Neural Networks and Deep Learning

Optimization II, Intro to Autoencoders

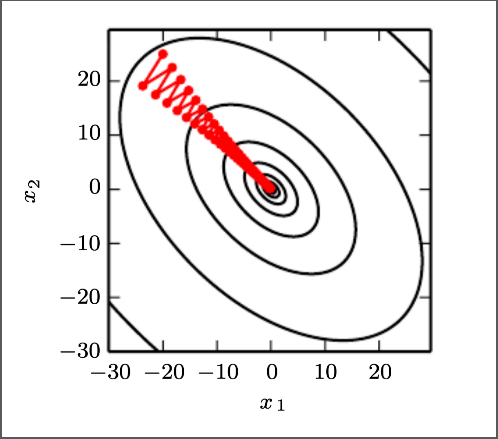
Adam Bloniarz

Department of Computer Science
adam.bloniarz@colorado.edu

March 30, 2020

University of Colorado **Boulder**

What's wrong with SGD?



Source: deeplearningbook.org, Figure 4.6

What's wrong with SGD?

Maybe nothing?

See The marginal value of adaptive gradient methods in machine learning

- SGD and adaptive methods arrive at very different solutions.
- SGD slower to minimize training error, but this paper argues that its solutions generalize better.
- "Our construction suggests that adaptive methods tend to give undue influence to spurious features that have no effect on out-of-sample generalization."

Controversy over optimization for deep learning algorithms continues, as was the case in 2012.

Hinton lecture

Momentum method (heavy ball method)

Imagine the parameter vector is a particle on a surface, subject to two forces:

- The gradient.
- A viscosity proportional to its velocity.

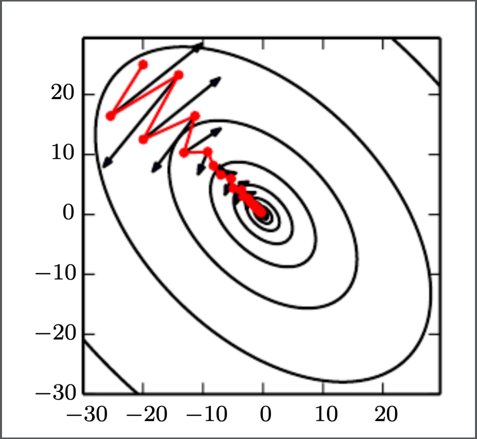
Momentum smooths out oscillating gradients of opposite sign, and accelerates the particle in directions with consistent gradients.

Hinton lecture

Momentum method: Let l be the function to be optimized, and let w_k be the parameter vector.

$$w_{k+1} = w_k - \alpha_k \nabla l(w_k) + \beta_k (w_k - w_{k-1})$$

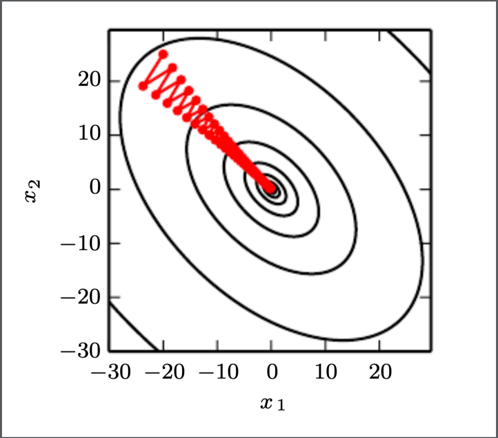
(in Hinton lecture, α_k and β_k are constants).



Source: deeplearningbook.org, Figure 8.5

Adaptive methods

Idea: each weight in the neural net should have its own learning rate.



Source: deeplearningbook.org, Figure 4.6

Hinton lecture

RMSProp: divide the gradient by the square root of an exponentially weighted average of its square.

Parameters with consistently large gradients end up with a smaller learning rate. Parameters with consistently small gradients end up with a larger learning rate.

Well-adapted to mini-batch stochastic gradient descent in neural networks.

Exponentially-weighted moving average

Given a series $x_1, x_2, \ldots, x_i, \ldots,$ the exponentially-weighted moving average is equal to

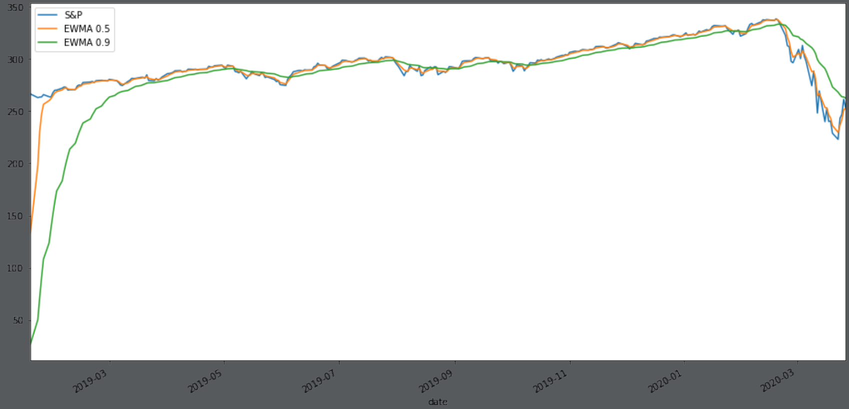
$$v_i = \beta v_{i-1} + (1-\beta)x_i$$

Unrolling this, we have

$$v_i = (1-\beta)\sum_{k=1}^{i} \beta^{i-k} x_k$$

Exponentially-weighted moving average

Initializing the series at 0:



Note the early bias due to initialization!

RMSprop: divide the gradient by the square root of an exponentially weighted average of its square.
Let $\nabla l(x_t)$ be the gradient of the loss at iteration t. We update a the moving average of the square of the gradient. Let $\circ$ denote element-wise multiplication of two vectors.

$$v_t = \beta v_{t-1} + (1-\beta)[\nabla l(w_{t-1}) \circ \nabla l(w_{t-1})]$$

The weight update is

$$w_t \leftarrow w_{t-1} - \eta \frac{\nabla l(w_{t-1})}{\sqrt{v_t} + \epsilon}$$

Where the division is carried out element-wise.

RMSprop is very similar to AdaGrad (Duchi, Hazan, Singer 2011), a very popular algorithm developed in the convex optimization literature.

$$v_t = v_{t-1} + [\nabla l(w_{t-1}) \circ \nabla l(w_{t-1})]$$

The weight update is

$$w_t \leftarrow w_{t-1} - \eta \frac{\nabla l(w_{t-1})}{\sqrt{v_t} + \epsilon}$$

Adagrad provably leads to faster convergence rates in the case of sparse features. Rarely seen but highly predictive features are given a relatively large learning rate.

Adam: Adaptive moment estimation (Kingma, Ba 2014)

Combines momentum and adaptive learning rates in a clever way.

The closest thing there is to a default optimization method for deep learning.

Adam

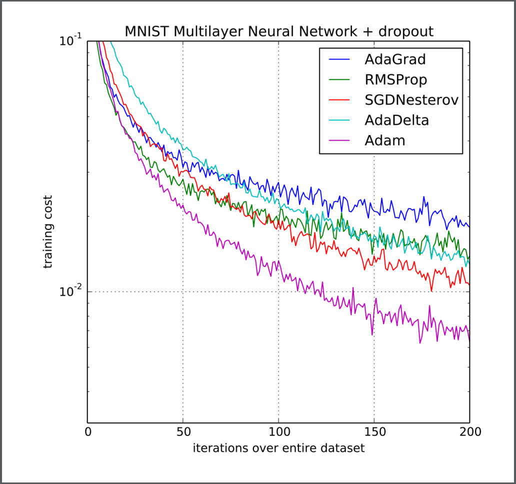
Maintain an exponentially weighted moving estimate of the first and second moments of the gradient.

$$\left.\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla l(w_{t-1}) \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)[\nabla l(w_{t-1}) \circ \nabla l(w_{t-1})] \end{aligned}\right\} \text{Update moment estimates}$$

$$\left.\begin{aligned} \hat{m}_t &= m_t/(1-\beta_1^t) \\ \hat{v}_t &= v_t/(1-\beta_2^t) \end{aligned}\right\} \text{Correct bias}$$

Update weights:

$$w_t \leftarrow w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$



Source: Kingma, Ba 2014, Figure 2

Sensible to have inputs normalized to mean 0 and standard deviation 1.

- $E[\hat{x}_i] = \frac{1}{p} \sum_{\alpha=1}^{p} \hat{x_{i,\alpha}} = 0$
  $\hat{x_{i,\alpha}}$: transformed input dimension i for training example α
- $E[\hat{x}_i{}^2] = \frac{1}{p} \sum_{\alpha=1}^{p} \hat{x_{i,\alpha}}^2 = 1$

To achieve this, compute mean μ_i and std dev σ_i for each input dimension i over training set.

- $\hat{x_{i,\alpha}} = \frac{x_{i,a} - \mu_i}{\sigma_i}$

For test set, need to apply same transformation.

- Use μ_i and σ_i from training set.
- They need to be stored along with network weights.

- When the input distribution to a learning system changes, it is said to experience covariate shift.
- BN is based on the premise that covariate shifts complicate the training of machine learning systems, and removing it from internal activations of the network may aid in training.
- Advantages
  - » Beneficial effect on gradient flow by reducing the dependence of gradients on the scale of the parameters or of their initial values. This allows the use of much higher learning rates without the risk of divergence.
  - » Claim: BN reduces the need for regularization

- Input

$$\text{Values of } x \text{ over a mini-batch : } x_i \ldots x_m$$

$$\text{Parameters to be learned : } \gamma, \beta$$

- Output

$$\text{Mini-batch mean : } \mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\text{Mini-batch variance : } \sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$

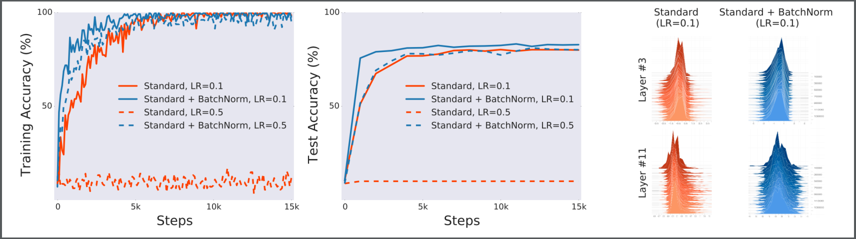
$$\text{Normalize : } \hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$$\text{Scale and shift : } y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$$

The mechanism of batch norm is controversial. Its effectiveness is not.

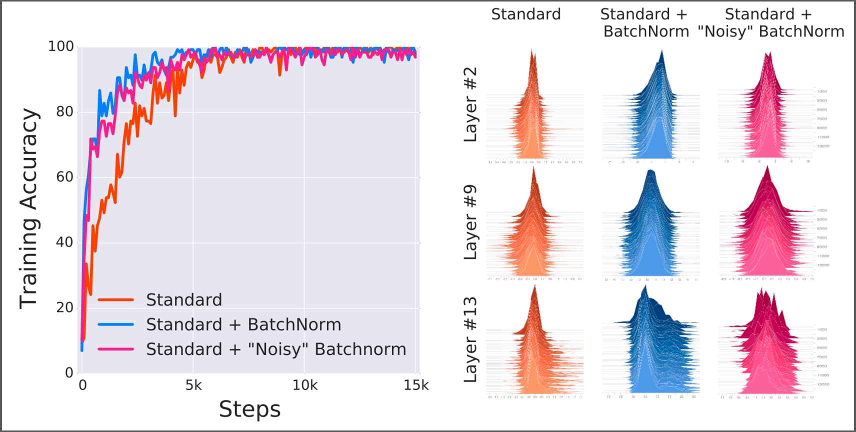
See How Does Batch Normalization Help Optimization?

- "Our point of start is demonstrating that there does not seem to be any link between the performance gain of BatchNorm and the reduction of internal covariate shift."
- " ... BatchNorm impacts network training in a fundamental way: it makes the landscape of the corresponding optimization problem significantly more smooth."



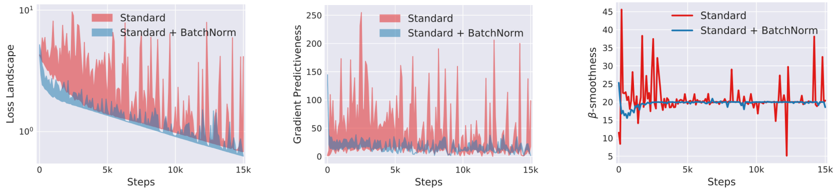
Source: Santurkar et al., 2017, Figure 1

Batch norm does not seem to reduce covariate shift.



Source: Santurkar et al., 2017, Figure 2

Advantages of batch norm persist even when *inducing* covariate shift after batch-norm.



(a) loss landscape (b) gradient predictiveness (c) "effective" β-smoothness

Source: Santurkar et al., 2017, Figure 4

Batch norm clearly smoothens the loss and gradient landscapes.

In unsupervised learning, we seek to build a useful representation of the data *without labels*. Such a representation can be useful for:

- Use unsupervised pre-training to reduce the need for training data on a supervised task.
- Decompose a signal into independent sources of variation.
- Find clusters of data.
- Given a dataset, train a machine to be able to generate samples of new data from the same distribution.
- Given a dataset, estimate a density function. Can be used for outlier / anomaly detection.

An **autoencoder** is a kind of neural network that learns to copy its input.

A deterministic autoencoder consists of two functions:

- Encoder function $\mathbf{h} = \mathbf{f}(\mathbf{x})$
- Decoder function $\mathbf{x}' = \mathbf{g}(\mathbf{h})$

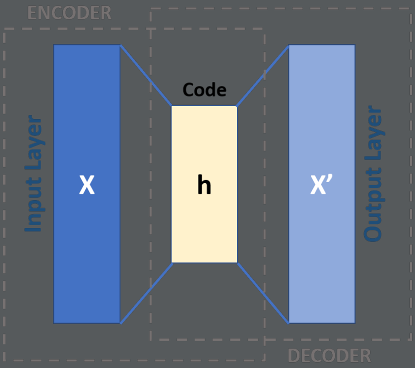
The autoencoder is trained to minimize a reconstruction error $L(\mathbf{x}, g(f(\mathbf{x})))$.

Often, the squared error loss is used:

$$L(\mathbf{x}, g(f(\mathbf{x}))) = \|\mathbf{x} - \mathbf{g}(\mathbf{f}(\mathbf{x}))\|^2$$

Undercomplete autoencoders

One strategy is to force the network to learn a low-dimensional representation of the data.



Consider a linear, undercomplete autoencoder. Consider the extreme case of representing an input with a single number. Let $\mathbf{x}_i \in \mathbb{R}^d, \mathbf{v} \in \mathbb{R}^d, \mathbf{w} \in \mathbb{R}^d$. Ignore the bias terms.

$$h_i = \mathbf{v}^T \mathbf{x}_i$$
$$\mathbf{x}_i' = \mathbf{w} h$$

Pack the dataset into a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$. We want to solve:

$$\min_{\mathbf{v},\mathbf{w}} \left\| \mathbf{X} - \mathbf{X}\mathbf{v}\mathbf{w}^T \right\|_F^2$$

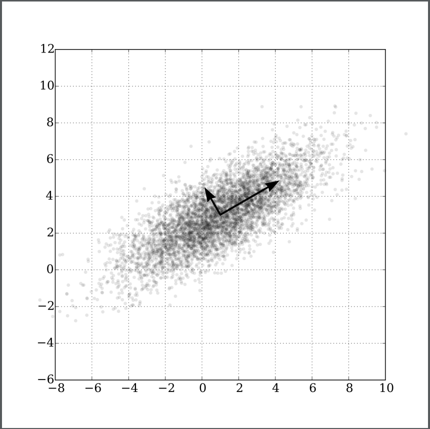
Note that $\mathbf{X}\mathbf{v}\mathbf{w}^T$ is a rank-1 matrix. So this problem boils down to solving

$$\min_{\mathbf{v},\mathbf{w}} \|\mathbf{X} - \mathbf{B}\|_F^2 \text{ s.t. } \mathbf{B} \text{ is rank } 1.$$

This is solved by setting $\mathbf{v} = \mathbf{w} =$ the first right-singular vector of $\mathbf{X}$.

See Theorem 3.6 in Foundations of data science: the best rank-k approximation of $\mathbf{X}$ is given by the first k singular vectors of $\mathbf{X}$.

The singular value decomposition of $\mathbf{X}$ corresponds to the principal component analysis (PCA) of $\mathbf{X}$.
PCA decomposes a signal into uncorrelated components such that for all k, the first k components account for the largest possible variance of $\mathbf{X}$.



If we train a linear auto-encoder

$$\min_{\mathbf{v},\mathbf{w}} \left\| \mathbf{X} - \mathbf{X}\mathbf{v}\mathbf{w}^T \right\|_F^2$$

with standard NN tools (e.g. minibatch SGD), it will not ‘know’ about eigenvalues or SVD, but if all goes well, it finds a solution such that $\mathbf{v}\mathbf{w}^T$ projects $\mathbf{X}$ into the space spanned by the first singular vector.