Initialization, Optimization, and Regularization

Moritz Wolter

September 27, 2023

High-Performance Computing and Analytics Lab, University of Bonn

Overview

Initialization

Network optimization

Overfitting

Regularization

Code snippets

Motivation

- Neural network optimization is non-convex. We aren't guaranteed to find a global optimum.
- The outcome depends on the starting point and the optimizer.
- Consequently, one should choose the best possible starting point and think about how to best traverse the optimization landscape.
- Both initialization and optimization are hot research topics.
- As you will see, the science is by no means settled.

Initialization

Glorot-Initialization

For a layer with m input dimensions and n output dimensions. A very common initialization was suggested by Glorot [GB10],

$$w_{ij} \sim \mathcal{U}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \tag{1}$$

For all possible position indices i, j. $\mathcal{U}$ denotes a Uniform distribution. ML-Frameworks generally implement pseudo-random versions of all major distributions.

He-Initialization

He-uniform-initialization [He+15] is the default for Linear-Layers in Pytorch.

$$w_{ij} \sim \mathcal{U}\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right), \tag{2}$$

this heuristic is also recommended in [GBC16].

LeCun-Normal-Initialization

For linear layers, Jax and Flax work with truncated normal distributions,

$$w_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{1}{n}}\right) \tag{3}$$

by default. $\mathcal{N}(\mu, \sigma)$ denotes the standard normal distribution with mean μ and standard deviation σ. Outliers are redrawn if they are larger than 2σ.

It is generally ok to stick to the default set by your favorite framework.

Summary

- We saw three different ways to initialize neural networks.
- Initialization is an active research matter.
- It is usually a good idea to stick to your framework's default.

Network optimization

Gradient Descent

Following the literature [GBC16, chapter 8], the vector θ will denote all learnable network parameters. This simplification makes it easier to write the general ideas down. Gradient descent now looks like this,

$$\mathbf{g}_\tau = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} C(f(\mathbf{x}^i; \theta), \mathbf{y}^\mathbf{i}), \tag{4}$$

$$\theta_{\tau+1} = \theta_\tau - \epsilon \mathbf{g}_\tau. \tag{5}$$

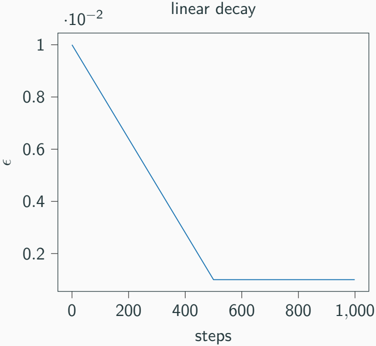
With the step counter τ, gradient operator ∇, cost function C, inputs $\mathbf{x}$, and outputs $\mathbf{y}$. It is efficient to process multiple batches at once. m denotes the batch size and ϵ the step size.

Learning rate decay

[GBC16, chapter 8] recommends linear decay until step τ

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{6}$$

with $\alpha = \frac{k}{\tau}$. After step τ the step size typically remains the same.



An example would be i.e. $\epsilon_0 = 0.01$, $\epsilon_\tau = 0.001$ and $\tau = 500$.

Momentum

Momentum helps the optimizer to traverse through locally minimal valleys, the formulation turns into,

$$\mathbf{g}_\tau = \frac{1}{m}\nabla_\theta \sum_{i=0}^{m} C(f(\mathbf{x}^i;\theta),\mathbf{y}^{\mathbf{i}}), \tag{7}$$

$$\mathbf{v}_\tau = \alpha\mathbf{v}_{\tau-1} - \epsilon\mathbf{g}_\tau, \tag{8}$$

$$\theta_{\tau+1} = \theta_\tau - \mathbf{v}_\tau. \tag{9}$$

A new velocity term $\mathbf{v}$ appeared. Use $\mathbf{v}_{-1} = 0$.

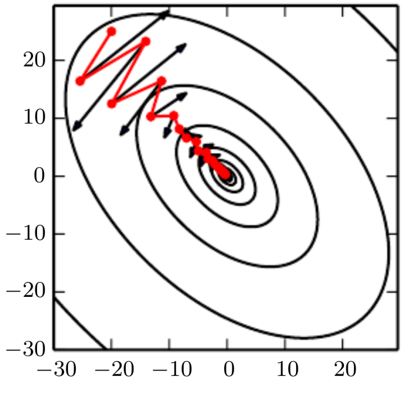


Figure: Illustration of the gradient steps taken by an optimizer with momentum. Image taken from [GBC16].

An adaptive learning rate: RMSprop

The RMSprop method works especially well when recurrent connections are present. It uses the update steps,

$$\mathbf{g}_\tau = \frac{1}{m} \nabla_\theta \sum_{i=0}^{m} C(f(\mathbf{x}^i; \theta), \mathbf{y}^\mathbf{i}), \tag{10}$$

$$\mathbf{r}_\tau = \mathbf{r}_{\tau-1} + \mathbf{g}_\tau \odot \mathbf{g}_\tau, \tag{11}$$

$$\theta_{\tau+1} = \theta_\tau - \frac{\epsilon}{\delta + \mathbf{r}_\tau} \odot \mathbf{g}. \tag{12}$$

The key novelty here is to scale the learning rate ϵ adaptively for every step. δ is a small number used to avoid division by zero.

The default: Adam

Adam (adaptive moments) introduces an additional scaling term,

$$\mathbf{g}_\tau = \frac{1}{m}\nabla_\theta \sum_{i=0}^{m} C(f(\mathbf{x}^i;\theta),\mathbf{y}^\mathbf{i}), \tag{13}$$

$$\mathbf{s}_\tau = \rho_1 \mathbf{s}_{\tau-1} + (1-\rho_1)\mathbf{g}_\tau \tag{14}$$

$$\mathbf{r}_\tau = \rho_2 \mathbf{r}_{\tau-1} + (1-\rho_2)\mathbf{g}_\tau \odot \mathbf{g}_\tau \tag{15}$$

$$\hat{\mathbf{s}}_\tau = \frac{\mathbf{s}_\tau}{1-\rho_1} \tag{16}$$

$$\hat{\mathbf{r}}_\tau = \frac{\mathbf{r}_\tau}{1-\rho_2} \tag{17}$$

$$\theta_{\tau+1} = \theta_\tau - \frac{\epsilon \hat{\mathbf{s}}_\tau}{\delta + \hat{\mathbf{r}}_\tau} \odot \mathbf{g}. \tag{18}$$

Adam combines the Rmsprop-idea with momentum. Major deep learning frameworks implement adam for you. Use `optax.adam` in today's exercise.

Summary

We saw the three optimizers that usually appear in the literature.

- Carefully tuned gradient descent with momentum can deliver excellent performance.
- RMSprop adds stability. Especially for hard problems, for example with recurrent connections.
- Adam often runs reliably for a wide range of problems.
- The best optimizers don't help us if we are overfitting.

Generally speaking, the question of the ideal optimizer choice is unsettled [GBC16].

Overfitting

Overfitting and early stopping

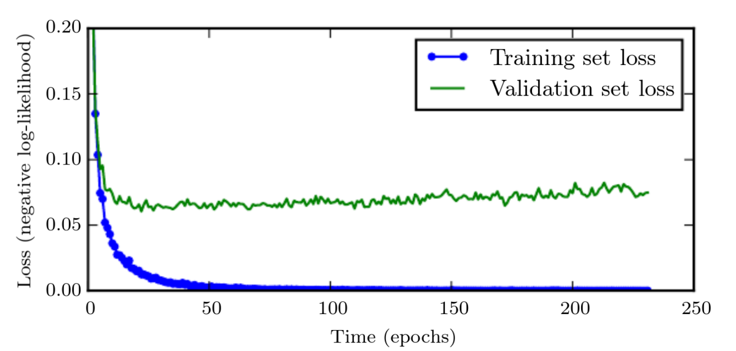


Figure: Overfitting of a CNN on the MNIST data set. Figure from [GBC16].

Extending the data-set

Collecting more data is the most elegant way to prevent overfitting.
If collecting more data is impossible artificial extensions can help.

- Input-noise
- Input transforms
  Consider, for example, an image:
  - random crops,
  - random left-right flips,
  - or small random rotations,

  are ways to avoid looking at an identical image again.

Regularization

Motivation

- Guard against overfitting.
- Improve generalization.
- Instead of regularization or additionally, it is also sometimes a good idea to reduce the number of weights.

L2-Norm Regularization

The idea here is to encourage sparsity in the weights by adding,

$$C_w(\theta) = \lambda_r \sum_{i=1}^{w} |\mathbf{W}_i|_2 \tag{19}$$

To the cost function. w denotes the total number of weight objects in the network. The scaling factor $\lambda_r \in \mathbb{R}$ must chosen by hand. This is Thikonov-regularization, the machine learning way.

Dropout

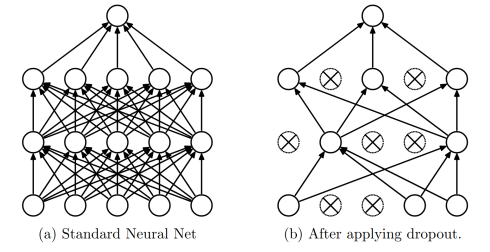


Figure: Dropout as described in [Sri+14].

Idea: Randomly remove connections **during training**.

Batch-Norm

Idea: Normalize before every layer and optimize a scale and shift separately [IS15]:

$$\hat{x}_{ij}^{(l)} = \frac{x_{ij}^{(l)} - \mu_x^{(l)}}{\sigma_x^{(l)}} \tag{20}$$

$$\tilde{x}_{ij}^{(l)} = \gamma \hat{x}_{ij}^{(l)} + \beta^{(l)} \tag{21}$$

Where $\mathbf{x}^{(l)}$ denotes the input at layer l, while $\mu_x^{(l)}$ and $\sigma_x^{(l)}$ are the batch mean and standard deviation. $\gamma^{(l)}$ and $\beta^{(l)}$ must be learned for each layer. For every feature position i, j up to the feature height and width.

Summary

- Regularization is sometimes required.
- Before spending a lot of time on regularization reduce the model size.
- Look for models for your problem in the literature.
- Most of the time a regularizer is already built in.

Conclusion

- We saw the most important initialization methods,
- the most important optimizers,
- and some regularization.
- Let's talk about Unets!

References

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

Code snippets

Using Optax-Optimizers

All common optimizers are available in the Optax library:

https://optax.readthedocs.io/en/latest/

Look for Adam in the documentation!

```
# creating an optimizer
opt = optax.adam(learning_rate=0.001)
# initializing an optimizer
opt_state = opt.init(weights)
# comuting an update.
updates, opt_state = opt.update(
  grads, opt_state, weights)
# apllying a weight update.
weights = optax.apply_updates(
  weights, updates)
```