Course Notes: Deep Learning for Visual Computing

Peter Wonka

August 30, 2021

Contents

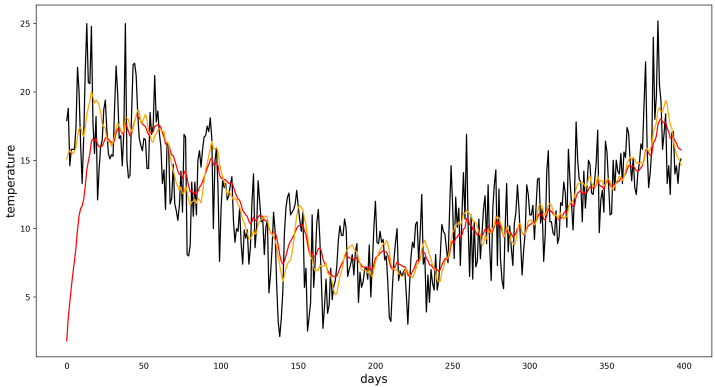
1 Optimization

1.1 Exponentially Moving Average Review

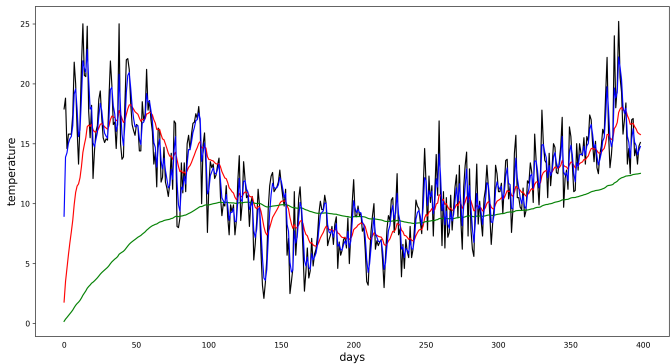
- Terms:
  - **Exponentially Moving Average** or
  - **Exponentially Weighted Moving Average**
- Example: time series of temperature: $x_1 = 4, \quad x_2 = 9, \quad \ldots, \quad x_{365} = 1$
- Exponentially moving average v:
  - $v_0 = 0$
  - $v_k = \beta v_{k-1} + (1-\beta) x_k$
- Explicit Example:
  - $v_0 = 0$
  - $v_1 = 0.9 v_0 + 0.1 x_1$
  - $v_2 = 0.9 v_1 + 0.1 x_2$

- *Advantage vs. Average*:
  - Computing the mean of the last k, e.g. 10, values requires us to store the last k values
  - Computing the EMA only requires to store the last value
  - If you want to compute some average of each parameter in the network / gradient of network parameters there is a big difference between regular mean and EMA
- Rule of Thumb:
  - exponentially moving average with β is similar to averaging over last $1/(1-\beta)$ values

- $\beta = 0.9$ (red) is similar to average over last 10 values (orange)

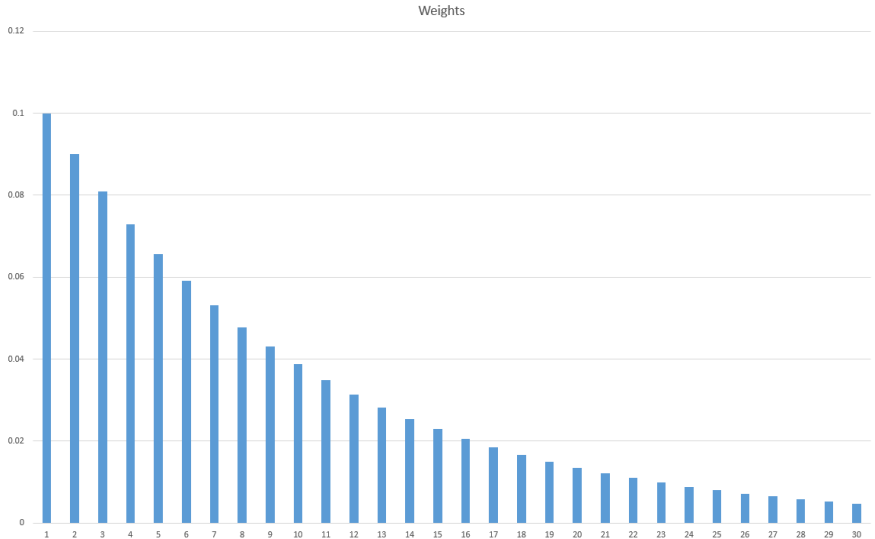


- red line: $\beta = 0.90 \approx 10$ values
- green line: $\beta = 0.98 \approx 50$ values
- blue line: $\beta = 0.5 \approx 2$ values



1.2 Exponentially Moving Average Example

- Example:
  - $v_{100} = 0.1\, x_{100} + 0.9\, v_{99}$
  - $v_{99} = 0.1\, x_{99} + 0.9\, v_{98}$
  - $v_{98} = 0.1\, x_{98} + 0.9\, v_{97}$
  - $\ldots$
  - $v_{100} = 0.1\, x_{100} + 0.9\, v_{99}$
  - $v_{100} = 0.1\, x_{100} + 0.9(0.9\, v_{98} + 0.1\, x_{99})$
  - $v_{100} = 0.1\, x_{100} + 0.1 * 0.9\, x_{99} + (0.9)^2\, v_{98}$
  - $v_{100} = 0.1\, x_{100} + 0.1 * 0.9\, x_{99} + 0.1(0.9)^2\, x_{98} + \ldots$
- Visualization (1 corresponds to the last weight of x_{100})


Weights
0.12
0.1
0.08
0.06
0.04
0.02
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

1.3 Bias Correction for Expontentially Moving Average

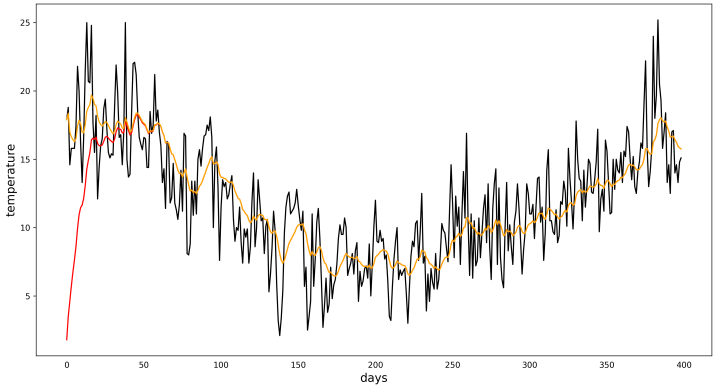
- In the beginning the sum of the weights is too low
- Requirement:
  - for a proper moving average the weights should sum to one
- Example:
  - $v_0 = 0$
  - $v_1 = 0.9 * 0 + 0.1 x_1$ (sum of weights is 0.1)
  - $v_2 = 0.1 x_1 + 0.1 * 0.9 x_2$ (sum of weights is 0.19)

- Solution: **Bias Correction Term**

$$\text{CorrectionTerm} = \frac{1}{1-\beta^k} \tag{1.1}$$

  - β parameter of the exponentially moving average
  - k iteration number
- Example: $(\beta = 0.9)$
  - $k = 2 \rightarrow \text{CorrectionTerm} = 1/(1-\beta^2) = 1/(1-0.81) = 1/0.19 = 5.26$
  - $k = 100 \rightarrow \text{CorrectionTerm} = 1/(1-\beta^{100}) = 1.000027$

- red line - original; orange line - bias corrected



- Discussion:
  - Bias correction is often ignored if only the last value is used (e.g. batch normalization)
  - Potentially a problem if initial values are also used (e.g. optimization)

1.4 Overview

- Problem formulation: finding the **parameters** ($\mathbf{w}$) of a neural network that significantly reduce a **loss function** $L(\mathbf{w})$
  - a loss function typically includes the **data loss** + **regularizers**
  - the **optimization algorithms** in this section determine how a **given** loss function is **optimized**
- Example Loss Function:

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \max(0, f(\mathbf{x_i}; \boldsymbol{W})_j - f(\mathbf{x_i}; \boldsymbol{W})_{y_i} + \Delta) + \lambda \sum_k \sum_l W_{kl}^2 \tag{1.2}$$

1.5 Classical Optimization

- Reference:
  - Book: "**Mathematical Programming: An Introduction to Optimization**" by Melvyn Jeter
  - Book: "**Convex Optimization**" by Stephen Boyd and Lieven Vandenberghe (free book and slides online)
  - Book: Nocedal, Wright, "**Numerical Optimization**"
  - Course: **AMCS211 Numerical Optimization**

1.6 General Form

- Many optimization problems can be represented in the following way:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g_i(\mathbf{x}) \geq 0, \quad \forall i \in \mathcal{I} \\ & g_j(\mathbf{x}) = 0, \quad \forall j \in \mathcal{E} \end{aligned}$$

  - Here $f(\cdot), g_i(\cdot), g_j(\cdot)$ are typically (not always) smooth functions.
  - $f(\cdot)$: the **cost/energy/objective** function
  - $g_i(\cdot), g_j(\cdot)$: the **constraints**
    - $\mathcal{I}$ is the index set of the **I**nequality constraints.
    - $\mathcal{E}$ is the index set of the **E**quality constraints

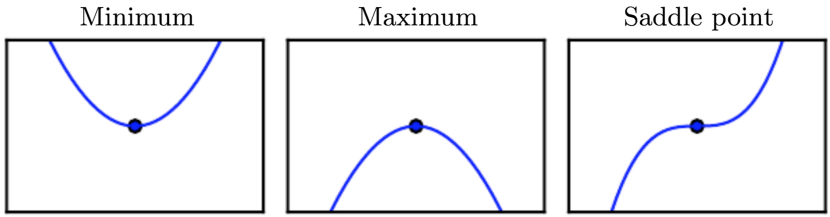
- **feasible set**: the set of the points that satisfy all the constraints:

$$\mathcal{F} := \left\{ \mathbf{x} \in \mathbb{R}^n \mid g_i(\mathbf{x}) \geq \mathbf{0} \; \forall i \in \mathcal{I}, \quad g_j(\mathbf{x}) = \mathbf{0} \; \forall j \in \mathcal{E} \right\}$$

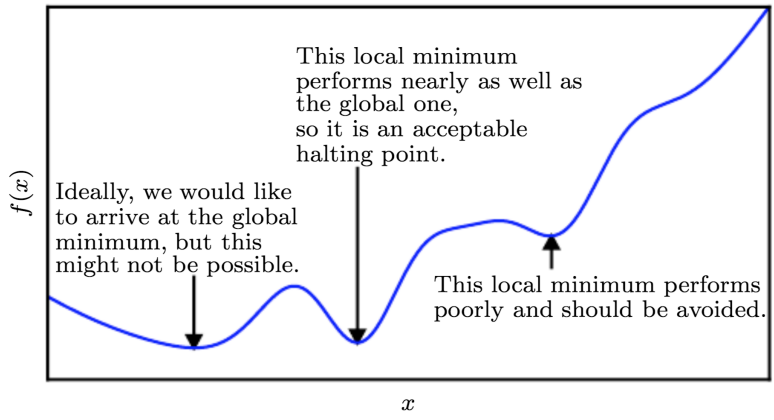
1.7 Optimization Concepts

- **constrained vs. unconstrained**: the feasible domain is the whole space $\mathbb{R}^n \rightarrow$ unconstrained (otherwise, the problem is constrained).
- **convex or non-convex**: the problem is convex if and only if the objective function $f(\cdot)$ is convex and the feasible domain $\mathcal{F}$ is a convex set.
- **continuous vs. discrete**: the variable $\mathbf{x}$ is a continuous or discrete variable.

- **minimum, maximum, saddle point**: stationary points with gradient zero



- **global vs. local minimum**:
  - $\mathbf{x}^*$ is called a **global** minimum if: for any feasible point $\mathbf{x} \in \mathcal{F}$ we have $f(\mathbf{x}^*) \leq f(\mathbf{x})$ holds.
  - $\mathbf{x}^*$ is called a **local** minimum if: (for some constant ϵ) for any feasible $\mathbf{x}$ that satisfies $\|\mathbf{x} - \mathbf{x}^*\| \leq \epsilon$ we have $f(\mathbf{x}^*) \leq f(\mathbf{x})$ holds.


This local minimum
performs nearly as well as
the global one,
so it is an acceptable
halting point.
Ideally, we would like
to arrive at the global
minimum, but this
might not be possible.
This local minimum performs
poorly and should be avoided.
$f(x)$
x

1.8 Classical Optimization Algorithm Examples

- Gradient descent methods
- Newton's method
- Quasi-Newton methods
- Line search methods
- Trust region methods
- Dual ascent method
- Method of multipliers
- Alternating direction method of multipliers (ADMM)
- Primal-Dual methods
- Interior point methods

1.9 Optimization in Neural Network

- The real performance measure P is defined w.r.t. the **test set**
- We optimize a loss function $L(\mathbf{w})$ instead of P directly
- We optimize on the **training set** rather than the **test set**
- For pure optimization, minimizing $L(\mathbf{w})$ is a goal in and of itself
- The objective function is a sum over the training examples
- The training set is large and all training samples do not fit into memory at once

1.10 Simple Taxonomy of Optimization Methods

- Optimization method types:
  - **batch methods** (also called **deterministic methods**): optimization algorithms that use the **entire** training set.
  - **stochastic methods** (also called **online methods**): optimization algorithms that use only a **single** example at a time.
  - **minibatch methods** (also called **minibatch stochastic methods**, or simply **stochastic methods** nowadays): algorithms that use more than one but fewer than all the training examples.
- **Batch size**: number of samples, e.g. images. in a minibatch

1.11 Basic algorithms

- Loss function L in neural network optimization has the following structure:

$$L(\mathbf{w}) = \frac{1}{n}\sum_i LD_i\big(f(\mathbf{x}_i;\mathbf{w}), y_i\big) + \sum_j \lambda_j R_j(\mathbf{w}) \tag{1.3}$$

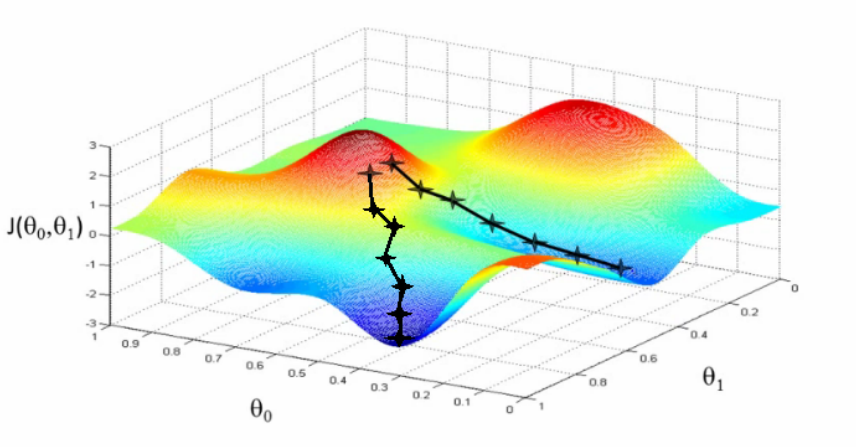
- where $(\mathbf{x}_i, y_i)$ are training samples, e.g., $\mathbf{x}_i$ represents the i-th image, and y_i is the corresponding label.
- $\mathbf{w}$ are the network parameters / weights to be trained / optimized, e.g., parameters of convolutional layers, linear layers, and normalization layers
- LD_i: data loss function applied to the i-th sample
- R_j: regularizers (e.g. sum of squared weights)
- n: number of samples
- We assume all network parameters are concatenated into a long vector $\mathbf{w}$

- Note: variables for the optimization are $\mathbf{w}$ not $\mathbf{x}$
- Alternate Formulation:

$$L(\mathbf{w}) = \mathbb{E}_{(\mathbf{x},y)\sim\tilde{p}} L\big(\ldots\big) \tag{1.4}$$

  - L is the per-example loss function
  - $\tilde{p}$ is the empirical distribution (from the training set)

1.12 Gradient Descent



- Algorithm name: **Gradient Descent**

- Algorithm reaches different local minima with different initial points
- **Intuition**:
  - pick initial guess
  - skii downhill along the gradient direction (with fixed step size)
  - keep fingers crossed

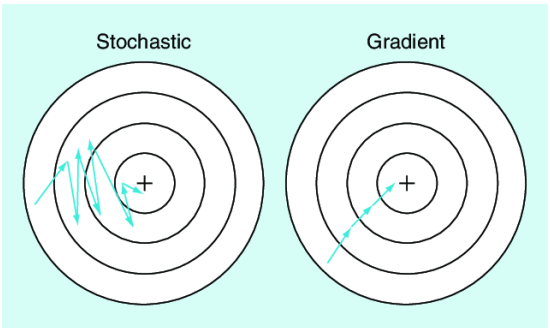
Algorithm 1: Gradient Descent

Input: learning rate α
Output: network parameters $\mathbf{w}$
initialize $\mathbf{w}$;
while *stopping criterion not fulfilled* **do**
  compute the gradient of the loss function: $\mathbf{g} = \nabla_{\mathbf{w}} L$;
  update the weights: $\mathbf{w} = \mathbf{w} - \alpha \mathbf{g}$;
end

- **Weaknesses**:
  - hard to find a proper **learning rate** (step size)
    - large learning rate: does not converge
    - small learning rate: converges slowly
    - (partial) solution: we will introduce a learning rate schedule
  - expensive to compute the exact gradient (process all data)
    - (partial) solution: we will work with mini-batches
  - No curvature information included
    - solution needs the second-order derivatives, the **Hessian**
    - computation of the Hessian is very expensive
    - second order methods are not very popular in deep learning
    - some approximate second order methods (L-BFGS) are sometimes used

1.13 Stochastic Gradient Descent

- Algorithm name: **Stochastic Gradient Descent**
- basic, but successful algorithm in deep learning



- **Intuition**:

- pick initial guess
- skii downhill along the gradient direction **estimated on a mimibatch of m samples** (with decreasing step size)
- keep fingers crossed

Algorithm 2: Stochastic Gradient Descent

Input: learning rate schedule α_k, minibatch size m
Output: network parameters $\mathbf{w}$
initialize $\mathbf{w}$;
while *stopping criterion not fulfilled* **do**
  sample a minibatch of m examples from the training set: $\mathcal{I} = i_1, i_2, \cdots, i_m$;
  estimate the gradient: $\mathbf{g} = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i \in \mathcal{I}} LD_i\big(f(\mathbf{x}_i; \mathbf{w}), y_i\big)$;
  update the weights: $w \leftarrow \mathbf{w} - \alpha_k \mathbf{g}$;
end

- Gradient estimate should include regularizers:

$$\frac{1}{m}\nabla_{\mathbf{w}}\sum_{i\in\mathcal{I}}LD_i\big(f(\mathbf{x}_i;\mathbf{w}),y_i\big)+\sum_j\lambda_j R_j(\mathbf{w}) \tag{1.5}$$

  - We omit the regularizers j to make the notation more concise
- **Discussion**:
  - Can be a great algortihm if you find a good learning rate schedule
  - Finding a good learning rate schedule is time consuming
- **Ideas for Improvement**:
  - add **momentum**: a function of the gradients of previous steps to alter the next direction (**SGD with momentum**)
  - different learning rate for **each** parameter (**RMSProp**)
  - add Momentum and different learning rate per parameter (**ADAM**)

1.14 How to choose the minibatch size?

- Minibatch sizes are generally driven by multiple factors
- Larger batches → **more accurate** estimate of the gradient
- Small batches → **underutilise** the hardware architectures
- Limited memory:
  - typically, all examples in the batch are to be processed in parallel
  - memory consumption scales with the batch size
  - available memory is a limiting factor in batch size.
  - problem for tasks like dense regression, e.g. segmentation papers might run inference on the CPU
- GPU favors specific sizes of arrays (e.g., power of 2 batch sizes)
- Non-trivial side effects, e.g. batch normalization

- Small batches can offer a regularizing effect (perhaps due to the noise they add to the learning process). E.g., generalization error might be best for a batch size of 1.
- For some problems people observe performance increasing with batch size and then decreasing.
- Typical batch sizes $= 1, 2, 4, 8, 16, 32, \ldots$

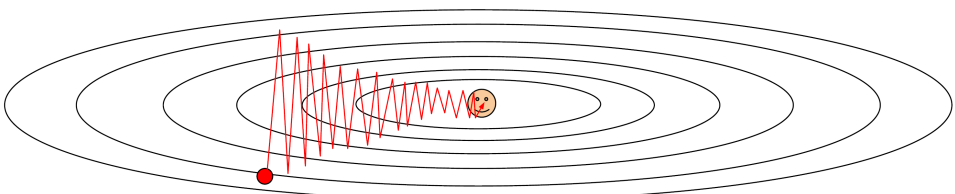
1.15 Ill-conditioning Problem

- A mathematical problem or series of equations is **ill-conditioned** if a small change in the input variable leads to a large change in the output.
- Leads to computational problems:
  - if the optimization is ill-conditioned, the solution is more difficult to find.
- The ill-conditioning problem is generally believed to be present in neural network training problems.
  - It can cause some algorithm to get stuck in the sense that even very small steps increase the loss function:
- Second-order Taylor approximation:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)^T \mathbf{g} + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^T \boldsymbol{H}(\mathbf{x}-\mathbf{x}_0) \tag{1.6}$$

  - $\mathbf{g}$: gradient of f at $\mathbf{x}_0$;

- $\boldsymbol{H}$: hessian of f at $\mathbf{x}_0$
- With learning rate ϵ, the new point $\mathbf{x}$ would be $\mathbf{x}_0 - \epsilon\mathbf{g}$
- Then we have $f(\mathbf{x}_0 - \epsilon\mathbf{g}) - f(\mathbf{x}_0) \approx -\epsilon\mathbf{g}^T\mathbf{g} + \frac{1}{2}\epsilon^2\mathbf{g}\boldsymbol{H}\mathbf{g}$
- Generally, the Hessian $\boldsymbol{H}$ is ill-conditioning $\rightarrow$ $\frac{1}{2}\epsilon^2\mathbf{g}\boldsymbol{H}\mathbf{g}$ can exceed $-\epsilon\mathbf{g}^T\mathbf{g}$, therefore the small step update cannot decrease the loss, and we get stuck.

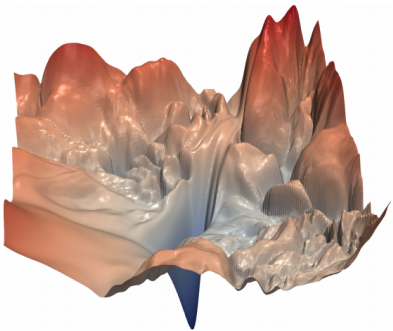


1.16 Many Local Minima and Saddle Points

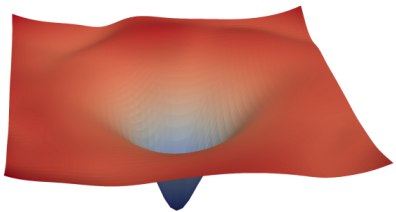
- Literature:
  - Pascanu et al., On the saddle point problem for non-convex optimization
  - Dauphin et al., Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, NIPS 2014
- **Local minima**: Loss functions are **non-convex** with **many** llocal minima
- **saddle points**:
  - gradient is zero, but the Hessian has both positive and negative eigenvalues
- Theoretical discussions:
  - What does the loss landscape look like?
  - Are there many global optima?
  - Are there a lot more saddle points than local minima?

- Are saddle points a problem? (or does SGD just march through)
- Are local minima a problem?

- Example: Visualizing the Loss Landscape of Neural Nets



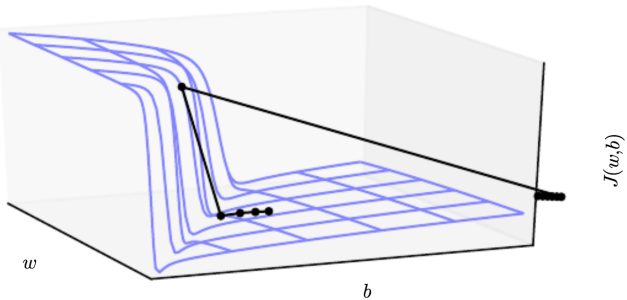
(a) without skip connections



(b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

1.17 Cliffs and Exploding Gradients



- **Cliffs** in the loss function
- **Exploding gradients** result from the multiplication of several large values together
- **Long-term dependencies**: (e.g., in recurrent networks) repeated application of the

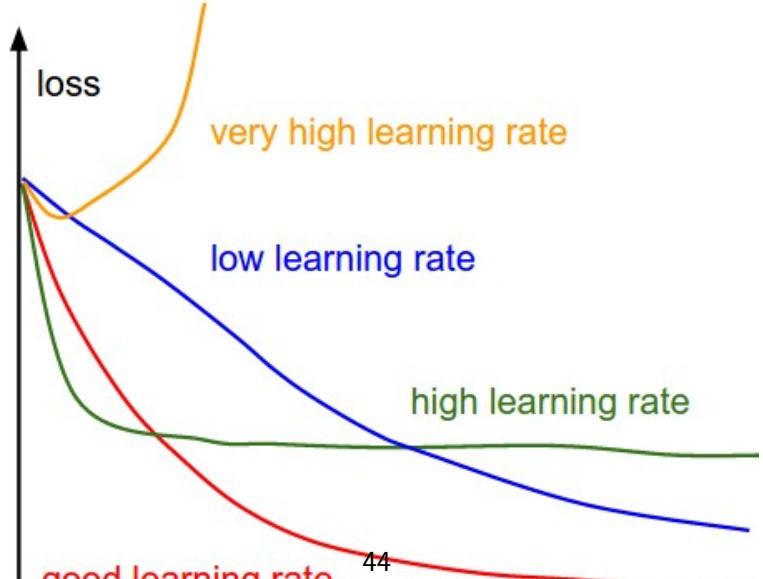
same parameters

- **Inexact gradients**: gradient estimate is noisy or even biased

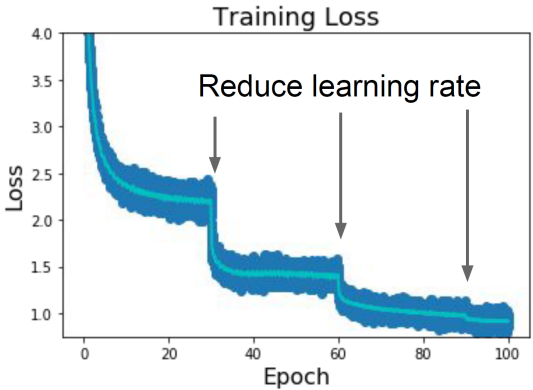
1.18 Stopping Criteria

- Follow an **established test protocol**:
  - to compare methods there might be an established protocol for a data set.
  - e.g., train for k epochs
- You run **out of time**:
  - often training still improves the loss, but you run out of time
- Stop if loss on the validation set gets worse
- Stop if performance metric (e.g. accuracy) on the validation set gets worse
- Stop based on manual inspection
- Stop with whatever criterion, but remember the best weights on validation set (requires many checkpoints)
- **Not allowed**: continuously test performance metric on test set and report best result

1.19 Basic Learning Rate Scheduling

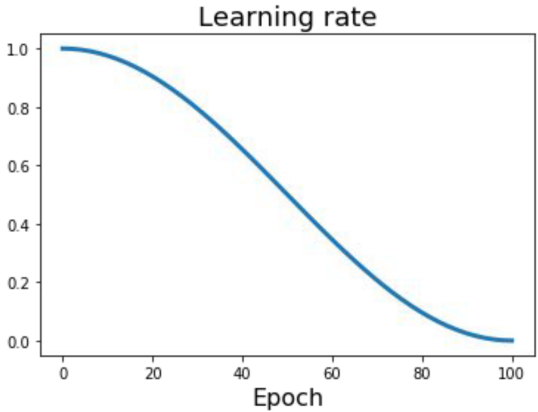


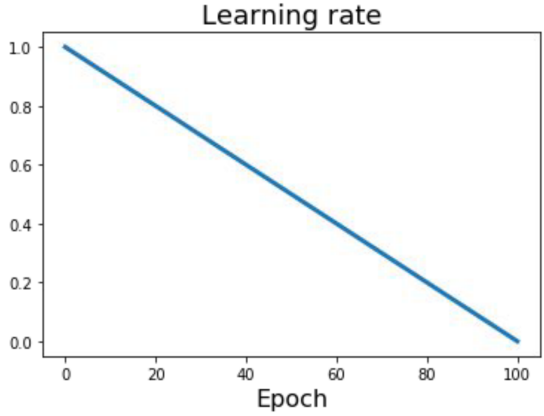
- Many learning rate scheduling ideas in practice.
- Basic schedules only decrease the learning rate over time at certain points
- Algorithm 1:
  - Step1: Find a starting learning rate by trying a few fixed learning rates
  - Step2: Train and manually inspect the loss curve on validation set
  - Step3: If the loss stagnates divide the learning rate by a factor, e.g. factor 10
  - Step 4: If not finished goto Step2
  - Step 5: hardcode the learning rate schedule into your program
- Example (ResNet Paper):
  - "We use SGD with a mini-batch size of 256. The learning rate starts from 0.1 and is divided by 10 when the error plateaus, and the models are trained for up to 60×104 iterations. We use a weight decay of 0.0001 and a momentum of 0.9."
  - multiply learning rate by 0.1 after epochs 30, 60, and 90

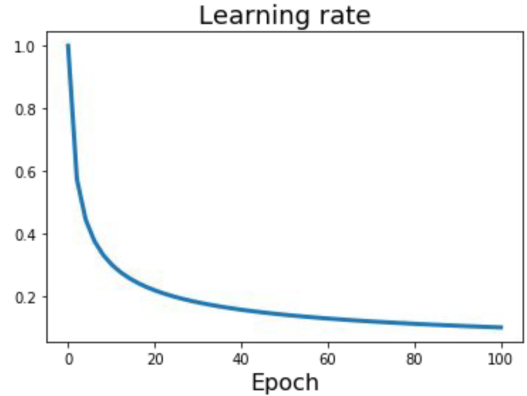

Training Loss
Reduce learning rate
Loss
Epoch
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
20
40
60
80
100

1.20 Advanced Learning Rate Scheduling

- **continuously changing learning rate**: gradually decrease the learning rate over time.




Learning rate
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Epoch



- **cosine:** $\alpha_k = \frac{1}{2}\alpha_0 (1 + cos(k\pi / T))$
- **linear:** $\alpha_k = \alpha_0(1 - k/T)$
- **Inverse sqrt:** $\alpha_k = \alpha_0 / \sqrt{k}$

- α_0: initial learning rate
- α_k: learning rate at epoch k
- T: total number of Epochs
  - Design choice: change learning rate after each mini-batch or after each epoch?

- Important: The advanced algorithms, such as SGD+Momentum, RMSProp, Adam do a lot of modifications, but they all have a base learning rate
  - you always need a learning rate schedule

1.21 SGD with Momentum

- Recall the standard SGD:
  - Pick initial guess
  - while stopping criterion not fulfilled (at iteration k):
    - sample a minibatch of m samples
    - estimate the gradient on the minibatch, denote it as $\mathbf{g}_k$
    - (*) **the update direction of the weights is** $-\alpha_k \mathbf{g}_k$
    - update the weights by $\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha_k \mathbf{g}_k$
- We can use **Momentum** to improve the standard SGD by changing the way to update the weights in SGD as in step (*).
- More specifically, in step (*), the information of the previous gradients will be added together with the current gradient to update the direction of the weights.
- In a general formulation, we can update the weights $\mathbf{w}_k = \mathbf{w}_{k-1} + \mathbf{v}_k$

- Classical SGD: $\mathbf{v}_k$ only depends on the current gradient estimation $\mathbf{g}_k$
- Momentum: $\mathbf{v}_k$ also depends on previous gradient estimation $\mathbf{g}_1, \cdots, \mathbf{g}_k$

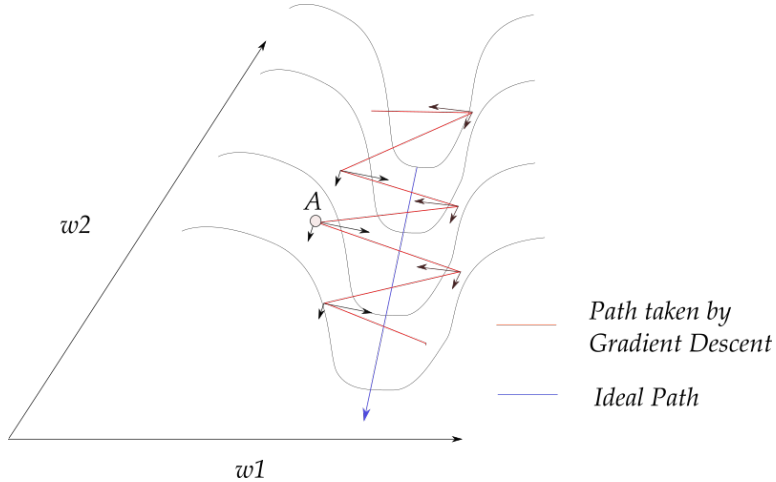
- let $\mathbf{g}_k$ denote the gradient estimation of m samples at k-th iteration
- let $\mathbf{v}_k$ denote the velocity at k-th iteration
- In classical SGD update direction is $-\alpha_k \mathbf{g}_k$
- In Momentum we have:
  - $\mathbf{v}_k = \beta_k \mathbf{v}_{k-1} - \alpha_k \mathbf{g}_k$
  - $\mathbf{w}_k = \mathbf{w}_{k-1} + \mathbf{v}_k$
- Alternative formulation:
  - $\mathbf{v}_k = \beta_k \mathbf{v}_{k-1} + \mathbf{g}_k$
  - $\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha_k \mathbf{v}_k$
- Alternative formulation:
  - $\mathbf{v}_k = \beta_k \mathbf{v}_{k-1} + (1 - \beta_k) \mathbf{g}_k$

- $\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha_k \mathbf{v}_k$
- Recommended by Andrew Ng: allows tuning of β and α more independently

- In practice, β_k is initialized to 0.5 and gradually annealed to 0.9 over multiple epochs
- Example:
  - Assume we have $\alpha = 1$ and $\mathbf{v}_1 = -\mathbf{g}_1$ (since we do not have any history information at the first iteration)
  - $\mathbf{v}_2 = \beta \mathbf{v}_1 - \mathbf{g}_2$
  - $\mathbf{v}_3 = \beta \mathbf{v}_2 - \mathbf{g}_3 = \beta(\beta \mathbf{v}_1 - \mathbf{g}_2) - \mathbf{g}_3 = -\beta^2 \mathbf{g}_1 - \beta \mathbf{g}_2 - \mathbf{g}_3$
  - ...
  - $\mathbf{v}_k = \beta^{k-1} \mathbf{g}_1 + \cdots + \beta^{k-i} \mathbf{g}_i + \cdots + \beta \mathbf{g}_{k-1} + \mathbf{g}_k$
  - ...
- Observations:
  - The previous gradients are included in subsequent updates

- The weight of the most recent previous gradients is more then less recent ones (since $\beta < 1$)
- The gradient update is an exponential average over previous gradients

1.22 Illustration of Momentum



- In this example, we only have two network parameters w_1 and w_2 to optimize

- black arrows: the value of the gradient w.r.t w_1 and w_2
- red path: the path taken by gradient descent
- we can notice that the value of the gradient w.r.t w_1 keeps switching the signs, which leads to a zig-zag path
- if we use Momentum in this case, i.e., (exponentially) sum up all the previous gradients, the value along the w_1 will be cancel out, and ideally we will move along w_2, as suggested by the blue path.

1.23 SGD with Momentum

Algorithm 3: Stochastic Gradient Descent with Momentum

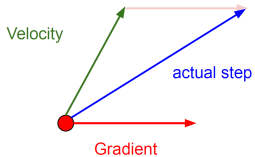
Input: learning rate schedule α_k, minibatch size m, friction schedule ρ_k
Output: network parameters $\mathbf{w}$
initialize $\mathbf{w}$;
while *stopping criterion not fulfilled* **do**
  sample a minibatch of m examples from the training set (say with sample ID $\mathcal{I} = i_1, i_2, \cdots, i_m$);
  estimate the gradient: $\mathbf{g} = \frac{1}{m} \nabla_{\mathbf{w}} L$;
  compute **velocity**: $\mathbf{v} = \beta \mathbf{v} + (1 - \beta)\mathbf{g}$;
  update the weights $\mathbf{w} = \mathbf{w} - \alpha_k \mathbf{v}$;
end

- build up velocity as a running mean of gradients

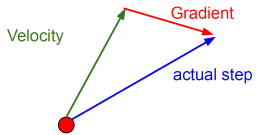
- β: friction; e.g. $\beta = 0.9$ or $\beta = 0.99$

1.24 Momentum vs. Nesterov Momentum

Momentum update:



Nesterov Momentum



- ...

1.25 RMSProp

- Algorithm name: **RMSProp**
  - **R**oot **M**ean **S**quare **Prop**agation
- Main idea:
  - update the learning rate for each variable / network parameter independently
- Recall the standard SGD:
  - update the weights by: $\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha_k \mathbf{g}_k$
  - the gradient $\mathbf{g}_k$ is a **vector**
  - it is multiplied by a **scalar** α_k
  - i.e., uniform learning rate (stepsize) for all directions/ variables.
- It is better to have different learning rate for different variable.
- The learning rate for each variable is automatically determined

- it depends on the gradient at this variable in all the previous iterations.

- Notation:
  - $\mathbf{p}_k^j$: the j-th component of the vector $\mathbf{p}_k$ (at k-th iteration of some vector $\mathbf{p}$)
    - this is a **scalar** not a vector, but we still keep the bold font
  - $\mathbf{g}_k^j$: the estimated gradient w.r.t the weight $\mathbf{w}^j$ at k-th iteration
    - i.e., the j-th entry of $\mathbf{g}_k$
    - scalar not a vector

- The learning rate at k-th iteration for the parameter $\mathbf{w}^j$ is computed as follows:
  - $\mathbf{s}_k^j = \beta \mathbf{s}_{k-1}^j + (1-\beta)\left(\mathbf{g}_k^j\right)^2$
  - $\mathbf{w}_k^j = \mathbf{w}_{k-1}^j - \frac{\alpha_k}{\sqrt{\mathbf{s}_k^j + \epsilon}} \mathbf{g}_k^j$

- Similar to the previous case, $\mathbf{s}_k^j$ is an exponential average of squares of the gradients w.r.t the j-th component/weight.

- The weighted sum of squared gradients is used to modify the learning rate
  - If w^j is larger, the learning rate will be set to be smaller
- In the example visualization before, we can see that the squared gradients w.r.t. the weight w_1 have a larger value, then the learning rate w.r.t w_1 will be set smaller. This can help use avoid bouncing between the ridges.

Algorithm 4: RMSProp: Root Mean Square Propagation

Input: learning rate schedule α_k, minibatch size m, moving average coefficient β
Output: network parameters $\mathbf{w}$
initialize $\mathbf{w}$;
while *stopping criterion not fulfilled* **do**
  sample a minibatch of m examples from the training set: $\mathcal{I} = i_1, i_2, \cdots, i_m$;
  estimate the gradient: $\mathbf{g} = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i \in \mathcal{I}} LD_i\big(f(\mathbf{x}_i; \mathbf{w}), y_i\big)$;
  update the weights: $\mathbf{w}_k^j = \mathbf{w}_{k-1}^j - \frac{\alpha_k}{\sqrt{\mathbf{s}_k^j + \epsilon}} \mathbf{g}_k^j$, where $\mathbf{s}_k^j = \beta \mathbf{s}_{k-1}^j + (1-\beta)\big(\mathbf{g}_k^j\big)^2$;
end

1.26 ADAM

- Literature: Diederik P. Kingma, Jimmy Ba, **Adam: A Method for Stochastic Optimization**, ICLR 2015
- Algorithm Name: **ADAM**
  - **Ada**ptive **m**oment estimation
- Momentum: exponential average over the gradients to update the moving direction
- RMSProp: exponential average over the squared gradients to update the learning rate
- We can combine the heuristics of both Momentum and RMSProp:
  - from momentum: the change of direction depends on all the previous gradient directions
  - from RMSProp: use different learning rate for different variables.
- Updating algorithm: (at k-th iteration)

- $\mathbf{v}_k^j = \beta_1 \mathbf{v}_{k-1}^j + (1-\beta_1)\mathbf{g}_k^j$
- $\mathbf{s}_k^j = \beta_2 \mathbf{s}_{k-1}^j + (1-\beta_2)\left(\mathbf{g}_k^j\right)^2$
- $\hat{\mathbf{v}}_k^j = \mathbf{v}_k^j / (1-\beta_1^k)$
- $\hat{\mathbf{s}}_k^j = \mathbf{s}_k^j / (1-\beta_2^k)$
- $\mathbf{w}_k^j = \mathbf{w}_{k-1}^j - \alpha_k \frac{1}{\sqrt{\hat{\mathbf{s}}_k^j + \epsilon}} \hat{\mathbf{v}}_k^j$

- Default Parameters:
  - β_1 is set to around 0.9 (first moment)
  - β_2 is set to around 0.999 (second moment)
  - ϵ is set to $1e-8$
  - Note: the parameter for the second moment includes a lot more values in the moving average.
- Discussion:

- Very popular
- Often not the best results, but **easier to pick parameters** that work well
- Very good for development

Algorithm 5: ADAM

Input: learning rate schedule α_k, minibatch size m, moving average coefficients β_1, β_2

Output: network parameters $\mathbf{w}$

initialize $\mathbf{w}$;

while *stopping criterion not fulfilled* **do**

sample a minibatch of m examples from the training set: $\mathcal{I} = i_1, i_2, \cdots, i_m$;

estimate the gradient: $\mathbf{g} = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{i \in \mathcal{I}} LD_i\big(f(\mathbf{x}_i; \mathbf{w}), y_i\big)$;

update the weights: $\mathbf{w}_k^j = \mathbf{w}_{k-1}^j - \alpha_k \frac{1}{\sqrt{\hat{\mathbf{s}}_k^j + \epsilon}} \hat{\mathbf{v}}_k^j$, where $\hat{\mathbf{v}}_k^j$ is bias-corrected velocity, $\hat{\mathbf{s}}_k^j$ is bias-corrected exponential average of squares of the gradients. Specifically, $\hat{\mathbf{v}}_k^j = \frac{1}{1-\beta_1^k}\Big(\beta_1 \mathbf{v}_{k-1}^j + (1-\beta_1)\mathbf{g}_k^j\Big)$, $\hat{\mathbf{s}}_k^j = \frac{1}{1-\beta_2^k}\Big(\beta_2 \mathbf{s}_{k-1}^j + (1-\beta_2)\big(\mathbf{g}_k^j\big)^2\Big)$;

end

1.27 Advanced Algorithms Overview

- We call these algorithms *advanced*, because they are not standard
- Examples of what people try to do in research:
  - Rectified Adam (Radam)
  - Lookahead
  - Novograd

1.28 Rectified ADAM

- Algorithm name: **Rectified ADAM** (RAdam)
- Literature: Liu et al., On the Variance of the Adaptive Learning Rate and Beyond
- Claimed advantages:
  - improved convergence
  - better training stability (less sensitive to chosen learning rates)
  - better accuracy and generalization for a lot of AI applications
- Notation: generic framework for adaptive methods (e.g., Momentum, RMSProp, ADAM)
- Observations:
  - SGD, RMSProp, ADAM, all have adaptive learning rate
  - big variance of the adaptive learning rate in the early stage of model learning leads to the convergence issue

Algorithm 1: Generic adaptive optimization method setup. All operations are element-wise.

Input: $\{\alpha_t\}_{t=1}^{T}$: step size, $\{\phi_t, \psi_t\}_{t=1}^{T}$: function to calculate momentum and adaptive rate, θ_0: initial parameter, $f(\theta)$: stochastic objective function.

Output: θ_T: resulting parameters

while $t = 1$ *to* T **do**

  $g_t \leftarrow \Delta_\theta f_t(\theta_{t-1})$ (Calculate gradients w.r.t. stochastic objective at timestep t)

  $m_t \leftarrow \phi_t(g_1, \cdots, g_t)$ (Calculate momentum)

  $v_t \leftarrow \psi_t(g_1, \cdots, g_t)$ (Calculate adaptive learning rate)

  $\theta_t \leftarrow \theta_{t-1} - \alpha_t m_t v_t$ (Update parameters)

return θ_T

- this is due to the lack of samples in the early stage
- previous solution: warmup - an initial period of training with a much lower learning rate

- empirical verification (see Fig. 1.1)

- x-axis: the histogram of the absolute value of gradients (on a log scale)
- y-axis: different iterations from 1st - 70k
- Up - ADAM without warmup: the distribution of the gradients changed/distorted a lot in the first a few iterations
- Bottom - ADAM with warmup: the distribution of the gradients did not change too much over 70k iterations
- This suggests that without applying warmup, Adam is trapped in bad/suspicious local optima after the first few updates.
- Controlled experiments: "Adam-2k"
  - first 2k iterations: only update the adaptive learning rate, while the momentum and the parameters are fixed
  - then add the vanilla Adam
  - "Adam-2k" achieves similar results as the Adam with warmup.
  - This supports the hypothesis that the lack of sufficient data samples in the

early stage is the root cause of the convergence issue.

- Given that warmup serves as a variance-reducer, but this is application-dependent
- RAdam: a dynamic variance reducer
  - see the paper to find how the analytical variance of the learning rate is computed
  - if the variance of the learning rate is tractable: update the parameters with adaptive momentum
    - where the adaptive learning rate is rectified w.r.t the estimated variance
  - if the variance is not tractable: update the parameters with un-adapted momentum

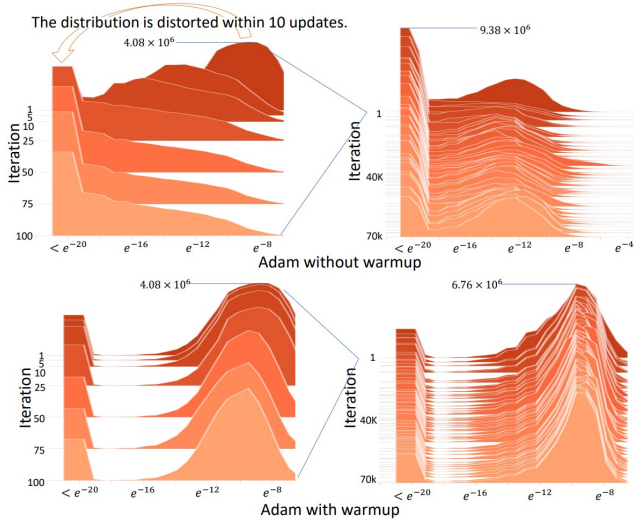


Figure 1.1: The histogram of the absolute value of gradients over iterations

Algorithm 2: Rectified Adam. All operations are element-wise.

Input: $\{\alpha_t\}_{t=1}^{T}$: step size, $\{\beta_1, \beta_2\}$: decay rate to calculate moving average and moving 2nd moment, θ_0: initial parameter, $f_t(\theta)$: stochastic objective function.

Output: θ_t: resulting parameters

$m_0, v_0 \leftarrow 0, 0$ (Initialize moving 1st and 2nd moment)

$\rho_\infty \leftarrow 2/(1-\beta_2) - 1$ (Compute the maximum length of the approximated SMA)

while $t = \{1, \cdots, T\}$ **do**

  $g_t \leftarrow \Delta_\theta f_t(\theta_{t-1})$ (Calculate gradients w.r.t. stochastic objective at timestep t)

  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ (Update exponential moving 2nd moment)

  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ (Update exponential moving 1st moment)

  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$ (Compute bias-corrected moving average)

  $\rho_t \leftarrow \rho_\infty - 2t\beta_2^t/(1-\beta_2^t)$(Compute the length of the approximated SMA)

  if *the variance is tractable, i.e.,* $\rho_t > 4$ **then**

    $\widehat{v_t} \leftarrow \sqrt{v_t/(1-\beta_2^t)}$ (Compute bias-corrected moving 2nd moment)

    $r_t \leftarrow \sqrt{\frac{(\rho_t-4)(\rho_t-2)\rho_\infty}{(\rho_\infty-4)(\rho_\infty-2)\rho_t}}$ (Compute the variance rectification term)

    $\theta_t \leftarrow \theta_{t-1} - \alpha_t r_t \widehat{m_t} / \widehat{v_t}$ (Update parameters with adaptive momentum)

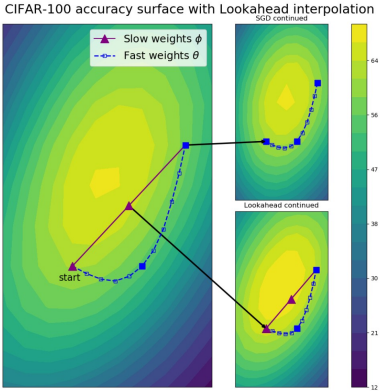
  else

    $\theta_t \leftarrow \theta_{t-1} - \alpha_t \widehat{m_t}$ (Update parameters with un-adapted momentum)

return θ_T

1.29 Lookahead

- **Lookahead**
  - Literature:
    - Lookahead Optimizer: k steps forward, 1 step back
    - authors: Michael R. Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba (author of ADAM)
  - Advantages:
    - less hyper-parameters tuning (works well with default params)
    - faster convergence rate
    - better results
  - Algorithm:
    - maintains two sets of weights: slow weights ϕ and fast weights θ



Algorithm 1 Lookahead Optimizer:

Require: Initial parameters ϕ_0, objective function L
Require: Synchronization period k, slow weights step size α, optimizer A
  for $t = 1, 2, \ldots$ **do**
    Synchronize parameters $\theta_{t,0} \leftarrow \phi_{t-1}$
    for $i = 1, 2, \ldots, k$ **do**
      sample minibatch of data $d \sim \mathcal{D}$
      $\theta_{t,i} \leftarrow \theta_{t,i-1} + A(L, \theta_{t,i-1}, d)$
    end for
    Perform outer update $\phi_t \leftarrow \phi_{t-1} + \alpha(\theta_{t,k} - \phi_{t-1})$
  end for
  return parameters ϕ

Figure 1.3: Lookahead algorithm

- the slow weights get synced with the fast weights every k updates

- the fast weights are updated through apply A, any standard optimization algorithm (on a minibatch samples D)
- After k inner optimizer updates using A, the slow weights are updated towards the fast weights by linear interpolation
- then the fast weights are reset to the current slow weights

- Comments:
  - standard optimization methods typically require carefully tuned learning rate to prevent oscillation and slow convergence
  - the fast weights updates makes rapid progress along the low curvature direction
  - the slow weights help smooth out the oscillation through the parameter interpolation
  - the combination of the fast weights and slow weights improves learning in high curvature directions, reduce variance, and improve convergence rate.
- RAdam + Lookahead: complementary to each other, better performance

1.30 Novograd

- Algortihm name: **Novograd**
  - Literature
    - Stochastic Gradient Methods with Layer-wise Adaptive Moments for Training of Deep Networks
    - Ginsburg et al.
  - First-order SGD-based algorithm, which computes second moments per layer instead of per weight as in Adam
  - Advantages:
    - robust to the choice of learning rate and weight initialization
    - works well in a large batch size
    - two times smaller memory footprint than Adam
  - Algorithm:

Algorithm 1 NovoGrad

Parameters: Initial learning rate λ_0, moments β_1, β_2, weight decay d, number of steps T
Weight initialization: $t = 0$, Initialize $\boldsymbol{w}_0$.
Moment initialization: $t = 1$, for each layer l set $v_1^l = ||\boldsymbol{g}_1^l||^2; \boldsymbol{m}_1^l = \frac{\boldsymbol{g}_1^l}{\sqrt{v_1^l}} + d \cdot \boldsymbol{w}_0^l$.

while $t \leq T$ **do**
  $\lambda_t \leftarrow LearningRateUpdate(\lambda_0, t, T)$ (compute the global learning rate)
  for each layer l **do**
    $\boldsymbol{g}_t^l \leftarrow \nabla_l L(w_t)$
    $v_t^l \leftarrow \beta_2 \cdot v_{t-1}^l + (1 - \beta_2) \cdot ||\boldsymbol{g}_t^l||^2$
    $\boldsymbol{m}_t^l \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1}^l + (\frac{\boldsymbol{g}_t^l}{\sqrt{v_t^l + \epsilon}} + d \cdot \boldsymbol{w}_t^l)$
    $\boldsymbol{w}_{t+1}^l \leftarrow \boldsymbol{w}_t^l - \lambda_t \cdot \boldsymbol{m}_t^l$
  end for
end while

- $\mathbf{g}_t^l$ is first used to compute the **layer-wise** 2nd moment v_t^l (not component/weight wise!)

$$v_t^l = \beta_2 v_{t-1}^l + (1 - \beta_2) \|\mathbf{g}_t^l\|^2$$

- Note in RMSProp or Adam, the 2nd moment $\mathbf{s}_k^j$ is component/weight wise.

- then the moment v_t^l is used to normalize the gradient $\mathbf{g}_t^l$: $\frac{\mathbf{g}_t^l}{\sqrt{v_t^l+\epsilon}}$
- then calculate the first moment $\mathbf{m}_t^l$ with the weight decay $d\mathbf{w}_t$ decoupled: $\mathbf{m}_t^l = \beta_1 \mathbf{m}_{t-1}^l + \frac{\mathbf{g}_t^l}{\sqrt{v_t^l+\epsilon}} + d\mathbf{w}_t$
- Finally the weights are updated with the first moment $\mathbf{m}_t^l$