Deep Learning

Abdelhak Mahmoudi
abdelhak.mahmoudi@um5.ac.ma

ENSIAS- 2020-2021

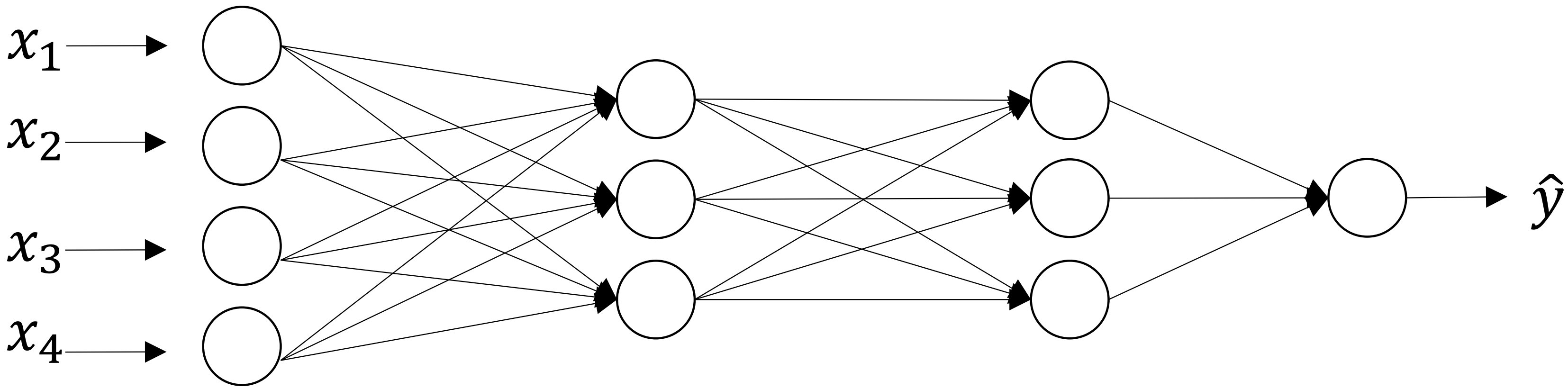
Content

1. Deep Artificial Neural Networks
2. Convolutional Neural Networks
3. Sequence Models
4. Generative Models
5. DL in Healthcare
6. DL bias and fairness

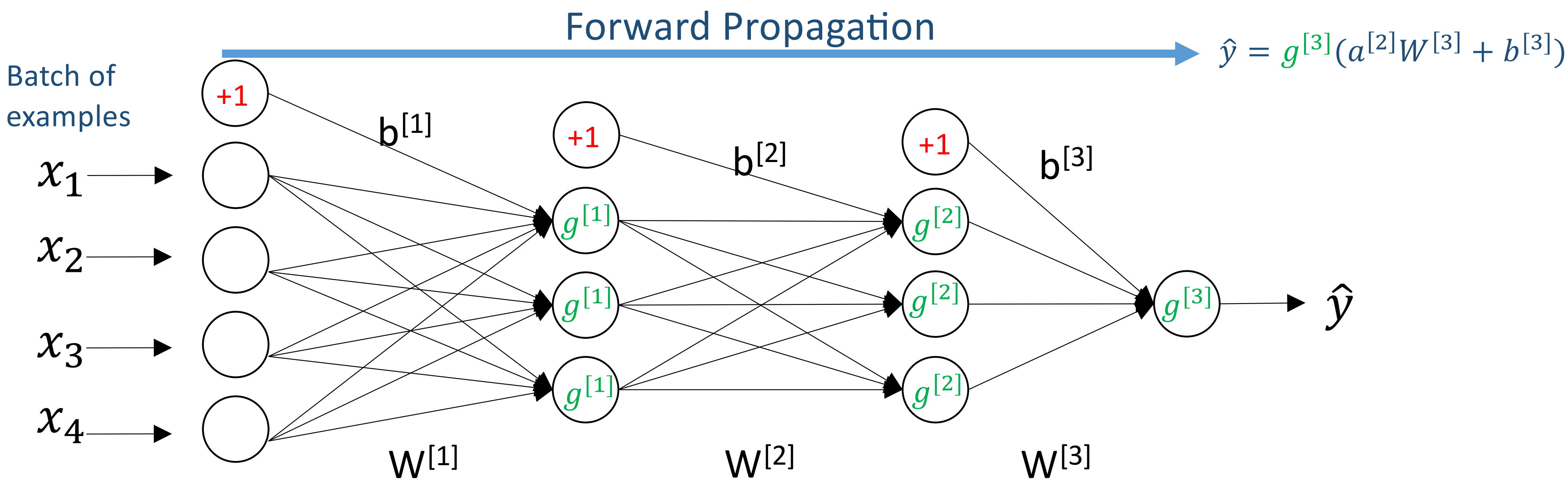
Content

1. Deep Artificial Neural Networks
    1. Architecture
    2. Activation Functions
    3. Loss Functions
    4. Optimizers
    5. Overfitting
    6. Vanishing/exploding Gradient
    7. Speedup Training
    8. Covariate Shift

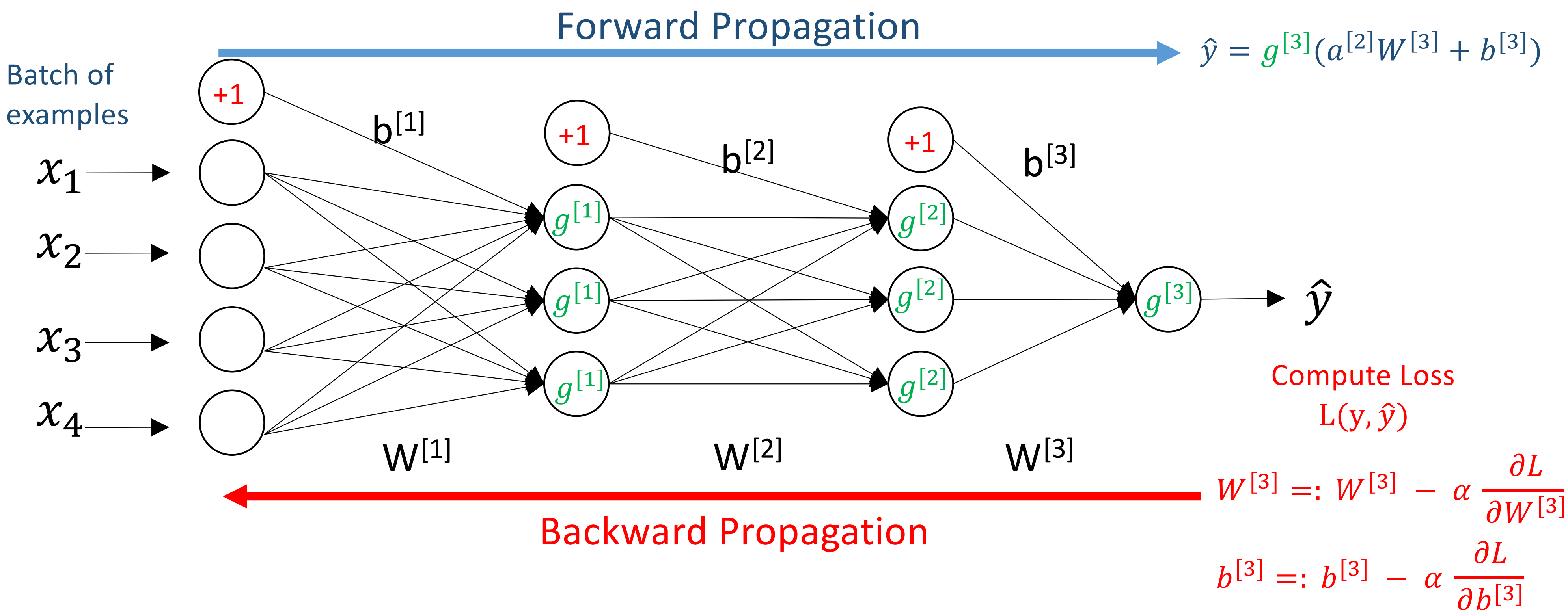
Artificial Neural Networks



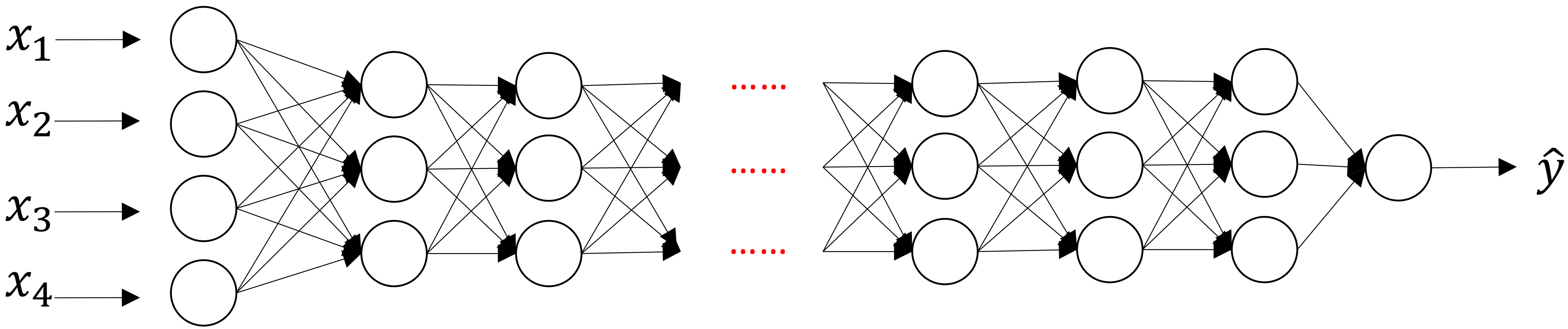
Artificial Neural Networks



Artificial Neural Networks

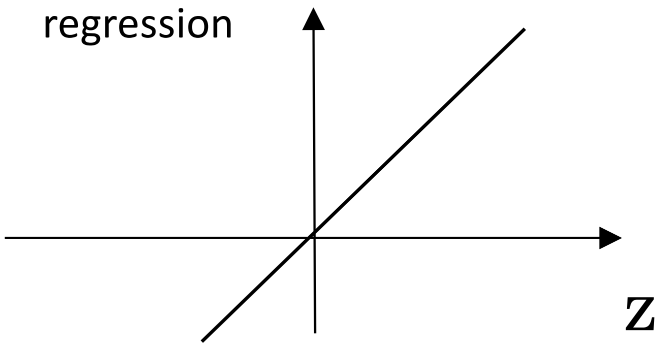


Deep Neural Networks

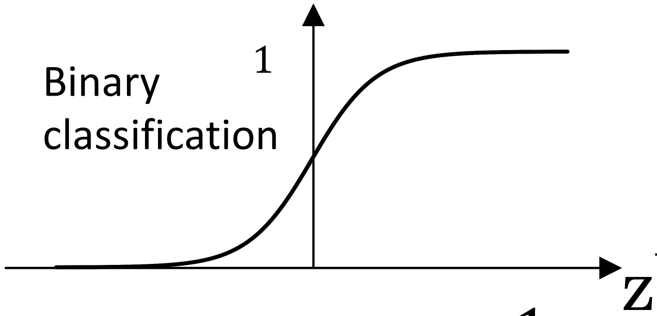


Activation Functions

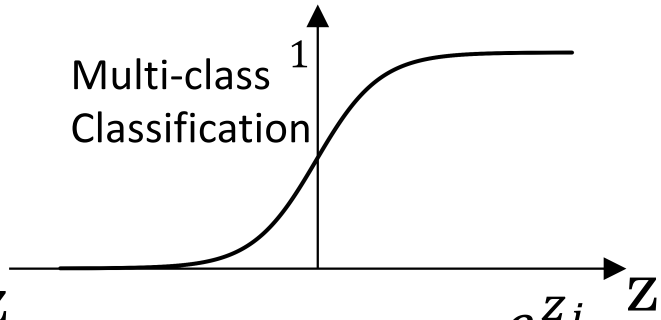
Output Layer



Linear $a = z$

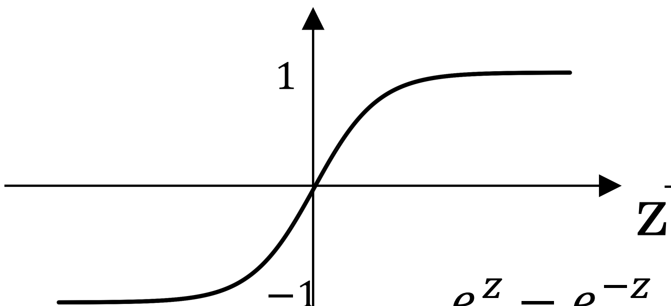


Sigmoid $a = \frac{1}{1 + e^{-z}}$

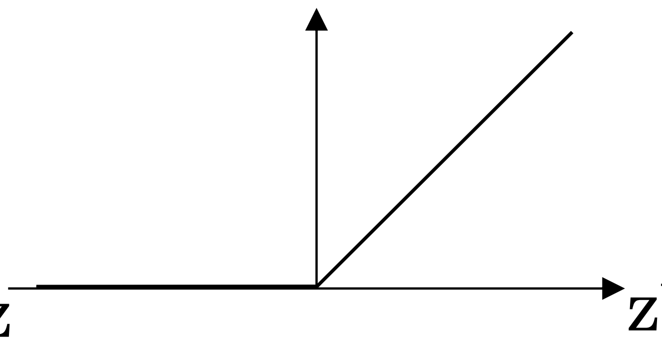


Softmax $a_j = \frac{e^{z_j}}{\sum_{j=1}^{K} e^{z_j}}$

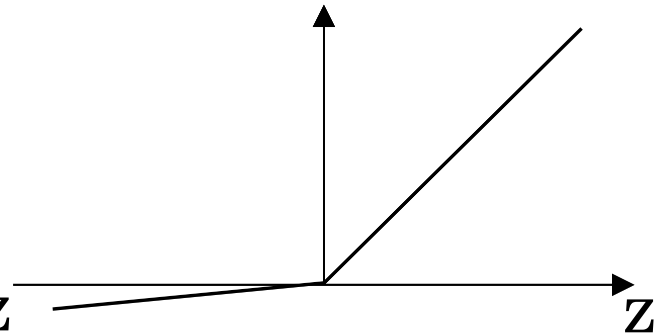
Hidden Layers



Tanh $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

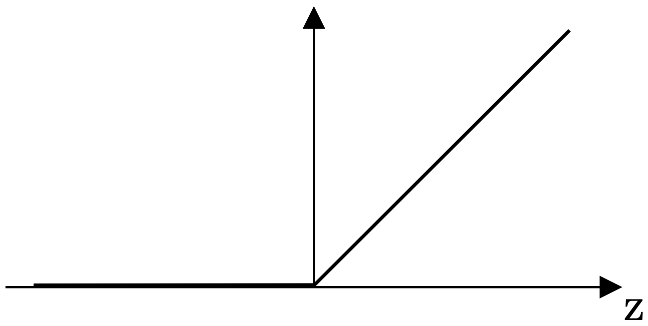


ReLU $a = \max(0, z)$



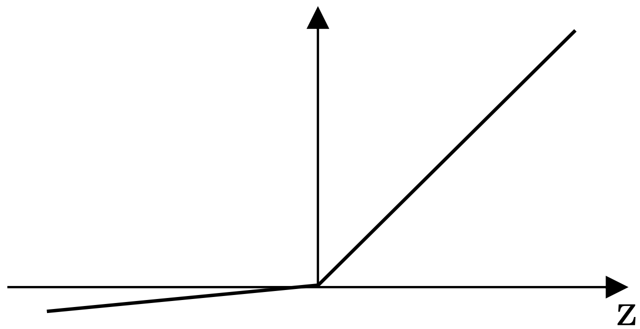
Leaky ReLU $a = \max(0.01z, z)$

Activation Functions



ReLU $a = \max(0, z)$

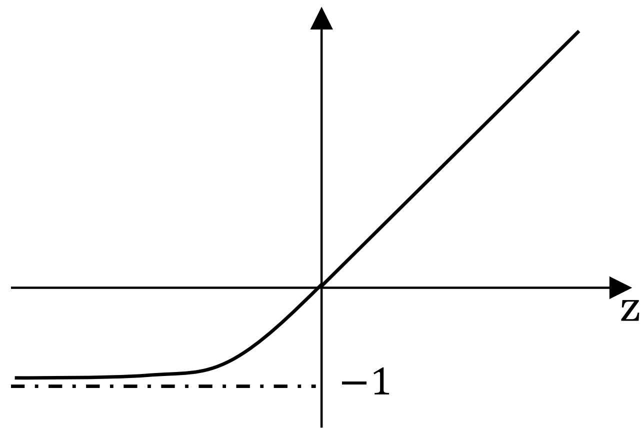
2010: Glorot and Bengio



Leaky ReLU $a = \max(\alpha z, z)$

$\alpha = 0.01$
$\alpha = 0.2$

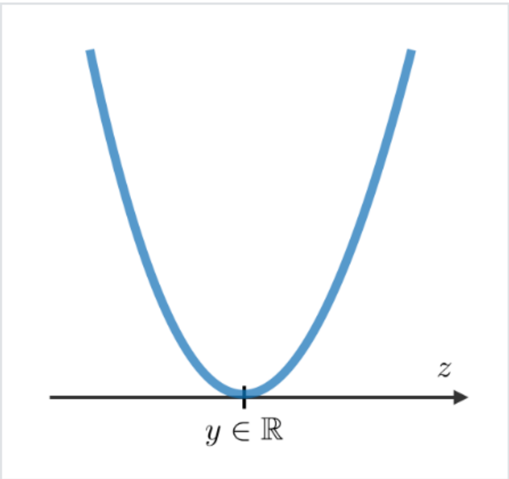
α can be learned: Parametric ReLU



Exponential ELU $a = \begin{cases} \alpha \exp(z - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$

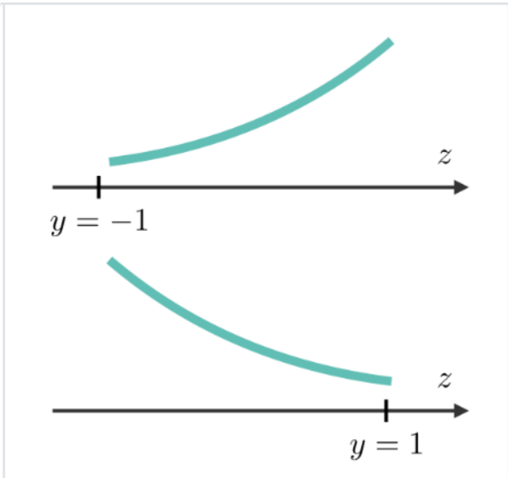
2015: Clevert et al.

Loss functions



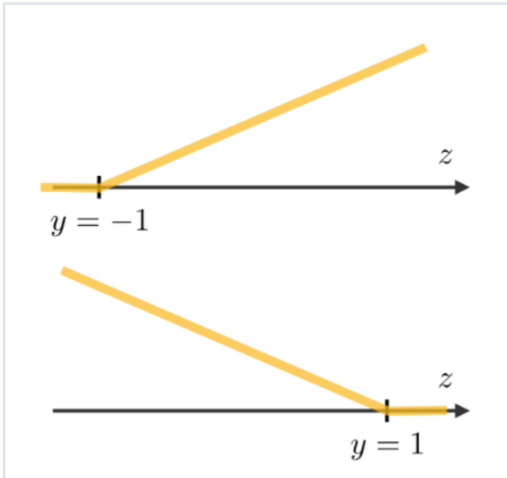
$L_i(\hat{y}, y) = (\hat{y} - y)^2$

Least squares loss



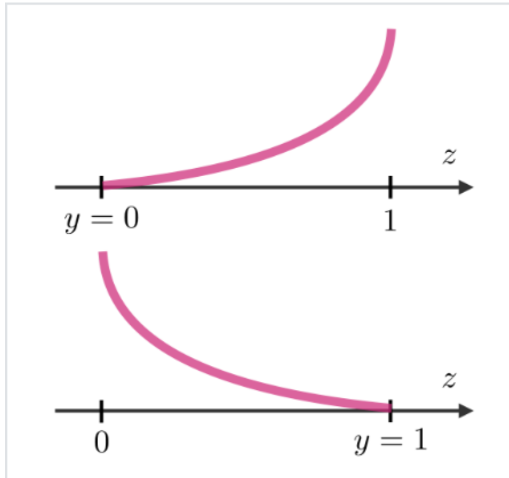
$L_i(\hat{y}, y) = \log(1 + \exp(-\hat{y}\, y))$

Logistic loss



$L_i(\hat{y}, y) = (1 - \hat{y}\, y)_+$

Hinge loss

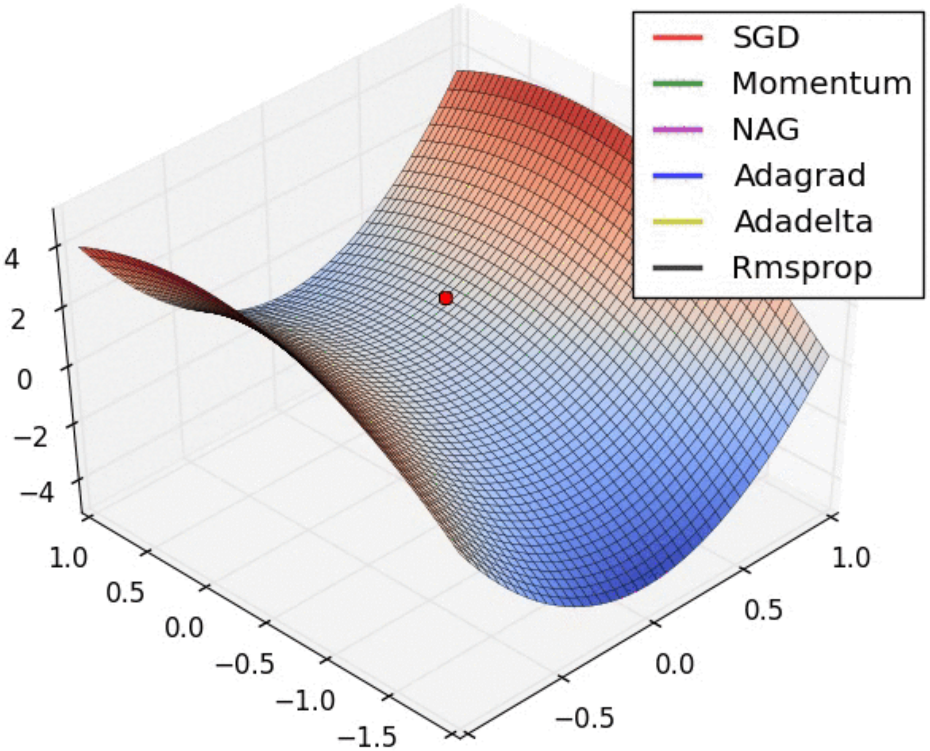


$L_i(\hat{y}, y) = -(y\log(\hat{y}) + (1 - y)\log(1 - \hat{y}))$

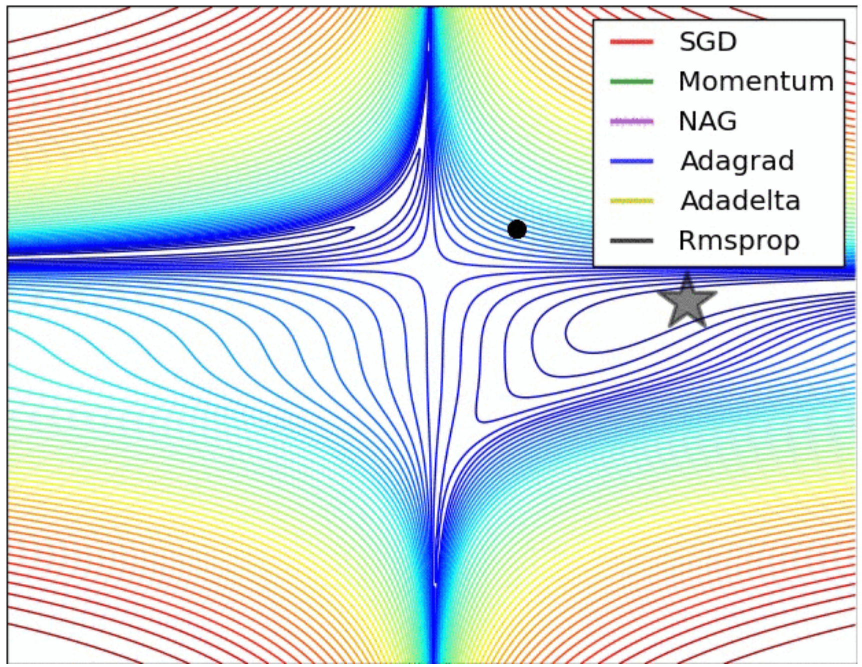
Cross Entropy loss

With K classes: $L_i(\hat{y}, y) = -\sum_{k=1}^{K} y_k \log(\hat{y}_k)$

Optimizers



SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5



SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

Update_parameters

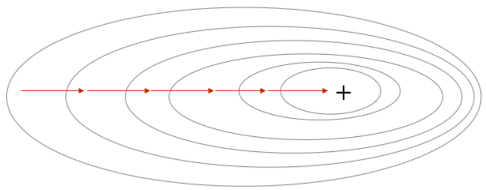
$$W := W - \alpha \, dW$$

Gradient Descent

Gradient Descent (All examples)

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for epoch in range(num_epochs):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost = compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)
```

Converge smoothly but slowly



Update_parameters

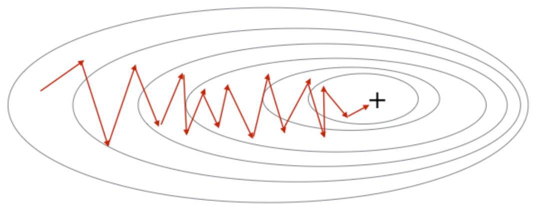
$$W := W - \alpha\, dW$$

Gradient Descent

Stochastic Gadient Descent (one single example each time)

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for epoch in range(num_epochs):
    for i in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,i], parameters)
        # Compute cost
        cost = compute_cost(a, Y[:,i])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

Converge quickly but oscillate



NB: Number of **epochs** is a hyperparameter that defines how many times we go through the **ENTIRE** training dataset.

Update_parameters

$W := W - \alpha \, dW$

Gradient Descent

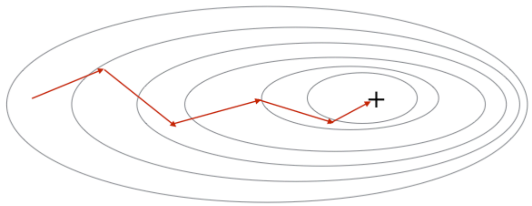
Mini-Batch Gadient Descent (a subset of examples each time)

```
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for epoch in range(num_epochs):
    mini_batches = random_mini_batches(X, Y, mini_batch_size)
    for mini_batch in mini_batches:
        # Forward propagation
        a, caches = forward_propagation(mini_batch[ 'X' ], parameters)
        # Compute cost
        cost = compute_cost(a, mini_batch[ 'Y' ])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

Shuffling and Partitioning

e.g., 16, 32, 64, 128

Converge quickly with few oscillations

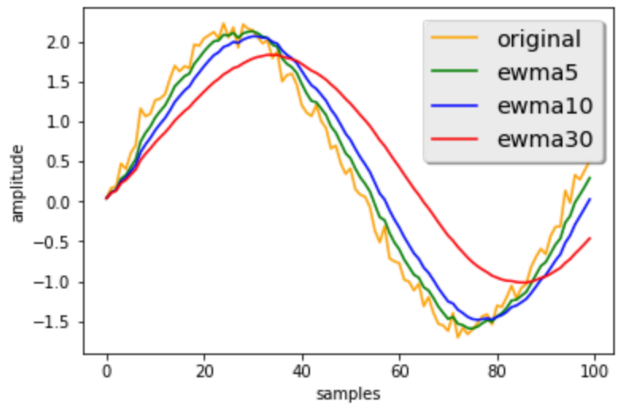


GD with Momentum

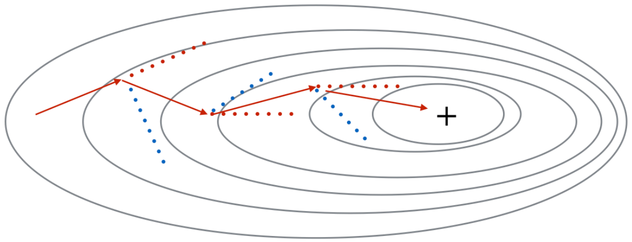
Update_parameters

$$v_{dW} := \beta v_{dW} + (1-\beta)dW$$
$$W := W - \alpha v_{dW}$$

- Generalization of mini-batch gradient descent ($\beta = 0$)
- Use EWMA (Exponentially Weighted Moving Averages) of dW and db.
- v_{dW} (velocity), dW (acceleration)
- Tuning hyperparameters α, β. Often ($\beta = 0.9$). Can be tuned using cross-validation.
- The larger the momentum β is, the smoother the update
- Nesterov Adaptive Gradient (NAG)
  - $v_{dW} := \beta v_{dW} + (1-\beta)d(W + \eta\, v_{dW})$
  - $W := W - \alpha v_{dW}$



Momentum



Reduce oscillations

Update_parameters

$$s_{dW} := \beta s_{dW} + (1-\beta)dW^2$$

$$W := W - \alpha \frac{dW}{\sqrt{s_{dW}} + \varepsilon}$$

GD with RMSprop

- RMSprop: Root Mean Square Propagation
- Uses Exponentially weighted averages of the second derivatives dW^2 (and db^2)
- ε avoid dividing by 0

GD with Adam

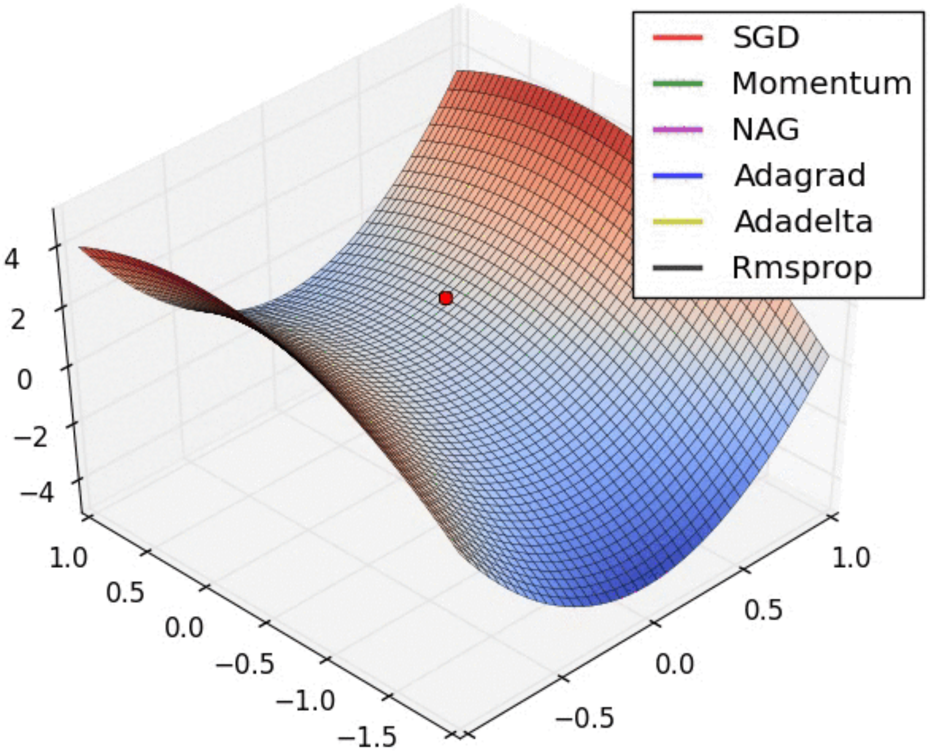
- Adam: Adaptive Moment Estimation
- Momentum + RMSprop
- Converges a lot faster and require low memory
- Usually works well even with little tuning of hyperparameters (except α)
- The most effective for DNNs !
- Variants
  - Nesterove Accelerated Gradient (NAG)
  - Adamax (infinity norm)
  - Nadam (Nesterov + RMSprop)

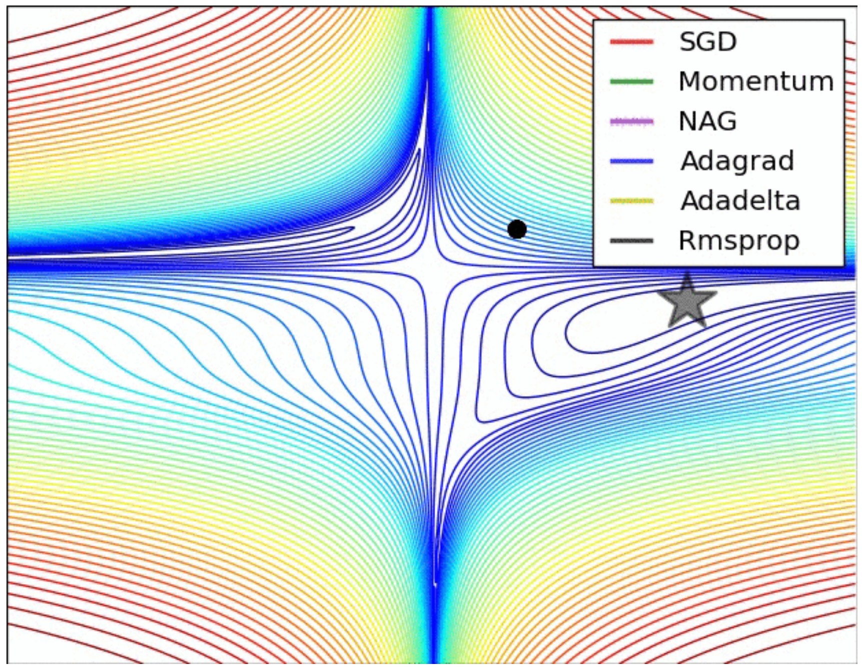
Update_parameters

$$v_{dW} := \beta_1 v_{dW} + (1-\beta_1)dW$$
$$v_{dW}^{corrected} = \frac{v_{dW}}{1-\beta_1^t}$$
$$s_{dW} := \beta_2 s_{dW} + (1-\beta_2)dW^2$$
$$s_{dW}^{corrected} = \frac{s_{dW}}{1-\beta_2^t}$$

$$W := W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \varepsilon}$$

Other Optimization Methods

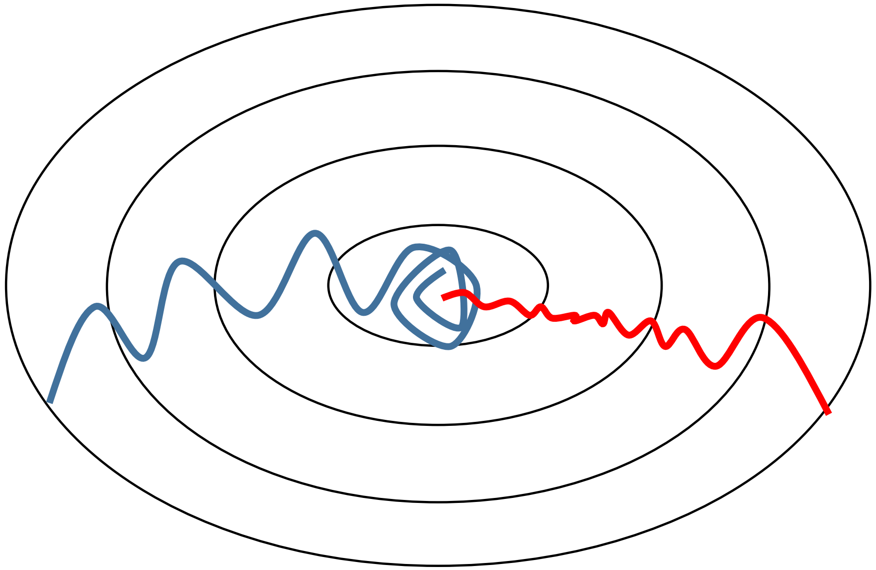




Learning Rate Decay

- Optimization with Constant learning rate may diverge

  → Reduce the learning rate every iteration
- Methods
  - Time based decay
  - Step decay
  - Exponential decay
  - Etc.



Learning Rate Decay

$\alpha = \alpha_0$

Fig 1 : Constant Learning Rate

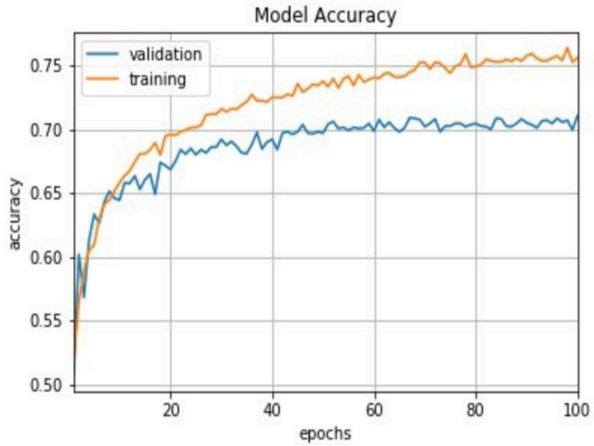


Fig 2 : Time-based Decay Schedule

$\alpha_t = \dfrac{1}{1 + kt} \alpha_0$

$\alpha_{t+s} = .5\ \alpha_t$

Reduce by half every step s

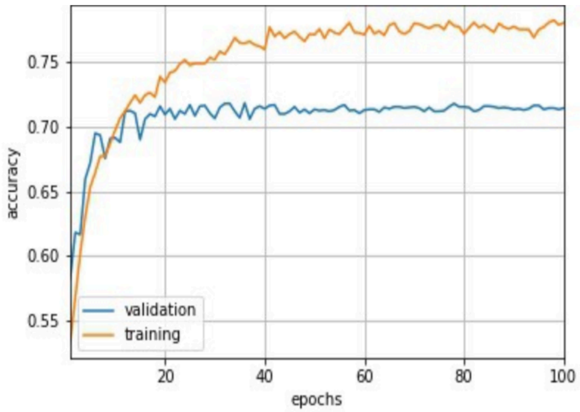


Fig 3a : Step Decay Schedule

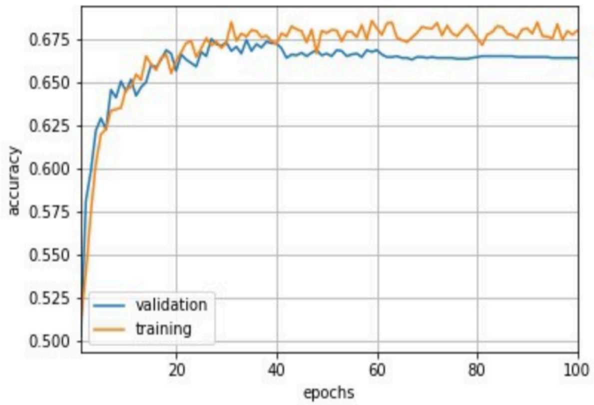
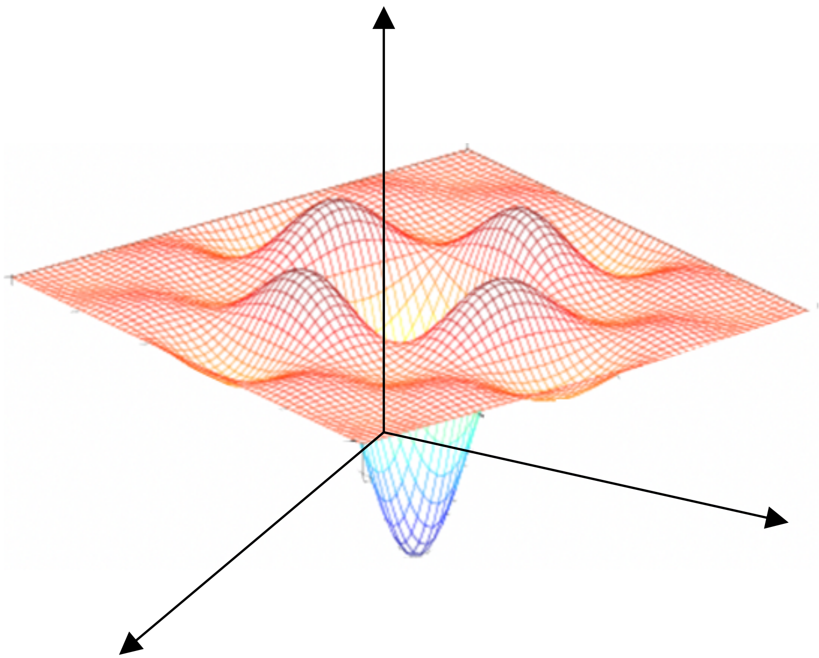
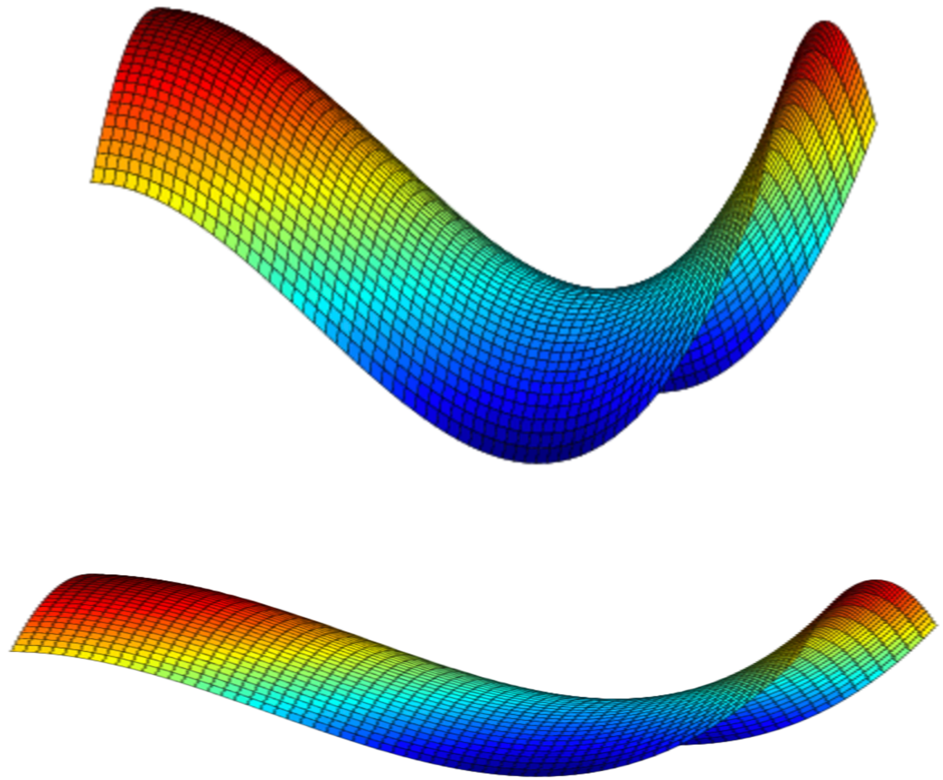


Fig 4a : Exponential Decay Schedule

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

Local Optima



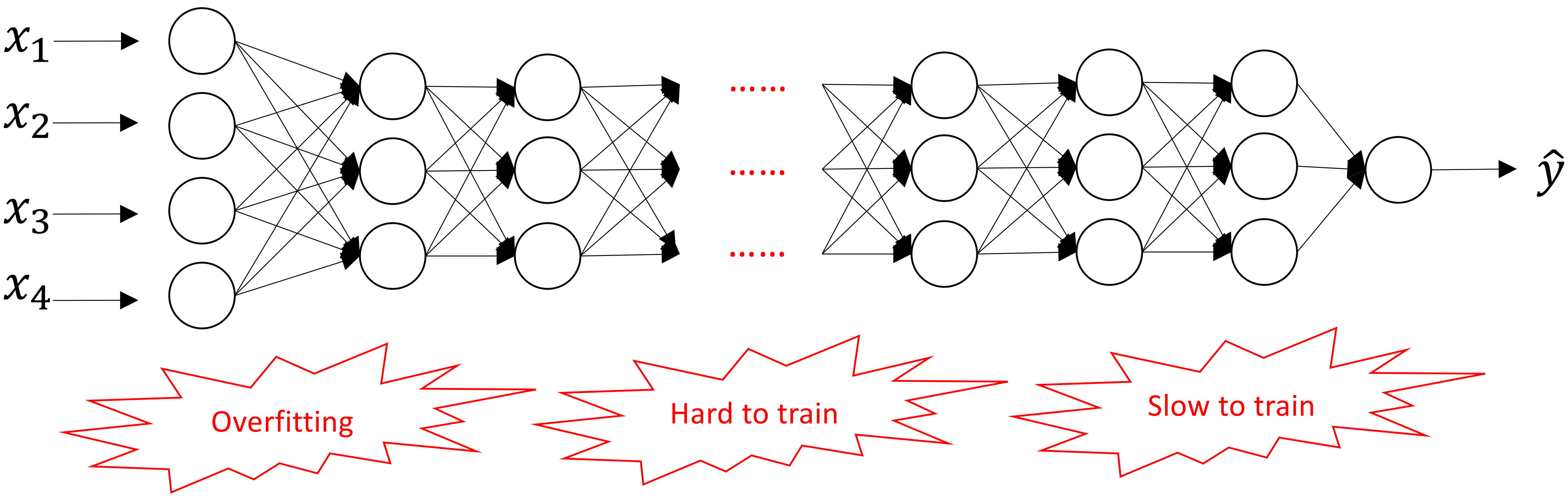


Plateaus can make learning slow

Hyper-parameter Tuning

- Hyperparameters
  - $\alpha, \lambda, \beta, \beta_1, \beta_2, \varepsilon, L, s_l$, minibatch size, epochs, etc.
- Using an appropriate scale to pick hyperparameters
- Random Search (Grid search is very costly)

Deep Neural Networks



A lot of parameters

Gradient vanishing
Gradient explosion

Deeper network
Covariate Shift

Reduce Overfitting

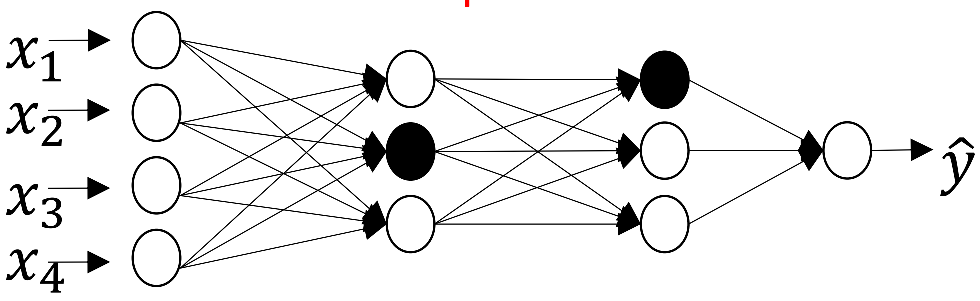
Regularization

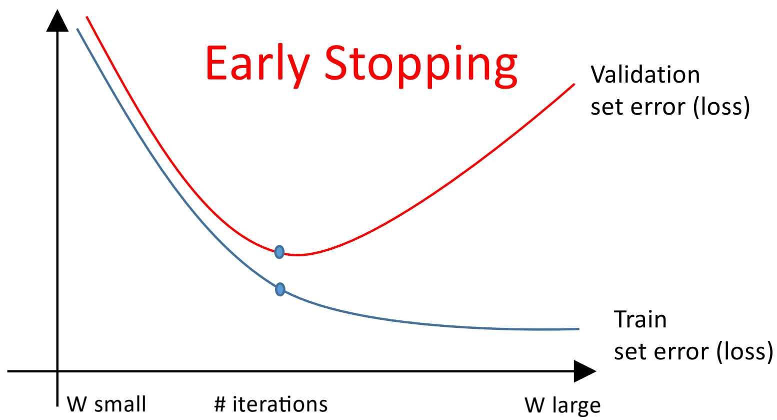
$$L(w) + \lambda\, w w^T$$

Data Augmentation

Big Data

Dropout





Vanishing/Exploding gradients

- $W^{[l]} =: W^{[l]} - \alpha \frac{\partial L}{\partial W^{[l]}}$
- $W^{[l]}$ <1 → $\frac{\partial L}{\partial W^{[l]}}$ <1→ Vanishing → slow down training
- $W^{[l]}$ >1 → $\frac{\partial L}{\partial W^{[l]}}$ >1→ Exploding → divergence

- Solution
  - Batch normalization
  - Random Weights Initialization

Addressing Vanishing/exploding Gradient

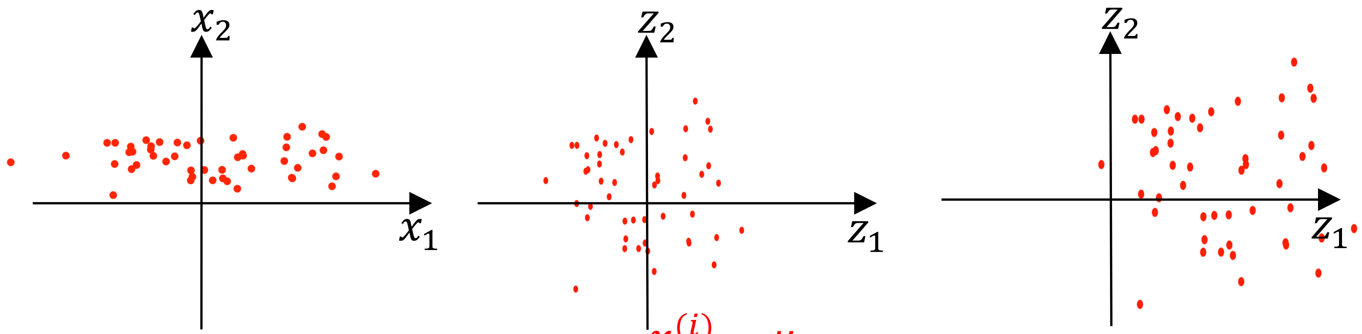
Random weights Initialization

Relu: $W^{[l]}$ = randn$(l-1, l)$ * $\sqrt{\frac{2}{n^{[l-1]}}}$ (He Initialization)

Tanh: $W^{[l]}$ = randn$(l-1, l)$ * $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$ (Xavier Glorot Initialization)

Addressing Vanishing/exploding Gradient and Speedup Training

Batch Normalization



μ_B, σ_B

Mean and variance of the Batch

$$\tilde{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B{}^2 + \epsilon}}$$

smoothing parameter

$$z^{(i)} = \gamma\tilde{x}^{(i)} + \beta$$

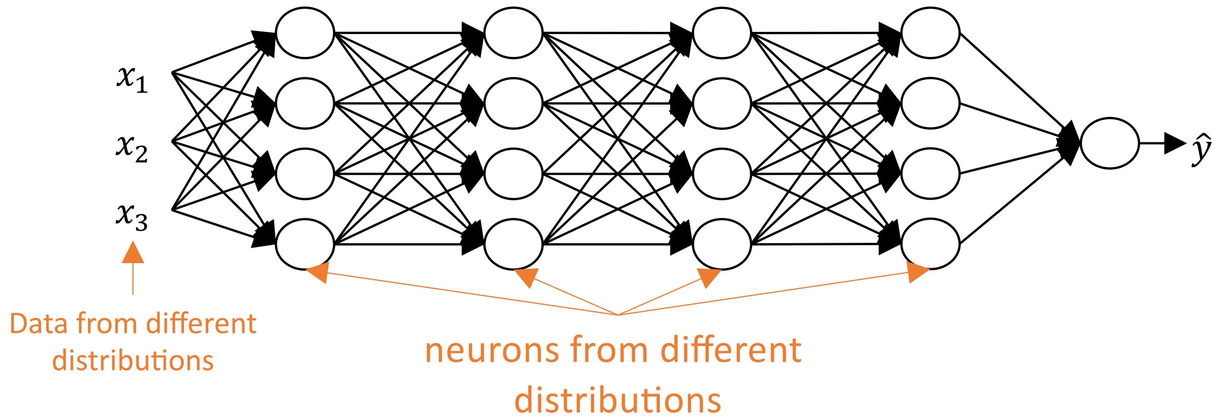
scaling parameter

shifting parameter

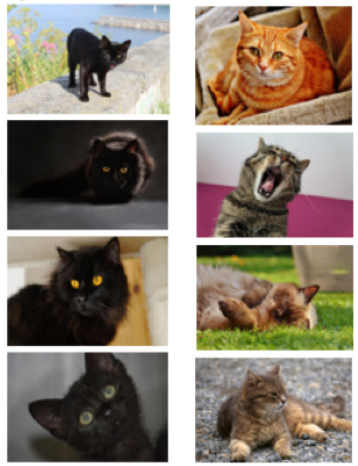
- Other methods
  - Synthetic gradients
  - Gradient Clipping

Problem of Covariate Shift

- **Definition**
  - different distributions in the data or from layer to layer !
  - The input data and the neurons of the hidden layers could be considered as coming from different distributions!
- **Solution**: Batch Normalization to normalize the hidden layers!



Cats from different distributions



Batch Normalization

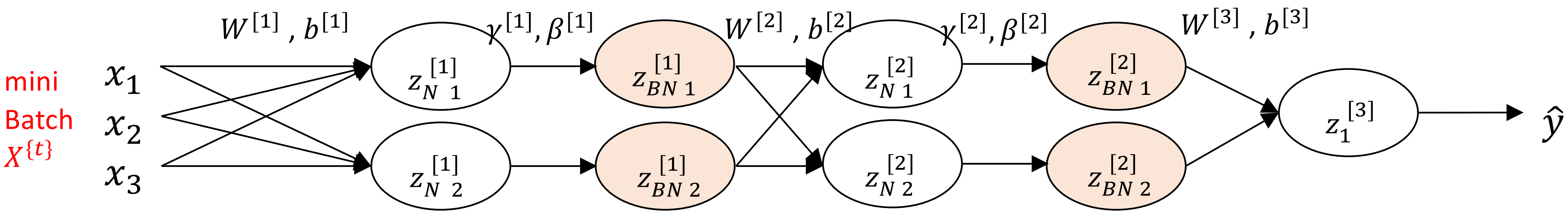
1. Pick a mini batch $X^{\{b\}}$ $(b \in 1..B)$ from X, compute $\mu^{\{b\}}$ and $\sigma^{\{b\}}$ and perform normalization: $x_N^{(i)} = \frac{(x^{(i)} - \mu^{\{b\}})}{\sqrt{\sigma^{\{b\}2} + \varepsilon}}$

Avoid dividing by zero

2. Normalize the activations $A^{[l]}$ (or logits $Z^{[l]}$) in each layer l : $z_N^{(i)[l]} = \frac{(z^{(i)[l]} - \mu^{\{b\}[l]})}{\sqrt{\sigma^{\{b\}[l]2} + \varepsilon}}$

3. Rescale: $z_{BN}^{(i)[l]} = \gamma^{[l]} z_N^{(i)[l]} + \beta^{[l]}$, with $\gamma^{[l]}$ and $\beta^{[l]}$ are learnable parameters like $W^{[l]}$ and $b^{[l]}$

- Note that if $\gamma^{[l]} = \sqrt{\sigma^{\{b\}[l]2} + \varepsilon}$ and $\beta^{[l]} = \mu^{\{b\}[l]}$, then $z_{BN}^{(i)[l]} = z_N^{(i)[l]}$ (no batch norm effect)
- At test time: $\mu^{\{test\}[l]}$ and $\sigma^{\{test\}[l]}$ are estimated using EWMA of all $\mu^{\{b\}[l]}$ and $\sigma^{\{b\}[l]}$ respectively.



Tune Hyper-parameters

- $\alpha, L, s_l,$ mini-batch size, epochs, etc.
- $\beta, \beta_1, \beta_2, \varepsilon,$ etc. (optimization hyper-parameters)
- Etc.

Deep learning Tools

- Python / Julia
- Anaconda
- VSCode
- Jupyter Notebook
- Github
- Etc,

- TensorFlow,
- Keras
- PyTorch
- Caffe/Caffe2,
- CNTK,
- DL4J,
- Lasagne,
- MxNet,
- Etc,

- Apache Spark Mllib
- Amazon ML (AML)
- Google Cloud ML Engine
- Google ML Kit for Mobile
- Apple’s Core ML
- Etc,