Gradient Descent Based Optimizers

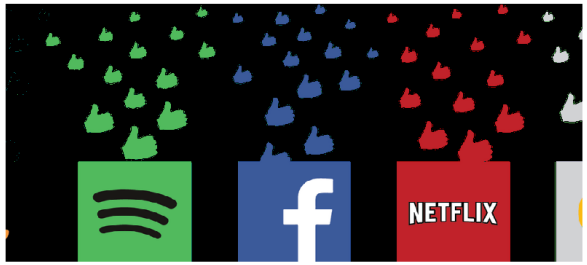
Falak Shah

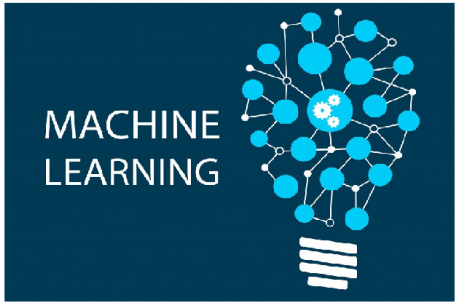
Infocusp Innovations Private Limited

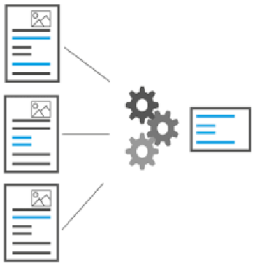
Scipy Conference, IIT Mumbai, 2018

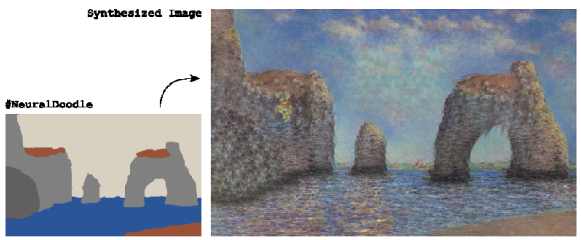
Outline

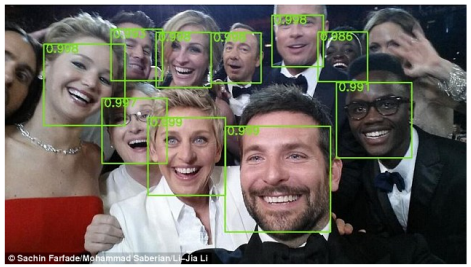
Machine Learning Applications



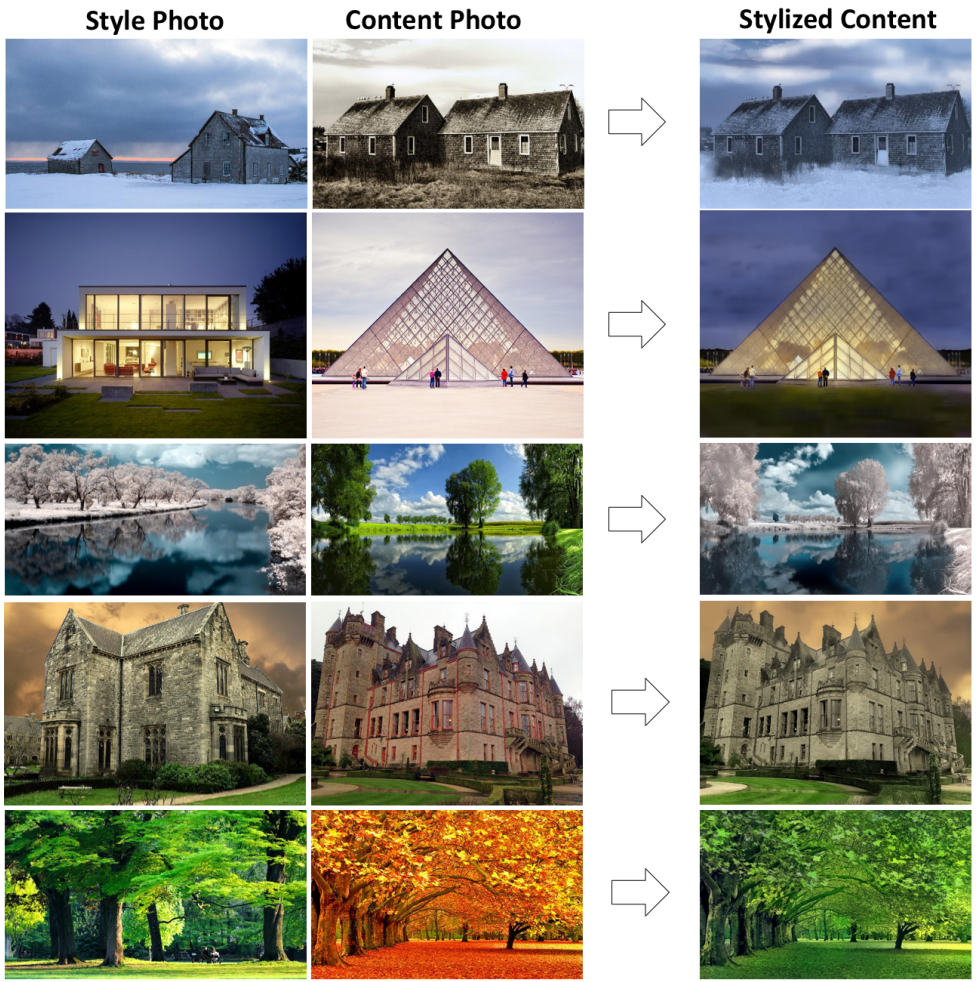








Machine Learning Applications



Nvidia's fast style transfer applying style of one image onto another

What does any ML/ DL algorithm do?

- Supervised, unsupervised, semi-supervised, reinforcement learning
- Today's focus on supervised learning algorithms
- Try to learn a mapping from an input to output
- Can also be understood as minimizing a loss function
- Example: Classifying MNIST digits

Optimizers drive the DL algorithms

- Optimizers are at the heart of the DL algorithms
- Responsible for the updates - starting from a random solution to the optimal value
- Improvement in optimizers implies effect on all DL algorithms
- Improvement also could save you resources
- Even today, top ML conferences see papers on optimizers- ICLR, ICML

Linear Regression

- One of the simplest ML algorithms
- Trying to find a linear mapping from input to output

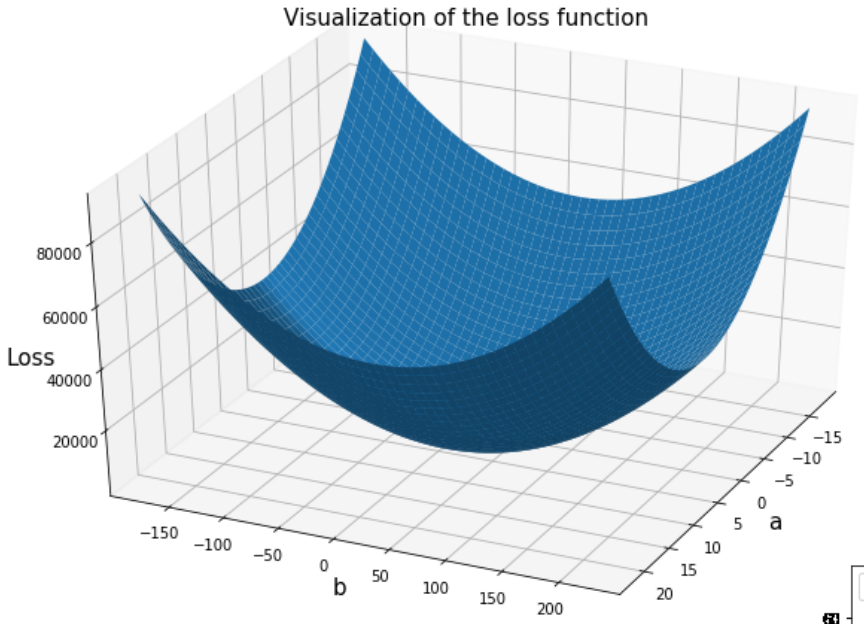
$$y = ax + b$$

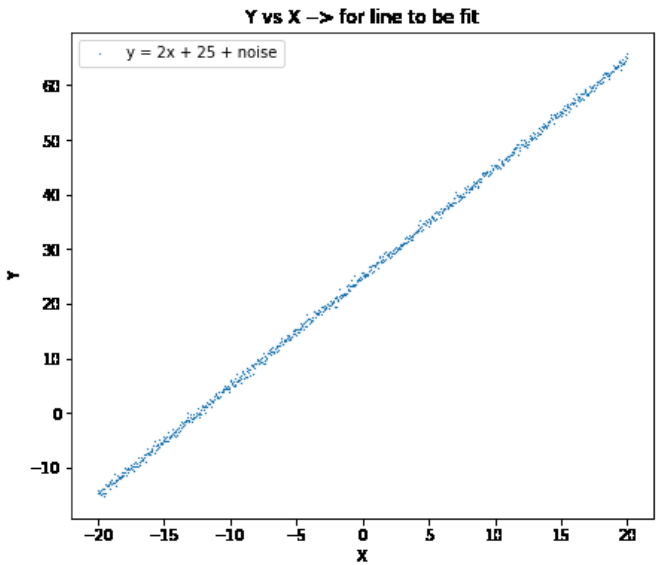
$$J = \frac{1}{2}\sum_{i=1}^{n}(y_i - (ax_i + b))^2$$

$$(a_{opt}, b_{opt}) = \underset{a,b}{\operatorname{argmin}}(J)$$

- Vectorize it to form $Y = XA$ where $Y_{n\times 1}$, $X_{n\times f}$ and $A_{f\times 1}$, n is number of samples and $(f-1)$ is number of features

Data and loss visualization





Why can't we just "solve" it

- Inverse of a matrix

$$Y = XA$$

$$A = X^{-1}Y$$

- Or pseudo inverse

$$A = (X^T X)^{-1} X^T Y$$

- Same solution above if you try setting gradient of loss function above to 0
- Dimensions become prohibitively large
- Mappings much more complex than linear regression

Actual problem much more complex

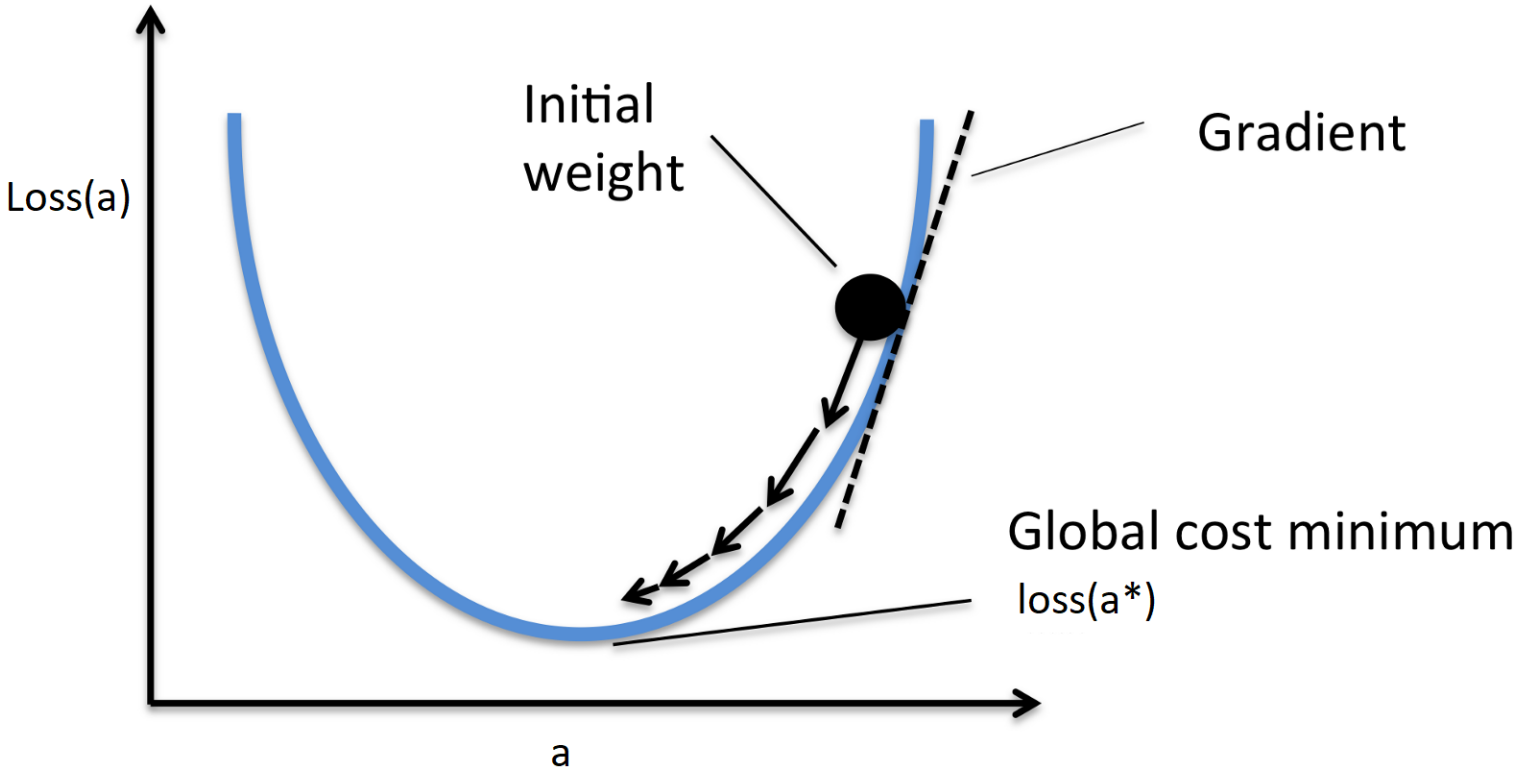
- Many more maxima and minima
- Many more dimensions (in order of millions for neural networks)
- Much more complex mapping from input to output

$$y = f(Wx + b)$$

where (W, b) are to be learnt. f is a non linear function.

- Above is one layer in the simplest form of neural network. Several such stacked layers in practice.
- An analogy is trying to find the lowest point in an unknown terrain

Gradient Descent Intuition



The solution: Gradient Descent

Pseudo code

- Initialize the parameters θ randomly
- Calculate the gradients $\frac{\partial J}{\partial \theta}$
- Update the weights θ by an amount proportional to $\frac{\partial J}{\partial \theta}$
- Repeat till *loss* stops reducing or other pre-defined termination criteria is met

Gradient descent for line fitting

$$J = \frac{1}{2}\sum_{i=1}^{n}(y_i - (ax_i + b))^2$$

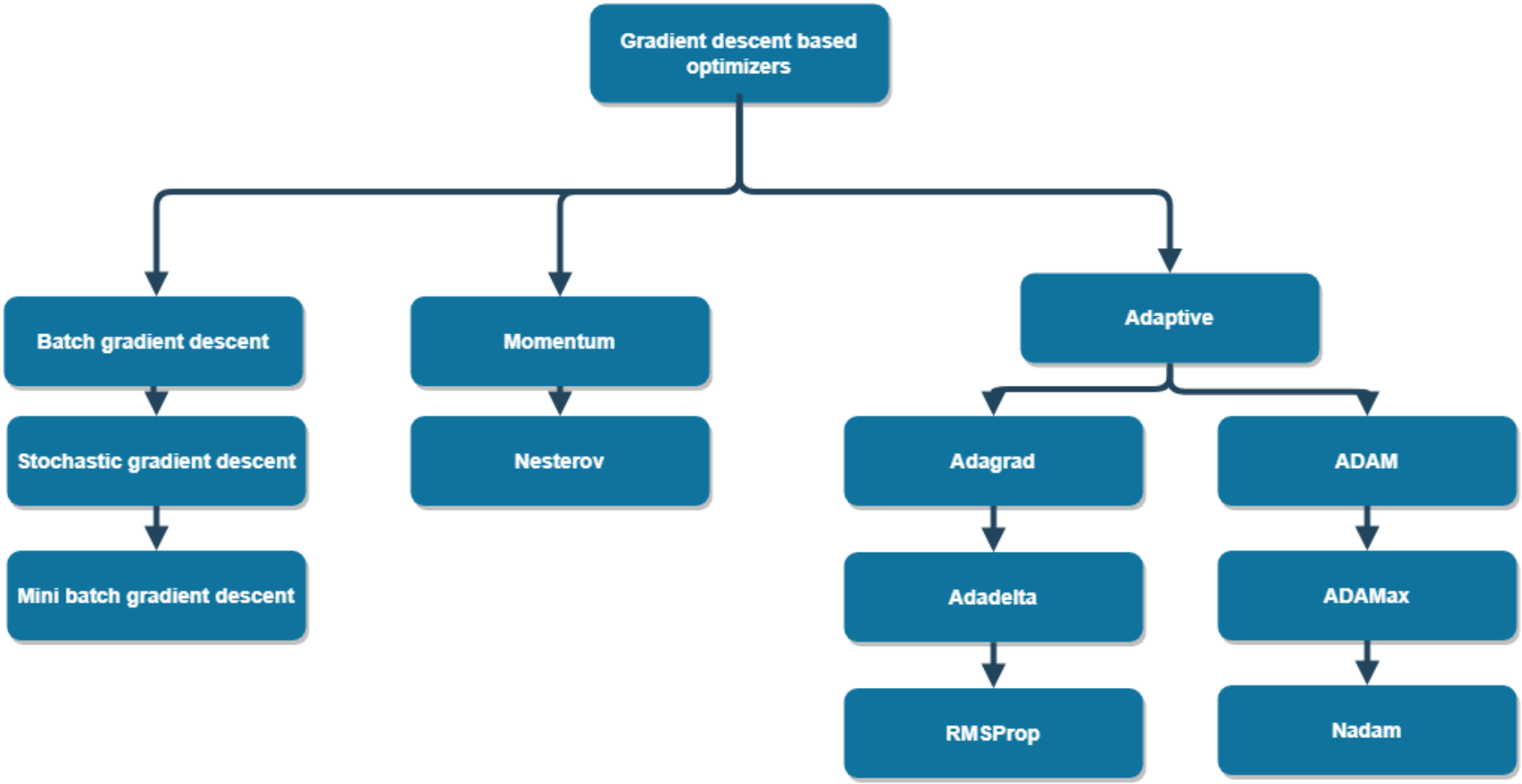
$$J = \frac{1}{2}\sum_{i=1}^{n}(y_i^2 + (ax_i)^2 + b^2 + 2ax_ib - 2ax_iy_i - 2y_ib)$$

$$\frac{\partial J}{\partial a} = \frac{1}{2}\sum_{i=1}^{n}(2ax_i^2 + 2x_ib - 2x_iy_i)$$

$$\frac{\partial J}{\partial a} = \sum_{i=1}^{n}x_i(ax_i + b - y_i)$$

$$\frac{\partial J}{\partial b} = \sum_{i=1}^{n}(ax_i + b - y_i)$$

Gradient Descent based optimizers



Gradient Descent

- Simplest of all optimizers
- Just one parameter - learning rate (denoted by η)
- Move in the direction opposite of the gradient
- Guaranteed to converge to global optimum (convex function) and local optimum otherwise

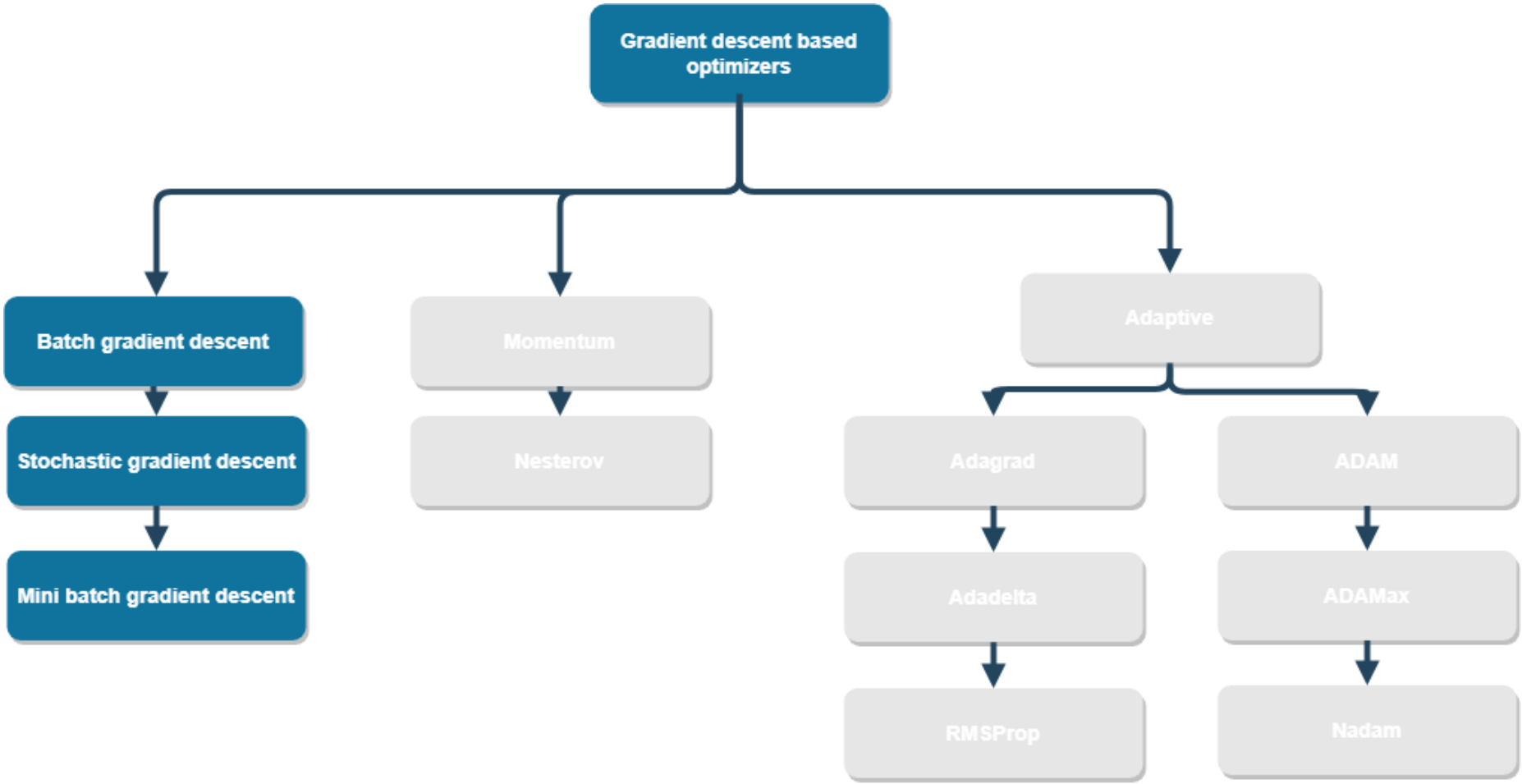
Update rule

$$\theta_t = \theta_{t-1} - \eta \frac{\partial(J)}{\partial\theta}$$

Cons

- Memory intensive - loads the full dataset and computes gradients

Gradient Descent based optimizers



Stochastic gradient descent

- Loss is computed and update performed at every single example
- Less memory intensive
- Very noisy convergence

Update rule

$$\theta_t = \theta_{t-1} - \eta \frac{\partial(J_{sample})}{\partial \theta}$$

where J_{sample} is the loss over single example

Mini batch gradient descent

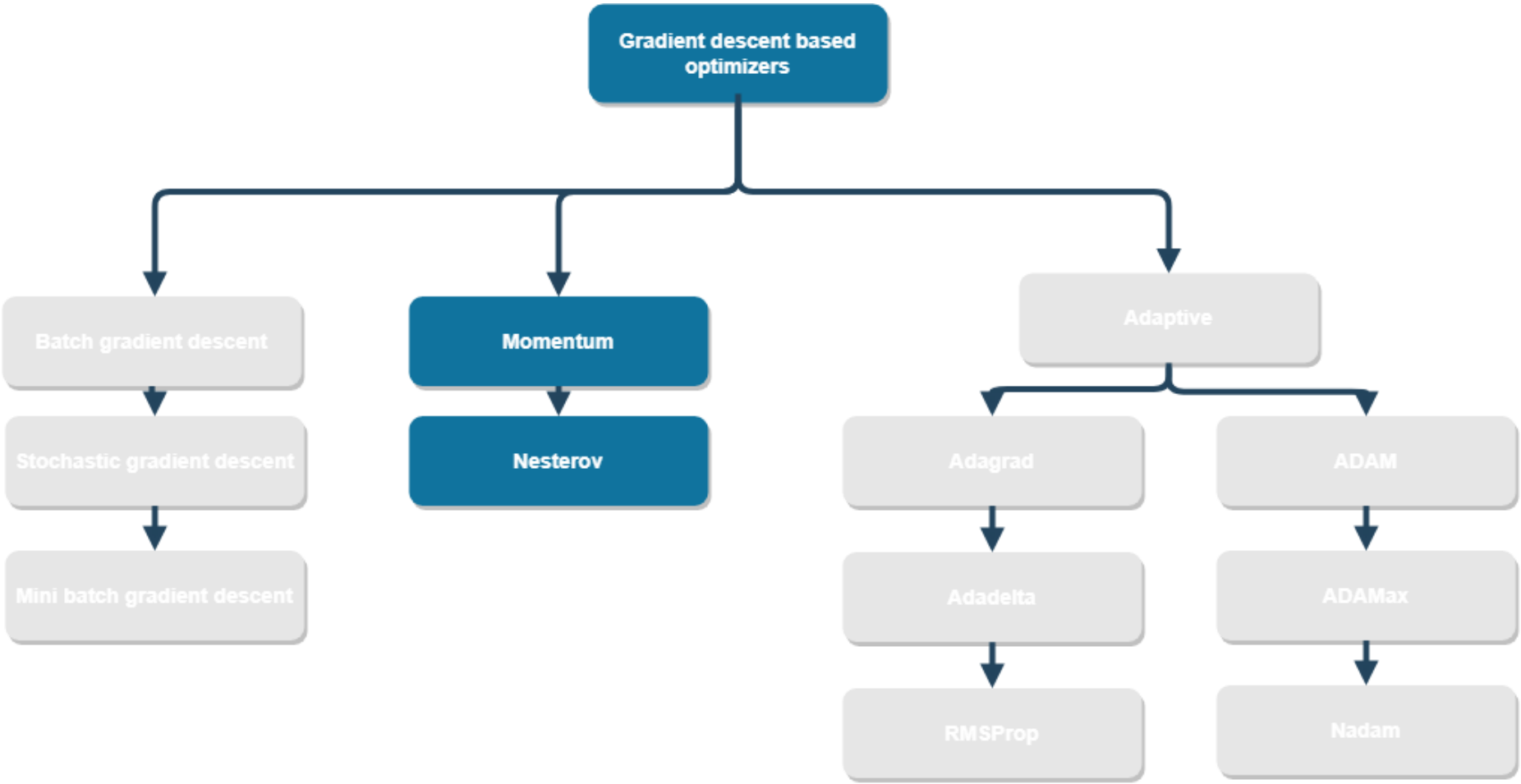
- Tradeoff between GD and SGD - update is performed on a mini batch of user specified size.
- Less memory intensive
- More commonly used in neural nets/ problems that don't fit in memory

Update rule

$$\theta_t = \theta_{t-1} - \eta \frac{\partial(J_{batch})}{\partial \theta}$$

where J_{batch} is the loss over the mini batch under process

Momentum based optimizers



Momentum

- lead to faster convergence when gradient changes slowly along one of the dimensions as is the case above
- Useful when there's a ravine and gradient moves quickly in one dimension while making hesitant progress in the other
  Update rule

$$v_t = \gamma v_{t-1} + \eta \frac{\partial J}{\partial \theta}$$

$$\theta_t = \theta_{t-1} - v_t$$

- Gains momentum if gradient remains in the same direction.
- Compare to above method in notebook.

Nesterov

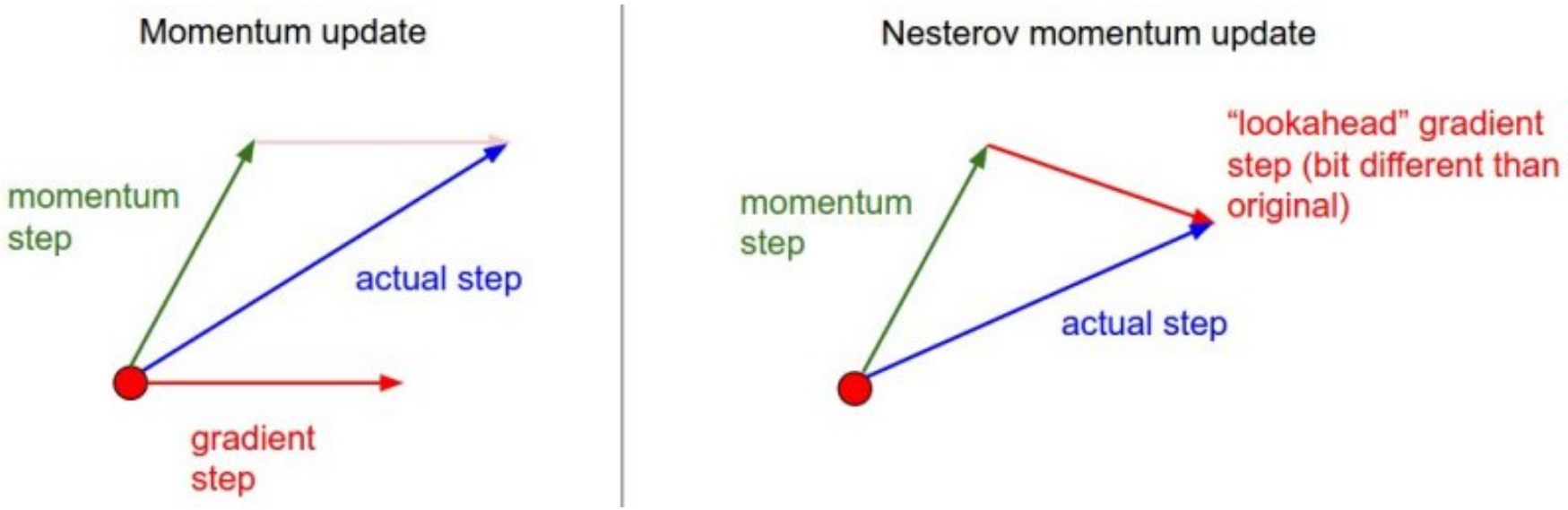
- To prevent the ball from sliding up the other slope with all the momentum it gained while coming down
- Still useful when gradient moves quickly in one dimension while making hesitant progress in the other Update rule

$$v_t = \gamma v_{t-1} + \eta \frac{\partial J(\theta - \gamma v_{t-1})}{\partial \theta}$$

$$\theta_t = \theta_{t-1} - v_t$$

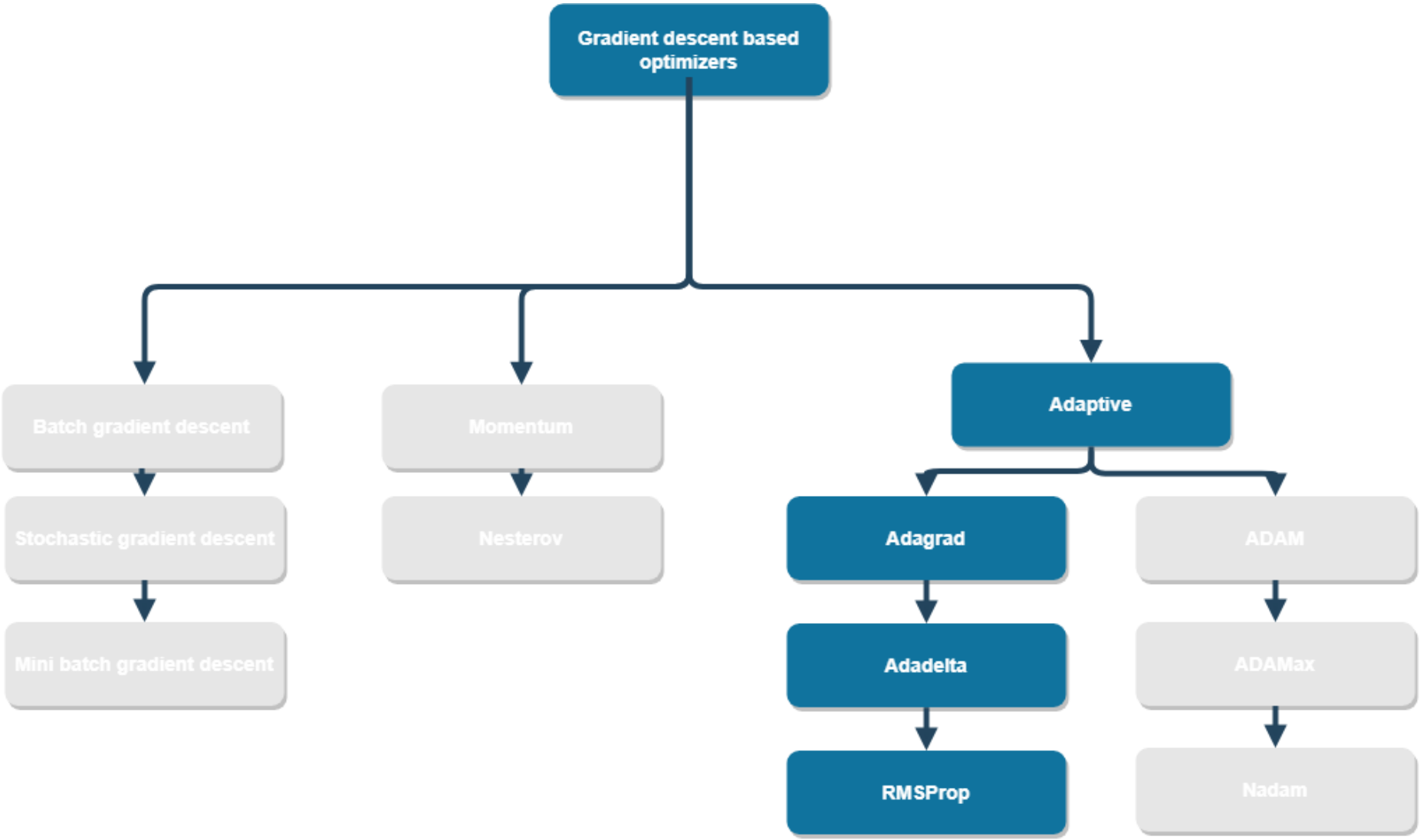
- Compare to momentum in notebook.

Momentum vs NAG



Credits: Andrej Karpathy CS229 slides

Adaptive optimizers



Adagrad

- Adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.
- well-suited for dealing with sparse data.

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

where $G_t \in R^{d \times d}$ is a diagonal matrix where each diagonal element (i, i) is the sum of the squares of the gradients w.r.t. the parameter up to time step t, while ϵ is a smoothing term that avoids division by zero. $g_{t,i}$ is the gradient w.r.t θ_i

Adadelta

- Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- Instead of accumulating all past squared gradients, Adadelta takes decaying average.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\eta}{\sqrt{E[g^2]_{t,i} + \epsilon}} \cdot \nabla J(\theta_{t-1,i})$$

The running average $E[g^2]_t$ at time step t then depends only on the previous average and the current gradient.

RMSprop

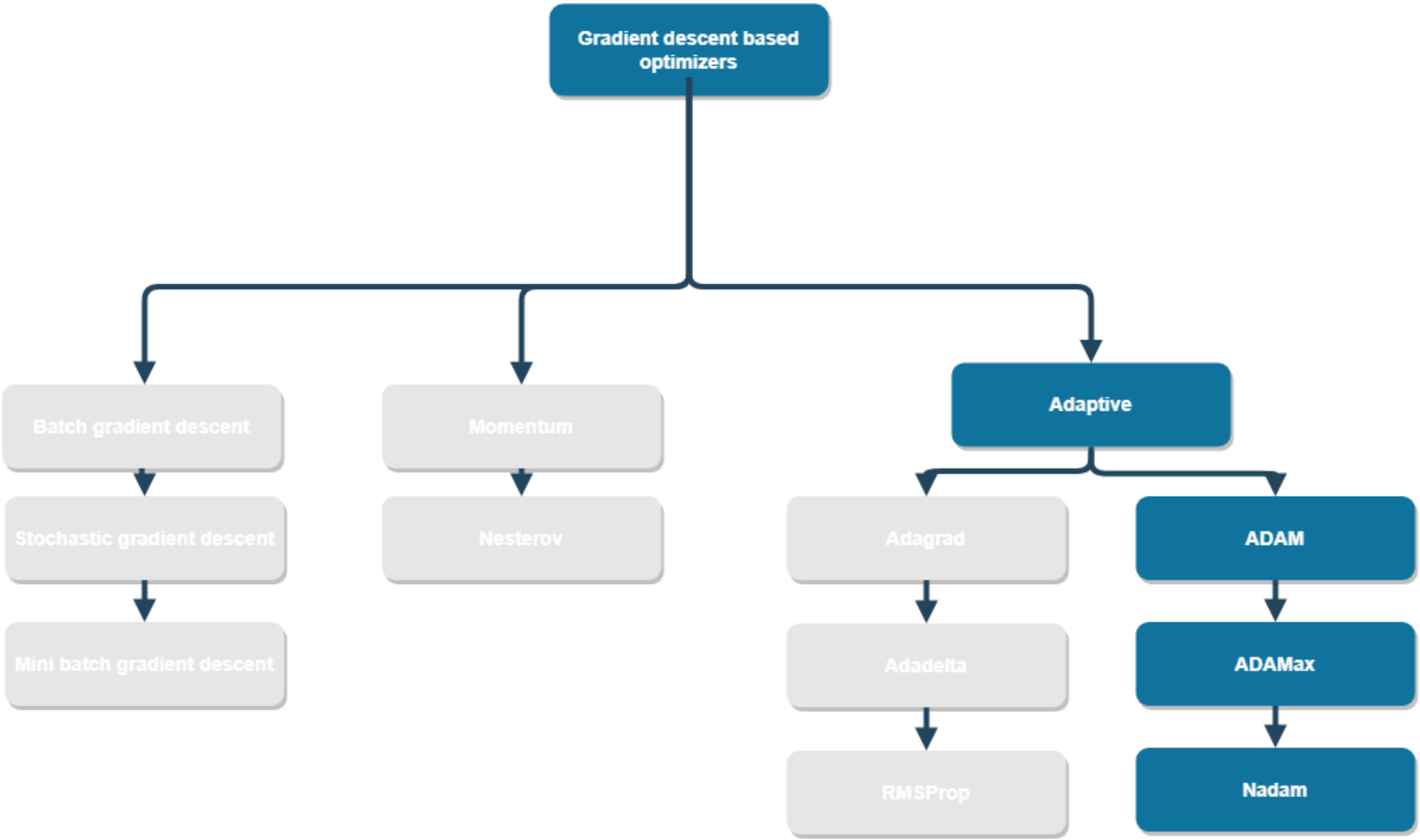
- RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_{t,i}$$

- RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients.
- RMSProp and adadelta evolved independently at the same time to overcome adagrad's rigorous decay in learning rate

Adaptive optimizers



Adam

- Adam stores exponentially decaying average of past squared gradients as well as gradients

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- β_1 and β_2 are close to 1 and m and v are intialized to 0. Hence to overcome the bias of values always being close to 0, the authors have proposed

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Adam

- With these, the update rule for Adam is

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \cdot \hat{m}_t$$

- Authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ

Adamax

- Norms for large p values generally become numerically unstable, which is why l_1 and l_2 norms are most common in practice.
- l_∞ also generally exhibits stable behavior. So, the authors of Adam propose Adamax showing that v_t with l_∞ converges to the following more stable value

$$u_t = \lim_{p \to \infty} \beta_2^p v_{t-1} + (1 - \beta_2^p)|g_t|^p$$

$$u_t = max(\beta_2 \cdot u_{t-1}, |g_t|)$$

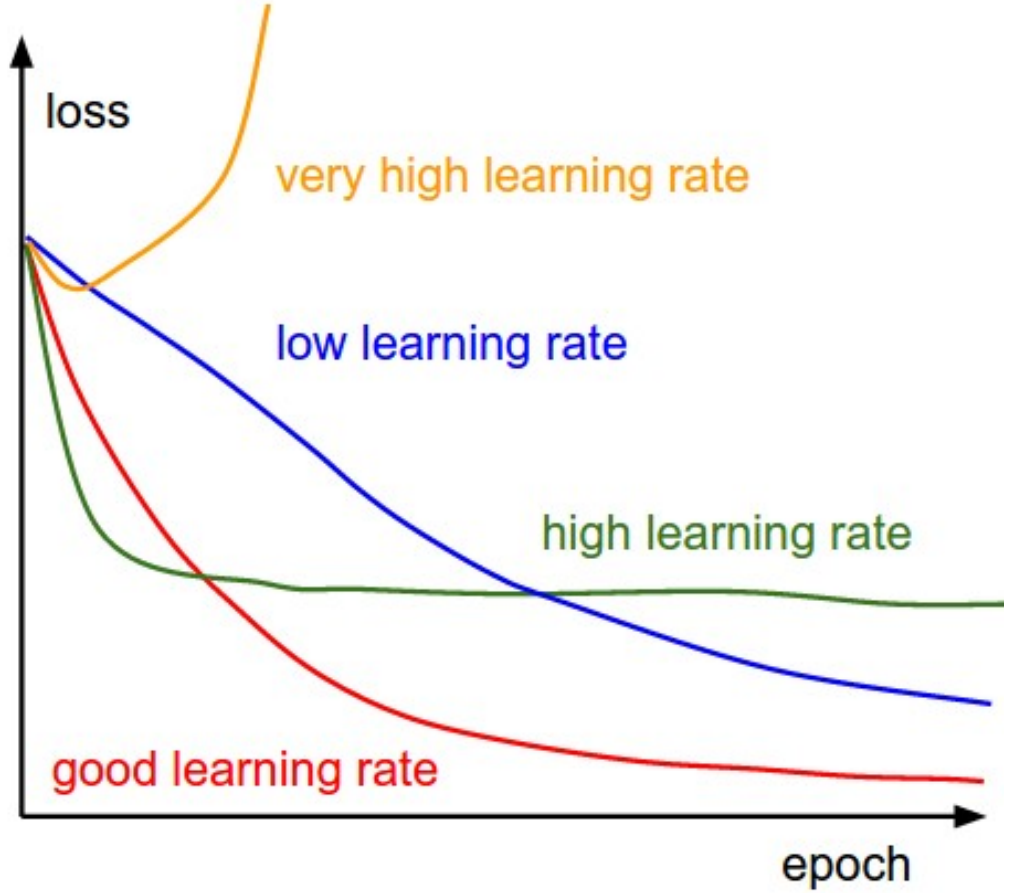
$$\theta_t = \theta_{t-1} - \eta \frac{m_t}{u_t}$$

Practical tips and tricks

- Adam might be the best overall choice for most cases
- Sparse input data - use adaptive methods (Adagrad known to work best)
- Mini batch gradient descent usually faster compared to gradient descent and less noisier than SGD
- Mostly learning rate is the only parameter requiring tweaking

Practical tips and tricks

- If very less decrease in loss function initially, start with slightly higher learning rate



Summary

- Gave an intuition of how gradient descent works
- Saw and understood the working of optimizers by looking at a linear regression problem
- Practical tips for using optimizers when designing deep networks

Resources

1. Optimizers paper by Sebastian Ruder [1]
2. CS231n lecture notes on the topic [2]
3. Adam/ Adamax [3]
4. NAG method [4]

For Further Reading I


Sebastian Ruder.
An overview of gradient descent optimization algorithms.
CoRR, abs/1609.04747, 2016.


Andrej Karpathy.
Neural Networks lecture 3, CS231N.
pages 1–7, July 2016.


Diederik P. Kingma and Jimmy Ba.
Adam: A method for stochastic optimization.
CoRR, abs/1412.6980, 2014.


Aleksandar Botev, Guy Lever, and David Barber.
Nesterov's Accelerated Gradient and Momentum as approximations to Regularised Update Descent.
arXiv e-prints, page arXiv:1607.01981, July 2016.