

Machine Learning for Robotics: **Regularization Methods**

Prof. Navid Dadkhah Tehrani



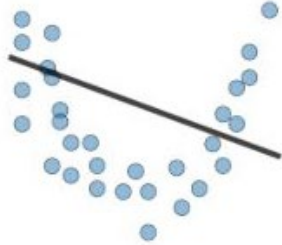

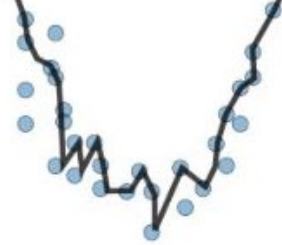
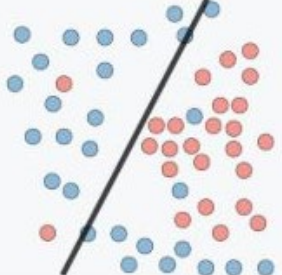
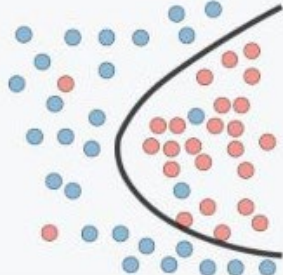
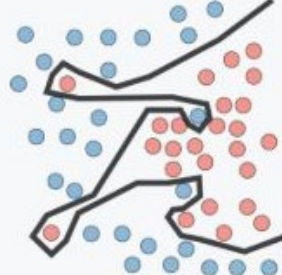

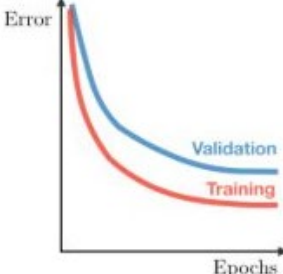

Overfitting and Underfitting

Underfitting:

The model is not complex enough to capture the data trend. In these cases we need to add more layers or more neurons to the network. Or change the network architecture altogether (for example use CNN or transformers instead of MLP).

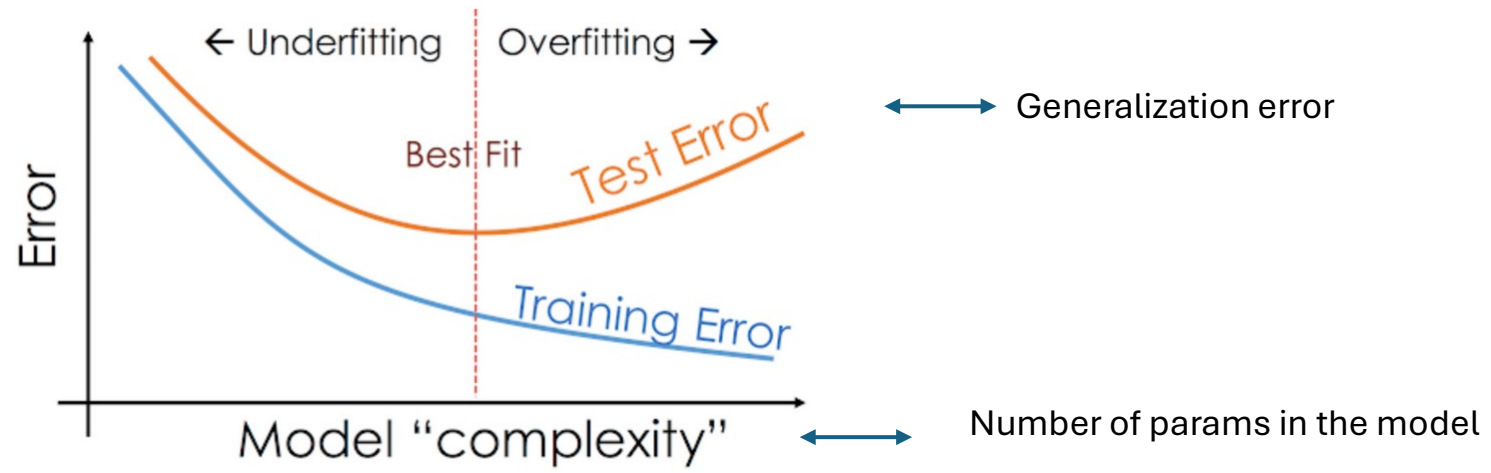
Overfitting:

When the model memorizes details in the training set that are only specific to training set and don't generalize to test set (as opposed to learning them).

	Underfitting	Just right	Overfitting
Symptoms	<ul style="list-style-type: none"> • High training error • Training error close to test error • High bias 	<ul style="list-style-type: none"> • Training error slightly lower than test error 	<ul style="list-style-type: none"> • Very low training error • Training error much lower than test error • High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	<ul style="list-style-type: none"> • Complexify model • Add more features • Train longer 		<ul style="list-style-type: none"> • Perform regularization • Get more data

- Plotting training and validation accuracy in each epoch is very important

- We can use Tensorboard in Pytorch to do that.



Hyperparameters versus Parameters:

Parameters:

- Weight and biases.

Hyperparameters:

- Mini-batch size
- Number of training epochs.
- Number of hidden layers.
- Learning rate.
- Loss function type (MSE, CE, ...).
- Activation function type (Relu, Sigmoid, tanh, ...).
- Optimization algorithm (different variation gradient descend).
- Regularization scheme.
- ...

The validation set is used more than once to get the proper hyperparameters. But the test set is used only once.

Regularization:

The process of preventing overfitting.

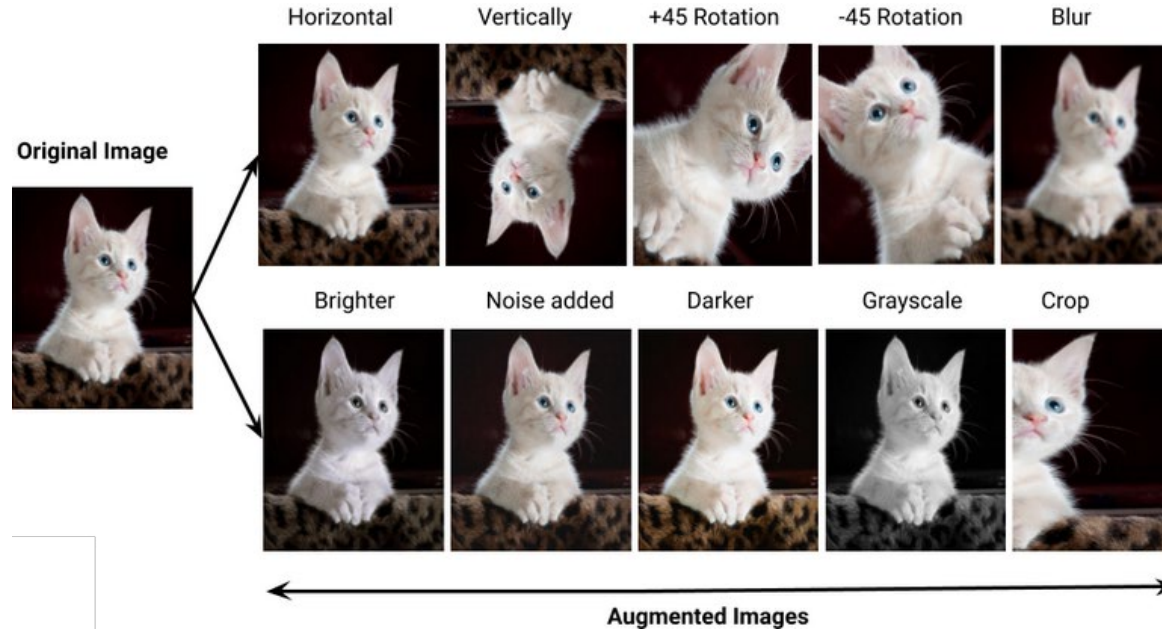
Note that the purpose of regularization is not reducing the training error but the overfitting error.

The training error can be reduced by playing with hyperparameters and the model architecture.

Throughout the rest of this lecture, we talk about different and independent ways of applying regularization.

These techniques can be applied to all kind of NN architectures and not just MLPs.

Data Augmentation



- A cheap way to increase the size of dataset and improve generalization and preventing the model to memorize the data.
- Usually test accuracy goes up with more data and also the gap between training error and test error reduces with more data.
- In pytorch, “torchvision.transforms” is used to add various data augmentation to images.

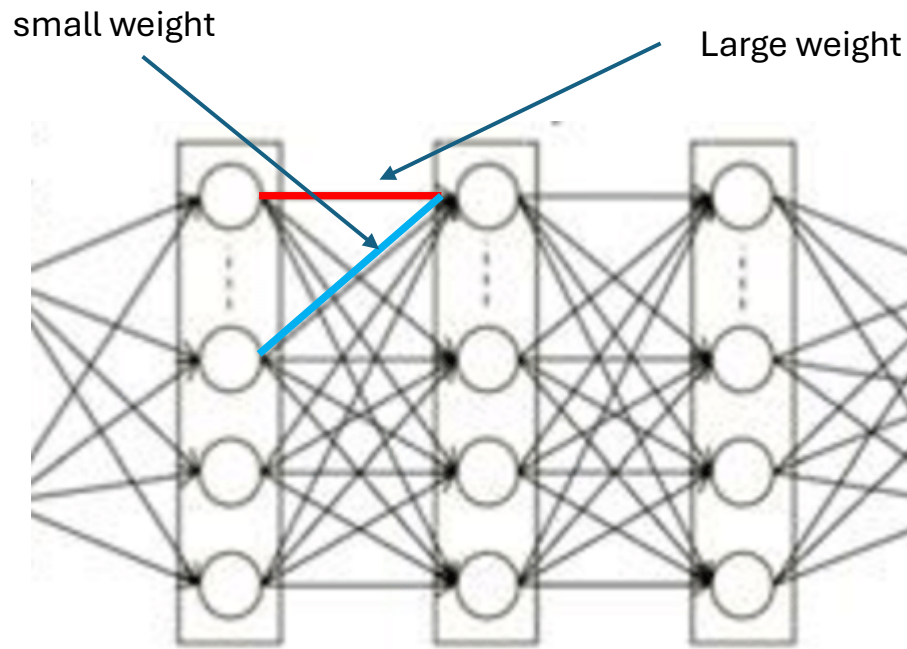
Early Stopping:

Note that training longer (more epochs) doesn't make the performance necessarily better.

Once you notice the gap between training error and test/validation error start to increase, it's a good time to stop.

L_2 Regularization

- We add norm of the parameters as a penalty to the loss function.
- It's a common practice in optimization.
- This is essentially penalty against large weights. The rational is that if there are large weights in the NN it might outweigh other weights.
- Note that only weights are being regularized not biases. So the NN can still produce large outputs if needed.



$$L_2 \text{ regularized loss} = \text{loss} + \lambda \sum_j w_j^2$$

$$L_1 \text{ regularized loss} = \text{loss} + \lambda \sum_j |w_j|$$

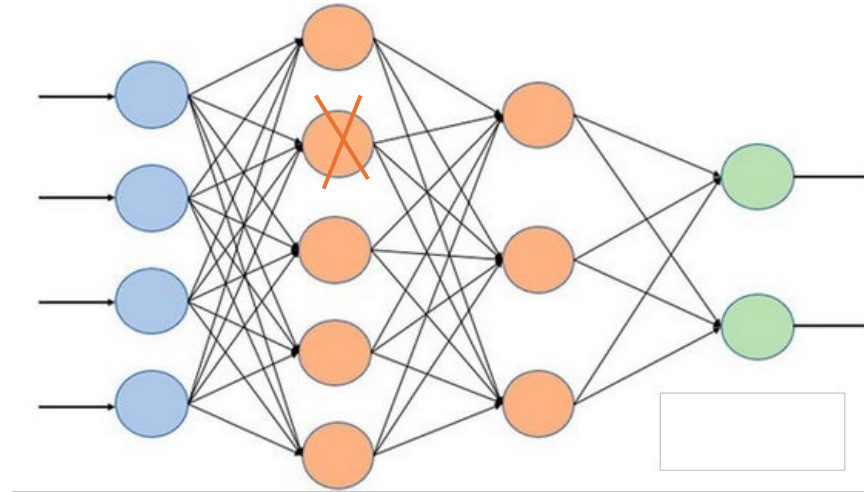
Weight regularization in Pytorch

```
bias_params = [p for name, p in self.named_parameters() if 'bias' in name]
others = [p for name, p in self.named_parameters() if 'bias' not in name]

optim.SGD([
    {'params': others},
    {'params': bias_params, 'weight_decay': 0}
], weight_decay=1e-2, lr=1e-2)
```

Dropout

The idea in dropout is to randomly dropping nodes in the NN with some probability during training.



10 percent dropout means that during training in each forward pass, with 10 percent probability we remove a neuron. i.e. In each forward pass a different neuron is deleted.

In other words, in each forward pass the model is slightly different.

This prevents the model to rely always on certain nodes and help all the nodes get involved equally in the training.

As a results, the weight values will be more spread out which leads to smaller weights -> L_2 regularization.

Note that the dropout is only applied to the hidden layers.

After training and during evaluation, the dropout needs to be removed.

Dropout in Pytorch

Note that pytorch do not really delete a node for dropout implementation. It just multiples the activation of the to be deleted nodes by zero that is sampled from a Bernoulli distribution.

```
self.my_network = torch.nn.Sequential(  
    # 1st hidden layer  
    torch.nn.Flatten(),  
    torch.nn.Linear(num_features, num_hidden_1),  
    torch.nn.ReLU(),  
    torch.nn.Dropout(drop_proba),  
    # 2nd hidden layer  
    torch.nn.Linear(num_hidden_1, num_hidden_2),  
    torch.nn.ReLU(),  
    torch.nn.Dropout(drop_proba),  
    # output layer  
    torch.nn.Linear(num_hidden_2, num_classes)  
)
```

Note:

Make sure you use `model.train()` before training and `model.eval()` during evaluation.
This makes Pytorch skip dropout during evaluation.

Tip:

Do not add dropout if the model does not overfit.

Recommendation:

Increase the capacity of the model (add more layer/neurons) until it overfit, then use dropout to mitigate overfitting.

Q: why don't we use dropout for weights of the NN?

A: this concept is called drop connect. And has been tried in the literature with no success!

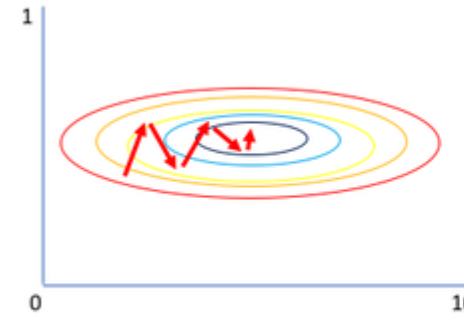
Input Normalization

Normalizing input to the NN improves gradient descent (common practice in optimization)

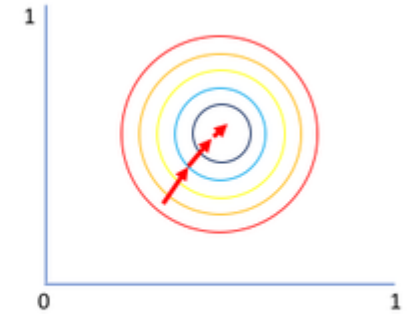
For example, if the input features are age and income:

Age of house $\in [0, 100]$

Household Income $\in [10000, 500000]$



Gradient of larger parameter dominates the update



Both parameters can be updated in equal proportions

In practice, we scale the features to have zero mean and standard deviation of one.

$$x_j = \frac{x_j - \mu_j}{\sigma_j}$$

Where μ is the mean and σ is the standard deviation.

$$\mu = \frac{1}{n} \sum x_i, \quad \sigma = \frac{1}{n} (x_i - y)^2$$

After the training, during the evaluation, we have to apply the same normalization on the new data.

When delivering the NN to the customer, you must deliver both the NN and the normalization scheme that was used.

Batch Normalization

Normalizing the input only affects the 1st layer of the NN. Now we are looking at normalization for hidden layers.

The important observation is that the hidden layer activations are the same thing as input layers (the hidden layer activation is input for the next layer.)

Q: can we apply the same normalization to the hidden layer, i.e. scale the features in the hidden layer zero mean and variance of one.

A: if we force all the layers have zero mean and variance of one, this removes some degree of freedom from the neural network.

We can recover this by including learnable parameter for scale (γ) and shift (β) that have the same shape as input to the layer.

$$\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \beta$$

$$\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon$$

Summary of what's happening inside batch norm operation:

Previous layer output batch \rightarrow scale to have $\mu = 0, \sigma = 1 \rightarrow$ apply $\gamma, \beta \rightarrow$ apply activation (sigmoid, relu, ...) \rightarrow next layer

Notes:

The β in batchnorm, play the role of bias in a neuron. So in Pytorch when you have batchnorm you have an option on setting bias to False.

The original batchnorm paper, put batch norm before activation. But nowadays is more common to put it after activation.

In general you can play with all of this and see which one works better for your dataset: adding/removing bias, batchnorm before and after activation, ...

If the batch size is too small, batchnorm might not work well.

BatchNorm in Pytorch:

```
self.my_network = torch.nn.Sequential(  
    # 1st hidden layer  
    torch.nn.Flatten(),  
    torch.nn.Linear(num_features, num_hidden_1, bias=False),  
    torch.nn.BatchNorm1d(num_hidden_1),  
    torch.nn.ReLU(),  
    # 2nd hidden layer  
    torch.nn.Linear(num_hidden_1, num_hidden_2, bias=False),  
    torch.nn.BatchNorm1d(num_hidden_2),  
    torch.nn.ReLU(),  
    # output layer  
    torch.nn.Linear(num_hidden_2, num_classes)  
)
```

Using Combination of BatchNorm and Dropout:

BatchNorm, Activation, Dropout (BAD) is bad!

When using dropout it is recommended to use batchnorm after activation.

But feel free to play with different combinations

Q: during evaluation, we don't have a minibatch, how do we compute μ and σ ?

A: Pytorch keep the moving average of mean and variance during the training, and use that during the evaluation.

$$\begin{aligned} \text{mean} &= 0.1 * \text{previous_minibatch_mean} + 0.9 * \text{current_minibatch_mean} \\ \text{var} &= 0.1 * \text{previous_minibatch_var} + 0.9 * \text{current_minibatch_var} \end{aligned}$$

Why Batch Normalization Work?

The exact reason why batchnorm works is not well understood.

The original bathnorm paper, the authors talked about the hypothesis that it provides internal covariate shift (a fancy way for saying that the layer input distribution changes). However they did not provide any theoretical guarantee.

Later in a paper by Santurkar, et. Al “how does batchnormalization help optimization” they said that internal covariate shift has little to do with the success of batchnorm.

They claim that batchnorm makes the optimization landscape smooth.

But there’re other papers that claim other things.

There are other papers that claim batchnorm is useless!

Brock, et. al. “high performance large scale image recognition without batch normalization”

In the absence of strong theoretical results about effectiveness of batch normalization, you have to try it for your dataset and see if it works for you.

Exploding gradient- vanishing gradient

This can happen because of multiplying many matrices in the gradient update (due to chain rule).

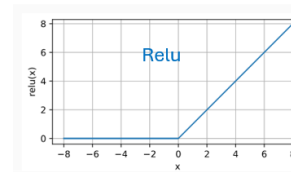
$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \dots \circ f_1(\mathbf{x}).$$

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \dots \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}.$$

```
M = torch.normal(0, 1, size=(4, 4))
print('a single matrix \n',M)
for i in range(100):
    M = M @ torch.normal(0, 1, size=(4, 4))
print('after multiplying 100 matrices\n', M)
```

```
a single matrix
tensor([[ -0.8755, -1.2171,  1.3316,  0.1357],
        [ 0.4399,  1.4073, -1.9131, -0.4608],
        [-2.1420,  0.3643, -0.5267,  1.0277],
        [-0.1734, -0.7549,  2.3024,  1.3085]])
after multiplying 100 matrices
tensor([[ -2.9185e+23,  1.3915e+25, -1.1865e+25,  1.4354e+24],
        [ 4.9142e+23, -2.3430e+25,  1.9979e+25, -2.4169e+24],
        [ 2.6578e+23, -1.2672e+25,  1.0805e+25, -1.3072e+24],
        [-5.2223e+23,  2.4899e+25, -2.1231e+25,  2.5684e+24]])
```

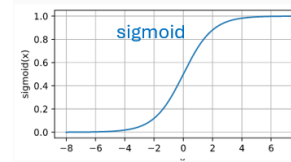
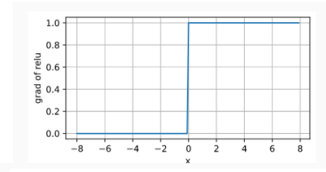
Different Activation Functions



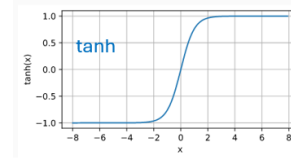
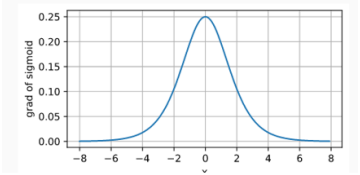
Math equation

$$\text{ReLU}(x) = \max(x, 0).$$

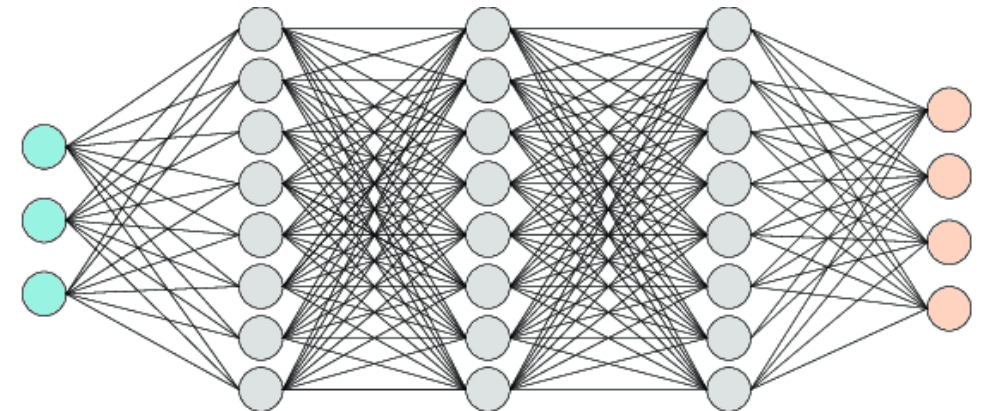
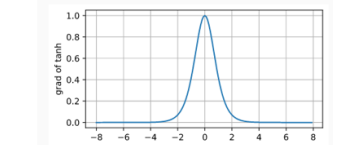
gradient



$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

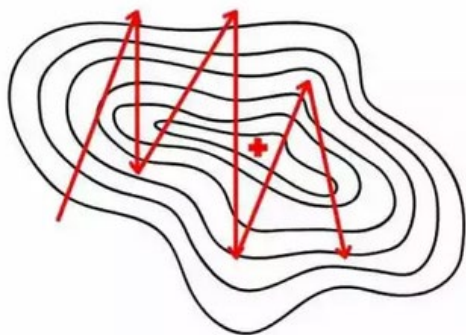


$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

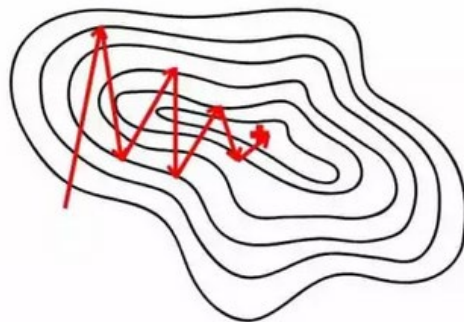


Gradient clipping

Without Gradient Clipping



With Gradient Clipping



Weight Initialization

One of the enabling factors for deep neural networks.

Xavier initialization:

(Xavier Glorot, et. al “Understanding the difficulty of training deep feedforward neural networks”.)

Obviously the weights are initialized randomly. But what should be mean and variance of the weights. Also should all the weights in all the layers have the same variance?

Basic idea:

We need to do statistical analysis on mean and variance of the layers output in both forward pass and backward pass.
and make sure the variance stay bounded.

Theory behind weight initialization

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j.$$

$$\begin{aligned} E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\ &= 0, \end{aligned}$$

Assume input layer x_j have zero mean and variance γ^2 and are independent of w_{ij} and independent of each other .

$$\begin{aligned} \text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\ &= n_{\text{in}} \sigma^2 \gamma^2. \end{aligned}$$

So if we keep $n_{\text{in}} \sigma^2 = 1$, then the output variance does not change.

We can also show that the output variance on backpropagation remain unchanged is $n_{\text{out}} \sigma^2 = 1$

Therefore, the base weight initialization would be:

$$\frac{1}{2} (n_{\text{in}} + n_{\text{out}}) \sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

Keiming He initialization:

Delving Deep into Rectifiers:
Surpassing Human-Level Performance on ImageNet Classification

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun

He initialization is specialized for Relu activation.

■ Xavier Normal initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

■ Xavier Uniform initialization

$$W \sim U\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, +\sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

■ He Normal initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in}}}$$

■ He Uniform initialization

$$W \sim U\left(-\sqrt{\frac{6}{n_{in}}}, +\sqrt{\frac{6}{n_{in}}}\right)$$

Notes:

If batch norm is used, weight initialization becomes less important.

Pytorch uses the He initialization by default. But can be easily changed via `torch.nn.init`

Improving Gradient Descent Optimization

In NN, we look at 1st order optimization methods. The 2nd order methods are computationally more expensive (require Hessian matrix) and do not provide better performance and currently are not being used in deep learning.

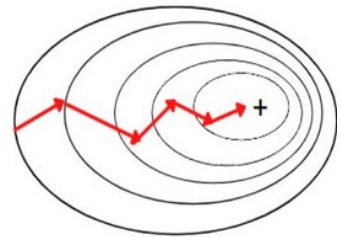
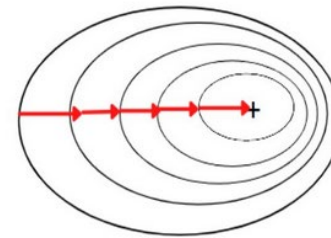
One problem with 1st order methods is their vulnerability to local minimum.

Minibatch training makes the gradient slightly noisy and it's good for avoiding local minima.

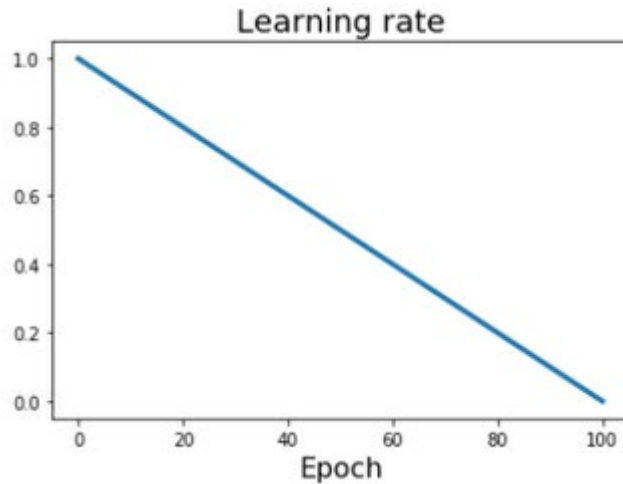
Too small learning rate makes the optimization slow.

Too big learning rate cause the optimization to overshoot the local minima.

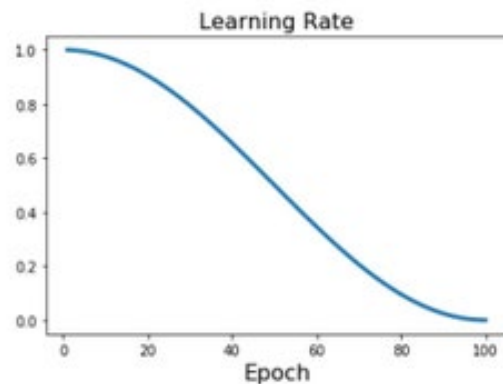
There are various ways of reducing learning rate.



Learning Rate Decay



$$\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$$



In practice it's better to train w/o learning rate decay and then add learning rate decay and see if it makes improvement.

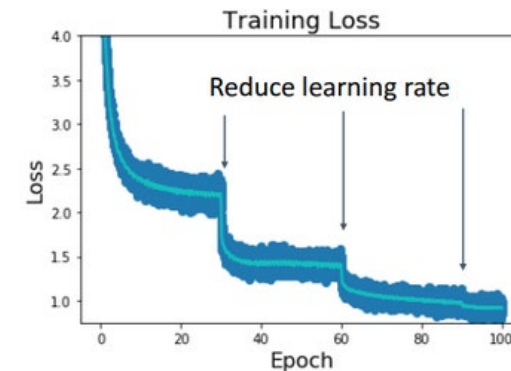
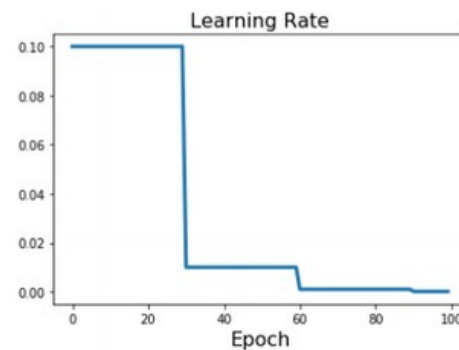
In pytorch: `torch.optim.lr_scheduler`

Docs > `torch.optim` > `ReduceLROnPlateau`

ReduceLROnPlateau

```
CLASS torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=10, threshold=0.0001, threshold_mode='rel', cooldown=0, min_lr=0, eps=1e-08, verbose='deprecated') [SOURCE]
```

Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.



In pytorch you can set your "ReduceLROnPlateau" or any other learning rate scheduler based on either training loss or validation loss.

SGD with momentum

Momentum avoids getting stuck in local minima.



The idea is not only move in the direction of the gradient, but also move in the average direction of the last few updates.

$$V(t) = \beta V(t-1) + (1 - \beta) \frac{\partial L}{\partial w}(t)$$
$$w(t) = w(t-1) - \lambda V(t)$$

Adaptive Learning Rate via RMSProp

These algorithms adaptively adjust the learning rate.

The main idea in adaptive learning algorithms is that the learning rate is decreased if the gradient changes its direction. And learning rate is increased if gradient stays consistent.

This is like slowing down in a zigzag road while driving.

We first form the exponential moving average of squared gradient. Then divide the learning rate by it:

$$E(t) = \gamma E(t - 1) + (1 - \gamma) \left(\frac{\partial L}{\partial w}(t) \right)^2, \gamma \approx 0.9$$

$$w(t) = w(t - 1) - \eta \frac{\partial L}{\partial w}(t) * \frac{1}{\sqrt{\epsilon + E(t)}}$$

Adaptive Learning Rate via ADAM (adaptive momentum estimation)

Adam is a combination of momentum + RMSprop with a slight twist where V_t and E_t are scaled to \hat{V}_t, \hat{E}_t

$$V(t) = \beta V(t-1) + (1 - \beta) \frac{\partial L}{\partial w}(t)$$

$$\hat{V}(t) = \frac{V(t)}{1 - \beta^t}$$

$$E(t) = \gamma E(t-1) + (1 - \gamma) \left(\frac{\partial L}{\partial w}(t) \right)^2$$

$$\hat{E}(t) = \frac{E(t)}{1 - \gamma^t}$$

$$w(t) = w(t-1) - \eta \hat{V}(t) * \frac{1}{\sqrt{\epsilon + \hat{E}(t)}}$$

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

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

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Q: Does adaptive learning rate algorithms such as Aadm always perform better?

The Marginal Value of Adaptive Gradient Methods in Machine Learning

Ashia C. Wilson[‡], Rebecca Roelofs[‡], Mitchell Stern[‡], Nathan Srebro[†], and Benjamin Recht[‡]
{ashia,roelofs,mitchell}@berkeley.edu, nati@ttic.edu, brecht@berkeley.edu

[‡]University of California, Berkeley

[†]Toyota Technological Institute at Chicago

Abstract

Adaptive optimization methods, which perform local optimization with a metric constructed from the history of iterates, are becoming increasingly popular for training deep neural networks. Examples include AdaGrad, RMSProp, and Adam. We show that for simple overparameterized problems, adaptive methods often find drastically different solutions than gradient descent (GD) or stochastic gradient descent (SGD). We construct an illustrative binary classification problem where the data is linearly separable, GD and SGD achieve zero test error, and AdaGrad, Adam, and RMSProp attain test errors arbitrarily close to half. We additionally study the empirical generalization capability of adaptive methods on several state-of-the-art deep learning models. We observe that the solutions found by adaptive methods generalize worse (often *significantly* worse) than SGD, even when these solutions have better training performance. These results suggest that practitioners should reconsider the use of adaptive methods to train neural networks.

Improving Generalization Performance by Switching from Adam to SGD

Nitish Shirish Keskar[†] Richard Socher[†]