Aalto University

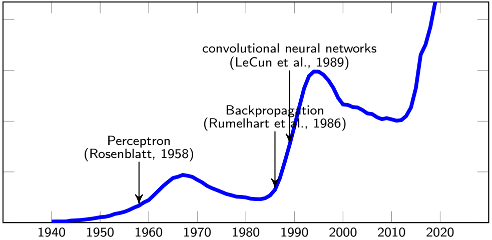
CS-E4890: Deep Learning
Optimization

Alexander Ilin

What did deep learning start only in 2010-2012?

- Many components of deep learning have been invented long time ago. Why did deep learning start only in 2010-2012?

- Geoff Hinton gave four reasons for that:
  - Our labeled datasets were thousands of times too small.
  - Our computers were millions of times too slow.
  - We initialized the weights in a stupid way.
  - We used the wrong type of non-linearity.



Frequency of phrases "cybernetics", "neural networks" and "deep learning" according to Google books.

- Training of deep neural network is a non-trivial optimization problem which requires multiple tricks: input normalization, weight initialization, mini-batch training (stochastic gradient descent), improved optimizers, batch normalization.

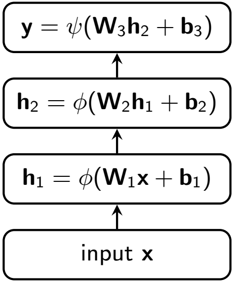
Loss functions

Supervised learning problems

- In this lecture, we will study how to train a neural network to produce desired output $\mathbf{y}$ for a given input $\mathbf{x}$.
- The network is trained using a set of training examples:

  $$\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\}$$

  We change the values of the network parameters to fit to the training data.
- Two most common tasks of supervised learning:
  - classification: the output is discrete (class label)
  - regression: the output a vector of real numbers



$\mathbf{y} = \psi(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$

$\mathbf{h}_2 = \phi(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$

$\mathbf{h}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

input $\mathbf{x}$

Classification problems: One-hot encoding of targets

- Classification tasks: a target can be represented as a one-hot vector $\mathbf{y}$.

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

- For example, for $K = 3$ classes:

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

Classification problems: Model outputs

- We want our neural network 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.
- For example, for $K = 3$ classes, the output of our model is

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \qquad \begin{matrix} f_1 \text{ is the probability that } \mathbf{x} \text{ belongs to class 1} \\ f_2 \text{ is the probability that } \mathbf{x} \text{ belongs to class 2} \\ f_3 \text{ is the probability that } \mathbf{x} \text{ belongs to class 3} \end{matrix}$$

- Thus, we need to make sure that:

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

- Suppose that $\mathbf{h}$ is the output of the last linear layer, for example, for $K = 3$:

$$\mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

  and vector $\mathbf{h}$ does not satisfy the desired conditions.

Classification problems: softmax nonlinearity

- We can transform $\mathbf{h}$ using the following function:

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

which guarantees that f_j can be treated as probabilities because

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

- This nonlinearity is called softmax.
  - If one of the elements h_j is much larger than the rest of the elements:

$$h_j \gg h_i, i \neq j$$

then $\mathbf{f} \approx [0, ..., 0, 1, 0, ..., 0]$, which is a one-hot representation of j, the index of the maximum element of $\mathbf{h}$.

Classification problems: Cross-entropy loss

- For one-hot encoded targets $\mathbf{y}$, it is common to tune the parameters of classifier $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$ by minimizing the following loss function:

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

  thus we maximize the probability of the correct class (there is one non-negative $y_j^{(n)}$ for each example $\mathbf{x}^{(n)}$).
- This loss $\mathcal{L}$ is usually called *cross-entropy loss* in popular frameworks such as PyTorch. The loss can be seen as the cross-entropy between the distribution defined by targets $\mathbf{y}^{(n)}$ and the distribution $\mathbf{f}(\mathbf{x}^{(n)}, \boldsymbol{\theta})$ defined by the output of the network.
- The loss is the negative log-likelihood for a probabilistic model with a categorical (also called multinoulli) distribution for $\mathbf{y}$ whose parameters are given by $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$.

Regression problems: Mean-squared error loss

- 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):

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N n_y} \sum_{n=1}^{N} \sum_{j=1}^{n_y} \left(y_j^{(n)} - f_j(\mathbf{x}^{(n)}, \boldsymbol{\theta}) \right)^2 ,$$

where $y_j^{(n)}$ is the j-th element of $\mathbf{y}^{(n)}$, f_j is the j-th element of the network output $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$,
n_y is the number of elements in $\mathbf{y}^{(n)}$,
N is the number of training samples.

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

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

Analysis of convergence of gradient descent

Toy example

- Consider a *linear regression* problem with two inputs (no bias term for simplicity):

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

- We have a data set with two examples:

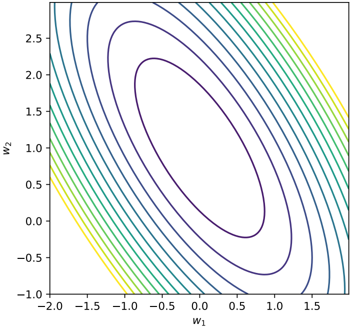
$$\mathbf{x}^{(1)} = (2, 2),\ y^{(1)} = 2 \qquad \mathbf{x}^{(2)} = (2, 0),\ y^{(2)} = 0$$

- We use the mean-squared error (MSE) loss:

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

which is a quadratic function, written in the matrix form:

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

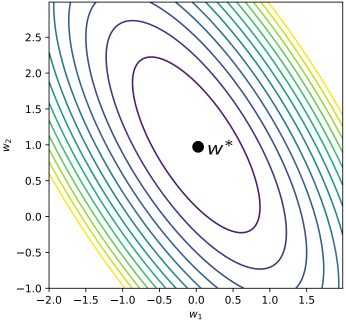


Toy example: optimization landscape

- Quadratic loss for our toy problem:

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

- The contour plot of our loss function contains ellipses concentrated around the global minimum $\mathbf{w}^*$.
- The axes of the ellipses are determined by the eigenvectors of matrix **A**.
- The eigenvalues λ_m of **A** determine the curvature of the objective function: Larger λ_m correspond to higher curvatures in the corresponding direction.

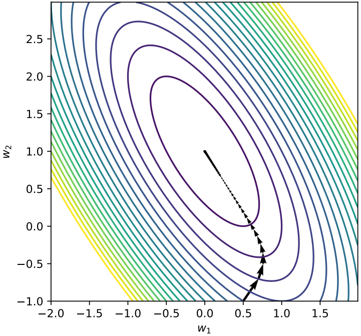


Effect of learning rate

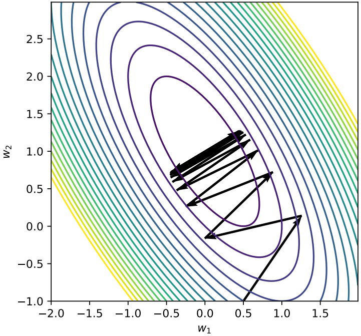
- Suppose that we use gradient descent to find the minimum of the loss:

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

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



small η: too slow convergence



large η: oscillates and can even diverge

Analysis of convergence of gradient descent

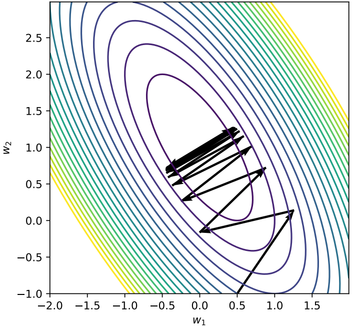
- Because our loss is a quadratic function

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

we can analyze (Goh, 2017):

- how to select the learning rate *optimally*

- how quickly the gradient descent converges with the optimal learning rate.



Analysis of convergence of gradient descent

$$\begin{aligned}
\text{loss:} \quad & \mathcal{L}(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top \mathbf{A}\mathbf{w} - \mathbf{b}^\top \mathbf{w} \\
\text{gradient:} \quad & \mathbf{g}(\mathbf{w}) = \mathbf{A}\mathbf{w}_t - \mathbf{b} \\
\text{gradient descent iterations:} \quad & \mathbf{w}_{t+1} = \mathbf{w}_t - \eta(\mathbf{A}\mathbf{w}_t - \mathbf{b})
\end{aligned}$$

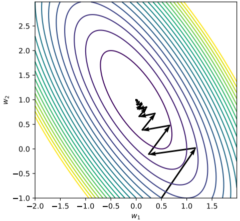
- Let us change the coordinate system such that the new basis is aligned with the eigenvectors of **A**.
  - We compute the eigenvalue decomposition of **A**:

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

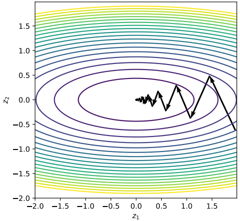
    where **Q** is an orthogonal matrix and λ_m are ordered eigenvalues $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_M$.
  - Then we use **Q** to rotate the coordinate system:

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

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



old system **w**



new system **z**

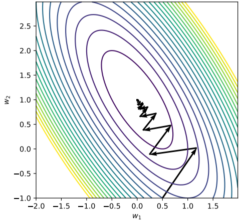
Analysis of convergence of gradient descent

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

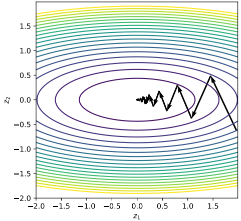
$$\begin{aligned}\mathbf{z}_{t+1} &= \mathbf{Q}^\top(\mathbf{w}_{t+1} - \mathbf{w}_*) = \mathbf{Q}^\top(\mathbf{w}_t - \eta(\mathbf{A}\mathbf{w}_t - \mathbf{b}) - \mathbf{w}_*)\\ &= \mathbf{Q}^\top(\mathbf{Q}\mathbf{z}_t - \eta(\mathbf{A}(\mathbf{w}_* + \mathbf{Q}\mathbf{z}_t) - \mathbf{b}))\\ &= \mathbf{Q}^\top(\mathbf{Q}\mathbf{z}_t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

- 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 $\mathbf{z}$

Analysis of convergence of gradient descent

- Gradient descent for the *m*-th element of **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

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

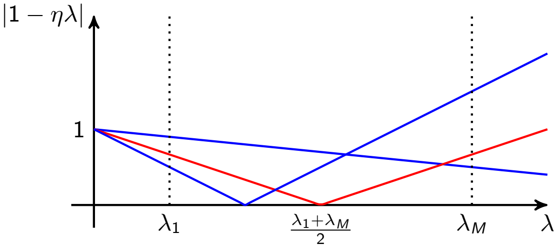
  - for convergence: $|1 - \eta\lambda_m| < 1$
  - ideally: $|1 - \eta\lambda_m| = 0$

This suggests that in order to achieve the best convergence in coordinate z_m, we need to set the learning rate to $\eta = \frac{1}{\lambda_m}$. The problem is that the optimal values of the learning rate is different for the different coordinates of **z** and therefore a value of η that leads to good convergence in one coordinate can cause slow convergence in another coordinate.

Analysis of convergence of gradient descent

- 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

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

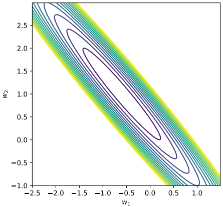
Analysis of convergence of gradient descent

- The rate of convergence for the optimal learning rate is

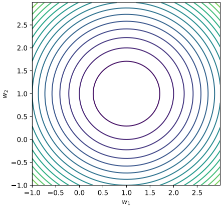
$$\mathrm{rate}(\eta_*) = \left|1 - \left(\frac{\lambda_1 + \lambda_M}{2}\right)^{-1} \lambda_1\right| = \left|\frac{\lambda_1 + \lambda_M - 2\lambda_1}{\lambda_1 + \lambda_M}\right|$$
$$= \frac{\lambda_M - \lambda_1}{\lambda_M + \lambda_1} = \frac{\lambda_M/\lambda_1 - 1}{\lambda_M/\lambda_1 + 1}$$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

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

- $\kappa(\mathbf{A})$ is a measure of how close to singular matrix **A** is.
- For our optimization problem: $\kappa(\mathbf{A})$ is a measure of how poorly gradient descent will perform.



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



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

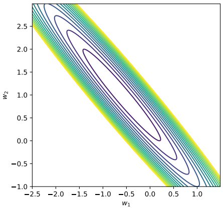
Convergence of gradient descent

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

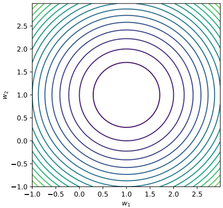
- The optimal learning rate depends on the curvature of the loss.
- The loss has different curvatures in different directions. We need to choose a *single* value of the learning rate to balance the convergence speeds in different directions.
- If we select the learning rate optimally, the rate of convergence of the gradient descent is determined by the condition number of matrix $\mathbf{A}$:

$$\text{rate}(\eta_*) = \frac{\kappa(\mathbf{A}) - 1}{\kappa(\mathbf{A}) + 1}$$

$\text{rate}(\eta_*) = 0$: convergence in one step
$\text{rate}(\eta_*) = 1$: no convergence.



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



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

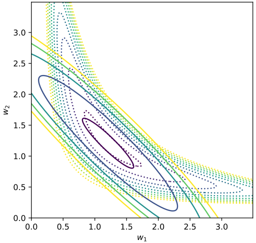
Quadratic approximation

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

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

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

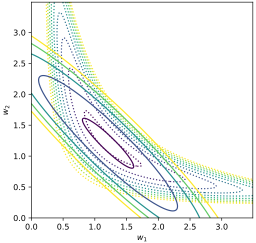
Quadratic approximation

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

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

- **H** is the matrix of second-order derivatives (called Hessian):

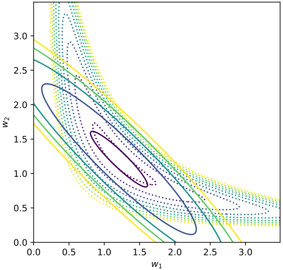
$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial w_M \partial w_1} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_M \partial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



- 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.

On Hessian matrix

- 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.



Input normalization

Simple example: Linear regression

- 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:

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

$$\mathbf{H} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^\top = \mathbf{C}_{\mathbf{x}}$$

- We can see that the Hessian is equal to the second order moment of the data (which is equal to the covariance matrix of the inputs if inputs have zero mean).

Input normalization

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

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

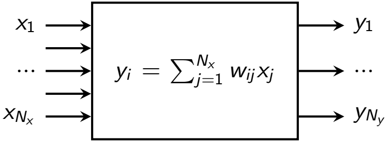
  where $\mathbf{EDE}^\top$ is the eigenvalue decomposition of the covariance matrix of $\mathbf{x}$.
- Multilayer neural networks are nonlinear models but normalizing the inputs usually improves convergence as well.
  - Simple: Centering+scaling to unit variance of all inputs (so that each component x_i has zero mean and unit variance).
  - More advanced: ZCA (when we want the whitened signals to be close to the original ones)

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

Weight initialization

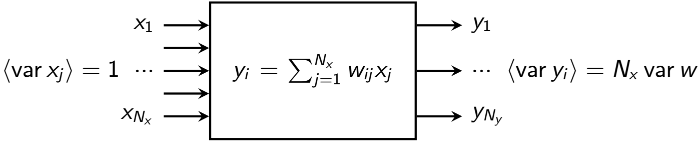
Initialization of weights in a linear layer

- 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



x_1

$\langle \operatorname{var} x_j \rangle = 1 \quad \cdots$

x_{N_x}

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

y_1

$\cdots \quad \langle \operatorname{var} y_i \rangle = N_x \operatorname{var} w$

y_{N_y}

- 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

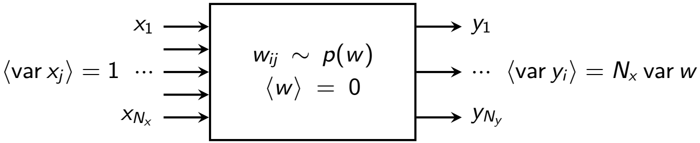
$$\operatorname{var} y_i = \sum_{j=1}^{N_x} w_{ij}^2 \operatorname{var} x_j = \sum_{j=1}^{N_x} w_{ij}^2$$

- Its expectation under the weight (initial) distribution is

$$\langle \operatorname{var} y_i \rangle = \sum_{j=1}^{N_x} \left\langle w_{ij}^2 \right\rangle = N_x \operatorname{var} w$$

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

Variance of signals in the forward computations



x_1

$\langle \operatorname{var} x_j \rangle = 1 \quad \cdots$

x_{N_x}

$w_{ij} \sim p(w)$

$\langle w \rangle = 0$

y_1

$\cdots \quad \langle \operatorname{var} y_i \rangle = N_x \operatorname{var} w$

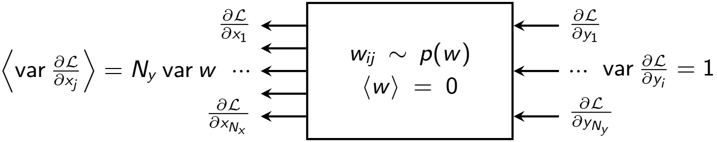
y_{N_y}

- 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 $\operatorname{var} w$).
- 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: $\langle \operatorname{var} y_i \rangle = \langle \operatorname{var} x_j \rangle = 1$, which means that we should select the distribution $p(w)$ such that the variance $\operatorname{var} w$ of the weights is equal to

$$v_{\mathrm{f}} = \frac{1}{N_x}$$

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 uncorrrelated and have unit variance:



$\frac{\partial \mathcal{L}}{\partial x_1}$

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

$\frac{\partial \mathcal{L}}{\partial x_{N_x}}$

$w_{ij} \sim p(w)$

$\langle w \rangle = 0$

$\frac{\partial \mathcal{L}}{\partial y_1}$

$\cdots \quad \operatorname{var} \frac{\partial \mathcal{L}}{\partial y_i} = 1$

$\frac{\partial \mathcal{L}}{\partial y_{N_y}}$

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

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

which means that the gradients can vanish if the initial values of the weights are too small.

- If we want to keep the variance at a constant level, $p(w)$ should be such that the variance $\operatorname{var} w$ of the initial weight values is equal to

$$v_{\mathrm{b}} = \frac{1}{N_y}$$

Xavier's initialization

- To keep the balance between the forward and backward variances, Glorot and Bengio (2010) proposed to use weight distribution $p(w)$ such that $\operatorname{var} w$ is the harmonic mean of v_f and v_b:

$$\operatorname{var} w = \left(\frac{1/v_\mathrm{f} + 1/v_\mathrm{b}}{2} \right)^{-1} = \frac{2}{N_x + N_y}$$

- If $p(w)$ is a uniform distribution $w_{ij} \sim \mathcal{U}\left[-\Delta, \Delta\right]$, the variance of the weights is

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

- 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 intialization scheme (called Xavier's initialization).

Mini-batch training
(stochastic gradient descent)

Mini-batch training

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

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

- 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 $\mathcal{B}_j$):

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

- By using mini-batches, we introduce "noise" to the gradient computations, thus the method is called *stochastic gradient descent*.

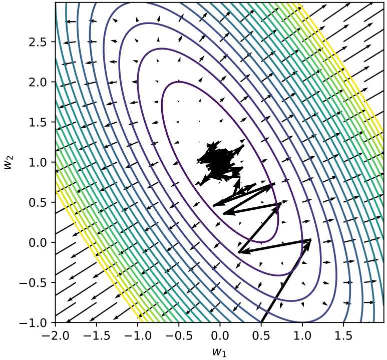
Practical considerations for mini-batch training

- *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).

Model fine-tuning during mini-batch training

- 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:

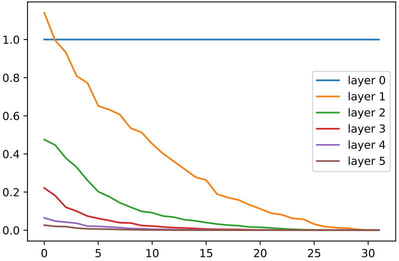
$$\boldsymbol{\theta}'_t = \gamma \boldsymbol{\theta}'_{t-1} + (1 - \gamma)\boldsymbol{\theta}_t$$



Batch normalization

Does depth cause problems for optimization?

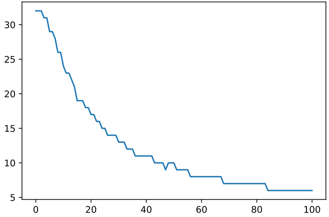
- Suppose that we have a deep neural network with $d = 32$ inputs. For simplicity, let us assume that the network does not have nonlinearities (it is a stack of linear layers).
- We do everything properly:
  - we whiten the inputs
  - we initialize the weights with Xavier's initialization (we initialize bias terms with zeros).
- Let us look at the eigenvalues of the covariance matrices of the signals after the first five layers.
  - Even though the inputs of the network are whitened, the covariance matrix of the intermediate signals quickly becomes ill-conditioned.



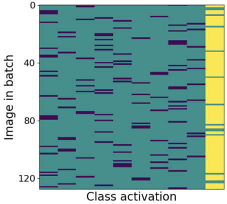
- That means that if we fix the first layers of the network and optimize the last layers with gradient descent, the convergence will be extremely slow.

Rank collapse

- In fact, the problem is even more severe. Let us plot the ranks of the matrices (as implemented in `torch.matrix_rank()`) containing intermediate signals as a function of the layer:
  - The rank decays quickly which means that we lose information on the way: some projections of the original data do not influence the output signal at all.
  - The rank collapse indicates that the direction of the output vector has become independent of the actual input (Daneshmand et al., 2020).
- Bjorck et al. (2018) report that a standard neural network initialized normally consistently predicts one specific class (very right column), irrespective of the input.



Rank of intermediate activations



Output gradients in the final classification layer (Bjorck et al., 2018)

Why does the rank collapse happen?

- The intuition is that the dominant eigenvectors of the weight matrices make the signals rotate more in the same direction. Recall the power iteration method:

  $$\mathbf{v} \leftarrow \frac{\mathbf{W}\mathbf{v}}{\|\mathbf{W}\mathbf{v}\|}$$

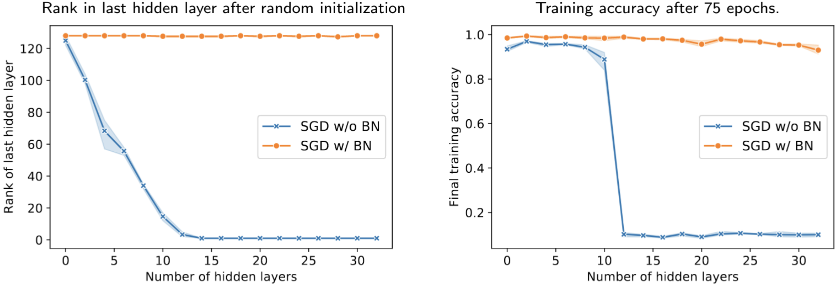
  vector $\mathbf{v}$ converges to the dominant eigenvector of $\mathbf{W}$. In our deep (linear) network

  $$\mathbf{y} = \mathbf{W}_n ... \mathbf{W}_3\mathbf{W}_2\mathbf{W}_1\mathbf{x}$$

  one layer $\mathbf{W}_1\mathbf{x}$ can be viewed as one iteration of the power method without the normalization step.
- Thus, even after the first layer $\mathbf{h}_1 = \mathbf{W}_1\mathbf{x}$, the intermediate signals $\mathbf{h}_1$ become (slightly) correlated even if the inputs $\mathbf{x}$ are whitened.
- When multiple layers are stacked together, the effect becomes very prominent: outputs $\mathbf{y}$ are more determined by the spectral structure of $\mathbf{W}_i$ rather then inputs $\mathbf{x}$. Some data projections simply become invisible in the outputs.
- Applying intermediate nonlinearities does not change the situation.

Rank collapse has negative effect on training

- The rank collapse problem has a severe negative effect on the training procedure:
  - Example (Daneshmand et al., 2020): training an MLP network with ReLU nonlinearities and 128 hidden units in each hidden layer.



  - An MLP network with a small rank in the last hidden layer (depth larger than 12) simply does not train!
  - Bjorck et al. (2018) report that a deep network with standard initialization can have very large gradient magnitudes, which can cause divergence of the training procedure.

Batch normalization (Ioffe and Szegedy, 2015)

- The rank collapse problem is diminished by the trick called *batch normalization*.
- Idea: Since input normalization has positive effect on training, can we also normalize the intermediate signals? The problem is that these signals change during training and we cannot perform normalization before the training.
- The solution is to normalize intermediate signals to zero mean and unit variance in *each training mini-batch*:

1. Compute the means and variances of the intermediate signals $\mathbf{x}$ from the current mini-batch $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}\}$.

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

2. Normalize signals to zero mean and unit variance.

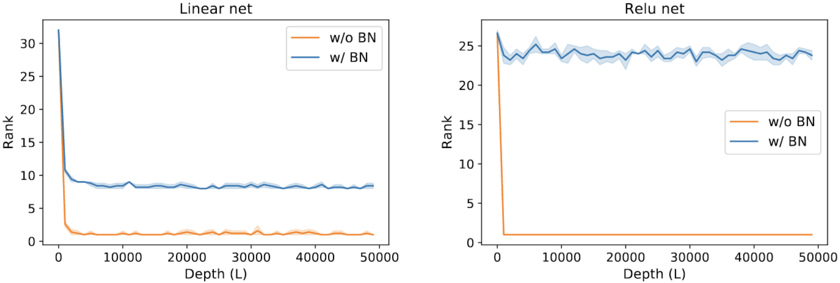
$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

3. Scale and shift the signals with trainable parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$.

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

Batch normalization partially fix the rank collapse problem

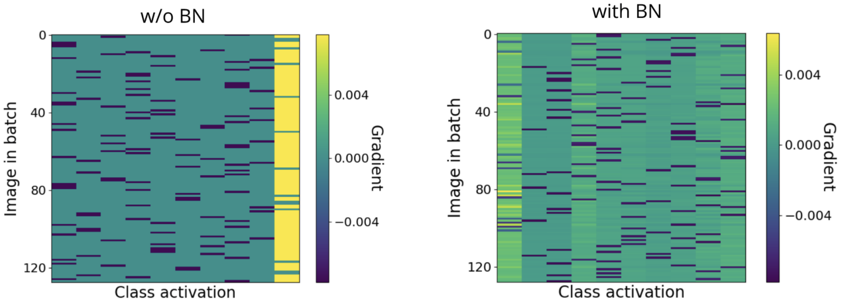
- Daneshmand et al. (2020) show that batch normalization has a positive effect on the rank of the intermediate representations:
  - For a linear model, the rank quickly drops with and without BN.
    Without BN, the rank goes to one but with BN the rank stabilizes at a larger value.
  - For an MLP with ReLU activations, the rank almost does not drop at all!



The rank of the last hidden layer's activation as a function of the total number of layers.

Batch normalization has positive effect on training

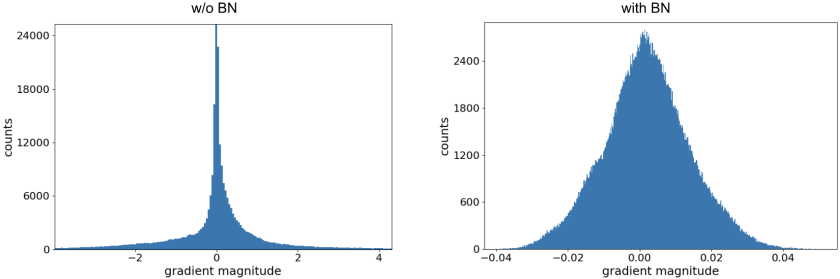
- Bjorck et al. (2018): A heat map of the output gradients in the final classification layer after initialization (the columns correspond to classes and the rows to samples in the mini-batch):



- The unnormalized network (left) consistently predicts one specific class (very right column), irrespective of the input (a consequence of the rank collapse). As a result, the gradients are highly correlated.
- For a batch normalized network (right), the dependence upon the input is much larger.

Batch normalization has positive effect on training

- Bjorck et al. (2018): Histograms over the gradients at initialization for a midpoint layer:



- For the unnormalized network, the gradients are distributed with heavy tails.
  - Large gradient magnitudes can cause divergence of the training procedure.
  - The learning rate has to be small to avoid divergence.
- For the normalized networks the gradients are concentrated around the mean.
  - BN enables training with larger learning rates, which is the cause for faster convergence and better generalization.

Batch normalization: Training and evaluation modes

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

$$\overline{\boldsymbol{\mu}} \leftarrow (1-\alpha)\overline{\boldsymbol{\mu}} + \alpha \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}^{(i)}$$

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

  where α is the momentum parameter (note a confusing name).

- It is the running statistics $\overline{\boldsymbol{\mu}}$ and $\overline{\boldsymbol{\sigma}^2}$ that are used at test time.

Batch normalization: Training and evaluation modes

- 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 $\overline{\boldsymbol{\mu}}$ and $\overline{\boldsymbol{\sigma}^2}$.
  - Evaluation: Use running statistics $\overline{\boldsymbol{\mu}}$ and $\overline{\boldsymbol{\sigma}^2}$, keep $\overline{\boldsymbol{\mu}}$ and $\overline{\boldsymbol{\sigma}^2}$ fixed.
- Important to remember: BN introduces dependencies between samples in a mini-batch in the computational graph.

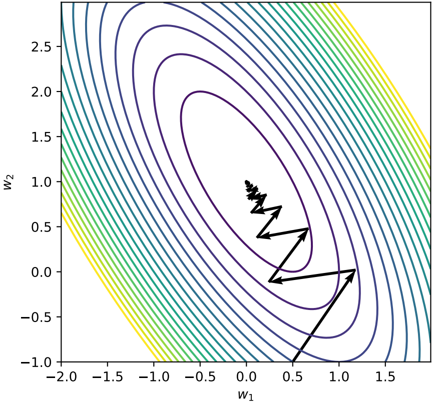
```
model = nn.Sequential(
    nn.Linear(1, 100),
    nn.BatchNorm1d(100),
    nn.ReLU(),
    nn.Linear(100, 1),
)

# Switch to training mode
model.train()
# train the model
...
# Switch to evaluation mode
model.eval()
# test the model
```

Improved optimization algorithms

Problems with gradient descent

- When the curvature of the objective function substantially varies in different directions, the optimization trajectory of the gradient descent can be zigzaging.

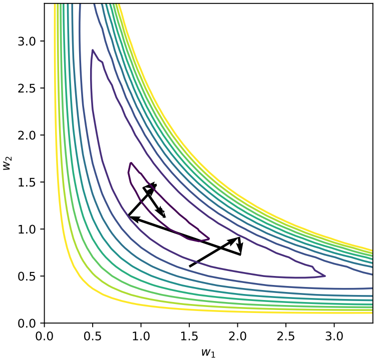


Newton's method

- 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:

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

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

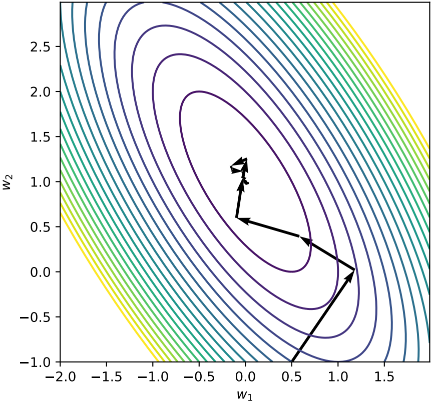


Momentum method (Polyak, 1964)

- Idea:
  - 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.
- Implementation: Aggregate negative gradients in momentum $\mathbf{m}_t$:

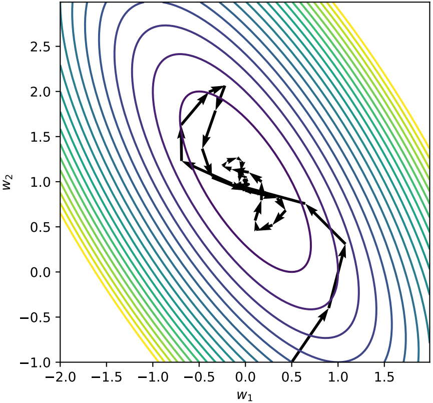
$$\mathbf{m}_{t+1} = \alpha \mathbf{m}_t - \eta_t \mathbf{g}_t$$

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



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.

Slide credit: (Hinton, 2012)

Rprop (Reidmiller and Brau, 1992)

- 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

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \boldsymbol{\eta}_t \odot \frac{\mathbf{g}_t}{\sqrt{\mathbf{g}_t^2 + \epsilon}}$$

  where $\mathbf{g}^2 = \mathbf{g} \odot \mathbf{g}$ and $\frac{\mathbf{a}}{\mathbf{b}}$ is elementwise division.
- Adapt the learning rates $\boldsymbol{\eta}_t$ 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.

Slide credit: (Hinton, 2012)

RMSProp (Hinton, 2012)

- 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:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{v}_t + \epsilon}}$$

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

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

Adam (Kingma and Ba, 2014)

- RMSProp plus the exponential moving average of the gradient:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t} + \epsilon}$$

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2$$

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

$$\widehat{\mathbf{m}}_t = \mathbf{m}_t / (1 - \beta_1^t)$$

$$\widehat{\mathbf{v}}_t = \mathbf{v}_t / (1 - \beta_2^t)$$

  β_1^t is β_1 to the power of t.

- The update rule is again unit-less. Thus, the optimization procedure is not affected by the scale of the objective function.

Why Adam works well

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

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

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

  Thus, we never take too big steps (which can be the case for standard gradient descent).
- We go with the maximum speed (step size η) only if g is the same between updates (mini-batches), that is when the gradients are consistent.
- 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.

Recap

Recap

- Loss functions:
  - Classification: softmax in the output layer and the cross-entropy loss
  - Regression: mean-squared error (MSE) loss

Recap

- Loss functions:
    - Classification: softmax in the output layer and the cross-entropy loss
    - Regression: mean-squared error (MSE) loss
- The learning rate has a major effect on the convergence of the gradient descent.

Recap

- Loss functions:
  - Classification: softmax in the output layer and the cross-entropy loss
  - Regression: mean-squared error (MSE) loss
- The learning rate has a major effect on the convergence of the gradient descent.
- The optimization landscape is determined by the structure of the Hessian matrix.
- Convergence of the gradient descent can be slow in complex landscapes.

Recap

- Loss functions:
  - Classification: softmax in the output layer and the cross-entropy loss
  - Regression: mean-squared error (MSE) loss
- The learning rate has a major effect on the convergence of the gradient descent.
- The optimization landscape is determined by the structure of the Hessian matrix.
- Convergence of the gradient descent can be slow in complex landscapes.
- Input normalization (centering+scaling to unit variance) typically has positive effect on the optimization landscape.

Recap

- Xavier's weight initialization balances the magnitudes in the forward and backward passes.

Recap

- Xavier's weight initialization balances the magnitudes in the forward and backward passes.
- Stochastic gradient descent speeds up training by computing gradients on small portions of data (mini-batches).

Recap

- Xavier’s weight initialization balances the magnitudes in the forward and backward passes.
- Stochastic gradient descent speeds up training by computing gradients on small portions of data (mini-batches).
- Batch normalization diminishes the problem of rank collapse and large gradient magnitudes in deep networks.

Recap

- Xavier's weight initialization balances the magnitudes in the forward and backward passes.
- Stochastic gradient descent speeds up training by computing gradients on small portions of data (mini-batches).
- Batch normalization diminishes the problem of rank collapse and large gradient magnitudes in deep networks.
- Better alternatives to stochastic gradient descent (SGD):
  - SGD with momentum
  - Adam (the most popular optimizer)

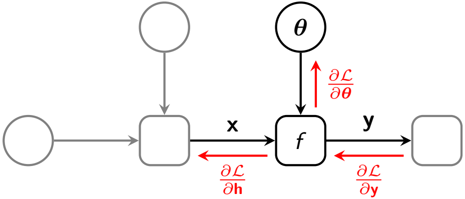
Recap

- Xavier's weight initialization balances the magnitudes in the forward and backward passes.
- Stochastic gradient descent speeds up training by computing gradients on small portions of data (mini-batches).
- Batch normalization diminishes the problem of rank collapse and large gradient magnitudes in deep networks.
- Better alternatives to stochastic gradient descent (SGD):
    - SGD with momentum
    - Adam (the most popular optimizer)
- Adam works well because the step size is bounded and it has a mechanism for automatic annealing of the learning rate.

Home assignment

Assignment `01_mlp`

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 $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\theta})$
   - 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: $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$


$\boldsymbol{\theta}$
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$
$\mathbf{x}$
f
$\mathbf{y}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{h}}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$

Recommended reading

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