Deep Learning

Pontus Giselsson

Learning goals

- Know what a deep neural network (DNN) is
- Know standard deep learning model structures
- Understand why training problem nonconvex
- Understand relation between DNN and convex supervised learning
- Know about different regularization methods in deep learning
- Know that backpropagation can compute gradient for DNN
- Understand backpropagation and that it is based on chain rule
- Be able to implement backpropagation in simple setting

Deep learning

- Can be used both for classification and regression
- Deep learning training problem is of the form

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^{N} L(m(x_i;\theta), y_i)$$

where typically

  - $L(u,y) = \frac{1}{2}\|u-y\|_2^2$ is used for regression
  - $L(u,y) = \log\left(\sum_{j=1}^{K} e^{u_j}\right) - y^T u$ is used for K-class classification

- Difference to previous convex methods: *Nonlinear model* $m(x;\theta)$
  - Deep learning regression generalizes least squares
  - DL classification generalizes multiclass logistic regression
  - Nonlinear model makes training problem nonconvex

Loss function gradient

- Loss functions defined as

$$L(u, y) = \left(\int \sigma(v) dv \right) (u) - y^T u$$

  where
    - $\sigma = I$ for regression (least squares loss)
    - σ is softmax for classification (multiclass logistic regression)
- Formula for gradient in both cases

$$\nabla L(\cdot, y)(u) = \sigma(u) - y$$

Deep learning – Prediction

- Least squares and multiclass logistic losses derived to satisfy

$$\sigma(m(x;\theta)) - y \approx 0$$

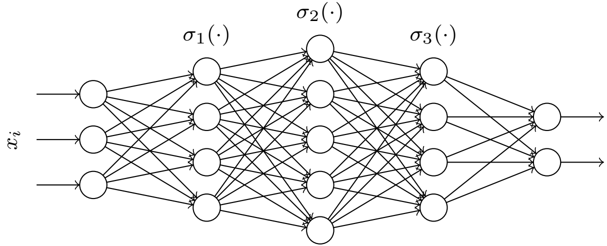
  (gradient equals zero) after training, where
    - $\sigma = I$ for regression
    - $\sigma : \mathbb{R}^K \to \Delta_K$ is *softmax* for multiclass classification
    - This derivation is independent of model structure
- Predict y for new data x same way as for convex methods
    - Regression: $m(x;\theta)$ is the prediction for y
    - Classification: $\sigma(m(x;\theta))$ outputs probabilities for class belonging, predict x in class with largest probability

Deep learning – Model

- Nonlinear model of the following form is often used:

$$m(x;\theta) := W_n\sigma_{n-1}(W_{n-1}\sigma_{n-2}(\cdots(W_2\sigma_1(W_1x+b_1)+b_2)\cdots)+b_{n-1})+b_n,$$

- The σ_j are nonlinear and called activation functions
- Composition of nonlinear (σ_j) and affine ($W_j(\cdot)+b_j$) operations
- Each σ_j function constitutes a hidden layer in the model network
- Graphical representation with three hidden layers

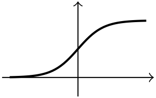
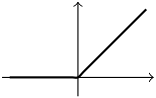
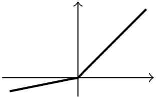
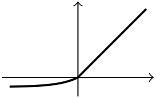
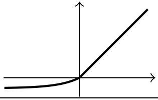


- Why this structure? (Assumed) universal function approximators

Activation functions

- Activation function σ_j takes as input the output of $W_j(\cdot) + b_j$
- Often a function $\bar{\sigma}_j : \mathbb{R} \to \mathbb{R}$ is applied to each element
  - Example: $\sigma_j : \mathbb{R}^3 \to \mathbb{R}^3$ is $\sigma_j(u) = \begin{bmatrix} \bar{\sigma}_j(u_1) \\ \bar{\sigma}_j(u_2) \\ \bar{\sigma}_j(u_3) \end{bmatrix}$
- We will use notation over-loading and call both functions σ_j

Examples of activation functions

Name	$\sigma(u)$	Graph
Sigmoid	$\frac{1}{1+e^{-u}}$	
ReLU	$\max(u, 0)$	
LeakyReLU	$\max(u, \alpha u)$	
ELU	$\begin{cases} u & \text{if } u \geq 0 \\ \alpha(e^u - 1) & \text{else} \end{cases}$	
SELU	$\lambda \begin{cases} u & \text{if } u \geq 0 \\ \alpha(e^u - 1) & \text{else} \end{cases}$	

Examples of affine transformations

- Dense (fully connected): Dense W_j
- Sparse: Sparse W_j
  - Convolutional layer (convolution with small pictures)
  - Fixed (random) sparsity pattern
- Subsampling: reduce size, W_j fat (smaller output than input)
  - max pooling
  - average pooling
  - 2-norm pooling

Learning features

- Used *prespecified* feature maps (or Kernels) in convex methods
- Deep learning instead *learns* feature map during training
  - Define parameter (weight) dependent feature vector:

    $$\phi(x;\theta) := \sigma_{n-1}(W_{n-1}\sigma_{n-2}(\cdots(W_2\sigma_1(W_1x+b_1)+b_2)\cdots)+b_{n-1})$$

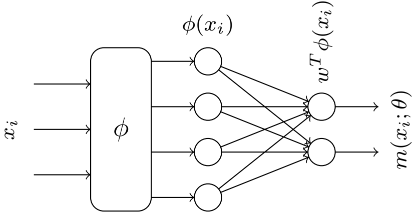
  - Model becomes $m(x;\theta) = W_n\phi(x;\theta) + b_n$
  - Inserted into training problem:

    $$\underset{\theta}{\text{minimize}} \sum_{i=1}^{N} L(W_n\phi(x_i;\theta) + b_n, y_i)$$

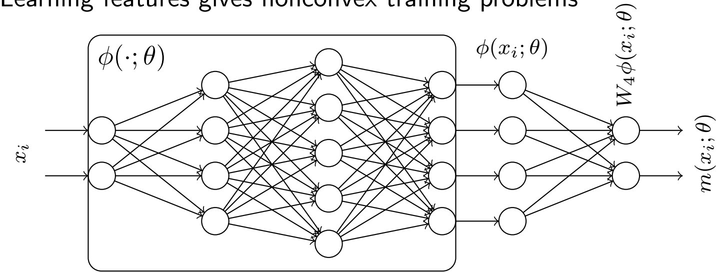
    same as before, but with learned (parameter-dependent) features
- Learning features at training makes training nonconvex

Learning features – Graphical representation

- Fixed features gives convex training problems



- Learning features gives nonconvex training problems



- Output of last activation function is feature vector

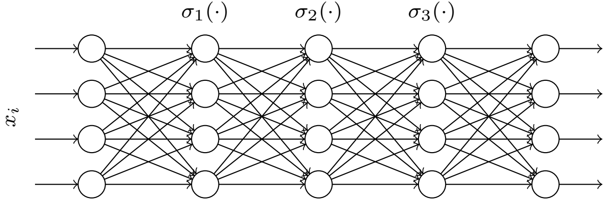
Design choices

Many design choices in building model to create good features

- Number of layers
- Width of layers
- Types of layers
- Types of activation functions
- Use different model structure (e.g., residual network)

Overparameterization

- Assume fully connected network with n layers and N samples
- Assume all layers have p outputs and data $x_i \in \mathbb{R}^p$
- Number of weights $(W_j)_{lk}$: $p^2 n$ and $(b_j)_l$: pn
- Assume $N \approx p^2$ then factor n more weights than samples
- Often overparameterized $\Rightarrow$ can lead to overfitting



Reduce overfitting

Reduce number of weights

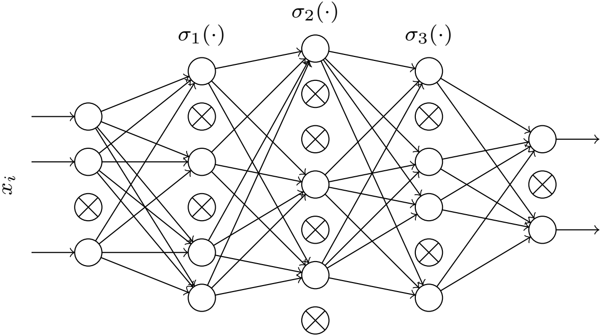
- Sparse weight tensors (e.g., convolutional layers)
- Subsampling (gives fewer weights deeper in network)

Regularization

- Explicit regularization term in cost function, e.g., Tikhonov
- Data augmentation – more samples, artificial often OK
- Early stopping – stop algorithm before convergence
- Dropouts – next slide

Dropouts

- Training problem solved by stochastic gradient method
- Compute gradients on different networks to avoid overfitting
- Take out nodes from network with probability ρ



- Use scaled $\rho\sigma$ in prediction (on average used $\rho\sigma$ in training)

Performance with increasing depth

- Increasing depth can deteriorate performance
- Deep networks may even have worse training errors than shallow
- Intuition: deeper layers bad at approximating identity mapping

Residual networks

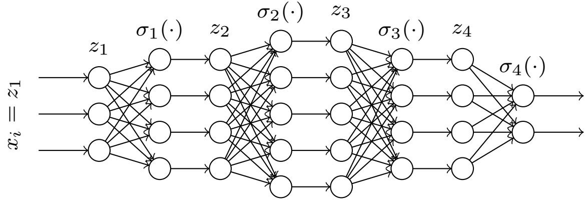
- Add skip connections between layers
- Instead of network architecture with $z_1 = x_i$ (see figure):

$$z_{j+1} = \sigma_j(W_j z_j + b_j) \text{ for } j \in \{1, \ldots, n-1\}$$

use residual architecture

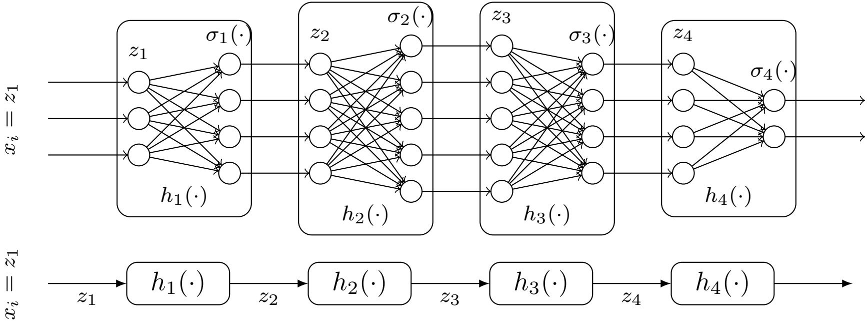
$$z_{j+1} = z_j + \sigma_j(W_j z_j + b_j) \text{ for } j \in \{1, \ldots, n-1\}$$

- Assume $\sigma(0) = 0$, $W_j = 0$, $b_j = 0$ for $j = 1, \ldots, m$ $(m < n-1)$ $\Rightarrow$ deeper part of network is identity mapping and does no harm
- Learns variation from identity mapping (residual)



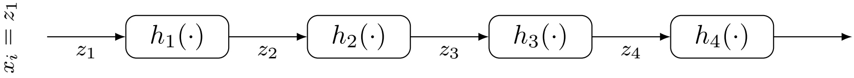
Graphical representation

For graphical representation, first collapse nodes into single node

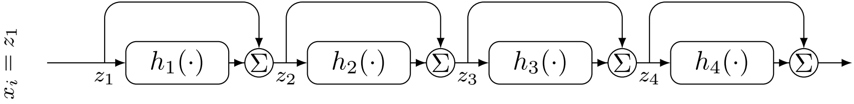


Graphical representation

- Collapsed network representation



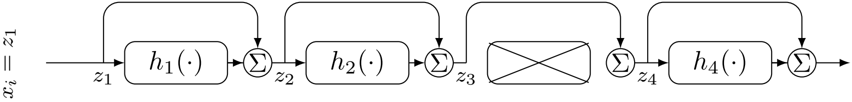
- Residual network



- If some $h_j = 0$ gives same performance as shallower network

Regularization – Layer Dropouts

- Compute gradient on different networks to avoid overfitting
- In residual networks, layers are approximately identity
- We can drop out layers instead of individual neurons
- Drop layer with probability ρ
- Called stochastic depth residual networks



Training algorithm

- Deep neural networks trained using stochastic gradient descent
- DNN weights are updated via gradients in training
- Gradient of cost is sum of gradients of summands (samples)
- Gradient of each summand computed using backpropagation

Backpropagation

- Backpropagation is reverse mode automatic differentiation
- Based on chain-rule
- Backpropagation must be performed per sample
- Our derivation assumes:
  - Fully connected layers (W full, if not, set elements in W to 0)
  - Activation functions $\sigma_j(v) = (\sigma_j(v_1), \ldots, \sigma_j(v_p))$ element-wise (overloading of σ_j notation)
  - Cost $L(u, y) = \left(\int \sigma(v)dv\right)(u) - y^T u$ for some mapping σ
  - Weights W_j are matrices, samples x_i and responses y_i are vectors
  - No residual connections

Preliminaries – Jacobians

- The Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is given by

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

- The Jacobian of a function $f : \mathbb{R}^{p \times n} \to \mathbb{R}$ is given by

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_{11}} & \cdots & \frac{\partial f}{\partial x_{1n}} \\ \vdots & \vdots & \vdots \\ \frac{\partial f}{\partial x_{p1}} & \cdots & \frac{\partial f}{\partial x_{pn}} \end{bmatrix} \in \mathbb{R}^{p \times n}$$

- The Jacobian of a function $f : \mathbb{R}^{p \times n} \to \mathbb{R}^m$ is at layer j given by

$$\left[\frac{\partial f}{\partial x}\right]_{:,j,:} = \begin{bmatrix} \frac{\partial f_1}{\partial x_{j1}} & \cdots & \frac{\partial f_1}{\partial x_{jn}} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_{j1}} & \cdots & \frac{\partial f_m}{\partial x_{jn}} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

the full Jacobian is a 3D tensor in $\mathbb{R}^{m \times p \times n}$

Jacobian vs gradient

- The Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}$ is given by

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}$$

- The gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$ is given by

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

  i.e., transpose of Jacobian for $f : \mathbb{R}^n \to \mathbb{R}$

- Chain rule holds for Jacobians:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z} \frac{\partial z}{\partial x}$$

Jacobian vs gradient – Example

- Consider differentiable $f : \mathbb{R}^m \to \mathbb{R}$ and $L \in \mathbb{R}^{m\times n}$
- Compute Jacobian of $g = (f \circ L)$ using chain rule:
  - Rewrite as $g(x) = f(z)$ where $z = Lx$
  - Compute Jacobian by partial Jacobians $\frac{\partial f}{\partial z}$ and $\frac{\partial z}{\partial x}$:

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial x} = \nabla f(z)^T L = \nabla f(Lx)^T L \in \mathbb{R}^{1\times n}$$

- Know gradient of $(f \circ L)(x)$ satisfies

$$\nabla (f \circ L)(x) = L^T \nabla f(Lx) \in \mathbb{R}^n$$

which is transpose of Jacobian

Backpropagation

- Compute gradient/Jacobian of

$$L(m(x_i;\theta), y_i)$$

w.r.t. $\theta = \{(W_j, b_j\}_{j=1}^{n}$, where

$$m(x_i;\theta) = W_n\sigma_{n-1}(W_{n-1}\sigma_{n-2}(\cdots(W_2\sigma_1(W_1x_i + b_1) + b_2)\cdots) + b_{n-1}) + b_n$$

- Rewrite as function with states z_j

$$\begin{aligned} & L(z_{n+1}, y_i) \\ \text{where} \quad & z_{j+1} = \sigma_j(W_j z_j + b_j) \text{ for } j \in \{1, \ldots, n\} \\ \text{and} \quad & z_1 = x_i \end{aligned}$$

where $\sigma_n(u) \equiv u$

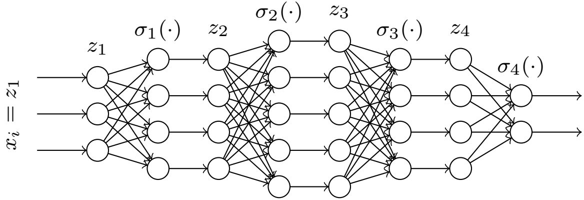
Graphical representation

- Per sample loss function

$$
\begin{aligned}
& L(z_{n+1}, y_i) \\
\text{where} \quad & z_{j+1} = \sigma_j(W_j z_j + b_j) \text{ for } j \in \{1, \ldots, n\} \\
\text{and} \quad & z_1 = x_i
\end{aligned}
$$

  where $\sigma_n(u) \equiv u$

- Graphical representation



Backpropagation

- Jacobian of L w.r.t. W_j and b_j can be computed as

$$\begin{aligned}\frac{\partial L}{\partial W_j} &= \frac{\partial L}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial z_n}\cdots\frac{\partial z_{j+2}}{\partial z_{j+1}}\frac{\partial z_{j+1}}{\partial W_j}\\ \frac{\partial L}{\partial b_j} &= \frac{\partial L}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial z_n}\cdots\frac{\partial z_{j+2}}{\partial z_{j+1}}\frac{\partial z_{j+1}}{\partial b_j}\end{aligned}$$

  where we mean derivative w.r.t. first argument in L

- Backpropagation evaluates partial Jacobians as follows

$$\begin{aligned}\frac{\partial L}{\partial W_j} &= \left(\left(\frac{\partial L}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial z_n}\right)\cdots\frac{\partial z_{j+2}}{\partial z_{j+1}}\right)\frac{\partial z_{j+1}}{\partial W_j}\\ \frac{\partial L}{\partial b_j} &= \left(\left(\frac{\partial L}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial z_n}\right)\cdots\frac{\partial z_{j+2}}{\partial z_{j+1}}\right)\frac{\partial z_{j+1}}{\partial b_j}\end{aligned}$$

Backpropagation

- Jacobian of $L(z_{n+1}, y_i)$ w.r.t. z_{n+1} (transpose of gradient)

$$\frac{\partial L}{\partial z_{n+1}} = (\sigma(z_{n+1}) - y_i)^T$$

i.e., z_{n+1} needed ⇒ forward pass: $z_1 = x_i$, $z_{j+1} = \sigma_j(W_j z_j + b_j)$

- Backward pass, store δ_j:

$$\frac{\partial L}{\partial z_j} = \underbrace{\left(\underbrace{\left(\underbrace{\frac{\partial L}{\partial z_{n+1}}}_{\delta_{n+1}^T} \frac{\partial z_{n+1}}{\partial z_n} \right)}_{\delta_n^T} \cdots \frac{\partial z_{j+2}}{\partial z_{j+1}} \right)}_{\delta_{j+1}^T}$$

- Compute

$$\frac{\partial L}{\partial W_j} = \frac{\partial L}{\partial z_{j+1}} \frac{\partial z_{j+1}}{\partial W_j} = \delta_{j+1} \frac{\partial z_{j+1}}{\partial W_j}$$
$$\frac{\partial L}{\partial b_j} = \frac{\partial L}{\partial z_{j+1}} \frac{\partial z_{j+1}}{\partial b_j} = \delta_{j+1} \frac{\partial z_{j+1}}{\partial b_j}$$

Dimensions

- Let $z_j \in \mathbb{R}^{n_j}$, consequently $W_j \in \mathbb{R}^{n_{j+1} \times n_j}$, $b_j \in \mathbb{R}^{n_j}$
- Dimensions

$$\frac{\partial L}{\partial W_j} = \underbrace{\underbrace{\left(\left(\underbrace{\underbrace{\frac{\partial L}{\partial z_{n+1}}}_{1\times n_{n+1}} \underbrace{\frac{\partial z_{n+1}}{\partial z_n}}_{n_{n+1}\times n_n}}_{1\times n_n} \right) \cdots \underbrace{\frac{\partial z_{j+2}}{\partial z_{j+1}}}_{n_{j+2}\times n_{j+1}} \right)}_{1\times n_{j+1}} \underbrace{\frac{\partial z_{j+1}}{\partial W_j}}_{n_{j+1}\times n_{j+1}\times n_j}}_{n_{j+1}\times n_j}$$

$$\frac{\partial L}{\partial b_j} = \underbrace{\underbrace{\left(\left(\frac{\partial L}{\partial z_{n+1}} \frac{\partial z_{n+1}}{\partial z_n} \right) \cdots \frac{\partial z_{j+2}}{\partial z_{j+1}} \right)}_{1\times n_{j+1}} \underbrace{\frac{\partial z_{j+1}}{\partial b_j}}_{n_{j+1}\times n_j}}_{1\times n_j}$$

- Vector matrix multiplies except for in last step
- Multiplication with tensor $\frac{\partial z_{j+1}}{\partial W_j}$ can be simplified
- Backpropagation variables $\delta_j \in \mathbb{R}^{n_j}$ are vectors (not matrices)

Partial Jacobian $\frac{\partial z_{j+1}}{\partial z_j}$

- Recall relation $z_{j+1} = \sigma_j(W_j z_j + b_j)$ and let $v_j = W_j z_j + b_j$
- Chain rule gives

$$\begin{aligned}\frac{\partial z_{j+1}}{\partial z_j} = \frac{\partial z_{j+1}}{\partial v_j}\frac{\partial v_j}{\partial z_j} &= \mathbf{diag}(\sigma_j'(v_j))\frac{\partial v_j}{\partial z_j} \\ &= \mathbf{diag}(\sigma_j'(W_j z_j + b_j))W_j\end{aligned}$$

  where, with abuse of notation (notation overloading)

$$\sigma_j'(u) = \begin{bmatrix} \sigma_j'(u_1) \\ \vdots \\ \sigma_j'(u_{n_{j+1}}) \end{bmatrix}$$

- Reason: $\sigma_j(u) = [\sigma_j(u_1), \ldots, \sigma_j(u_{n_{j+1}})]^T$ with $\sigma_j : \mathbb{R}^{n_{j+1}} \to \mathbb{R}^{n_{j+1}}$, gives

$$\frac{d\sigma_j}{du} = \begin{bmatrix} \sigma_j'(u_1) & & \\ & \ddots & \\ & & \sigma_j'(u_{n_{j+1}}) \end{bmatrix} = \mathbf{diag}(\sigma_j'(u))$$

Partial Jacobian $\delta_j^T = \frac{\partial L}{\partial z_j}$

- For any vector $\delta_{j+1} \in \mathbb{R}^{n_{j+1} \times 1}$, we have

$$\begin{aligned}\delta_{j+1}^T \frac{\partial z_{j+1}}{\partial z_j} &= \delta_{j+1}^T \, \mathbf{diag}(\sigma_j'(W_j z_j + b_j)) W_j \\ &= (W_j^T (\delta_{j+1}^T \, \mathbf{diag}(\sigma_j'(W_j z_j + b_j)))^T)^T \\ &= (W_j^T (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j)))^T\end{aligned}$$

  where $\odot$ is element-wise (Hadamard) product
- We have defined $\delta_{n+1}^T = \frac{\partial L}{\partial z_{n+1}}$, then

$$\delta_n^T = \frac{\partial L}{\partial z_n} = \delta_{n+1}^T \frac{\partial z_{n+1}}{\partial z_n} = (\underbrace{W_n^T (\delta_{n+1} \odot \sigma_n'(W_n z_n + b_n))}_{\delta_n})^T$$

- Consequently, using induction:

$$\delta_j^T = \frac{\partial L}{\partial z_j} = \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial z_j} = (\underbrace{W_j^T (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))}_{\delta_j})^T$$

Information needed to compute $\frac{\partial L}{\partial z_j}$

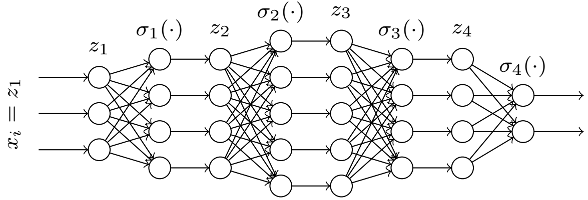
- To compute first Jacobian $\frac{\partial L}{\partial z_n}$, we need $z_n \Rightarrow$ forward pass
- Computing

$$\frac{\partial L}{\partial z_j} = \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial z_j} = (W_j^T(\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j)))^T = \delta_j^T$$

is done using a backward pass

$$\delta_j = W_j^T(\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))$$

- All z_j (or $v_j = W_j z_j + b_j$) need to be stored for backward pass



Partial Jacobian $\frac{\partial L}{\partial W_j}$

- Computed by

$$\frac{\partial L}{\partial W_j} = \frac{\partial L}{\partial z_{j+1}} \frac{\partial z_{j+1}}{\partial W_j} = \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial W_j}$$

  where $z_{j+1} = \sigma_j(v_j)$ and $v_j = W_j z_j + b_j$

- Recall $\frac{\partial z_{j+1}}{\partial W_l}$ is 3D tensor, compute Jacobian w.r.t. row l $(W_j)_l$

$$\delta_{j+1}^T \frac{\partial z_{j+1}}{\partial (W_j)_l} = \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial v_j} \frac{\partial v_j}{\partial (W_j)_j} = \delta_{j+1}^T \,\mathbf{diag}(\sigma_j'(v_j)) \begin{bmatrix} 0 \\ \vdots \\ z_j^T \\ \vdots \\ 0 \end{bmatrix}$$

$$= (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))^T \begin{bmatrix} 0 \\ \vdots \\ z_j^T \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))_j z_j^T \\ \vdots \\ 0 \end{bmatrix}$$

Partial Jacobian $\frac{\partial L}{\partial W_j}$ cont'd

- Stack Jacobians w.r.t. rows to get full Jacobian:

$$\begin{aligned}\frac{\partial L}{\partial W_j} &= \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial W_j} = \begin{bmatrix} \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial (W_j)_1} \\ \vdots \\ \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial (W_j)_{n_{j+1}}} \end{bmatrix} = \begin{bmatrix} (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j)_1) z_j^T \\ \vdots \\ (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j)_{n_{j+1}}) z_j^T \end{bmatrix} \\ &= (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j)) z_j^T\end{aligned}$$

for all $j \in \{1, \ldots, n-1\}$

- Dimension of result is $n_{j+1} \times n_j$, which matches W_j
- This is used to update W_j weights in algorithm

Partial Jacobian $\frac{\partial L}{\partial b_j}$

- Recall $z_{j+1} = \sigma_j(v_j)$ where $v_j = W_j z_j + b_j$
- Computed by

$$\begin{aligned}\frac{\partial L}{\partial b_j} &= \frac{\partial L}{\partial z_{j+1}}\frac{\partial z_{j+1}}{\partial v_j}\frac{\partial v_j}{\partial b_j} = \delta_{j+1}^T \frac{\partial z_{j+1}}{\partial v_j}\frac{\partial v_j}{\partial b_j} = \delta_{j+1}^T\,\mathbf{diag}(\sigma_j'(v_j)) \\ &= (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))^T\end{aligned}$$

Backpropagation summarized

1. Forward pass: Compute and store z_j (or $v_j = W_j z_j + b_j$):

$$z_{j+1} = \sigma_j(W_j z_j + b_j)$$

   where $z_1 = x_i$ and $\sigma_n = \mathrm{Id}$

2. Backward pass:

$$\delta_j = W_j^T(\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))$$

   with $\delta_{n+1} = (\sigma(z_{n+1}) - y_i)^T$

3. Weight update Jacobians (used in SGD)

$$\frac{\partial L}{\partial W_j} = (\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j)) z_j^T$$
$$\frac{\partial L}{\partial b_j} = (\delta_{j+1} \odot \sigma_j'(W_j x_j + b_j))^T$$

Vanishing and exploding gradient problem

- For some activation functions, gradients can vanish
- For other activation functions, gradients can explode

Vanishing gradient example: Sigmoid

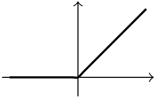
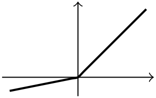
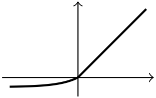
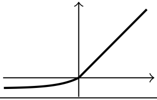
- Assume $\|W_j\| \leq 1$ for all j and $\|\delta_{n+1}\| \leq C$
- Maximal derivative of sigmoid (σ) is 0.25
- Then

$$\begin{aligned}\left\|\frac{\partial L}{\partial z_j}\right\| &= \|\delta_j\| = \|W_j^T(\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))\| \leq 0.25\|\delta_{j+1}\| \\ &\leq 0.25^{n-j+1}\|\delta_{n+1}\| \leq 0.25^{n-j+1}C\end{aligned}$$

- Hence, as n grows, gradients can become very small for small i
- In general, vanishing gradient if $\sigma' < 1$ everywhere
- Similar reasoning: exploding gradient if $\sigma' > 1$ everywhere
- Hence, need $\sigma' = 1$ in large regions

Examples of activation functions

Activation functions that (partly) avoid vanishing gradients

Name	$\sigma(u)$	Graph
ReLU	$\max(u, 0)$	
LeakyReLU	$\max(u, \alpha u)$	
ELU	$\begin{cases} u & \text{if } u \geq 0 \\ \alpha(e^u - 1) & \text{else} \end{cases}$	
SELU	$\lambda \begin{cases} u & \text{if } u \geq 0 \\ \alpha(e^u - 1) & \text{else} \end{cases}$	

Avoiding exploding gradient – Gradient clipping

- "Clip" (constrain) gradients, e.g.,: $\|W_j\|_2 \leq 1$, $|(W_j)_{lk}| \leq c$
- Sometime enforced:
  - within backpropagation (no gradient computed)
  - after backpropagation using projection (projected gradient)
- Using $\|W_j\|_2 \leq 1$, $\|b_j\|_2 \leq d$ and 1-Lipschitz σ_j controls growth:
  - Forward pass (assuming $\sigma_j(0) = 0$): $z_{j+1} = \sigma_j(W_j z_j + b_j)$

$$\begin{aligned}\|z_{j+1}\|_2 = \|\sigma_j(W_j z_j + b_j)\|_2 \leq \|W_j z_j + b_j\|_2 \leq \|W_j z_j\|_2 + \|b_j\|_2 \\ \leq \|z_j\|_2 + d\end{aligned}$$

  - Backward pass: $\delta_j = W_j^T(\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))$

$$\begin{aligned}\|\delta_j\|_2 = \|W_j^T(\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j))\|_2 \leq \|W_j^T\|_2 \|\delta_{j+1} \odot \sigma_j'(W_j z_j + b_j)\| \\ \leq \|\delta_{j+1}\|_2\end{aligned}$$

- Initialize weights from normal distr., scale to have $\|W_j\|_2 = 1$

For large networks

- For large networks $\|W_j\|_2$ may be too expensive to compute
- Approximate, e.g., using (where $W_j \in \mathbb{R}^{n_{j+1} \times n_j}$)
  - Frobenius norm $\|W_j\|_F \leq 1$:

$$\|W_j\|_2 \leq \|W_j\|_F \leq 1$$

  - Element-wise constraints $|(W_j)_{jk}| \leq \frac{1}{\sqrt{n_{j+1}n_j}}$:

$$\|W_j\|_2^2 \leq \|W_j\|_F^2 = \sum_{l,k} (W_j)_{lk}^2 \leq \sum_{l,k} \frac{1}{\sqrt{n_{j+1}n_j}^2} = \frac{n_{j+1}n_j}{\sqrt{n_{j+1}n_j}^2} = 1$$

- Maybe increase upper bounds since $\|W_j\|_2$ upper approximated
- Initialize weights from normal distr., scale according to above
- Many other heuristics exist