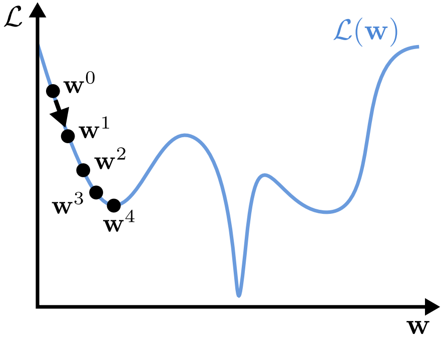
6.1
Optimization Challenges

By: Andreas Ginger

Optimization Challenges

Gradient Descent:

$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$$

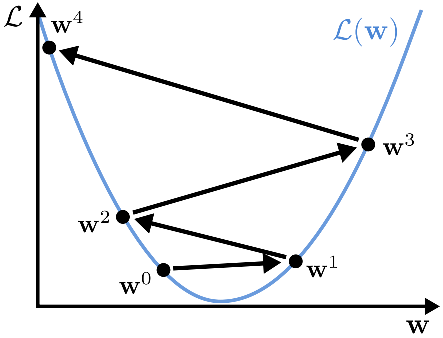


- Neural network loss $\mathcal{L}(\mathbf{w})$ is **not convex** wrt. the network parameters $\mathbf{w}$
- There exist **multiple local minima**, but we will find only one through optimization
- Example: we can permute all hidden units in a layer and get the **same solution**
- Good news: it is known that **many local minima** in deep networks **are good ones**

Optimization Challenges

Gradient Descent:

$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$$



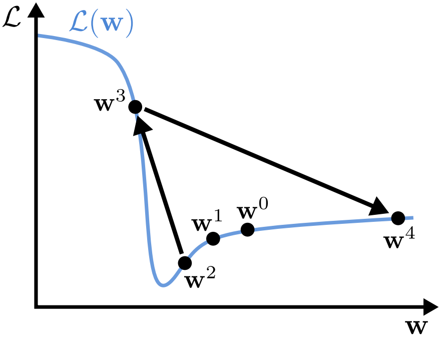
- ► Choosing the learning rate too low leads to very **slow progress**
- ► Choosing the learning rate too high might lead to **divergence**

Optimization Challenges

Gradient Descent:

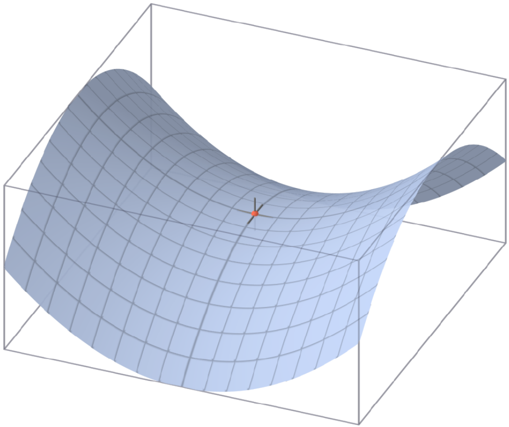
$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$$



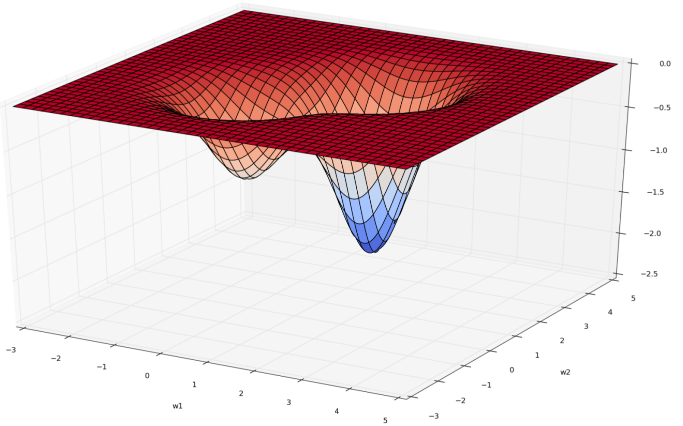
- Steep **cliffs** can pose great challenges to optimization
- Very high derivatives **catapult** the parameters $\mathbf{w}$ very far off
- **Gradient clipping** is a common heuristics to counteract such effects

Optimization Challenges



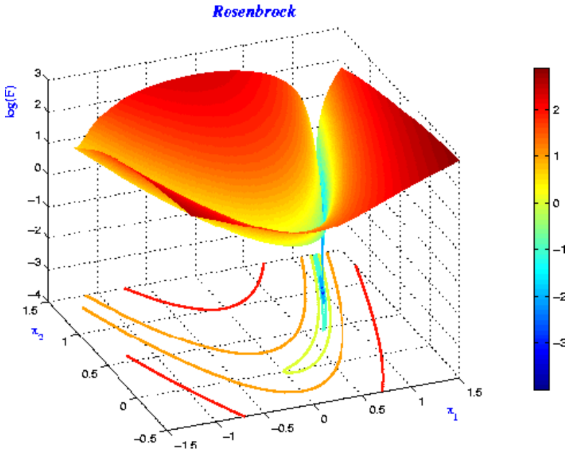
- At a **saddle point**, we have $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = \mathbf{0}$, but we are not at a minimum
- Many saddle points in DL, but only problematic if we exactly “hit” the saddle point

Optimization Challenges



- A flat region is called a **plateau** where $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) \approx \mathbf{0} \Rightarrow$ **Slow progress**
- Example: Saturated sigmoid activation function, dead ReLUs, etc.

Optimization Challenges



- ► Small gradient along the slope of the **valley** ⇒ hard to follow, easy to diverge

6.2
Optimization Algorithms

Gradient Descent

Algorithm:

1. Initialize weights $\mathbf{w}^0$ and pick learning rate η
2. For all data points $i \in \{1, \dots, N\}$ do:
   2.1 Forward propagate $\mathbf{x}_i$ through network to calculate prediction $\hat{\mathbf{y}}_i$
   2.2 Backpropagate to obtain gradient $\nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i, \mathbf{w}^t)$
3. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \frac{1}{N} \sum_i \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t)$
4. If validation error decreases, go to step 2, otherwise stop

Remark:

- Typically, millions of parameter $\Rightarrow \dim(\mathbf{w}) =$ 1 million or more
- Typically, millions of training points $\Rightarrow N =$ 1 million or more
- Becomes extremely expensive to compute and doesn't fit into memory

Stochastic Gradient Descent

Stochastic Gradient Descent

Solution:

- The total loss over the entire training set can be expressed as the expectation:

$$\frac{1}{N}\sum_i \mathcal{L}_i(\mathbf{w}^t) = \mathbb{E}_{i\sim\mathcal{U}\{1,N\}}\left[\mathcal{L}_i(\mathbf{w}^t)\right]$$

- This expectation can be approximated by a smaller subset $B \ll N$ of the data:

$$\mathbb{E}_{i\sim\mathcal{U}\{1,N\}}\left[\mathcal{L}_i(\mathbf{w}^t)\right] \approx \frac{1}{B}\sum_b \mathcal{L}_b(\mathbf{w}^t)$$

- Thus, the gradient can also be approximated:

$$\frac{1}{N}\sum_i \nabla_{\mathbf{w}}\mathcal{L}_i(\mathbf{w}^t) = \mathbb{E}_{i\sim\mathcal{U}\{1,N\}}\left[\nabla_{\mathbf{w}}\mathcal{L}_i(\mathbf{w}^t)\right] \approx \frac{1}{B}\sum_b \nabla_{\mathbf{w}}\mathcal{L}_b(\mathbf{w}^t)$$

Stochastic Gradient Descent

Remarks:

- We call $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_B, \mathbf{y}_B)\} \subseteq \mathcal{X}$ a “minibatch”
- You should choose B as large as your (GPU) memory allows [Goyal et al., 2017]
- Typically $B \ll N$, e.g., $B = 8, 16, 32, 64, 128$
- Smaller batch sizes lead to larger variance in the gradients (noisy updates)
- Batches can be chosen randomly or by partitioning the dataset (but they should be as independent as possible $\Rightarrow$ shuffle training set)

Terminology:

- Iteration = a single gradient update based on a single minibatch $\mathbf{w}^t \rightarrow \mathbf{w}^{t+1}$
- Epoch = complete pass through the training set ($= \frac{N}{B}$ iterations)

Goyal et al.: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. Arxiv, 2017.

Stochastic Gradient Descent

Algorithm:

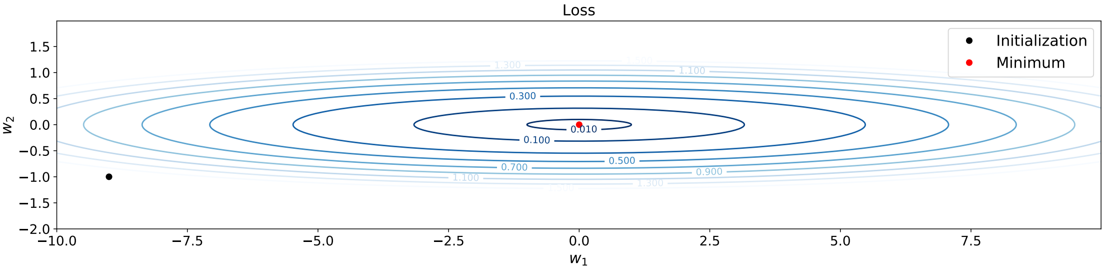
1. Initialize weights $\mathbf{w}^0$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw random minibatch $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_B, \mathbf{y}_B)\} \subseteq \mathcal{X}$ (with $B \ll N$)
3. For all minibatch elements $b \in \{1, \ldots, B\}$ do:
   3.1 Forward propagate $\mathbf{x}_b$ through network to calculate prediction $\hat{\mathbf{y}}_b$
   3.2 Backpropagate to obtain batch element gradient $\nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_b, \mathbf{y}_b, \mathbf{w}^t)$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$
5. If validation error decreases, go to step 2, otherwise stop

Remark:

- Allows for training with limited (GPU) memory, by splitting data into chunks
- Introduces stochasticity, batch gradients approximate the true gradient

Example

Stochastic Gradient Descent

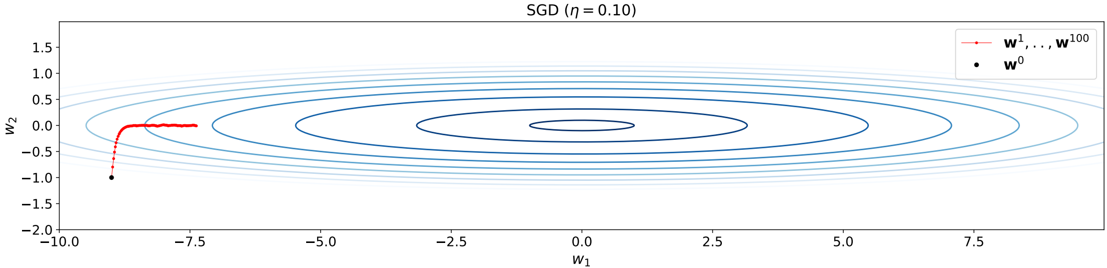


$$\mathcal{L}(\mathbf{w}) = (0.1\,w_1)^2 + w_2^2$$

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = (0.02\,w_1 \quad 2w_2)^\top + \mathcal{N}(0, 0.03)$$

- Toy example: Loss is 2D non-isotropic quadratic of parameter $\mathbf{w} = (w_1, w_2)^\top$
- To simulate minibatches, we have added Gaussian noise to the gradient

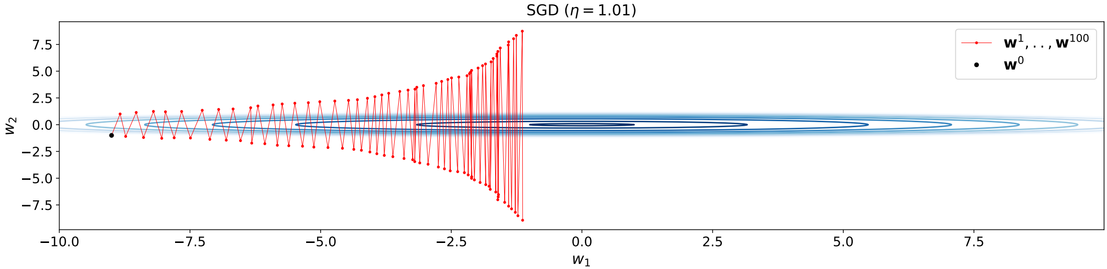
Stochastic Gradient Descent



$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$$

- Choosing the learning rate η too small does not lead to convergence (in 100 steps)

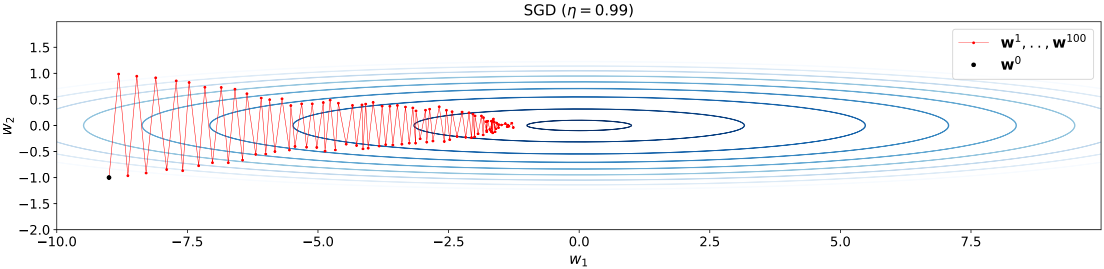
Stochastic Gradient Descent



$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$$

- Choosing the learning rate η too high leads to oscillations (also no convergence)

Stochastic Gradient Descent



$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$$

- A good learning rate η works better, but still slow, inefficient and no convergence

Finding the right Learning Rate

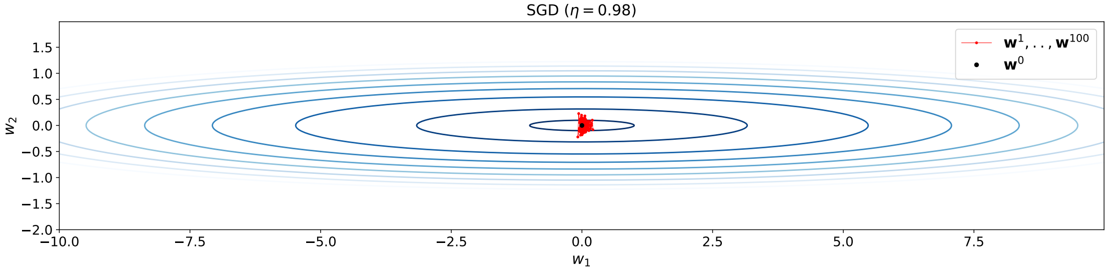
Line Search:

1. Compute minibatch gradient: $\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \equiv \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$
2. Find optimal step size: $\eta^* = \operatorname{argmin}_\eta \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t))$
3. Update weights: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta^* \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$

Remarks:

- Not practical for DL as we need to solve very large system at each step
- Not necessarily optimal step size of total loss as η^* is only the optimal step size for the current batch
- Thus, not used in practice

Convergence of SGD



$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$$

- Remark: Due to stochasticity, a fixed learning rate η will never lead to convergence
- In this example, the weights have been initialized at the optimum: $\mathbf{w}^0 = \mathbf{0}$

Convergence of SGD

A **series** is the sum of terms of an infinite sequence of numbers $(a_1, a_2, \dots)$

$$s_n = \sum_{k=1}^{n} a_k \qquad n \to \infty$$

A series is **convergent** if there exists a number s^* such that for every arbitrarily small positive number ϵ, there exists an integer N such that for all $n \geq N$:

$$\left| s_n - s^* \right| < \epsilon$$

Convergence of SGD

The SGD update leads to the following **parameter series:**

$$
\begin{aligned}
\mathbf{w}^{t+1} &= \mathbf{w}^{t} - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}(\mathbf{w}^{t})\\
\mathbf{w}^{1} &= \mathbf{w}^{0} - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{0}\\
\mathbf{w}^{2} &= \mathbf{w}^{1} - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{1} = \mathbf{w}^{0} - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{0} - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{1}\\
\mathbf{w}^{3} &= \mathbf{w}^{2} - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{2} = \underbrace{\mathbf{w}^{0} - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{0}}_{=\mathbf{a}_1} \underbrace{-\eta\,\nabla_{\mathbf{w}}\mathcal{L}_{1}}_{=\mathbf{a}_2} \underbrace{-\eta\,\nabla_{\mathbf{w}}\mathcal{L}_{2}}_{=\mathbf{a}_3}
\end{aligned}
$$

Optimization **converges** if there exists a vector $\mathbf{w}^*$ such that for every arbitrarily small positive number ϵ, there exists an integer T such that for all $t \geq T$:

$$\|\mathbf{w}^{t} - \mathbf{w}^{*}\| < \epsilon$$

Convergence of SGD

Theorem

Let $(\eta_1, \eta_2, \dots)$ be a sequence of positive step sizes with

$$\sum_{t=1}^{\infty} \eta_t = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

and let $\mathbf{g}_t$ be an unbiased estimate of $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$, i.e., $\mathbb{E}[\mathbf{g}_t] = \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$.

Then, the series

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \mathbf{g}_t \qquad t \to \infty$$

converges to a local minimum of $\mathcal{L}(\mathbf{w})$. Example: $\eta_t = \frac{\eta}{t}$

Robbins and Monro. A Stochastic Approximation Method. The Annals of Mathematical Statistics, 1951.

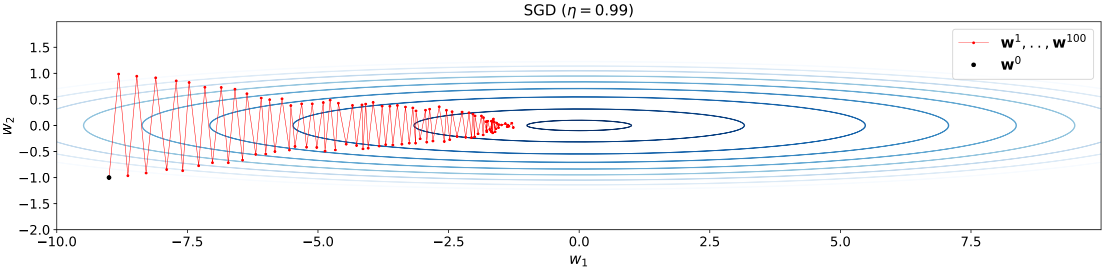
Problems of SGD

Problems of SGD:

- Gradient scaled equally across all dimensions
- Requires conservative learning rate to avoid divergence
- However, in this case the updates become very small $\Rightarrow$ slow progress
- Finding a good learning rate is difficult

SGD with Momentum

SGD with Momentum

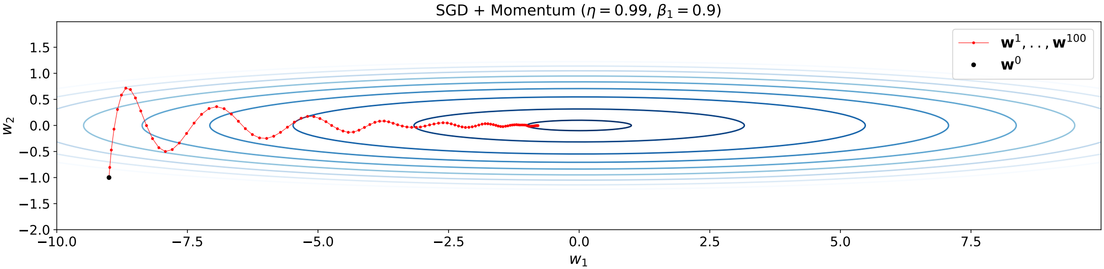


Motivation for SGD with Momentum:

- SGD oscillates along w_2 axis $\Rightarrow$ we should dampen, e.g., by averaging over time
- SGD makes slow progress along w_1 axis $\Rightarrow$ we like to accelerate in this direction
- Idea of momentum: update weights with exponential moving average of gradients

Sutskever, Martens, Dahl and Hinton: On the importance of initialization and momentum in deep learning. ICML, 2013.

SGD with Momentum



$$\mathbf{m}^{t+1} = \beta_1 \, \mathbf{m}^t - \eta \, \nabla_\mathbf{w} \mathcal{L}_\mathcal{B}(\mathbf{w}^t)$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \mathbf{m}^{t+1}$$

- With velocity $\mathbf{m}$ and momentum β_1, typically $\beta_1 = 0.9$

Sutskever, Martens, Dahl and Hinton: On the importance of initialization and momentum in deep learning. ICML, 2013.

SGD with Momentum

Traditionally, momentum is introduced as

$$\begin{aligned}\mathbf{m}^{t+1} &= \beta_1\,\mathbf{m}^t - \eta\,\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)\\ \mathbf{w}^{t+1} &= \mathbf{w}^t + \mathbf{m}^{t+1}\end{aligned}$$

However, this expression **couples** the momentum β_1 and learning rate η hyperparameters. A better parameterization is the following decoupled one:

$$\begin{aligned}\mathbf{m}^{t+1} &= \beta_1\,\mathbf{m}^t + (1-\beta_1)\,\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)\\ \mathbf{w}^{t+1} &= \mathbf{w}^t - \eta\,\mathbf{m}^{t+1}\end{aligned}$$

Sutskever, Martens, Dahl and Hinton: On the importance of initialization and momentum in deep learning. ICML, 2013.

Exponential Moving Average

Let us abbreviate the gradient at iteration t with $\mathbf{g}_t \equiv \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$. We have:

$$\mathbf{m}^{t+1} = \beta_1\, \mathbf{m}^t + (1-\beta_1)\, \mathbf{g}_t \quad (\text{with } \ \mathbf{m}^0 = \mathbf{0})$$

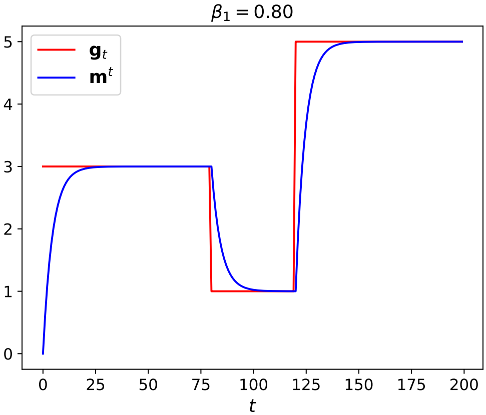
$$\begin{aligned}
\mathbf{m}^1 &= \beta_1\, \mathbf{m}^0 + (1-\beta_1)\, \mathbf{g}_0 = (1-\beta_1)\, \mathbf{g}_0 \\
\mathbf{m}^2 &= \beta_1\, \mathbf{m}^1 + (1-\beta_1)\, \mathbf{g}_1 \\
&= \beta_1\, (1-\beta_1)\, \mathbf{g}_0 + (1-\beta_1)\, \mathbf{g}_1 \\
\mathbf{m}^3 &= \beta_1\, \mathbf{m}^2 + (1-\beta_1)\, \mathbf{g}_2 \\
&= \beta_1^2\, (1-\beta_1)\, \mathbf{g}_0 + \beta_1 (1-\beta_1)\, \mathbf{g}_1 + (1-\beta_1)\, \mathbf{g}_2
\end{aligned}$$

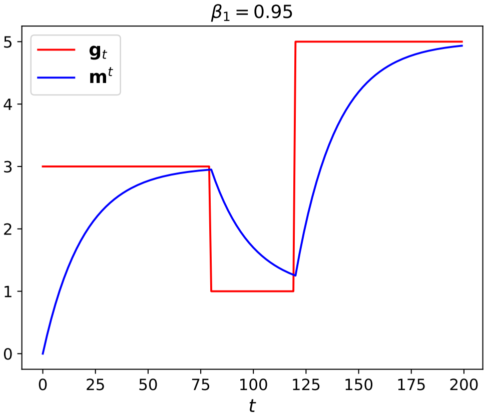
We see that the **weight decays exponentially:**

$$\mathbf{m}^t = (1-\beta_1) \sum_{i=0}^{t-1} \beta_1^{t-i-1}\, \mathbf{g}_i$$

Sutskever, Martens, Dahl and Hinton: On the importance of initialization and momentum in deep learning. ICML, 2013.

Exponential Moving Average

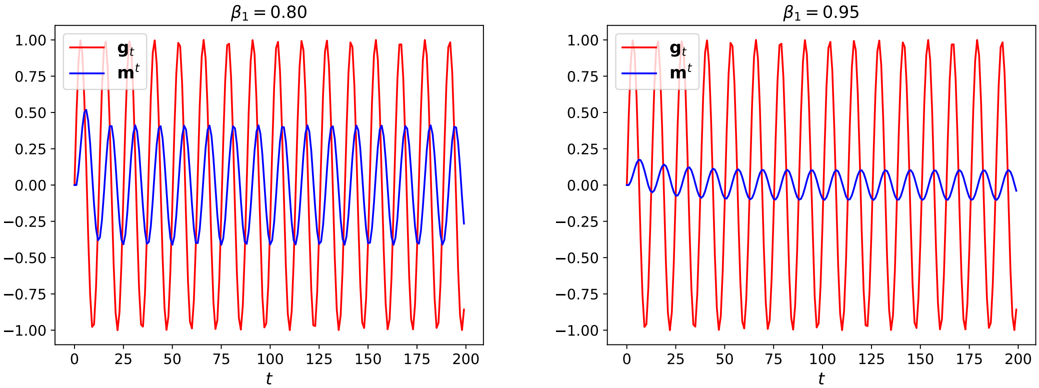




- The moving average effectively **dampens** the behavior of the function

Sutskever, Martens, Dahl and Hinton: On the importance of initialization and momentum in deep learning. ICML, 2013.

Exponential Moving Average



$\beta_1 = 0.80$

$\beta_1 = 0.95$

- ► The moving average effectively **dampens** the behavior of the function

Sutskever, Martens, Dahl and Hinton: On the importance of initialization and momentum in deep learning. ICML, 2013.

SGD with Nesterov Momentum

SGD with Nesterov Momentum

Better: **Look ahead** and calculate the gradient wrt. predicted parameters $\hat{\mathbf{w}}^{t+1}$:

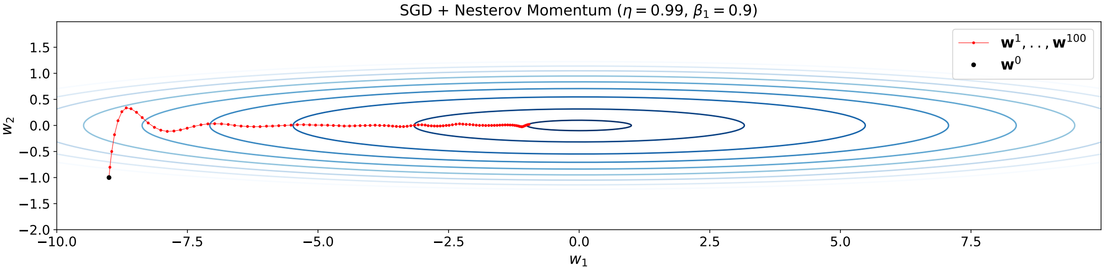
$$\begin{aligned}\hat{\mathbf{w}}^{t+1} &= \mathbf{w}^{t} + \beta_1 \mathbf{m}^{t} \\ \mathbf{m}^{t+1} &= \beta_1\, \mathbf{m}^{t} - \eta\, \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\hat{\mathbf{w}}^{t+1}) \\ \mathbf{w}^{t+1} &= \mathbf{w}^{t} + \mathbf{m}^{t+1}\end{aligned}$$

Again, this expression can be rewritten such that the hyperparameters are decoupled:

$$\begin{aligned}\hat{\mathbf{w}}^{t+1} &= \mathbf{w}^{t} - \eta\, \beta_1 \mathbf{m}^{t} \\ \mathbf{m}^{t+1} &= \beta_1\, \mathbf{m}^{t} + (1 - \beta_1)\, \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\hat{\mathbf{w}}^{t+1}) \\ \mathbf{w}^{t+1} &= \mathbf{w}^{t} - \eta\, \mathbf{m}^{t+1}\end{aligned}$$

Nesterov: A method for solving the convex programming problem with convergence rate O $(1/k^2)$. Dokl. akad. nauk Sssr. Vol. 269. 1983.

SGD with Nesterov Momentum

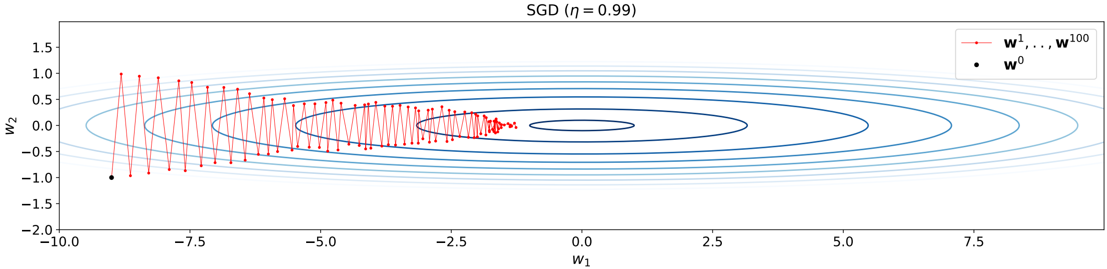


- This anticipatory update **increases responsiveness**
- Leads to **faster dampening**
- Has significantly increased the performance of RNNs on a variety of tasks

Nesterov: A method for solving the convex programming problem with convergence rate O $(1/k^2)$. Dokl. akad. nauk Sssr. Vol. 269. 1983.

RMSprop

RMSprop

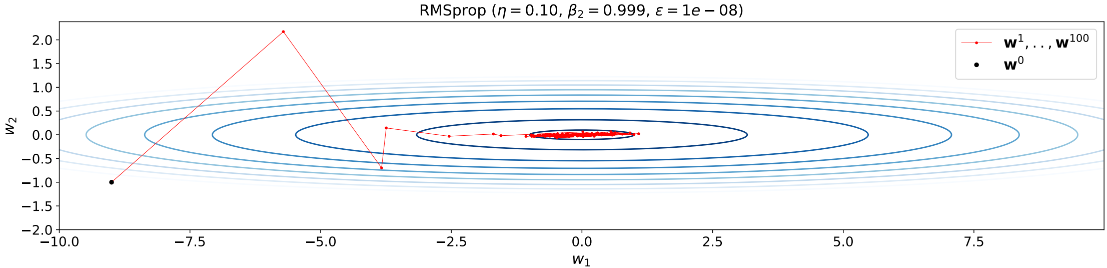


Motivation for RMSprop:

- Gradient distribution is very **uneven** and thus requires conservative learning rates
- In this SGD example, gradients are very large in w_2 but small in w_1
- Idea of RMSprop: **divide learning rate by moving average of squared gradients**

Hinton: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. 2012.

RMSprop



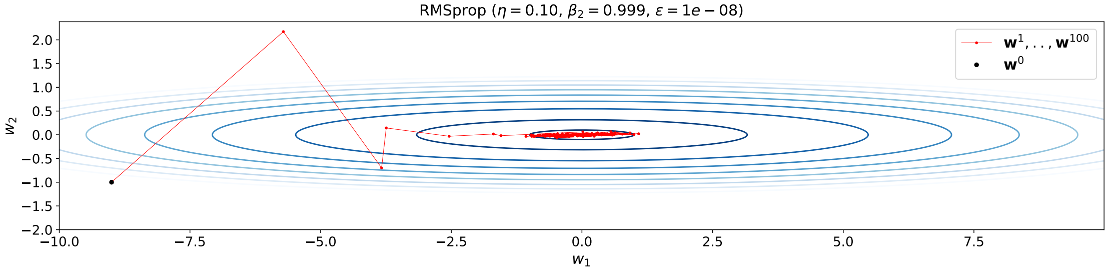
$$\mathbf{v}^{t+1} = \beta_2 \mathbf{v}^t + (1 - \beta_2) \left(\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \odot \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \right)$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \, \frac{\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)}{\sqrt{\mathbf{v}^{t+1}} + \epsilon}$$

- With uncentered variance of gradient $\mathbf{v}$, momentum $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$
- Note: All operations (division, square root) are **elementwise** operations

Hinton: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. 2012.

RMSprop



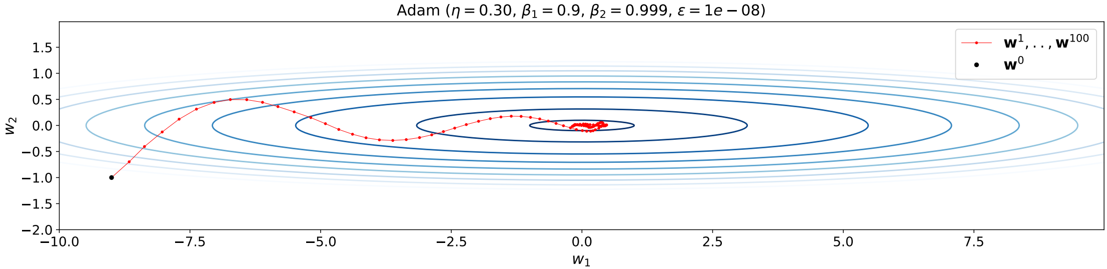
Remarks:

- Division by running average of squared gradients **adjusts per-weight step size**
  $\Rightarrow$ Division in w_2 direction will be large, division in w_1 direction will be small
- This **allows for increasing the learning rate** compared to vanilla SGD
- However: In first iterations, moving average biased towards zero

Hinton: Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. 2012.

Adam

Adam



$$\mathbf{m}^{t+1} = \beta_1\, \mathbf{m}^t + (1-\beta_1)\, \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$$

$$\mathbf{v}^{t+1} = \beta_2 \mathbf{v}^t + (1-\beta_2)\, \left(\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \odot \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)\right)$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\, \frac{\mathbf{m}^{t+1}}{\sqrt{\mathbf{v}^{t+1}} + \epsilon}$$

- Adam combines the benefits of Momentum and RMSprop

Kingma and Ba: Adam: A Method for Stochastic Optimization. ICLR, 2015.

Adam with Bias Correction

Adam Update Equation:

$$\begin{aligned}
\mathbf{m}^{t+1} &= \beta_1\,\mathbf{m}^t + (1-\beta_1)\,\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)\\
\mathbf{v}^{t+1} &= \beta_2\mathbf{v}^t + (1-\beta_2)\,\left(\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)\odot\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)\right)\\
\mathbf{w}^{t+1} &= \mathbf{w}^t - \eta\,\frac{\mathbf{m}^{t+1}}{\sqrt{\mathbf{v}^{t+1}}+\epsilon}
\end{aligned}$$

- What happens at $t = 1$?
- As $\mathbf{m}^0 = \mathbf{v}^0 = \mathbf{0}$, both $\mathbf{m}^1$ and $\mathbf{v}^1$ are strongly biased towards their initial value $\mathbf{0}$
- We should correct for this bias at the beginning of training

Kingma and Ba: Adam: A Method for Stochastic Optimization. ICLR, 2015.

Adam with Bias Correction

Adam Update Equation:

$$\mathbf{m}^{t+1} = \beta_1\, \mathbf{m}^{t} + (1-\beta_1)\, \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^{t})$$

$$\mathbf{v}^{t+1} = \beta_2 \mathbf{v}^{t} + (1-\beta_2)\left(\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^{t}) \odot \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^{t})\right)$$

$$\hat{\mathbf{m}}^{t+1} = \frac{\mathbf{m}^{t+1}}{1-\beta_1^{t+1}} \qquad \hat{\mathbf{v}}^{t+1} = \frac{\mathbf{v}^{t+1}}{1-\beta_2^{t+1}}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta\, \frac{\hat{\mathbf{m}}^{t+1}}{\sqrt{\hat{\mathbf{v}}^{t+1}} + \epsilon}$$

- Why is this modification removing the bias from the Adam update equations?

Kingma and Ba: Adam: A Method for Stochastic Optimization. ICLR, 2015.

Adam with Bias Correction

Let $\mathbf{g}_t = \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$ denote the gradient of the stochastic objective $\mathcal{L}(\mathbf{w}^t)$.
Let further $\mathbf{m}^0 = \mathbf{0}$. Then, the update $\mathbf{m}^{t+1} = \beta_1 \, \mathbf{m}^t + (1 - \beta_1) \, \mathbf{g}_t$ can be written as:

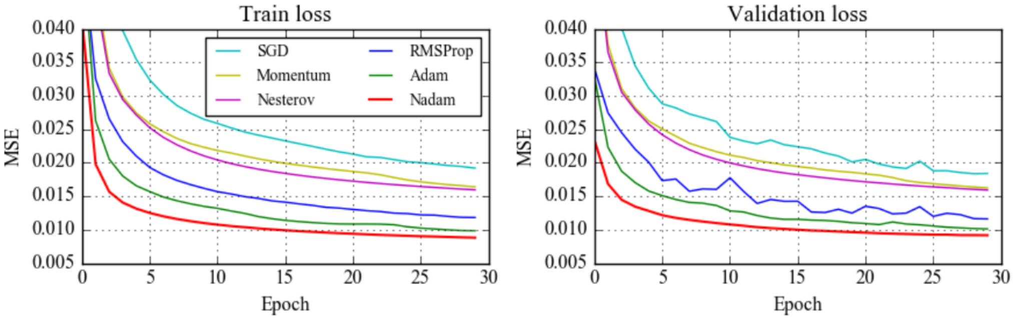
$$\mathbf{m}^t = (1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^{t-i-1} \, \mathbf{g}_i$$

Thus, the expectation over $\mathbf{m}^t$ is given by:

$$\begin{aligned} \mathbb{E}[\mathbf{m}^t] &= \mathbb{E}\left[(1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^{t-i-1} \, \mathbf{g}_i \right] \\ &\approx \mathbb{E}\left[\mathbf{g}_t\right] \cdot (1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^{i} \\ &= \mathbb{E}\left[\mathbf{g}_t\right] \cdot (1 - \beta_1^t) \end{aligned}$$

Kingma and Ba: Adam: A Method for Stochastic Optimization. ICLR, 2015.

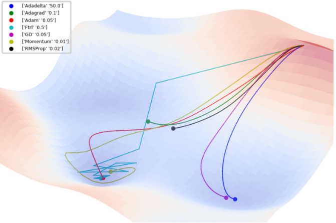
Adam with Nesterov Momentum

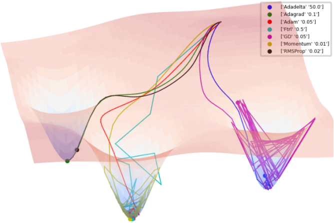


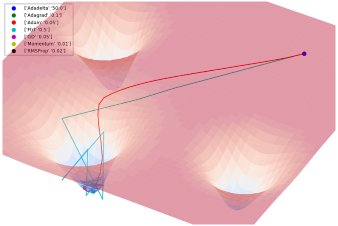
- It is also possible to combine Adam with Nesterov's Momentum
- However, this variant is less well known/less used in practice

Dozat: Incorporating Nesterov Momentum into Adam. 2016.

Visualization







https://github.com/Jaewan-Yun/optimizer-visualization

First-order Methods

There exist many variants:

- SGD
- SGD with Momentum
- SGD with Nesterov Momentum
- RMSprop
- Adagrad
- Adadelta
- Adam
- AdaMax
- NAdam
- AMSGrad

Adam is mostly the method of choice due to its robustness. Visualizations:

- Sebastian Ruder: `https://ruder.io/optimizing-gradient-descent/`
- Jaewan Yun: `https://github.com/Jaewan-Yun/optimizer-visualization`

Second-order Methods

Second-order methods:

- Newton
- Gauss-Newton
- BFGS / L-BFGS
- Levenberg-Marquardt

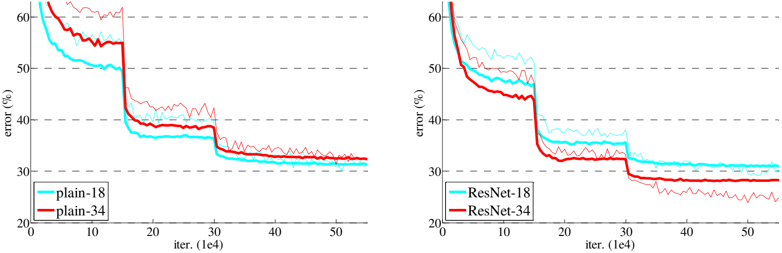
Remarks:

- Second-order methods are faster as they exploit the second derivative (curvature)
- However, not applicable to mini-batches as the Hessian estimates are too inaccurate if computed from small batches
- They are also not tractable as they require the inversion of a very large matrix
- Thus, second-order methods are typically not applied for training deep models

6.3
Optimization Strategies

Learning Rate Schedules

Learning Rate Schedules



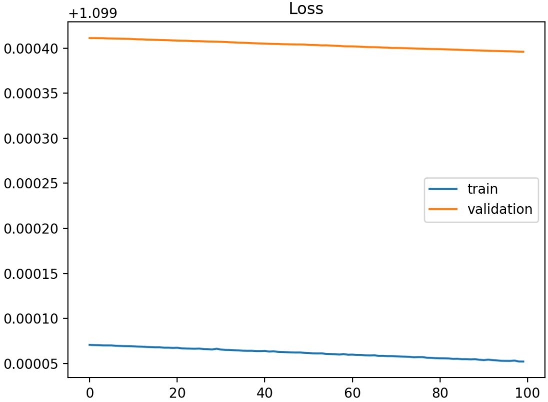
Learning rate schedules:

- Fixed learning rate (not a good idea: too slow in the beginning and fast in the end)
- Inverse proportional decay: $\eta_t = \eta / t$ (Robbins and Monro)
- Exponential decay: $\eta_t = \eta \alpha^t$
- Step decay: $\eta \leftarrow \alpha \eta$ (every K iterations/epochs, common in practice: $\alpha = 0.5$)

He, Zhang, Ren andSun: Deep Residual Learning for Image Recognition. CVPR, 2016.

Monitoring the Training Process

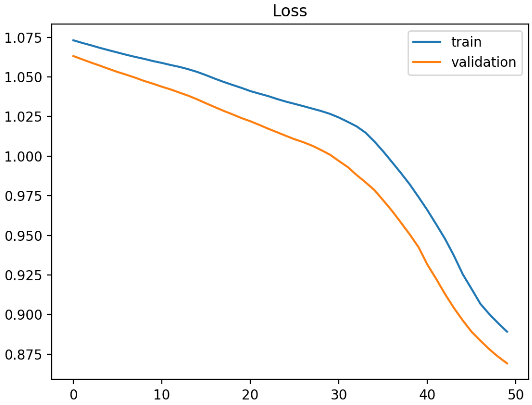
Monitoring the Training Process



- Underfitting: Model does not have enough capacity to decrease losses.

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

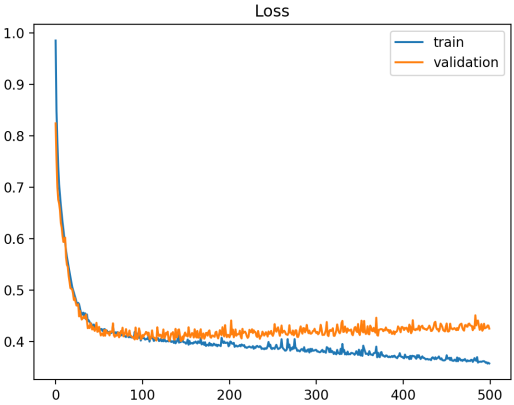
Monitoring the Training Process



- Not converged: Model requires more iterations to converge.

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

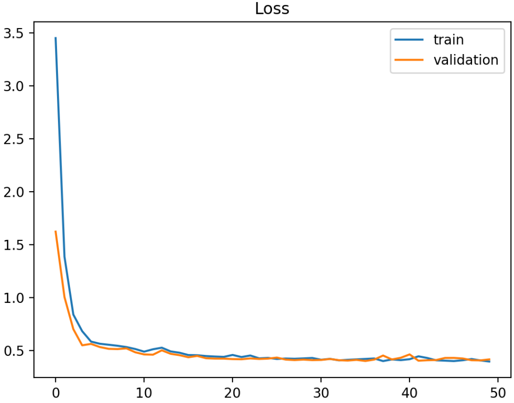
Monitoring the Training Process



- Overfitting: Training loss decreases but validation loss increases.

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

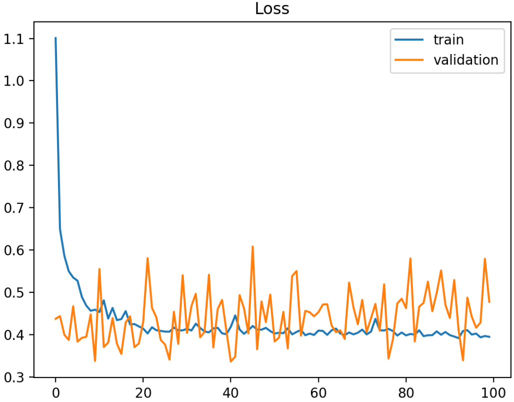
Monitoring the Training Process



- Example of train and validation curves showing a good fit.

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

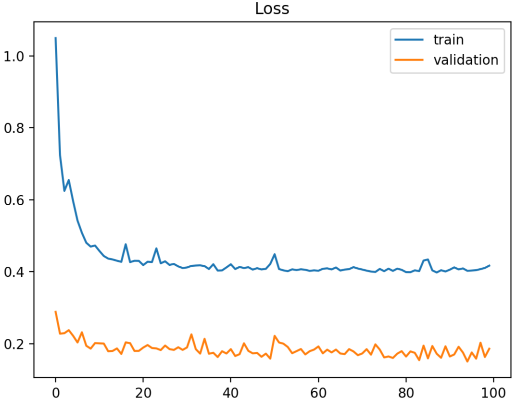
Monitoring the Training Process



- Noisy validation cuves: Validation set might be too small.

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

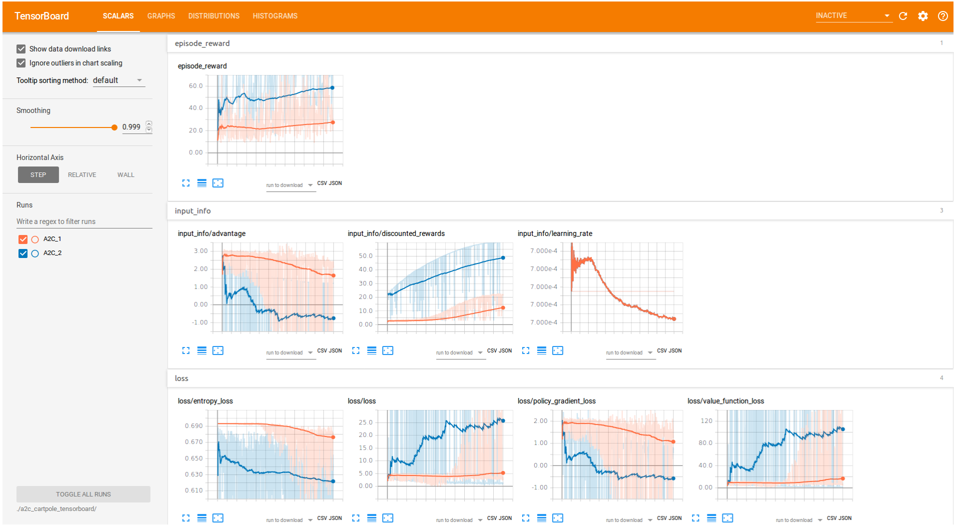
Monitoring the Training Process



- Might also happen: Validation set is easier than training set.

https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

Tensorboard



☑ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.999

Horizontal Axis: STEP

Runs

☑ A2C_1

☑ A2C_2

./a2c_cartpole_tensorboard/

`https://www.tensorflow.org/tensorboard`

Hyperparameter Search

Hyperparameter Search

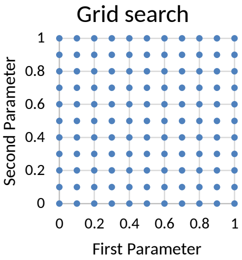
Hyperparameters:

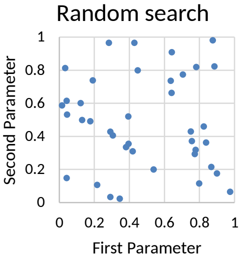
- Network architecture
- Number of iterations
- Batch size
- Learning rate schedule
- Regularization

Hyperparameter Search

Methods:

- Manual search
  - Most common
  - Build intuitions
- Grid search
  - Define ranges
  - Systematically evaluate
  - Requires large resources
- Random search
  - Like grid search but hyperparameters selected based on random draws





How to Start

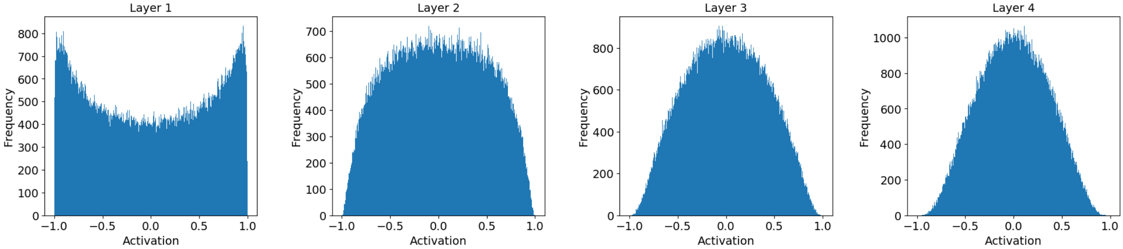
1. **Start with single training sample and use a small network**
   - First verify that the output is correct
   - Then overfit, accuracy should be 100%, fast training/debug cycles
   - Choose a good learning rate $(0.1, 0.01, 0.001, ..)$

2. **Increase to 10 training samples**
   - Again, verify that the output is correct
   - Measure time for one iteration $(< 1s) \Rightarrow$ identify bottlenecks (e.g., data loading)
   - Overfit to 10 samples, accuracy should be near 100%

3. **Increase to 100, 1000, 10000 samples and increase network size**
   - Plot train and validation error $\Rightarrow$ now you should start to see generalization
   - Important: Make only one change at a time to identify causes

Improving Gradient Flow

Initialization



Xavier/He Initialization:

- Initialize such that activation distribution constant Gaussian across all layers

Glorot and Bengio: Understanding the difficulty of training deep feedforward neural networks. AISTAT, 2010.

Batch Normalization

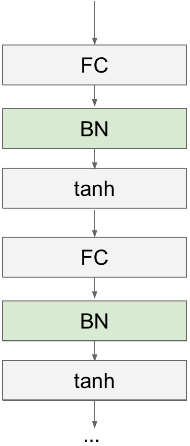
$$\mu_c = \frac{1}{B}\sum_{b=1}^{B} x_{b,c}$$

$$\sigma_c^2 = \frac{1}{B}\sum_{b=1}^{B} (x_{b,c} - \mu_c)^2$$

$$\hat{x}_{b,c} = \frac{x_{b,c} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

$$y_{b,c} = \gamma_c \, \hat{x}_{b,c} + \beta_c$$

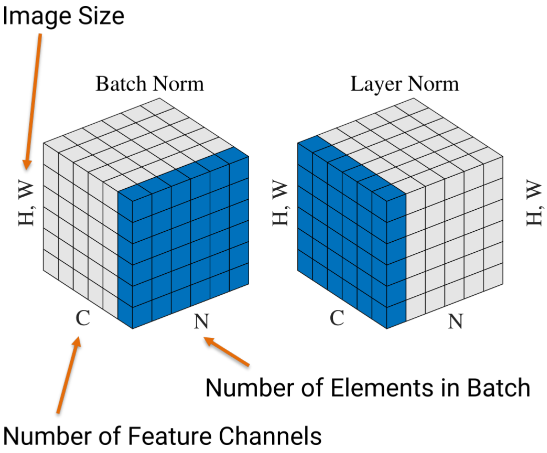
- Normalize each neuron (=channel) c by mean and variance over batch
- At test time: average during training
- Add learnable scale/shift parameter



- Insert before activation functions

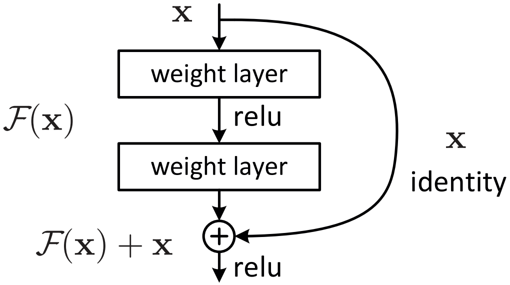
Ioffe and Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML, 2015.

Batch Normalization



Wu, He: Group Normalization. ECCV, 2018.

Residual Networks



- Learning very deep networks is hard due to slow propagation of gradients
- Observation: deeper networks often perform worse than more shallow ones
- Solution: **learn residual mappings** $\Rightarrow$ now it is much easier to realize the identity

He, Zhang, Ren and Sun: Deep Residual Learning for Image Recognition. CVPR, 2016.

Training Schedules

Training Schedules

Pretraining: (common practice)

- Pretrain backbone (all layers except last few layers) on another task for which a large dataset with labels is available
- Finetune last layers/full architecture on target task/dataset

Self Supervision:

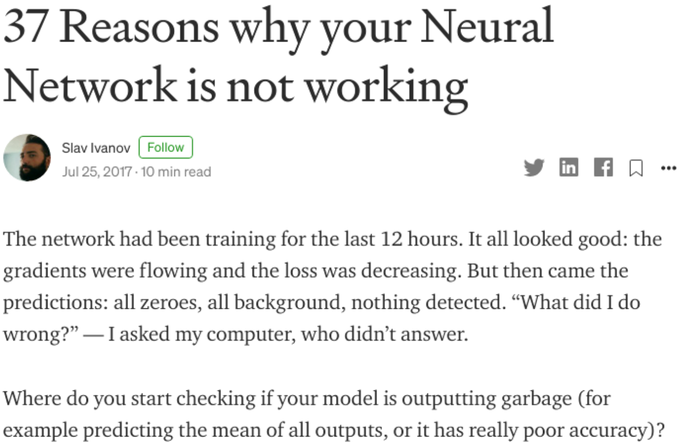
- Pretrain backbone on a task for which supervision is generated from the data itself (e.g., denoising, inpainting, contrastive learning)

Curriculum Learning:

- Start learning on easy datasets and successively increase difficulty

6.4
Debugging Strategies

Debugging Strategies



37 Reasons why your Neural Network is not working

Slav Ivanov

Jul 25, 2017 · 10 min read

The network had been training for the last 12 hours. It all looked good: the gradients were flowing and the loss was decreasing. But then came the predictions: all zeroes, all background, nothing detected. “What did I do wrong?” — I asked my computer, who didn’t answer.

Where do you start checking if your model is outputting garbage (for example predicting the mean of all outputs, or it has really poor accuracy)?

`https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607`

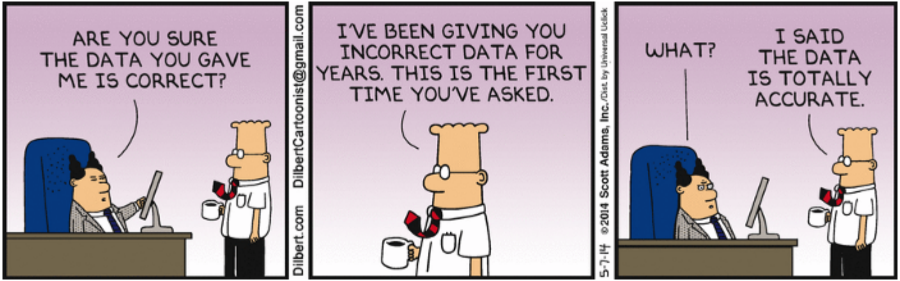
Emergency First Response

Emergency first response:

- Start with a simple model that is known to work for this type of data (for example, VGG for images). Use a standard loss if possible.
- Turn off all bells and whistles, e.g. regularization and data augmentation.
- If finetuning a model, double check the preprocessing,
  for it should be the same as the original model's training.
- Verify that the input data is correct.
- Start with a really small dataset (1–10 samples).
  Overfit on it and gradually add more data.
- Start gradually adding back all the pieces that were omitted:
  augmentation/regularization, custom loss functions, try more complex models.

Dataset Issues

Dataset Issues


ARE YOU SURE THE DATA YOU GAVE ME IS CORRECT?
I'VE BEEN GIVING YOU INCORRECT DATA FOR YEARS. THIS IS THE FIRST TIME YOU'VE ASKED.
WHAT?
I SAID THE DATA IS TOTALLY ACCURATE.
Dilbert.com DilbertCartoonist@gmail.com
5-7-14 ©2014 Scott Adams, Inc./Dist. by Universal Uclick

Check your input data

Check if the input data you are feeding the network makes sense. For example, I've more than once mixed the width and the height of an image. Sometimes, I would feed all zeroes by mistake. Or I would use the same batch over and over. So print/display a couple of batches of input and target output and make sure they are OK.

Dataset Issues

Try random input

Try passing random numbers instead of actual data and see if the error behaves the same way. If it does, it’s a sure sign that your net is turning data into garbage at some point. Try debugging layer by layer /op by op/ and see where things go wrong.

Check the data loader

Your data might be fine but the code that passes the input to the net might be broken. Print the input of the first layer before any operations and check it.

Make sure input is connected to output

Check if a few input samples have the correct labels. Also make sure shuffling input samples works the same way for output labels.

Dataset Issues

Verify noise in the dataset

This happened to me once when I scraped an image dataset off a food site. There were so many bad labels that the network couldn't learn. Check a bunch of input samples manually and see if labels seem off.

Shuffle the dataset

If your dataset hasn't been shuffled and has a particular order to it (ordered by label) this could negatively impact the learning. Shuffle your dataset to avoid this. Make sure you are shuffling input and labels together.

Reduce class imbalance

Are there a 1000 class A images for every class B image? Then you might need to balance your loss function or try other class imbalance approaches.

Dataset Issues

Verify number of training examples

If you are training a net from scratch (i.e. not finetuning), you probably need lots of data. For image classification, people say you need a 1000 images per class or more.

Make sure your batches don’t contain a single label

This can happen in a sorted dataset (i.e. the first 10k samples contain the same class). Easily fixable by shuffling the dataset.

Use a standard dataset

When testing new network architecture or writing a new piece of code, use the standard datasets first, instead of your own data. This is because there are many reference results for these datasets and they are proved to be ‘solvable’. There will be no issues of label noise, train/test distribution difference etc.

Data Normalization / Augmentation Issues

Data Normalization / Augmentation Issues

Standardize the features.

Did you standardize your input to have zero mean and unit variance?

Check for too much data augmentation

Augmentation has a regularizing effect. Too much of this combined with other forms of regularization (weight L2, dropout, etc.) can cause the net to underfit.

Check the preprocessing of your pretrained model

If you are using a pretrained model, make sure you are using the same normalization and preprocessing as the model was when training and testing. For example, should an image pixel be in the range [0, 1], [-1, 1] or [0, 255]?

Data Normalization / Augmentation Issues

Check the preprocessing for train/validation/test set

Any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation/test data. E.g. computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a mistake.

Implementation Issues

Implementation Issues

Try solving a simpler version of the problem

This will help with finding where the issue is. For example, if the target output is an object class and coordinates, try limiting the prediction to object class only.

Make sure your training procedure is correct

It is easy to forget toggling train/validation mode. Make sure you are using the right inputs. It is also easy to forget setting the gradients to zero before backpropagation.

Check your loss function

If you implemented your own loss function, check it for bugs and add unit tests. Often, wrong losses hurt the performance of the network in a subtle way.

Implementation Issues

Verify loss input

If you are using a loss function provided by your framework, make sure you are passing to it what it expects. For example, in PyTorch I would mix up the NLLLoss and CrossEntropyLoss as the former requires a softmax input and the latter doesn’t.

Adjust loss weights

If your loss is composed of several smaller loss functions, make sure their magnitude relative to each is correct. This might involve testing different loss weights.

Monitor other metrics

Sometimes the loss is not the best predictor of whether your network is training properly. If you can, use other metrics like accuracy.

Implementation Issues

Test any custom layers

Did you implement any of the layers in the network yourself? Check and double-check to make sure they are working as intended. Make sure the output has the right format (e.g., did you pass a softmax to a loss that expects raw logits?)

Check for “frozen” layers or variables

Check if you unintentionally disabled gradient updates for some layers/variables that should be learnable.

Increase network size

Maybe the expressive power of your network is not enough to capture the target function. Try adding more layers or more hidden units in fully connected layers.

Implementation Issues

Check for hidden dimension errors

If your input looks like (k, H, W) = (64, 64, 64) it’s easy to miss errors related to wrong dimensions. Use weird numbers for input dimensions (for example, different prime numbers for each dimension) and check how they propagate through the network.

Explore gradient checking

If you implemented Gradient Descent by hand, gradient checking makes sure that your backpropagation works like it should.

Training Issues

Training Issues

Solve for a really small dataset

Overfit a small subset of the data and make sure it works. For example, train with just 1 or 2 examples and see if your network can learn to differentiate these. Move on to more samples per class.

Check weights initialization

If unsure, use Xavier or He initialization. Also, your initialization might be leading you to a bad local minimum, so try a different initialization and see if it helps.

Change your hyperparameters

Maybe you using a particularly bad set of hyperparameters. Try a grid search.

Training Issues

Reduce regularization

Too much regularization can cause the network to underfit badly. Reduce regularization such as dropout, batch norm, weight/bias L2 regularization, etc.

Give it time

Maybe your network needs more time to train before it starts making meaningful predictions. If your loss is steadily decreasing, let it train some more.

Switch from Train to Test mode

Some frameworks have layers like Batch Norm, Dropout, and other layers behave differently during training and testing. Switching to the appropriate mode might help your network to predict properly.

Training Issues

Visualize the training process

- Monitor the activations, weights, and updates of each layer. Make sure their magnitudes match. For example, the magnitude of the updates to the parameters (weights and biases) should be 1-e3.
- Consider a visualization library like Tensorboard and Crayon. In a pinch, you can also print weights/biases/activations.
- Be on the lookout for layer activations with a mean much larger than 0. Try Batch Norm or ELUs.
- Weight histograms should have an approximately Gaussian (normal) distribution, after some time. For biases, these histograms will generally start at 0, and will usually end up being approximately Gaussian. Keep an eye out for parameters that are diverging or biases that become very large.

Training Issues

Try a different optimizer

Your choice of optimizer shouldn’t prevent your network from training unless you have selected particularly bad hyperparameters. However, the proper optimizer for a task can be helpful in getting the most training in the shortest amount of time.

Exploding / Vanishing gradients

Check layer updates, as very large values can indicate exploding gradients. Gradient clipping may help. Check layer activations. A good standard deviation for the activations is on the order of 0.5 to 2.0. Significantly outside of this range may indicate vanishing or exploding activations.

Training Issues

Increase/Decrease Learning Rate

A low learning rate will cause your model to converge very slowly. A high learning rate will quickly decrease the loss in the beginning but might have a hard time finding a good solution. Play around with your current learning rate by multiplying it by 0.1 or 10.

Overcoming NaNs

- Decrease the learning rate, especially if you are getting NaNs in the first 100 iterations.
- NaNs can arise from division by zero or natural log of zero or negative number.
- Try evaluating your network layer by layer and see where the NaNs appear.