

Neural Network Estimation

Session outline

Backpropagation

- A practical explanation

Gradient Descent

- Batch, Stochastic and Mini-Batch Gradient Descent

Momentum

- Accelerating Gradient Updates

Learning rate

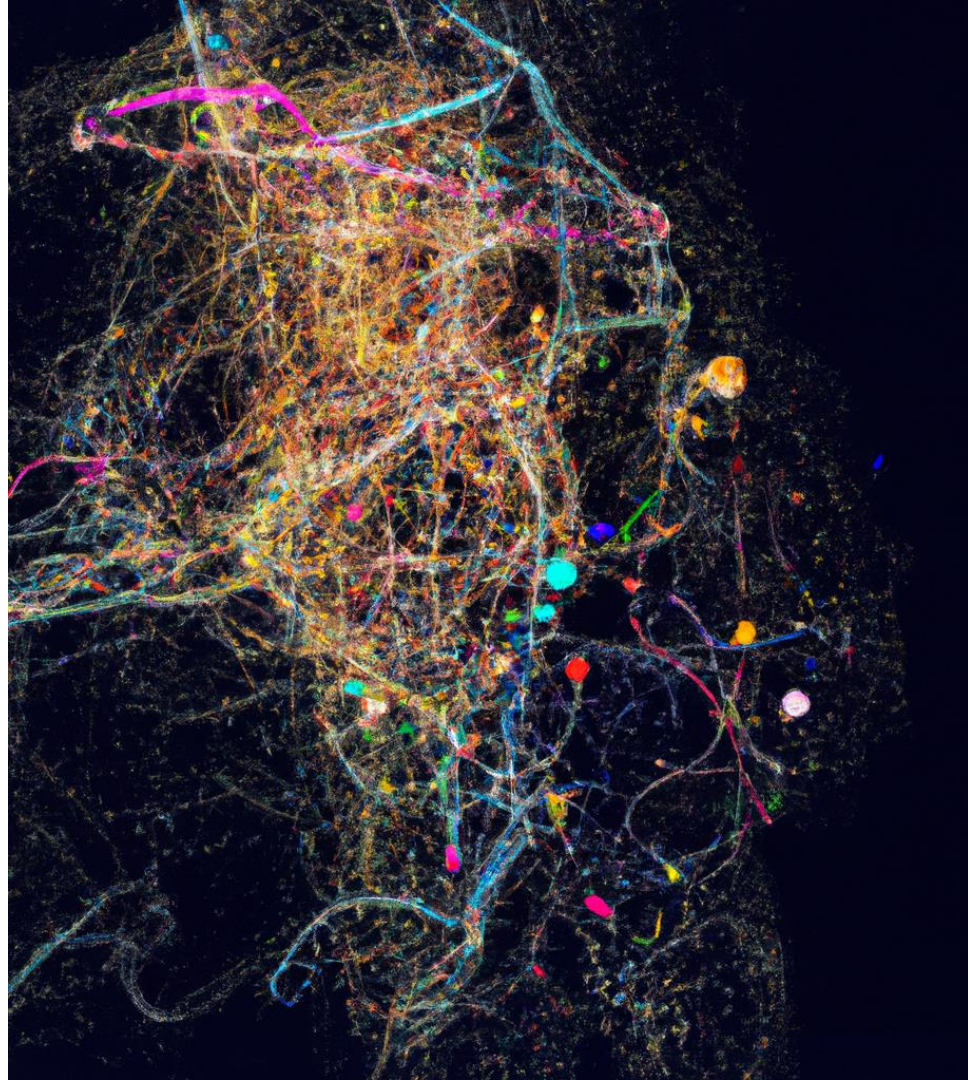
- Adaptive learning

Hyperparameters

- Meta-learning

Backpropagation

A practical explanation



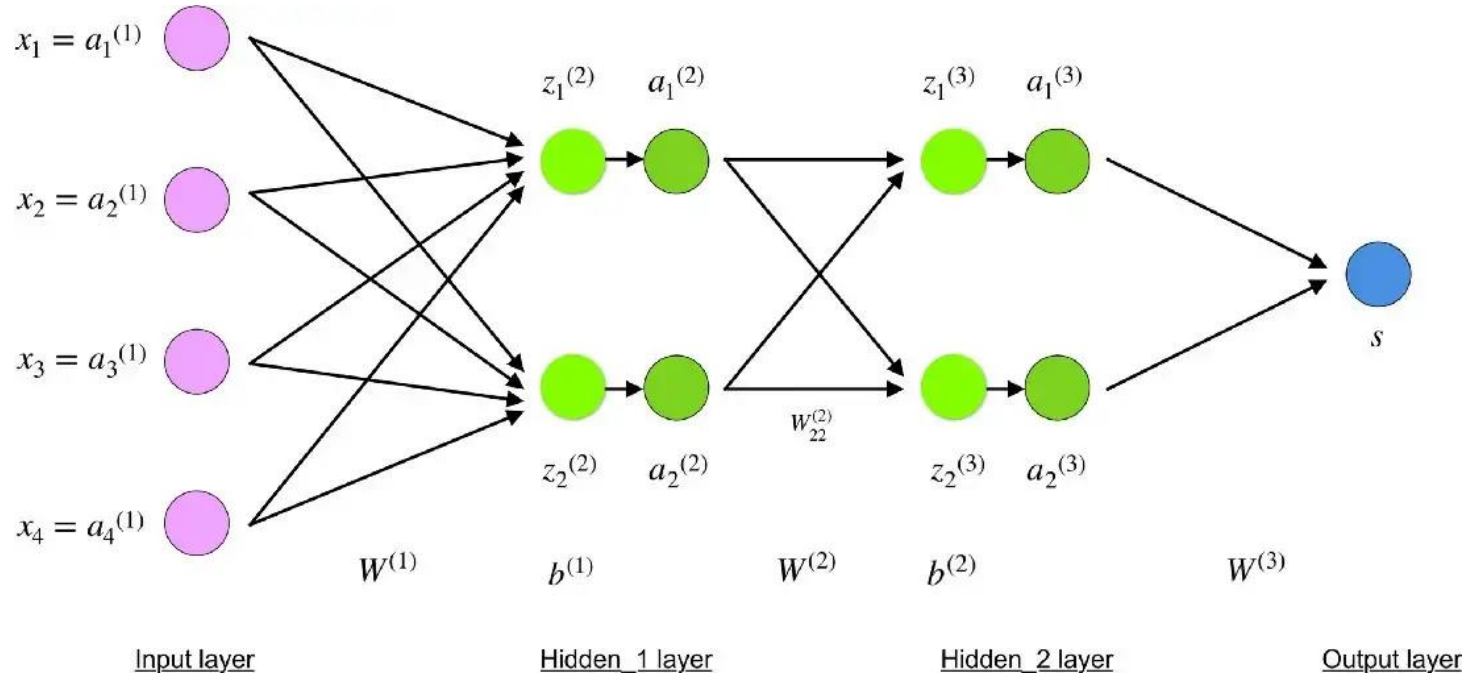
Backpropagation explained

A practical example

- Backpropagation is an algorithm that is used to train a neural network by applying the **chain rule**. In this process, after the input data has been processed by the network in a forward pass, the backpropagation algorithm performs a backward pass through the network and adjusts the weights and biases of the model in order to improve the accuracy of the model's predictions.
- It uses the gradient of the loss function with respect to the weights to determine the direction in which the weights should be adjusted. The weights are adjusted in the opposite direction of the gradient, which is why it is called "backpropagation"
- Backpropagation is an important part of many machine learning algorithms, and it is widely used in various applications including image recognition, natural language processing, and speech recognition.
- Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. *Nature* 323, 533–536 (**1986**). <https://doi.org/10.1038/323533a0>

Backpropagation explained

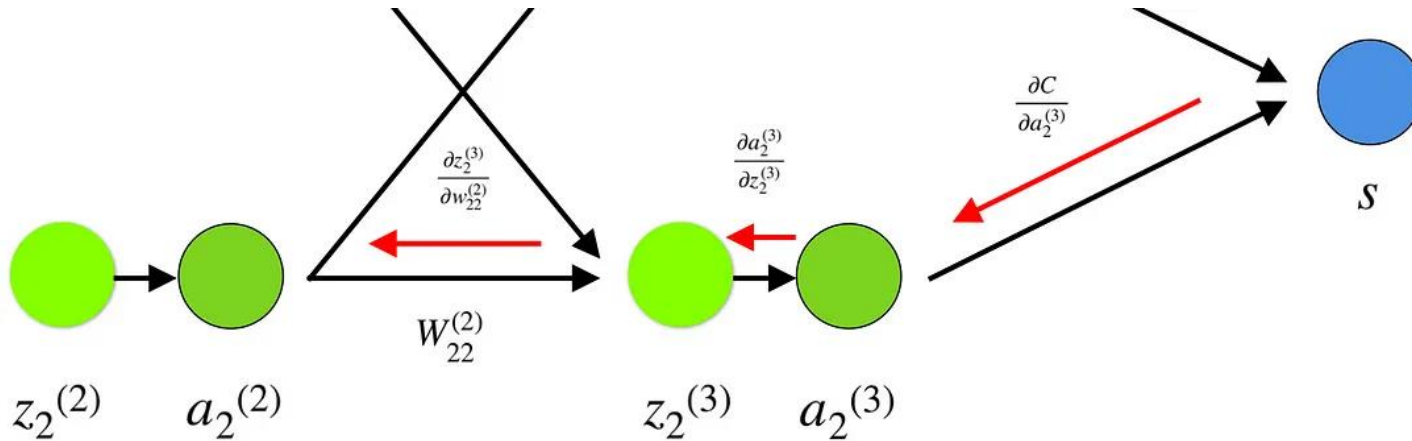
A practical example



- In the forward pass, the input data is passed through the network and the output is calculated
- In the backward pass, the error is calculated, and the weights are updated based on the error

Backpropagation explained

A practical example



Computing the update rule for $W_{22}^{(2)}$ requires to calculate three gradients

Gradient descent

Batch, stochastic & mini-batch



Nomenclature

Functions and variables

Observations

- $\mathbf{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R}$ with $i \in \{1, \dots, n\}$ and n the number of observations
- For binary classification problems $y_i \in \{0,1\}$

Network parameters

- $\boldsymbol{\theta} = (\theta_p)_{p \in \{1, \dots, P\}}$ with $p \in \mathbb{N}$ (e.g., weights and biases)

Loss function

- $l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$ with $i \in \{1, \dots, n\}$
- Example: $l(y, \hat{y}) = (y - \hat{y})^2$ quadratic loss

Goodness-of-fit (cost) function

- $C_n(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$

Batch Gradient Descent (BGD)

All observations

Gradient calculation

- $g_t = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} l(y_i, f(x_i, \theta))$
- Requires all observations

Update rule

- $\theta^{t+1} = \theta^t - \gamma g_t$
- with γ the learning rate

Remarks

- BGD is the most natural way to calculate the gradient
- g_t is the gradient of an unbiased estimator of the expected risk $E_{\theta}[C_n(\theta)]$
- With $n \rightarrow \infty$ the computational time becomes unrealistic
- $C_n(\theta)$ being non-convex in general, BGD is highly reliant on initial conditions as we might hit one of the local minima if starting far from the global minimum

Stochastic Gradient Descent (SGD)

One observation

Gradient calculation

- $g_t = \nabla_{\theta} l(y_{i(t)}, f(x_{i(t)}, \theta))$
- Requires one observation with $i(t)$ uniformly sampled in $\{1, \dots, n\}$ at each iteration t

Update rule

- $\theta^{t+1} = \theta^t - \gamma g_t$
- with γ the learning rate

Remarks

- SGD requires less computations than BGD with about n times more gradient steps taken for each epoch
- Its stochastic behaviour enables the gradient descent to evade local minima
- g_t is also an unbiased estimator of the expected risk but with greater variance than BGD
- SGD can lead to noisy gradient estimates

Mini-Batch Gradient Descent (MBGD)

Subset of observations

Gradient calculation

- $g_t = \frac{1}{\#B_t} \sum_{i \in B_t} \nabla_{\theta} l(y_i, f(x_i, \theta))$
- Requires a subset of observations belonging to the mini-batch B_t

Update rule

- $\theta^{t+1} = \theta^t - \gamma g_t$
- with γ the learning rate

Remarks

- MBGD is the most common way to compute gradient descent for backpropagation
- g_t is still the gradient of an unbiased estimator of the expected risk with larger variance than BGD but smaller variance than SGD
- Provides a good setup for parallel computing

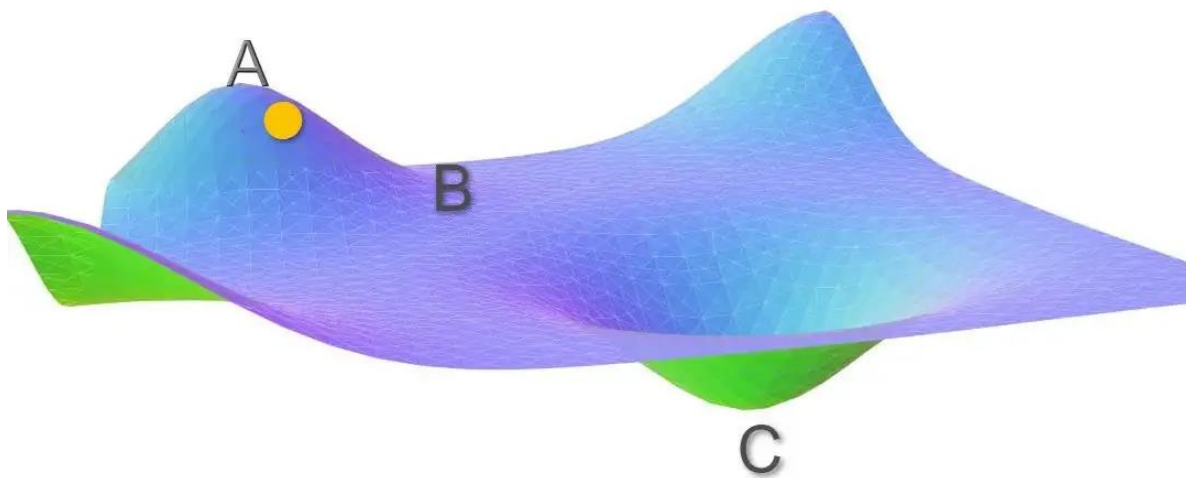
Momentum

Accelerate Gradient Updates



Valleys and saddle points

Zero updates



At saddle points or valleys in the cost function, the gradient is either very small or non-existent, resulting in minimal or no changes to the weights. This can cause the network to become stuck and prevent further learning.

Momentum

Exponential moving average

General idea

- To overcome valley issues, one idea is to keep the « momentum » of the gradient descent almost as if it had some kind of inertia
- For that, we use an exponential moving average of the gradient with a smoothing parameter α which is usually set to be equal to 0.9

Update step with momentum

- $\theta^{t+1} = \theta^t + \mathbf{u}_t$ with $\mathbf{u}_t = \alpha \mathbf{u}_{t-1} - \gamma \mathbf{g}_t$
- \mathbf{u}_t is called the velocity of the gradient descent
- \mathbf{u}_t is a vector with norm representing the “speed” at which the model parameters change during the learning process and with the same direction as the parameters’ variations

Momentum

Exponential moving average

A practical example

- For a (small) constant value of the gradient $g_t = g$ for a large number m of subsequent iterations we would get the following updates
- Without momentum: $\theta^{t+1} = \theta^t - \gamma g \approx \theta^t$
- With momentum: $\theta^{t+1} = \theta^t + \mathbf{u}_t$ with $\mathbf{u}_t = -\frac{\gamma}{1-\alpha} g$ for $m \gg 1$

Remarks

- \mathbf{u}_t is a vector with norm representing the “speed” at which the model parameters change during the learning process and with the same direction as the descent
- This momentum term enables to go through local obstacles

Momentum

One remark

Polyak's Momentum (1964)

- We can rewrite the momentum update step previously explained as follows:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \mathbf{u}_t \text{ with } \mathbf{u}_t = \alpha(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}) - \gamma \mathbf{g}_t$$

Remark

- This algorithm was first introduced in the paper "Some methods of speeding up the convergence of iteration methods" by Boris Polyak in 1964, and it was found to be effective in various optimization problems particularly in the case of non-convex functions

Momentum

A variation on Polyak's momentum

Nesterov's Momentum

- Inspired from Polyak's momentum, it uses a slightly different update rule that incorporates a "lookahead" term that predicts the position of the parameter a few steps ahead in the optimization process.
- The only modification occurs during the calculation of \mathbf{g}_t

$$\mathbf{g}_t = \frac{1}{\#B_t} \sum_{i \in B_t} \nabla_{\boldsymbol{\theta}} l(y_i, f(\mathbf{x}_i, \boldsymbol{\theta} + \alpha \mathbf{u}_{t-1}))$$

Update step

- The update step remains the same as well as the formula for the velocity
- $\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \mathbf{u}_t$ with $\mathbf{u}_t = \alpha \mathbf{u}_{t-1} - \gamma \mathbf{g}_t$

Momentum

Nesterov's momentum bound

A bound for the rate of convergence

- The bound for the rate of convergence of Nesterov's momentum is $O\left(\frac{1}{k^2}\right)$, which means that the distance between the current solution and the optimal solution decreases at a rate of $\frac{1}{k^2}$, where k is the number of iterations (for convex problems)
- The bound for the rate of convergence of GD with a fixed learning rate is $O\left(\frac{1}{k}\right)$

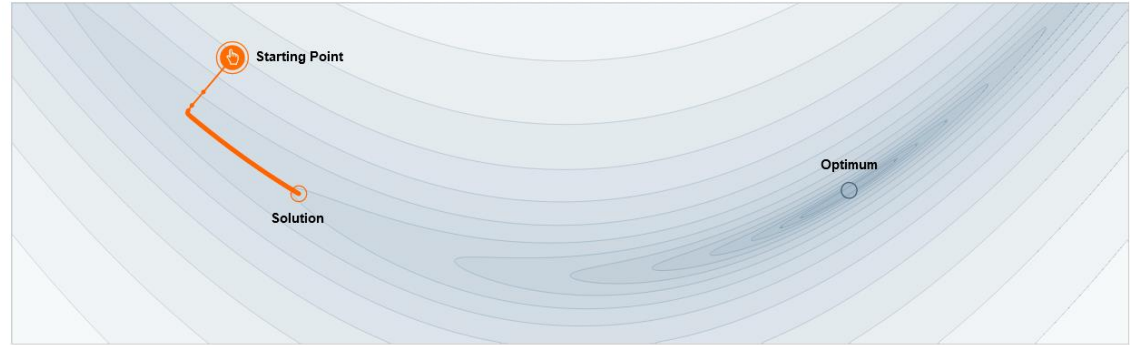
Remarks

- In practice, Nesterov's momentum can converge much faster than the rate of $O\left(\frac{1}{k^2}\right)$ on some problems as it can escape saddle points more effectively
- Nesterov's momentum was introduced by Yurii Nesterov in 1983 in the paper "A method for unconstrained convex minimization problem with the rate of convergence $O\left(\frac{1}{k^2}\right)$ "

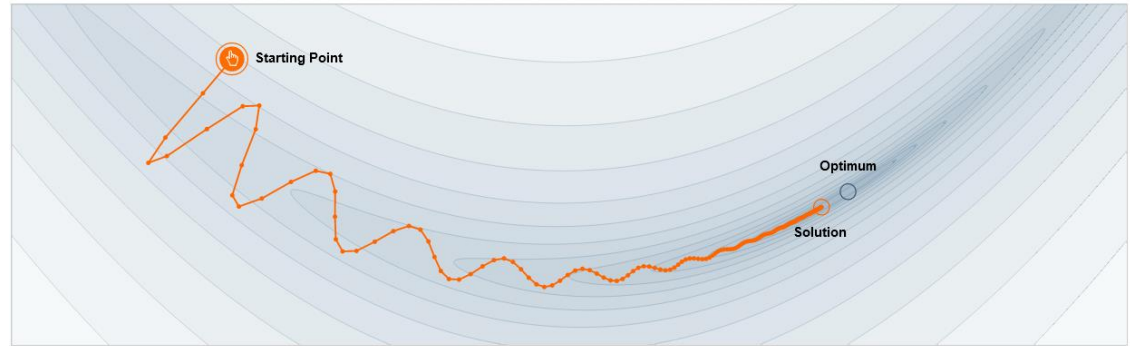
Momentum

A visual explanation

Update without Momentum



Update with Polyak's Momentum ($\alpha = 0.9$)



Learning rate

Adaptive learning



Adaptive learning rates

The importance of adaptive methods

- Adaptive learning rates are used to adjust the step size of the gradient descent optimization algorithm in neural networks.
- The goal is to find the optimal learning rate that allows the model to converge to a good solution quickly and efficiently.
- Without adaptive learning rates, the learning rate would need to be set manually, which can be difficult and time-consuming.
- A fixed learning rate may not be optimal for all parts of the training process, leading to slow convergence or suboptimal solutions.
- Adaptive learning rates can automatically adjust the step size during training, which can help the model converge faster and reach a better solution.

New update rule

- $\theta^{t+1} = \theta^t - \gamma_t g_t$
- with γ_t the learning rate

Adagrad

Adaptive learning rates

General form

- We introduce a new term r_t which accumulates the squared value of the gradients

$$r_t = r_{t-1} + g_t \odot g_t \text{ with } \odot \text{ the Hadamard product (term by term multiplication)}$$

- with the following update step

$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{r_t}} \odot g_t \text{ with } \delta \text{ a stabilizing term to avoid dividing by 0 (usually } \delta = 10^{-8}\text{)}$$

Remarks

- γ does not particularly need to be tuned (usually γ is set to 0.01)
- The per-parameter learning rate is commensurate to the historical gradient
- Parameters that have received larger gradients in the past receive smaller learning rates
- Tends to converge fast in early iterations but may become too aggressive
- r_t increases unboundedly (sum of positive terms) which can lead to very small learning rates

RMSProp

Adaptive learning rates

General form

- Like AdaGrad, but with the addition of an exponential decay applied to the historical gradients being accumulated.

$$r_t = \rho r_{t-1} + (1 - \rho) g_t \odot g_t$$

- with the following update step

$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{r_t}} \odot g_t$$

Remarks

- RMSprop is an adaptive learning rate method proposed by Geoff Hinton in a Coursera class
- Its purpose is to overcome the drastically diminishing learning rates of Adagrad
- It divides the learning rate by an exponentially decaying average of squared gradients.
- Hinton suggests ρ to be set to 0.9 and γ to 0.001

AdaDelta

Adaptive learning rates

General form

- AdaDelta was introduced around the same time as RMSProp to improve Adagrad with the following update step

$$\theta_{t+1} = \theta_t - \frac{\gamma \sqrt{\delta + s_{t-1}}}{\delta + \sqrt{r_t}} \odot g_t$$

- with

$$\begin{aligned} r_t &= \rho r_{t-1} + (1 - \rho) g_t \odot g_t \\ s_t &= \rho s_{t-1} + (1 - \rho) (\theta_{t+1} - \theta_t) \odot (\theta_{t+1} - \theta_t) \end{aligned}$$

Remarks

- $(\theta_{t+1} - \theta_t)$ is the gradient step
- The numerator enforces larger steps in directions where large steps were previously made
- The denominator is the same as before and contributed to decrease the learning rate

Adam

Adaptive learning rates

General form

- Adam was introduced in 2015 and can be seen as a version of RMSProp with momentum

$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{\hat{\mathbf{r}}_t}} \odot \hat{\mathbf{s}}_t$$

- with

$$\hat{\mathbf{r}}_t = \frac{\mathbf{r}_t}{1 - (\rho_1)^t}, \quad \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - (\rho_2)^t}$$

$$\mathbf{r}_t = \rho_1 \mathbf{r}_{t-1} + (1 - \rho_1) \mathbf{g}_t \odot \mathbf{g}_t, \quad \mathbf{s}_t = \rho_2 \mathbf{s}_{t-1} + \mathbf{g}_t$$

Remarks

- In addition to storing an exponentially decaying average of past squared gradients (like Adadelta and RMSprop), Adam also keeps an exponentially decaying average of past gradients like momentum
- It behaves like a heavy ball with friction, which prefers flat minima in the error surface
- The authors propose default values of 0.999 for ρ_1 , 0.9 for ρ_2 , and 10^{-8} for δ
- Adam is one of the default optimizers in deep learning

Adamax

Adaptive learning rates

General form

- Adamax is a slightly modified version of Adam with the following update step

$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + v_t} \odot \hat{\mathbf{s}}_t$$

- with

$$v_t = \max(\rho_1 v_{t-1}, |\mathbf{g}_t|) \text{ (constrained infinity norm of } \mathbf{g}_t \text{)}$$

Remarks

- More efficient method than Adam from a computational perspective
- Can obtain similar results as Adam

Other important optimisers

Adaptive learning rates

Nadam

- While Adam can be seen as RMSProp with Polyak's momentum we can define Nadam which can be seen as RMSProp with Nesterov's momentum
- Nadam combines the advantages of both Nesterov momentum and RMSProp, which can result in more robust and stable optimization performance than Adam in some cases

AMSGrad

- AMSGrad uses a running maximum of the squared gradient to update the learning rate, rather than the running average used by Adam, which allows AMSGrad to have a more stable learning rate and faster convergence in some cases
- It attempts to address the oscillation and slow convergence issues that Adam can face

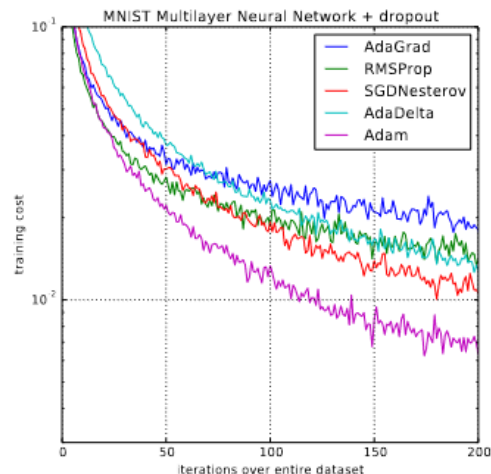
Comparison of Optimiser performances

Adaptive learning rates

Table 10. Comparison of the algorithms on Kaggle Flowers for reconstruction.

Algorithm	CAE		Denoising CAE	
	CAE-1 Loss	CAE-2 Loss	CAE-1 Loss	CAE-2 Loss
SGD	0.0359	0.0385	0.0367	0.0347
SGD - momentum	0.0217	0.0220	0.0320	0.0217
SGD - Nesterov	0.0217	0.0214	0.0230	0.0219
AdaGrad	0.0298	0.0179	0.0254	0.0234
AdaDelta	0.0220	0.0180	0.0236	0.0203
RMSProp	0.0202	0.0159	0.0216	0.0179
Adam	0.0144	0.0121	0.0269	0.0148
AdaMax	0.0205	0.0133	0.0209	0.0164
Nadam	0.0170	0.0138	0.0208	0.0166
AMSGrad	0.0145	0.0122	0.0270	0.0146

“A Comparison of Optimization Algorithms for Deep Learning”, Soydaner D. (2020)



“Adam: A Method For Stochastic Optimization”, Kingma, D. P., & Ba, J., ICLR (2015)

Learning rate scheduling

Adaptive learning rates

Beyond adaptive learning rate methods

- It was found empirically that it can be useful to « anneal » the learning rate over time
- In other words, we can also specify a way for the learning rate to evolve over time regardless of the data but only based on the number of iterations

Examples

- Step decay (dividing the learning rate after a certain number of iterations) if $t \equiv 0[10]$ then $\gamma_t = \frac{\gamma_{t-10}}{10}$ otherwise $\gamma_t = \gamma_{t-1}$
- Exponential decay: $\gamma_t = \gamma_0 \exp(-kt)$ with γ_0 and k hyperparameters
- $\frac{1}{t}$ decay: $\gamma_t = \frac{\gamma_0}{1+kt}$ with γ_0 and k hyperparameters