Introduction to Deep Learning

Copernicus Master on Digital Earth

Lecture 5: How to train a network?

Dr. Charlotte Pelletier
charlotte.pelletier@univ-ubs.fr

Content

- Parameter initialisations
- Vanishing and exploding gradient issues
- Gradient descent and its variants
- Regularization techniques
  - lambda regularization
  - early stopping
  - dropout
  - batch-normalization

Parameter initialisations

Parameter initialisations

For a standard MultiLayer Perceptron (MLP)

Forward pass

$$\begin{aligned} z^{(l)} &= w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} &= g(z^{(l)}) \end{aligned} \quad , \forall l = 1, \cdots, L$$

where $x^0 = x$.

Backward pass

$$\frac{\partial \mathcal{L}}{\partial w^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(l)}} x^{(l-1)} \qquad \frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(l)}}$$

where

$$\frac{\partial \mathcal{L}}{\partial z^{(l)}} = \frac{\partial \mathcal{L}}{\partial x^{(l)}} \odot g'(z^{(l)})$$

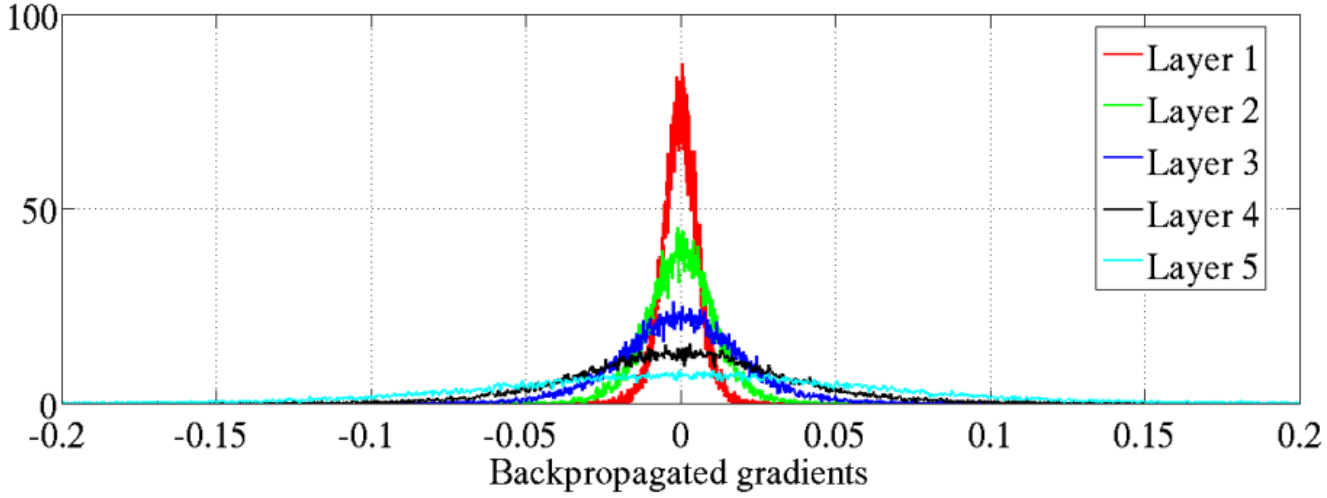
If $l < L$,

$$\frac{\partial \mathcal{L}}{\partial x^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial x^{(l)}} = w^{(l+1)} \frac{\partial \mathcal{L}}{\partial z^{(l+1)}}$$

Thus, we have

$$\frac{\partial \mathcal{L}}{\partial x^{(l)}} = w^{(l+1)} \left(\frac{\partial \mathcal{L}}{\partial x^{(l+1)}} \odot g' \left(z^{(l+1)} \right) \right)$$

So the gradient "vanishes" exponentially with the depth if the weights w are ill-conditioned or the activations are in the saturating domain of g.



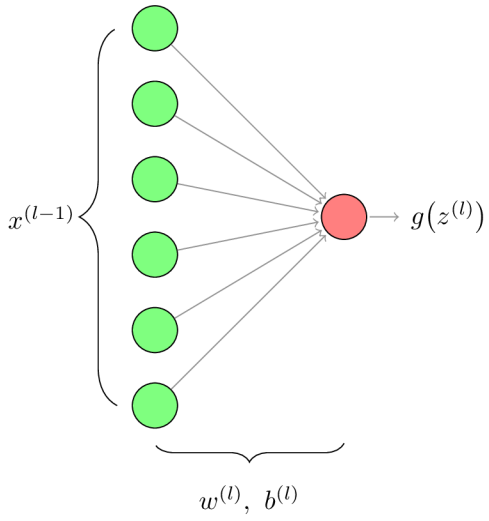
In mathematics, an ill-conditioned problem is one where, for a small change in the inputs (the independent variables or the right-hand-side of an equation) there is a large change in the answer or dependent variable. Source: https://en.wikipedia.org/wiki/Condition_number.

Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256).

The design of the weight initialization aims at controlling the variance of the parameters w and b so that they evolve at the same rate across layers during training, and no layer reaches a saturation behavior before others

The design of the weight initialization aims at controlling the variance of the parameters w and b so that they evolve at the same rate across layers during training, and no layer reaches a saturation behavior before others

Let us take the example of a hidden layer comprised of one neuron:



$$z^{(l)} = w_1^{(l)} x_1^{(l-1)} + w_2^{(l)} x_2^{(l-1)} + w_3^{(l)} x_3^{(l-1)} + w_4^{(l)} x_4^{(l-1)} + w_5^{(l)} x_5^{(l-1)} + \cdots + b$$

Let us denote by $K^{(l-1)}$ the number of inputs of the layer:

$$z^{(l)} = \sum_{i=1}^{K^{(l-1)}} w_i^{(l)} x_i^{(l-1)} + b$$

Thus, the value of $z^{(l)}$ depends on $K^{(l-1)}$.

As we do not know nothing about the data, a good solution is to assign the weights from a Gaussian distribution with a zero mean and some finite variance.

$$\mathbb{V}\left(w_i^{(l)} x_i^{(l-1)}\right) = \mathbb{E}\left(x_i^{(l-1)}\right)^2 \mathbb{V}\left(w_i^{(l)}\right) + \mathbb{E}\left(w_i^{(l)}\right)^2 \mathbb{V}\left(x_i^{(l-1)}\right) + \mathbb{V}\left(w_i^{(l)}\right)\mathbb{V}\left(x_i^{(l-1)}\right)$$

We assume here that $w_i^{(l)}$ and $x_i^{(l-1)}$ are all identically and independently distributed (iid), so we can work out the variance of $z^{(l)}$:

$$\begin{aligned}
\mathbb{V}\left(z^{(l)}\right) &= \mathbb{V}\left(w_1^{(l)} x_1^{(l-1)} + w_2^{(l)} x_2^{(l-1)} + \cdots + \right) \\
&= \mathbb{V}\left(w_1^{(l)} x_1^{(l-1)}\right) + \mathbb{V}\left(w_2^{(l)} x_2^{(l-1)}\right) + \cdots \\
&= K^{(l-1)} \mathbb{V}\left(w_i^{(l)}\right)\mathbb{V}\left(x_i^{(l-1)}\right)
\end{aligned}$$

As we do not know nothing about the data, a good solution is to assign the weights from a Gaussian distribution with a zero mean and some finite variance.

$$\mathbb{V}\left(w_i^{(l)} x_i^{(l-1)}\right) = \mathbb{E}\left(x_i^{(l-1)}\right)^2 \mathbb{V}\left(w_i^{(l)}\right) + \mathbb{E}\left(w_i^{(l)}\right)^2 \mathbb{V}\left(x_i^{(l-1)}\right) + \mathbb{V}\left(w_i^{(l)}\right)\mathbb{V}\left(x_i^{(l-1)}\right)$$

We assume here that $w_i^{(l)}$ and $x_i^{(l-1)}$ are all identically and independently distributed (iid), so we can work out the variance of $z^{(l)}$:

$$\begin{aligned}
\mathbb{V}\left(z^{(l)}\right) &= \mathbb{V}\left(w_1^{(l)} x_1^{(l-1)} + w_2^{(l)} x_2^{(l-1)} + \cdots + \right) \\
&= \mathbb{V}\left(w_1^{(l)} x_1^{(l-1)}\right) + \mathbb{V}\left(w_2^{(l)} x_2^{(l-1)}\right) + \cdots \\
&= K^{(l-1)} \mathbb{V}\left(w_i^{(l)}\right)\mathbb{V}\left(x_i^{(l-1)}\right)
\end{aligned}$$

The variance of the output $z^{(l)}$ is thus the variance of the input $x^{(l-1)}$, but it is scaled by $K^{(l-1)}\mathbb{V}\left(w_i^{(l)}\right)$.

If we want the variance of the output $z^{(l)}$ to be equal to the variance of the input $x^{(l-1)}$, the term $K^{(l-1)}\mathbb{V}\left(w_i^{(l)}\right)$ should be equal to 1.

Thus,

$$\mathbb{V}\left(w_i^{(l)}\right) = \frac{1}{K^{(l-1)}}$$

If we want the variance of the output $z^{(l)}$ to be equal to the variance of the input $x^{(l-1)}$, the term $K^{(l-1)}\mathbb{V}\left(w_i^{(l)}\right)$ should be equal to 1.

Thus,

$$\mathbb{V}\left(w_i^{(l)}\right) = \frac{1}{K^{(l-1)}}$$

Similarly, if we go through backpropagation, we have:

$$\mathbb{V}\left(w_i^{(l)}\right) = \frac{1}{K^{(l)}}$$

If we want the variance of the output $z^{(l)}$ to be equal to the variance of the input $x^{(l-1)}$, the term $K^{(l-1)}\mathbb{V}\left(w_i^{(l)}\right)$ should be equal to 1.

Thus,

$$\mathbb{V}\left(w_i^{(l)}\right) = \frac{1}{K^{(l-1)}}$$

Similarly, if we go through backpropagation, we have:

$$\mathbb{V}\left(w_i^{(l)}\right) = \frac{1}{K^{(l)}}$$

As usually $K^{(l-1)}$ is different from $K^{(l)}$, a compromise is to use the average:

$$\mathbb{V}\left(w_i^{(l)}\right) = \frac{2}{K^{(l-1)} + K^{(l)}}$$

Classical parameter initialisations

Xavier (Glorot) initialisation

The idea of Xavier (Glorot) initialisation is then to initialize weights from Gaussian distribution with zero-mean and variance:

$$\mathbb{V}\left(w_i^{(l)}\right) = \sqrt{\frac{2}{K^{(l-1)} + K^{(l)}}}$$

He initialisation

Glorot and Bengio (2010) considered the logistic sigmoid activation function for their initialisation scheme, which was the default one in 2010. However, it is not quite well adapted for the most common ReLU activation function. He et al. (2015) proposed a tiny adjustment, which is to multiply the variance of the weights by 2:

$$\mathbb{V}\left(w_i^{(l)}\right) = \sqrt{\frac{4}{K^{(l-1)} + K^{(l)}}}$$

Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth International Conference on Artificial Intelligence and Statistics (pp. 249-256).
He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE International Conference on Computer Vision (pp. 1026-1034).

Data normalisation

The analysis for the weight initialization relies on keeping the activation variance constant.

For this to be true, not only the variance has to remain unchanged through layers, but it has to be correct for the input too.

This is why, we traditionally mean-centre the input data:

```
>>> mu, std = X_train.mean(0), X_train.std(0)
>>> X_train.sub_(mu).div_(std)
>>> X_test.sub_(mu).div_(std)
```

Gradient Descent Algorithm

To find a minimum of the loss function, the gradient descent algorithm iteratively updates the parameters as follows:

$$w_{t+1} = w_t - \alpha \nabla \mathcal{L}(w_t, b_t)$$

where α is the learning rate, t the epoch number and ∇ the gradient.

The total loss is averaged over all the training instances:

$$\begin{aligned} \mathcal{L} &= \frac{1}{m} \sum_{i=1}^{m} \ell(\hat{y}_i, y_i) \\ &= \frac{1}{m} \sum_{i=1}^{m} \ell_i \end{aligned}$$

where ℓ can be the mean-squared error, the cross-entropy loss, etc.

A straight-forward implementation would be

```
for e in range(1,nb_epochs+1):
  output = model(X_train)
  loss = criterion(output, Y_train)
  model.zero_grad()
  loss.backward()
  with torch.no_grad():
    for p in model.parameters(): p -= alpha * p.grad
```

While it makes sense in principle to compute the gradient exactly, in practice:

- it requires to process all the training data in the network (forward pass) before applying the gradient descent algorithm,
- the model parameters are thus not updated often, resulting in a network that improves slowly
- and it takes time to compute the full-gradient (more exactly all the time!)
- it is an empirical estimation of an hidden quantity, and any partial sum is also an unbiased estimate, although of greater variance.

Stochastic Gradient Descent

The stochastic gradient descent (SGD) algorithm updates parameters performed after every sample:

$$w_{t+1} = w_t - \alpha \nabla \ell_i, \forall i = 1, \cdots, m$$

However, SGD does not benefit from matrix vectorization, and thus parallel computing that speeds up the training process.

Mini-batch Gradient Descent

The mini-batch gradient descent is the standard procedure for deep learning.
It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time

$$w_{t+1} = w_t - \alpha \sum_{x \in \mathcal{B}_i} \nabla \ell_x, \forall i = 1, \cdots, \frac{m}{B}$$

where $\mathcal{B}_i$ is a mini-batch of size B.

The batch-size B is a hyperparameter that sets the trade-off between the speed of the training and the progress made by the network at each iteration.
Its value is usually in the range of $\{8, 16, 32, 64, 128\}$.

Mini-batch Gradient Descent

The mini-batch gradient descent is the standard procedure for deep learning.
It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time

$$w_{t+1} = w_t - \alpha \sum_{x \in \mathcal{B}_i} \nabla \ell_x, \forall i = 1, \cdots, \frac{m}{B}$$

where $\mathcal{B}_i$ is a mini-batch of size B.

The batch-size B is a hyperparameter that sets the trade-off between the speed of the training and the progress made by the network at each iteration.
Its value is usually in the range of $\{8, 16, 32, 64, 128\}$.

The stochastic behavior of this procedure helps evade local minima.

Vocaburlary note

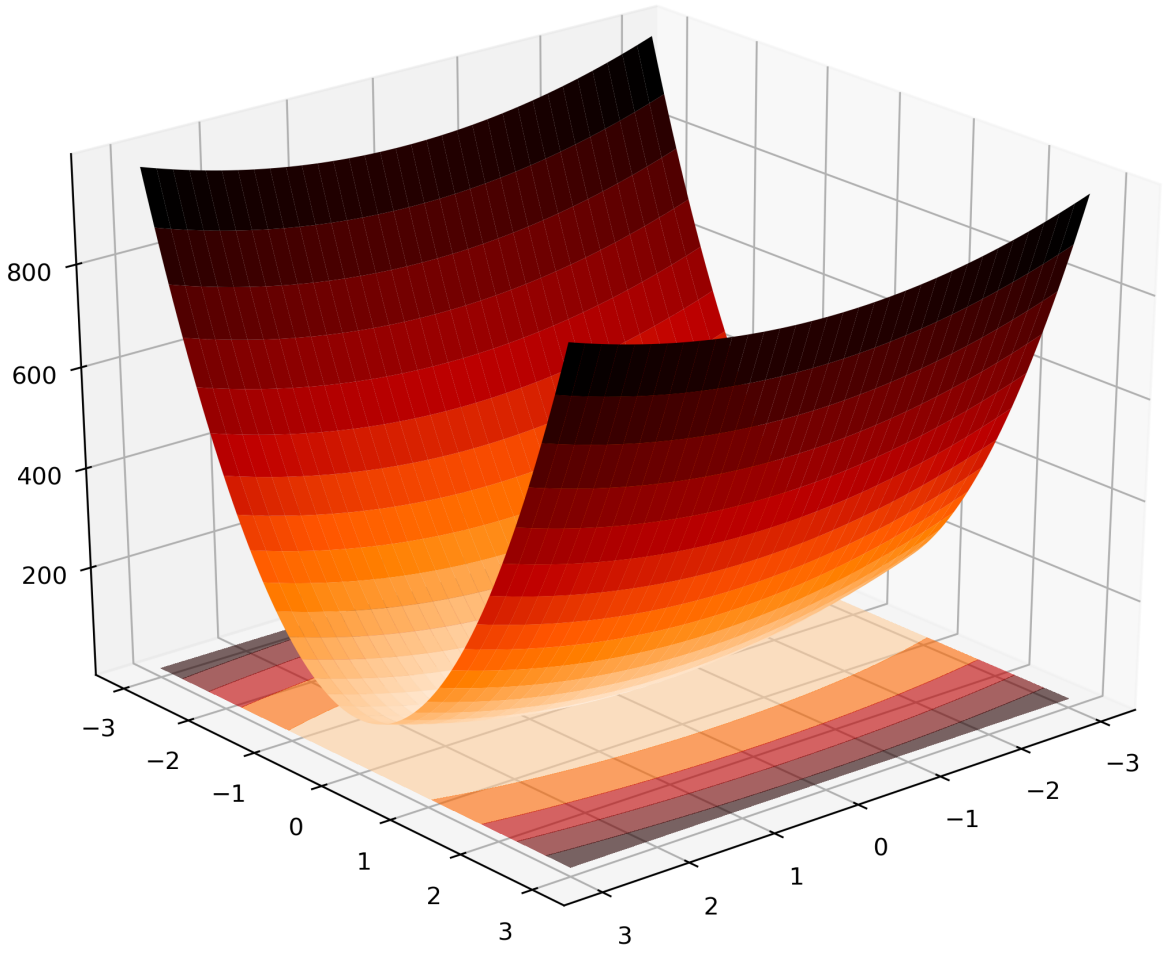
- An iteration is a successive forward and backward pass performed on a batch.
- An epoch corresponds to the process of all the training data.
  - In SGD, one epoch is completed after m iterations (where m is the number of training instances).
  - In gradient descent, an epoch is completed every iteration.
  - In mini-batch gradient descent, an epoch is completed every $\frac{m}{B}$ iterations.

A mini-batch gradient descent implementation is

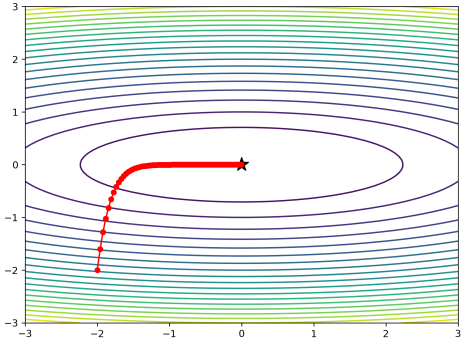
```
for e in range(1,nb_epochs+1):

  for b in range(0, X_train.size(0), batch_size):
    output = model(X_train[b:b+batch_size])
    loss = criterion(output, y_train[b:b+batch_size])

    model.zero_grad()
    loss.backward()
    with torch.no_grad():
      for p in model.parameters(): p -= alpha * p.grad
```

Let us illustrate the gradient descent algorithm on a simple function:

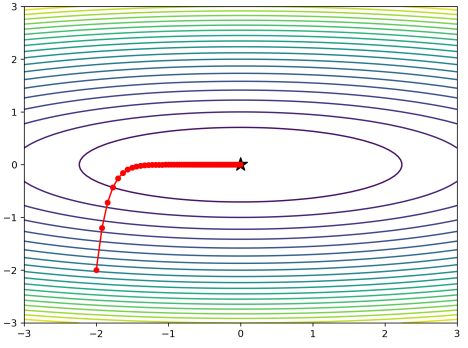
$$f(x, y) = ax^2 + by^2, \text{ where } a = 10 \text{ and } b = 100$$



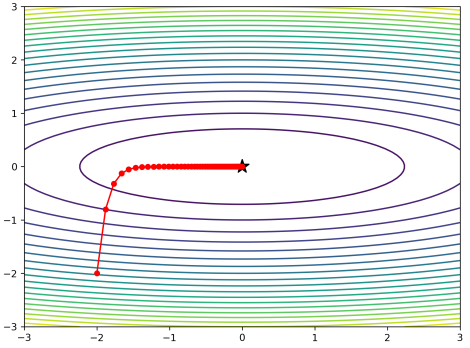
$\alpha = 0.001$



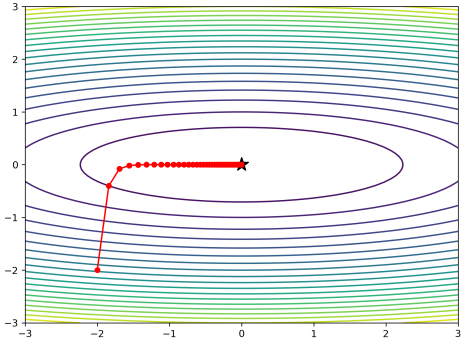
$\alpha = 0.002$



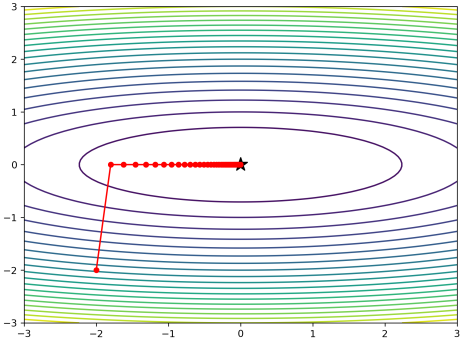
$\alpha = 0.003$



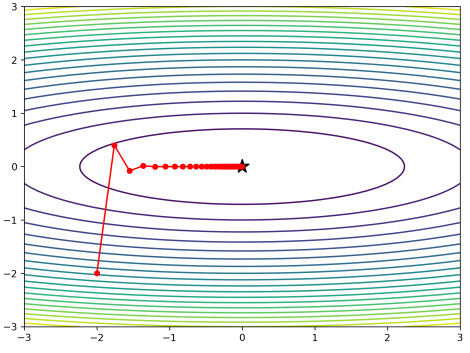
$\alpha = 0.004$



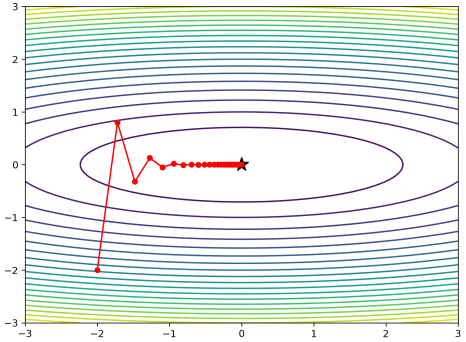
$\alpha = 0.005$



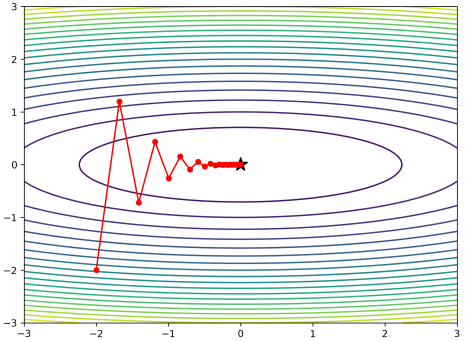
$\alpha = 0.006$



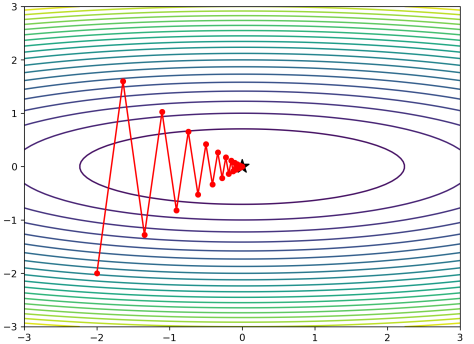
$\alpha = 0.007$



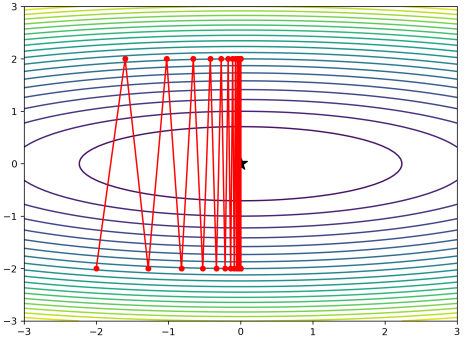
$\alpha = 0.008$



$\alpha = 0.009$



$\alpha = 0.01$



Limitations

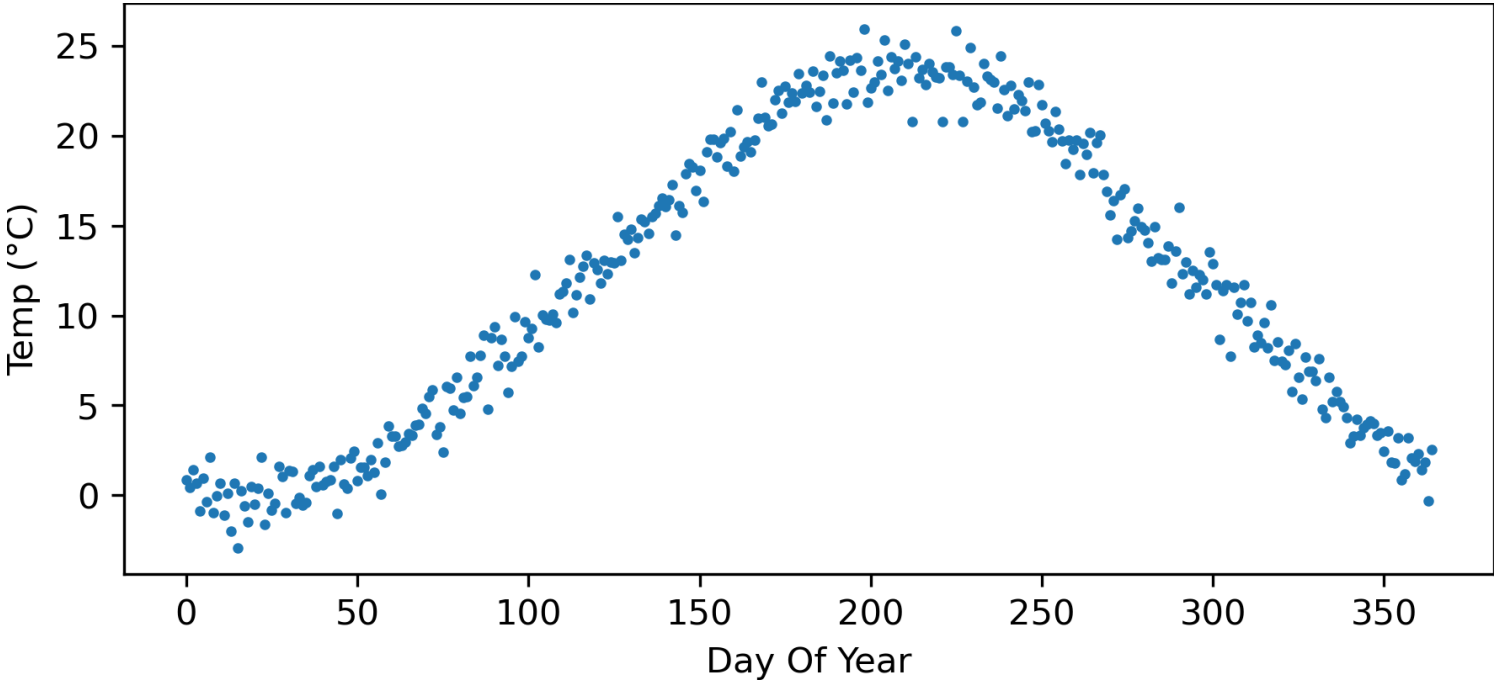
The gradient descent method makes strong assumptions

- about the magnitude of the "local curvature" to set up the step size (the learning rate), and
- about its isotropy so that the same step size makes sense in all directions (in other words for all the parameters)

- Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize
- For a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the eventual optimization is worse.
- Deep-learning generally relies on a smarter use of the gradient, using statistics over its past values to make a "smarter step" with the current one.

Exponential moving average

Let us take the example of a series of daily temperatures:

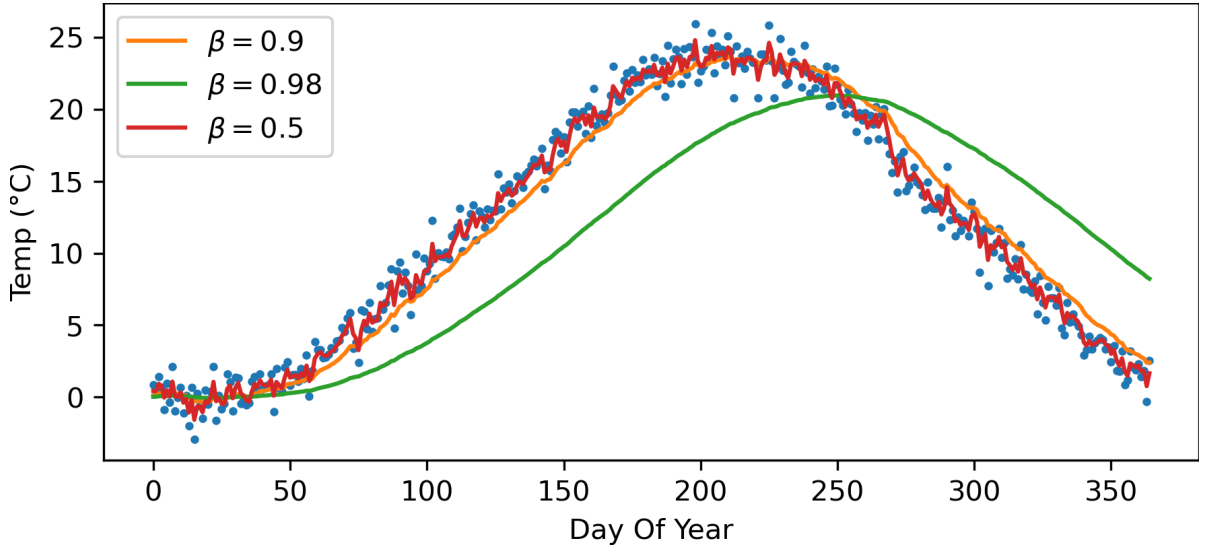


Source: New-York city average daily temperature, 2010, Wantagah cedar creek NY US station, https://www.ncdc.noaa.gov/cdo-web

The exponential moving average, also known as exponentially weighted moving average, exponentially weights the terms of a parameter. The weighting for each older term decreases exponentially:

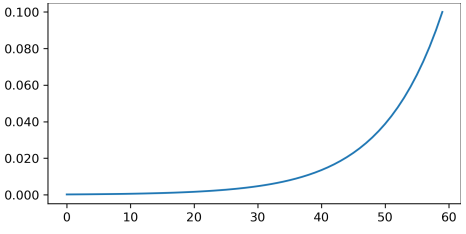
$$v_0 = 0$$
$$v_t = \beta v_{t-1} + (1 - \beta) \underbrace{\Theta_t}_{\text{parameter}}$$

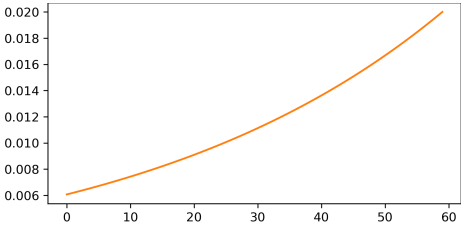
where β represents the degree of weighting decrease, a constant smoothing factor between 0 and 1.

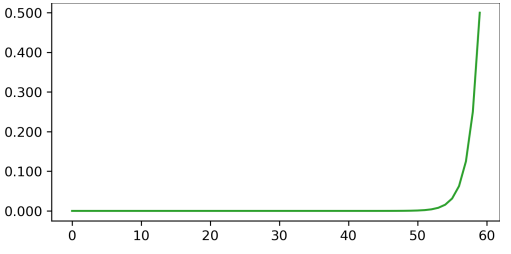


v_t is approximately an average over $\approx \frac{1}{1-\beta}$ [days]

- $\beta = 0.9 \Rightarrow \approx 10$ [days]
- $\beta = 0.98 \Rightarrow \approx 50$ [days]: it adapts more slowly, much smoother
- $\beta = 0.5 \Rightarrow \approx 2$ [days]: it is more sensitive to the outliers







By recurrence, we can show that

$$v_0 = 0$$

$$v_t = (1-\beta)\sum_{k=0}^{t}\beta^k\Theta_{t-k}$$

The intuition behind the recurrence

$$\begin{aligned}
v_0 &= 0 \\
v_1 &= \beta v_0 + (1-\beta)\Theta_1 \\
&= (1-\beta)\Theta_1 \\
v_2 &= \beta v_1 + (1-\beta)\Theta_2 \\
&= \beta(1-\beta)\Theta_1 + (1-\beta)\Theta_2 \\
v_3 &= \beta v_2 + (1-\beta)\Theta_3 \\
&= \beta^2(1-\beta)\Theta_1 + \beta(1-\beta)\Theta_2 + (1-\beta)\Theta_3 \\
&\text{etc.}
\end{aligned}$$

It is possible to correct the bias by dividing each term of the exponentially weighted average v_t by the factor $\frac{1}{1-\beta^t}$.

This bias correction is efficient for small t values, but it does not make too much difference for high t values.

Momentum

The "vanilla" mini-batch stochastic gradient descent (SGD) can be expressed as

$$w_{t+1} = w_t - \alpha g_t$$

where

$$g_t = \sum_{\mathcal{B}_i} \nabla \ell_{x \in \mathcal{B}_i}$$

is the gradient summed over a mini-batch.

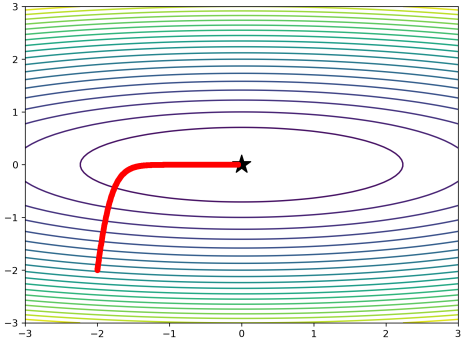
The momentum adds inertia in the choice of the step direction by computing an exponentially weighted moving average.

$$
\begin{aligned}
u_0 &= 0 \\
u_t &= \beta \underbrace{u_{t-1}}_{\text{velocity}} + (1-\beta) \underbrace{g_t}_{\text{gradient}} \\
w_{t+1} &= w_t - \alpha u_t
\end{aligned}
$$

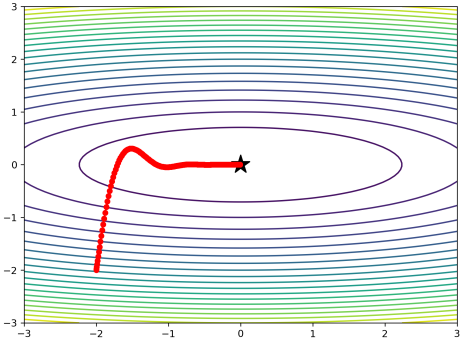
If

- $\beta = 0$, this is the same as vanilla SGD.
- $\beta > 0$,
  - it accelerates the training if the gradient does not change much
  - it dampens oscillations in narrow valleys
- usually $\beta = 0.9$: an average over about the last 10 gradients is computed

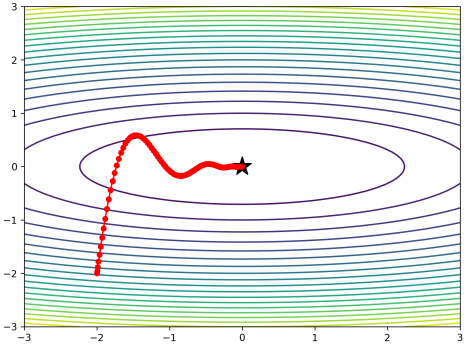
$\alpha = 0.0001$



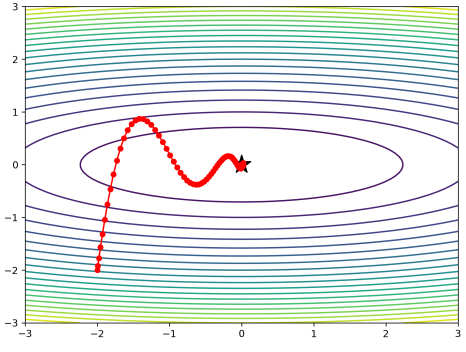
$\alpha = 0.0005$



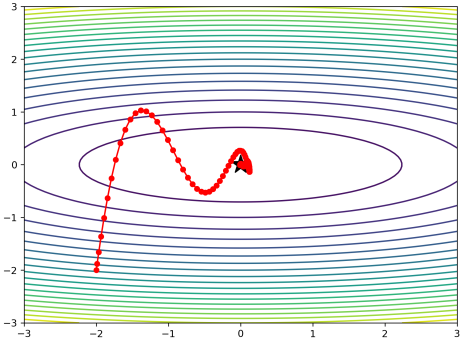
$\alpha = 0.001$



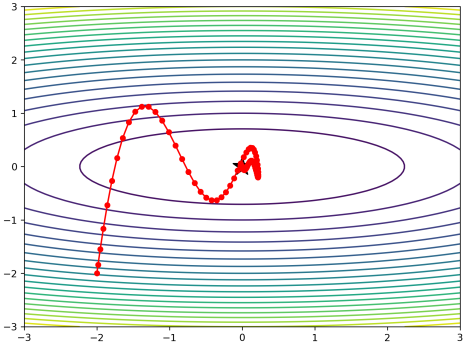
$\alpha = 0.002$



$\alpha = 0.003$



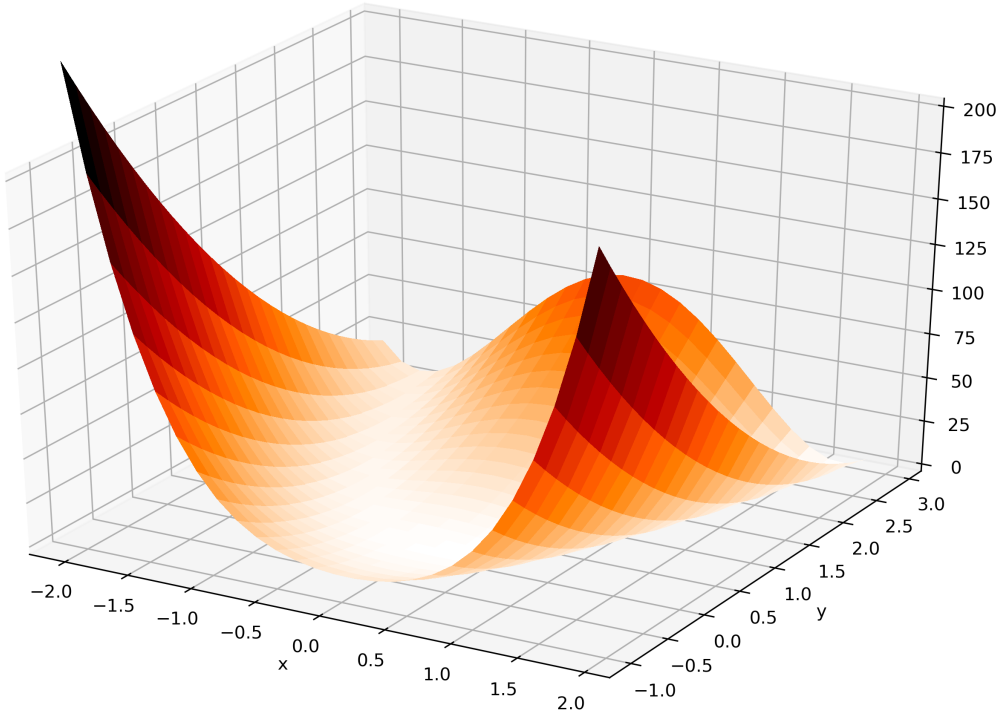
$\alpha = 0.004$



Let us take another example, the Rosenbrock function:

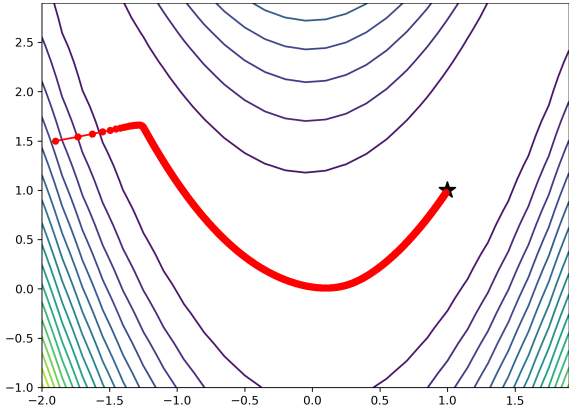
$$f(x, y) = (x - 1)^2 + b(y - x^2)^2, \text{ where } b = 10$$

$$\frac{\partial f}{\partial x} = 2(x - 1) - 4b(y - x^2)x \qquad \frac{\partial f}{\partial y} = 2b(y - x^2)$$

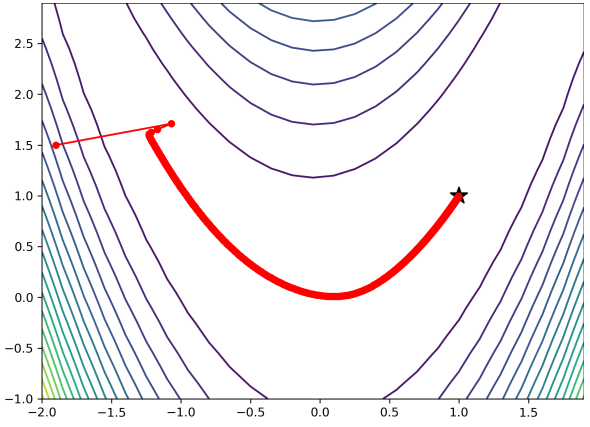


Gradien Descent

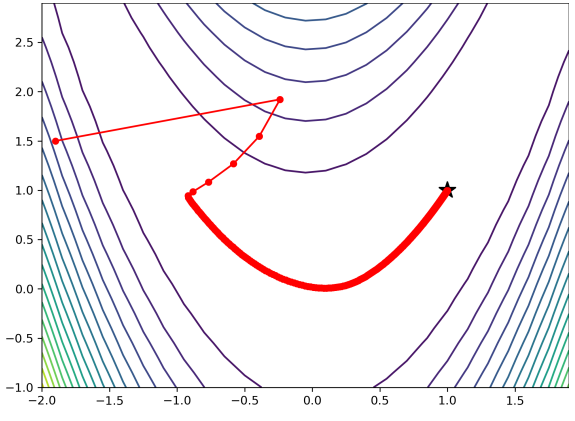
$\alpha = 0.001$



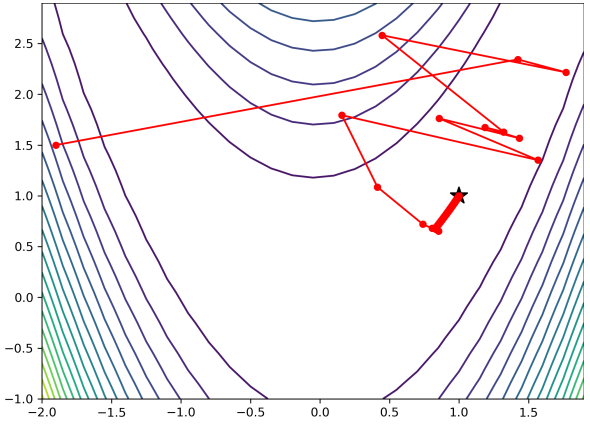
$\alpha = 0.005$



$\alpha = 0.01$

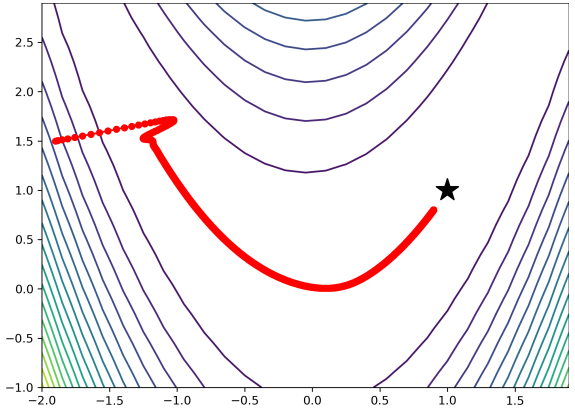


$\alpha = 0.02$

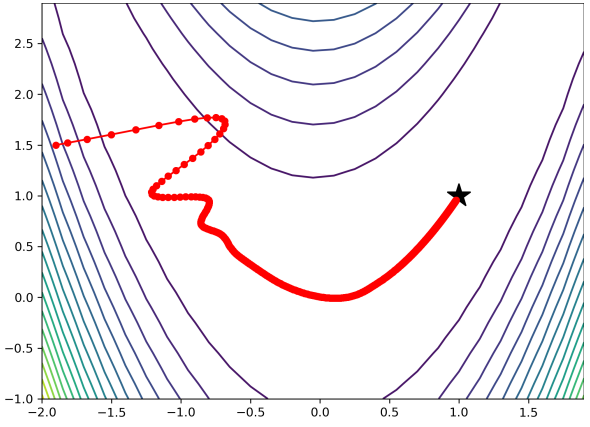


Momentum

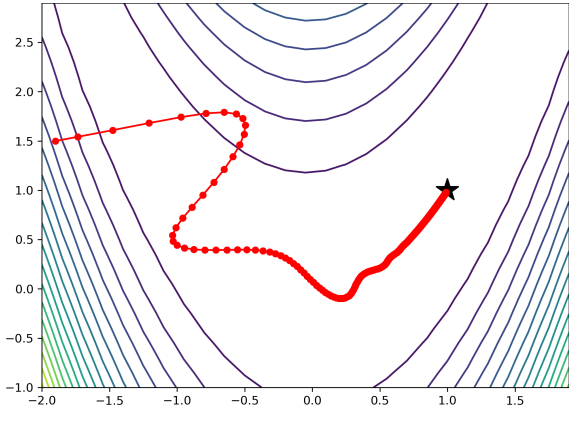
$\alpha = 0.001$



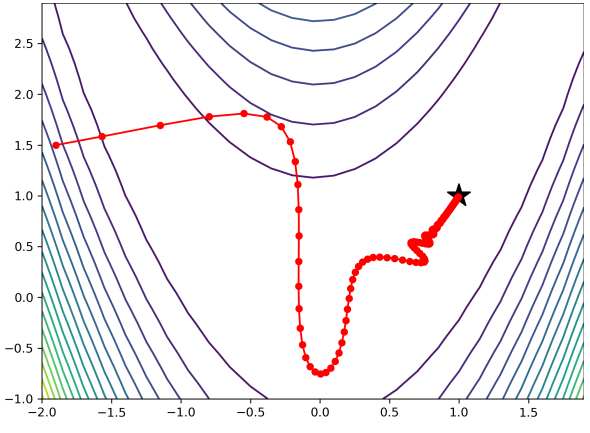
$\alpha = 0.005$



$\alpha = 0.01$



$\alpha = 0.02$



RMSprop

The Root Mean Square propagation (RMSprop) is similar to momentum. It dampens out the large sums in directions in which there are big oscillations. It speeds-up the gradient descent, and thus faster learning.

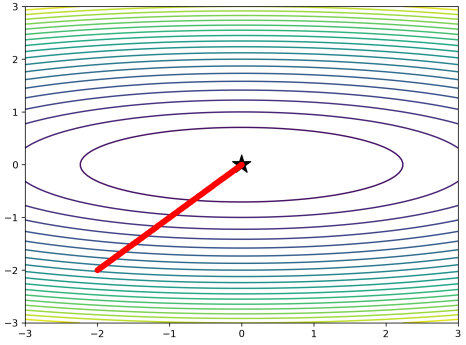
It applies the weighted moving average on squared derivatives:

$$
\begin{aligned}
v_0 &= 0 \\
v_t &= \beta v_{t-1} + (1-\beta) g_t^2 \\
w_{t+1} &= w_t - \alpha \frac{g_t}{\sqrt{v_t} + \epsilon}
\end{aligned}
$$

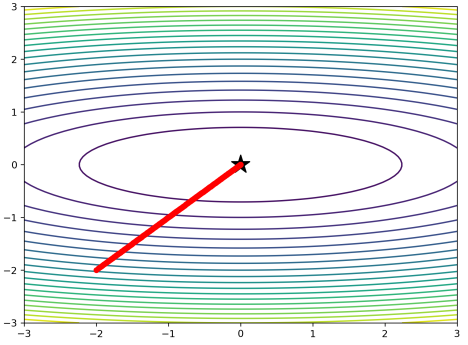
where

- $\beta = 0.99$
- ϵ ensures that we do not end up dividing by zero. It is generally set up to $1e^{-8}$.

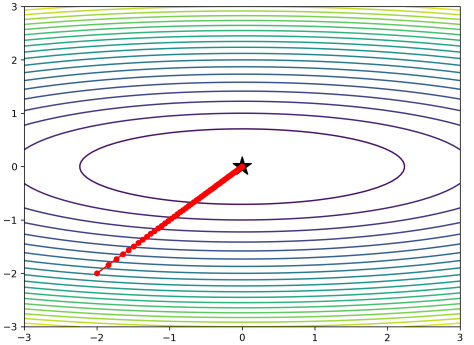
$\alpha = 0.005$



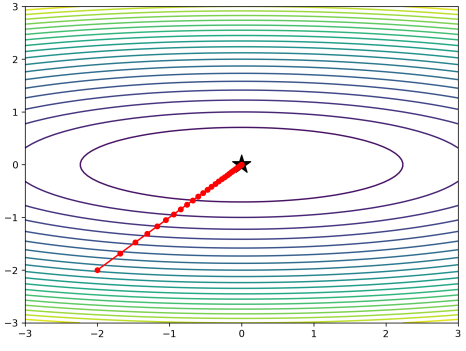
$\alpha = 0.0005$



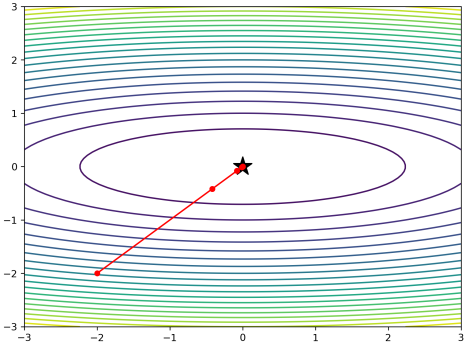
$\alpha = 0.05$



$\alpha = 0.01$



$\alpha = 0.05$



Adam

Adaptive Moment Estimation (Adam) is probably one of the main optimizer: it also computes adaptive learning rates for each parameter. It combines both momentum and RMSprop principles:

$$u_0 = 0$$
$$u_t = \beta_1 u_{t-1} + (1 - \beta_1) g_t$$
$$u_t = \frac{u_t}{1 - \beta_1^t}$$

$$v_0 = 0$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$v_t = \frac{v_t}{1 - \beta_2^t}$$

$$w_{t+1} = w_t - \alpha \frac{u_t}{\sqrt{v_t} + \epsilon}$$

where, in the original paper,

- $\beta_1 = 0.9$
- $\beta_2 = 0.99$
- $\epsilon = 1e^{-8}$

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Other variants

And many more

- Neterov's accelerated gradient
- Adagrad
- Adadelta
- Adamax
- Nadam

In Pytorch

Various optimizers are implemented in the `torch.optim` package.

1. To use `torch.optim`, you have to construct an optimizer object that will hold the current state and will update the parameters based on the computed gradients.

2. You need to give the `Optimizer` an iterable containing the parameters to optimise. And then, if required, to specify optimizer-specific options such as the learning rate, β values, ...

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

3. All optimizers implement a `step()` method, that updates the parameters. This function can be called once the gradients are computed using the `backward()` method.

```
for input, target in dataset:
  optimizer.zero_grad()
  output = model(input)
  loss = criterion(output, target)
  loss.backward()
  optimizer.step()
```

https://pytorch.org/docs/stable/optim.html

Learning rate decay

Instead of having a learning rate α constant through the iteration, a learning rate decay scheme decreases its value through the iterations.
The objective is to take smaller and smallers steps as we approach the minimum:

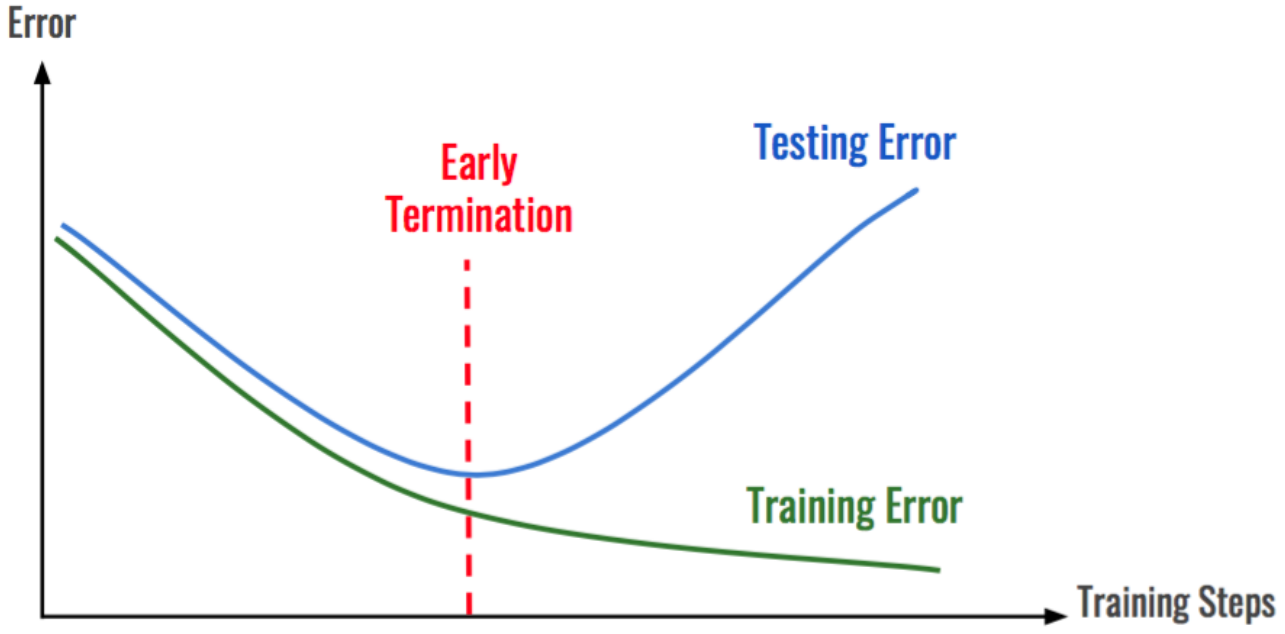
$$\alpha_t = \frac{\alpha_0}{1 + \gamma t}$$

where γ is the decay rate and t the epoch number (not the iteration here).

Regularization

Regularization

While the network is learning, both training and testing errors decreasing at the same rate. However, at some point, the testing error starts to increase while the training loss continues to decrease.
This often occurs when overfitting starts to happen.



Source: https://mc.ai/early-stopping-with-pytorch-to-restrain-your-model-from-overfitting/

Note that there is a correlation between overfitting and the capacity of the network.

The capacity refers to the ability of the network to learn complex functions. It usually increases with the number of units and layers, and thus the number of trainable parameters.

Note that there is a correlation between overfitting and the capacity of the network.

The capacity refers to the ability of the network to learn complex functions. It usually increases with the number of units and layers, and thus the number of trainable parameters.

What to do if

- the network is not learning on the training data, i.e. the loss is not decreasing at all?
  - It is an underfitting situation usually due to a too small network capacity.
  - You can try increasing the number of units and/or the number of hidden layers. Do not go crazy with the numbers, make them as small as possible.
- the network is overfitting, that is the testing error is increasing while the training loss is decreasing?
  - The network has too much capacity.
  - You can can look at either reducing the model capacity by lowering the number units and layers or applying regularization techniques.

λ regularization

λ regularization

Goal: add some information to avoid an overfitting situation.

λ regularization penalizes models that are too complex. For example, it applies a $\mathcal{L}_2$-regularisation.

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} loss(\hat{y}_i, y_i) + \frac{\lambda}{2m} \sum_{l=1}^{L} ||w^{(l)}||_F^2$$

where λ is the regularization hyperparameter to be defined by the user. It controls the importance of the regularization term.

$||\cdot||_2$ is known as the Frobenius norm:

$$||w^{(l)}||_F^2 = \sum_{i=1}^{K^{(l-1)}} \sum_{j=1}^{K^{(l)}} \left(w_{ij}^{(l)}\right)^2$$

where $K^{(l)}$ is the number of neurons in layer l.

It is also possible to apply a $\mathcal{L}_1$-regularisation.

$$||w^{(l)}||_1 = \sum_{i=1}^{K^{(l-1)}} \sum_{j=1}^{K^{(l)}} \left|w_{ij}^{(l)}\right|$$

In neural network, λ regularization is known as the weight decay.

$$
\begin{aligned}
w_{t+1}^{(l)} &= w_t^{(l)} - \alpha\left(g_t + \frac{\lambda}{m} w_t^{(l)}\right) \\
&= w_t^{(l)} - \alpha\frac{\lambda}{m} w_t^{(l)} - \alpha g_t \\
&= \underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{\text{weight decay}} w_t^{(l)} - \alpha g_t
\end{aligned}
$$

where g_t is the gradient computed through backpropagation.

Why regularization reduces overfitting?

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} loss(\hat{y}_i, y_i) + \frac{\lambda}{2m} \sum_{l=1}^{L} ||w^{(l)}||_F^2$$

If λ is very big, then $w^{(l)} \approx 0$

- $\Rightarrow$ most of the neurons are useless
- $\Rightarrow$ the network is simplified

Implementation

There are two ways of implementing a regularization in PyTorch.

1. Set `weight_decay` parameter to a non-zero value in a standard optimizer, e.g.

```
torch.optim.SGD(network.parameters(), lr = learning_rate, weight_decay=lamda)
```

where `lamda` is the λ value for a $\mathcal{L}_2$ regularization.
In this case, the optimizer handle the regularization.

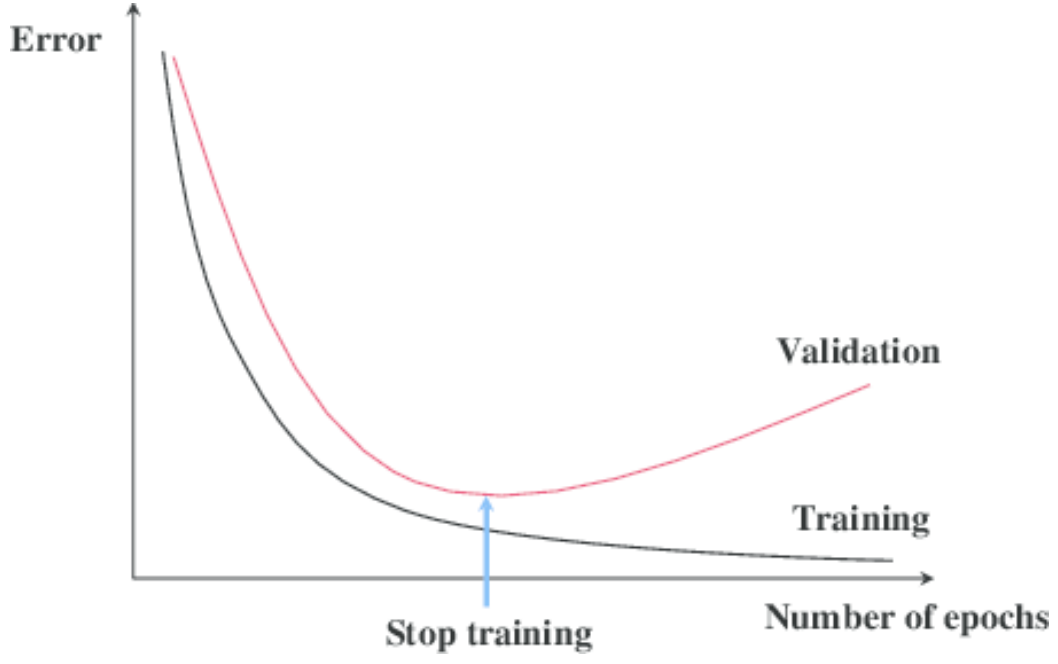
2. Implement your own custom-loss function

```
regularization_loss = 0

for param in model.parameters():
    regularization_loss += torch.sum(abs(param))

classif_loss = criterion(pred,target)
loss = classif_loss + lamda * regularization_loss

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Early stopping

Early stopping

Early stopping is a simple technique that consists in stopping the training when the model performance reaches an optimum on an independent validation set.

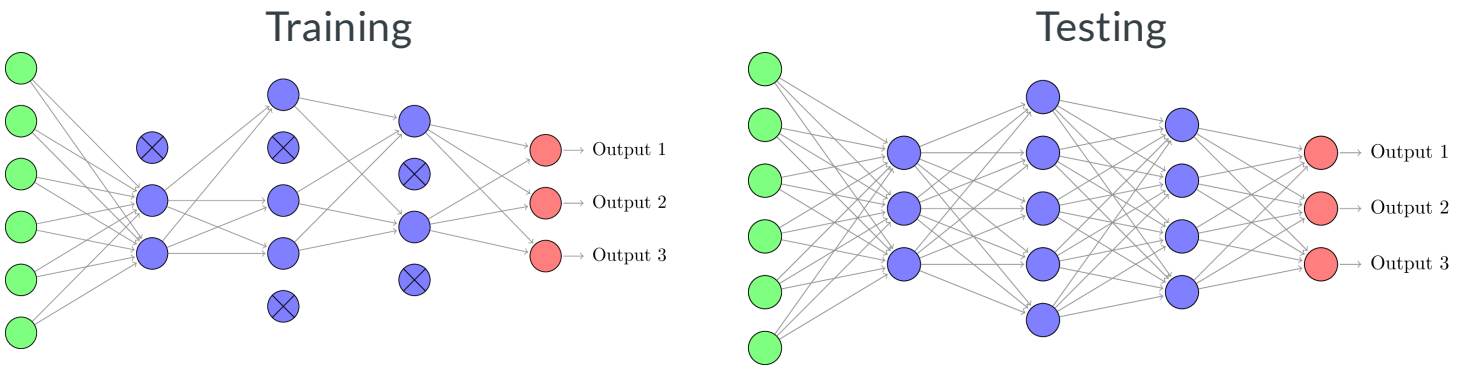


Source: http://fouryears.eu/2017/12/06/the-mystery-of-early-stopping/comment-page-1/

Dropout

Dropout

Dropout consists of removing units at random during the forward pass on each sample, and putting them all back during test.



Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958.

- The probability of keeping a neuron, called the keep probability $1-p$, is an hyperparameter.
- The keep probability can be same for the whole network or different for each layer.
- Concretely, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.
  Let $x^{(l)}$ be a neuron activation, and $m^{(l)}$ be an independent Boolean random variable of probability $1-p$. We have

$$\begin{aligned}\mathbb{E}\left(m^{(l)}x^{(l)}\right) &= \mathbb{E}\left(m^{(l)}\right)\mathbb{E}\left(x^{(l)}\right)\\ &= (1-p)\mathbb{E}\left(x^{(l)}\right)\end{aligned}$$

The most common way to implement dropout is through a technique named inverted dropout. It consists in keeping the expected value of the activation $x^{(l)}$ unchanged by scaling-up it during the training.

```
>>> p = 0.6
>>> x = 10*torch.rand((3, 5))
>>> x
tensor([[9.9548, 9.4780, 7.0771, 6.7888, 8.8192],
        [0.5637, 9.4359, 4.3755, 7.1627, 9.6331],
        [8.1057, 2.7380, 2.0878, 1.8551, 2.5148]])
>>> m = torch.randn(x.shape) < 1-p
tensor([[ True,  True, False, False, False],
        [ True, False,  True,  True, False],
        [ True, False,  True, False,  True]])
>>> x *= m
>>> x /= (1-p)
>>> x
tensor([[24.8870, 23.6950,  0.0000,  0.0000,  0.0000],
        [ 1.4093,  0.0000, 10.9387, 17.9067,  0.0000],
        [20.2642,  0.0000,  5.2195,  0.0000,  6.2870]])
```

Note that at predicting time all the neurons contribute to the decision, none of them are turned off.

Why dropout works?

- At each iteration, a slightly different network is trained.
- As a neuron might be shut down at anytime, the network becomes less sensitive to the activation of specific neurons.
- Each neuron has to spread out its weights, resulting in a behavior similar to that of the $\mathcal{L}_2$ regularization, which shrinks weights

Implementation

Dropout is implemented in PyTorch as `nn.Dropout`, which is a `torch.Module`.

- As explained previously, in the forward pass, it samples a Boolean variable with respect to the keep probability for each element of the tensor it gets as input, and zeroes entries accordingly.
- The default probability to drop is $p = 0.5$, but other values can be specified.

```
>>> x = torch.full((3, 5), 1.0).requires_grad_()
>>> x
tensor([[ 1.,  1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.,  1.]])
>>> dropout = nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y
tensor([[ 0.,  0.,  4.,  0.,  4.],
        [ 0.,  4.,  4.,  4.,  0.],
        [ 0.,  0.,  4.,  0.,  0.]])
```

If we have the following (MultiLayer Perceptron) network:

```
model = nn.Sequential(
          nn.Linear(10, 100),
          nn.ReLU(),
          nn.Linear(100, 50),
          nn.ReLU(),
          nn.Linear(50, 2));
```

If we have the following (MultiLayer Perceptron) network:

```
model = nn.Sequential(
          nn.Linear(10, 100),
          nn.ReLU(),
          nn.Linear(100, 50),
          nn.ReLU(),
          nn.Linear(50, 2));
```

Dropout layers can simply be added:

```
model = nn.Sequential(
          nn.Linear(10, 100),
          nn.ReLU(),
          nn.Dropout(),
          nn.Linear(100, 50),
          nn.ReLU(),
          nn.Dropout(),
          nn.Linear(50, 2));
```

As a network that comprises dropout acts differently in training and testing steps, it is crucial to set the model in "train" or "test" mode.

- train mode (default mode) $\Rightarrow$ `model.train(True)` or **`model.train()`**
- test mode $\Rightarrow$ `model.train(False)` or **`model.eval()`**

```
>>> model = nn.Sequential(
               nn.Linear(3, 10),
               nn.Dropout(),
               nn.Linear(10, 3))
>>> model.training
True
>>> model.eval()
Sequential (
     (0): Linear (3 -> 10)
     (1): Dropout (p = 0.5)
     (2): Linear (10 -> 3))
>>> model.training
False
```

Batch normalization

Batch normalization

- We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.
- It was the main motivation behind Xavier's and He's weight initialization rules.
- A complementary approach consists in explicitely controlling activation statistics during the forward pass by renormalizing them.
- Batch normalization, proposed by Ioffe and Szegedy (2015), was the first method introducing this idea

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

"Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization [...]"

(Ioffe and Szegedy, 2015)

- Batch normalization can be done anywhere in a network. It helps to control the inputs first (mean) and second (variance) order moments, so that the following layers do not need to adapt to their drift.
- Concretely, batch normalization
  - during training, shifts and rescales inputs according to the mean and variance estimated on the batch.
  - during testing, shifts and rescales inputs according to the empirical moments estimated during training.

Training

Let $z_b \in \mathbb{R}^d, b = 1, \cdots, B$ be B instances in a batch $\mathcal{B}$.

First, the empirical mean $\mu_\mathcal{B} \in \mathbb{R}^d$ and variance $\sigma^2_\mathcal{B} \in \mathbb{R}^d$ is computed per-component (for each feature) on the batch:

$$\mu_\mathcal{B} = \frac{1}{B} \sum_{b=1}^{B} z_b$$

$$\sigma^2_\mathcal{B} = \frac{1}{B} \sum_{b=1}^{B} (z_b - \mu_\mathcal{B})^2$$

Then the instances are normalized

$$z_{b_{\text{norm}}} = \frac{z_b - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}}, \forall b = 1, \cdots, B$$

And the batch normalization layer outputs

$$\tilde{z}_b = \gamma z_{b_{\text{norm}}} + \beta$$

where γ and β are learnable parameters.

In practice, batch normalization is applied on mini-batches during training as follows:

$$
\begin{aligned}
z_b^{(l)} &= w^{(l)} x_b^{(l-1)} + b^{(l)} \\
z_{b_{\text{norm}}}^{(l)} &= \frac{z_b - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
\tilde{z_b}^{(l)} &= \gamma z_{b_{\text{norm}}}^{(l)} + \beta \\
a^{(l)} &= g\left(\tilde{z_b}^{(l)}\right)
\end{aligned}
$$

In practice, batch normalization is applied on mini-batches during training as follows:

$$
\begin{aligned}
z_b^{(l)} &= w^{(l)} x_b^{(l-1)} + b^{(l)} \\
z_{b_{\text{norm}}}^{(l)} &= \frac{z_b - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
\tilde{z_b}^{(l)} &= \gamma z_{b_{\text{norm}}}^{(l)} + \beta \\
a^{(l)} &= g\left(\tilde{z_b}^{(l)}\right)
\end{aligned}
$$

There exists discussion to know if it better to batch-normalize $z_b^{(l)}$ or $a_b^{(l)}$.

In practice, batch normalization is applied on mini-batches during training as follows:

$$
\begin{aligned}
z_b^{(l)} &= w^{(l)} x_b^{(l-1)} + b^{(l)} \\
z_{b_{\text{norm}}}^{(l)} &= \frac{z_b - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\
\tilde{z_b}^{(l)} &= \gamma z_{b_{\text{norm}}}^{(l)} + \beta \\
a^{(l)} &= g\left(\tilde{z_b}^{(l)}\right)
\end{aligned}
$$

There exists discussion to know if it better to batch-normalize $z_b^{(l)}$ or $a_b^{(l)}$.

Testing

During the inference step, μ and σ^2 are estimated during the training using an exponentially moving average:

$$
z_{\text{norm}}^{(l)} = \frac{zb^{(l)} - \mu}{\sqrt{\sigma^2 + \epsilon}}
$$

As dropout, batch normalization is implemented in a separate module that processes input components independently:

```
>>> bn = nn.BatchNorm1d(3)
>>> with torch.no_grad():
...     bn.bias.copy_(torch.tensor([2., 4., 8.]))
...     bn.weight.copy_(torch.tensor([1., 2., 3.]))
...
Parameter containing:
tensor([2., 4., 8.], requires_grad=True)
Parameter containing:
tensor([1., 2., 3.], requires_grad=True)
>>> x = torch.empty(1000, 3).normal_()
>>> x = x * torch.tensor([2., 5., 10.]) + torch.tensor([-10., 25., 3.])
>>> x.mean(0)
tensor([-9.9669, 25.0213,  2.4361])
>>> x.std(0)
tensor([1.9063, 5.0764, 9.7474])
>>> y = bn(x)
>>> y.mean(0)
tensor([2.0000, 4.0000, 8.0000], grad_fn=<MeanBackward2>)
>>> y.std(0)
tensor([1.0005, 2.0010, 3.0015], grad_fn=<StdBackward1>)
```

Layer normalization

Another normalization in the same spirit is the layer normalization.

Given an input $x \in \mathbb{R}^d$, it normalizes the elements of x, hence normalizing inputs accross the layer instead of doing it accross the batch:

$$\mu = \frac{1}{d}\sum_{k=1}^{d} x_k$$

$$\sigma^2 = \frac{1}{d}\sum_{k=1}^{d}(x_k - \mu)$$

$$x_{d_{\text{norm}}} = \frac{x_d - \mu}{\sigma}$$

Although it gives slightly worst improvements than batch normalization, it has the advantage of behaving similarly in train and test, and processing samples individually.

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. arXiv preprint arXiv:1607.06450.

References

References

- François Fleuret's lectures: https://fleuret.org/ee559/