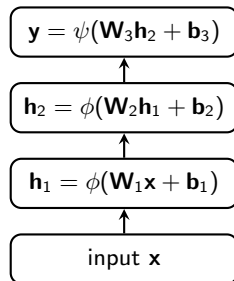


CS-E4890: Deep Learning

Lecture 2: Optimization

Alexander Ilin

- Suppose we have a supervised learning task with training data: $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \dots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\}$.
- In this lecture, we will study how to train a neural network to produce the correct output \mathbf{y} for a given input \mathbf{x} .
- Training of a neural network is tuning the values of its parameters to fit to the training data.
- Two most common tasks:
 - classification: the output is discrete (class label)
 - regression: the output is a real number



- Classification tasks: a target can be represented as a one-hot vector \mathbf{y} .
For example, for $K = 3$ classes:

$$\text{class 1: } \mathbf{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{class 2: } \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{class 3: } \mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$y_j \in \{0, 1\} \quad \sum_{j=1}^K y_j = 1$$

- We want our neural network to produce as output vector \mathbf{f} whose j -th element f_j is the probability that input \mathbf{x} belongs to class j . Thus, we need to make sure that:

$$0 \leq f_j \leq 1 \quad \sum_{j=1}^K f_j = 1$$

- We can guarantee that by transforming the output \mathbf{h} of the last layer in the following way:

$$f_j = \frac{\exp h_j}{\sum_{j'=1}^K \exp h_{j'}}$$

- This nonlinearity is called softmax.
 - If $h_j \rightarrow \infty$ and the other $h_{j' \neq j}$ are fixed, then $f_j \rightarrow [0, \dots, 0, 1, 0, \dots, 0]$, which is a one-hot representation of j , the index of the maximum element of \mathbf{h} (thus soft max function).

- It is common to tune parameters θ by minimizing the following loss function:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{n=1}^N \sum_{j=1}^K y_j^{(n)} \log f_j(\mathbf{x}^{(n)}, \theta)$$

which is the negative log-likelihood for a probabilistic model with a categorical (also called multinoulli) distribution for \mathbf{y} whose parameters are given by $\mathbf{f}(\mathbf{x}, \theta)$

$$p(\mathbf{y} \mid \mathbf{x}, \theta) = \text{Cat}(\mathbf{y} \mid \mathbf{f}(\mathbf{x}, \theta)) = \prod_{j=1}^K f_j^{y_j} = f_{j'} \quad \text{where } y_{j'} = 1$$

- Cross-entropy* between two discrete probability distributions p and q is defined as

$$\mathcal{H}(p, q) = - \sum_{x \in \mathcal{X}} p(x) \log q(x)$$

thus our loss \mathcal{L} can be seen as the cross-entropy between the correct distribution defined by $\mathbf{y}^{(n)}$ and the distribution $\mathbf{f}(\mathbf{x}^{(n)}, \theta)$ defined by the output of the network.

- Regression tasks: targets are $\mathbf{y}^{(n)} \in \mathbb{R}^K$.
- We can tune the parameters of the network by minimizing the mean-squared error (MSE):

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \right\|^2.$$

- In the probabilistic view, the minimized function is the negative log-likelihood of the following probability distribution:

$$p(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y} \mid \mathbf{f}(\mathbf{x}, \boldsymbol{\theta}), \sigma^2 \mathbf{I}).$$

Minimization of the loss with gradient descent

Toy optimization problem

- Consider a simple linear regression problem with two parameters:

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \mathbf{x} = w_1 x_1 + w_2 x_2$$

and a tiny data set: $\mathbf{x}^{(1)} = (2, 2)$, $y^{(1)} = 2$, $\mathbf{x}^{(2)} = (2, 0)$, $y^{(2)} = 0$

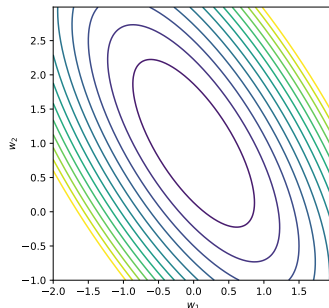
- The MSE loss function is a quadratic function

$$\mathcal{L}(w_1, w_2) = \frac{1}{2} \sum_{n=1}^2 \left(y^{(n)} - f(\mathbf{x}^{(n)}, \mathbf{w}) \right)^2$$

which can be written in the matrix notation as

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \mathbf{w}^\top \mathbf{A} \mathbf{w} - \mathbf{b}^\top \mathbf{w} + \mathbf{c}$$

- We need to minimize \mathcal{L} wrt \mathbf{w} (w_1 and w_2).



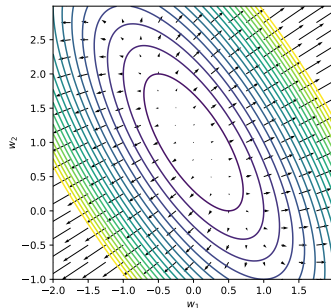
- Gradient is a vector of partial derivatives:

$$\mathbf{g}(\mathbf{w}) = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_M} \end{pmatrix}$$

- Gradient points in the direction of the greatest rate of increase of \mathcal{L} , its magnitude is the slope of the graph of \mathcal{L} in that direction.

- For our quadratic function $\mathcal{L}(\mathbf{w}) = \frac{1}{2} \mathbf{w}^\top \mathbf{A} \mathbf{w} - \mathbf{b}^\top \mathbf{w} + \mathbf{c}$, the gradient is

$$\mathbf{g}(\mathbf{w}) = \mathbf{A} \mathbf{w} - \mathbf{b}$$



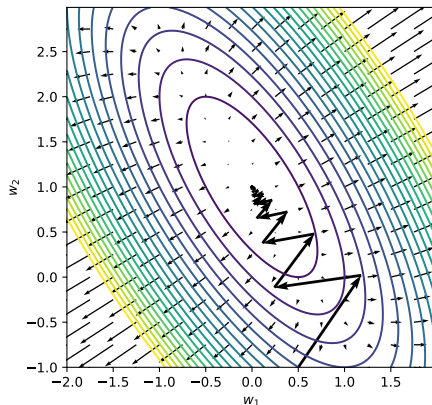
- Gradient descent: update the parameters in the direction opposite to the gradient:

$$\theta \leftarrow \theta - \eta \mathbf{g}(\theta)$$

with some step size η (also called *learning rate*).

- We reduce the error but do not end up at the minimum, so we need to iterate

$$\theta_{t+1} = \theta_t - \eta_t \mathbf{g}(\theta_t)$$

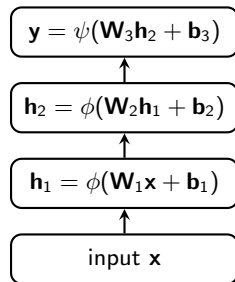


Gradient computation with
the backpropagation algorithm

- We want to use gradient-descent optimization method to minimize loss function $\mathcal{L}(\theta)$:

$$\theta_{t+1} = \theta_t - \eta_t \mathbf{g}(\theta_t)$$

- In order to do that, we need to compute gradient $\mathbf{g}(\theta)$.
 - Parameters θ include \mathbf{W}_1 , \mathbf{b}_1 , \mathbf{W}_2 , \mathbf{b}_2 , \mathbf{W}_3 , \mathbf{b}_3 .
- Backpropagation: An algorithm to compute gradient $\mathbf{g}(\theta)$ for a multilayer neural network.



- The chain rule is a formula to compute the derivative of a composite function:

$$F(x) = f(g(x))$$

$$F'(x) = f'(g(x))g'(x)$$

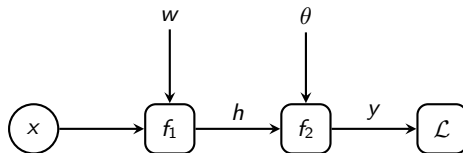
Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

$$\mathcal{L} = \mathcal{L}(y), \quad y = f_2(h, \theta), \quad h = f_1(x, w)$$

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial \theta} \\ \frac{\partial \mathcal{L}}{\partial w} &= \underbrace{\frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial h}}_{\frac{\partial \mathcal{L}}{\partial h}} \frac{\partial h}{\partial w} \end{aligned}$$



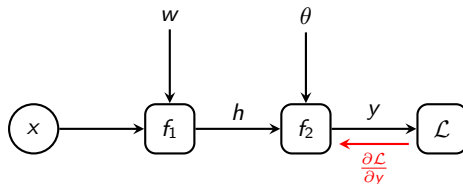
Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

$$\mathcal{L} = \mathcal{L}(y), \quad y = f_2(h, \theta), \quad h = f_1(x, w)$$

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial \theta} \\ \frac{\partial \mathcal{L}}{\partial w} &= \underbrace{\frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial h}}_{\frac{\partial \mathcal{L}}{\partial h}} \frac{\partial h}{\partial w} \end{aligned}$$



- We can compute the derivatives efficiently by storing intermediate results.

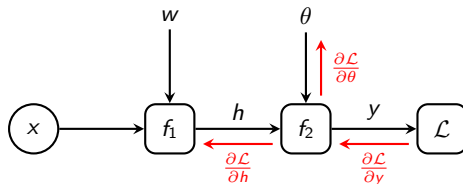
Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

$$\mathcal{L} = \mathcal{L}(y), \quad y = f_2(h, \theta), \quad h = f_1(x, w)$$

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial \theta} \\ \frac{\partial \mathcal{L}}{\partial w} &= \underbrace{\frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial h}}_{\frac{\partial \mathcal{L}}{\partial h}} \frac{\partial h}{\partial w} \end{aligned}$$



- We can compute the derivatives efficiently by storing intermediate results.

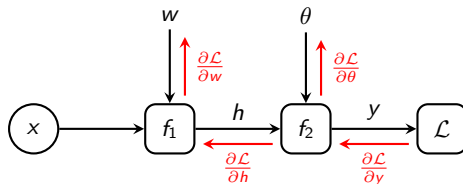
Backpropagation: An example with scalars

- Consider a multi-layer model that operates only with scalars:

$$\mathcal{L} = \mathcal{L}(y), \quad y = f_2(h, \theta), \quad h = f_1(x, w)$$

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta} &= \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial \theta} \\ \frac{\partial \mathcal{L}}{\partial w} &= \underbrace{\frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial h}}_{\frac{\partial \mathcal{L}}{\partial h}} \frac{\partial h}{\partial w} \end{aligned}$$



- We can compute the derivatives efficiently by storing intermediate results.

- For multi-variable functions, the chain rule can be written in terms of Jacobian matrices.

$$\mathbf{y} = f(\mathbf{u}), \quad \mathbf{u} = g(\mathbf{x}) \quad \mathbf{y} \in \mathbb{R}^M, \mathbf{u} \in \mathbb{R}^K, \mathbf{x} \in \mathbb{R}^N$$

$$\text{Jacobian matrix: } \mathbf{J}_{f \circ g} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_N} \end{bmatrix}$$

- The chain rule is:

$$\mathbf{J}_{f \circ g}(\mathbf{x}) = \mathbf{J}_f(\mathbf{u})\mathbf{J}_g(\mathbf{x})$$

or each element of the Jacobian is:

$$\frac{\partial y_j}{\partial x_i} = \sum_{k=1}^K \frac{\partial y_j}{\partial u_k} \frac{\partial u_k}{\partial x_i}$$

Backpropagation for multi-variable functions

- Consider a multi-layer model:

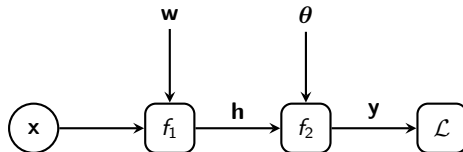
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \quad \mathbf{y} \in \mathbb{R}^K, \mathbf{h} \in \mathbb{R}^L, \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial h_l} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial h_l}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{l=1}^L \frac{\partial \mathcal{L}}{\partial h_l} \frac{\partial h_l}{\partial w_i}$$



Backpropagation for multi-variable functions

- Consider a multi-layer model:

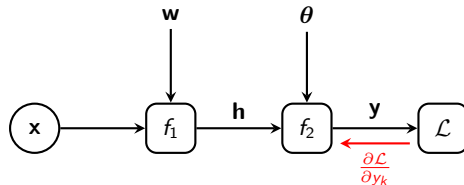
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \quad \mathbf{y} \in \mathbb{R}^K, \mathbf{h} \in \mathbb{R}^L, \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial h_l} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial h_l}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{l=1}^L \frac{\partial \mathcal{L}}{\partial h_l} \frac{\partial h_l}{\partial w_i}$$



- We can compute the derivatives sequentially going from the outputs of the network towards the inputs (thus the name of the algorithm *backpropagation*).

Backpropagation for multi-variable functions

- Consider a multi-layer model:

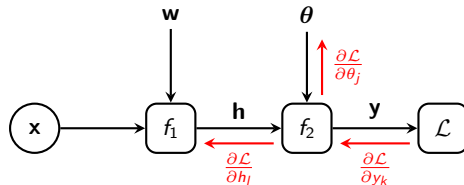
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \quad \mathbf{y} \in \mathbb{R}^K, \mathbf{h} \in \mathbb{R}^L, \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial h_l} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial h_l}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{l=1}^L \frac{\partial \mathcal{L}}{\partial h_l} \frac{\partial h_l}{\partial w_i}$$



- We can compute the derivatives sequentially going from the outputs of the network towards the inputs (thus the name of the algorithm *backpropagation*).

Backpropagation for multi-variable functions

- Consider a multi-layer model:

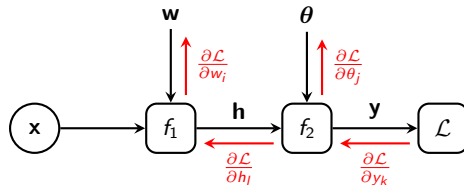
$$\mathcal{L} = \mathcal{L}(\mathbf{y}), \quad \mathbf{y} = f_2(\mathbf{h}, \boldsymbol{\theta}), \quad \mathbf{h} = f_1(\mathbf{x}, \mathbf{w}) \quad \mathbf{y} \in \mathbb{R}^K, \mathbf{h} \in \mathbb{R}^L, \mathbf{x} \in \mathbb{R}^N$$

- We apply the chain rule to compute the derivatives wrt the model parameters (and re-use intermediate derivatives):

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial h_l} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial h_l}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{l=1}^L \frac{\partial \mathcal{L}}{\partial h_l} \frac{\partial h_l}{\partial w_i}$$



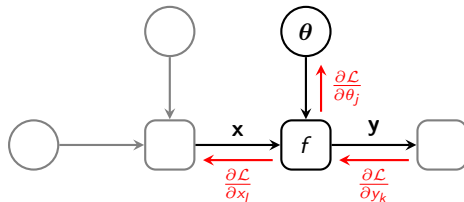
- We can compute the derivatives sequentially going from the outputs of the network towards the inputs (thus the name of the algorithm *backpropagation*).

Implementing backpropagation in software

- For each block of a neural network, we need to implement the following computations:
 - forward computations $\mathbf{y} = f(\mathbf{x}, \theta)$
 - backward computations that transform the derivatives wrt the block's outputs $\frac{\partial \mathcal{L}}{\partial y_k}$ into the derivatives wrt all its inputs: $\frac{\partial \mathcal{L}}{\partial x_l}, \frac{\partial \mathcal{L}}{\partial \theta_j}$

$$\frac{\partial \mathcal{L}}{\partial \theta_j} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial \theta_j}$$

$$\frac{\partial \mathcal{L}}{\partial x_l} = \sum_{k=1}^K \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial y_k}{\partial x_l}$$



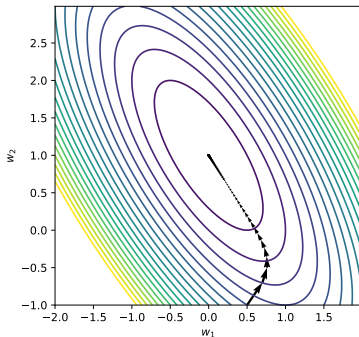
- We will practice implementing forward and backward computations in the first assignment.

- The algorithm that is now called backpropagation was proposed by many researchers (e.g., Linnainmaa, 1970; Werbos, 1982).
- In application to training multi-layer neural networks, the algorithm became popular after a paper by [Rumelhart, Hinton and Williams \(1986\)](#).

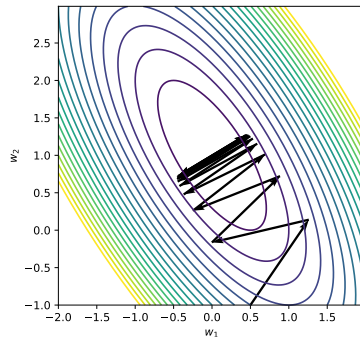
Analysis of convergence of gradient descent

- The learning rate η has a major effect on the convergence of the gradient descent.

$$\theta_{t+1} = \theta_t - \eta g(\theta_t)$$



small η : too slow convergence



large η : oscillates and can even diverge

- Apart from the learning rate, what affects the rate of convergence of the gradient descent?
- Let us analyze convergence of gradient descent for a quadratic function (Goh, 2017)

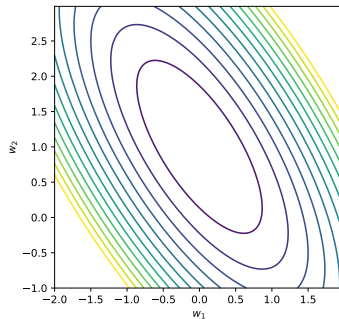
$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \mathbf{w}^\top \mathbf{A} \mathbf{w} - \mathbf{b}^\top \mathbf{w}$$

- We can compute optimal \mathbf{w} analytically: $\mathbf{w}_* = \mathbf{A}^{-1} \mathbf{b}$
- Gradient descent iterations:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta(\mathbf{A} \mathbf{w}_t - \mathbf{b})$$

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \mathbf{w}^\top \mathbf{A} \mathbf{w} - \mathbf{b}^\top \mathbf{w}$$

- The axes of the ellipses of the contour plot are defined by the eigenvectors of matrix \mathbf{A} .
- The eigenvalues λ_m of \mathbf{A} determine the curvature of the objective function: Larger λ_m correspond to higher curvatures in the corresponding direction.



- Let us change the coordinate system such that the new basis is aligned with the eigenvectors of \mathbf{A} .
 - We compute the eigenvalue decomposition of \mathbf{A} :

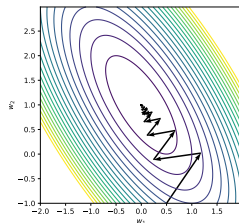
$$\mathbf{A} = \mathbf{Q} \text{diag}(\lambda_1, \dots, \lambda_M) \mathbf{Q}^\top$$

where \mathbf{Q} is an orthogonal matrix and λ_m are ordered eigenvalues $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_M$.

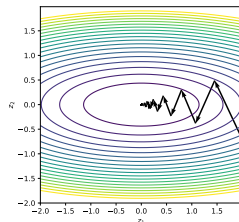
- Then we use \mathbf{Q} to rotate the coordinate system:

$$\mathbf{z} = \mathbf{Q}^\top (\mathbf{w} - \mathbf{w}_*)$$

$$\mathbf{w} = \mathbf{w}_* + \mathbf{Q} \mathbf{z}$$



old system \mathbf{w}



new system \mathbf{z}

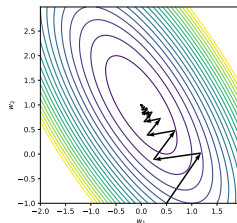
Analysis of convergence of gradient descent

- Change of basis: $\mathbf{z} = \mathbf{Q}^\top(\mathbf{w} - \mathbf{w}_*)$ and $\mathbf{w} = \mathbf{w}_* + \mathbf{Q}\mathbf{z}$
- Gradient descent in the new coordinates:

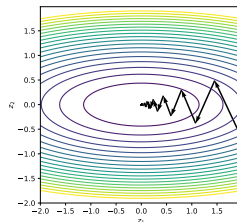
$$\begin{aligned}\mathbf{z}_{t+1} &= \mathbf{Q}^\top(\mathbf{w}_{t+1} - \mathbf{w}_*) = \mathbf{Q}^\top(\mathbf{w}_t - \eta(\mathbf{A}\mathbf{w}_t - \mathbf{b}) - \mathbf{w}_*) \\ &= \mathbf{Q}^\top(\mathbf{Q}\mathbf{z}_t - \eta(\mathbf{A}(\mathbf{w}_* + \mathbf{Q}\mathbf{z}_t) - \mathbf{b})) \\ &= \mathbf{Q}^\top(\mathbf{Q}\mathbf{z}_t - \eta(\mathbf{b} + \mathbf{A}\mathbf{Q}\mathbf{z}_t - \mathbf{b})) \\ &= \mathbf{z}_t - \eta\mathbf{Q}^\top\mathbf{A}\mathbf{Q}\mathbf{z}_t = \mathbf{z}_t - \eta\text{diag}(\lambda_1, \dots, \lambda_M)\mathbf{z}_t\end{aligned}$$

- In the new coordinate system, we can write the update equation separately for each element of \mathbf{z} :

$$(z_m)_{t+1} = (z_m)_t - \eta\lambda_m(z_m)_t = (1 - \eta\lambda_m)(z_m)_t$$



old system \mathbf{w}



new system \mathbf{z}

- Gradient descent for the m -th element of \mathbf{z} :

$$(z_m)_{t+1} = (1 - \eta\lambda_m)(z_m)_t$$

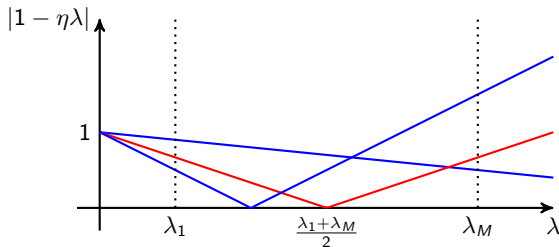
- Since the optimum $\mathbf{z}_* = 0$, the rate of convergence of z_m (see, e.g, [here](#)) is defined by

$$\text{rate}(\eta) = \frac{|(z_m)_{t+1}|}{|(z_m)_t|} = |1 - \eta\lambda_m|$$

- for convergence: $|1 - \eta\lambda_m| < 1$
- ideally: $|1 - \eta\lambda_m| = 0$

- The overall convergence rate is determined by the slowest component (either λ_1 or λ_M):

$$\begin{aligned}\text{rate}(\eta) &= \max_m |1 - \eta\lambda_m| \\ &= \max \{|1 - \eta\lambda_1|, |1 - \eta\lambda_M|\}\end{aligned}$$



- This overall rate is minimized when the rates for λ_1 and λ_M are the same, which is true for the learning rate

$$\eta_* = \left(\frac{\lambda_1 + \lambda_M}{2} \right)^{-1}$$

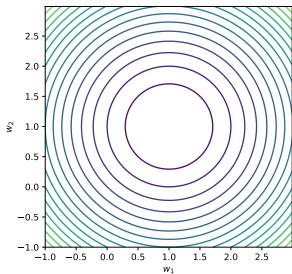
- The rate of convergence for the optimal learning rate is

$$\begin{aligned}\text{rate}(\eta_*) &= \left| 1 - \left(\frac{\lambda_1 + \lambda_M}{2} \right)^{-1} \lambda_1 \right| = \left| \frac{\lambda_1 + \lambda_M - 2\lambda_1}{\lambda_1 + \lambda_M} \right| = \frac{\lambda_M - \lambda_1}{\lambda_M + \lambda_1} \\ &= \frac{\lambda_M/\lambda_1 - 1}{\lambda_M/\lambda_1 + 1} = \frac{\kappa(\mathbf{A}) - 1}{\kappa(\mathbf{A}) + 1}\end{aligned}$$

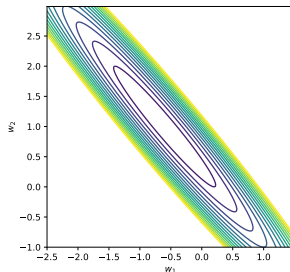
where $\kappa(\mathbf{A}) = \frac{\lambda_M}{\lambda_1}$ is the condition number of matrix \mathbf{A} .

- $\kappa(\mathbf{A})$ is a measure of how close to singular matrix \mathbf{A} is.
- It is a measure of how poorly gradient descent will perform:
 - $\kappa(\mathbf{A}) = 1$ is ideal
 - The larger $\kappa(\mathbf{A})$ is, the slower gradient descent will be.

- For quadratic function $c(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top \mathbf{A}\mathbf{w} - \mathbf{b}^\top \mathbf{w}$, the rate of convergence of the gradient descent is determined by the condition number of matrix \mathbf{A} :



$\kappa(\mathbf{A}) = 1$: can converge in one iteration



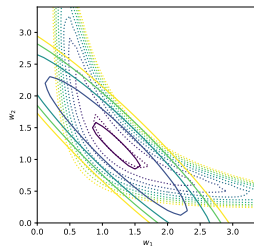
Large $\kappa(\mathbf{A})$: slow convergence

- For non-quadratic functions, the error surface locally is well approximated by a quadratic function:

$$\mathcal{L}(\mathbf{w}) \approx \mathcal{L}(\mathbf{w}_t) + \mathbf{g}^\top (\mathbf{w} - \mathbf{w}_t) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_t)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_t)$$

- \mathbf{H} is the matrix of second-order derivatives (called Hessian):

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial w_M \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_M \partial w_M} \end{pmatrix}$$



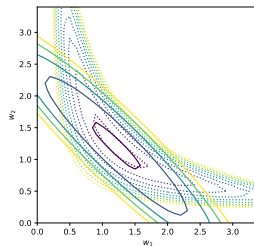
- What is the Hessian matrix for the quadratic loss $\mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top \mathbf{A}\mathbf{w} - \mathbf{b}^\top \mathbf{w}$?

- For non-quadratic functions, the error surface locally is well approximated by a quadratic function:

$$\mathcal{L}(\mathbf{w}) \approx \mathcal{L}(\mathbf{w}_t) + \mathbf{g}^\top (\mathbf{w} - \mathbf{w}_t) + \frac{1}{2} (\mathbf{w} - \mathbf{w}_t)^\top \mathbf{H} (\mathbf{w} - \mathbf{w}_t)$$

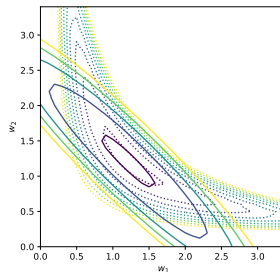
- \mathbf{H} is the matrix of second-order derivatives (called Hessian):

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial w_M \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_M \partial w_M} \end{pmatrix}$$



- What is the Hessian matrix for the quadratic loss $\mathcal{L}(\mathbf{w}) = \frac{1}{2} \mathbf{w}^\top \mathbf{A} \mathbf{w} - \mathbf{b}^\top \mathbf{w}$?
- $\mathbf{H} = \mathbf{A}$: the convergence of the gradient descent is affected by the properties of the Hessian.

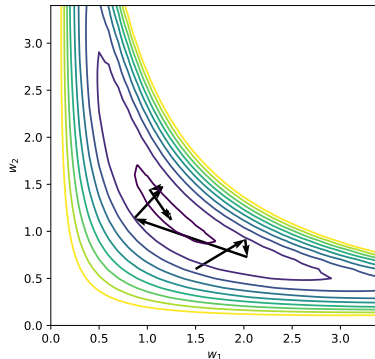
- The eigenvalues of \mathbf{H} determine the curvature of the objective function: Larger λ correspond to higher curvatures in the corresponding direction.
- We can check whether a critical point \mathbf{w}_* (a point with zero gradient) is a saddle point, a maximum or a minimum:
 - if all eigenvalues of \mathbf{H} are positive: \mathbf{w}_* is local minimum
 - if all eigenvalues of \mathbf{H} are negative: \mathbf{w}_* is local maximum
 - if \mathbf{H} has both positive and negative eigenvalues: \mathbf{w}_* is a saddle point.



- In principle, we could use the Hessian matrix in the optimization procedure.
- This is done in the Newton's method: On each iteration we find the minimum of the quadratic approximation:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{H}_t^{-1} \mathbf{g}_t$$

- Can be efficient but not practical for large neural networks: The computational complexity is $\#params^3$.



Part 2. Tricks to improve training of deep neural networks

- We have a deep neural network model that maps input \mathbf{x} to output $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$.
- We have a loss function, for example

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \right\|^2$$

- We have a gradient-descent optimizer

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{g}(\boldsymbol{\theta}_t)$$

- We can compute the gradient $\mathbf{g}(\boldsymbol{\theta})$ efficiently with backpropagation.

1. Training on large data sets
 - Mini-batch training
2. Improved optimizers
 - Momentum method
 - Adam
3. Input normalization
4. Weight initialization
5. Batch normalization

Mini-batch training
(stochastic gradient descent)

- The cost function contains N terms corresponding to the training samples, for example:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{n=1}^N \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \theta) \right\|^2.$$

- Large data sets are redundant: gradient computed on two different parts of data are likely to be similar. Why to waste computations?
- We can compute gradient using only part of training data (a mini-batch \mathcal{B}_j):

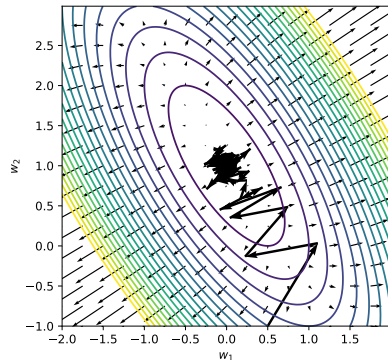
$$\frac{\partial \mathcal{L}}{\partial \theta} \approx \frac{1}{|\mathcal{B}_j|} \sum_{n \in \mathcal{B}_j} \frac{\partial}{\partial \theta} \left\| \mathbf{y}^{(n)} - \mathbf{f}(\mathbf{x}^{(n)}, \theta) \right\|^2$$

- By using mini-batches, we introduce “noise” to the gradient computations, thus the method is called *stochastic gradient descent*.
- *Epoch*: going through all of the training examples once (usually using mini-batch training).

- It is good to shuffle the data between epochs when producing mini-batches (otherwise gradient estimates are biased towards a particular mini-batch split).
- Mini-batches need to be balanced for classes.
- The recent trend is to use as large batches as possible (depends on the GPU memory size).
 - Using larger batch sizes reduces the amount of noise in the gradient estimates.
 - Computing the gradient for multiple samples at the same time is computationally efficient (requires matrix-matrix multiplications which are efficient, especially on GPUs).

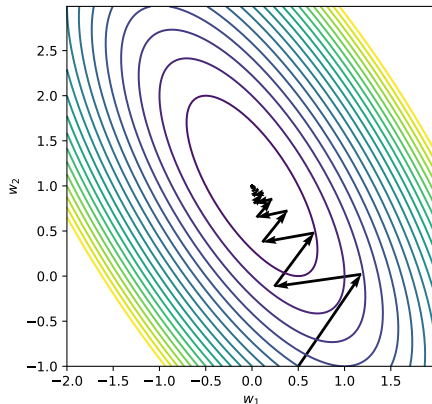
- In mini-batch training, we always use noisy estimates of the gradient. Therefore, the magnitude of the gradient can be non-zero even when we are close to the optimum.
- One way to reduce this effect is to anneal the learning rate η_t towards the end of training.
 - The simplest schedule is to decrease the learning rate after every n updates.
- Another popular way to fine-tune a model is to use exponential moving average of the model parameters:

$$\theta'_t = \gamma \theta'_{t-1} + (1 - \gamma) \theta_t$$



Improved optimization algorithms

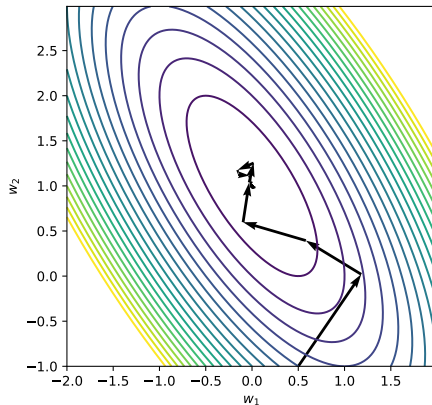
- When the curvature of the objective function substantially varies in different directions, the optimization trajectory of the gradient descent can be zigzagging.
- Momentum method (Polyak, 1964):
 - We would like to move faster in directions with small but consistent gradients.
 - We would like to move slower in directions with big but inconsistent gradients.



- Momentum method: Aggregate negative gradients in momentum \mathbf{m} :

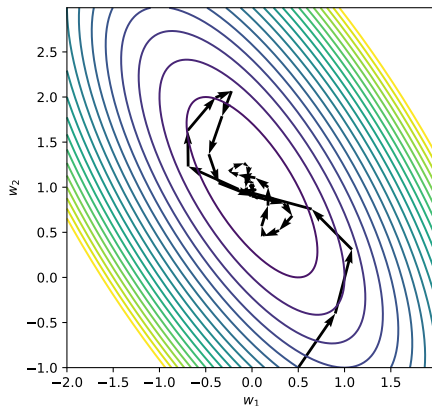
$$\mathbf{m}_{t+1} = \alpha \mathbf{m}_t - \eta_t \mathbf{g}_t$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{m}_{t+1}$$



The intuition behind the momentum method

- A ball moving on the error surface: The location of the ball represents the value of the parameters (w_1 , w_2).
- At $t = 0$, the ball follows the gradient. Once it has velocity, it no longer does steepest descent: Its momentum makes it keep going in the previous direction.
- It damps oscillations in directions of high curvature (by combining gradients with opposite signs) and it builds up speed in directions with a gentle but consistent gradient.
- See [\(Goh, 2017\)](#) for the analysis the convergence of the momentum method.



- The magnitude of the gradient can be very different for different weights and can change during learning. This makes it hard to choose a single global learning rate.
- Rprop (full batch training): Use the sign of the gradient

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \odot \frac{\mathbf{g}_t}{\sqrt{\mathbf{g}_t^2 + \epsilon}}$$

where $\mathbf{g}^2 = \mathbf{g} \odot \mathbf{g}$ and $\frac{\mathbf{a}}{\mathbf{b}}$ is elementwise division.

- Adapt the learning rates η_t individually for each parameter:
 - Increase the step size for a weight multiplicatively (e.g. times 1.2) if the signs of its last two gradients agree
 - Otherwise decrease the step size multiplicatively (e.g. times 0.5)
 - Limit the step sizes
- This escapes from plateaus with tiny gradients quickly.

- Rprop does not work well for mini-batch training:
 - Consider a weight that gets a gradient of +0.1 on nine mini-batches and a gradient of -0.9 on the tenth mini-batch: We want this weight to stay roughly where it is.
 - Rprop would increment the weight nine times and decrement it once by about the same amount (assuming any adaptation of the step sizes is small on this time-scale).
 - So the weight would grow a lot.
- RMSprop: Divide the gradient by a number similar for adjacent mini-batches:

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{v}_t + \epsilon}}$$

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1 - \beta) \mathbf{g}_t^2$$

where we use the exponential moving average of \mathbf{g}_t^2 .

- RMSProp plus the exponential moving average of the gradient:

$$\begin{aligned}\boldsymbol{\theta}_t &\leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t} + \epsilon} \\ \mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t \\ \mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2\end{aligned}$$

- Correct the bias related to starting the estimates from zero:

$$\begin{aligned}\widehat{\mathbf{m}}_t &= \mathbf{m}_t / (1 - \beta_1^t) \\ \widehat{\mathbf{v}}_t &= \mathbf{v}_t / (1 - \beta_2^t)\end{aligned}$$

β_1^t is β_1 to the power of t .

- The update rule is again unit-less.

$$\begin{aligned}\theta_t &\leftarrow \theta_{t-1} - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \\ \mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t \\ \mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2\end{aligned}$$

- In Adam, the effective step size $|\Delta_t|$ is bounded. In the most common case:

$$|\Delta_t| = \left| \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}} \right| \approx \left| \eta \frac{E[g]}{\sqrt{E[g^2]}} \right| \leq \eta \quad \text{because } E[g^2] = E[g]^2 + E[(g - E[g])^2]$$

Thus, we never take too big steps (which can be the case for standard gradient descent).

- At convergence, when we start fluctuating around the optimum: $E[g] \approx 0$ and $E[g^2] > 0$. The effective step size gets smaller. Thus, Adam has a mechanism for automatic annealing of the learning rate.

Input normalization

- Consider solving a linear regression problem (no bias term) with gradient descent

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^N \left(y_n - \mathbf{w}^\top \mathbf{x}_n \right)^2$$

- We know that the convergence of the gradient descent is determined by the properties of the Hessian matrix. Let us compute the Hessian matrix:

$$\begin{aligned} \nabla_{\mathbf{w}} \mathcal{L} &= \frac{2}{2N} \sum_{n=1}^N \left(y_n - \mathbf{w}^\top \mathbf{x}_n \right) (-\mathbf{x}_n) = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top \mathbf{w} - \frac{1}{N} \sum_{n=1}^N y_n \mathbf{x}_n \\ \mathbf{H} &= \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top = \mathbf{C}_x \end{aligned}$$

- We can see that the Hessian is equal to the sample covariance matrix of the inputs.

- Linear regression: For fastest convergence, the covariance matrix of the inputs should be the identity matrix $\mathbf{H} = \mathbf{C}_x = \mathbf{I}$.
- We can achieve this by decorrelating the input components (whitening) using principal component analysis (PCA):

$$\mathbf{x}_{\text{PCA}} = \mathbf{D}^{-1/2} \mathbf{E}^T (\mathbf{x} - \boldsymbol{\mu})$$

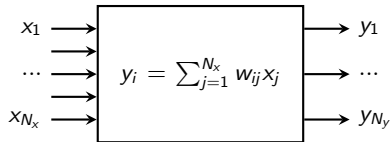
where $\mathbf{E} \mathbf{D} \mathbf{E}^T = \mathbf{C}$ is the eigenvalue decomposition of the covariance matrix of \mathbf{x} .

- Multilayer neural networks are nonlinear models but normalizing the inputs usually improves convergence as well.
 - Simple: Centering+scaling to unit variance of all inputs (so that each component x_i has zero mean and unit variance).
 - More advanced: ZCA (when we want to preserve the original dimensions, e.g., for images)

$$\mathbf{x}_{\text{ZCA}} = \mathbf{E} \mathbf{D}^{-1/2} \mathbf{E}^T (\mathbf{x} - \boldsymbol{\mu})$$

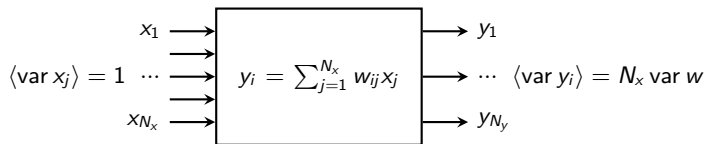
Weight initialization

- Let us consider a linear layer



- It makes sense to initialize weights with random values. For example, we can draw the initial values of the weights from some distribution $p(w)$ with zero mean $\langle w \rangle = 0$.

Variance of signals in the forward computations



- Suppose that the inputs x_j are normalized to have zero mean and unit variance and they are also uncorrelated. Then, the variance of the output signals is

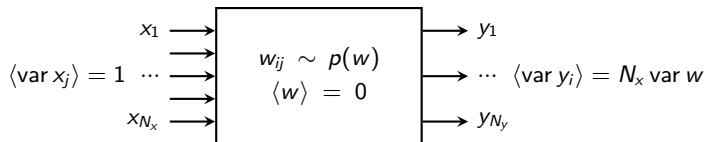
$$\text{var } y_i = \sum_{j=1}^{N_x} w_{ij}^2 \text{var } x_j$$

- Its expectation under the weight (initial) distribution is

$$\langle \text{var } y_i \rangle = \sum_{j=1}^{N_x} \langle w_{ij}^2 \rangle \text{var } x_j = \sum_{j=1}^{N_x} \langle w_{ij}^2 \rangle = N_x \text{var } w$$

where $\text{var } w$ is the variance of the initial weight values.

Variance of signals in the forward computations

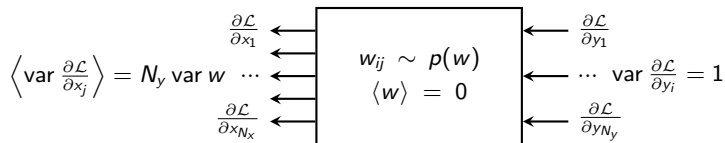


- The variance of y_i can grow (become larger than the variance of the inputs) or decrease depending on N_x and the values of the weights (determined by $\text{var } w$).
- When we stack multiple layers on top of each other: The variance can grow/decay quickly if the weights are too large/small.
- It is a good idea to keep the variance at a constant level: $\langle \text{var } y_i \rangle = \langle \text{var } x_j \rangle = 1$, which means that we should select the distribution $p(w)$ such that

$$\text{var } w = \frac{1}{N_x}$$

Variance of signals in the backward computations

- How about the variance of signals in the backpropagation phase?
- Let us assume that the inputs of the block $\frac{\partial \mathcal{L}}{\partial y_i}$ in the backward phase are also uncorrelated and have unit variance:



- With similar arguments, the expected variance of the outputs is

$$\left\langle \text{var} \frac{\partial \mathcal{L}}{\partial x_j} \right\rangle = N_y \text{var } w$$

and if we want to keep the variance at a constant level, $p(w)$ should be such that

$$\text{var } w = \frac{1}{N_y}$$

- [Glorot and Bengio \(2010\)](#) proposed to keep the balance between the forward and backward variances by choosing the weight distribution $p(w)$ such that

$$\text{var } w = \frac{2}{N_x + N_y}$$

- If we use the uniform distribution $w_{ij} \sim \mathcal{U}[-\Delta, \Delta]$, the variance of the weights is

$$\text{var } w = \langle w_{ij}^2 \rangle = \int_{-\Delta}^{\Delta} w_{ij}^2 p(w_{ij}) dw_{ij} = \int_{-\Delta}^{\Delta} w_{ij}^2 \frac{1}{2\Delta} dw_{ij} = 2 \frac{\Delta^3}{3} \frac{1}{2\Delta} = \frac{\Delta^2}{3}$$

- The proposed scheme is then

$$w_{ij} \sim \mathcal{U} \left[-\frac{\sqrt{6}}{\sqrt{N_x + N_y}}, \frac{\sqrt{6}}{\sqrt{N_x + N_y}} \right]$$

which is perhaps the most popular initialization scheme (called Xavier's initialization).

Batch normalization

- It usually helps if intermediate signals also have zero mean and unit variance.
- Batch normalization layer:
 - Normalize intermediate signals \mathbf{x} to zero mean and unit variance:

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

- The mean and standard deviation computed from the current mini-batch $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}\}$:

$$\boldsymbol{\mu} = \frac{1}{N} \sum_{i=1}^N \mathbf{x}^{(i)} \quad \boldsymbol{\sigma}^2 = \frac{1}{N} \sum_{i=1}^N (\mathbf{x}^{(i)} - \boldsymbol{\mu})^2$$

- The layer can control the mean and the variance of the outputs with two trainable parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$

$$\mathbf{y} = \boldsymbol{\gamma} \odot \tilde{\mathbf{x}} + \boldsymbol{\beta}$$

- [Santurkar et al. \(2018\)](#): BN makes the optimization landscape smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.
- [Bjorck et al. \(2018\)](#): BN primarily enables training with larger learning rates, which is the cause for faster convergence and better generalization.
- Important to remember: BN introduces dependencies between samples in a mini-batch in the computational graph.

- The mean and standard deviation are computed for each mini-batch. What to do at test time when we need to use a trained network for a test example?
- Batch normalization layer keeps track of the batch statistics (mean and standard deviation) during training:

$$\begin{aligned}\bar{\mu} &\leftarrow (1 - \beta)\bar{\mu} + \beta \frac{1}{N} \sum_{i=1}^N \mathbf{x}^{(i)} \\ \overline{\sigma^2} &\leftarrow (1 - \beta)\overline{\sigma^2} + \beta \frac{1}{N} \sum_{i=1}^N (\mathbf{x}^{(i)} - \bar{\mu})^2\end{aligned}$$

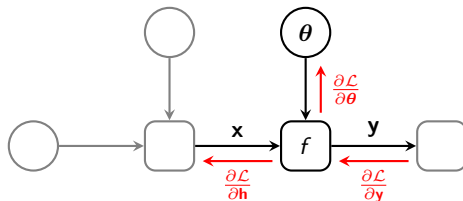
where β is the momentum parameter (note confusing name). It is the running statistics $\bar{\mu}$ and $\overline{\sigma^2}$ that are used at test time.

- Pytorch: If you have a batch normalization layer, the behavior of the network in the training and evaluation modes will be different:
 - Training: Use statistics from a mini-batch, update running statistics $\bar{\mu}$ and $\bar{\sigma}^2$.
 - Evaluation: Use running statistics $\bar{\mu}$ and $\bar{\sigma}^2$, keep $\bar{\mu}$ and $\bar{\sigma}^2$ fixed.

```
model = nn.Sequential(  
    nn.Linear(1, 100),  
    nn.BatchNorm1d(100),  
    nn.ReLU(),  
    nn.Linear(100, 1),  
)  
  
# Switch to training mode  
model.train()  
# train the model  
...  
# Switch to evaluation mode  
model.eval()  
# test the model
```

Home assignment

1. Implement and train a multilayer perceptron (MLP) network in PyTorch.
2. Implement backpropagation for a multilayer perceptron network in numpy. For each block of a neural network, you need to implement the following computations:
 - forward computations $\mathbf{y} = f(\mathbf{x}, \theta)$
 - backward computations that transform the derivatives wrt the block's outputs $\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ into the derivatives wrt all its inputs: $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$, $\frac{\partial \mathcal{L}}{\partial \theta}$



- Chapter 8 of the Deep Learning book.
- G. Hinton, 2012. Overview of mini-batch gradient descent.
- G. Goh, 2017. Why momentum really works.