

# Introduction à l'apprentissage automatique - Polytech

## Deep learning - Part 2

Caio Corro

Université Paris-Saclay

# Table of contents

The training loop

Backpropagation

Vanishing gradient, activation functions and initialization

Regularization

Better optimizers

## The training loop

# The big picture

## Data split and usage

- ▶ Training set: to learn the parameters of the network
- ▶ Development (or dev or validation) set: to monitor the network during training
- ▶ Test set: to evaluate our model at the end

Generally you don't have to split the data yourself: there exists standard splits to allow benchmarking.

# The big picture

## Data split and usage

- ▶ Training set: to learn the parameters of the network
- ▶ Development (or dev or validation) set: to monitor the network during training
- ▶ Test set: to evaluate our model at the end

Generally you don't have to split the data yourself: there exists standard splits to allow benchmarking.

## Training loop

1. Update the parameters to minimize the loss on the training set
2. Evaluate the prediction accuracy on the dev set
3. If not satisfied, go back to 1
4. Evaluate the prediction accuracy on the test set with the best parameters on dev

## Pseudo-code

**function** TRAIN( $f, \theta, \mathcal{T}, \mathcal{D}$ )

## Pseudo-code

```
function TRAIN( $f, \theta, \mathcal{T}, \mathcal{D}$ )  
  bestdev =  $-\infty$   
  for epoch = 1 to  $E$  do  
    Shuffle  $\mathcal{T}$   
    for  $x, y \in \mathcal{T}$  do  
      loss =  $\mathcal{L}(f(x; \theta), y)$   
       $\theta = \theta - \epsilon \nabla \text{loss}$ 
```

## Pseudo-code

```
function TRAIN( $f, \theta, \mathcal{T}, \mathcal{D}$ )  
  bestdev =  $-\infty$   
  for epoch = 1 to  $E$  do  
    Shuffle  $\mathcal{T}$   
    for  $x, y \in \mathcal{T}$  do  
      loss =  $\mathcal{L}(f(x; \theta), y)$   
       $\theta = \theta - \epsilon \nabla \text{loss}$   
  
    devacc = EVALUATE( $f, \mathcal{D}$ )  
    if devacc > bestdev then  
       $\hat{\theta} = \theta$   
      bestdev = devacc  
  
return  $\hat{\theta}$ 
```



## Pseudo-code

```
function TRAIN( $f, \theta, \mathcal{T}, \mathcal{D}$ )  
  bestdev =  $-\infty$   
  for epoch = 1 to  $E$  do  
    Shuffle  $\mathcal{T}$   
    for  $x, y \in \mathcal{T}$  do  
      loss =  $\mathcal{L}(f(x; \theta), y)$   
       $\theta = \theta - \epsilon \nabla \text{loss}$   
  
  devacc = EVALUATE( $f, \mathcal{D}$ )  
  if devacc > bestdev then  
     $\hat{\theta} = \theta$   
    bestdev = devacc  
  
return  $\hat{\theta}$ 
```

```
function EVALUATE( $f, \mathcal{D}$ )  
   $n = 0$   
  for  $x, y \in \mathcal{D}$  do  
     $\hat{y} = \arg \max_y f(x; \theta)_y$   
    if  $\hat{y} = y$  then  
       $n = n + 1$   
  
return  $n/|\mathcal{D}|$ 
```

## Further details

### Sampling without replacement

- ▶ shuffle the training set
- ▶ loop over the new order

Experimentally it works better than "true" sampling and it seems to also have good theoretical properties [Nagaraj et al., 2019]

### Verbosity

At each epoch, it is useful to display:

- ▶ mean loss
- ▶ accuracy on training data
- ▶ accuracy on dev data
- ▶ timing information
- ▶ (sometimes) evaluate on dev several times by epoch

## Step-size

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon^{(t)} \nabla \text{loss} \quad \Rightarrow \quad \text{How to choose the step size } \epsilon^{(t+1)}?$$

## Step-size

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon^{(t)} \nabla \text{loss} \quad \Rightarrow \quad \text{How to choose the step size } \epsilon^{(t+1)}?$$

### Convex optimization

- ▶ Nonsummable diminishing step size:  $\sum_{t=1}^{\infty} \epsilon^{(t)} = \infty$  and  $\lim_{t \rightarrow \infty} \epsilon^{(t)} = 0$
- ▶ Backtracking/exact line search

## Step-size

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon^{(t)} \nabla \text{loss} \quad \Rightarrow \quad \text{How to choose the step size } \epsilon^{(t+1)}?$$

### Convex optimization

- ▶ Nonsummable diminishing step size:  $\sum_{t=1}^{\infty} \epsilon^{(t)} = \infty$  and  $\lim_{t \rightarrow \infty} \epsilon^{(t)} = 0$
- ▶ Backtracking/exact line search

### Simple neural network heuristic

1. Start with a small value, e.g.  $\epsilon = 0.01$
2. If dev accuracy did not improve during the last N epochs:  
decay the learning rate by a small value  $\alpha$ , e.g.  $\epsilon = \alpha * \epsilon$  with  $\alpha = 0.1$

## Step-size

$$\theta^{(t+1)} = \theta^{(t)} - \epsilon^{(t)} \nabla \text{loss} \quad \Rightarrow \quad \text{How to choose the step size } \epsilon^{(t+1)}?$$

### Convex optimization

- ▶ Nonsummable diminishing step size:  $\sum_{t=1}^{\infty} \epsilon^{(t)} = \infty$  and  $\lim_{t \rightarrow \infty} \epsilon^{(t)} = 0$
- ▶ Backtracking/exact line search

### Simple neural network heuristic

1. Start with a small value, e.g.  $\epsilon = 0.01$
2. If dev accuracy did not improve during the last N epochs:  
decay the learning rate by a small value  $\alpha$ , e.g.  $\epsilon = \alpha * \epsilon$  with  $\alpha = 0.1$

### Step-size annealing

- ▶ Step decay: multiple  $\epsilon$  by  $\alpha \in [0, 1]$  every N epochs
- ▶ Exponential decay:  $\epsilon^{(t)} = \epsilon^{(0)} \exp(-\alpha \cdot t)$
- ▶  $1/t$  decay:  $\epsilon^{(t)} = \frac{\epsilon^{(0)}}{1+\alpha \cdot t}$

# Backpropagation

# Neural network libraries

## Problem

- ▶ We need the gradient of the objective for training
- ▶ We don't want to compute it by ourselves, too complicated

## Backpropagation algorithm

- ▶ Forward pass: define the function to compute (i.e. the objective)
- ▶ Backward pass: automatically compute the gradient wrt parameters :)

## Computational graph

During the forward pass, we construct a computational graph that retain all operations used to compute the objective



# A typology of neural network libraries

## Static computational graphs

Defines the computation graph once for all, just update the inputs (ex: Tensorflow, Dynet C++ API)

## Dynamic computational graphs

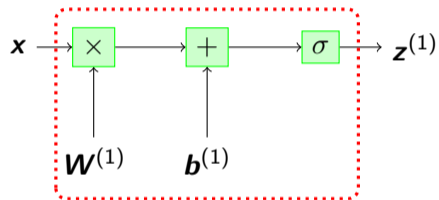
Each time we need to compute a value, we have to rebuild the full graph

- ▶ Eager: computation are done immediately (ex: Pytorch 1&2, Tensorflow)
- ▶ Lazy: first define the computation, the execute it (ex: Dynet, Pytorch 2)  
=> allows for forward pass optimization!

## Computation Graph (CG) 1/2

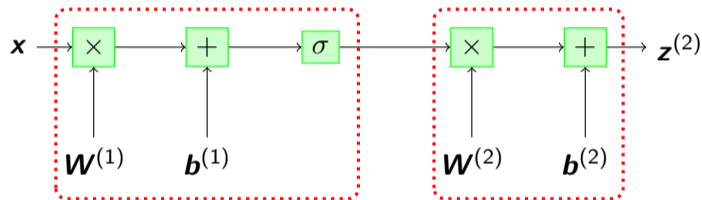
**x**

## Computation Graph (CG) 1/2



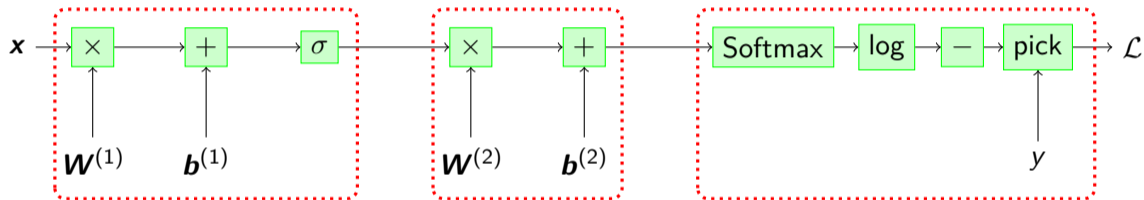
$$\mathbf{z}^{(1)} = \sigma \left( \mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

## Computation Graph (CG) 1/2



$$z^{(1)} = \sigma \left( W^{(1)}x + b^{(1)} \right) \quad z^{(2)} = W^{(2)}x + b^{(2)}$$

## Computation Graph (CG) 1/2

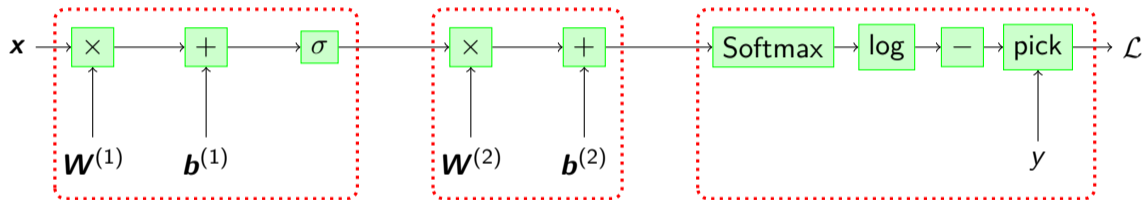


$$z^{(1)} = \sigma \left( W^{(1)}x + b^{(1)} \right)$$

$$z^{(2)} = W^{(2)}x + b^{(2)}$$

$$\mathcal{L} = -\log \frac{\exp(z_y^{(2)})}{\sum_{y'} \exp(z_{y'}^{(2)})}$$

## Computation Graph (CG) 1/2

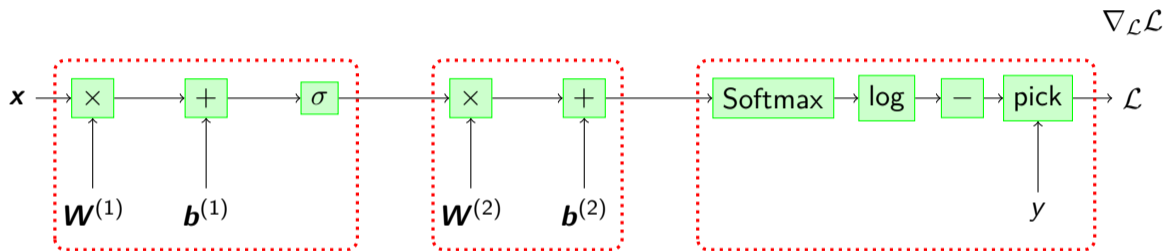


$$z^{(1)} = \sigma \left( W^{(1)}x + b^{(1)} \right)$$

$$z^{(2)} = W^{(2)}x + b^{(2)}$$

$$\mathcal{L} = -\log \frac{\exp(z_y^{(2)})}{\sum_{y'} \exp(z_{y'}^{(2)})}$$

# Computation Graph (CG) 1/2

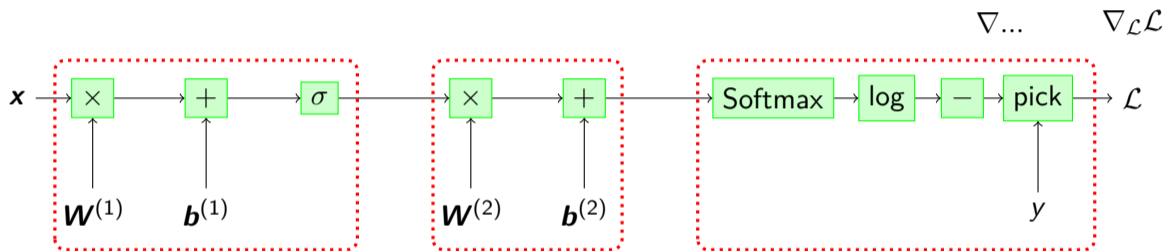


$$z^{(1)} = \sigma \left( W^{(1)}x + b^{(1)} \right)$$

$$z^{(2)} = W^{(2)}x + b^{(2)}$$

$$\mathcal{L} = -\log \frac{\exp(z_y^{(2)})}{\sum_{y'} \exp(z_{y'}^{(2)})}$$

# Computation Graph (CG) 1/2



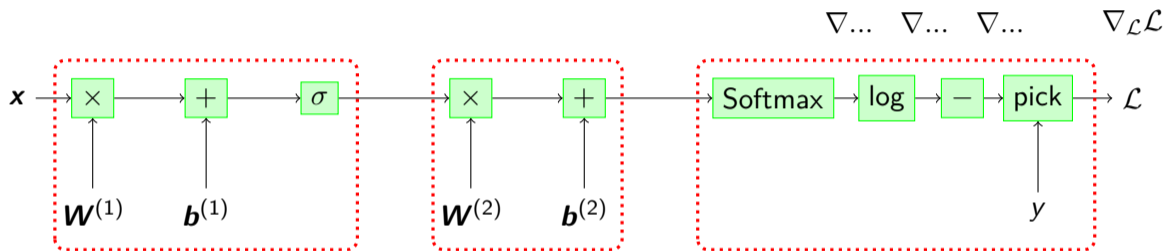
$$z^{(1)} = \sigma \left( W^{(1)}x + b^{(1)} \right)$$

$$z^{(2)} = W^{(2)}x + b^{(2)}$$

$$\mathcal{L} = -\log \frac{\exp(z_y^{(2)})}{\sum_{y'} \exp(z_{y'}^{(2)})}$$



# Computation Graph (CG) 1/2

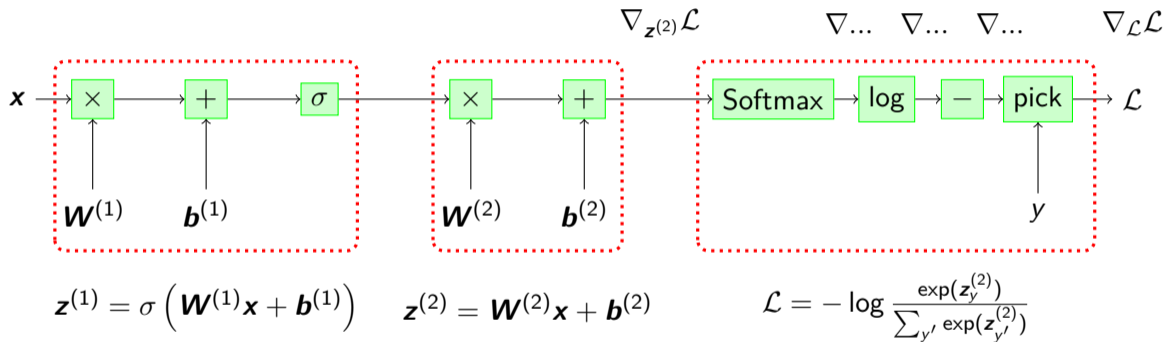


$$z^{(1)} = \sigma \left( W^{(1)}x + b^{(1)} \right)$$

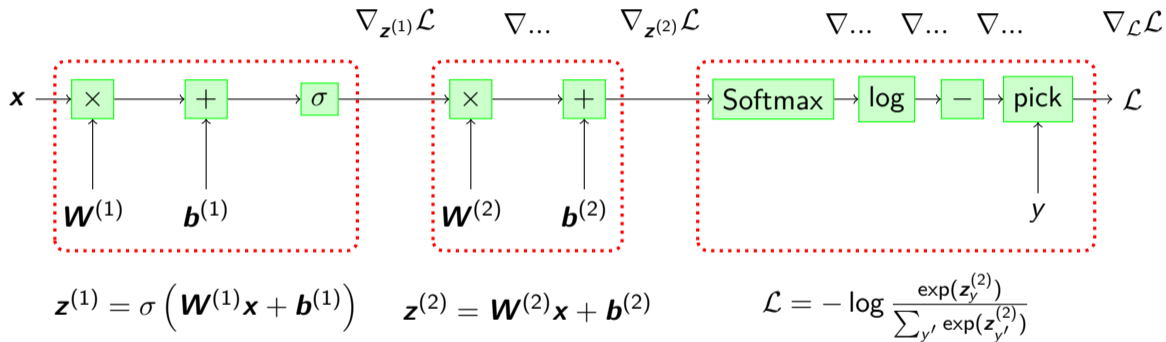
$$z^{(2)} = W^{(2)}x + b^{(2)}$$

$$\mathcal{L} = -\log \frac{\exp(z_y^{(2)})}{\sum_{y'} \exp(z_{y'}^{(2)})}$$

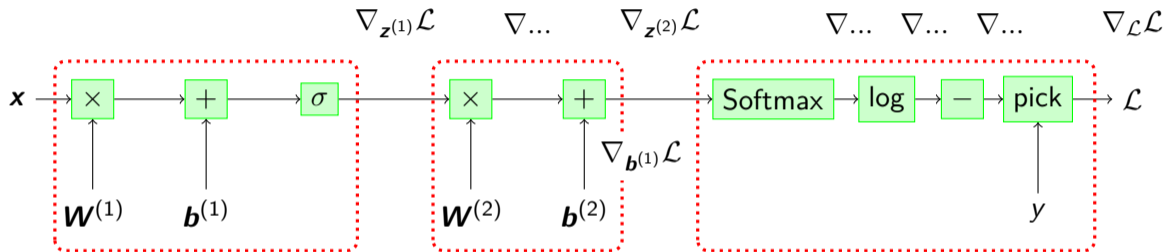
# Computation Graph (CG) 1/2



# Computation Graph (CG) 1/2



# Computation Graph (CG) 1/2

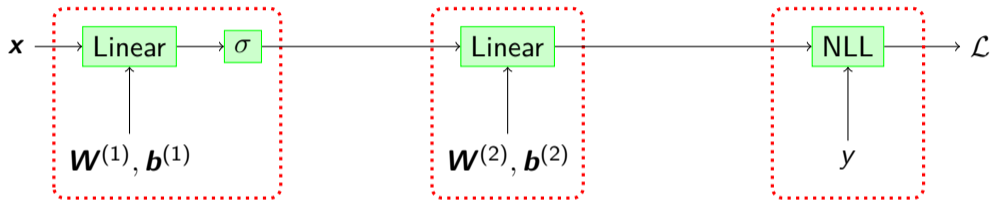


$$\mathbf{z}^{(1)} = \sigma \left( \mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)} \right)$$

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)} \mathbf{x} + \mathbf{b}^{(2)}$$

$$\mathcal{L} = -\log \frac{\exp(\mathbf{z}_y^{(2)})}{\sum_{y'} \exp(\mathbf{z}_{y'}^{(2)})}$$

## Computation Graph (CG) 2/2



$$z^{(1)} = \sigma \left( W^{(1)}x + b^{(1)} \right)$$

$$z^{(2)} = W^{(2)}x + b^{(2)}$$

$$\mathcal{L} = -\log \frac{\exp(z_y^{(2)})}{\sum_{y'} \exp(z_{y'}^{(2)})}$$

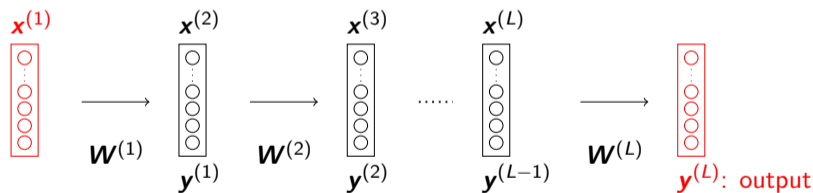
## Vanishing gradient, activation functions and initialization

## Experimental observations

### The MNIST database

8 2 9 4 4 6 4 9 7 0 9 2 9 5 1 5 9 1 0 3  
2 3 5 9 1 7 6 2 8 2 2 5 0 7 4 9 7 8 3 2  
1 1 8 3 6 1 0 3 1 0 0 1 1 2 7 3 0 4 6 5  
2 6 4 7 1 8 9 9 3 0 7 1 0 2 0 3 5 4 6 5

### Comparison of different depth for feed-forward architecture



- ▶ Hidden layers have a sigmoid activation function.
- ▶ The output layer is softmax.

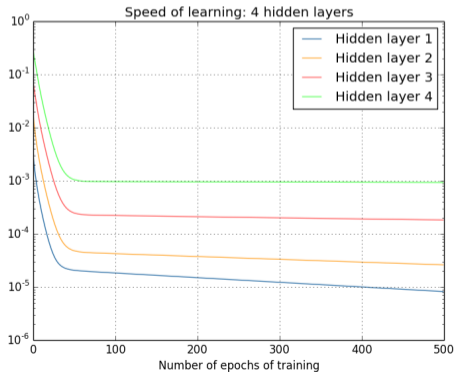
## Experimental observations: <http://neuralnetworksanddeeplearning.com/chap5.html>

- ▶ Without hidden layer:  $\approx 88\%$  accuracy
- ▶ 1 hidden layer (30):  $\approx 96.5\%$  accuracy
- ▶ 2 hidden layer (30):  $\approx 96.9\%$  accuracy
- ▶ 3 hidden layer (30):  $\approx 96.5\%$  accuracy
- ▶ 4 hidden layer (30):  $\approx 96.5\%$  accuracy



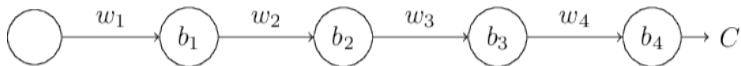
## Experimental observations: <http://neuralnetworksanddeeplearning.com/chap5.html>

- ▶ Without hidden layer:  $\approx 88\%$  accuracy
- ▶ 1 hidden layer (30):  $\approx 96.5\%$  accuracy
- ▶ 2 hidden layer (30):  $\approx 96.9\%$  accuracy
- ▶ 3 hidden layer (30):  $\approx 96.5\%$  accuracy
- ▶ 4 hidden layer (30):  $\approx 96.5\%$  accuracy



## Intuitive explanation 1/2

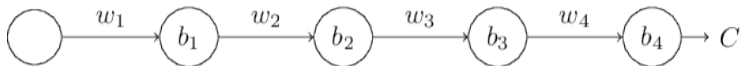
Let consider the simplest deep neural network, with just a single neuron in each layer.



$w_i, b_i$  are resp. the weight and bias of neuron  $i$  and  $C$  some loss function.

## Intuitive explanation 1/2

Let consider the simplest deep neural network, with just a single neuron in each layer.



$w_i, b_i$  are resp. the weight and bias of neuron  $i$  and  $C$  some loss function.

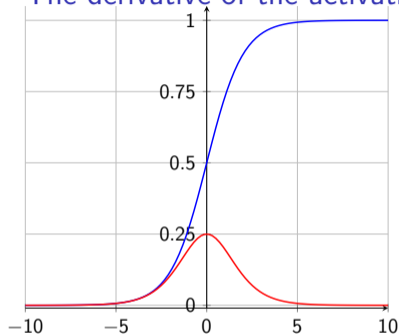
Compute the gradient of  $C$  w.r.t the bias  $b_1$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial y_4} \times \frac{\partial y_4}{\partial a_4} \times \frac{\partial a_4}{\partial y_3} \times \frac{\partial y_3}{\partial a_3} \times \frac{\partial a_3}{\partial y_2} \times \frac{\partial y_2}{\partial a_2} \times \frac{\partial a_2}{\partial y_1} \times \frac{\partial y_1}{\partial a_1} \times \frac{\partial a_1}{\partial b_1} \quad (1)$$

$$= \frac{\partial C}{\partial y_4} \times \sigma'(a_4) \times w_4 \times \sigma'(a_3) \times w_3 \times \sigma'(a_2) \times w_2 \times \sigma'(a_1) \quad (2)$$

## Intuitive explanation 2/2

The derivative of the activation function:  $\sigma'$



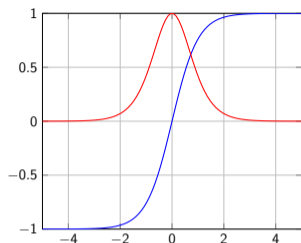
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

### Vanishing gradient

- ▶ if the last layer are well trained (and outputs "strong values" close to 0 or 1),
- ▶ early layers receive a really small incoming gradient.

In the "best case", we successive multiplications by 0.25!

## Other activation functions

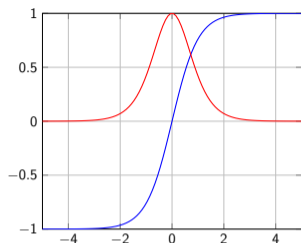


### Hyperbolic tangent

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)} \quad \tanh'(x) = 1 - \tanh(x)^2$$

- ▶ Better gradient than sigmoid around 0
- ▶ Popular in Natural Language Processing

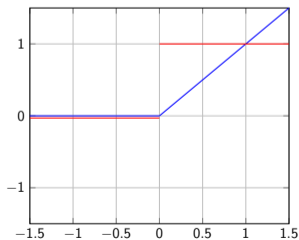
## Other activation functions



### Hyperbolic tangent

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)} \quad \tanh'(x) = 1 - \tanh(x)^2$$

- ▶ Better gradient than sigmoid around 0
- ▶ Popular in Natural Language Processing



### Rectified Linear Unit

$$\text{relu}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{otherwise} \end{cases} \quad \text{relu}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- ▶ No vanishing gradient issue
- ▶ "Dead units" problem (i.e.  $b_i \ll 0$ )
- ▶ Popular in Computer Vision (very deep networks)

# Parameters initialization

## What do we want?

- ▶ Values close to 0 prevent gradient vanishing  
(or gradient exploding/disappearing in the case of relu)
- ▶ Gradient magnitude approximately similar for all layers  
(to prevent that a subset of layers do all the works while others are useless)

# Parameters initialization

## What do we want?

- ▶ Values close to 0 prevent gradient vanishing (or gradient exploding/disappearing in the case of relu)
- ▶ Gradient magnitude approximately similar for all layers (to prevent that a subset of layers do all the works while others are useless)

## Hyperbolic tangent

Let  $\mathbf{W} \in \mathbb{R}^{m \times n}$  and  $\mathbf{b} \in \mathbb{R}^m$ :

- ▶  $\mathbf{W} \sim \mathcal{U} \left[ -\frac{\sqrt{6}}{\sqrt{m+n}}, +\frac{\sqrt{6}}{\sqrt{m+n}} \right]$
- ▶  $\mathbf{b} = 0$

Usually called Xavier or Glorot initialization  
[Glorot and Bengio, 2010]

## Rectified Linear Unit

Let  $\mathbf{W} \in \mathbb{R}^{m \times n}$  and  $\mathbf{b} \in \mathbb{R}^m$ :

- ▶  $\mathbf{W} \sim \mathcal{U} \left[ -\frac{\sqrt{6}}{\sqrt{n}}, +\frac{\sqrt{6}}{\sqrt{n}} \right]$
- ▶  $\mathbf{b} = 0$

(or  $\mathbf{b} = 0.01$  to prevent dying units)

Usually called Kaiming or He initialization  
[He et al., 2015]



# Regularization

# 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** → does not generalize well to unseen data
- ▶ Are easier to optimize in practice

# 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** → does not generalize well to unseen data
- ▶ Are easier to optimize in practice

## Monitoring the training process

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

# 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** → does not generalize well to unseen data
- ▶ Are easier to optimize in practice

## Monitoring the training process

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

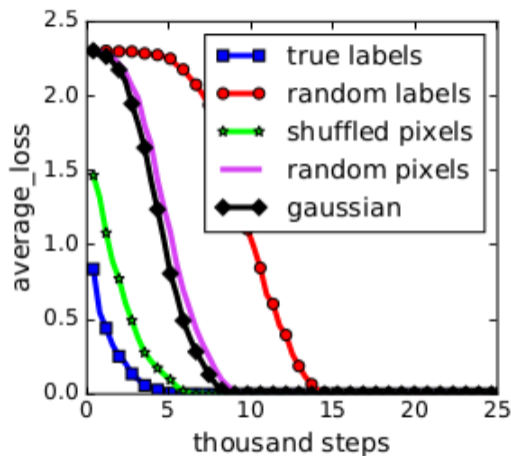
## Regularization

Techniques to control parameters during learning and prevent overfitting

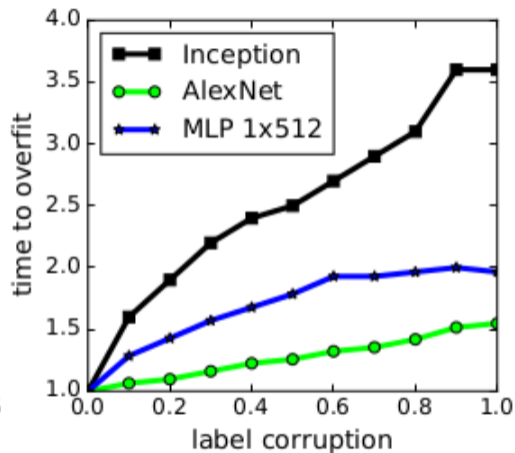
# Learning with random inputs and labels 1/2 [Zhang et al., 2017]

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
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
MLP 3x512	1,735,178	no	yes	100.0	53.35
		no	no	100.0	52.39
		(fitting random labels)	no	no	100.0
MLP 1x512	1,209,866	no	yes	99.80	50.39
		no	no	100.0	50.51
		(fitting random labels)	no	no	99.34

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



(a) learning curves



(b) convergence slowdown

## L2 Regularization or weight decay

### L2 regularization

$$\begin{aligned}\hat{\theta} &= \arg \min_{\theta} \ell(y, s_{\theta}(x)) + \frac{\beta}{2} \|\theta\|^2 \\ &= \arg \min_{\theta} \ell(y, s_{\theta}(x)) + \mathcal{R}(\theta; \beta)\end{aligned}$$

- ▶ We don't actually care about the regularization term value, we only care about its gradient
- ▶ The regularization term is expensive to compute, and even "difficult" to define (need to list all of the parameters of the networks)

### Weight decay

- ▶ Simply modify the gradient instead of adding a term in the objective
- ▶ We can show it is equivalent to L2 regularization

$$\theta^{(t+1)} = (1 - \lambda)\theta^{(t)} - \epsilon \nabla_{\theta^{(t)}} \ell(y, s_{\theta^{(t)}}(x))$$

## L2Regularization or weight decay 3/3

Implementation from Pytorch (slightly modified):

```
class SGD(Optimizer):
    def step(self, closure=None):
        """Performs a single optimization step."""
        for group in self.param_groups:
            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 the gradient

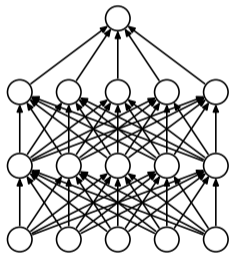
                p.data.add_(-group['lr'], d_p) # update parameters
```



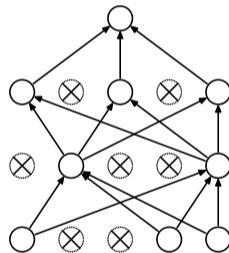
# Dropout 1/4 [Hinton et al., 2012, Srivastava et al., 2014]

## How does dropout work?

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



(a) Standard Neural Net

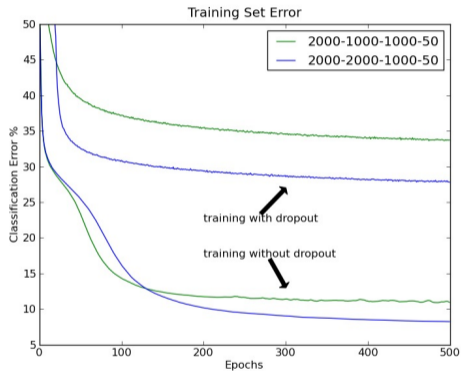


(b) After applying dropout.

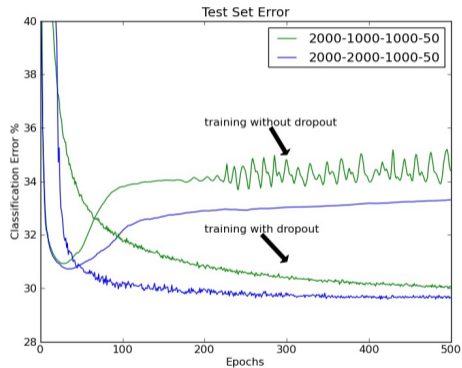
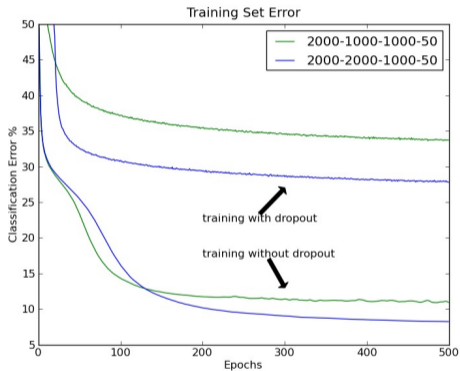
## Intuition

- ▶ 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  $p \in [0, 1]$ :

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

## Dropout 3/4

### Dropout layer

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

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

### Implementation

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

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

## Dropout 3/4

### Dropout layer

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

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

### Implementation

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

#### Forward pass:

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

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

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

## Dropout 3/4

### Dropout layer

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

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

### Implementation

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

#### Forward pass:

$$\mathbf{m} \sim \text{Bernoulli}(1 - p)$$
$$z'_i = \frac{z_i * m_i}{1 - p}$$

#### Backward pass:

$$\frac{\partial z'_i}{z_i} = \frac{m_i}{1 - p}$$

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

The mask  $\mathbf{m}$  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  $\mathbf{x}$
- ▶ **After** activation functions ( $\sigma(0) \neq 0$ )
- ▶ **Do not** apply dropout on the output logits



## Dropout 4/4

### Where do you apply dropout?

- ▶ On the input of the neural network  $\mathbf{x}$
- ▶ **After** activation functions ( $\sigma(0) \neq 0$ )
- ▶ **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)

Better optimizers

# Stochastic Gradient Descent (SGD)

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

## Advantages

- ▶ Simple
- ▶ Single hyper-parameter: the step-size  $\epsilon$

# Stochastic Gradient Descent (SGD)

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

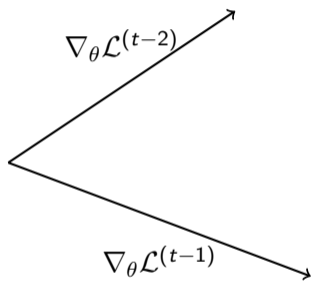
## Advantages

- ▶ Simple
- ▶ Single hyper-parameter: the step-size  $\epsilon$

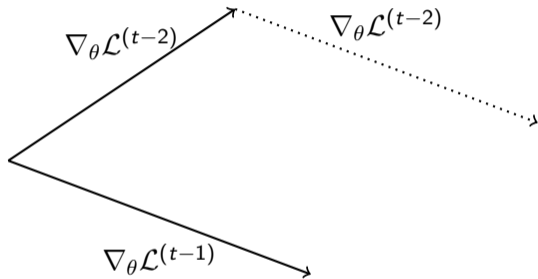
## Downsides

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

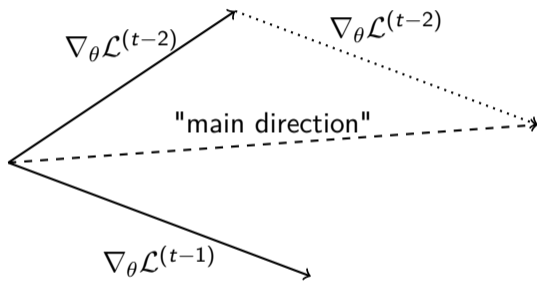
## Momentum 1/3



# Momentum 1/3



## Momentum 1/3



## Momentum 2/3

[Polyak, 1964]

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

$$\begin{aligned}\gamma^{(t+1)} &= \mu\gamma^{(t)} + \nabla_{\theta}\mathcal{L} \\ \theta^{(t+1)} &= \theta^{(t)} - \epsilon\gamma^{(t+1)}\end{aligned}$$

### 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 (slightly modified):

```
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  $l_2$ -norm  
⇒ 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  $l_2$ -norm  
⇒ 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  
⇒ 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  
⇒ 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  
⇒ less sensitive to the learning rate hyper-parameter!

	SGD	MOMENTUM	ADAGRAD
$\epsilon = 1e^0$	<b>2.26%</b>	89.68%	43.76%
$\epsilon = 1e^{-1}$	2.51%	<b>2.03%</b>	2.82%
$\epsilon = 1e^{-2}$	7.02%	2.68%	<b>1.79%</b>
$\epsilon = 1e^{-3}$	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  
⇒ 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  
⇒ less sensitive to the learning rate hyper-parameter!

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

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

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

**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  $l_2$ -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.
-  Glorot, X. and Bengio, Y. (2010).  
Understanding the difficulty of training deep feedforward neural networks.  
In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.






## References II

-  He, K., Zhang, X., Ren, S., and Sun, J. (2015).  
Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.  
*In Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV '15*, pages 1026–1034, Washington, DC, USA. IEEE Computer Society.
-  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.
-  Kingma, D. P. and Ba, J. (2015).  
Adam: A method for stochastic optimization.  
*ICLR*.



## References III

-  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*.
-  Nagaraj, D., Jain, P., and Netrapalli, P. (2019). SGD without replacement: Sharper rates for general smooth convex functions. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4703–4711, Long Beach, California, USA. PMLR.
-  Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

## References IV

-  Reddi, S. J., Kale, S., and Kumar, S. (2018).  
On the convergence of adam and beyond.  
*ICLR*.
-  Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014).  
Dropout: A simple way to prevent neural networks from overfitting.  
*Journal of Machine Learning Research*, 15:1929–1958.
-  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.

## References V

-  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*.