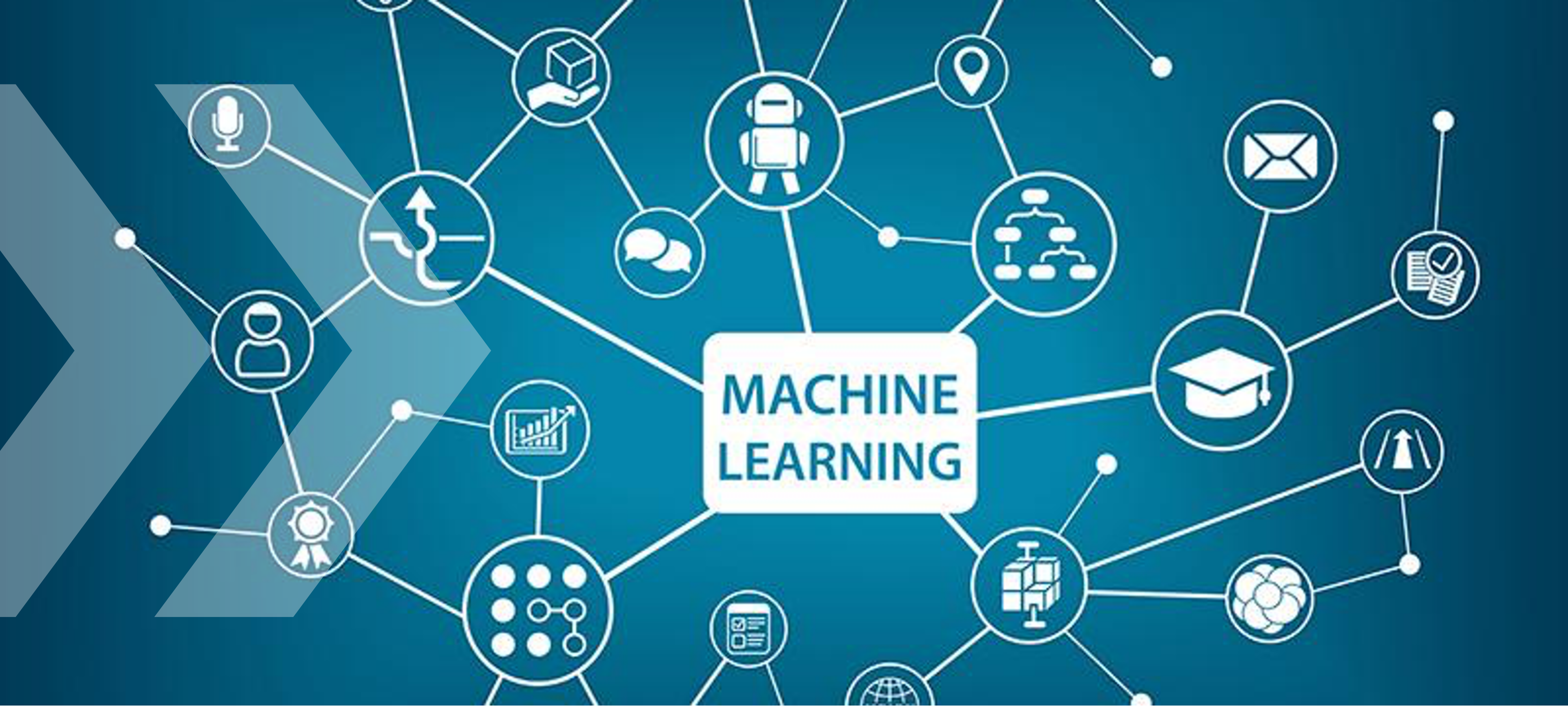


CS480/680: Intro to ML

Lecture 13: Deep Neural Networks

UNIVERSITY OF WATERLOO

Universality: the dark side

- Only proves the **existence** of such a NN; how to find it is another issue (training of NNs) → NP-hard

- May need exponentially many hidden units

- Increase depth may reduce number of hidden units, significantly

Backprop for NN

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
$\quad \boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
$\quad \boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

forward

f: activation function
J = L + λΩ: training obj.

After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ **do**
Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
Compute gradients on weights and biases (including the regularization term, where needed):
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for

backward

Sigmoid and tanh

Sigmoid

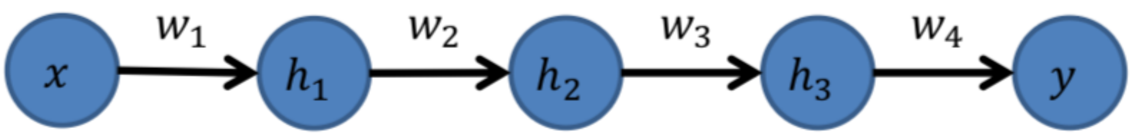
- Output range (0,1)
- Gradient range (0,1): $\sigma(x)\big(1-\sigma(x)\big)$
- Small gradient at saturated regions
  - x = 0, gradient = 0.25
  - x = 10, gradient = 4.5396e-05
  - x = -10, gradient = 4.5396e-05

Tanh

- Output range (-1,1)
- Gradient range (0,1): $1-\tanh^2(x)$
- Small gradient at saturated regions

Vanishing gradients

$$y = \sigma\left(w_4\,\sigma\left(w_3\,\sigma\left(w_2\,\sigma(w_1\,x)\right)\right)\right)$$



- Common weight initialization in (-1, 1)
- Denote input of the i-th $\sigma()$ as a_i

This leads to vanishing gradients:

$$\frac{\partial y}{\partial w_4} = \sigma'(a_4)\sigma(a_3)$$

$$\frac{\partial y}{\partial w_3} = \sigma'(a_4)w_4\sigma'(a_3)\sigma(a_2) \leq \frac{\partial y}{\partial w_4}$$

$$\frac{\partial y}{\partial w_2} = \sigma'(a_4)w_4\sigma'(a_3)w_3\sigma'(a_2)\sigma(a_1) \leq \frac{\partial y}{\partial w_3}$$

$$\frac{\partial y}{\partial w_1} = \sigma'(a_4)w_4\sigma'(a_3)w_3\sigma'(a_2)w_2\sigma'(a_1)x \leq \frac{\partial y}{\partial w_2}$$

UNIVERSITY OF WATERLOO

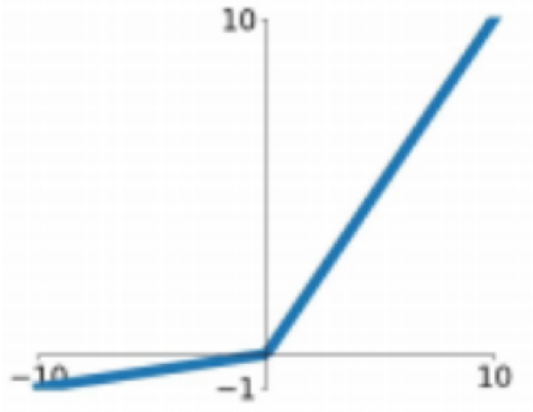
Avoiding vanishing gradients

- Proper initialization of network parameters
- Choose activation functions that do not saturate
- Use Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) architectures.
- …

More activations

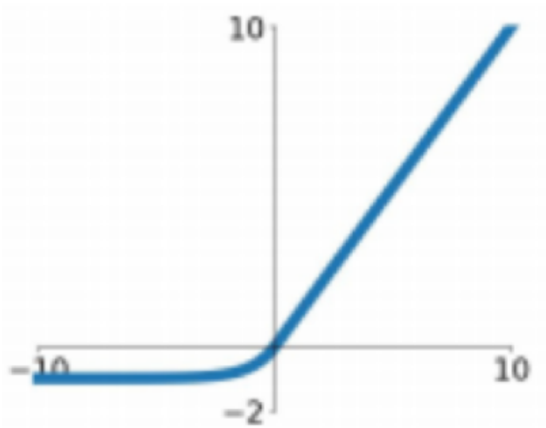
- **Leaky ReLU**

$$f(t) = \max\{0.1t, t\}$$



- **ELU**

$$f(t) = \begin{cases} t, & t \geq 0 \\ \alpha(e^t - 1), & t \leq 0 \end{cases}$$



UNIVERSITY OF WATERLOO

ReLU, Leaky ReLU, and ELU

ReLU

- Computationally efficient
- Gradient = 1 when x> 0
- Gradient = 0 when x<0 (cannot update parameter)
  - Initialize ReLU units with slightly positive values, e.g., 0.01
  - Try other activation functions, e.g., Leaky ReLU

Leaky ReLU

- Computationally efficient
- Constant gradient for both x>0 and x<0

ELU

- Small gradient at negative saturation region

Tips

- Sigmoid and tanh functions are sometimes avoided due to the vanishing/exploding gradients

- Use ReLU as a default choice

- Explore other activation functions, e.g., Leaky ReLU and ELU, if ReLU results in dead hidden units

Overfitting and regularization

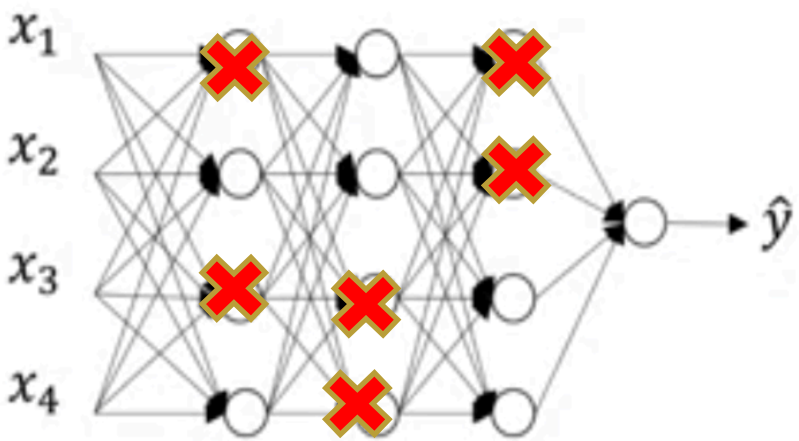
- High capacity increases risk of overfitting
  - # of parameters is often larger than # of data

- Regularization: modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error
  - Parameter norm penalties (e.g., L1 and L2 norms)
  - Bagging
  - Dropout
  - Data augmentation
  - Early stopping
  - …

Dropout

For each training example keep hidden units with probability p.

- A different and random network for each training example
- A smaller network with less capacity



e.g., let p = 0.5 for all hidden layers

delete ingoing and outgoing links for eliminated hidden units

Implementation

Consider implementing dropout in layer 1.

- Denote output of layer 1 as **h** with dimension 4*1
- Generate a 4*1 vector **d** with elements 0 or 1 indicating whether units are kept or not
- Update **h** by multiplying **h** and **d** element-wise
- Update **h** by **h**/p ("inverted dropout")

Inverted dropout

Motivation: keep the mean value of output unchanged

Example: Consider $\boldsymbol{h}^{(1)}$ in the previous slide. $\boldsymbol{h}^{(1)}$ is used as the input to layer 2. Let $p = 0.8$.

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}$$

- 20% of hidden units in layer 1 are eliminated
- Without "inverted dropout", the mean of $z^{(2)}$ would be decreased by 20% roughly

Prediction

Training

- Use ("inverted") dropout for each training example

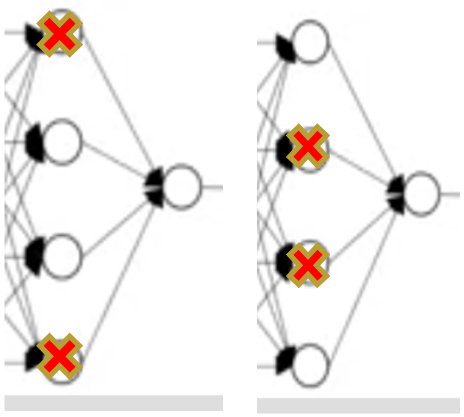
Prediction

- Usually do not use dropout
- If "inverted dropout" is used in training, no further scaling is needed in testing

Why does it work?

Intuition: Since each hidden unit can disappear with some probability, the network cannot rely on any particular units and have to spread out the weights

→ similar to L2 regularization



Another interpretation:

Dropout is training a large ensemble of models sharing parameters

Hyperparameter p

p: probability of keeping units

How to design p?

- Keep p the same for all layers
- Or, design p for each layer: set a lower p for overfitting layers, i.e., layers with a large number of units
- Can also design p for the input, but usually set $p = 1$ or very close to 1

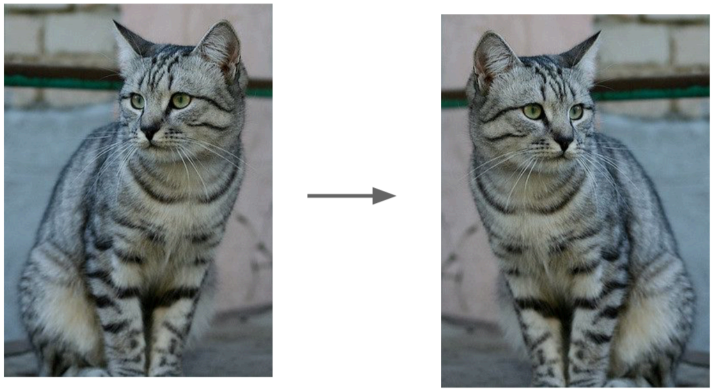
Data augmentation

The best way to improve generalization is to have more training data. But often, data is limited.

One solution: create fake data, e.g., transform input

Example: object recognition (classification)

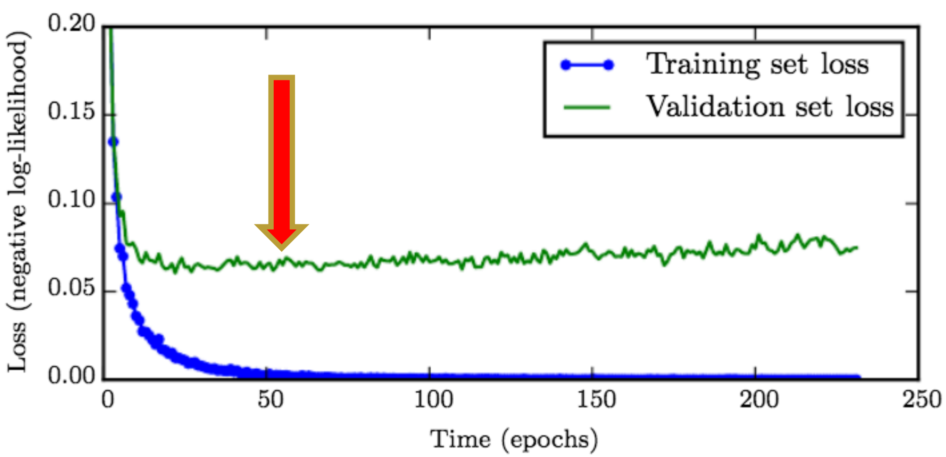
- Horizontal/vertical flip
- Rotation
- Stretching
- …



Early stopping

For models with large capacity,

- training error decreases steadily
- validation error first decreases then increases



UNIVERSITY OF WATERLOO

Batch normalization (BN)

Use BN for each dimension in hidden layers, either before activation function or after

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

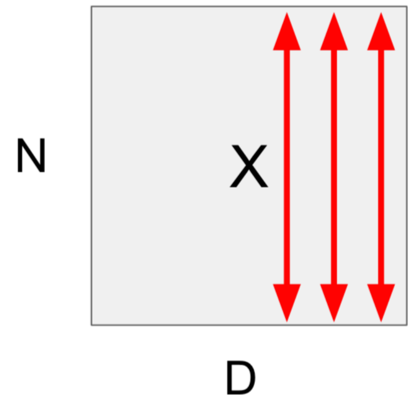
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



$$\mathbf{h}^{(3)} = f_3(\underbrace{\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}}_{z^{(3)}})$$

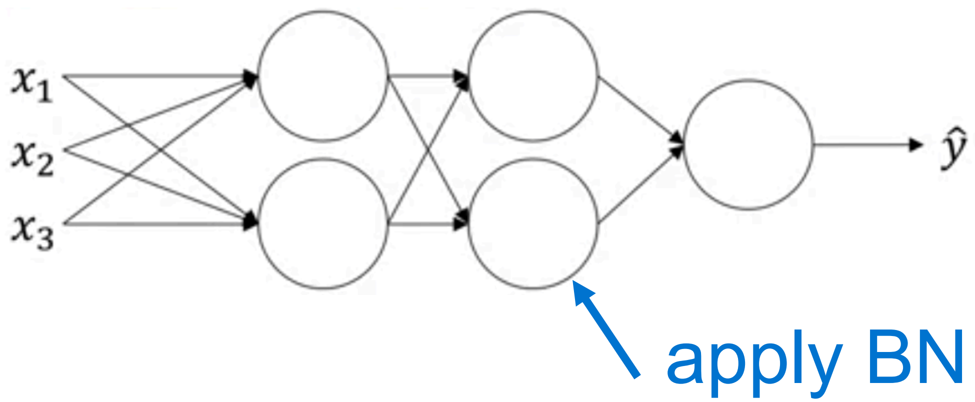
either apply BN to $z^{(3)}$ or $h^{(3)}$

γ and β are optimization variables; not hyperparameters

Why BN? (1)

Consider applying BN to z (i.e., before activation)

- z : mean β, variance γ^2
- Robust to parameter changes in previous layers
- Independent learning of each layer



Without BN, layer 2 depends on output of layer 1 hence params of layer 1
With BN, mean and variance of $z^{(2)}$ unchanged

Why BN? (2)

Assume the orders of features in hidden layers are significantly different

- E.g., feature 1: 1; feature 2: 10^3 ; feature 3: 10^{-3}
- With BN, features are of similar scale
- Speed up learning

Why BN? (3)

BN has a slight regularization effect (not intended)

- Each mini-batch is scaled by the mean/variance computed on that mini-batch
- This adds some noise to z and to each layer's activations

Optimization

Problem of SGD: slow convergence

SGD + momentum

$$\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}) \right)$$

$$\mathbf{v} \leftarrow \alpha \mathbf{v} + (1 - \alpha)\mathbf{g}$$ exponentially weighted averages

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \mathbf{v}$$

- Accumulate an exponentially decaying moving average of past gradients
- Hyperparameter α determines how quickly the contributions of previous gradients exponentially decay (α = 0.9 usually works well)

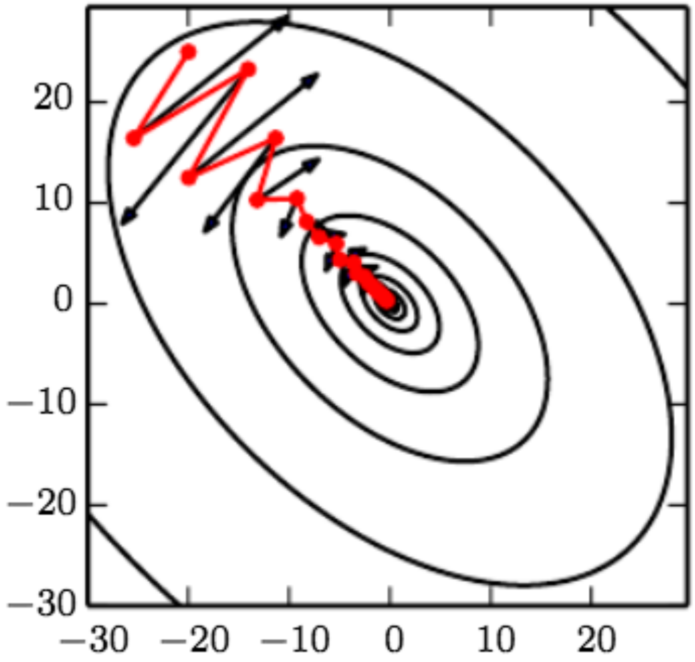
Exponentially weighted averages

Let $\mathbf{v}_t = \alpha \mathbf{v}_{t-1} + (1-\alpha)\mathbf{g}_t; \quad \alpha = 0.9$

$$\mathbf{v}_{100} = 0.1\mathbf{g}_{100} + 0.1 * 0.9\mathbf{g}_{99} + 0.1 * 0.9^2\mathbf{g}_{98} + \cdots + 0.1 * 0.9^{99}\mathbf{g}_1$$

total weight: $1 - \alpha^t$ (close to 1 if t is large)

Illustration



Contour of loss function

- red: SGD + momentum
- black: SGD

RMSProp (root mean square propagation)

- Greater progress in the more gently sloped directions
- One of the go-to optimization methods for deep learning

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.
end while

discard history from extreme past

Adam (adaptive moments)

A variant combining of RMSProp and momentum

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$ both 1st and 2nd moments included
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$ when t is large, the effect of bias correction is small
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Learning rate

SGD, SGD+Momentum, AdaGrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

→Reduce learning rate over time!

- Step decay

  e.g., decrease learning rate by halving every few epochs

- Exponential decay

$$\alpha = \alpha_0 e^{-kt}$$

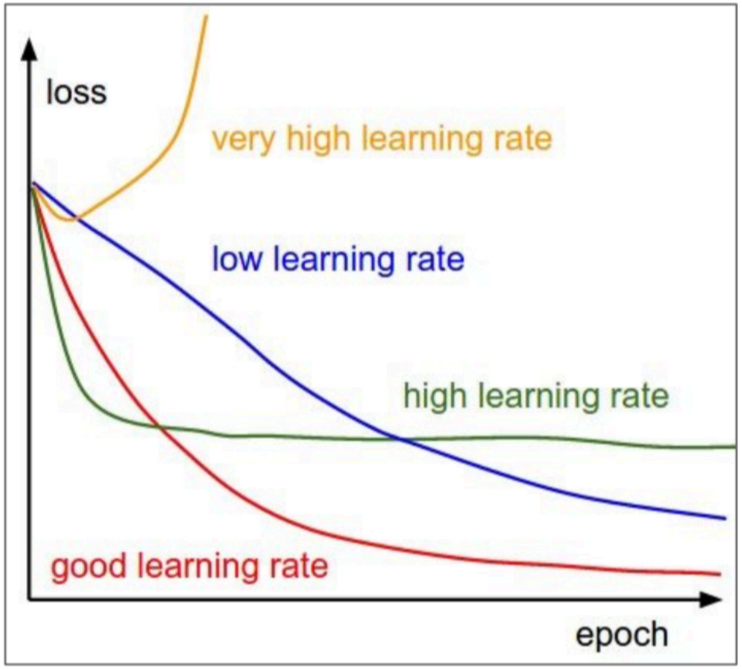
- 1/t decay

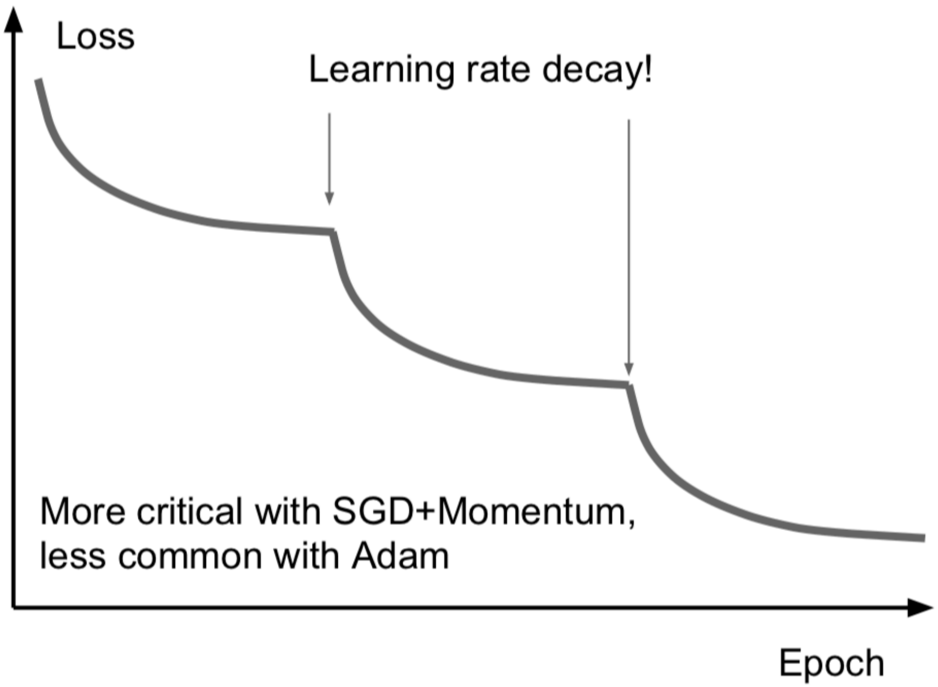
$$\alpha = \alpha_0 / (1 + kt)$$

t: iteration number

UNIVERSITY OF WATERLOO

Illustration





UNIVERSITY OF WATERLOO

Questions?