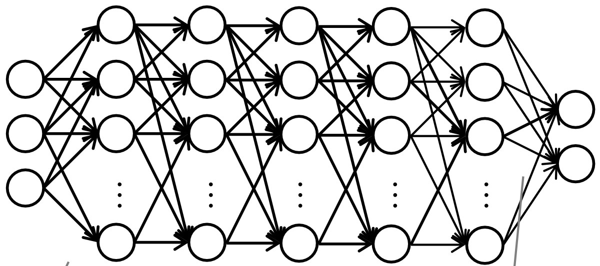
TRAINING OF NEURAL NETWORKS --- BASICS

Outline of training neural networks (recap)

- Training is formulated as a minimization problem

Given a set of I/O pairs: $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{d}_1), \ldots, (\mathbf{x}_N, \mathbf{d}_N)\}$

$\mathbf{x}_n$ ⇨  ⇨ $\mathbf{y}_n \leftrightarrow \mathbf{d}_n$

Weights：

$\mathbf{w} = \{(\mathbf{W}^{(2)}, \mathbf{b}^{(2)}), \cdots, (\mathbf{W}^{(L)}, \mathbf{b}^{(L)})\}$

Loss: $E(\mathbf{w}, \mathcal{S})$

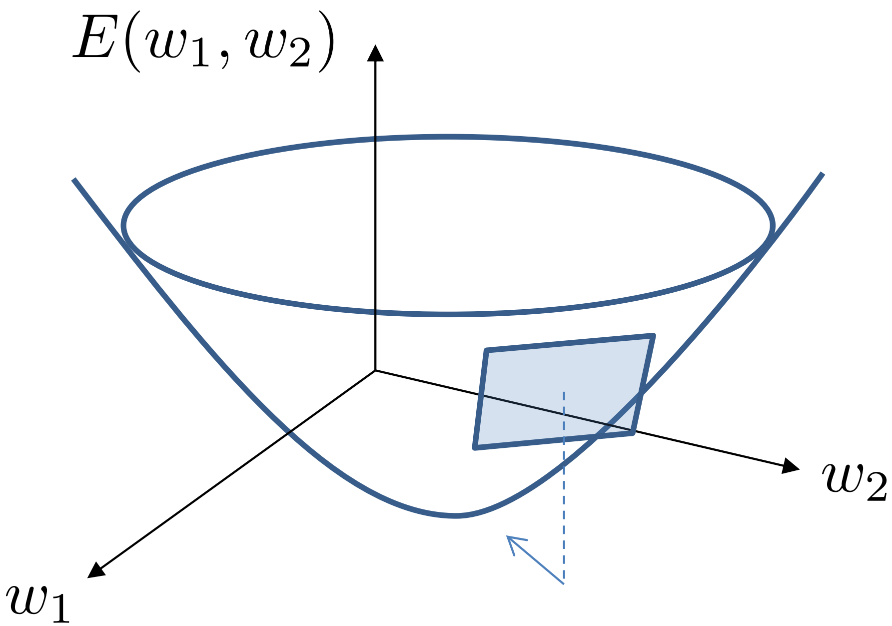
We want to solve $\min_{\mathbf{w}} E(\mathbf{w}, \mathcal{S})$

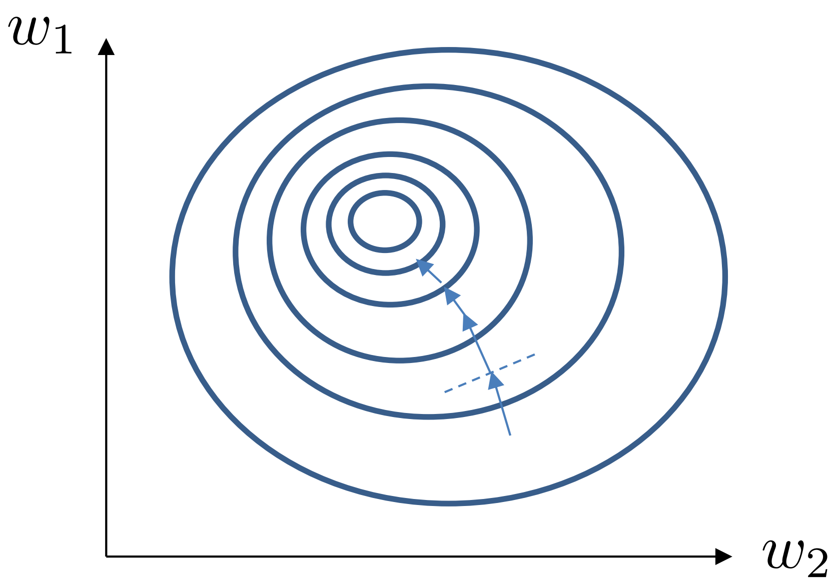
Basic algorithm: gradient descent

- Update the parameter in the direction of gradient of E(w)

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E \qquad \varepsilon : \text{learning rate}$$

$$\nabla E \equiv \frac{dE}{d\mathbf{w}} = \left[\frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_M} \right]^\top$$





Computing gradients is cumbersome

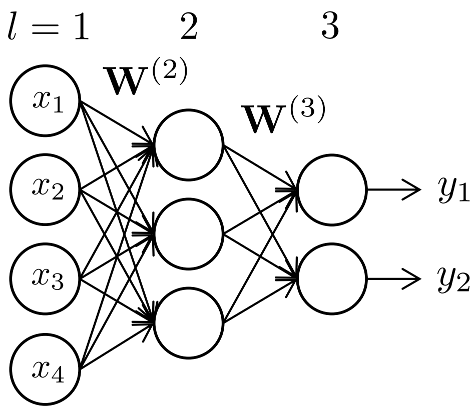
- Because our variables are inside deeply nested functions

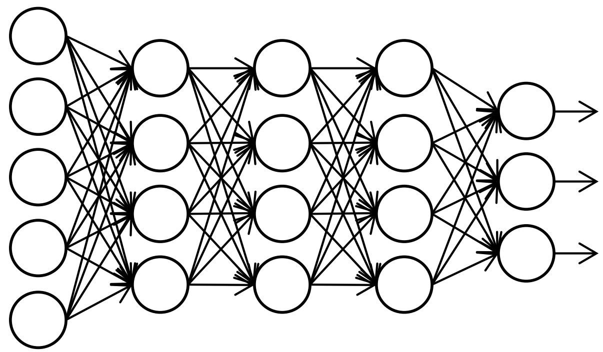
E.g. $$E_n = \frac{1}{2}\|\mathbf{y} - \mathbf{d}\|^2 = \frac{1}{2}\sum_j (y_j - d_j)^2$$

$$\frac{\partial E_n}{\partial w_{ji}^{(l)}} = (\mathbf{y}(\mathbf{x}_n) - \mathbf{d}_n)^\top \frac{\partial \mathbf{y}}{\partial w_{ji}^{(l)}}$$

$$\left(\quad E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \quad \right)$$

$$\begin{aligned}
\mathbf{y}(\mathbf{x}) &= \mathbf{f}(\mathbf{u}^{(L)}) \\
&= \mathbf{f}(\mathbf{W}^{(L)}\mathbf{z}^{(L-1)} + \mathbf{b}^{(L)}) \\
&= \mathbf{f}(\mathbf{W}^{(L)}\mathbf{f}(\mathbf{W}^{(L-1)}\mathbf{z}^{(L-2)} + \mathbf{b}^{(L-1)}) + \mathbf{b}^{L}) \\
&= \mathbf{f}(\mathbf{W}^{(L)}\mathbf{f}(\mathbf{W}^{(L-1)}\mathbf{f}(\cdots\mathbf{f}(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)})\cdots)) + \mathbf{b}^{(L)})
\end{aligned}$$



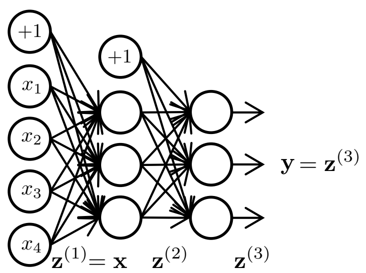


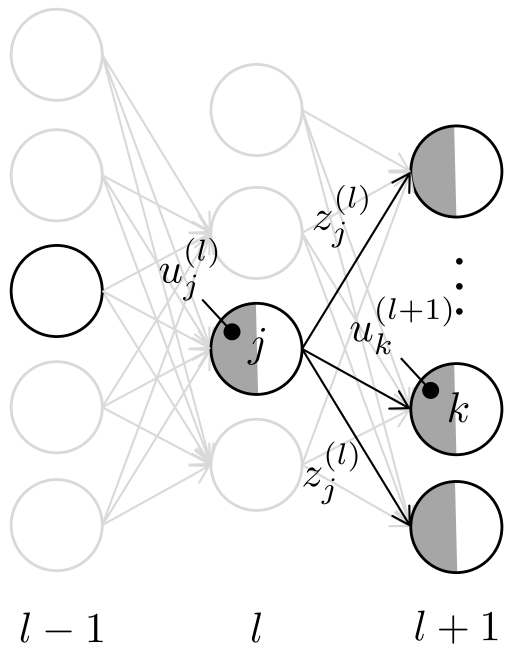
Outline of backpropagation (BP) algorithm

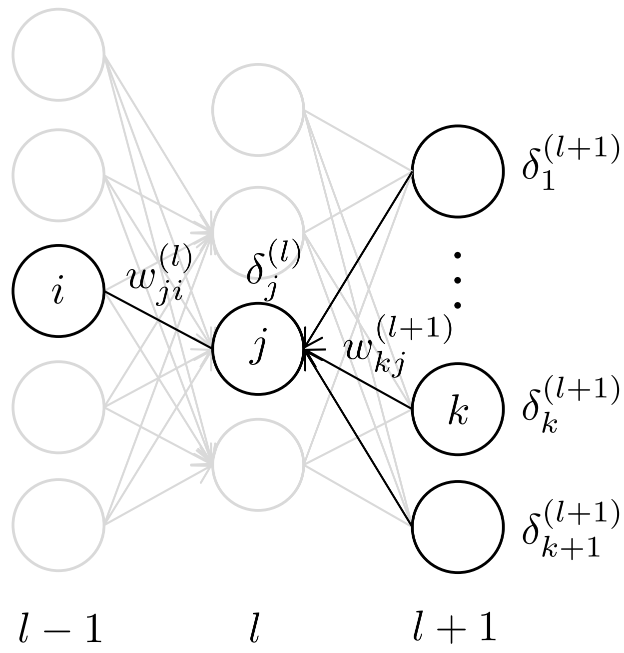
- Using *delta* $\delta_j^{(l)} \equiv \dfrac{\partial E_n}{\partial u_j^{(l)}}$, gradient is given by $\dfrac{\partial E_n}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)}$

- Deltas can be computed by their *backpropagation:*

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} \left(w_{kj}^{(l+1)} f'(u_j^{(l)}) \right)$$



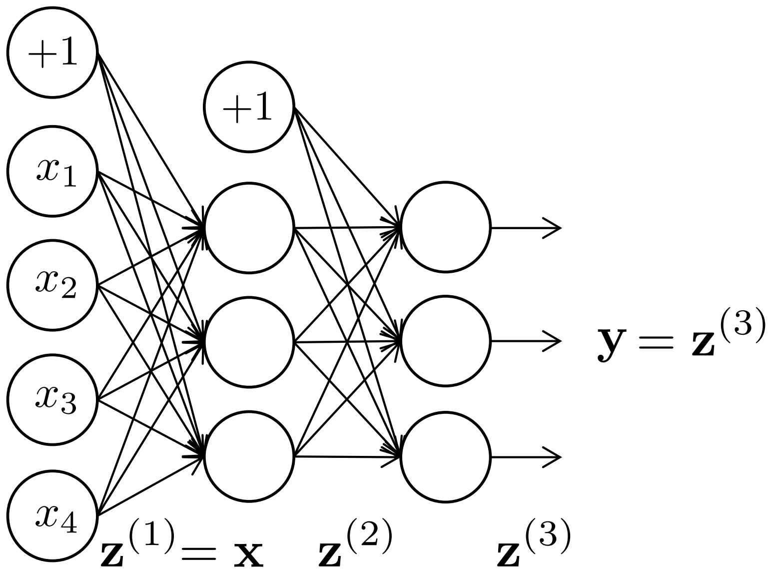




Derivation of BP algorithm: Preparation

- We represent the bias b by a weight w_{j0} from an imaginary unit that always outpus +1 (*A trick making analysis easier*)

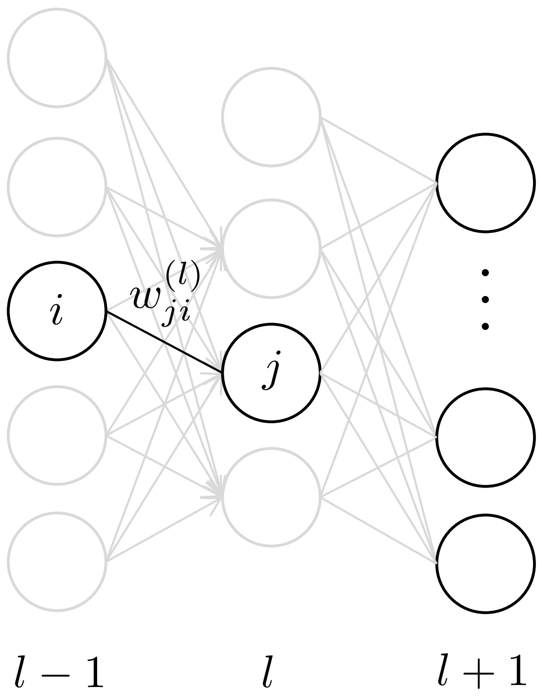
$$u_j^{(l)} = \sum_{i=1}^{n} w_{ji}^{(l)} z_i^{(l-1)} + b_j = \sum_{i=0}^{n} w_{ji}^{(l)} z_i^{(l-1)}$$



Derivation of BP algorithm: Chain rule 1/2

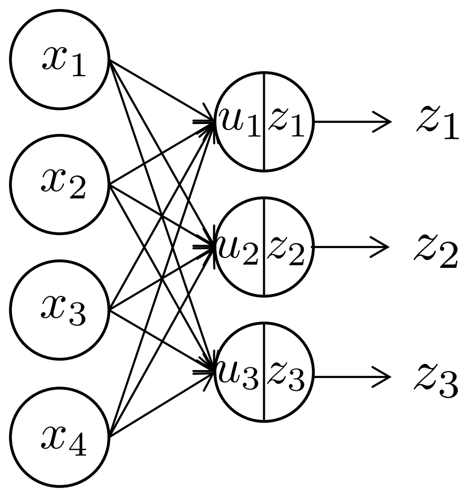
- The derivative wrt. $w_{ji}^{(l)}$ (a lth layer weight) is written as:

$$\frac{\partial E_n}{\partial w_{ji}^{(l)}} = \frac{\partial E_n}{\partial u_j^{(l)}} \frac{\partial u_j^{(l)}}{\partial w_{ji}^{(l)}}$$



$$u_j = \sum_{i=1}^{I} w_{ji} x_i + b_j$$

$$z_j = f(u_j)$$



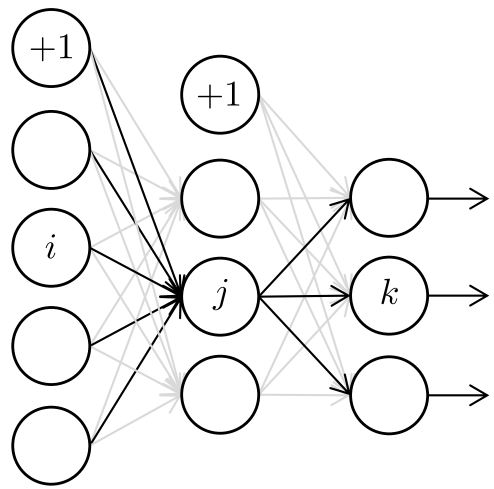
Derivation of BP algorithm: Chain rule 2/2

$$\frac{\partial E_n}{\partial w_{ji}^{(l)}} = \frac{\partial E_n}{\partial u_j^{(l)}} \frac{\partial u_j^{(l)}}{\partial w_{ji}^{(l)}}$$

- 2nd term is rewritten as $\frac{\partial u_j^{(l)}}{\partial w_{ji}^{(l)}} = z_i^{(l-1)}$ $\left(\Leftarrow \; u_j^{(l)} = \sum_i w_{ji}^{(l)} z_i^{(l-1)} \right)$

- $u_j^{(l)}$ affects $u_k^{(l+1)}$ (i.e., $u_k^{(l+1)}$ is a function of $u_j^{(l)}$) (k=1,…)

$$\frac{\partial E_n}{\partial u_j^{(l)}} = \sum_k \frac{\partial E_n}{\partial u_k^{(l+1)}} \frac{\partial u_k^{(l+1)}}{\partial u_j^{(l)}}$$



Derivation of BP algorithm: Deltas

$$\underline{\frac{\partial E_n}{\partial u_j^{(l)}}} = \sum_k \underline{\frac{\partial E_n}{\partial u_k^{(l+1)}}}\,\underline{\frac{\partial u_k^{(l+1)}}{\partial u_j^{(l)}}}$$

- We define delta as: $\delta_j^{(l)} \equiv \dfrac{\partial E_n}{\partial u_j^{(l)}}$
- Then the equation is rewritten as:

$$\underline{\delta_j^{(l)}} = \sum_k \underline{\delta_k^{(l+1)}}\,\underline{\left(w_{kj}^{(l+1)} f'(u_j^{(l)})\right)}$$

$$u_k^{(l+1)} = \textstyle\sum_j w_{kj}^{(l+1)} z_j^{(l)} = \sum_j w_{kj}^{(l+1)} f(u_j^{(l)})$$

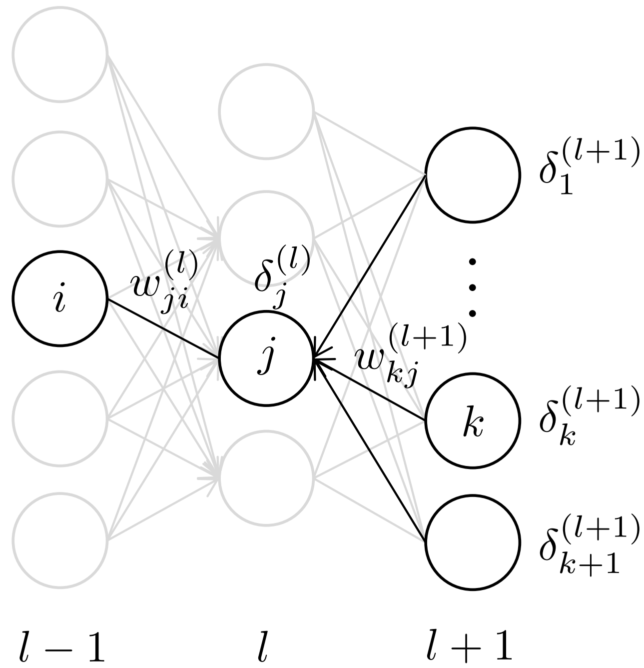
$$\Rightarrow\ \partial u_k^{(l+1)} / \partial u_j^{(l)} = w_{kj}^{(l+1)} f'(u_j^{(l)})$$

Derivation of BP algorithm: Backprop of deltas

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} \left(w_{kj}^{(l+1)} f'(u_j^{(l)}) \right)$$

- This eq means deltas can be calculated by backpropagation
- We can start it from deltas at the output layer

$$\delta_j^{(L)} = \frac{\partial E_n}{\partial u_j^{(L)}}$$



Derivation of BP algorithm: Summary

- The derivative we want is given by using the delta as

$$\frac{\partial E_n}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)} \quad \left(\Leftarrow \ \frac{\partial E_n}{\partial w_{ji}^{(l)}} = \frac{\partial E_n}{\partial u_j^{(l)}} \frac{\partial u_j^{(l)}}{\partial w_{ji}^{(l)}} \right)$$

- If the loss is sum over multiple samples, we need only to calculate the sum of gradients over them as follows:

$$E = \sum_n E_n$$

$$\frac{\partial E}{\partial w_{ji}^{(l)}} = \sum_n \frac{\partial E_n}{\partial w_{ji}^{(l)}}$$

Complete backpropagation algorithm

- **Input:** a pair of input x_n and desired output d_n
- **Output:** the derivative of a loss wrt. each of all layer weights

1. Forward propagation from the input layer
   - z_n and u_n at each layer l are computed for l=2,3,…,L
2. Compute deltas at the output layer
3. Backward propagation from the output layer
   - Compute deltas $\delta_j^{(l)}$ at each layer l for l=L,L-1,…2 according to

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} \left(w_{kj}^{(l+1)} f'(u_j^{(l)}) \right)$$

4. Compute the derivative of the loss wrt. each weight $w_{ji}^{(l)}$ by

$$\frac{\partial E_n}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)}$$

Deltas at the output layer

- Regression with squared loss and identity act-func at output layer

$$E_n = \frac{1}{2}\|\mathbf{y} - \mathbf{d}\|^2 = \frac{1}{2}\sum_j (y_j - d_j)^2 \qquad \left(y_j = z_j^{(L)} = u_j^{(L)} \right)$$

$$\delta_j^{(L)} = \frac{\partial E_n}{\partial u_j^{(L)}} = u_j^{(L)} - d_j = z_j^{(L)} - d_j = y_j - d_j$$

- Multi-class classification: cross-entropy loss and softmax

$$E_n = -\sum_k d_k \log y_k = -\sum_k d_k \log\left(\frac{\exp(u_k^{(L)})}{\sum_i \exp(u_i^{(L)})}\right)$$

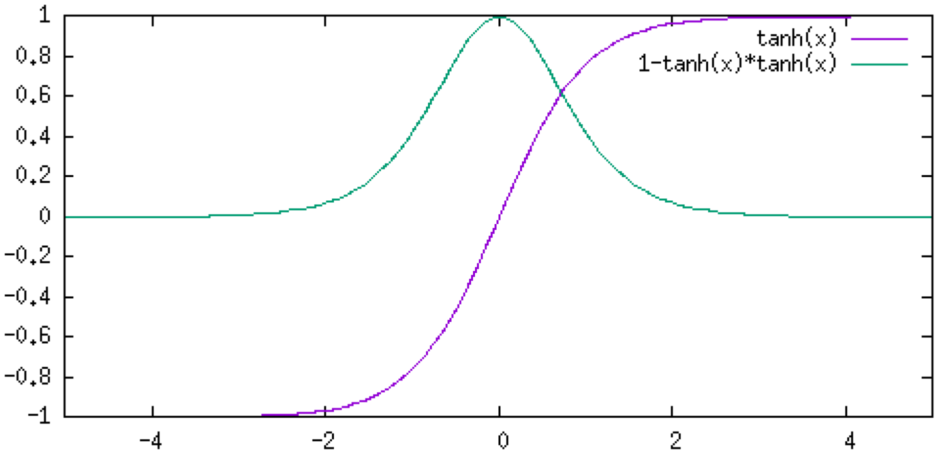
$$\delta_j^{(L)} = -\sum_k d_k \frac{1}{y_k}\frac{\partial y_k}{\partial u_j^{(L)}} \qquad \left(d_k^2 = d_k \quad d_k d_j = 0\ (k \neq j) \quad \sum_k d_k = 1 \right)$$

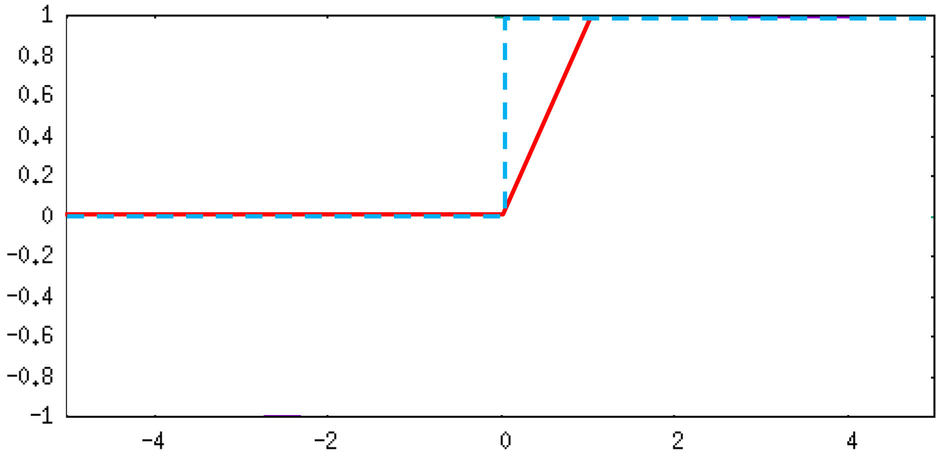
$$= -d_j(1 - y_j) - \sum_{k \neq j} d_k(-y_j) = \sum_k d_k(y_j - d_j) = y_j - d_j$$

$$(f/g)' = (f'g - fg')/g^2$$

Derivation of activation functions

$f(u)$	$f'(u)$
$f(u) = 1/(1+e^{-u})$	$f'(u) = f(u)(1-f(u))$
$f(u) = \tanh(u)$	$f'(u) = 1 - \tanh^2(u)$
$f(u) = \max(u, 0)$	$f'(u) = \begin{cases} 1 & u \geq 0 \\ 0 & u < 0 \end{cases}$





Outline of training a neural net

1. Design your network
   – Number of layers, units at each layer, activation functions etc.
2. Choose an optimizer w/ *hyper-parameters*
   – SGD w/momentum, Adam, …
   – Learning rate, momentum, …
   – Initialize weights and biases
3. Prepare your data
   – Divide all the available data into train, validation, test splits
   – Preprocess the data (e.g., standardization, data augmentation)
   – Create minibatches
4. Run your optimizer on the train split
   – Weights are updated per a minibatch (= one *iteration*)
   – Repeat updates for one or more *epochs*
     • One epoch = iterations over all minibatches
   – Check at times generalization performance of your net using the val split

Stochastic gradient descent: Use of minibatch

- Batch optimization
  - Minimize the sum of errors (losses) over all training samples

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \nabla E_n \quad \text{where} \quad E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

    - Remark: This appears reasonable considering our goal; however this does not work well in practice (→ it will be trapped in bad solutions)

- Stochastic gradient descent (SGD)
  - Compute gradient of *E* of a single sample or at most hundreds samples (= *minibatch*) and then update w

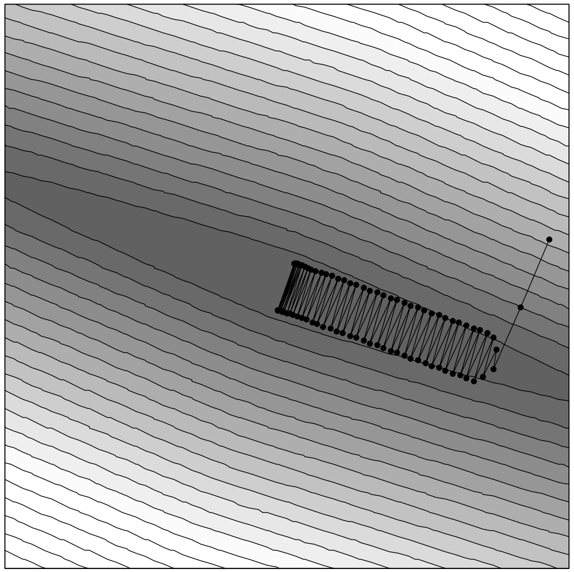
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \nabla E_t \quad \text{where} \quad E_t(\mathbf{w}) = \frac{1}{N_t} \sum_{n \in \mathcal{D}_t} E_n(\mathbf{w})$$

    - Remark: We minimize a different objective func at each update; but this works much better

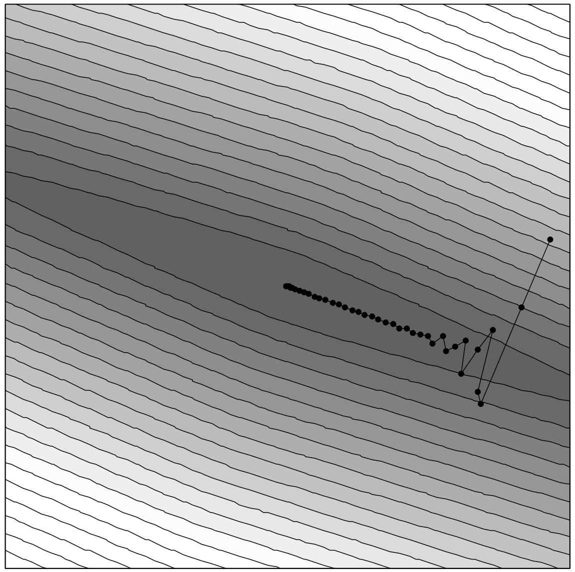
SGD w/ momentum

- That said, each update tends to be unstable; needs many iterations when trapped around *ravine*
- Some amount (μ~0.5-0.9) of the last update is added
  - Rolling ball with inertia; improved stability; efficient exploration

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \nabla E_t + \mu \mathbf{v}_t \quad \text{where} \quad \mathbf{v}_t \equiv \mathbf{w}_t - \mathbf{w}_{t-1}$$



w/o momentum

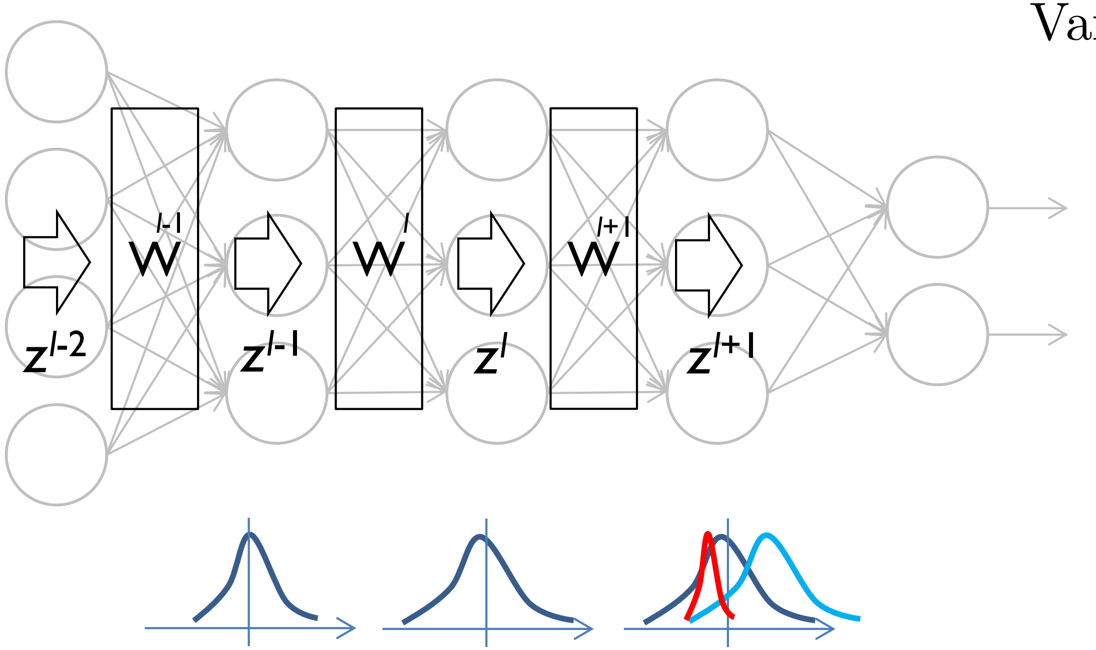


w/ momentum

Initialization of weights

- Weights and biases are randomly initialized
  - Chosen from a normal distribution $w_{ji} \sim N(0, \sigma^2)$
  - Or a uniform distribution $w_{ji} \sim U(-a, a)$
- How do we select the range, i.e., σ or a?
  - Its choice is very, very important
- A requirement: the layer outputs between neighboring layers should have similar distribution widths

$$\mathrm{Var}(z^{(l+1)}) \sim \mathrm{Var}(z^{(l)})$$

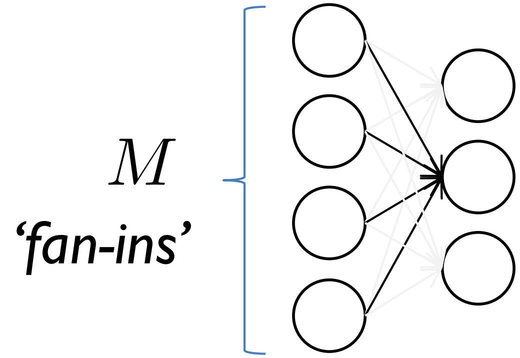


Initialization of weights

- Problem: What's the condition on *σ* to enable $\mathrm{Var}(z^{(l+1)}) \sim \mathrm{Var}(z^{(l)})$?

- A layer propagation before act-func.:

$$u_j^{(l+1)} = \sum_{i=1}^{M} w_{ji}^{(l+1)} z_i^{(l)}$$



- Assuming independence of the variables plus *E*(*w*)=0, we have

$$\mathrm{Var}(u^{(l+1)}) = M\mathrm{Var}(w^{(l+1)} z^{(l)}) = M\mathrm{Var}(w^{(l+1)}) E(z^{(l)2})$$

  – We used here $\mathrm{Var}(XY) = E(X^2)E(Y^2) - E(X)^2 E(Y)^2$

- If *E*(*z*)=0, then the above reduces to

$$\mathrm{Var}(u^{(l+1)}) = M\mathrm{Var}(w^{(l+1)})\mathrm{Var}(z^{(l)})$$

Initialization of weights

- If we assume an identity function for *f*
  - We can assume *E*(z) = 0 and

$$\mathrm{Var}(z^{(l+1)}) = \mathrm{Var}(u^{(l+1)}) = M\mathrm{Var}(w^{(l+1)})\mathrm{Var}(z^{(l)})$$

- We require $\mathrm{Var}(z^{(l+1)}) \sim \mathrm{Var}(z^{(l)})$

$$\mathrm{Var}(w) = 1/M \;\;\Rightarrow\;\; w \sim N(0, 1/\sqrt{M})$$

--- A standard initialization (the default in PyTorch)

- When *f* is ReLU, *E*(z) ≠ 0 and instead we have

$$\mathrm{Var}(u^{(l+1)}) = \frac{1}{2}M\mathrm{Var}(w^{(l+1)})\mathrm{Var}(u^{(l)})$$

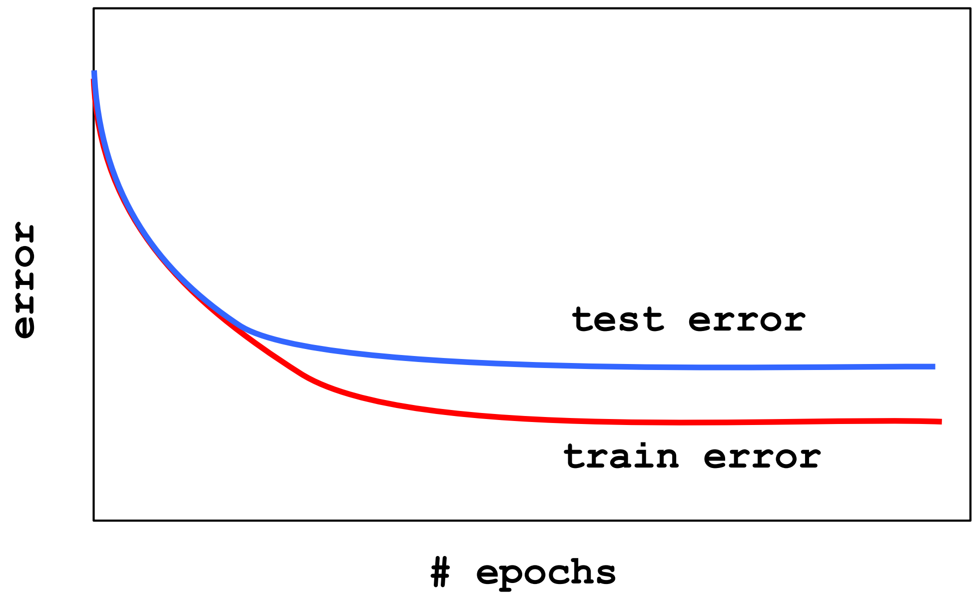
  - We used here: $E(z^{(l)2}) = \frac{1}{2}\mathrm{Var}(u^{(l)})$ for $z^{(l)} = \max(0, u^{(l)})$
  - The requirement reduces to

$$\mathrm{Var}(w) = 2/M \;\;\Rightarrow\;\; w \sim N(0, \sqrt{2/M})$$

--- A method proposed in [Kaiming He+ 2015]

Generalization ability

- Goal of training: to make it possible to predict outcome values for *previously unseen* data
- We minimize loss on training data = *training loss/error*
  - You can make it as small as you like, even toward zero
    - E.g., By using a model (net) with an excessively large number of parameters
- To check how close to the above goal, we evaluate the performance of our model on the samples we haven't used for training
  - We save a portion of the data for this purpose = *validation/test data*



Standard practice of handling data

Available data: (x,d) pairs

Training data	Validation data	Test data
Optimize model parameters	Estimate perf. on unseen data	Final evaluation
• Standardization is found and applied • Further split to minibatches	• Check generalization ability • Select hyper-parameters Don't do this on test data • Same standardization needs to be applied	• Estimate real perf. • Same standardization needs to be applied

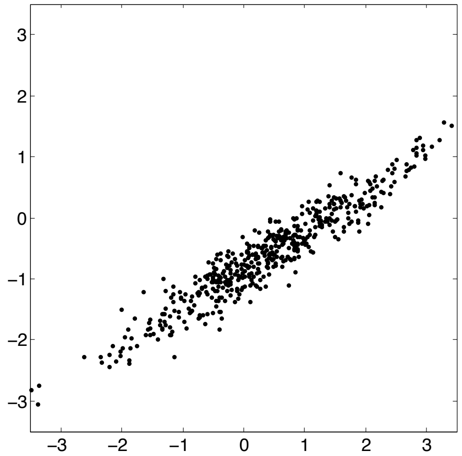
Cross-validation

Each used to …

Standardization of inputs

$$\mathbf{x}_n = [x_{n1}, x_{n2}, \ldots, x_{nI}]^\top$$

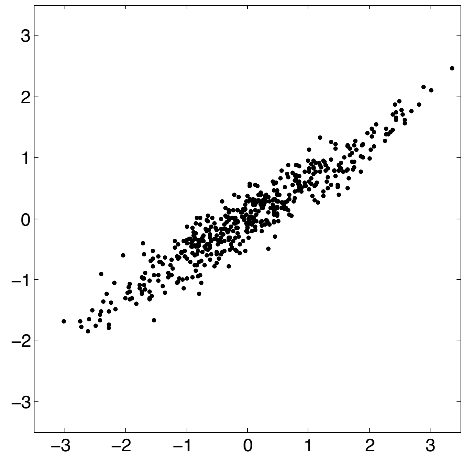
Input distrib.



Step 1: Subtract mean

$$x_{ni} \leftarrow x_{ni} - \bar{x}_i$$

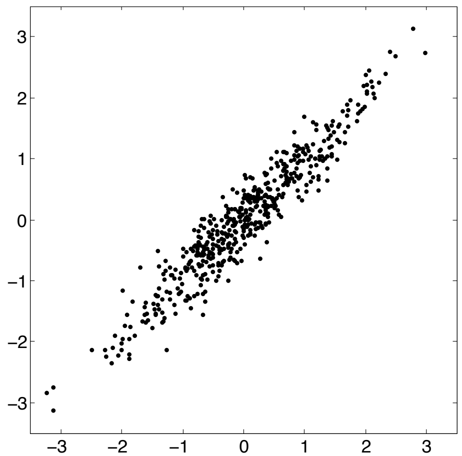
$$\bar{x}_i \equiv \sum_{n=1}^{N} x_{ni}/N$$



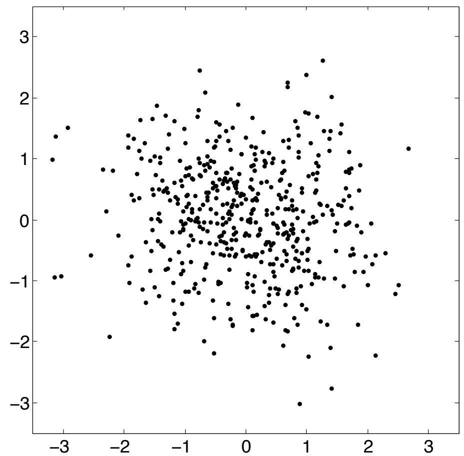
Step 2: Divide by std. dev.

$$x_{ni} \leftarrow \frac{x_{ni} - \bar{x}_i}{\sigma_i}$$

$$\sigma_i = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(x_{ni} - \bar{x}_i)^2}$$



Step 3: Decorrelate elements (*whitening*)

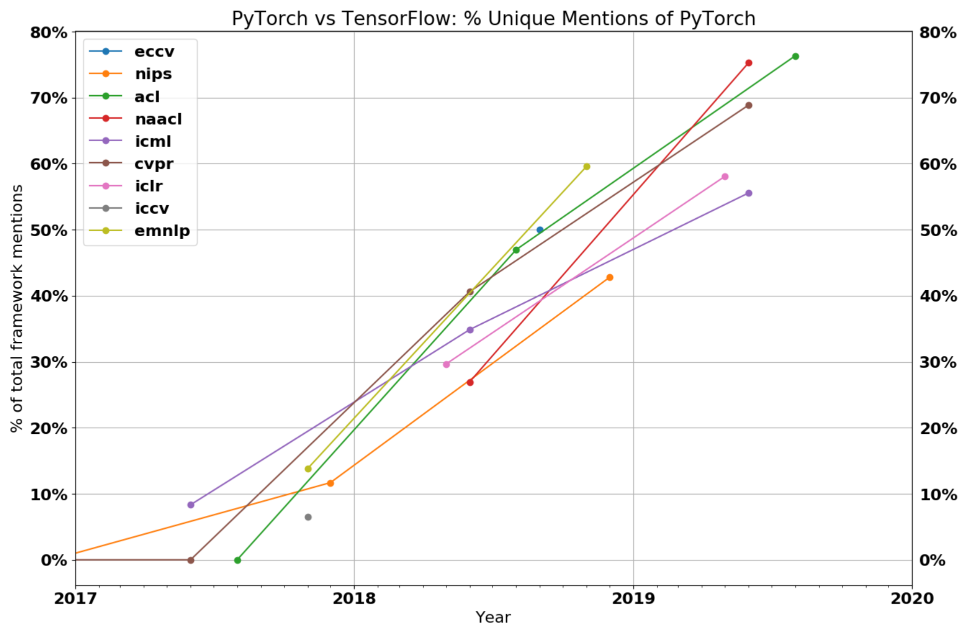


How to get started on *Google Colaboratory*

- You need to have a Google account and log in to it with your browser
- Access https://colab.research.google.com/
  - Choose 'File'–'New Python 3 notebook' to create a blank notebook
  - Then 'File'–'Rename…' to name it and 'File'–'Save' to save it in your Drive
    - Don't forget choose 'Runtime'–'Change runtime type' and set 'Hardware accelerator' to 'GPU' with the notebook
- Or click here to open a sample notebook, and either
  - 'Open in Playground' below the menu bar to immediately test it
  - Or 'File'–'Save a copy in Drive…' to make its copy on your own Drive
  - Click a cell and press SHIFT+RETURN to run the code at the cell

Programming language & DL framework

- We will use *Python 3* for writing code
  - Get familiar with the language using online learning resources
    - E.g., https://wiki.python.org/moin/BeginnersGuide/NonProgrammers
  - Learn by practicing; you can start with minimum knowledge
- and *PyTorch* for building/training/testing neural nets
  - Primarily developed by *Facebook AI*
  - Shares popularity with Google's *Tensorflow*



https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/