Back-propagation, Gradient Descent and Optimization

Thomas Gerald

March 20, 2025

Laboratoire Interdisciplinaire des Sciences du Numérique – LISN, CNRS

The lecture

Content of the lecture: Optimisation Approaches

The lecture

Content of the lecture: Optimisation Approaches

1. Vanishing Gradient activation functions and initialization

The lecture

Content of the lecture: Optimisation Approaches

1. Vanishing Gradient activation functions and initialization
2. Regularization

The lecture

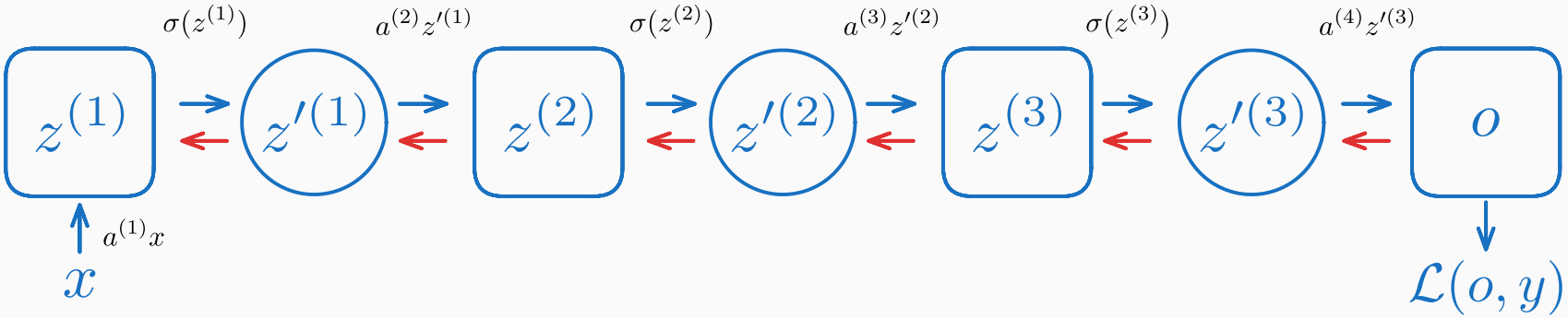
Content of the lecture: Optimisation Approaches

1. Vanishing Gradient activation functions and initialization
2. Regularization
3. Gradient Descent variants

Vanishing Gradient activation function and initialization

Vanishing Gradient: intuition

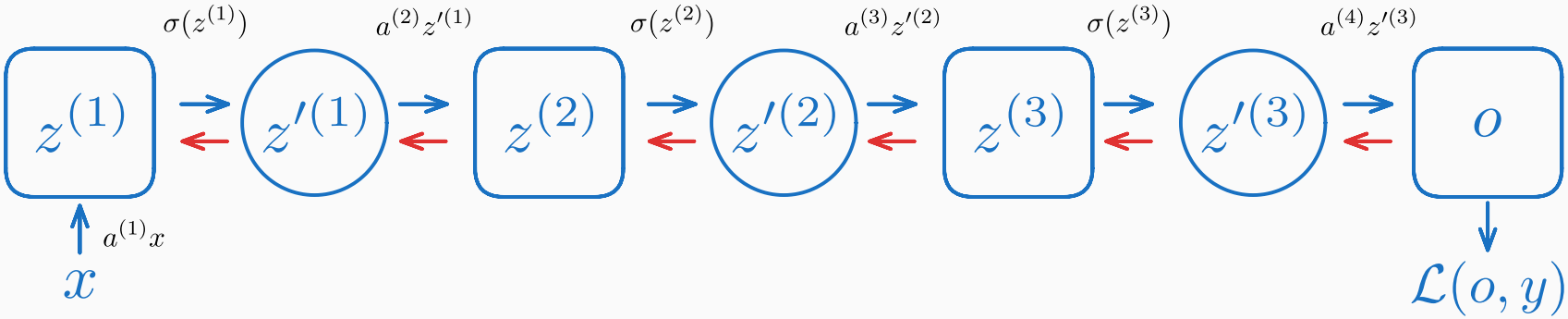
Consider a MLP with only a single neuron in each layer (scalar network)



What is the gradient $\mathcal{L}(o, y)$ with respect to $a^{(1)}$?

Vanishing Gradient: intuition

Consider a MLP with only a single neuron in each layer (scalar network)



What is the gradient $\mathcal{L}(o, y)$ with respect to $a^{(1)}$?

$$\frac{d\mathcal{L}(o,y)}{da^{(1)}} = \frac{d\mathcal{L}(o,y)}{do}\frac{do}{dz'^{(3)}} \times \frac{dz'^{(3)}}{dz^{(3)}} \times \frac{dz^{(3)}}{dz'^{(2)}} \times \frac{dz'^{(2)}}{dz^{(2)}} \times \frac{dz^{(2)}}{dz'^{(1)}} \times \frac{dz'^{(1)}}{dz^{(1)}} \times \frac{dz^{(1)}}{da^{(1)}}$$

$$= \frac{d\mathcal{L}(o,y)}{do} \times a^{(4)} \times \frac{d\sigma(z^{(3)})}{dz^{(3)}} \times a^{(3)} \times \frac{d\sigma(z^{(2)})}{dz^{(2)}} \times a^{(2)} \times \frac{d\sigma(z^{(1)})}{dz^{(1)}} \times a^{(1)}$$

Vanishing Gradient: intuition (Sigmoid)

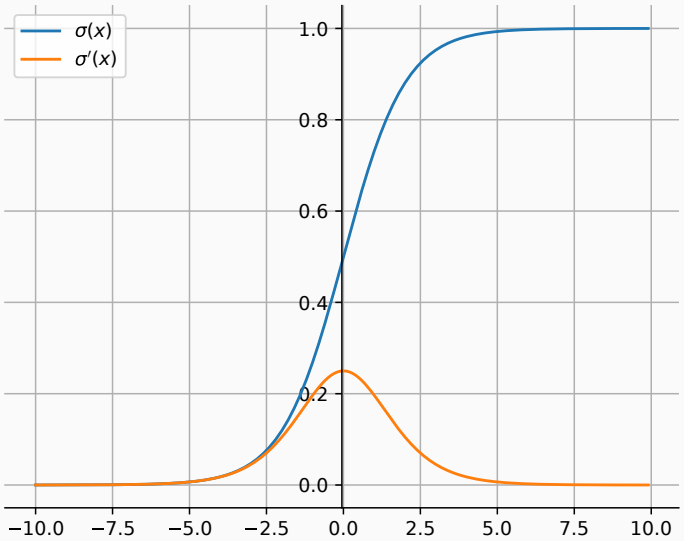


Figure 1: The sigmoid function and its derivative ("logistic function")

Sigmoid function (logistic function)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Vanishing Gradient: intuition (Sigmoid)

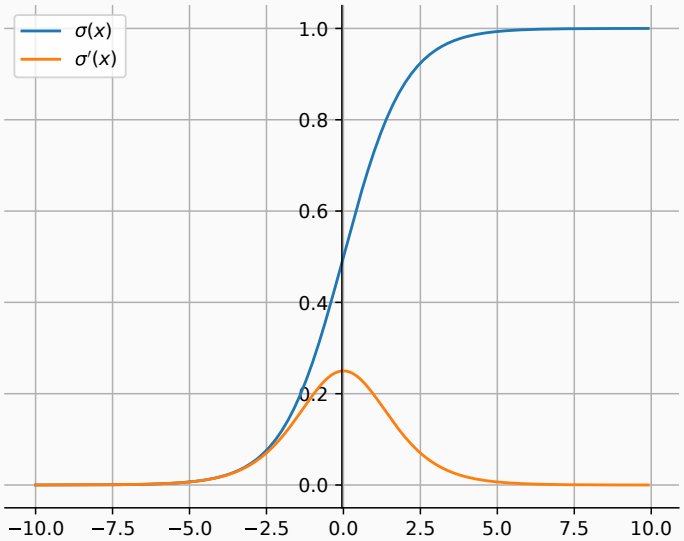


Figure 1: The sigmoid function and its derivative ("logistic function")

Sigmoid function (logistic function)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Vanishing gradient

- If last layers produce value close to 0 or 1

Vanishing Gradient: intuition (Sigmoid)

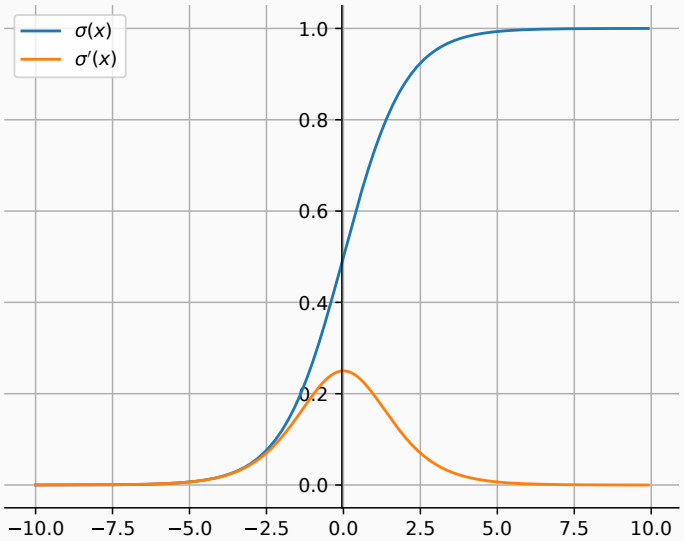


Figure 1: The sigmoid function and its derivative ("logistic function")

Sigmoid function (logistic function)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Vanishing gradient

- If last layers produce value close to 0 or 1
  $\rightarrow$ Very small gradient (then first layer receive a very small gradient)

Vanishing Gradient: intuition (Sigmoid)

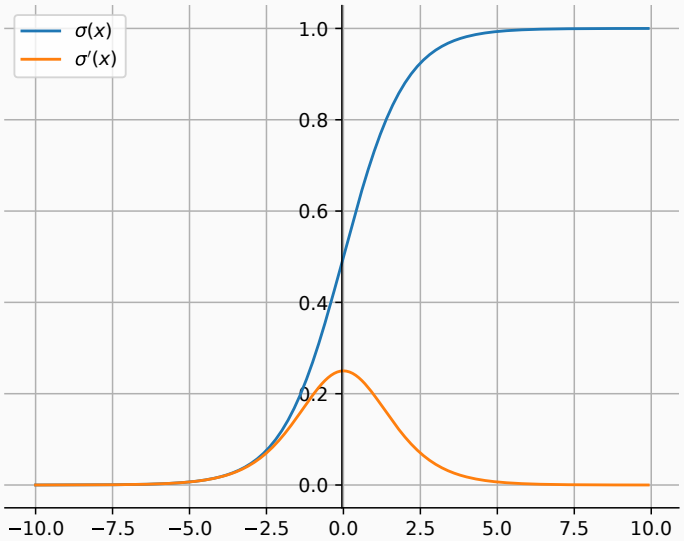


Figure 1: The sigmoid function and its derivative ("logistic function")

Sigmoid function (logistic function)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Vanishing gradient

- If last layers produce value close to 0 or 1
  $\rightarrow$ Very small gradient (then first layer receive a very small gradient)
- In the best cases, we multiply successively by 0.25

activation and Gradient: TanH

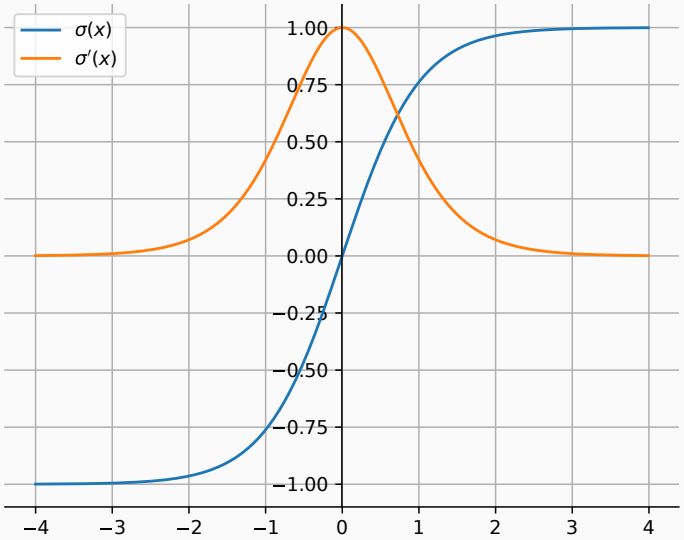


Figure 2: The TanH (Hyperbolic tangent) function and its derivative

TanH function (Hyperbolic tangent)

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\sigma'(x) = 1 - \sigma(x)^2$$

activation and Gradient: TanH

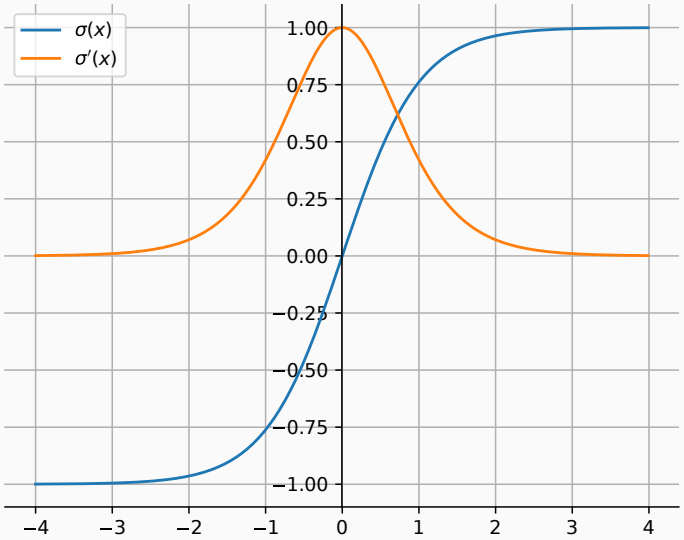


Figure 2: The TanH (Hyperbolic tangent) function and its derivative

TanH function (Hyperbolic tangent)

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\sigma'(x) = 1 - \sigma(x)^2$$

Vanishing gradient

- Better that sigmoid around 0

activation and Gradient: TanH

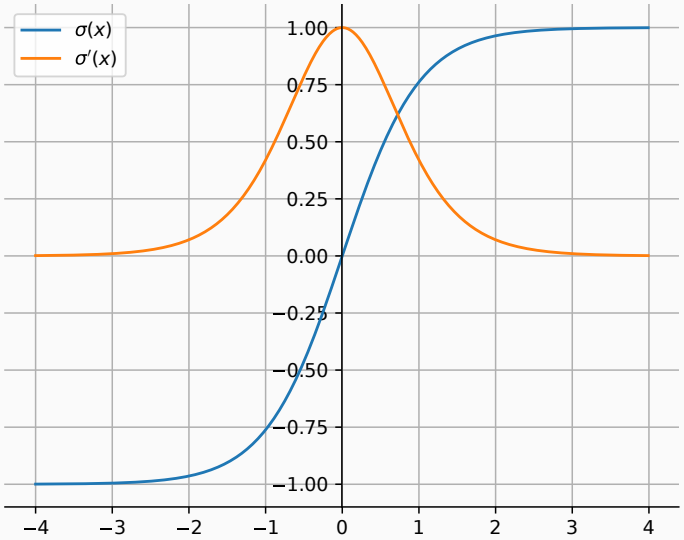


Figure 2: The TanH (Hyperbolic tangent) function and its derivative

TanH function (Hyperbolic tangent)

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\sigma'(x) = 1 - \sigma(x)^2$$

Vanishing gradient

- Better that sigmoid around 0
  $\rightarrow$ Still vanishing gradient issues for strong values

activation and Gradient: TanH

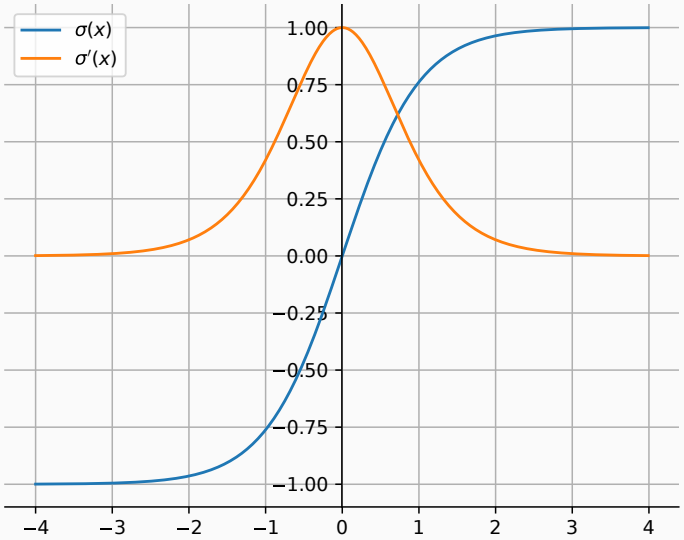


Figure 2: The TanH (Hyperbolic tangent) function and its derivative

TanH function (Hyperbolic tangent)

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\sigma'(x) = 1 - \sigma(x)^2$$

Vanishing gradient

- Better that sigmoid around 0
  $\rightarrow$ Still vanishing gradient issues for strong values
- Popular in Natural Language Processing (e.g. in RNN architectures)

activation and Gradient: ReLU

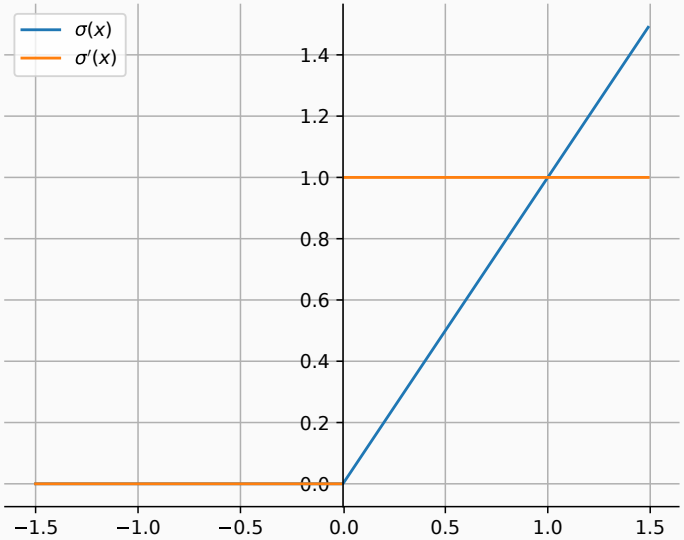


Figure 3: The ReLU (Rectified Linear Unit) function and its derivative

ReLU function (Rectified Linear Unit)

$$\sigma(x) = \begin{cases} x & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

$$\sigma'(x) = \begin{cases} 1 & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

activation and Gradient: ReLU

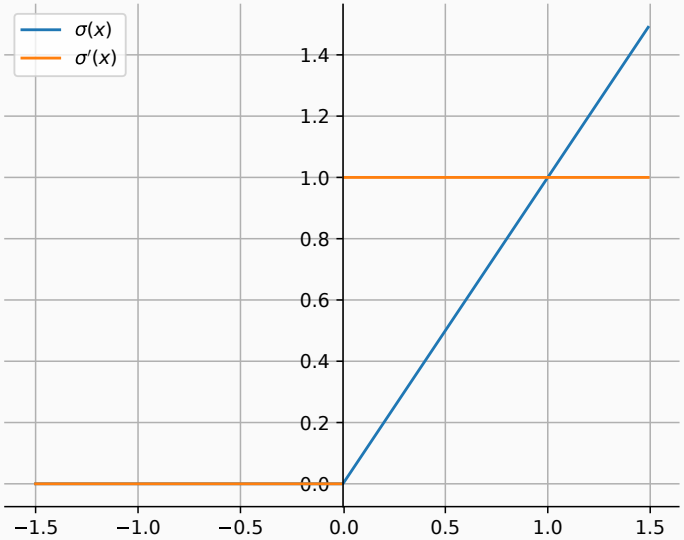


Figure 3: The ReLU (Rectified Linear Unit) function and its derivative

ReLU function (Rectified Linear Unit)

$$\sigma(x) = \begin{cases} x & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

$$\sigma'(x) = \begin{cases} 1 & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

Vanishing gradient

- No vanishing gradient issue but..

activation and Gradient: ReLU

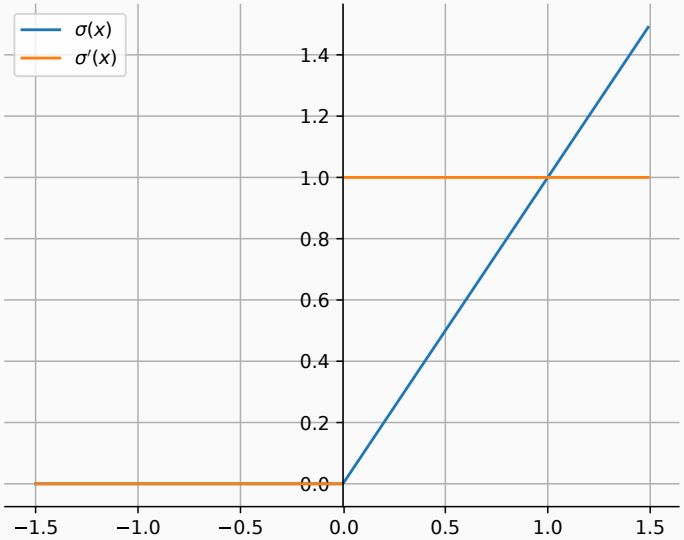


Figure 3: The ReLU (Rectified Linear Unit) function and its derivative

ReLU function (Rectified Linear Unit)

$$\sigma(x) = \begin{cases} x & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

$$\sigma'(x) = \begin{cases} 1 & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

Vanishing gradient

- No vanishing gradient issue but..
- "Dead units" problems (when $x << 0$ for instance a very low bias)

activation and Gradient: ReLU

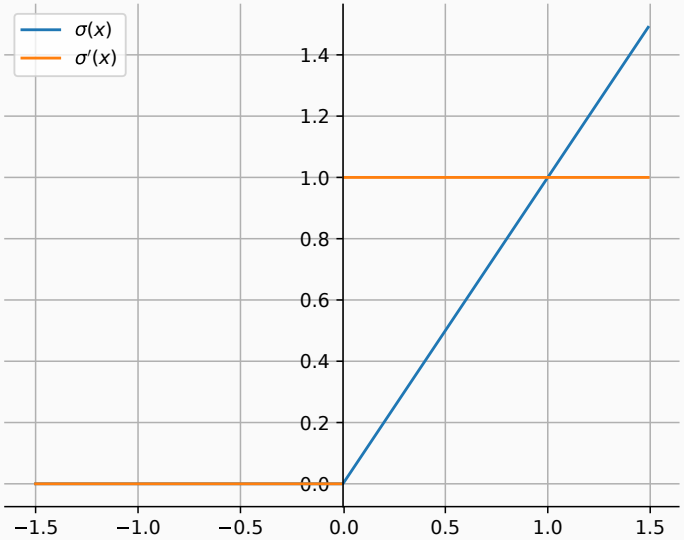


Figure 3: The ReLU (Rectified Linear Unit) function and its derivative

ReLU function (Rectified Linear Unit)

$$\sigma(x) = \begin{cases} x & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

$$\sigma'(x) = \begin{cases} 1 & \text{if x } \geq 0 \\ 0 & \text{if x } < 0 \end{cases}$$

Vanishing gradient

- No vanishing gradient issue but..
- "Dead units" problems (when $x << 0$ for instance a very low bias)
- Used a lot in computer vision (CNN)

Vanishing Gradient: How to prevent it?

What do we want?

- Value close to zero often prevent vanishig gradient
  $\rightarrow$ Ensure input of activation close to zero at the begining of the training
- Having a similar gradient magnitude for all layers
  $\rightarrow$ Avoid one layer to do all the works while others are useless

Vanishing Gradient: How to prevent it?

What do we want?

- Value close to zero often prevent vanishig gradient
  $\rightarrow$ Ensure input of activation close to zero at the begining of the training
- Having a similar gradient magnitude for all layers
  $\rightarrow$ Avoid one layer to do all the works while others are useless

Hyperbolic Tangent
Let $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$:

- A sampled uniformly between $\left[-\frac{\sqrt{6}}{\sqrt{m+n}}, \frac{\sqrt{6}}{\sqrt{m+n}}\right]$
- b initialized to 0

Called Xavier or Glorot initialization

$\rightarrow$ "Understanding the difficulty of training deep feedforward neural networks", Glorot, X. and Bengio, Y. (2010)

Vanishing Gradient: How to prevent it?

What do we want?

- Value close to zero often prevent vanishig gradient
  $\rightarrow$ Ensure input of activation close to zero at the begining of the training
- Having a similar gradient magnitude for all layers
  $\rightarrow$ Avoid one layer to do all the works while others are useless

Hyperbolic Tangent
Let $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$:

- A sampled uniformly between $\left[-\frac{\sqrt{6}}{\sqrt{m+n}}, \frac{\sqrt{6}}{\sqrt{m+n}}\right]$
- b initialized to 0

Called Xavier or Glorot initialization

$\rightarrow$ "Understanding the difficulty of training deep feedforward neural networks", Glorot, X. and Bengio, Y. (2010)

Rectified Linear Unit
Let $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$:

- A sampled uniformly between $\left[-\frac{\sqrt{6}}{\sqrt{n}}, \frac{\sqrt{6}}{\sqrt{n}}\right]$
- b initialized to 0 (or $b = 0.01$)

Called Kaiming or He initialization

$\rightarrow$ "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.", He, K. et.al. (2015)

Regularization approaches

Principle of generalization

Overparameterized neural networks

Networks where the number of parameters exceed the training dataset size.

- Can learn by heart the dataset,
  i.e. **overfit the data** $\rightarrow$ does not generalize well to unseen data
- Are easier to optimize in practice

Principle of generalization

Overparameterized neural networks

Networks where the number of parameters exceed the training dataset size.

- Can learn by heart the dataset,
  i.e. **overfit the data** $\rightarrow$ does not generalize well to unseen data
- Are easier to optimize in practice

Monitoring the training process

- Loss should go down $\Rightarrow$ otherwise your step-size is probably too big!
- Training accuracy should go up
- Dev accuracy should go up $\Rightarrow$ otherwise the network is overfitting!

Principle of generalization

Overparameterized neural networks

Networks where the number of parameters exceed the training dataset size.

- Can learn by heart the dataset,
  i.e. **overfit the data** $\rightarrow$ does not generalize well to unseen data
- Are easier to optimize in practice

Monitoring the training process

- Loss should go down $\Rightarrow$ otherwise your step-size is probably too big!
- Training accuracy should go up
- Dev accuracy should go up $\Rightarrow$ otherwise the network is overfitting!

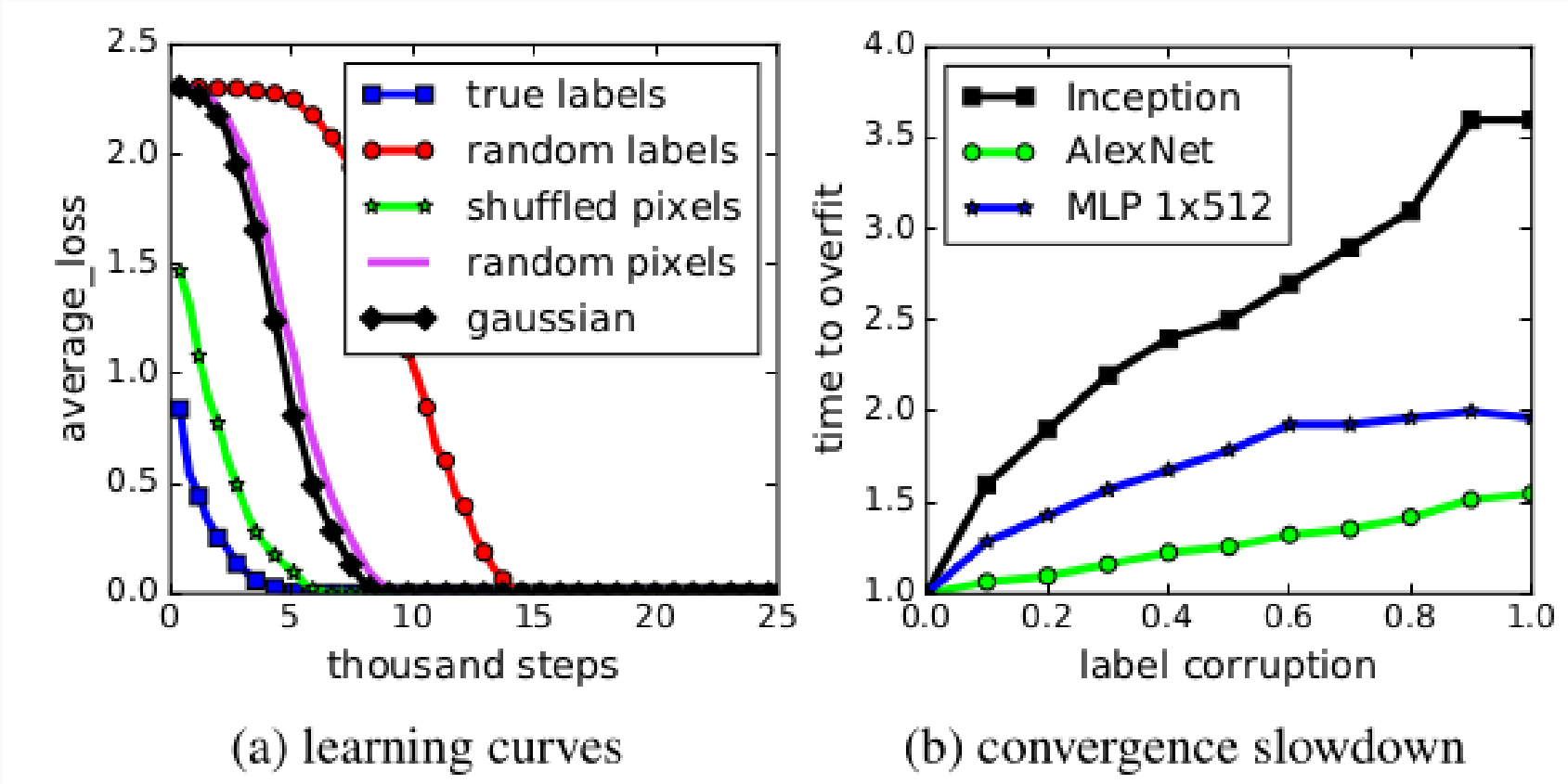
Regularization

Techniques to control parameters during learning and prevent overfitting

Learning with random inputs and labels 1/2 [Zhang et al., 2017]

model	# params	random crop	weight decay	train accuracy	test accuracy
Inception	1,649,402	yes	yes	100.0	89.05
		yes	no	100.0	89.31
		no	yes	100.0	86.03
		no	no	100.0	85.75
(fitting random labels)		no	no	100.0	9.78
Inception w/o BatchNorm	1,649,402	no	yes	100.0	83.00
		no	no	100.0	82.00
(fitting random labels)		no	no	100.0	10.12
Alexnet	1,387,786	yes	yes	99.90	81.22
		yes	no	99.82	79.66
		no	yes	100.0	77.36
		no	no	100.0	76.07
(fitting random labels)		no	no	99.82	9.86
MLP 3x512	1,735,178	no	yes	100.0	53.35
		no	no	100.0	52.39
(fitting random labels)		no	no	100.0	10.48
MLP 1x512	1,209,866	no	yes	99.80	50.39
		no	no	100.0	50.51
(fitting random labels)		no	no	99.34	10.61

Learning with random inputs and labels 2/2 [Zhang et al., 2017]



(a) learning curves (b) convergence slowdown

Regularization L2, gaussian or with weight decay

Regularisation term:

$$\hat{\Theta} = \arg\min_{\Theta} \mathcal{L}(f(x;\Theta), y) + \frac{\lambda}{2}||\Theta||^2$$
$$= \arg\min_{\Theta} \mathcal{L}(f(x;\Theta), y) + \mathcal{R}(\Theta;\lambda)$$

The right term is called the regularization term it can be equivalently interpreted as:

- A soft constraint on the magnitude of parameters (**L2 regularization**)
- A Gaussian prior on parameters $\mathcal{N}(0, \frac{1}{\lambda})$ (**Gaussian regularization**)
- A re-scaling of the parameters during gradient-descent (**weight-decay**)

Using "classical" gradient descent algorithm

Regularization L2, gaussian or with weight decay

Regularisation term:

$$\begin{aligned}\hat{\Theta} &= \arg\min_{\Theta} \mathcal{L}(f(x;\Theta),y) + \frac{\lambda}{2}||\Theta||^2 \\ &= \arg\min_{\Theta} \mathcal{L}(f(x;\Theta),y) + \mathcal{R}(\Theta;\lambda)\end{aligned}$$

During the gradient update:

$$\begin{aligned}\Theta^{t+1} &= \Theta^t - \epsilon\nabla_{\Theta^t}\mathcal{L} - \epsilon\nabla_{\Theta^t}\mathcal{R}(\Theta;\lambda) \\ &= \Theta^t - \epsilon\left(\nabla_{\Theta^t}\mathcal{L} - \nabla_{\Theta^t}\mathcal{R}(\Theta;\lambda)\right)\end{aligned}$$

What is the gradient of the regularization term

$$\frac{\partial\mathcal{R}(\Theta;\lambda)}{\partial\Theta} = \frac{\partial\frac{\lambda}{2}||\Theta||^2}{\partial\Theta} = \frac{2\lambda\Theta}{2} = \lambda\Theta$$

The update $\Theta^{t+1} = \Theta^t - \lambda\Theta^t$ is called weight decay

$\rightarrow$ ⚠ **It is coupled with the optimisation of a loss !!!**

Regularization L2, gaussian or with weight decay: Implementation

Implementation from Pytorch (with minor modifications)

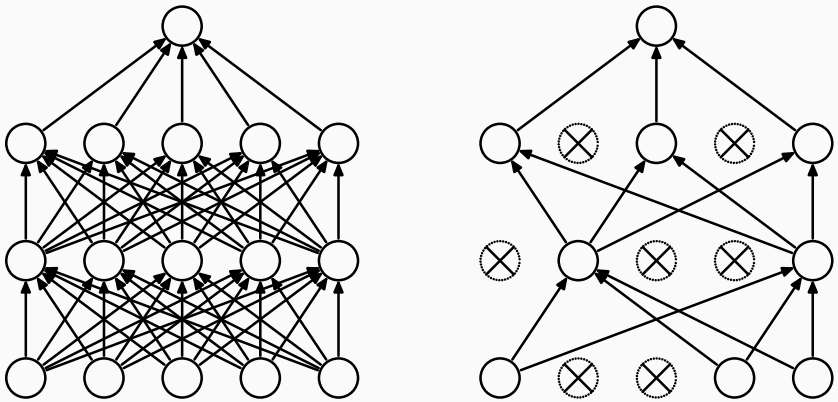
```
class SGD(Optimizer):
  def step(self, closure=None):
    """Performs a single optimization step."""
    for group in self.param_groups: # iterate over parameters
      for p in group['params']:
        if p.grad is None:
          continue

        d_p = p.grad.data # get gradient
        weight_decay = group['weight_decay']
        if weight_decay != 0:
          d_p.add_(weight_decay, p.data) # add weight decay to
     gradient
        p.data.add_(-group['lr'], d_p) # update parameters
```

Dropout

How does dropout work?

- During training, we randomly "turn off" neurons,
  i.e. we randomly set elements of hidden layers $\boldsymbol{z}$ to 0
- During test, we do use the full network

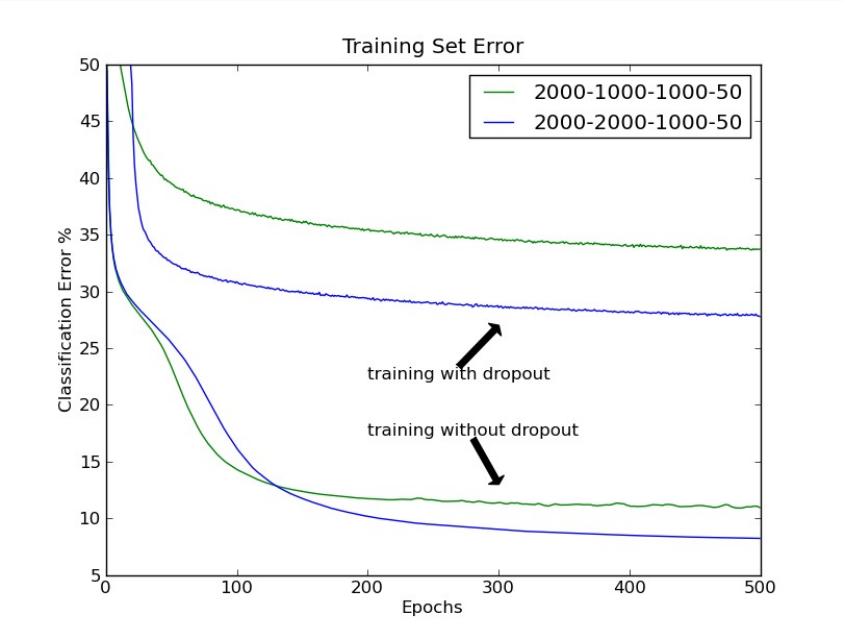


(a) Standard Neural Net (b) After applying dropout.

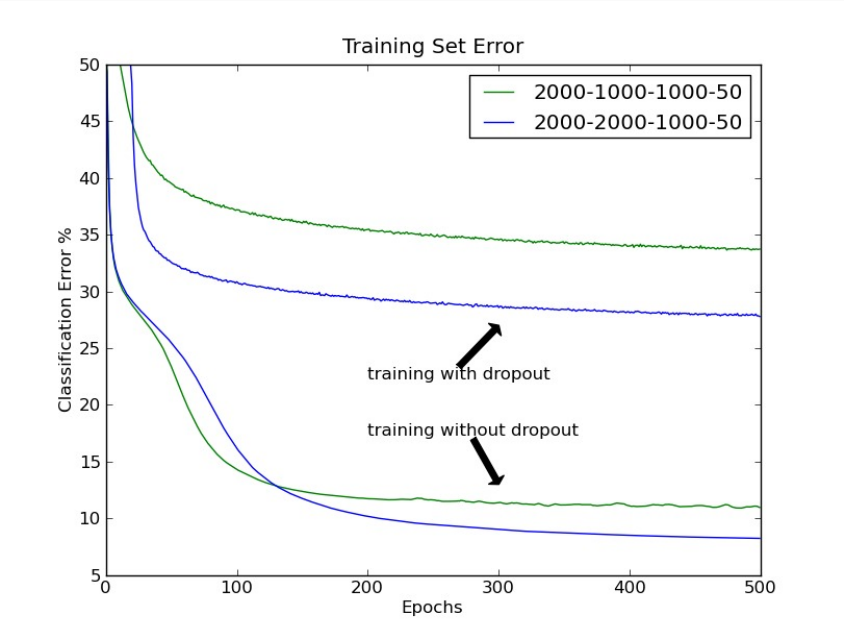
Intuition

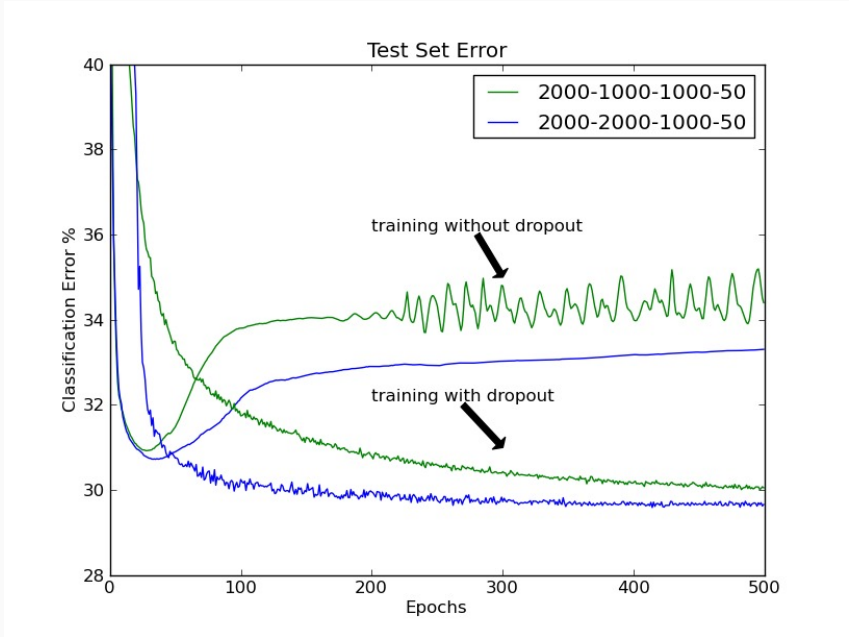
- prevents co-adaptation between units
- equivalent to averaging different models that have different structure but share parameters

Dropout 2/4 [Hinton et al., 2012]



Dropout 2/4 [Hinton et al., 2012]





Dropout 3/4

Dropout layer
A dropout layer is parameterized by the probability of "turning off" a neuron $p \in [0, 1]$:

$$\boldsymbol{z}' = \text{Dropout}(\boldsymbol{z}; p = 0.5)$$

Dropout 3/4

Dropout layer
A dropout layer is parameterized by the probability of "turning off" a neuron $p \in [0, 1]$:

$$\boldsymbol{z}' = \text{Dropout}(\boldsymbol{z}; p = 0.5)$$

Implementation

- $\boldsymbol{z} \in \mathbb{R}^n$: output of a hidden layer
- $p \in [0, 1]$: dropout probability
- $\boldsymbol{m} \in \{0, 1\}^n$: mask vector
- $\boldsymbol{z}'$: hidden values after dropout application

The mask $\boldsymbol{m}$ is a vector of booleans stating if neurons z_i is kept ($m_i = 1$) or "turned off" ($m_i = 0$).

Dropout layer
A dropout layer is parameterized by the probability of "turning off" a neuron $p \in [0, 1]$:

$$\boldsymbol{z}' = \text{Dropout}(\boldsymbol{z}; p = 0.5)$$

Implementation

- $\boldsymbol{z} \in \mathbb{R}^n$: output of a hidden layer
- $p \in [0, 1]$: dropout probability
- $\boldsymbol{m} \in \{0, 1\}^n$: mask vector
- $\boldsymbol{z}'$: hidden values after dropout application

Forward pass:

$$\boldsymbol{m} \sim \text{Bernoulli}(1 - p)$$

$$z_i' = \frac{z_i * m_i}{1 - p}$$

The mask $\boldsymbol{m}$ is a vector of booleans stating if neurons z_i is kept ($m_i = 1$) or "turned off" ($m_i = 0$).

Dropout layer
A dropout layer is parameterized by the probability of "turning off" a neuron $p \in [0, 1]$:

$$\boldsymbol{z}' = \text{Dropout}(\boldsymbol{z}; p = 0.5)$$

Implementation

- $\boldsymbol{z} \in \mathbb{R}^n$: output of a hidden layer
- $p \in [0, 1]$: dropout probability
- $\boldsymbol{m} \in \{0, 1\}^n$: mask vector
- $\boldsymbol{z}'$: hidden values after dropout application

Forward pass:

$$\boldsymbol{m} \sim \text{Bernoulli}(1 - p)$$

$$z_i' = \frac{z_i * m_i}{1 - p}$$

Backward pass:

$$\frac{\partial z_i'}{z_i} = \frac{m}{1 - p}$$

$\Rightarrow$ no gradient for "turned off" neurons.

The mask $\boldsymbol{m}$ is a vector of booleans stating if neurons z_i is kept ($m_i = 1$) or "turned off" ($m_i = 0$).

Dropout 4/4

Where do you apply dropout?

- On the input of the neural network $\boldsymbol{x}$
- **After** activation functions ($sigmoid(0) \neq 0$)
- **Do not** apply dropout on the output logits

Dropout 4/4

Where do you apply dropout?

- On the input of the neural network $\boldsymbol{x}$
- **After** activation functions ($sigmoid(0) \neq 0$)
- **Do not** apply dropout on the output logits

Which dropout probability should you use?

- Empirical question: you have to test!
- Dropout probability at different layers can be different (especially input vs. hidden layers)
- Usually $0.1 \leq p \leq 0.5$

Dropout variants

Dropout can be applied differently for special neural network architectures (e.g. convolutions, recurrent neural networks)

Optimizer

Stochastic Gradient Descent (SGD)

$$\Theta^{(t+1)} = \Theta^{(t)} - \epsilon \nabla_{\Theta^{(t)}} \mathcal{L}$$

Advantages

- Simple
- Single hyper-parameter: the step-size ϵ

Stochastic Gradient Descent (SGD)

$$\Theta^{(t+1)} = \Theta^{(t)} - \epsilon \nabla_{\Theta^{(t)}} \mathcal{L}$$

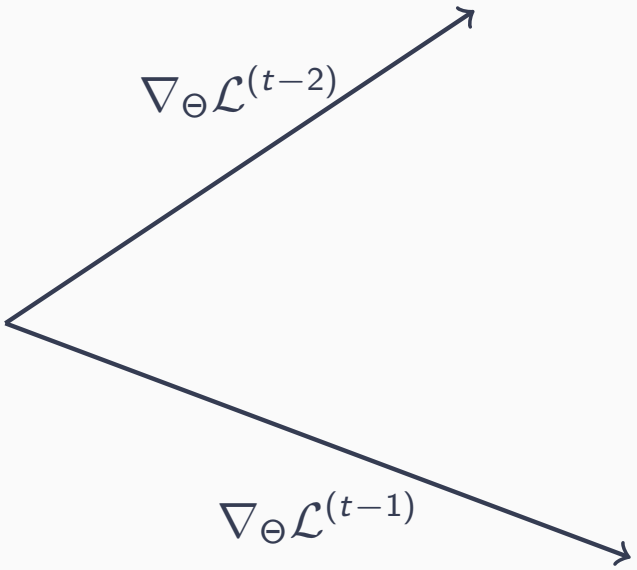
Advantages

- Simple
- Single hyper-parameter: the step-size ϵ

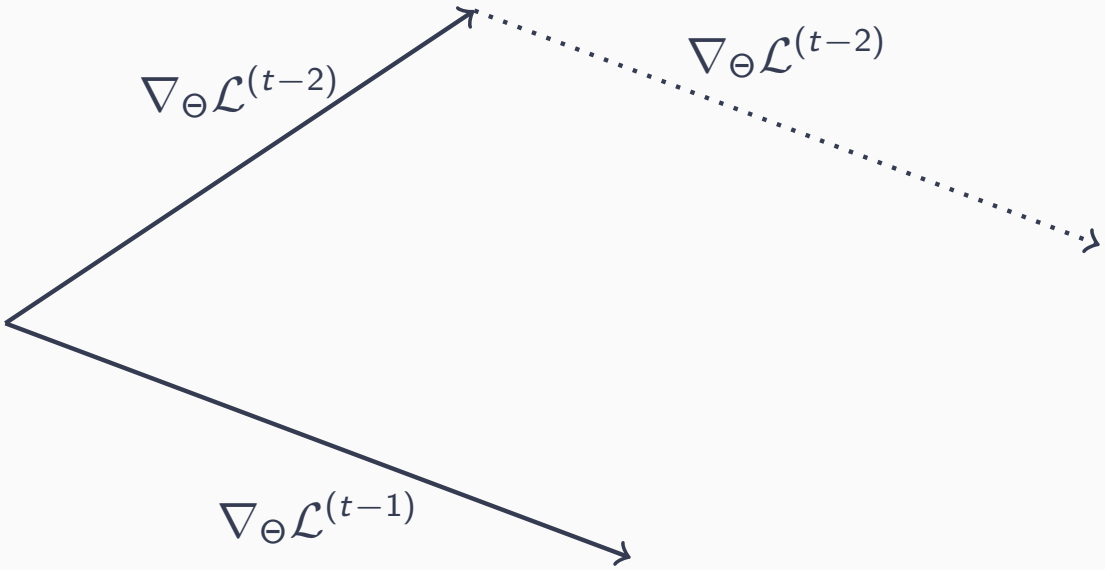
Downsides

- Forget information about previous updates
- Require to search for the best step-size strategy
- Require step-size annealing in practice (decrease step-size): how? what scaling factor?
- Based on first-order information only
  (i.e. the curvature of the optimized function is ignored)

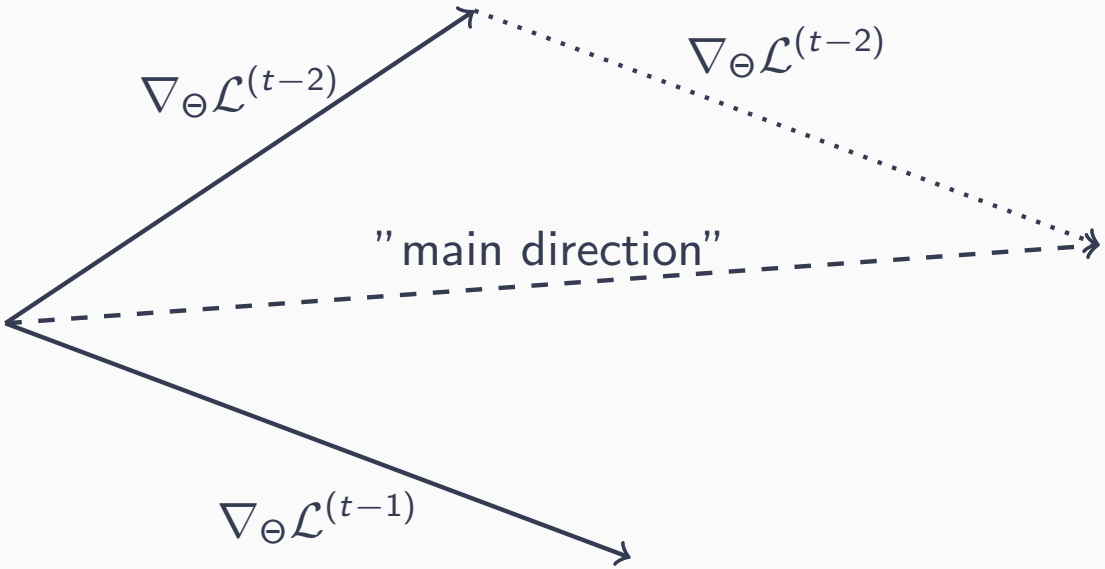
Momentum 1/3



Momentum 1/3


$\nabla_\Theta \mathcal{L}^{(t-2)}$
$\nabla_\Theta \mathcal{L}^{(t-2)}$
$\nabla_\Theta \mathcal{L}^{(t-1)}$

Momentum 1/3


$\nabla_\Theta \mathcal{L}^{(t-2)}$
$\nabla_\Theta \mathcal{L}^{(t-2)}$
"main direction"
$\nabla_\Theta \mathcal{L}^{(t-1)}$

Momentum 2/3

[Polyak, 1964]

- γ: velocity of parameters, i.e. cumulative information about past gradients
- $\mu \in [0, 1]$: momentum, i.e. how much information must be preserved?

$$\gamma^{(t+1)} = \mu\gamma^{(t)} + \nabla_{\Theta}\mathcal{L}$$
$$\Theta^{(t+1)} = \Theta^{(t)} - \epsilon\gamma^{(t+1)}$$

Variants

- Gradient dampening, i.e. diminish the contribution of the current gradient
- Nesterov's Accelerated Gradient [Sutskever et al., 2013]

Momentum 3/3

Implementation from Pytorch (with minor modifications):

```
for group in self.param_groups:
␣for p in group['params']:
␣␣if p.grad is None:
␣␣␣continue
␣
␣d_p = p.grad.data  # get the gradient
␣if momentum != 0:
␣␣param_state = self.state[p]
␣␣if 'momentum_buffer' not in param_state:  # initialize velocity
    vector
␣␣␣buf = param_state['momentum_buffer'] = torch.clone(d_p).detach()
␣␣else:
␣␣␣buf = param_state['momentum_buffer']  # retrieve velocity vector
␣␣␣buf.mul_(momentum).add_(d_p)  # update velocity vector
␣d_p = buf
␣
␣p.data.add_(-group['lr'], d_p)  # update parameters
```

Adaptive learning rates 1/2

Adagrad [Duchi et al., 2011]

- Replace global step-size with dynamic per parameter step-size + global learning rate
- The dynamic per parameter step-size is computed w.r.t. previous gradient l2-norm $\Rightarrow$ parameters with small (resp. large) gradient will have a large (resp. small) step-size

Adaptive learning rates 1/2

Adagrad [Duchi et al., 2011]

- Replace global step-size with dynamic per parameter step-size + global learning rate
- The dynamic per parameter step-size is computed w.r.t. previous gradient l2-norm $\Rightarrow$ parameters with small (resp. large) gradient will have a large (resp. small) step-size

Adadelta [Zeiler, 2012]

- Dynamic per parameter rate is computed with a fixed window of past gradients
- Approximate second-order information to incorporate curvature information $\Rightarrow$ less sensitive to the learning rate hyper-parameter!

Adaptive learning rates 1/2

Adagrad [Duchi et al., 2011]

- Replace global step-size with dynamic per parameter step-size + global learning rate
- The dynamic per parameter step-size is computed w.r.t. previous gradient l2-norm $\Rightarrow$ parameters with small (resp. large) gradient will have a large (resp. small) step-size

Adadelta [Zeiler, 2012]

- Dynamic per parameter rate is computed with a fixed window of past gradients
- Approximate second-order information to incorporate curvature information $\Rightarrow$ less sensitive to the learning rate hyper-parameter!

	SGD	MOMENTUM	ADAGRAD
$\epsilon = 1e^{0}$	**2.26%**	89.68%	43.76%
$\epsilon = 1e^{-1}$	2.51%	**2.03%**	2.82%
$\epsilon = 1e^{-2}$	7.02%	2.68%	**1.79%**
$\epsilon = 1e^{-3}$	17.01%	6.98%	5.21%
$\epsilon = 1e^{-4}$	58.10%	16.98%	12.59%

Table 1. MNIST test error rates after 6 epochs of training for various hyperparameter settings using SGD, MOMENTUM, and ADAGRAD.

Adaptive learning rates 1/2

Adagrad [Duchi et al., 2011]

- Replace global step-size with dynamic per parameter step-size + global learning rate
- The dynamic per parameter step-size is computed w.r.t. previous gradient l2-norm $\Rightarrow$ parameters with small (resp. large) gradient will have a large (resp. small) step-size

Adadelta [Zeiler, 2012]

- Dynamic per parameter rate is computed with a fixed window of past gradients
- Approximate second-order information to incorporate curvature information $\Rightarrow$ less sensitive to the learning rate hyper-parameter!

	SGD	MOMENTUM	ADAGRAD
$\epsilon = 1e^{0}$	**2.26%**	89.68%	43.76%
$\epsilon = 1e^{-1}$	2.51%	**2.03%**	2.82%
$\epsilon = 1e^{-2}$	7.02%	2.68%	**1.79%**
$\epsilon = 1e^{-3}$	17.01%	6.98%	5.21%
$\epsilon = 1e^{-4}$	58.10%	16.98%	12.59%

Table 1. MNIST test error rates after 6 epochs of training for various hyperparameter settings using SGD, MOMENTUM, and ADAGRAD.

	$\rho = 0.9$	$\rho = 0.95$	$\rho = 0.99$
$\epsilon = 1e^{-2}$	2.59%	2.58%	2.32%
$\epsilon = 1e^{-4}$	2.05%	1.99%	2.28%
$\epsilon = 1e^{-6}$	1.90%	**1.83%**	2.05%
$\epsilon = 1e^{-8}$	2.29%	2.13%	2.00%

Table 2. MNIST test error rate after 6 epochs for various hyperparameter settings using ADADELTA.

Adaptive learning rate 2/2

Adam [Kingma and Ba, 2015]

- Combine dynamic per parameter learning rate and momentum
- Initialization bias correction

Convergence issue but works very well in practice [Reddi et al., 2018]
Variants: AdaMax, Nadam [Dozat, 2016], Radam [Liu et al., 2019], AMSGrad

Adaptive learning rate 2/2

Adam [Kingma and Ba, 2015]

- Combine dynamic per parameter learning rate and momentum
- Initialization bias correction

Convergence issue but works very well in practice [Reddi et al., 2018]
Variants: AdaMax, Nadam [Dozat, 2016], Radam [Liu et al., 2019], AMSGrad

Rule of thumb

- Optimizers based on adaptive learning rates usually work out of the box e.g. Adam is really popular in Natural Language Processing
- Fine-tuned SGD with step-size annealing can provide better results at the cost of expensive hyper-parameter tuning

Regularization issue
Weight decay is not equivalent to l2-norm when using adaptive learning rates!

References i

Dozat, T. (2016).

Incorporating nesterov momentum into adam.

ICLR Workshop.

Duchi, J., Hazan, E., and Singer, Y. (2011).

Adaptive subgradient methods for online learning and stochastic optimization.

Journal of Machine Learning Research, 12(Jul):2121–2159.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012).

Improving neural networks by preventing co-adaptation of feature detectors.

CoRR, abs/1207.0580.

References ii

Kingma, D. P. and Ba, J. (2015).

Adam: A method for stochastic optimization.

ICLR.

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., and Han, J. (2019).

On the variance of the adaptive learning rate and beyond.

arXiv preprint arXiv:1908.03265.

Polyak, B. T. (1964).

Some methods of speeding up the convergence of iteration methods.

USSR Computational Mathematics and Mathematical Physics, 4(5):1–17.

Reddi, S. J., Kale, S., and Kumar, S. (2018).

On the convergence of adam and beyond.

ICLR.

References iii

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013).

On the importance of initialization and momentum in deep learning.

In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA. PMLR.

Zeiler, M. D. (2012).

Adadelta: an adaptive learning rate method.

arXiv preprint arXiv:1212.5701.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2017).

Understanding deep learning requires rethinking generalization.

ICLR 2017.