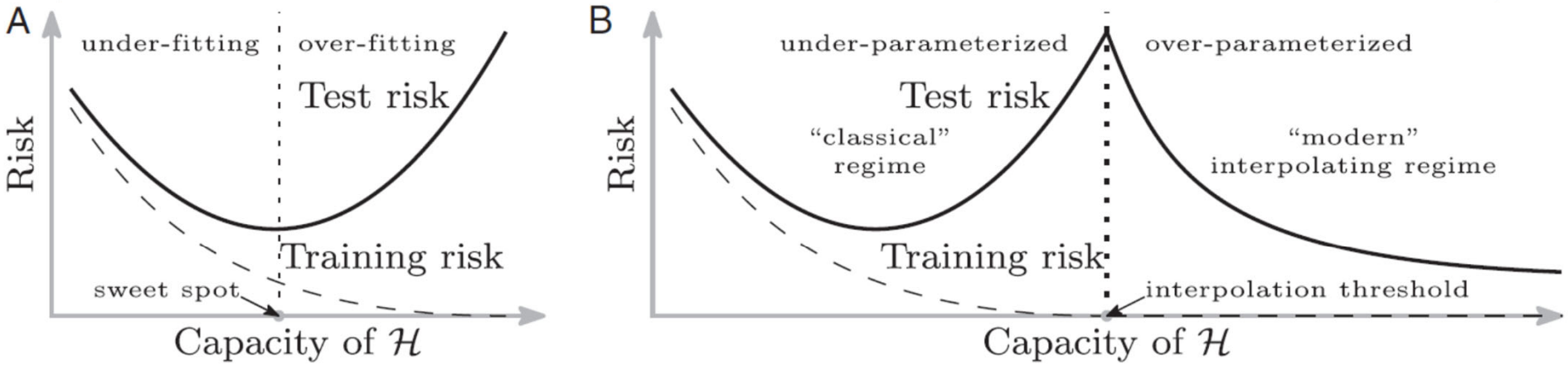


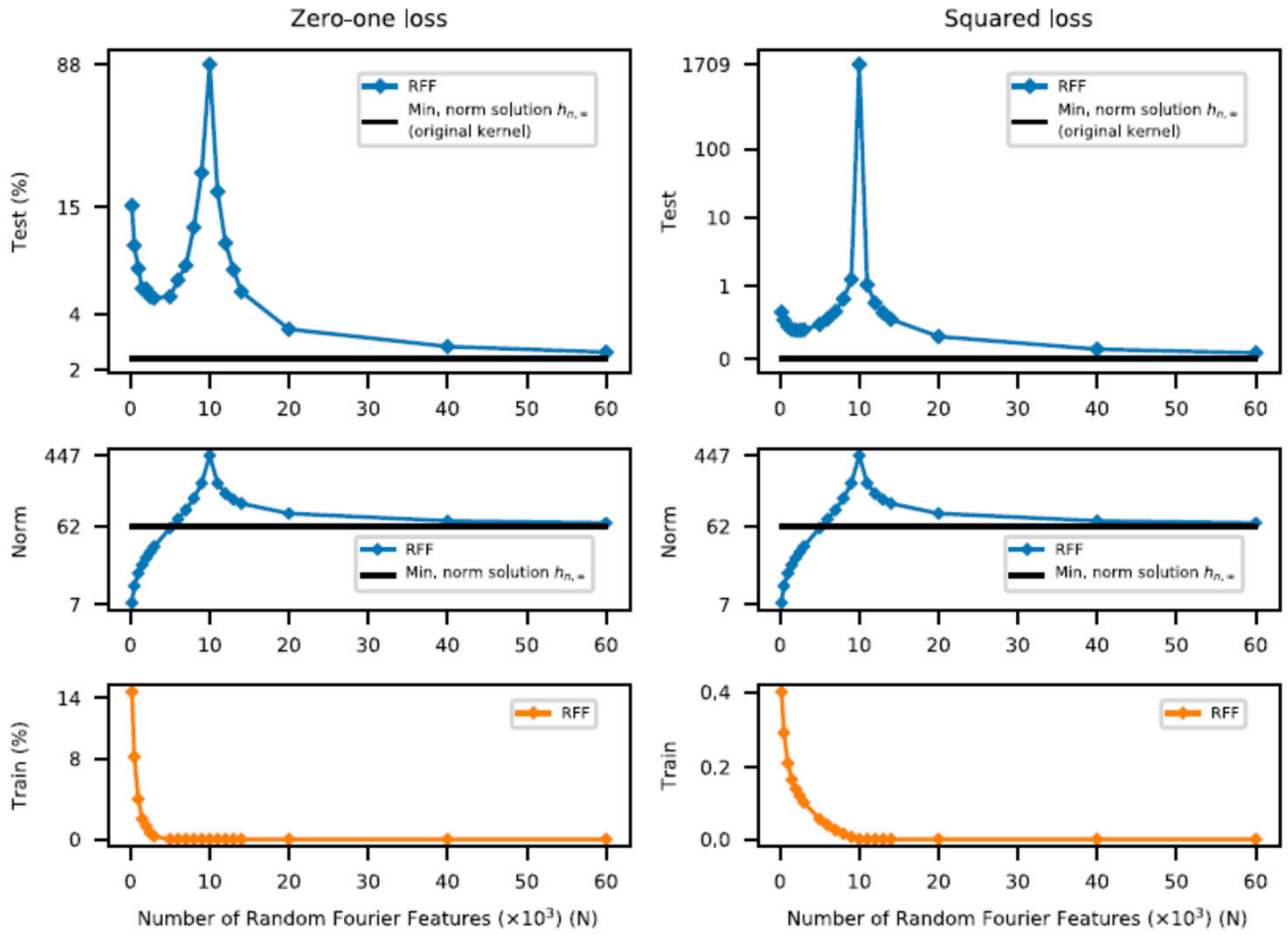
Fig. 1. Curves for training risk (dashed line) and test risk (solid line). (*A*) The classical U-shaped risk curve arising from the bias–variance trade-off. (*B*) The double-descent risk curve, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high-capacity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk.

- All the models to the right of the interpolation threshold have a zero training error
- Tentative explanation
  - The notion of « capacity of the function class » does not fit the inductive bias appropriate for the problem and cannot explain the observed behavior
  - The inductive bias seems to be the smoothness of a function as measured by a certain function space norm

Generalization in modern Deep Learning - Double Descent Intuition (Belkin 2019)

- Caracterization on classification problems
  - Model: Random Fourier Features
  - Equivalent to 1 hidden layer NN with fixed weights in the first layer
    - i.e. only the last weight layers are learned, i.e. convex problem
    - Because of the linearity of the trainable component, the complexity can be measured by the number of basis functions (nb of hidden cells)
      - Or at least this provides a proxy for the complexity
- Random Fourier Features
  - Consider a class of function denoted $\mathcal{H}_N : h(x): R^d \rightarrow R$
    - With $h(x) = \sum_{k=1}^{N} a_k \phi(x; v_k)$ with $\phi(x; v) = \exp(i < v, x >)$ - (the complex exponential)
    - Where the $v_1, \dots, v_N$ are sampled independently from the standard normal distribution in R^d
    - The $\phi(x; v)$ are N complex basis functions
    - This may be implemented as a NN with $2N$ basis functions corresponding to the real and imaginary parts of ϕ
  - Learning procedure
    - Given a training set $(x^1, y^1) \dots (x^n, y^n)$, train via ERM, i.e. minimize $\frac{1}{n}\sum_{i=1}^{n}(h(x^i) - y^i)^2$
    - When the minimizer is not unique (always the case when $N > n$) choose the one with coefficients $(a_1, \dots, a_N)$ of minimum l_2 norm, i.e. the smoothest one

Generalization in modern Deep Learning - Double Descent Intuition (Belkin 2019)



ig. 2. Double-descent risk curve for the RFF model on MNIST. Shown are test risks (log scale), coefficient ℓ_2 norms (log scale), and training risks of the RFF nodel predictors $h_{n,N}$ learned on a subset of MNIST ($n = 10^4$, 10 classes). The interpolation threshold is achieved at $N = 10^4$.

Generalization in modern Deep Learning - Double Descent Intuition (Nakkiran 2020)

- Characterize the double descent phenomenon for
  - A large variety of NN models: CNN, ResNet, Transformers
  - Several settings: model-wise, epoch-wise, sample-wise (defined later)
- Propose a measure of complexity called « effective model complexity »
  - For non linear models, the number of parameters is not a characterization of the function class complexity

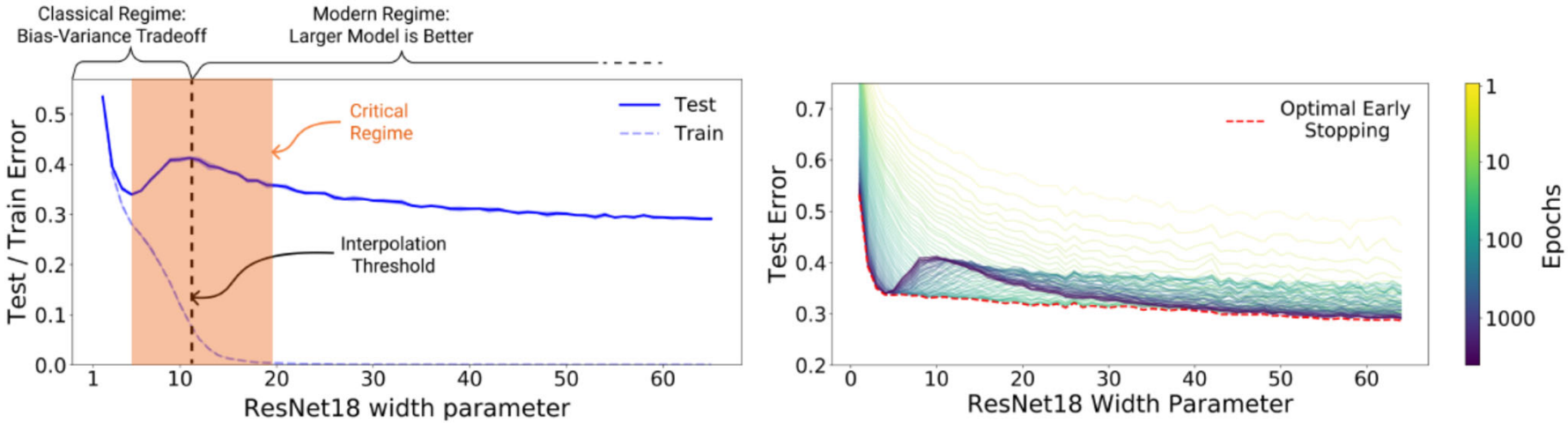


Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

Generalization in modern Deep Learning - Double Descent Intuition (Nakkiran 2020)

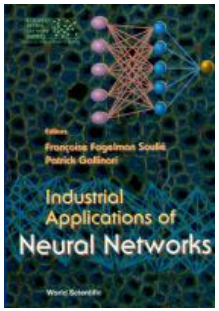
- Effective model complexity (EMC)
  - A training procedure $\mathcal{T}$ takes as input a training set $D = \{(x^1, y^1), ..., (x^n, y^n)\}$ and outputs a classifier $\mathcal{T}(D)$
  - The effective complexity of $\mathcal{T}$ w.r.t. the distribution $\mathcal{D}$ of D is the maximum number of samples n on which $\mathcal{T}$ achieves on average a zero training error
- The EMC of training procedure $\mathcal{T}$ w.r.t. distribution $\mathcal{D}$ and parameter`$\epsilon > 0$, is defined as:
  - $EMC_{\mathcal{D},\epsilon}(\mathcal{T}) = \max\{n | E_{D\sim\mathcal{D}^n}[Error_D(\mathcal{T}(D))] \leq \epsilon$
- Regimes
  - Under-parameterized: $EMC_{\mathcal{D},\epsilon}(\mathcal{T})$ smaller than n, increasing EMC will decrease the test error
  - Over-parameterized: $EMC_{\mathcal{D},\epsilon}(\mathcal{T})$ larger than n, increasing EMC will decrease the test error
  - Critical: $EMC_{\mathcal{D},\epsilon}(\mathcal{T})$ around n, increasing EMC may decrease or increase the test error (see figure)

Generalization in modern Deep Learning - Double Descent Intuition (Nakkiran 2020)

- Different settings for characterizing the double-descent phenomenon
  - i.e. the phenomenon appears under each setting and not only under the Model-wise setting characterized by Belkin et al.
  - Model-wise
    - Fixed large number of training steps, models of increasing size,
  - Epoch-wise
    - Fixed large architecture, increase the number of training epochs
  - Sample-wise
    - Fixed model and training procedure, change the number of training samples

Summary

- Non linear machines were widely developed in the 90^{ies}
- Fundations for modern statistical machine learning
- Fundations for statistical learning theory
- Real world applications



- Also during this period
  - Recurrent Neural Networks
    - Extension of back propagation
  - Reinforcement Learning
    - Early work mid 80ies
    - Sutton – Barto Book 1998, including RL + NN

Deep learning

Interlude: new actors – new practices

- GAFA (Google, Apple, Facebook, Amazon) , BAT (Baidu, Tencent, Alibaba), …, Startups, are shaping the data world
- Research
  - Big Tech. actors are leading the research in DL
  - Large research groups
    - Google Brain, Google Deep Mind, Facebook FAIR, Baidu AI lab, Baidu Institute of Deep Learning, etc
  - Standard development platforms, dedicated hardware, etc
  - DL research requires access to ressources
    - sophisticated libraries
    - large computing power e.g. GPU clusters
    - large datasets, …

DeepMind

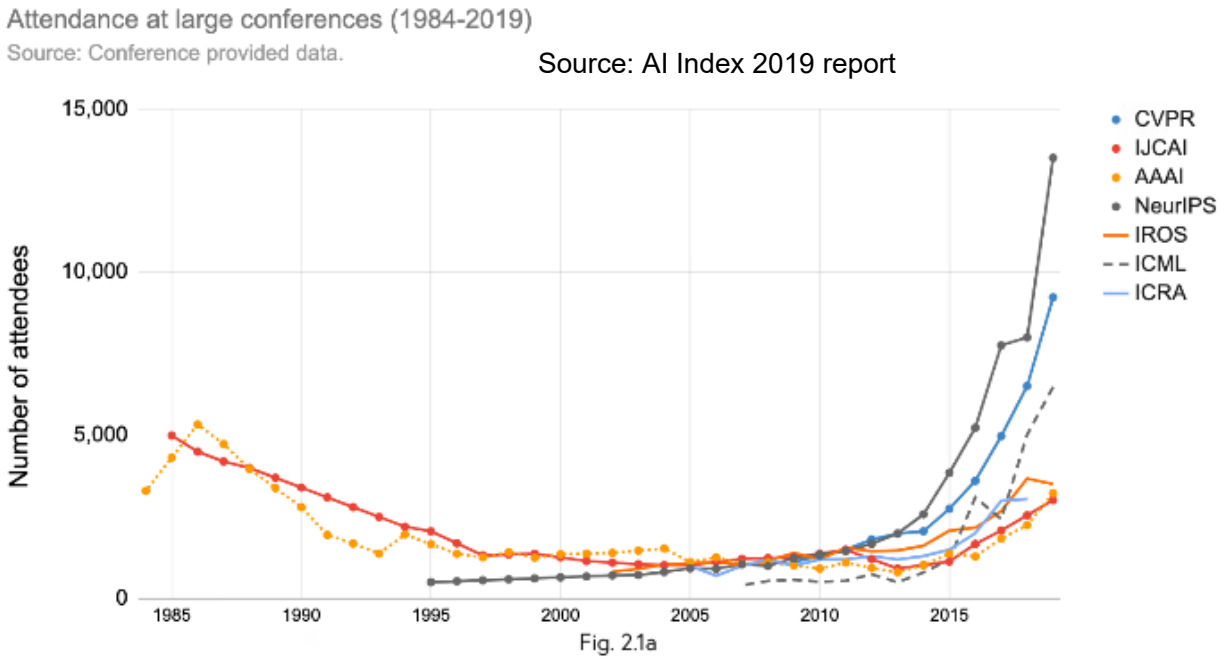
Baidu IDL 百度深度学习实验室 Relentless for Ultimate Intelligence

NVIDIA DEEP LEARNING INSTITUTE

Facebook AI Research

Interlude – ML conference attendance growth

- ML and AI conference Attendence



Source: AI Index 2019 report

Fig. 2.1a

- NIPS (Neurips)
  - 2017 sold out 1 week after registration opening, 7000 participants
  - 2018, 2k inscriptions sold in 11 mn!

Interlude – Deep Learning platforms

- Deep Learning platforms offer
  - Classical DL models
  - Optimization algorithms
  - Automatic differentiation
  - Popular options/ tricks
  - Pretrained models
  - CUDA/ GPU/ CLOUD support
- Contributions by large open source communities: lots of code available
- Easy to build/ train sophisticated models

- Among the most popular platforms:
  - TensorFlow - Google Brain - Python, C/C++
  - PyTorch – Facebook- Python
  - Caffe – UC Berkeley / Caffe2 Facebook, Python, MATLAB
  - Higher level interfaces
    - e.g. Keras for TensorFlow
- And also:
  - PaddlePaddle (Baidu), MXNet (Amazon), Mariana (Tencent), PA 2.0 (Alibaba), …..

Interlude - Modular programming: Keras simple example MLP From https://keras.io/

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# Load and format training and test data
# Not shown - (x_train, y_train), (x_test, y_test)

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Load Training – Test data

Specify NN architecture:
- here basic MLP with 3 weight layers

Optimisation algorithm
- SGD

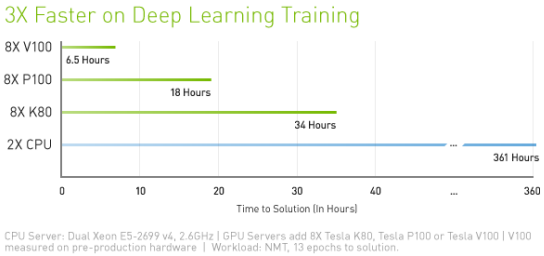
Loss criterion
- Cross entropy

Train for 20 epochs

Evaluate performance on test set

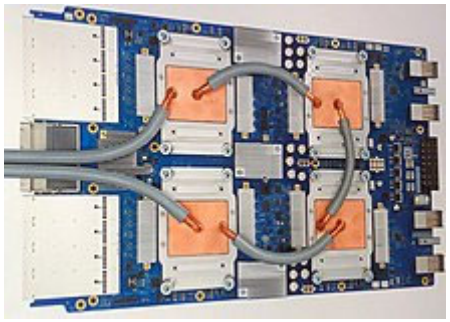
Interlude – Hardware

- 2017 - NVIDIA V100 – optimized for Deep Learning

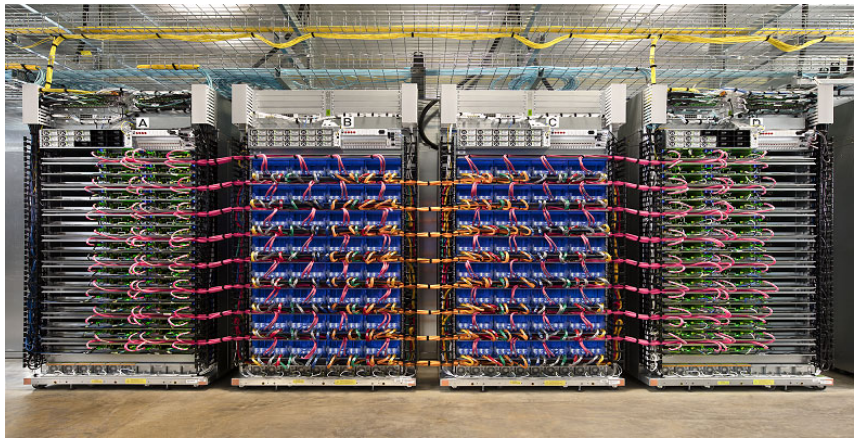


- "With 640 Tensor Cores, Tesla V100 is the world's first GPU to break the 100 teraflops (TFLOPS) barrier of deep learning performance. The next generation of NVIDIA NVLink™ connects multiple V100 GPUs at up to 300 GB/s to create the world's most powerful computing servers."

- Google Tensor Processor Unit – TPU V3



- Cloud TPU



Motivations

- Learning representations
  - Handcrafted versus learned representation
    - Often complex to define what are good representations
  - General methods that can be used for
    - Different application domains
    - Multimodal data
    - Multi-task learning
  - Learning the latent factors behind the data generation
  - Unsupervised feature learning
    - Useful for learning data/ signal representations
- Deep Neural networks
  - Learn high level/ abstract representations from raw data
    - Key idea: stack layers of neurons to build deep architectures
    - Find a way to train them

Motivations and historical folklore
High Level Representations in Videos – Google (Le et al. 2012)

- Objective
  - Learn high level representations without teacher
    - 10 millions images 200x200 from YouTube videos
    - Auto-encoder 10^9 connexions
- « High level » detectors
  - Show test images to the network
    - E.g. faces
  - Look for neurons with maximum response
- Some neurons respond to high level characteristics
  - Faces, cats, silhouettes, …

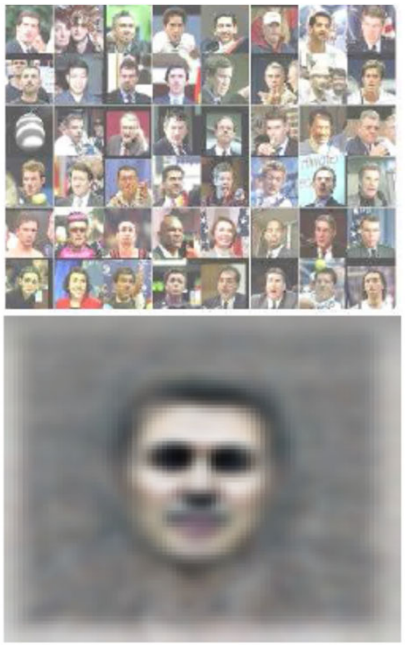


Figure 3. Top: Top 48 stimuli of the best neuron from the test set. Bottom: The optimal stimulus according to numerical constraint optimization.

Top: most responsive stimuli on the test set for uron. Bottom: Most responsive human body the test set for the human body neuron.

Useful Deep Learning heuristics

Deep NN make use of several (essential) heuristics for training large architecture: type of units, normalization, optimization…

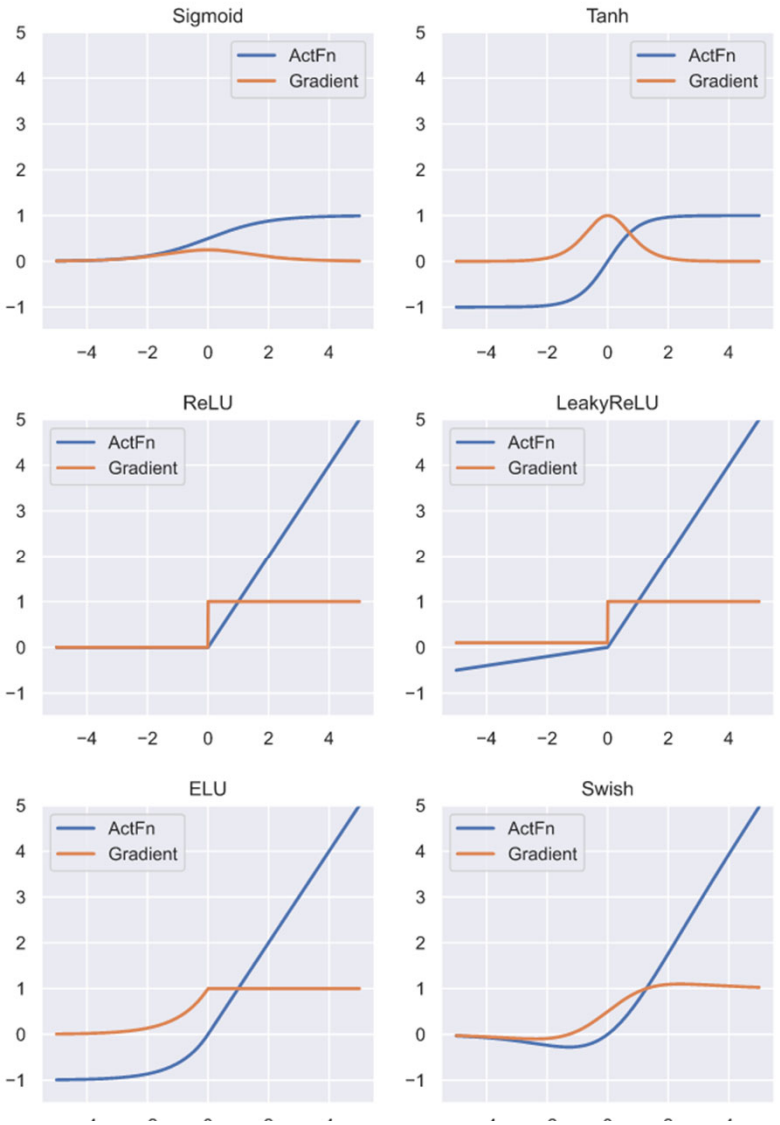
We introduce some of these ideas

Deep Learning heuristics -Activation functions

Figures from:
https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial3/Activation_Functions.html

- In addition to the logistic or tanh units,, other forms are used in deep architectures – Some of the popular forms are:
  - Let $z = b + \boldsymbol{w}.\boldsymbol{x}$
  - RELU - Rectified linear units (used for internal layers)
    - $g(\boldsymbol{z}) = \max(0, \mathbf{z})$
      - Rectified units allow to draw activations to 0 (used for sparse representations) + derivative remain large when unit is active
  - Leaky RELU (used for internal layers)
    - $g(\boldsymbol{z}) = \begin{cases} \mathbf{z} \text{ if} b + \boldsymbol{w}.\boldsymbol{x} > 0 \\ 0.01(\boldsymbol{z}) \text{ otherwise} \end{cases}$
      - Introduces a small derivative when $b + \boldsymbol{w}.\boldsymbol{x} < \mathbf{0}$
  - ELU (used for internal layers)
    - $g(\boldsymbol{z}) = \begin{cases} \mathbf{z} \text{ if } \mathbf{z} > 0 \\ \alpha(\exp(b + \boldsymbol{w}.\boldsymbol{x}) - 1) \text{ otherwise} \end{cases}$
  - Swish
    - $g(\boldsymbol{z}) = \frac{\boldsymbol{z}}{1+\exp(-\boldsymbol{z})}$

x axis $b + \boldsymbol{w}.\boldsymbol{x}$, y axis $g(\boldsymbol{x})$

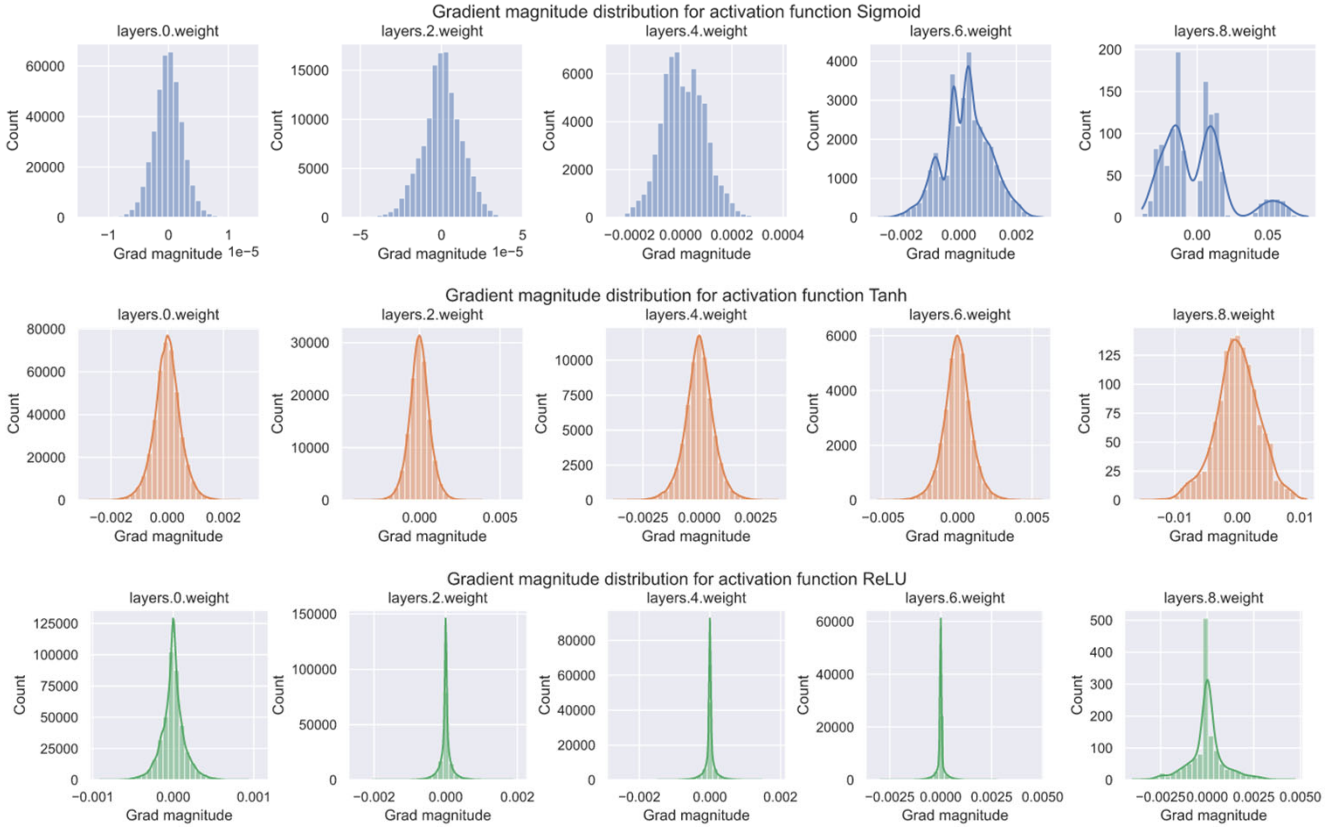


Deep Learning heuristics -Activation functions

Figures from:

https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial3/Activation_Functions.html

- Visualisation of the gradient at different layers of a NN after initialisation of the weights
- Dataset : FashionMNIST (images) 10 classes, gradient computed on a batch of 256 images



Deep Learning heuristics - Activation functions

- In addition to the logistic or tanh units, other forms are used in deep architectures – Some of the popular forms are:
  - Maxout
    - $g(\boldsymbol{x}) = \max_i (b_i + \boldsymbol{w}_i . \boldsymbol{x})$
    - Generalizes the rectified unit
    - There are multiple weight vectors for each unit
  - Softmax (used for output layer)
    - Used for classification with a 1 out of p coding (p classes)
      - Ensures that the sum of predicted outputs sums to 1
      - $g(\boldsymbol{x}) = softmax(\boldsymbol{b} + W\boldsymbol{x}) = \frac{e^{b_i + (Wx)_i}}{\sum_{j=1}^{p} e^{b_j + (Wx)_j}}$

Deep Learning heuristics
Normalisation

- Units: Batch Normalization (Ioffe 2015)
  - Normalize the activations of the units (hidden units) so as to coordinate the gradients accross layers
  - Let $B = \{x^1, \dots, x^N\}$ be a mini batch, $h_i(x^j)$ the activation of hidden unit i for input x^j before non linearity
  - Training
    - Set $h'_i(x^j) = \frac{h_i\ (x^j) - \mu_i}{\sigma_i + \epsilon}$ where μ_i is the mean of the activities of hidden unit i on batch B, and σ_i its standard deviation
    - μ_i and σ_i are estimated on batch B, ϵ is a small positive number
    - The output of unit i is then $z_i = \gamma_i h'_i(x^j) + \beta_i$
      - Where γ and β are learned via SGD
  - Testing
    - μ_i and σ_i for test are estimated as a moving average during training, and need not be recomputed on the whole training dataset

Deep Learning heuristics
Normalization

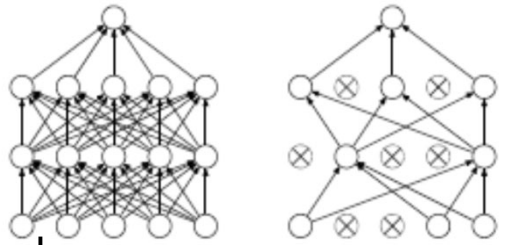
- Note on B.N.
  - No clear agreement if BN should be performed before or after non linearity
  - L^2 normalization could be used together with BN but reduced
  - One of the most effective tricks for learning with deep NNs
  - Other types of normalization have been proposed e.g. Layerwise Normalization similar to BN, but layerwise and datum wise, etc.

- Gradient/ gradient clipping
  - Avoid very large gradient steps when the gradient becomes very large - different strategies work similarly in practice.
  - Let $\nabla_w c$ be the gradient computed over a minibatch
  - A possible clippling strategy is (Pascanu 2013)
    - $\nabla_w c = \frac{\nabla_w c}{||\nabla_w c||} v$, with v a norm threshold

Deep Learning heuristics
Dropout

- Dropout (Srivastava 2014)
  - Training
    - Randomly drop units at training time
      - Parameter: dropout percentage p
      - Each unit is dropped with probability p
        - This means that it is inactive in the forward and backward pass
  - Testing
    - Initial paper (Srivastava 2014)
      - Keep all the units
      - Multiply the units activation by p during test
        - The expected output for a given layer during the test phase should be the same as during the training phase

Figure from Srivastava 2014



(b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Deep Learning heuristics
Dropout

- Inverted Dropout
  - Current implementations use « inverted dropout » - easier implementation: the network does not change during the test phase (see next slide)
    - Units are dropped with probability p
    - Multiplies activations by $\frac{1}{1-p}$ during training, and keep the network untouched during testing
- Effects
  - Increases independence between units and better distributes the representation
  - Interpreted as an ensemble model; reduces model variance

Deep Learning heuristics
Dropout

- Dropout for a single unit
  - Let p be the dropout probability
  - Consider a neuron i with inputs $\boldsymbol{x} \in R^n$ and weight vector $\boldsymbol{w} \in R^n$ including the bias term
  - The activation of neuron i is $z_i = f(\boldsymbol{w}.\boldsymbol{x})$ with f a non linear function (e.g. Relu)
  - Let b_i a binomial variable of parameter $1 - p$
  - Original dropout
    - Training phase
      - $z_i = b_i f(\boldsymbol{w}.\boldsymbol{x}), b_i \in \{0,1\}$
    - Test phase
      - $z_i = \frac{1}{1-p} f(\boldsymbol{w}.\boldsymbol{x})$
  - Inverted dropout
    - Training phase
      - $z_i = \frac{1}{1-p} b_i f(\boldsymbol{w}.\boldsymbol{x}), b_i \in \{0,1\}$
    - Test phase
      - $z_i = f(\boldsymbol{w}.\boldsymbol{x})$
  - Note
    - The total number of neurons dropped at each step is the sum of Bernoullis b_i, it follows a binomial distribution $B(m,p)$ where m is the number of neurons on the layer of neuron i.
    - Its expectation is the $\mathrm{E}[B(m,p)] = mp$
    - L^2 normalization could be used together with dropout but reduced

The loss landscape of deep neural networks

from Li et al. 2018, https://arxiv.org/pdf/1712.09913.pdf

- Developed a method for vizualizing the loss landscape that allows to compare different NNs
- Hints
  - Given θ^* a solution learned by a NN and δ , η two random vectors of the same size as θ^* , plus normalization heuristics on these vectors, plot the surface $f(\alpha, \beta) = L(\theta^* + \alpha\delta + \beta\eta)$
- Examples
  - Networks trained on CIFAR-10 (image dataset for classification)
- Some messages
  - NN depth has a dramatic effect on loss surface when no skip connection is used
  - Wide models tend to have smoother surfaces
  - Landscape geometry has a dramatic effect on generalization. Flat minimizers tend to have lower test errors

The loss landscape of deep neural networks

from Li et al. 2018, https://arxiv.org/pdf/1712.09913.pdf

- 3-D plots
  - ResNet-56 without and with skip connections

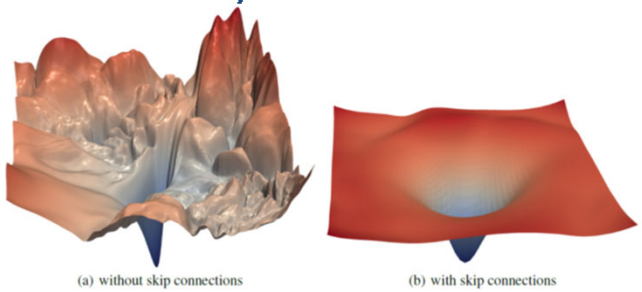


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

- 2-D plots
  - Resnets of different sizes (20, 56, 110 layers) without and with skip connections
    - Centered on the learned min θ^*

Skip connections

No skip connections

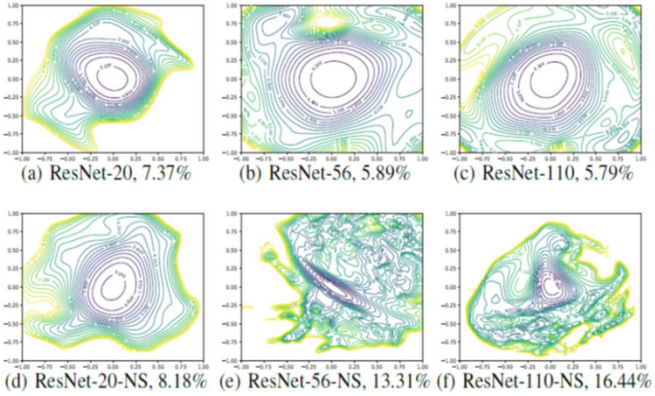


Figure 5: 2D visualization of the loss surface of ResNet and ResNet-noshort with different depth.

Convex landscape for small (20 layers) NNs and for Skip connections

Highly non convex landscape for noSkip NNs when size increases.