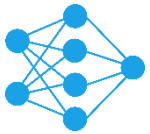
Deep Neural Networks

Hamidreza Baradaran Kashani

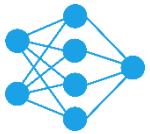
Deep Neural Networks



Objectives

- Diving into deep learning!
- Understanding the main challenges of deep neural network optimization and getting familiar with several popular optimizers in deep learning.
- Discussing the vanishing & exploding gradient problem and getting familiar with some solutions to avoid it.

List of Contents

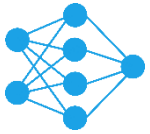


What is Deep Learning?

- Artificial intelligence allows machines to model and even improve upon the capabilities of the human mind.

- In general, we can say an intelligent system operates according to one of these paradigms:
  - Rule-based paradigm
  - Classical machine learning approach
  - Deep learning approach

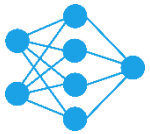
- The last two paradigms above also called data-driven methods.



What is Deep Learning?

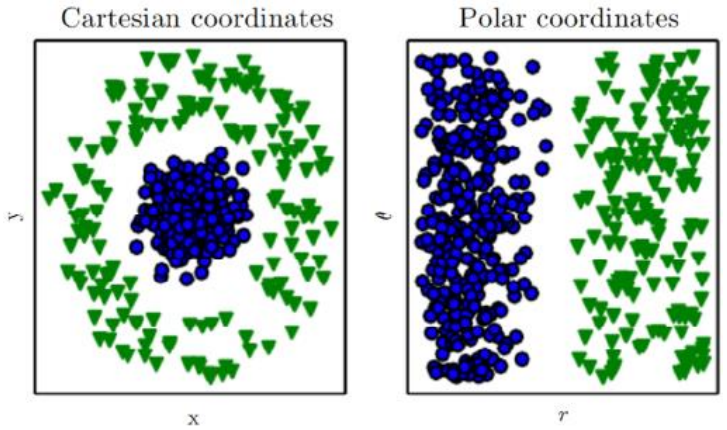
- **Rule-based systems:**
  - Requires hard-code knowledge in a formal language
  - It strongly depends on human knowledge about the field

- **Classic Machine Learning approach** (e.g. Logistic regression & Naïve bayes):
  - Extracts knowledge automatically from data
  - Allows tackling complex systems that their description is difficult for human
  - Limited in ability to process natural data in raw form
  - Requires careful engineering and domain expertise to transform raw data (e.g. pixel values) into a feature vector for model



What is Deep Learning?

- Traditional ML depends heavily on representation of the data.
  - For example: straight line separation
    - ✓ Impossible in Cartesian coordinates
    - ✓ Simple in Polar coordinates



❑ Designing right set of features:

- Difficult to know what set of features are good for detecting a car in photographs



- Which features should we consider to all these photos be classified as car?

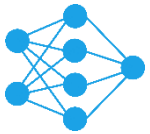
What is Deep Learning?

❑ Deep Learning is Representation Learning!

- Representation Learning is a kind of ML that does not learn mapping from representation to output, instead it learns from representation itself.

- It better results than hand-coded representations and allows AI systems to rapidly adapt to new tasks.

- Designing features can take great human effort, and even can take decades for a community of researchers.

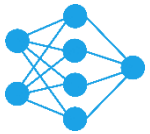


What is Deep Learning?

- But deep learning methods allow a machine to be fed with raw data to automatically discover representations needed for detection or classification

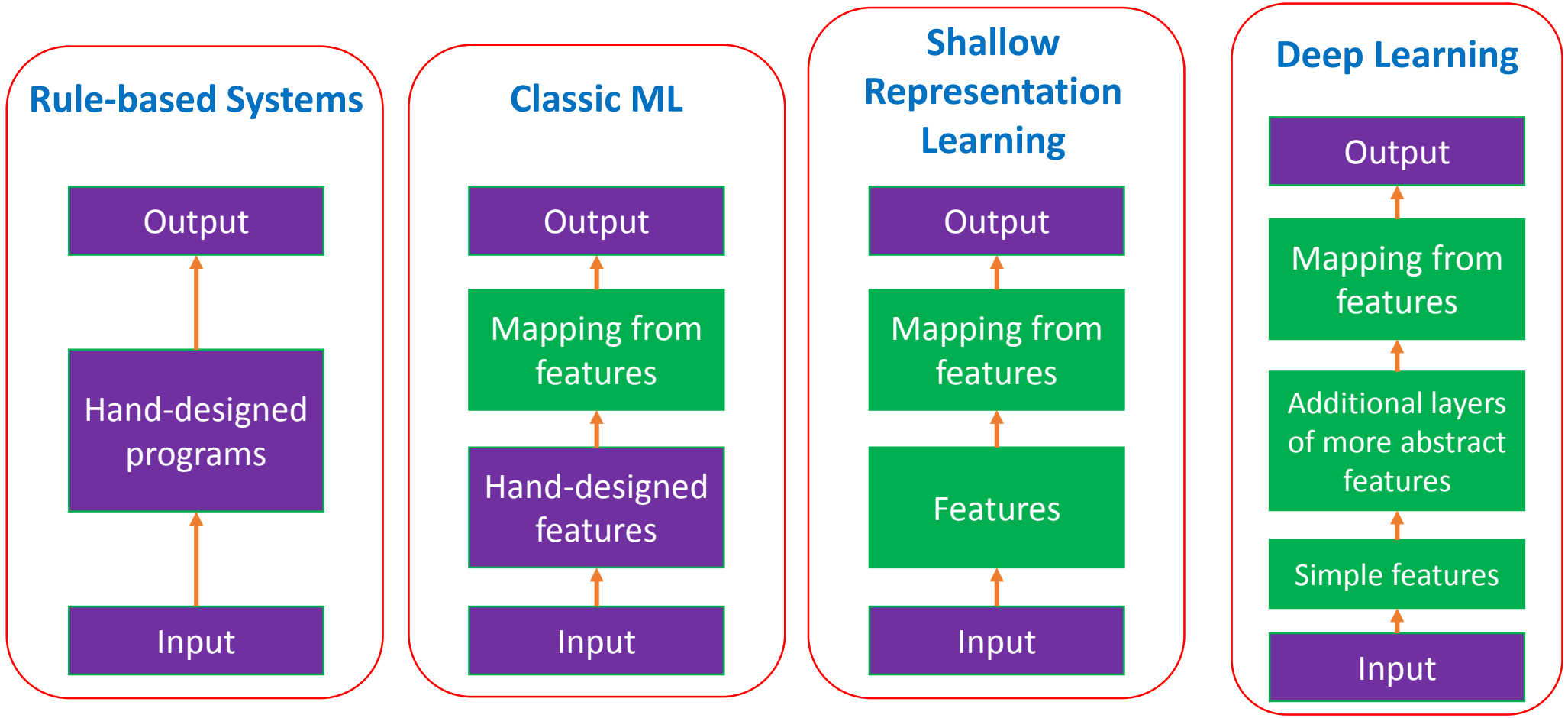
- Compose simple but non-linear modules that transform representation at one level (starting with raw input) into a representation at a higher slightly more abstract level.

- Complex functions can be learned. Higher layers of representation amplify aspects of input important for discrimination and suppress irrelevant variations.



What is Deep Learning?

- Paradigms of AI: (Green boxes indicate components that can learn from data)



What is Deep Learning?

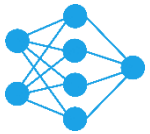
❑ **Characteristics of Deep Learning:**

- Deep learning is inspired by neural networks of the brain to build learning machines which discover rich and useful internal representations, computed as a composition of learned features and functions.

- A type of Machine Learning that improves with experience and data.

- Obtains its power by a nested hierarchy of concepts. each concept defined by relationship to simpler concepts

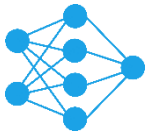
- More abstract representations computed in terms of less abstract ones



What is Deep Learning?

❑ **Example: Deep Learning for image classification**

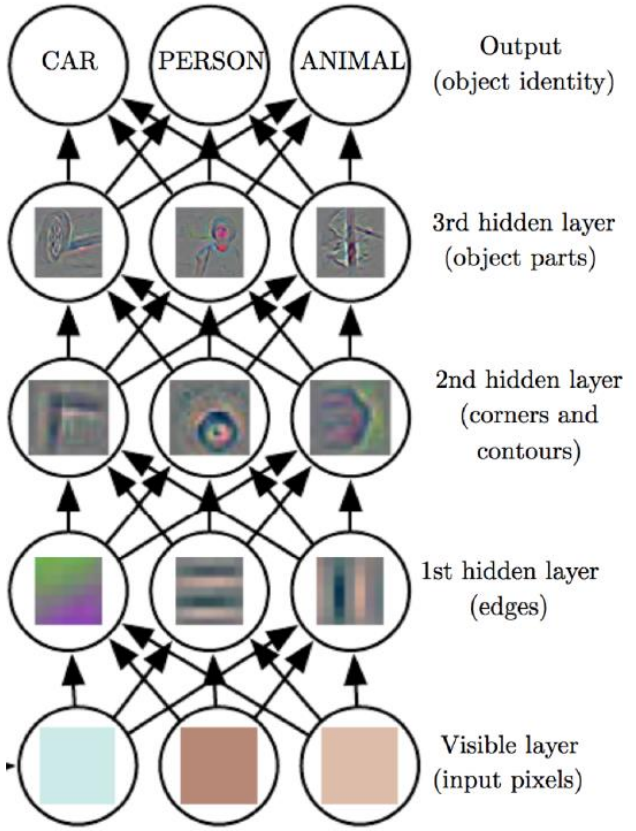
- Input: an array of pixel values
- First Layer: presence or absence of edges at particular locations and orientations of image
- Second layer: detect motifs (the main ideas) by spotting arrangements of edges, regardless of small variations in edge positions
- Third layer: assemble motifs into larger combinations that corresponds to parts of familiar objects
- Subsequent layers: detect objects as combinations of these parts



What is Deep Learning?

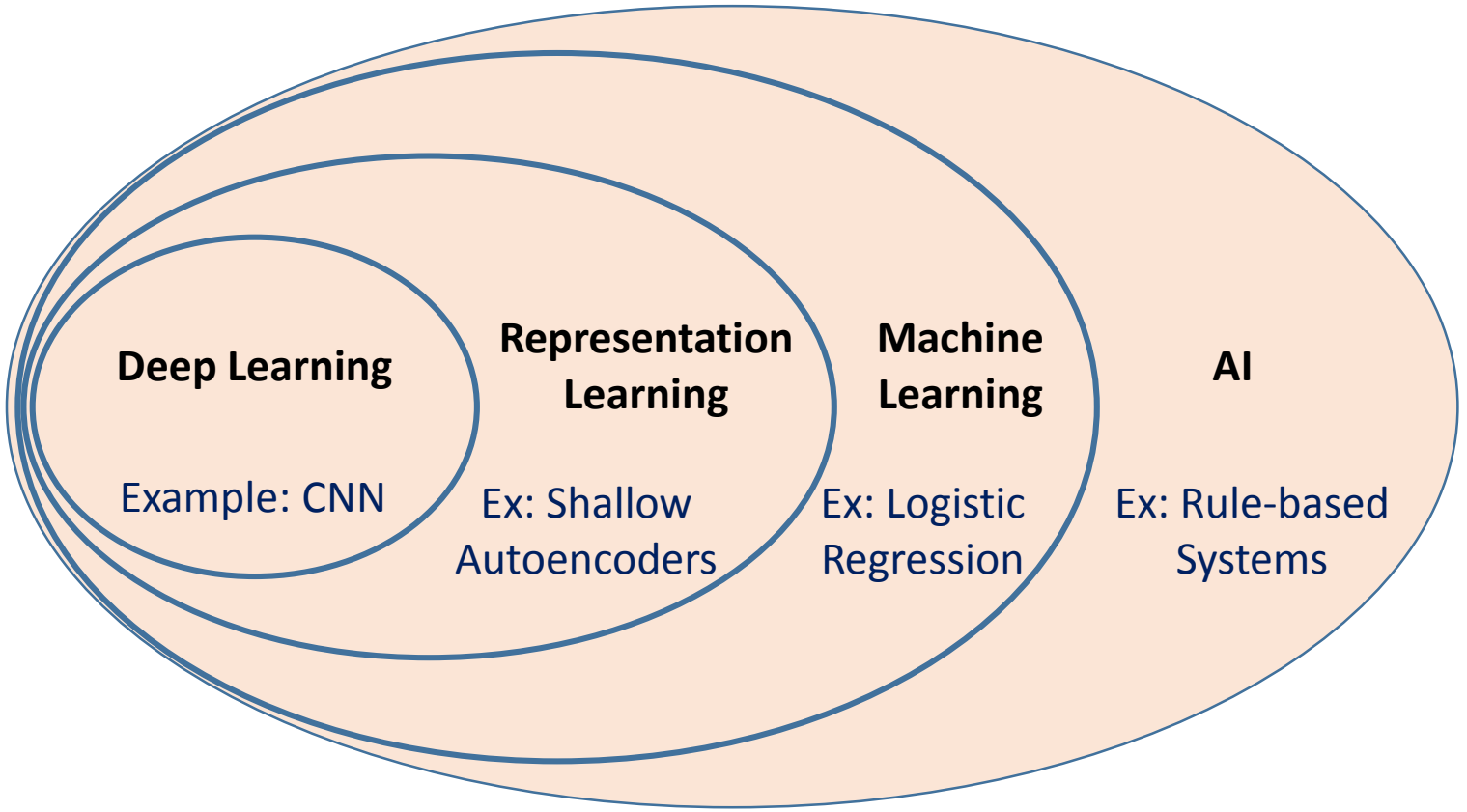
- Function to map pixels to object identity is complicated.
  - ✓ Series of hidden layers extract increasingly abstract features
  - ✓ Final decision made by a simple classifier (like Softmax)

- The key aspect of deep learning: Layer attributes not designed by human engineers, Learned from data using a general purpose learning procedure.



What is Deep Learning?

- Relation between AI, ML and DL



Source: https://blogsinsider.blogspot.com/2018/10/relationship-between-ai-ml-dl.html

What is Deep Learning?

❑ Deep versus Shallow Classifiers:

- Image and speech recognition require input-output function to be:
  - ✓ **insensitive** to irrelevant variations of the input, e.g. position, orientation and illumination of an object, Variations in pitch or accent of speech
  - ✓ **sensitive** to minute variations, e.g. white wolf and breed of wolf-like white dog called Samoyed

- At pixel level two Samoyeds in different positions may be very different, whereas a Samoyed and a wolf in the same position and background may be very similar.

Samoyed

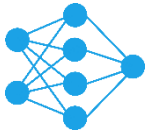
White wolf

What is Deep Learning?

- Shallow classifiers: need a good feature extractor. They can only carve the input space into very simple regions.

- Deep classifiers: produce representations that are selective to important aspects of image for discrimination but invariant to irrelevant aspects such as pose of the animal.

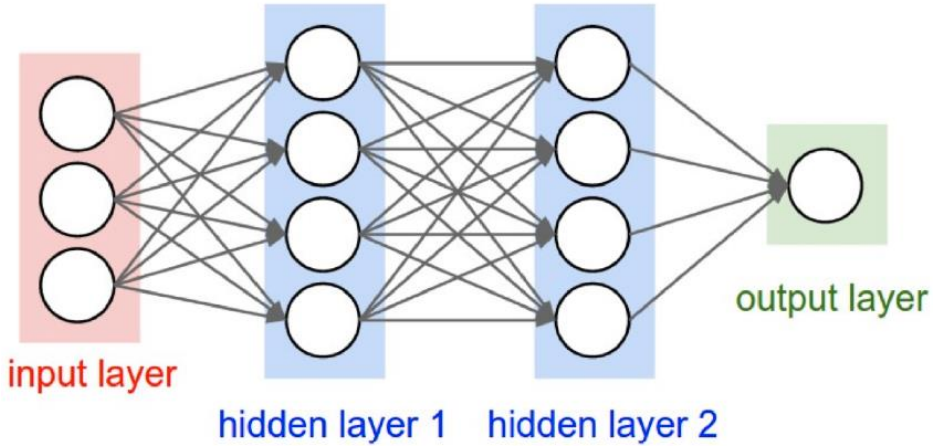
❖ Hand-designing good feature extractors requires engineering skill and domain expertise, but Deep learning learns features automatically.



What is Deep Learning?

- Many non-linear modules transform input to improve both selectivity and invariance of the representation.

- With depth of 5 to 20 layers can implement extremely complex functions of input
  - ✓ Sensitive to minute details: Distinguish Samoyeds from white wolves
  - ✓ Insensitive to irrelevant variations: Background, pose, lighting, surrounding objects

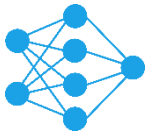


What is Deep Learning?

- Limitations of Deep Neural Networks:
  - ✓ Need too many number of training samples
  - ✓ Slow learning (convergence time)
  - ✓ Inadequate parameter selection techniques that lead to poor minima

- Recipe of deep learning models:
  - ✓ Specification of a dataset
  - ✓ A loss function
  - ✓ A model (architecture)
  - ✓ An optimization procedure

- The backpropagation algorithm is the heart of deep learning!



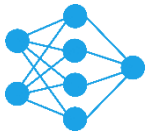
What is Deep Learning?

❑ A glimpse of the **Backpropagation algorithm**:

- After propagating the input features forward to the output layer through the various hidden layers consisting of different/same activation functions, we come up with a predicted output.

- Then, the backpropagation algorithm propagates backward from the output layer to the input layer calculating the error gradients on the way.

- Once the computation for gradients of the loss function w.r.t each parameter (weights and biases) in the neural network is done, the algorithm takes a gradient descent step towards the minimum to update the value of each parameter in the network using these gradients.

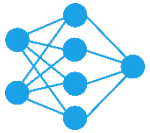


Challenges in DNN Optimization

- Learning in DNNs is a sort of optimization. It seeks to find parameters θ of a neural network that significantly reduces a loss function $J(\theta)$.
- Optimization is an extremely difficult task!
- In traditional machine learning, we design objective function and constraints carefully to ensure convex optimization.

✓ Convex function: a function that has only one local (=global) optimum.

- When training neural networks, we must face with the nonconvex case. In this section we discuss some challenges in DNN optimization.



Challenges in DNN Optimization

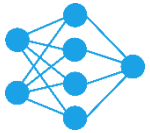
1. ill-conditioning of the Hessian: Even when optimizing convex functions, one problem is an ill conditioned Hessian matrix.

- Condition number: the ration of the maximal and minimal eigenvalues of the Hessian $\nabla^2 F(x)$. condition number = $\frac{\lambda_{max}}{\lambda_{min}}$
- If the condition number is large, we say the Hessian is ill-conditioned.
- Convergence rate of SGD is slow for ill-conditioned problems.
- In this situation, although the gradient is strong, but learning is very slow.

Challenges in DNN Optimization

2. Local Minima: In convex optimization, the problem is to find a local minimum.

- Some convex functions have a flat region rather than a global minimum point. Any point within the flat region is acceptable.
- With nonconvex functions, such as neural nets, many local minima are possible.
- Many deep models are guaranteed to have an extremely large number of local minima.

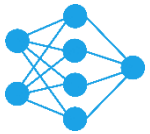


Challenges in DNN Optimization

3. Plateaus, Saddle point and Ravine:

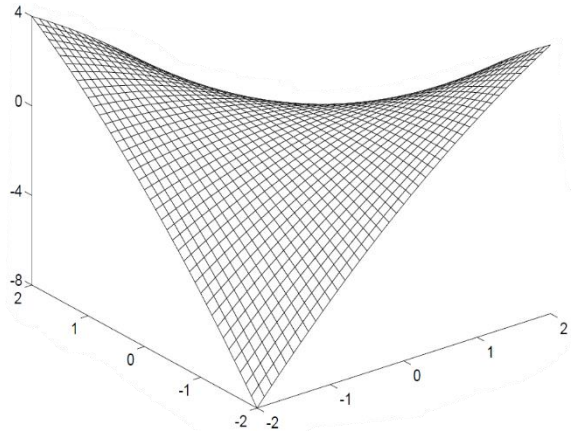
- Saddle points are zero gradient points at which Hessian has both positive and negative values.
  - Positive: have greater loss than saddle point
  - Negative: have lower loss than saddle point

- In low dimension, local minima problem is more common. But in high dimensions (as in deep learning) local minima are rare and saddle points are more common.



Challenges in DNN Optimization

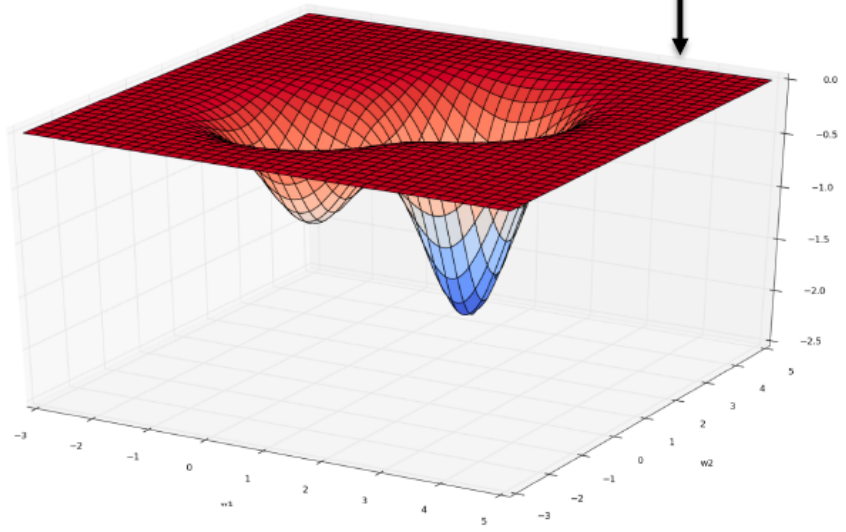
- Why second order methods (such as Newton's method) have not replaced SGD? Because of saddle points.

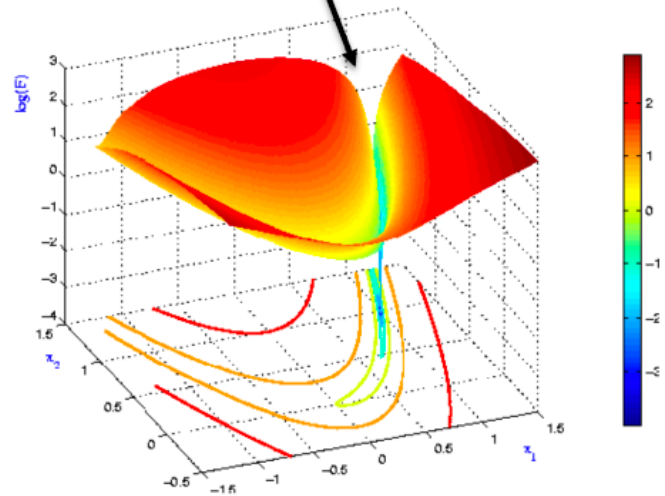


- Primary obstacle is not multiple minima but saddle points.

Challenges in DNN Optimization

Plateau and Ravine

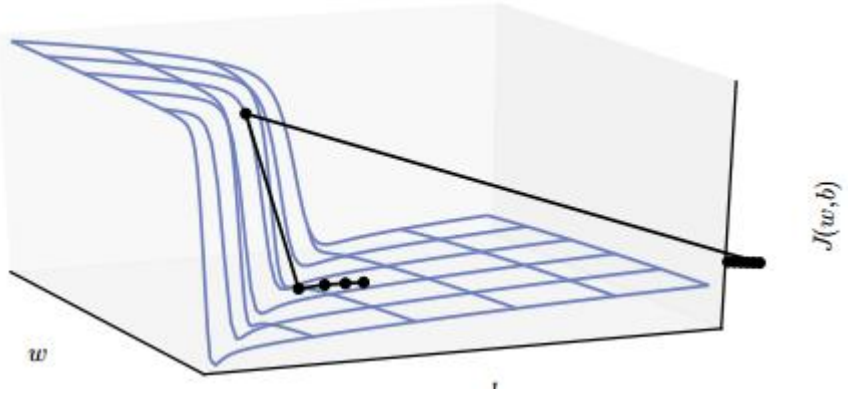




Challenges in DNN Optimization

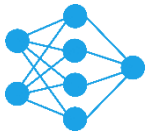
4. Cliffs and Exploding Gradients: Neural networks with many layers have steep regions resembling cliffs

- Gradient update step can move parameters extremely far, jumping off cliff altogether.
- Cliffs are dangerous from either direction.
- Cliff structures are most common in the cost functions for recurrent neural networks.



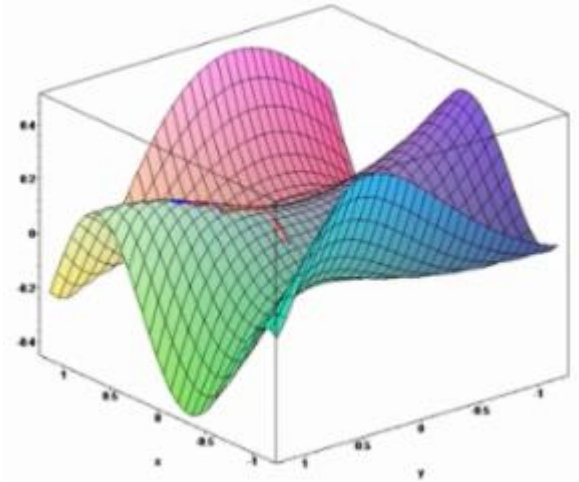
Optimizers

- Challenges we discussed in the previous section are crucial in deep learning and in some cases, the traditional optimizers like SGD have problem with them.
- In recent years, many alternative optimizers have been introduced and the study of optimization has become a main part in the field of deep learning.
- These optimizers seek to correct the gradient descent approach and are usually based on the idea of variable learning rate.
- We will discuss a number of most popular deep learning optimizers such as AdaGrad, RMSProp, Adam and finally compare them.



Optimizers

- The basic idea is that the dependence of the loss function on each parameter is not the same.
- The gradient might be really steep in one direction, but might be really flat in another direction.
- **Solution:**
  - Adapt the learning rate for each parameter individually based on how much the cost has changed in the past relative to that parameter.

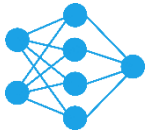


AdaGrad (Duchi et al. 2011)

- Individually adapts learning rates of all parameters by scaling them inversely proportional to the square root of the sum of the historical squared values of the gradient.

- The parameters with the largest partial derivative of the loss have a rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.

- AdaGrad performs well for some but not all deep learning models.



AdaGrad

- Define a variable called **cache** for each parameter of neural network, that accumulate the squared gradients relative to each parameter.
- If one parameter has had a lot of large gradients in the past, then its cache will be very large, and its effective learning rate will be very small, so it will change more slowly in the future.
- If a parameter has had a lot of small gradients in the past then its cache will be small, so it's effective learning rate will remain large and it will have more opportunity to change in the future.

$$cache = cache + gradient^2$$

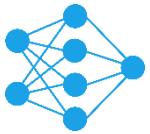
$$\theta \leftarrow \theta - \eta \frac{\nabla J}{\sqrt{cache + \varepsilon}}$$

Cache is updated at every batch

AdaGrad

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$ → **cache**
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: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
→ Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
→ Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

Each scalar parameter and its learning rate is updated independently of the others

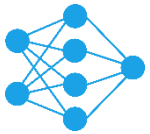


RMSProp (Hinton 2012)

- It has been observed that AdaGrad decreases learning rate too aggressively.
- The learning rate would approach zero too quickly when in fact there was still more learning to be done.
- Since cache is growing to fast, RMSProp decreases it on each update by taking a weighted average of the old cache and the new squared gradient:

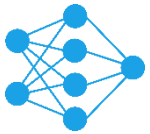
$$cache = decay * cache + (1 - decay) * gradient^2$$

- Typical values for decay: 0.99, 0.999, etc.
- We say **the cache is "leaky"!**



RMSProp

- Modifies AdaGrad to perform better in the nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average.
- RMSProp uses an exponentially decaying average to discard history from the extreme past, so that it can converge rapidly after finding a convex bowl.
- RMSProp converges rapidly when applied to convex function.



RMSProp

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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

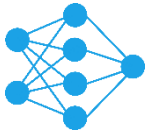
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

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)

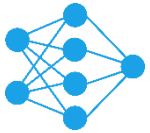
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.

end while



Adam (Kingma & Ba 2014)

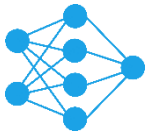
- Adam (Adaptive moments) is another adaptive learning rate optimization algorithm and is a variant of RMSProp with momentum.
- Calling "Adam" as "RMSProp with momentum" is a little confusing!!! Why?
- Because Tensorflow has RMSProp, to which we can add momentum, and that is not Adam.
- Note that, simply adding a momentum term to RMSProp doesn't quite give you Adam.
- Adam does in a sense add something like momentum to RMSProp but in a very specific way.



Adam vs. RMSProp (Tensorflow)

```
tf.keras.optimizers.experimental.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    weight_decay=None,
    clipnorm=None,
    clipvalue=None,
    global_clipnorm=None,
    use_ema=False,
    ema_momentum=0.99,
    ema_overwrite_frequency=100,
    jit_compile=True,
    name='RMSprop',
    **kwargs
)
```

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    weight_decay=None,
    clipnorm=None,
    clipvalue=None,
    global_clipnorm=None,
    use_ema=False,
    ema_momentum=0.99,
    ema_overwrite_frequency=None,
    jit_compile=True,
    name='Adam',
    **kwargs
)
```



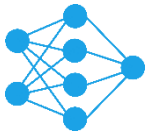
Adam vs. RMSProp (PyTorch)

RMSPROP

CLASS torch.optim.RMSprop(*params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*, *foreach=None*, *maximize=False*, *differentiable=False*)

ADAM

CLASS torch.optim.Adam(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, ***, *foreach=None*, *maximize=False*, *capturable=False*, *differentiable=False*, *fused=False*) [SOURCE]



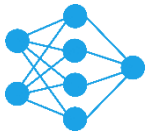
Adam

- **Sample Mean (Average)**

$$\bar{X} = \frac{1}{T} \sum_{t=1}^{T} X_t$$

- Now, suppose we have so much data, and we can't store X_1, X_2, …, X_T all at the same time

- How can we calculate the average for this data more intelligently?



Adam

$$M_T = \frac{1}{T}\sum_{t=1}^{T} X_t = \frac{1}{T}\sum_{t=1}^{T-1} X_t + \frac{1}{T}X_T = (1-\frac{1}{T})M_{T-1} + \frac{1}{T}X_T$$

- Now, instead of store all X's, we can just store the previous mean (M_{T-1}) and the current sample (X_T).

- If we set (1-1/T) to be constant value, called "decay":

$$M_T = decay * M_{T-1} + (1 - decay)X_T$$

- This looks exactly like the RMSProp cache update!

$$cache = decay * cache + (1 - decay) * gradient^2$$

Adam

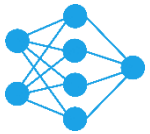
- Back to the cache update formula in RMSPRop:

$$cache = decay * cache + (1 - decay) * gradient^2$$

$$v_t = decay * v_{t-1} + (1 - decay) * g^2$$

- What is the “cache” of RMSProp really estimating?
  - The average of the squared gradient!

$$v_t = decay * v_{t-1} + (1 - decay) * g^2 \approx mean(g^2)$$



Adam

- **Expected value**
- The mean of a random variable is its expected value, e.g. mean(X)=E(X)

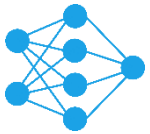
$$v_t = decay * v_{t-1} + (1 - decay) * g^2 \approx mean(g^2)$$

$$v \approx E(g^2)$$

- $E(X^2)$ = 2nd moment of *X*
- $E(X^n)$ = nth moment of *X*
- ***1st moment***

$$m \approx E(g)$$

$$m_t = \mu m_{t-1} + (1 - \mu) g_t$$



Adam

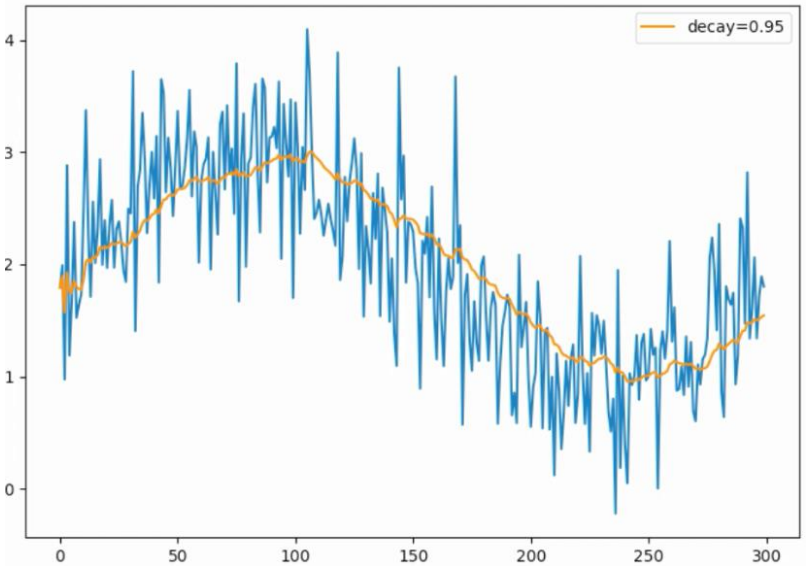
- Adam makes use of **the 1st and 2nd moments of gradient (g)**, which explains what "Adam" stands for: "Adaptive Moment Estimation"

- Adam is just a combination of these 2 things:
  - m (1st moment of g)
  - v (2nd moment of g)

- The 1st moment of g is called as the exponentially-smoothed average of g
- The 2nd moment of g is called as the exponentially-smoothed average of g^2

Adam

- **Problem with exponentially-smoothed average**



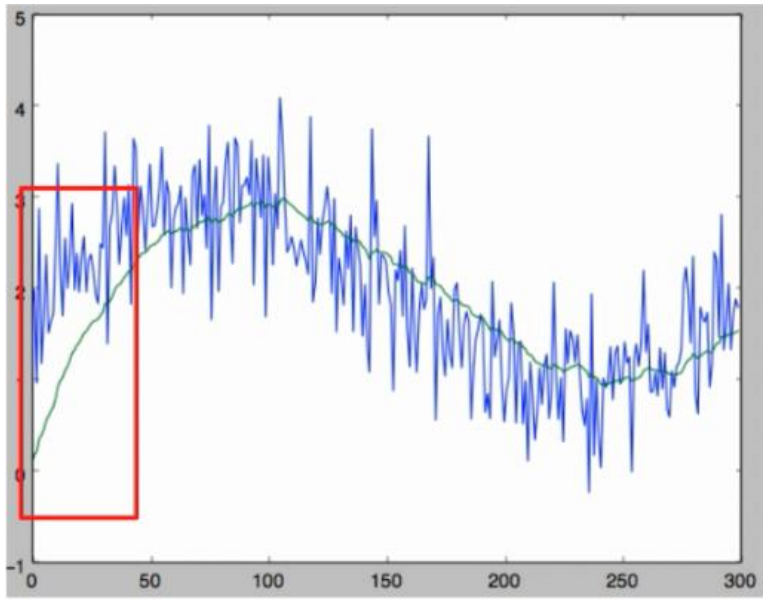
- Exponentially-smoothed average acts as a smoothing function (a low-pass filter):
  - Ignores a lot of very rapid random spikes
  - Follows the main trend of the signal

Adam

- **Problem with exponentially-smoothed average**
- What is the initial value for output?
  - If we set decay=0.99 and Y(0)=0:

$$Y(1) = 0.99 * Y(0) + 0.01 * X(1) = 0.01 * X(1)$$

  - **So the problem is:**
    - Initially our output is going to be biased towards zero.



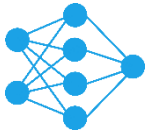
Adam

- **The solution for this problem is: Bias correction**

$$\hat{Y}(t) = \frac{Y(t)}{1-decay^t}$$

- Now, when t is small, we are dividing by a very small number, which makes the initial value (for output) bigger.

- It's a good solution, because without bias correction the initial value was to small.



Adam

- **Example for Bias correction**

$$\hat{Y}(t) = \frac{Y(t)}{1-decay^t}$$

Given: decay = 0.99

Y(1) = 0.01X(1)

Y(2) = 0.0099X(1) + 0.01X(2)

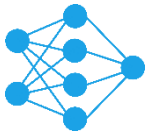
Corrected:

Y_hat(1) = 0.01/(1 - 0.99)X(1) = X1

Y_hat(2) = Y(2) / 0.0199
= 0.497X(1) + 0.503X(2)

- When t approaches infinity, the bias correction term approaches **1**. So we converged to the standard exponentially-smoothed average.

- Now our output has the correct range of values when compared with the inputs, for all values of the input including the initial ones.



Adam

- ADAM optimizer uses the bias corrected version for the 1st and 2nd moments as:

$$\hat{m}_t = \frac{m_t}{1-\beta_1{}^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2{}^t}$$

- Now the formula for the parameter update by ADAM is:

$$\theta_{t+1} \leftarrow \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

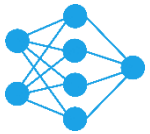
- Note that we have 2 different decay rates for m and v as β_1 and β_2
  - β_1=0.99, β_2=0.999

```
CLASS torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0,
      amsgrad=False, *, foreach=None, maximize=False, capturable=False, differentiable=False,
      fused=False) [SOURCE]
```

Recap: Adam

- Adam is a modern adaptive learning rate technique that is default for many deep learning practitioners.
- Combine RMSProp (cache mechanism) with momentum (kepping track of old geadients).
- Generalize as the 1st and 2nd moments
- Apply bias correction

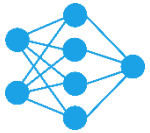
$$\hat{m}_t = \frac{m_t}{1-\beta_1{}^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2{}^t} \qquad \theta_{t+1} \leftarrow \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$



Adam

- Adam Algorithm

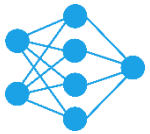
Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$



Adam

- Adam Algorithm (Cont.)

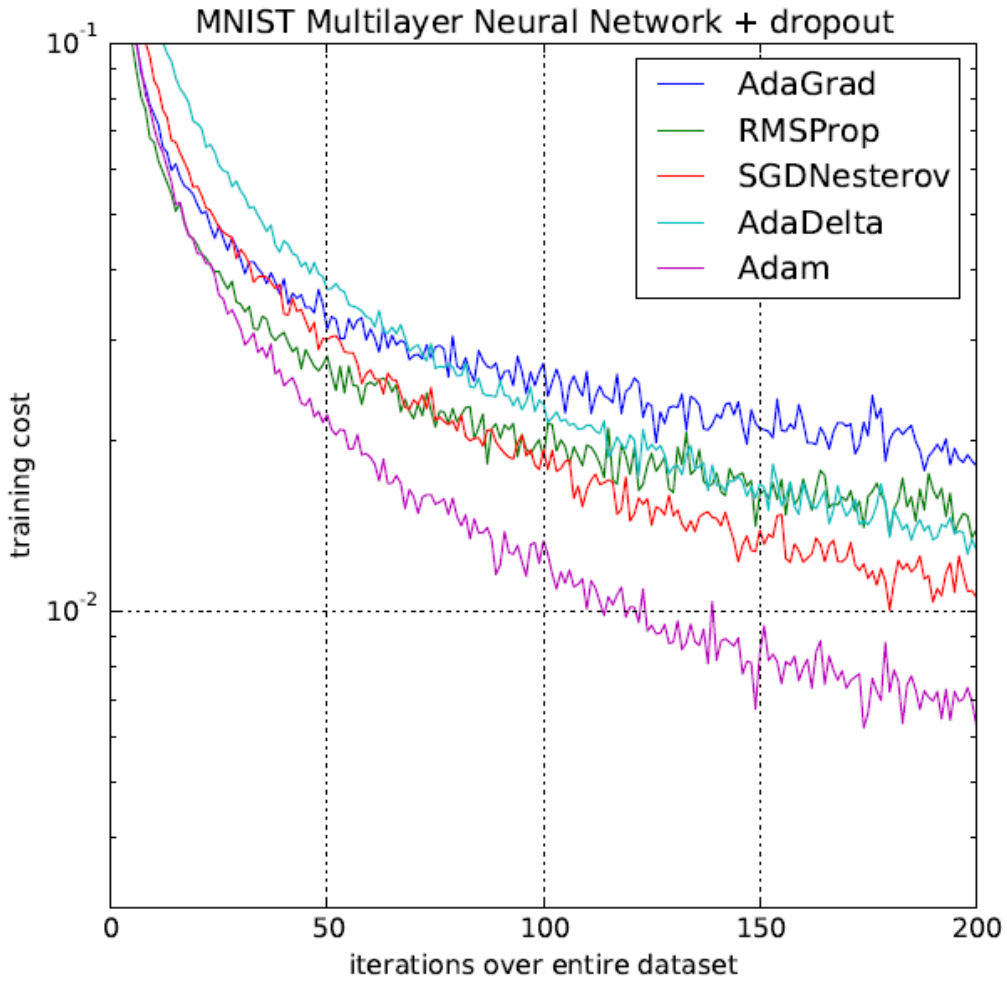
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  $t \leftarrow t + 1$
  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: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
  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



Optimizers

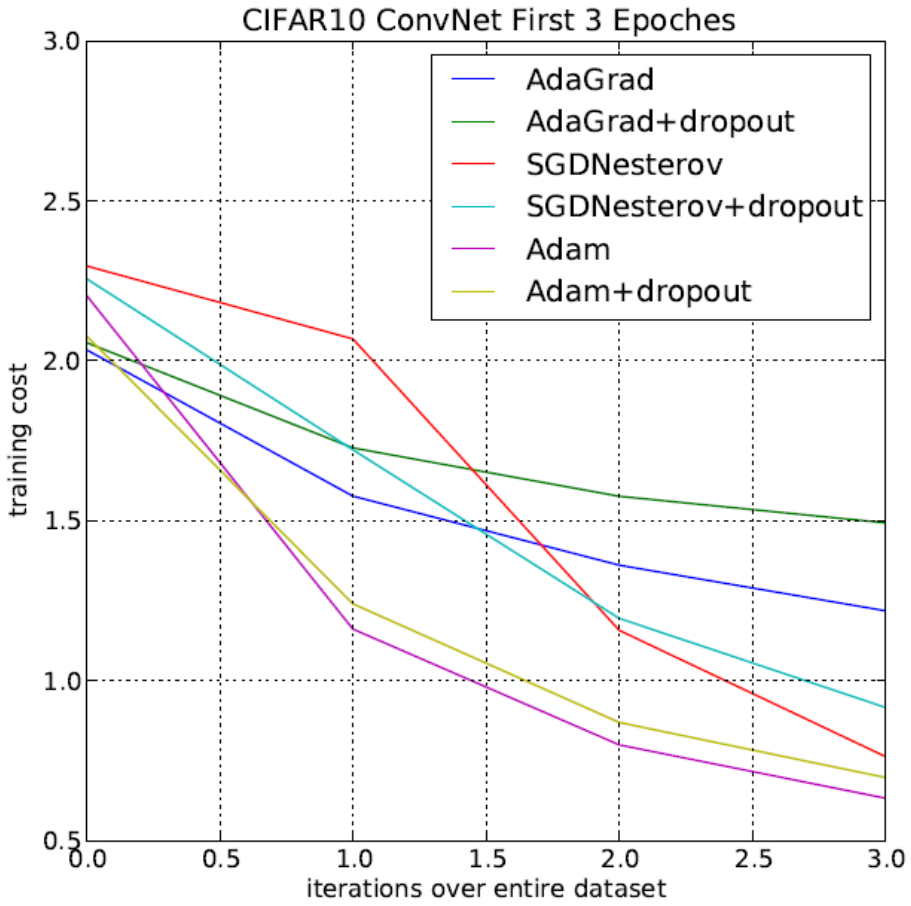
❑ Comparison (Kingma & Ba 2014)

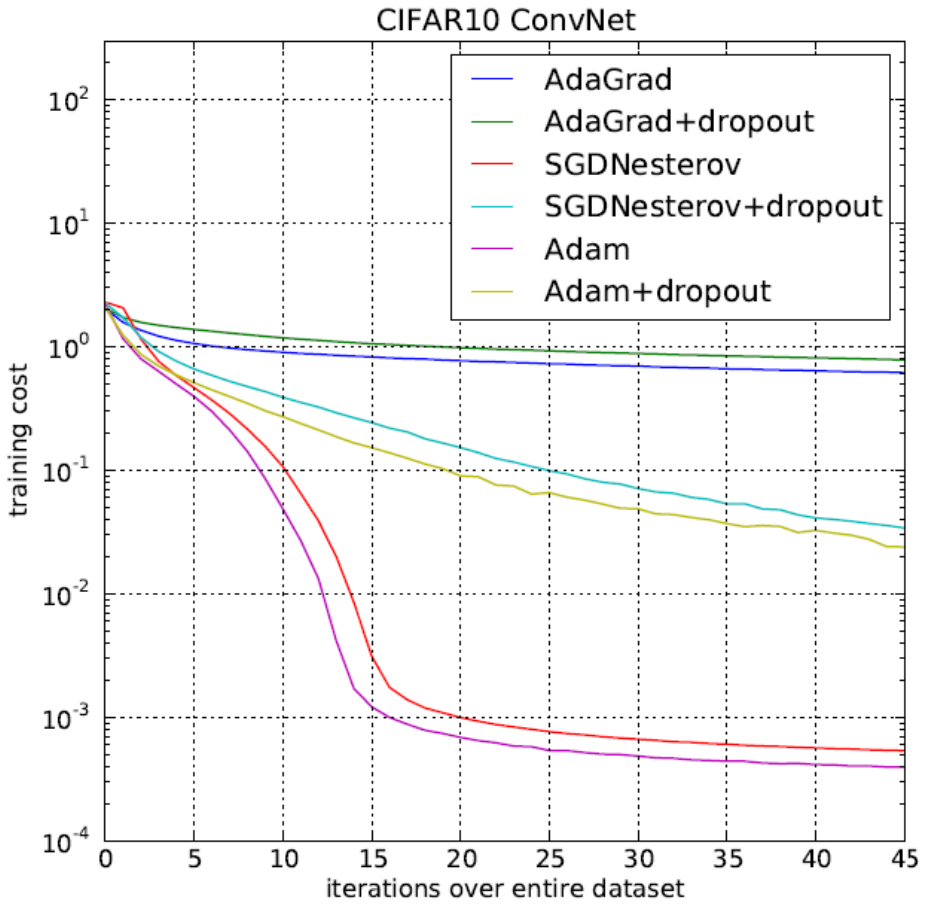
- Performance comparison in multilayer neural networks on MNIST dataset.



Optimizers

- Performance comparison in CNN on CIFAR10 dataset.

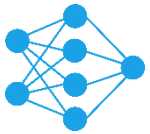




Optimizers

❑ Choosing the Right Optimizer

- We have discussed several methods of optimizing deep models by adapting the learning rate for each model parameter.
- Most popular algorithms actively in use: SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaGrad and Adam.
- Which algorithm to choose? There is no consensus. But Adam is usually better in most deep learning tasks.



Vanishing & Exploding Gradients

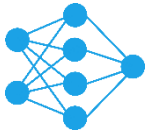
- Consider a deep network with *L* layers, input $\mathbf{x}$ and output $\mathbf{o}$.

- Each layer l defined by a transformation f_l parameterized by $\mathbf{W}^{(l)}$, whose hidden layer output is $\mathbf{h}^{(l)}$ so we have:

$$\mathbf{h}^{(l)} = f_l\left(\mathbf{h}^{(l-1)}\right)$$

- Suppose $\mathbf{h}^{(0)} = \mathbf{x}$ and $\mathbf{h}^{(L)} = \mathbf{o}$.

- Note that the input and output of all layers are vectors.



Vanishing & Exploding Gradients

- Forward propagation

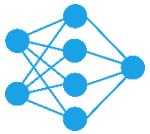
$$\mathbf{o} = f_L \circ f_{L-1} \ \dots \circ f_1(\mathbf{x})$$

- Computation of loss function, for simplicity, for one sample $\mathbf{x}$

$$loss = \mathcal{L}(\mathbf{o}, \mathrm{y})$$

- Backpropagation (Compute the gradient of $loss$ w.r.t. $\mathbf{W}^{(l)}$)

$$\frac{\partial loss}{\partial \mathbf{W}^{(l)}} = \frac{\partial loss}{\partial \mathbf{o}} \underbrace{\frac{\partial \mathbf{o}}{\partial \mathbf{h}^{(L-1)}} \frac{\partial \mathbf{h}^{(L-1)}}{\partial \mathbf{h}^{(L-2)}} \cdots \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}}_{\text{Multiplication of } L\text{-}l \text{ matrices}} \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}}$$



Vanishing & Exploding Gradients

- At the core of the backpropagation formula, we have:

$$\frac{\partial loss}{\partial \mathbf{W}^{(l)}} = \frac{\partial loss}{\partial \mathbf{o}} \prod_{i=l}^{L-1} \frac{\partial \mathbf{h}^{(i+1)}}{\partial \mathbf{h}^{(i)}} \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}}$$

- If the values of these gradients are more than one, we have the problem of **Gradient Exploding**.

$$1.5^{100} \approx 4 \times 10^{17}$$

- If the values of these gradients are less than one, we have the problem of **Gradient Vanishing**.

$$0.8^{100} \approx 2 \times 10^{-10}$$

Vanishing & Exploding Gradients

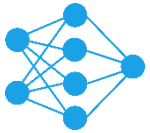
- Assume MLP (without bias for simplicity):

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}\right), \qquad \sigma \text{ is the activation function}$$

$$\frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{h}^{(l-1)}} = diag\left(\sigma'\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}\right)\right)\left(\mathbf{W}^{(l)}\right)^T, \quad \sigma' \text{ is the gradient of } \sigma$$

- So, we have:

$$\prod_{i=l}^{L-1} \frac{\partial \mathbf{h}^{(i+1)}}{\partial \mathbf{h}^{(i)}} = \prod_{i=l}^{L-1} diag\left(\sigma'\left(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)}\right)\right)\left(\mathbf{W}^{(i)}\right)^T$$



Vanishing & Exploding Gradients

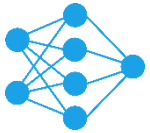
- Main question?

Why do the gradients even vanish/explode?

- Given the equation below, some factors are more effective such as:

$$\prod_{i=l}^{L-1} \frac{\partial \mathbf{h}^{(i+1)}}{\partial \mathbf{h}^{(i)}} = \prod_{i=l}^{L-1} diag\left(\sigma'\left(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)}\right)\right)\left(\mathbf{W}^{(i)}\right)^T$$

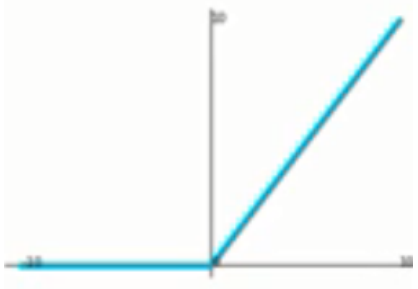
- **Activation Function**
- **The initial values of weights**



Effect of ReLU on Exploding Gradients

- Use ReLU as the activation function

$$\sigma(x) = \max(0,x) \quad \text{and} \quad \sigma'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

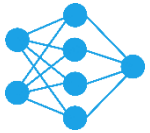


- So, for equation bellow:

$$\prod_{i=l}^{L-1} \frac{\partial \mathbf{h}^{(i+1)}}{\partial \mathbf{h}^{(i)}} = \prod_{i=l}^{L-1} diag\left(\sigma'\left(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)}\right)\right)\left(\mathbf{W}^{(i)}\right)^T$$

- the values of $\sigma'\left(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)}\right)$ go towards 0 or 1, so, we may have:

$$\prod_{i=l}^{L-1} \frac{\partial \mathbf{h}^{(i+1)}}{\partial \mathbf{h}^{(i)}} = \prod_{i=l}^{L-1} \left(\mathbf{W}^{(i)}\right)^T$$



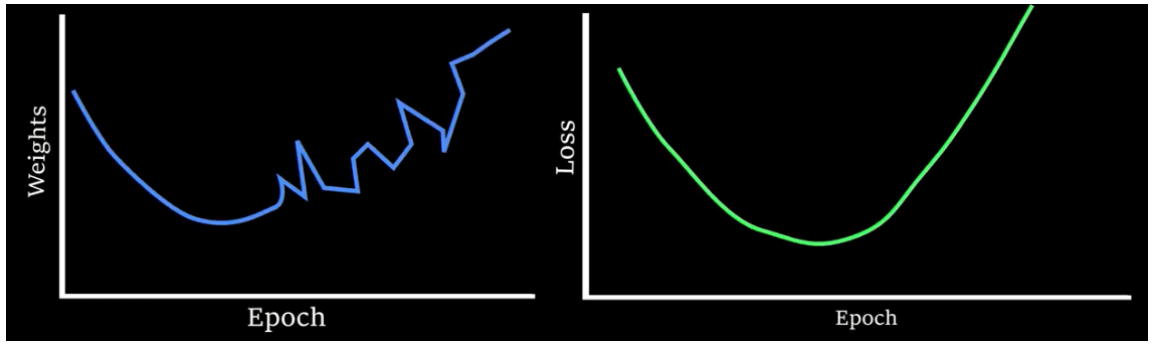
Exploding Gradients

$$\prod_{i=l}^{L-1} \frac{\partial \mathbf{h}^{(i+1)}}{\partial \mathbf{h}^{(i)}} = \prod_{i=l}^{L-1} \left(\mathbf{W}^{(i)} \right)^T$$

- So, if $\mathbf{W}^{(i)}$ contains larger values than 1, and $L-l$ is large (especially for bottom layers close to the input layer), this multiplication of gradient values could be very large. As a result the exploding of gradients could occur.

Issues with Exploding Gradients

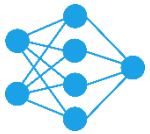
- **Value out of range: infinity value**
  - Severe for using 16-bit floating points
  - The parameters (weights) will quickly become large or infinity (NaN values).
- **Sensitive to learning rate (LR)**
  - Not small enough LR → large weights → larger gradients → larger weights → …
  - Too small LR → No progress
  - May need to change LR dramatically during training



Solutions for Exploding Gradients

- **Gradient Clipping**

- **Batch Normalization**
  - It’s not easily implemented for RNNs, so clipping is better solution for RNNs.

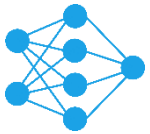


Solutions for Exploding Gradients

❑ Gradient Clipping:

- A popular technique to mitigate the exploding gradients problem.
- It clips the gradients during backpropagation so that they never exceed some threshold.

- **Two techniques for Clipping gradient values :**
  - Applying a threshold to each gradient value (for example –1 and +1)
  - Multiplying each gradient value by (Threshold/Gradient Norm)



Solutions for Exploding Gradients

Gradient Clipping

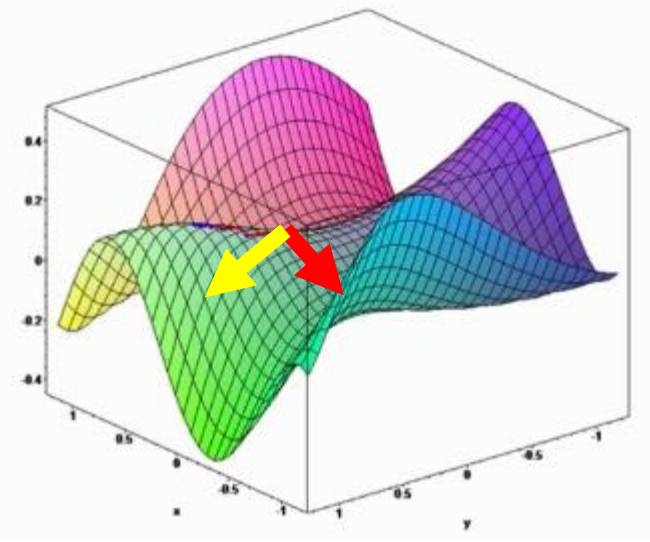
- Applying a threshold to each gradient value (for example –1 and +1)

$$\text{Gradient vector} = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial w_2} \\ \cdots \\ \frac{\partial C}{\partial w_n} \end{bmatrix}$$

$$\begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix} \Rightarrow \begin{bmatrix} 0.9 \\ 1.0 \\ 1.0 \\ -1.0 \\ 0.3 \end{bmatrix}$$

The ratio between values in gradient vector change!

A major problem:
Change the direction of the gradient vector



Solutions for Exploding Gradients

❑ Gradient Clipping

- "Clip by norm" a good solution to keep the same direction of the gradient

$$\begin{bmatrix} 0.9 \\ 3.2 \\ 150.0 \\ -2.1 \\ 0.3 \end{bmatrix}$$

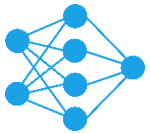
Clip by maximum of norm

$$\begin{bmatrix} 0.006 \\ 0.021 \\ 1.0 \\ -0.014 \\ 0.002 \end{bmatrix}$$

Another Problem!!!

Some of the weights are very small

Solution:
Experiments with both approaches and with different threshold values



Solutions for Exploding Gradients

❑ Gradient Clipping in PyTorch:

TORCH.NN.UTILS.CLIP_GRAD_NORM_

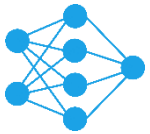
torch.nn.utils.clip_grad_norm_(*parameters*, *max_norm*, *norm_type=2.0*, *error_if_nonfinite=False*) [SOURCE]

Clips gradient norm of an iterable of parameters.

The norm is computed over all gradients together, as if they were concatenated into a single vector. Gradients are modified in-place.

Parameters:

- **parameters** (*Iterable*[*Tensor*] *or Tensor*) – an iterable of Tensors or a single Tensor that will have gradients normalized
- **max_norm** (*float or int*) – max norm of the gradients
- **norm_type** (*float or int*) – type of the used p-norm. Can be `'inf'` for infinity norm.
- **error_if_nonfinite** (*bool*) – if True, an error is thrown if the total norm of the gradients from `parameters` is `nan`, `inf`, or `-inf`. Default: False (will switch to True in the future)

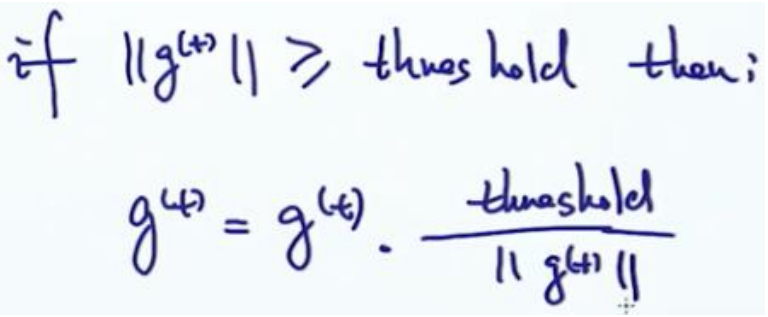


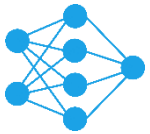
Solutions for Exploding Gradients

Gradient Clipping in PyTorch:

```
[docs]def clip_grad_norm_(
        parameters: _tensor_or_tensors, max_norm: float, norm_type: float = 2.0,
        error_if_nonfinite: bool = False) -> torch.Tensor:
```

```
    if isinstance(parameters, torch.Tensor):
        parameters = [parameters]
    grads = [p.grad for p in parameters if p.grad is not None]
    max_norm = float(max_norm)
    norm_type = float(norm_type)
    if len(grads) == 0:
        return torch.tensor(0.)
    device = grads[0].device
    if norm_type == inf:
        norms = [g.detach().abs().max().to(device) for g in grads]
        total_norm = norms[0] if len(norms) == 1 else torch.max(torch.stack(norms))
    else:
        total_norm = torch.norm(torch.stack([torch.norm(g.detach(), norm_type).to(device) for g
in grads]), norm_type)
    if error_if_nonfinite and torch.logical_or(total_norm.isnan(), total_norm.isinf()):
        raise RuntimeError(
            f'The total norm of order {norm_type} for gradients from '
            '`parameters` is non-finite, so it cannot be clipped. To disable '
            'this error and scale the gradients by the non-finite norm anyway, '
            'set `error_if_nonfinite=False`')
    clip_coef = max_norm / (total_norm + 1e-6)
    # Note: multiplying by the clamped coef is redundant when the coef is clamped to 1, but
doing so
    # avoids a `if clip_coef < 1:` conditional which can require a CPU <=> device
synchronization
    # when the gradients do not reside in CPU memory.
    clip_coef_clamped = torch.clamp(clip_coef, max=1.0)
    for g in grads:
        g.detach().mul_(clip_coef_clamped.to(g.device))
    return total_norm
```



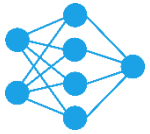


Vanishing Gradients

- As the BP algorithm advances from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged.

- When this phenomenon occurs in neural networks, the gradient descent (or almost any other optimizer) never converges to the optimum.

- As we go backward through layers, the gradient value vanishes, so this is known as the **vanishing gradients** problem.

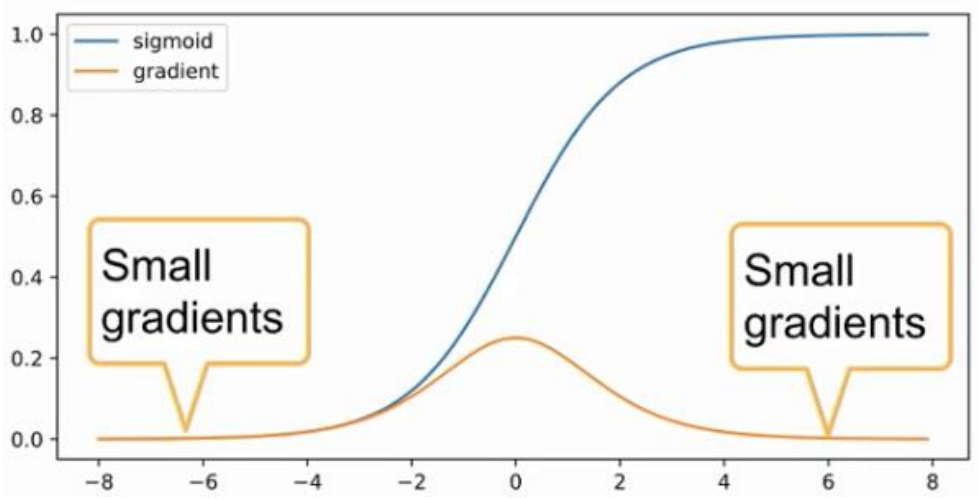


Effect of Sigmoid on Vanishing Gradients

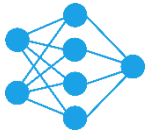
- Use Sigmoid as the activation function

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

$$\prod_{i=l}^{L-1} \frac{\partial \mathbf{h}^{(i+1)}}{\partial \mathbf{h}^{(i)}} = \prod_{i=l}^{L-1} diag\left(\sigma'\left(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)}\right)\right)\left(\mathbf{W}^{(i)}\right)^T$$



During gradient descent, the gradient depends on the derivative of the activation function to its input.



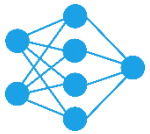
Effect of Sigmoid on Vanishing Gradients

- Observing the graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero.

- When the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network.

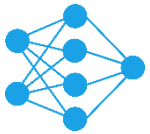
- Even if little residual gradients exist, as the algorithm progresses down through the top layers, the gradient keeps on diluting. So, this leaves nothing for the lower layers.

- Sigmoid function is only used as the activation of the output layer for binary classification.



Issues with Vanishing Gradients

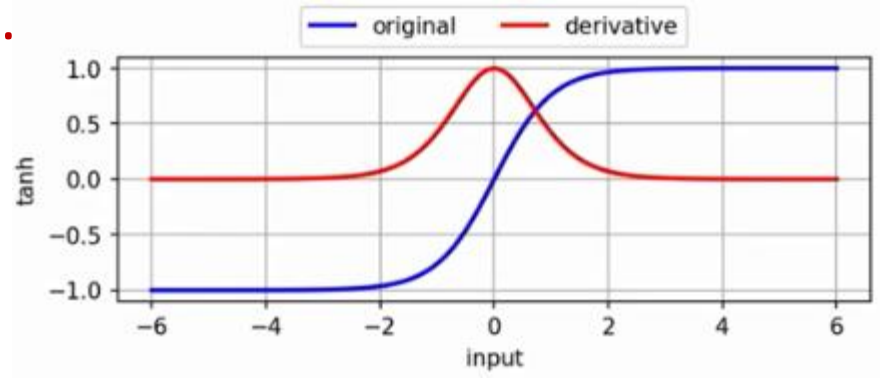
- **Gradient with values close to 0**
  - The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).
- **No progress in training**
  - No matter how to choose learning rate
- **Severe with bottom layers**
  - Only top layers are well trained
  - No benefit to make networks deeper



Solutions for Vanishing Gradients

❑ Using different activation functions instead of Sigmoid:

- **Tanh function**
  - It always works better than sigmoid.
  - In a way, it centralizes the data (the outputs of the activations), such the mean of data is close to zero.
  - As a result, it can be found that the centering conducted by using Tanh activation function, makes the learning for the next layer a little bit easier.
  - But similar to that of Sigmoid, Tanh function gives small gradient values for larger inputs (negative or positive).

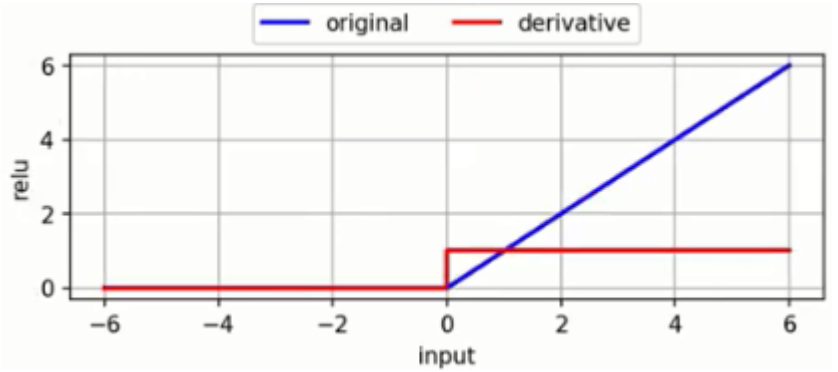


Solutions for Vanishing Gradients

❑ Using different activation functions instead of Sigmoid:

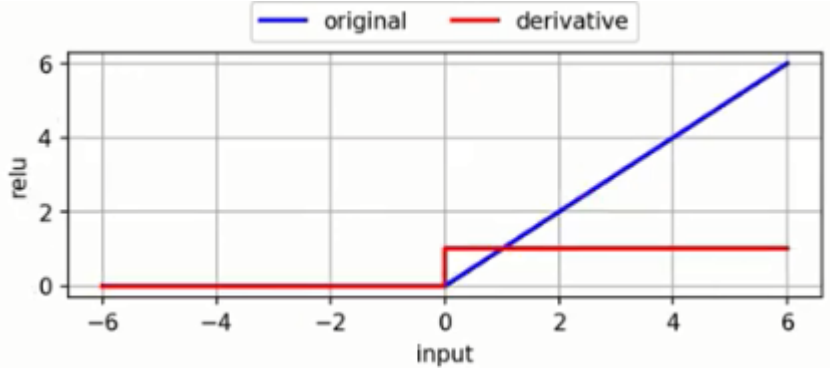
- **ReLU function**
  - To tackle the issue regarding the saturation of activation functions like sigmoid and tanh, we must use some other non-saturating functions like ReLU and its alternatives.

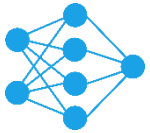
  - The derivative of ReLU function for positive input values is 1 (different from zero as in sigmoid and TanH). So, the NN learns much faster than sigmoid and tanh.



Solutions for Vanishing Gradients

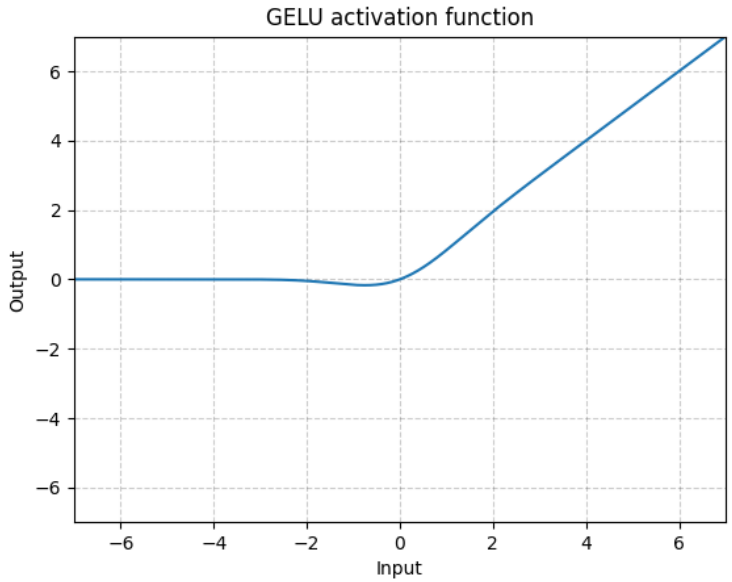
- **ReLU function**
  - It suffers from a problem known as dying ReLUs wherein some neurons just die out, meaning they keep on throwing 0 as outputs with the advancement in training.

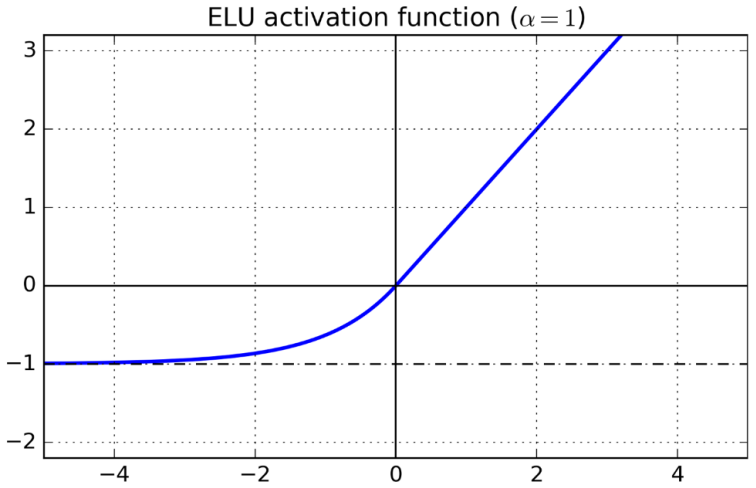


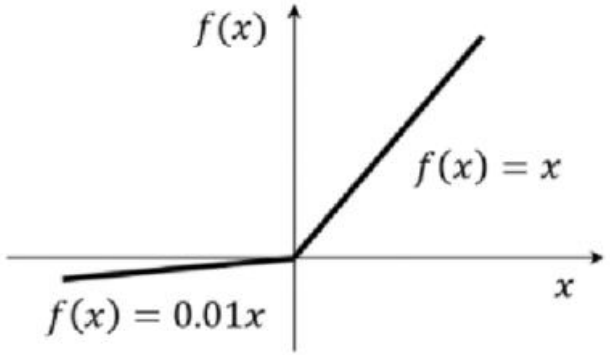


Solutions for Vanishing Gradients

- Some popular alternative functions of the ReLU that mitigates the problem of vanishing gradients when used as activation for the intermediate layers of the network are GELU, ELU, LeakyReLU:







LeakyReLU activation function

Vanishing & Exploding Gradients

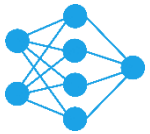
❑ Proper weight initialization:

- Different methods for initialization have been introduced in recent years (like He initialization or LeCun initialization).
- One of the most popular initialization strategies is Xavier initialization (Xavier Glorot et al. 2011)

torch.nn.init.xavier_normal_(*tensor*, *gain=1.0*) [SOURCE]

Fills the input *Tensor* with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks* - Glorot, X. & Bengio, Y. (2010), using a normal distribution. The resulting tensor will have values sampled from $\mathcal{N}(0, \text{std}^2)$ where

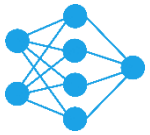
$$\text{std} = \text{gain} \times \sqrt{\frac{2}{\text{fan\_in} + \text{fan\_out}}}$$



Weight initialization (PyTorch)

```python
# Simple CNN example
class CNN(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=in_channels, out_channels=6, kernel_size=3, stride=1, padding=1
        self.pool = nn.MaxPool2d(kernel_size=(2,2), stride = (2,2))
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(16*7*7, num_classes)

        self.initialize_weights()

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = x.reshape(x.shape[0], -1)
        x = self.fc1(x)

        return x

    def initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_uniform_(m.weight)

                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)

            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

            elif isinstance(m, nn.Linear):
                nn.init.kaiming_uniform_(m.weight)
                nn.init.constant_(m.bias, 0)

if __name__ == '__main__':
    model = CNN(in_channels=3,num_classes=10)
```

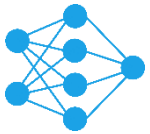
```python
def initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_uniform_(m.weight)

            if m.bias is not None:
                nn.init.constant_(m.bias, 0)

        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

        elif isinstance(m, nn.Linear):
            nn.init.kaiming_uniform_(m.weight)
            nn.init.constant_(m.bias, 0)
```



Recap of Solutions for E/V Gradients

- Solutions to avoid vanishing/exploding gradients:

1. Proper Weight Initialization
2. Using Non-saturating Activation Functions
3. Gradient Clipping (only for exploding gradients)
4. **Batch Normalization**

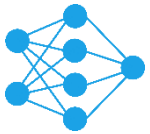


Batch Normalization

- One of the most exciting recent innovations in deep learning is Batch normalization (Sergey Ioffe & Christian Szegedy 2015)

- The motivation for batch normalization was difficulty of choosing learning rate in deep networks and E/V gradients problem in DNNs.

- BN adds an additional step between layers, in which the output of the earlier layer:
  - is normalized by standardizing the mean and standard deviation of each individual unit,
  - and then applying two learnable parameters called rescale and offset.



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

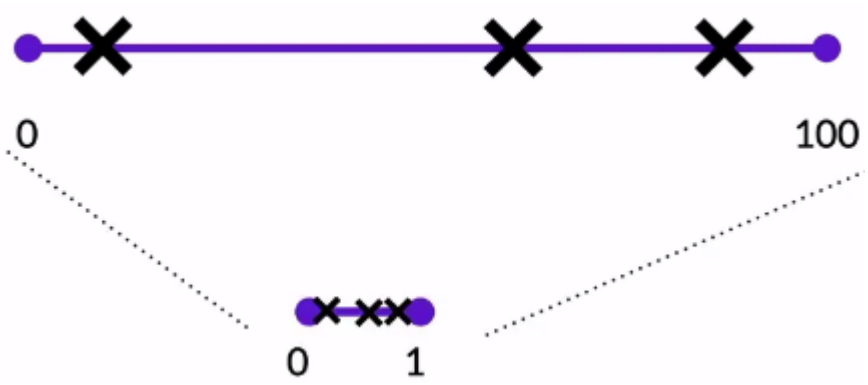
Sergey Ioffe SIOFFE@GOOGLE.COM
Christian Szegedy SZEGEDY@GOOGLE.COM
Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043

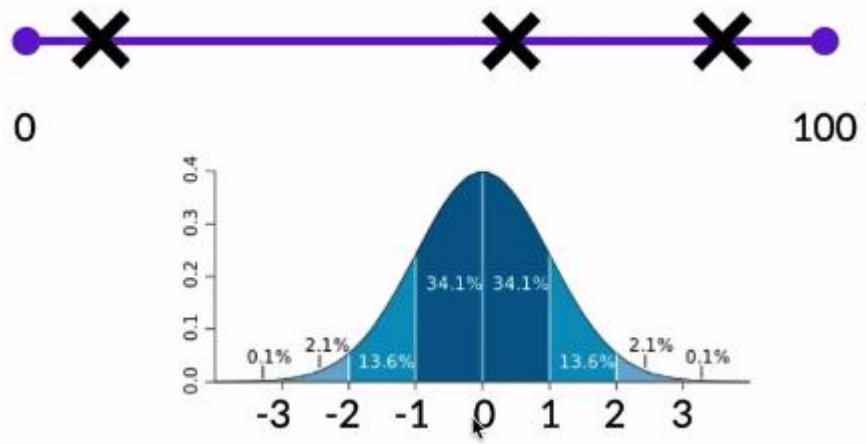
Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization *for each training mini-batch*. Batch Normalization allows us to use much higher learning rates and be less careful about initialization, and in some cases eliminates the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.82% top-5 test error, exceeding the accuracy of human raters.

Batch Normalization

- **Normalization**
  - Collapse inputs to be between 0 and 1.
- **Standardization**
  - Make mean 0 and variance 1

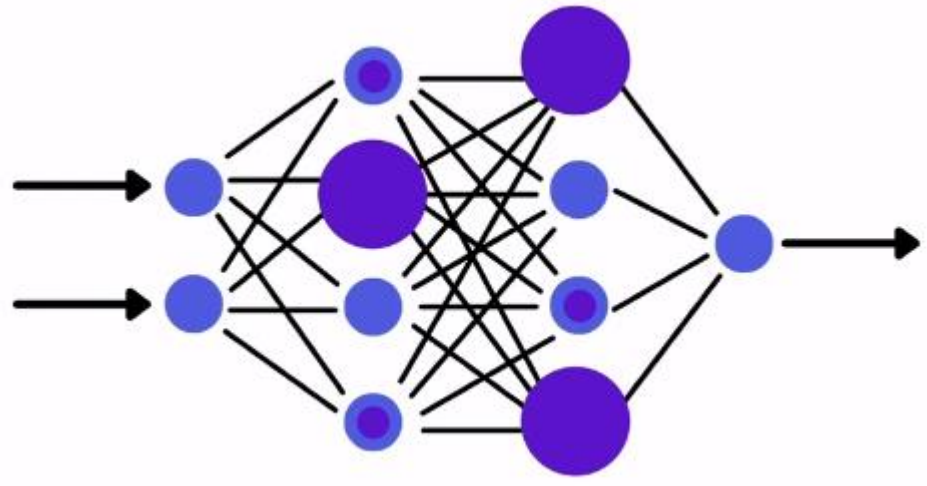




Sometimes both methods referred to as same

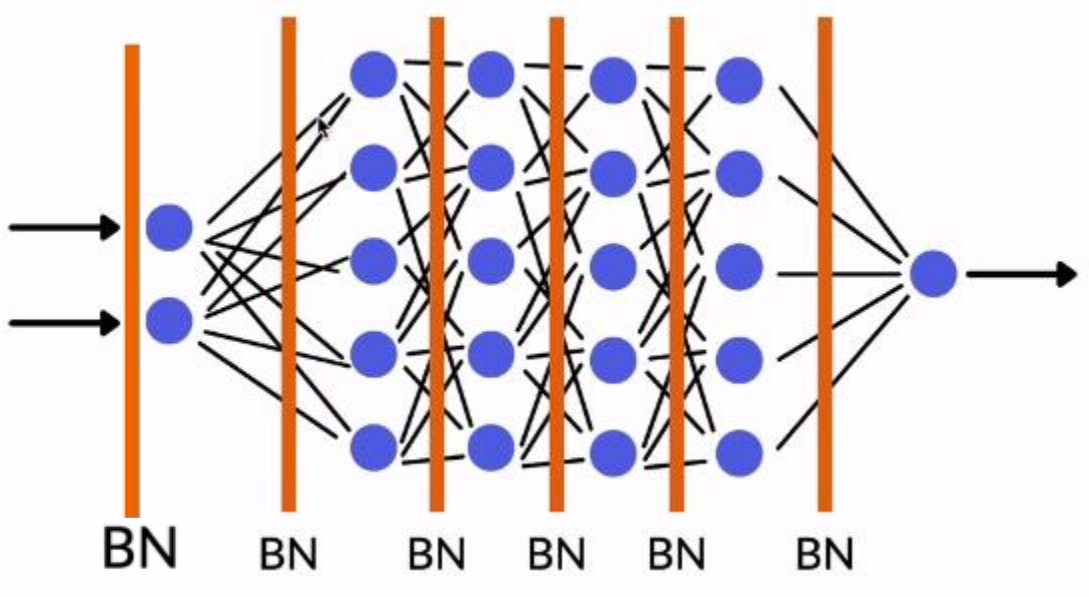
Batch Normalization

- **Why do we need normalization/standardization?**
  - Without normalization/standardization, higher values will dominant lower values, making the lower values useless in the dataset. So, the network will be unstable which increases the convergence time.
  - So, we need that our data or input values be on the same scale

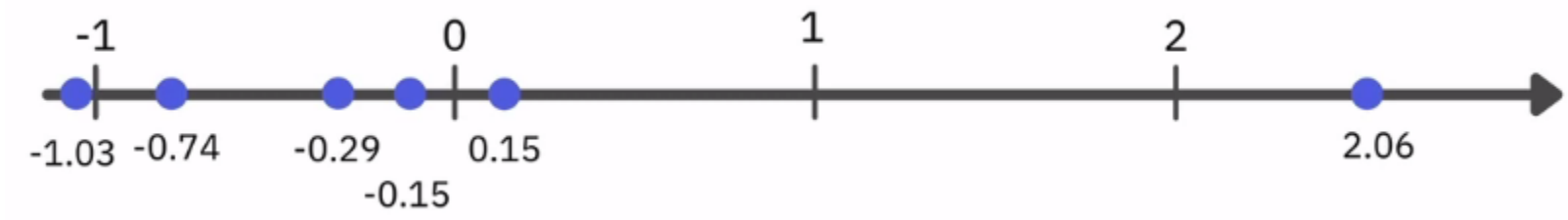


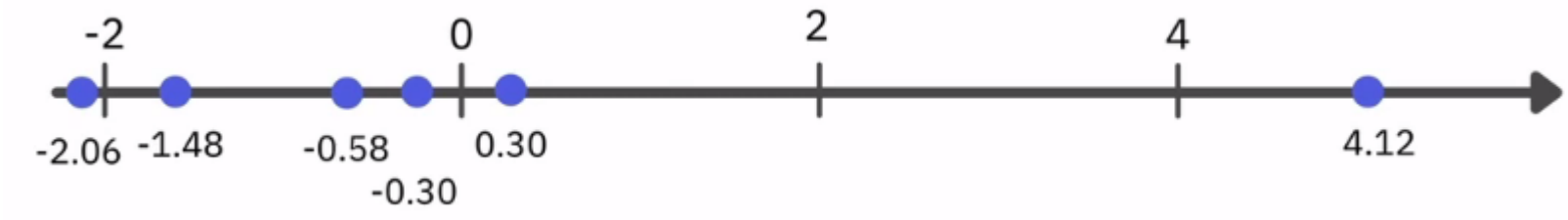
Batch Normalization

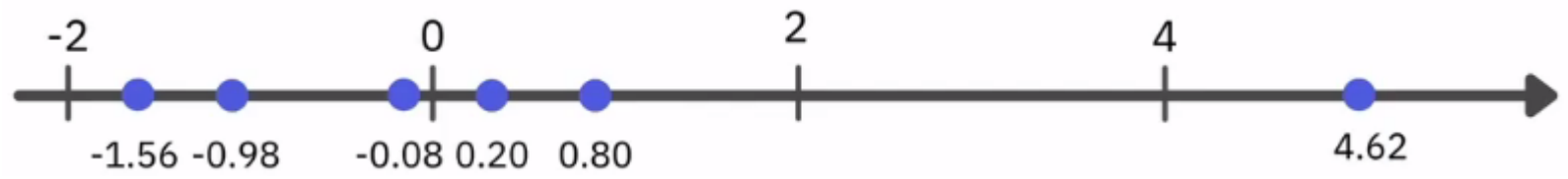
- **The maim idea of batch normalization (BN)**
  - Instead of only input normalization, we apply the normalization to all the hidden layers.
  - Insert a BN layer between each 2 layer (**even before the input layer**).



Batch Normalization







$$z^{(i)} = \underset{2}{\gamma} \otimes \hat{x}^{(i)} + \underset{0.5}{\beta}$$

Batch Normalization

The number of samples in a batch

$$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} x^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(x^{(i)} - \mu_B\right)^2$$

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

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

rescale offset

[3, 5, 8, 9, 11, 24]

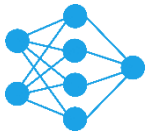
$$\mu_\beta = \frac{1}{6}(3+5+8+9+11+24) = 10$$

$$\sigma_\beta^2 = \frac{1}{6}(3-10)^2 + (5-10)^2 + \ldots + (24-10)^2 = 46$$

$$\hat{x}^{(0)} = \frac{3-10}{\sqrt{46^2 + 0.00001}} = -1.03$$

[-1.03, -0.74, -0.29, -0.15, 0.15, 2.06]

Mean = 0
Std = 0.998

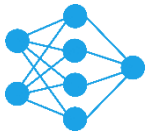


Two parameters, i.e. rescale and offset, are the learnable parameters for each BN.

Batch Normalization

❑ Advantages of BN

- ✓ Achieves same accuracy faster
- ✓ Can lead to better performance
- ✓ Using much higher learning rates and be less care about initialization
- ✓ No need to have a standardization layer
- ✓ Reduce the need for other regularization
- ✓ Epochs take longer due to the amount of computations, but convergence will be faster



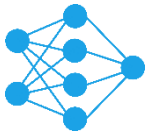
Batch Normalization

Some Notes about BN

- At training stage, mean and variance are computed using the samples of each mini-batch. So, it's called batch normalization.

- Mean and variance are computed for each neuron in each layer, such that the values of mean and variance for each neuron is 0 and 1, respectively.

- For example, if we have 512 neurons, the number of learnable parameters for the BN applied to the output of this layer is 1024.

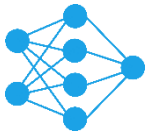


Batch Normalization

❑ Some Notes about BN

- Generally different units (neurons) can have different distributions. So, BN uses two learnable parameters, as gamma and beta, to provide different distributions for each unit, not necessarily distributions of 0 mean and unit variance for all neurons.

- Gamma and beta are two learnable parameters and should be learned using a gradient descent algorithm such as ADAM, as we would update the weight parameters.

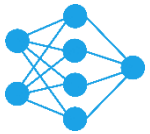


Batch Normalization

❑ Some Notes about BN

- Applying before or after the activation function?

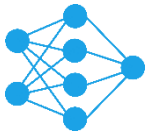
  - Before activation is more often, but both of them are correct.
  - Check for your problem which one is better!
  - If you use BN before activation, you can omit the bias parameter. Why?

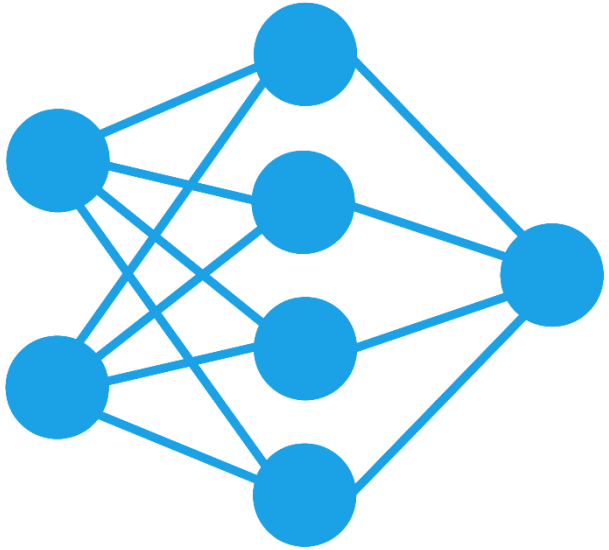


Batch Normalization

❑ Some Notes about BN

- How is BN used at the test time?
  - At test time, we might test the NN using just single sample, so we have no batch for computing the mean and variance parameters.`
  - The mean (μ) and variance (σ^2) may be replaced by running averages that were collected during training time.
  - Two parameters gamma (γ) and beta (β) are the final optimal parameters obtained from the learning procedure.





Thanks for your attention

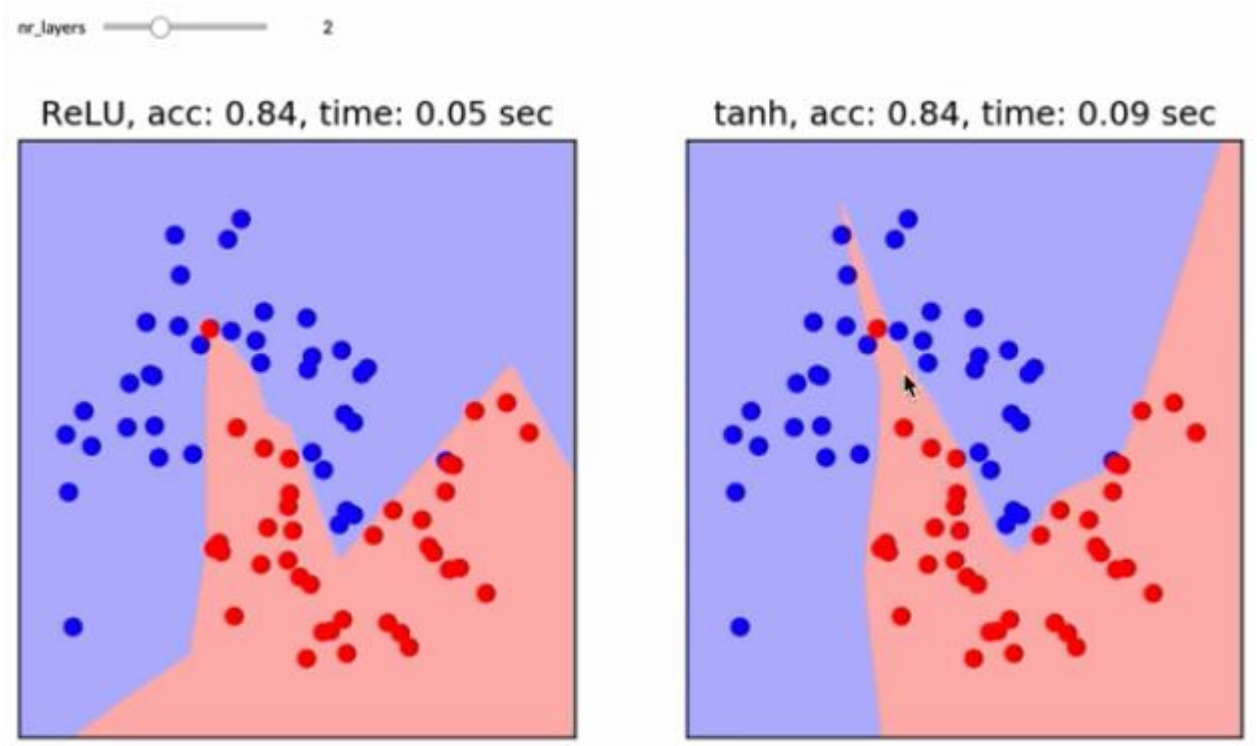
End of chapter 8

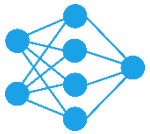
Hamidreza Baradaran Kashani

ReLU vs Tanh activations

❑ **What is the effect of using non-smooth activation functions**

- ReLU produces piece-wise linear boundaries, but allow deeper networks
- Tanh produces smoother decision boundaries, but is slower





ReLU vs Tanh activations

❑ **What is the effect of using non-smooth activation functions**

- ReLU produces piece-wise linear boundaries, but allow deeper networks
- Tanh produces smoother decision boundaries, but is slower