NYU

Machine Learning

Training Deep Neural Networks

Overview:

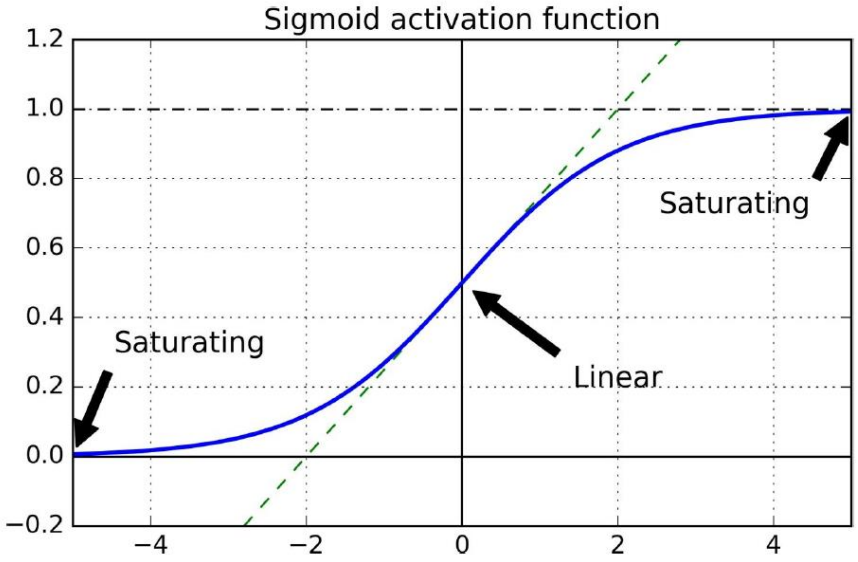
- Vanishing/Exploding Gradients Problem
  - Parameter Initialization
  - Non-saturating activation function
- Batch Normalization
- Fast Optimizers
- Overfitting and Generalization Performance
  - Weight Regularization
  - Dropout
  - Noise on Input tensors
- Hyper-parameter Tuning

Problems:

- Some tasks require very deep neural nets with tens of hidden layers containing hundreds of hidden units per layer (e.g. Krizhevsky, Sutskever, Hinton 2012).
- Vanishing and exploding gradients makes deeper layers very hard to train.
- Deep neural nets with millions of parameters would severely risk overfitting the training set.

Vanishing/Exploding Gradients

- Backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way.
- Algorithm has computes the gradient of the cost function with regards to each parameter in the network.
- Uses these gradients to update each parameter with a Gradient Descent step.
- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- Gradient Descent update leaves the lower layer connection weights virtually unchanged.


Sigmoid activation function
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−4
−2
0
2
4
Saturating
Saturating
Linear

Vanishing/Exploding Gradients

- Opposite to vanishing gradients: the gradients can grow bigger and bigger.
- Causing many layers to compute insanely large weight updates and the algorithm diverges.
- This is the exploding gradients problem, which is mostly encountered in recurrent neural networks.
- Sigmoid activation function and the random (gaussian) initialization scheme, cause the variance of the outputs of each layer to be much greater than the variance of its inputs.
- More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

Vanishing/Exploding Gradients

Solutions:

- Need the variance of the outputs of each layer to be equal to the variance of its inputs, and the gradients to have equal variance before and after flowing through a layer in the reverse direction
- Xavier and He Initialization.
- Using the Xavier initialization strategy can speed up training considerably, and it is one of the tricks that led to the current success of Deep Learning.
- Iinitialization strategy for the ReLU activation function is sometimes called He initialization.

- “Understanding the Difficulty of Training Deep Feedforward Neural Networks”, Glorot and Bengio, 2010.

- “Delving Deepinto Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, He et al., 2015.

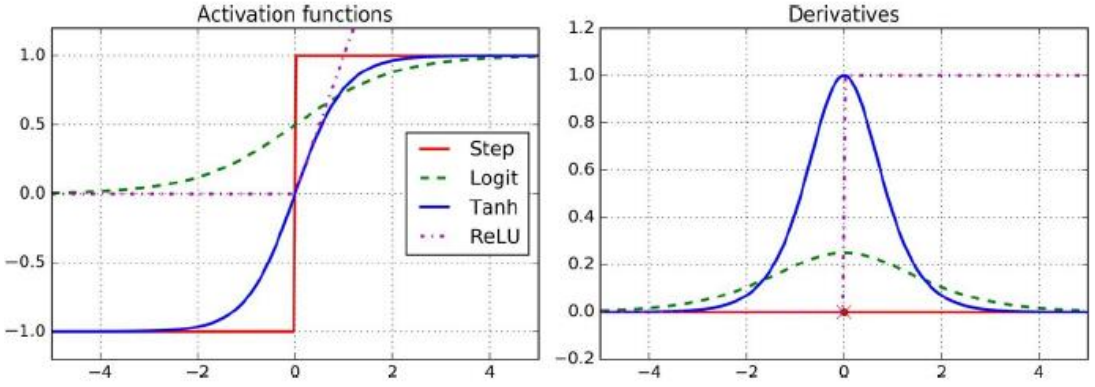
Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Vanishing/Exploding Gradients

Solutions:

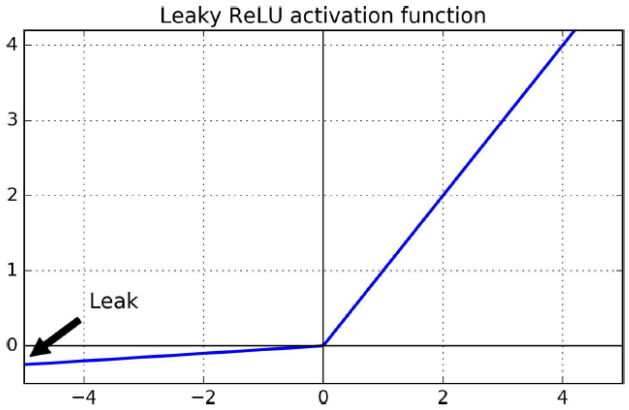
- Non-saturating activation function.
- Vanishing/exploding gradients problems were in part due to a poor choice of activation function.
- Other activation functions behave much better in deep neural networks, in particular the ReLU activation function.
- Does not saturate for positive values (and also because it is quite fast to compute).

Vanishing/Exploding Gradients



Leaky ReLU:

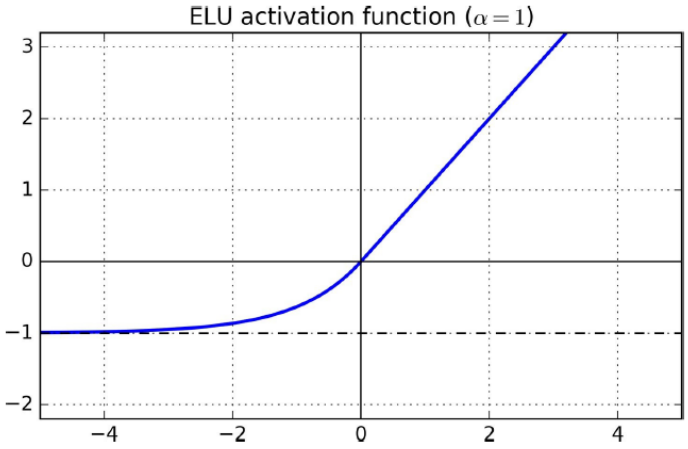
- During training, some neurons effectively die, meaning they stop outputting anything other than 0.
- If a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0.
- May want to use a variant of the ReLU function, such as the leaky ReLU.
- Leaky variants always outperformed the strict ReLU activation function.



$$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$$

Exponential Linear Unit (ELU):

- Outperformed all the ReLU variants in their experiments.
- Also has a nonzero gradient for z < 0, which avoids the dying units issue.
- ELU function is smooth everywhere, including around z = 0.
- Drawback of the ELU activation function is that it is slower to compute due to the use of the exponential function.



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

Batch Normalization (BN)

Batch Normalization (BN):

- He initialization along with ELU (or any variant of ReLU) can significantly reduce the vanishing/exploding gradients problems.
- Doesn’t guarantee that they won’t come back during training.
- Covariate shift problem - distribution of each layer’s inputs changes during training, as the parameters of the previous layers change.
- BN technique consists of adding an operation before activation of each layer.
- Zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters.

Batch Normalization

1. $\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^{\,2} = \frac{1}{m_B}\sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^{\,2} + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$

- μ_B is the empirical mean, evaluated over the whole mini-batch B.
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.
- $\hat{\mathbf{x}}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

Batch Normalization at Test Time:

- There is no mini-batch to compute the empirical mean and standard deviation.
- Instead we use the whole training set's mean and standard deviation.
- Typically efficiently computed during training using a moving average.
- Four parameters are learned for each batch normalized layer:

γ (scale), β (offset), μ (mean), and σ (standard deviation).

$$\hat{v} \leftarrow \hat{v} \times \text{decay} + v \times (1 - \text{decay})$$

Fast Optimizers:

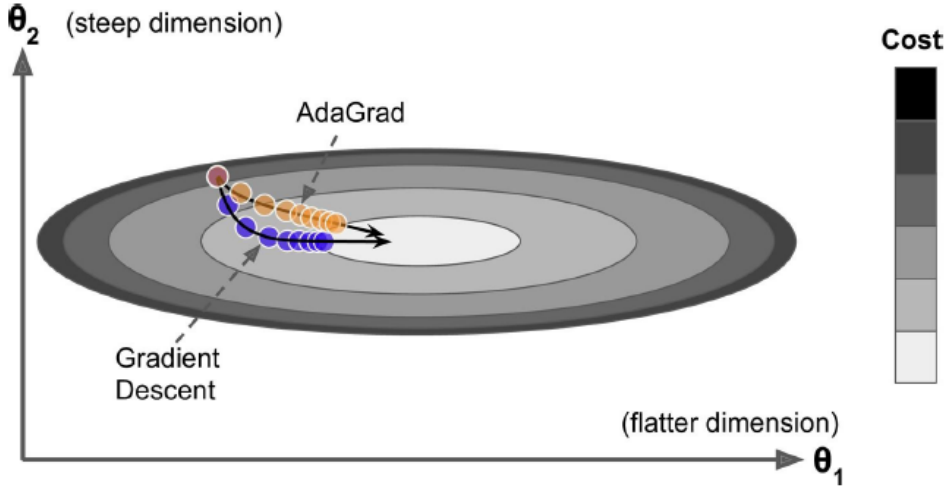
- Training a very large deep neural network can be painfully slow.
- Four ways to speed up training:
  - Good initialization strategy for the connection weights,
  - Good activation function,
  - Batch Normalization,
  - Mini-batch gradient descent.
- Speed booster also comes from using a faster optimizer:
  - Momentum optimization,
  - Nesterov Accelerated Gradient,
  - AdaGrad,
  - RMSProp,
  - Adam optimization*.

Momentum Optimization:

1. $\mathbf{m} \leftarrow \beta \mathbf{m} + \eta \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \mathbf{m}$

RMS Prop Optimization:

1. $\boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1 - \beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\boldsymbol{s} + \epsilon}$


θ2 (steep dimension)
AdaGrad
Gradient Descent
(flatter dimension)
θ1
Cost

Fast Optimizer - ADAM

- Adaptive Moment (ADAM) algo combines Momentum and RMS prop.
- The ADAM algorithm have been shown to work well across a wide range of deep learning architectures.
- Cost contours: ADAM damps out oscillations in gradients that prevents the use of large learning rate.
  - Momentum: speed ups training in horizontal direction.
  - RMS Prop: Slow down learning in vertical direction.
- ADAM is appropriate for noisy financial data.
- Kingma and Ba., 2015. ADAM: A Method For Stochastic Optimization.

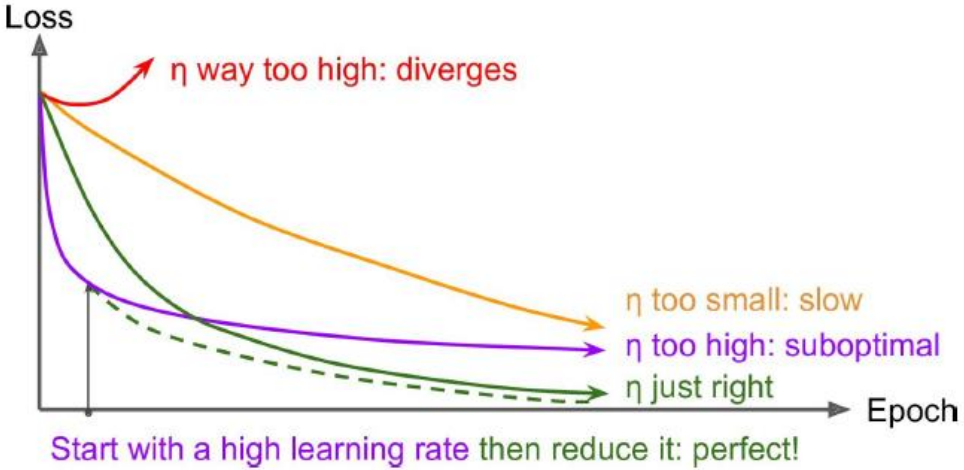
Fast Optimizer - ADAM

Algorithm 1 ADAM Optimization Algorithm [26]. dw^2 denotes elementwise square $dw \odot dw$. β_1^t and β_2^t denotes β_1 and β_2 to the power of t. Default values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Require: α: Learning rate
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $\epsilon : 10^{-8}$
Require: $f(\Theta)$: Cross-entropy objective function in chapter 4 equation 4.5
Require: Θ: Initial parameter vector W and b

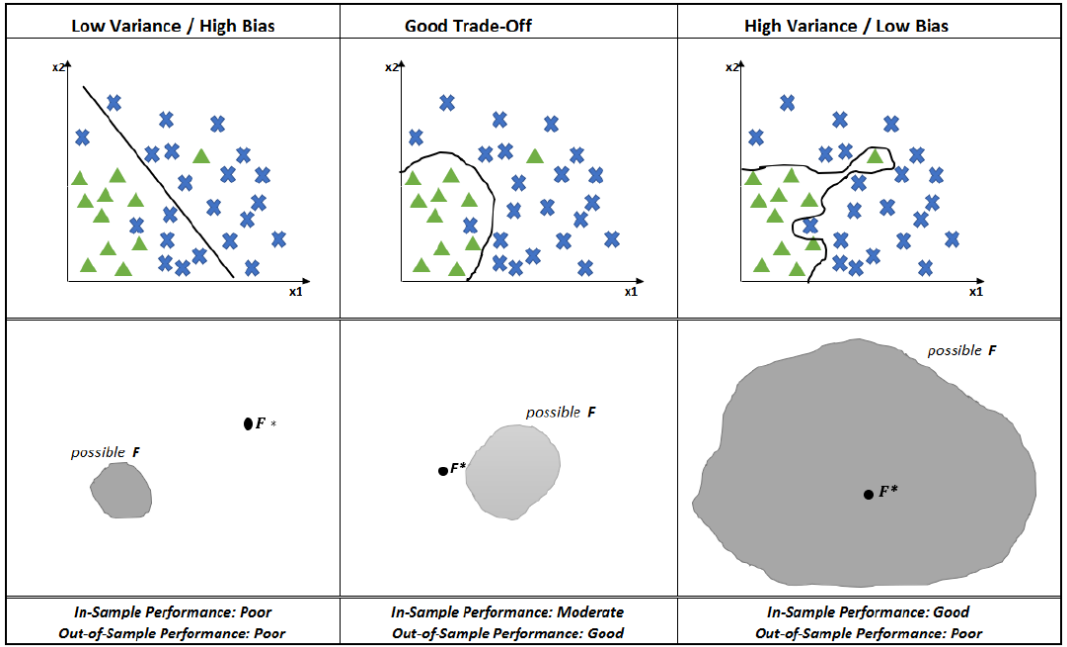
$V_{dw} \leftarrow 0$ (Initialize 1^{st} moment vector)
$V_{db} \leftarrow 0$ (Initialize 1^{st} moment vector)
$S_{dw} \leftarrow 0$ (Initialize 2^{nd} moment vector)
$S_{db} \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while Θ not converged on iteration t **do**
  $dw, db \leftarrow \nabla_\Theta f(\Theta)$ (Compute dw, db w.r.t objective function on iteration t)
  $V_{dw} \leftarrow \beta_1 \cdot V_{dw} + (1 - \beta_1) \cdot dw$ (Update biased first moment estimate)
  $V_{db} \leftarrow \beta_1 \cdot V_{db} + (1 - \beta_1) \cdot db$
  $S_{dw} \leftarrow \beta_1 \cdot S_{dw^2} + (1 - \beta_2) \cdot dw^2$ (Update biased second moment estimate)
  $S_{db} \leftarrow \beta_1 \cdot S_{db^2} + (1 - \beta_2) \cdot db^2$
  $\hat{V}_{dw} \leftarrow V_{dw}/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
  $\hat{V}_{db} \leftarrow V_{db}/(1 - \beta_1^t)$
  $\hat{S}_{dw} \leftarrow S_{dw}/(1 - \beta_2^t)$ (Compute bias-corrected second moment estimate)
  $\hat{S}_{db} \leftarrow S_{db}/(1 - \beta_2^t)$
  $W \leftarrow W - \alpha \cdot \hat{V}_{dw}/(\sqrt{\hat{S}_{dw}} + \epsilon)$ (Update parameters)
  $b \leftarrow b - \alpha \cdot \hat{V}_{db}/(\sqrt{\hat{S}_{db}} + \epsilon)$ (Update parameters)
  end while
  return Θ **(Resulting parameters)**

Learning Rate Scheduling



Bias-Variance Tradeoff

- Generalization error decomposes into bias and variance.
- Variance: does model vary for another training dataset.
- Bias: closeness of average model to the true model F*.

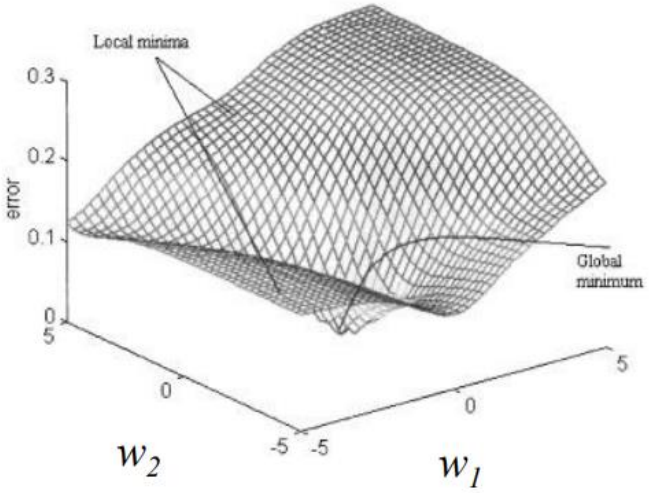


Overfitting and Generalization Power

- Add weight regularization term to objective function

$$\underset{\theta}{\text{argmin}} \ \frac{1}{m} \sum_{i=1}^{m} l(\mathbf{F}(\mathbf{x}^{(i)}; \theta), \boldsymbol{y}^{(i)}) + \lambda \Omega(\boldsymbol{\theta})$$

- *L1* and *L2* Regularization



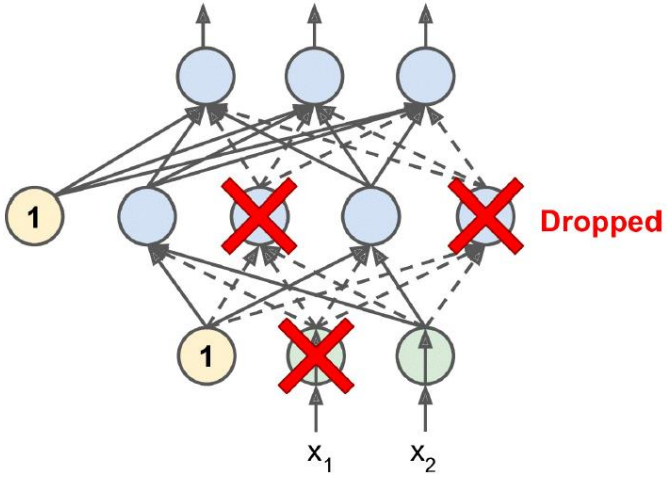
Cho & Chow, Neurocomputing 1999

- **Dropout is a simple algorithm:**

  - at every training step, every neuron has a probability *p* of being temporarily “dropped out”.

  - meaning it will be entirely ignored during this training step, but it may be active during the next step.

  - hyperparameter *p* is called the dropout rate.



Default Configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None