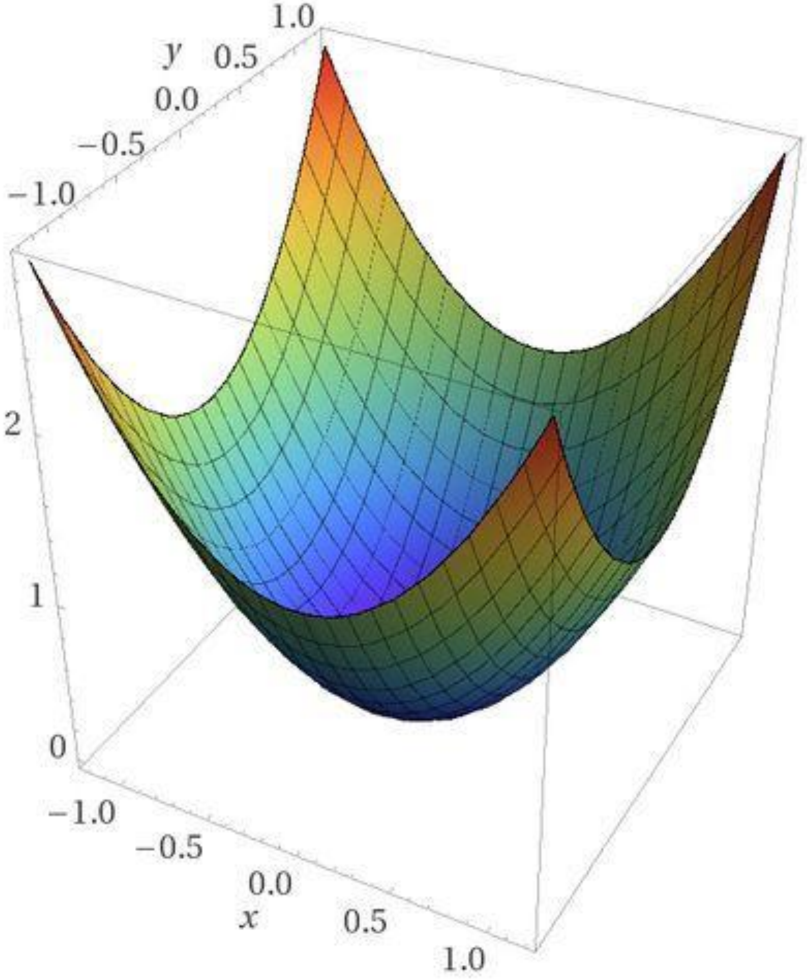
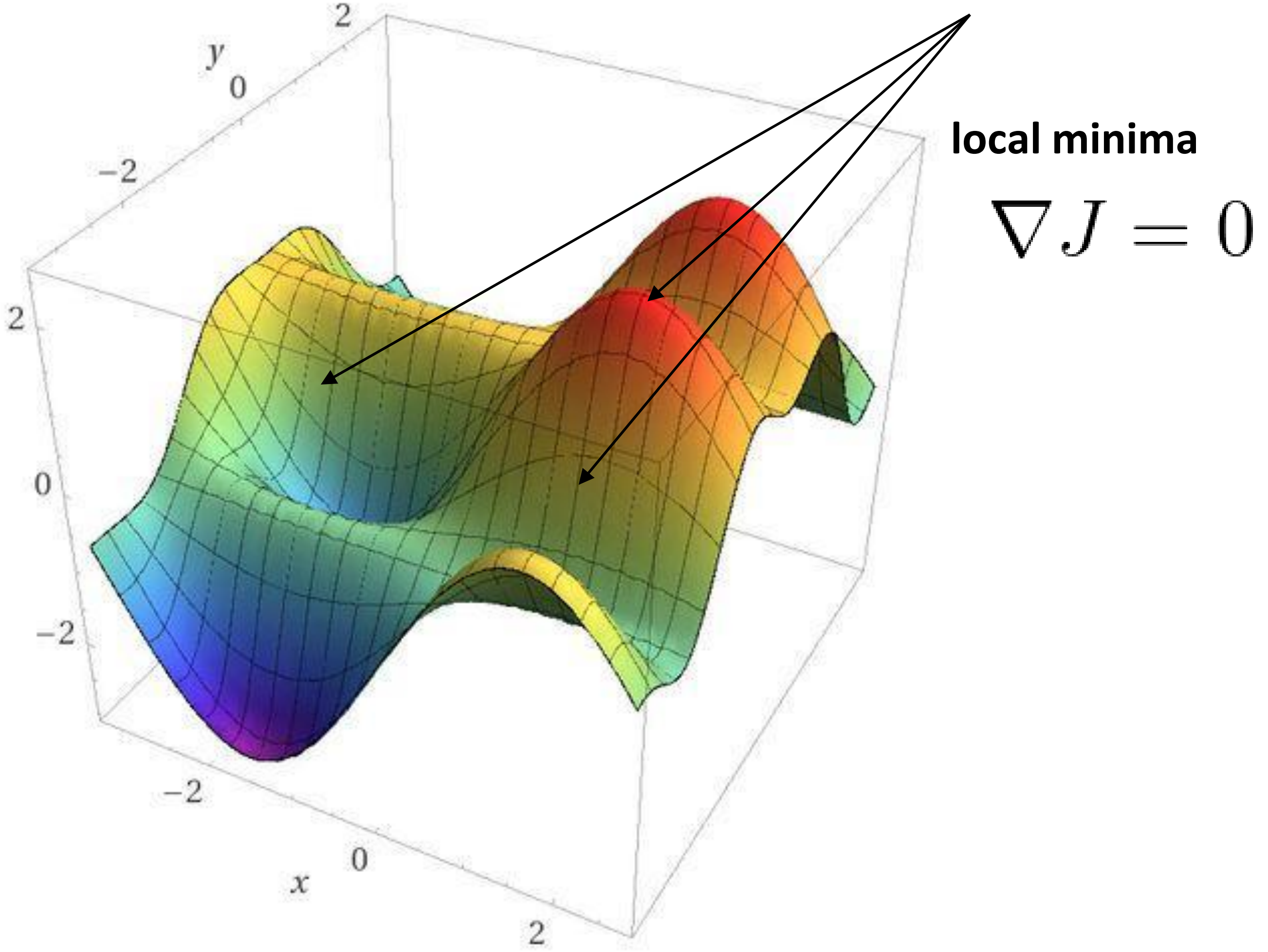
Beyond Backpropagation

Tips and tricks for training deep neural networks

Finding Global Minima: Why is it hard?



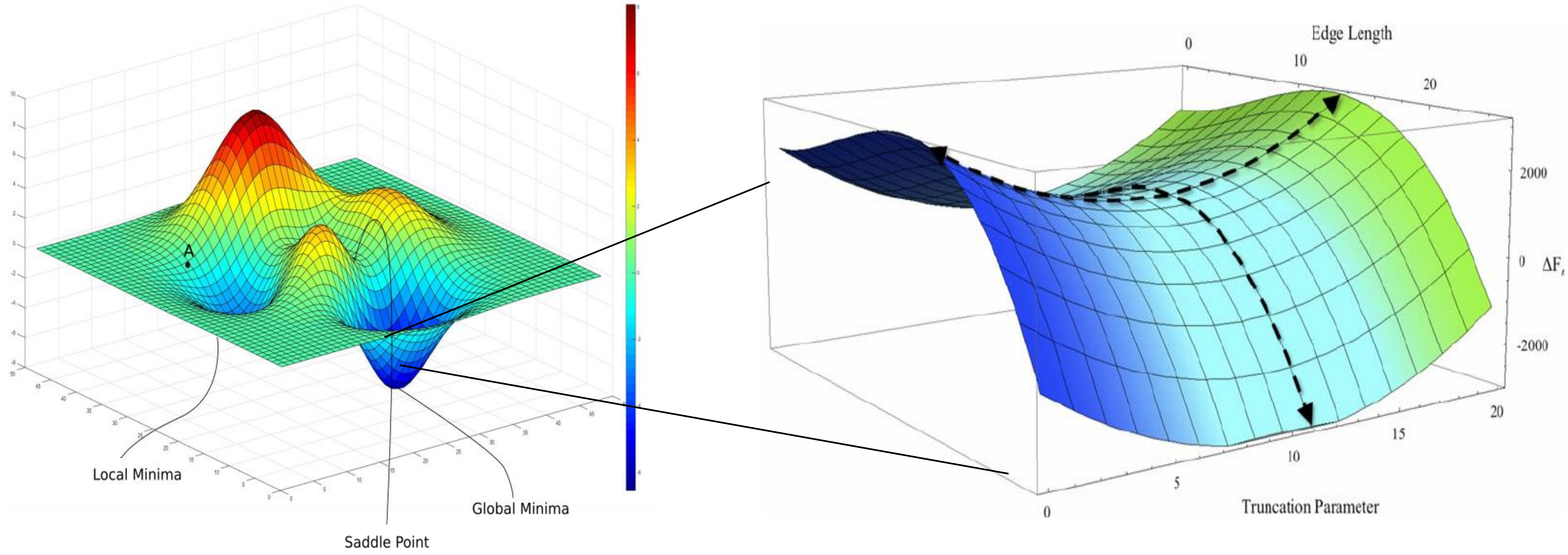
Loss space in our expectation



Loss space in reality

Image Credit: https://blog.paperspace.com

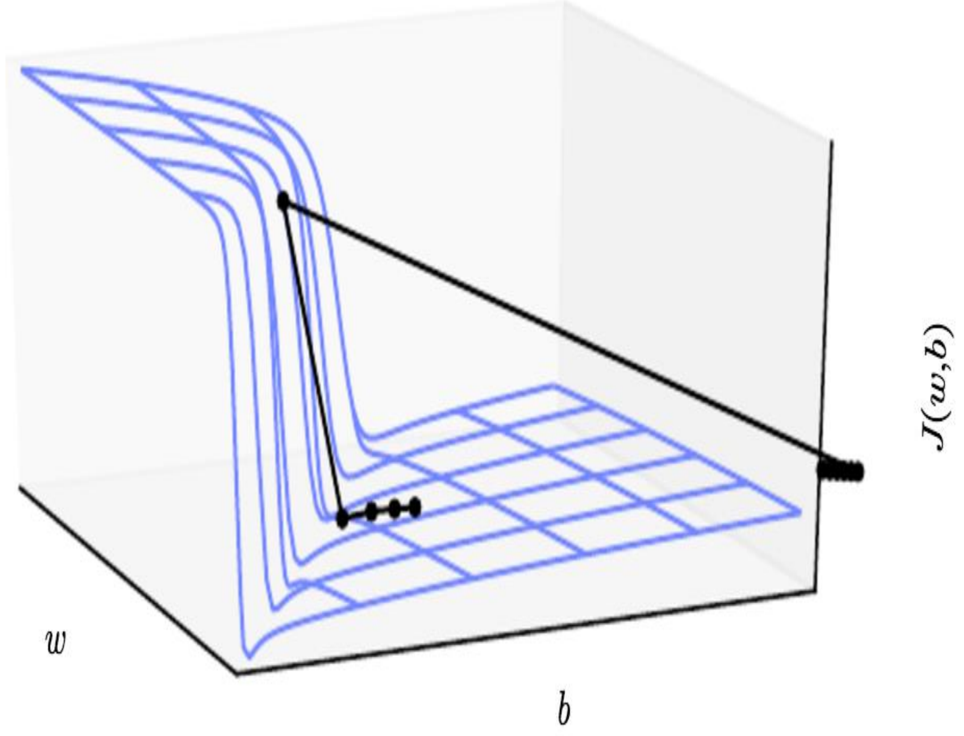
Finding Global Minima: Why is it hard?



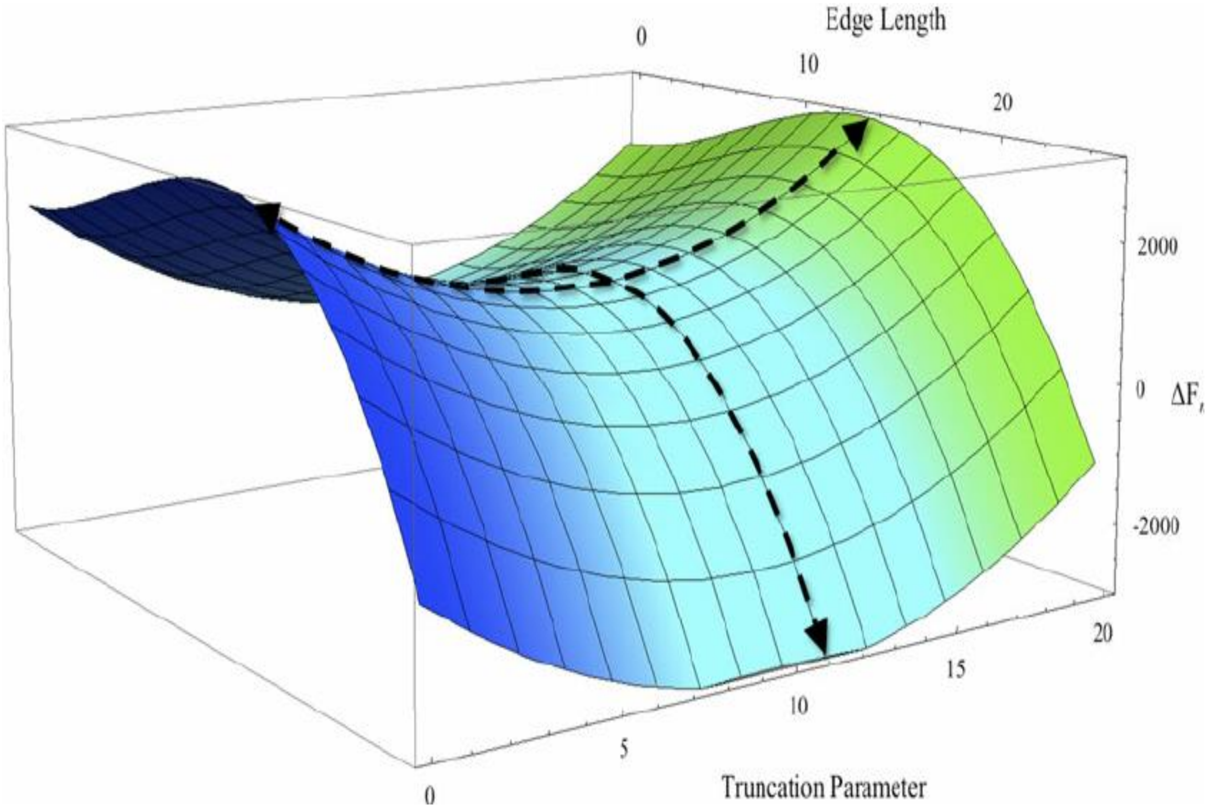
Saddle Point: Local <u>minima</u> in one direction, local <u>maxima</u> in another direction

Image Credit: https://blog.paperspace.com

Finding Global Minima: Why is it hard?



Cliffs : gradient is too high



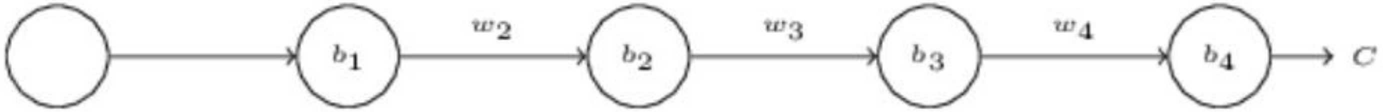
Plateau : gradient is almost zero

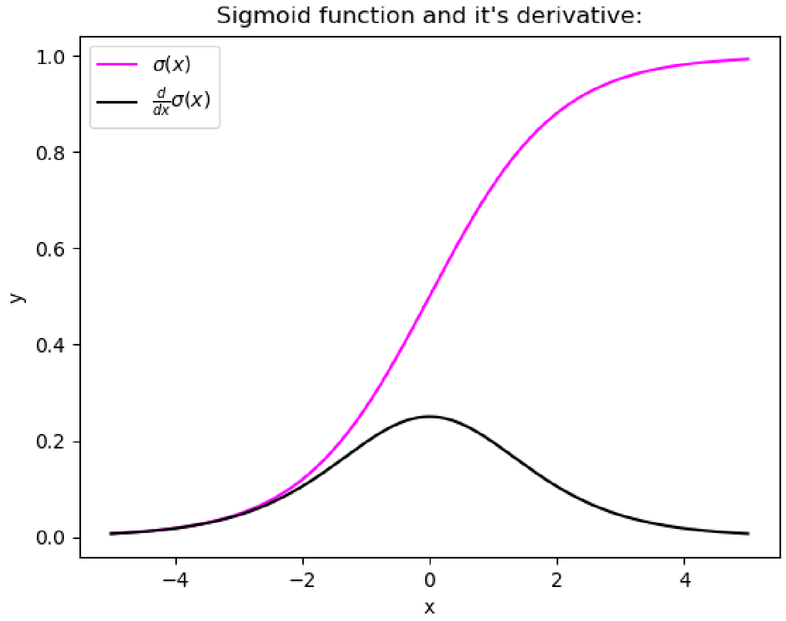
Image Credit: towrdsdatascience.com

Finding Global Minima: Why is it hard?

The Vanishing Gradient Problem:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

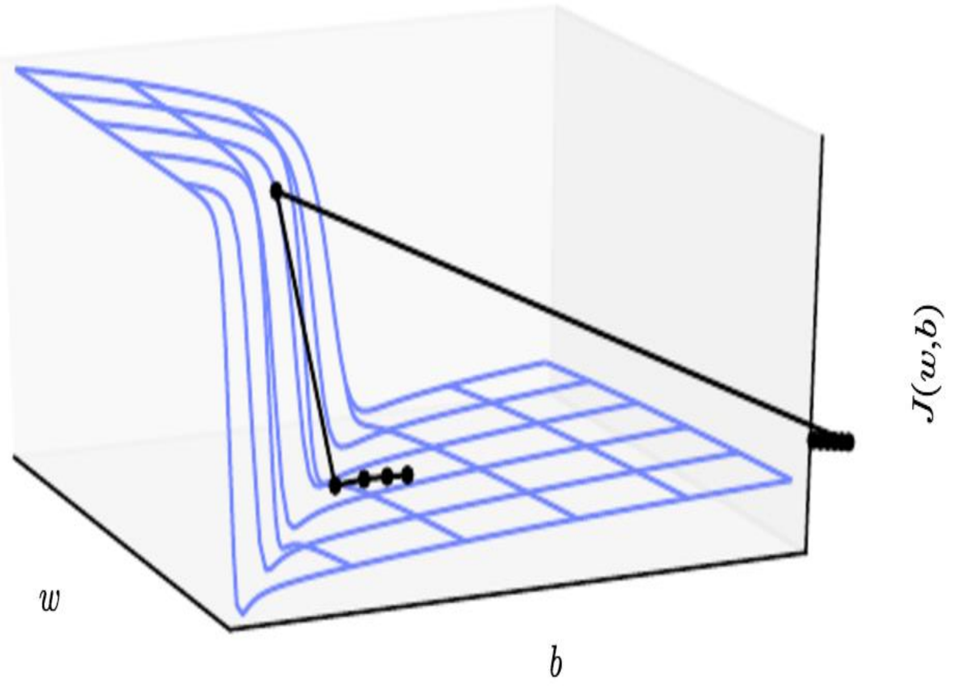




$$|| \frac{\delta h_i}{\delta h_{i-1}} ||_2 < 1$$

Image Credit:machinelearningjourney.com

Finding Global Minima: Why is it hard?



$$||\frac{\delta h_i}{\delta h_{i-1}}||_2 > 1$$

Solution: Gradient clipping

When gradient is too high, the repetitive multiplication results in exploding gradient

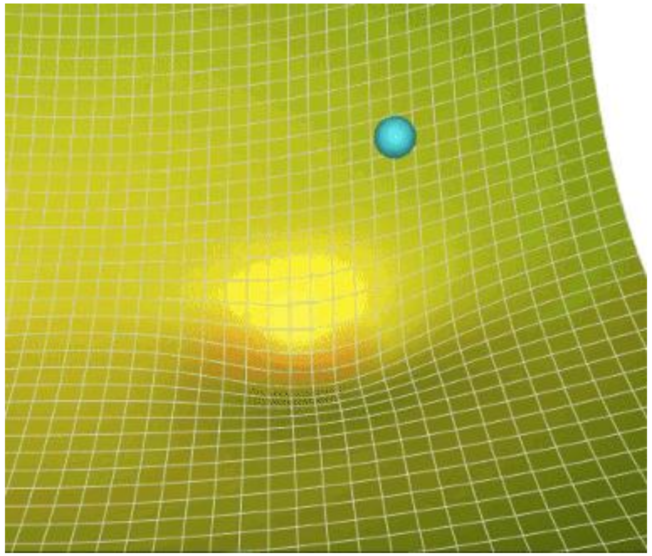
Image Credit: towrdsdatascience.com

Solution – I: Better Optimizers

$$w_{t+1} = w_t - \alpha * \frac{\delta L}{\delta w_t}$$

Ideal optimizer:

- Finds minimum fast and reliably well
- Doesn't get stuck in local minima, saddle points, or plateau region



Vanilla Gradient Descent: One step for the entire dataset

Stochastic Gradient Descent: One step for each stochastically chosen sample

Mini-batch Gradient Descent: One step for each mini-batch of samples chosen stochastically

Best of both worlds!

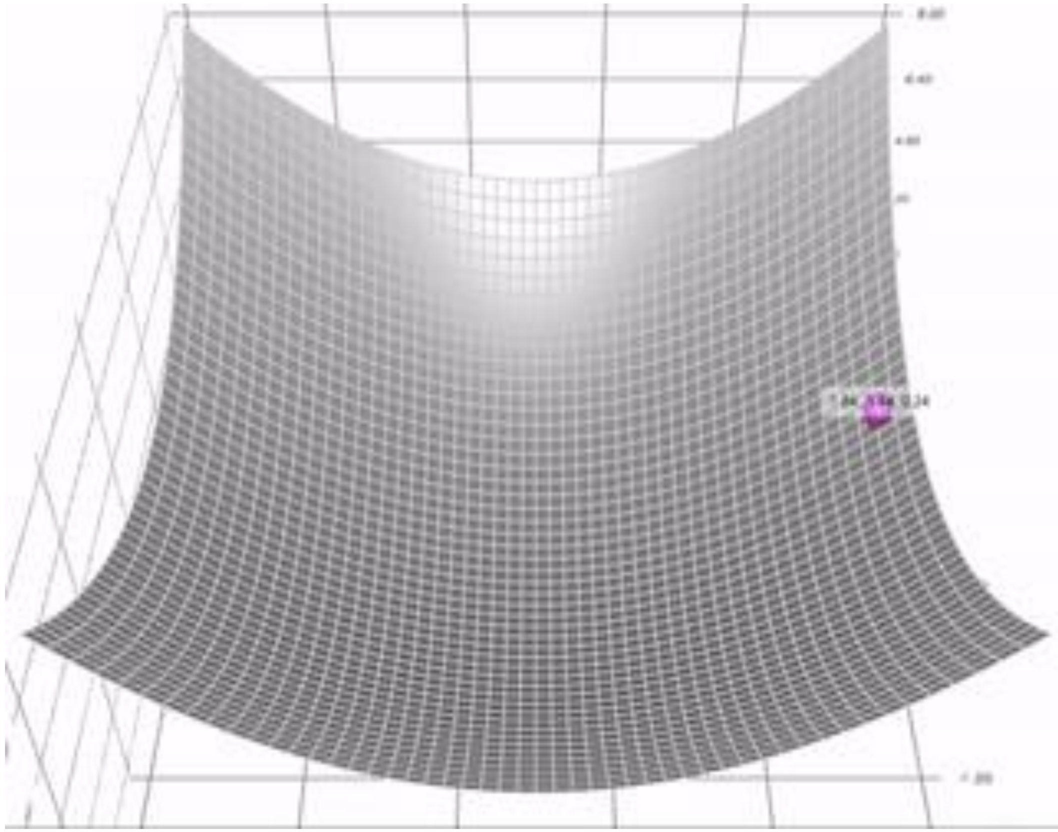
```python
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0)
```

Image Credit: towrdsdatascience.com

Solution – I: Better Optimizers

Gradient Descent with Momentum

- Imagine a rolling down a ball inside a frictionless bowl
- The ball doesn't stop at the bottom of the surface
- Uses the accumulated momentum to go forward



$$v_t = \boxed{\gamma\, v_{t-1}} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

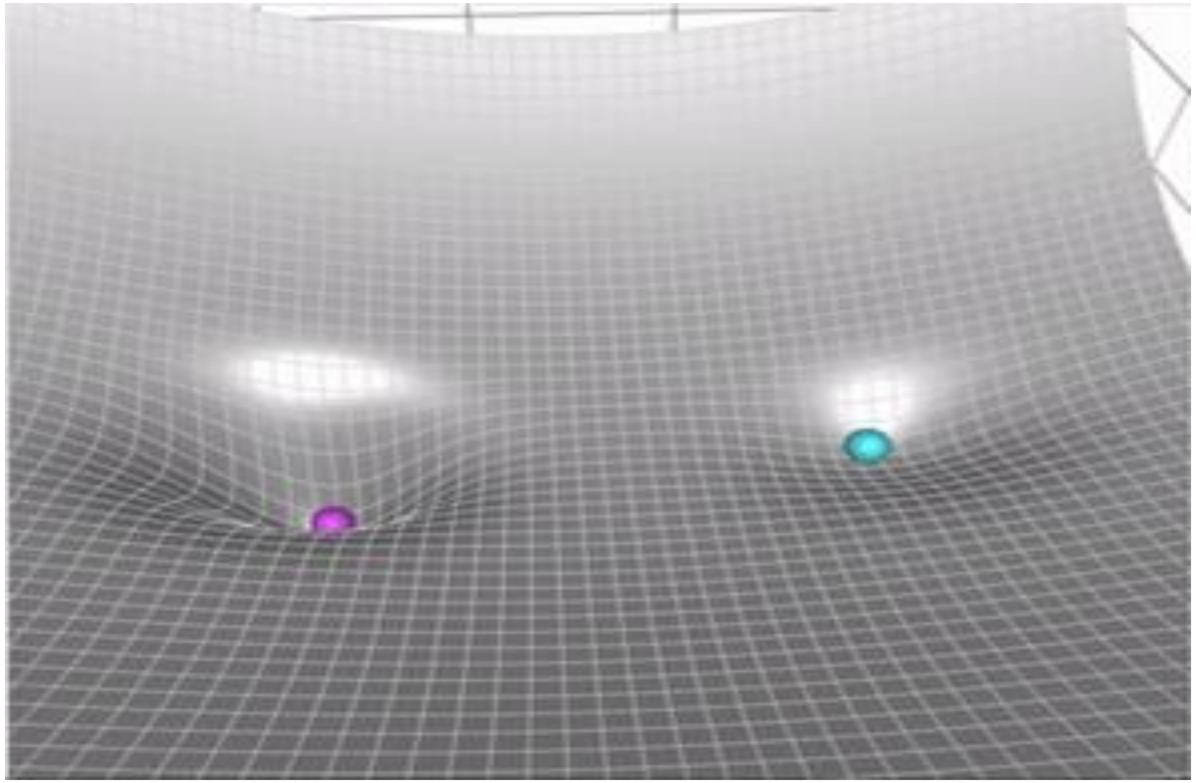
Momentum

Image Credit: towrdsdatascience.com

Solution – I: Better Optimizers

Gradient Descent with Momentum

Intuitively, this helps us to come out of local minima



$$v_t = \underbrace{\boxed{\gamma\, v_{t-1}}}_{\text{Momentum}} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

```
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

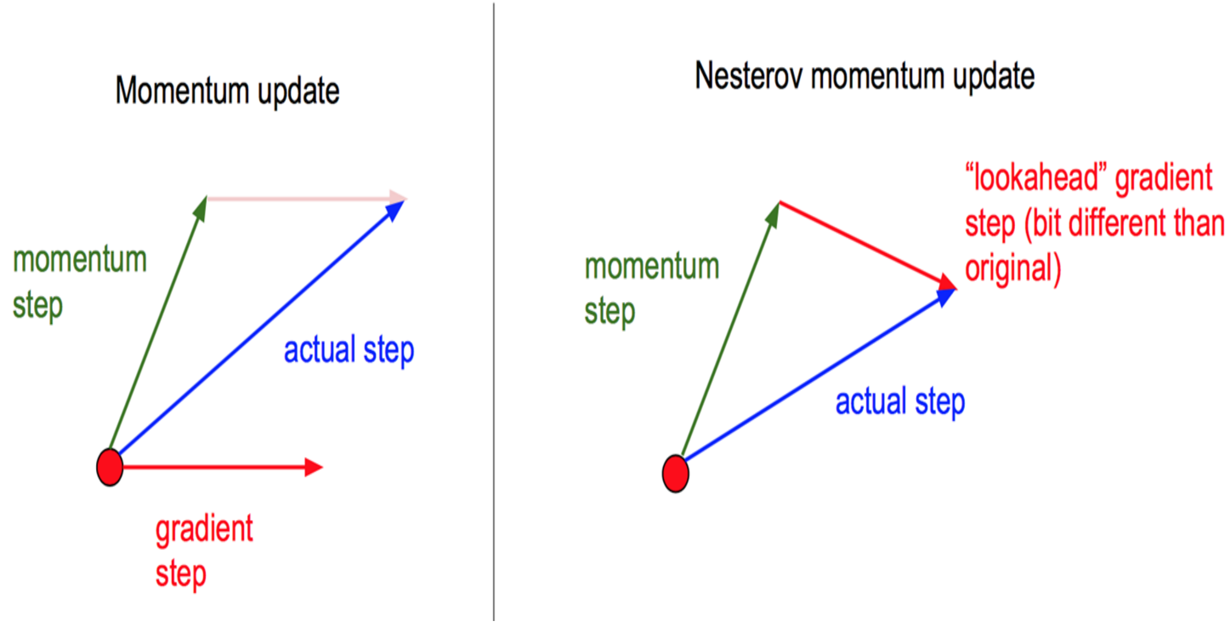
Image Credit: towrdsdatascience.com

Solution – I: Better Optimizers

Gradient Descent with Nesterov Momentum

- Following the slope blindly is not desirable and optimal
- The ball should predict and slow itself down before going up again

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta \boxed{- \gamma v_{t-1}})$$
$$\theta = \theta - v_t$$




```python
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, nesterov=True)
```

Image Credit: towrdsdatascience.com

Solution – I: Better Optimizers

Other Variants:

- RMSprop
- Adagrad
- **Adam**
- Adadelta
- etc.

```python
import torch.optim as optim

optimizer = optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99)
```

```python
import torch.optim as optim

optimizer = optim.Adagrad(model.parameters(), lr=0.01)
```

```python
import torch.optim as optim

optimizer = optim.Adam(model.parameters(), lr=0.01, betas=(0.9, 0.999))
```

```python
import torch.optim as optim

optimizer = optim.Adadelta(model.parameters(), lr=0.01, rho=0.9)
```

Simplified View: Variable Learning Rates for Features/Dimensions

Solution – II: Learning Rate Scheduling

- Adjust the learning rate during training by reducing the learning rate at per predefined scheduled

- Common schedulers –
  - step decay
  - exponential decay
  - cosine decay
  - reduce on plateau
  - etc.

```python
import torch.optim as optim


scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[30,80], gamma=0.1)
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.1)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode = 'min')
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer)
```

Solution – III: Weight Initialization

- All zero initialization
  - Initializing all the weights with zeros leads the neurons to learn the same features during training.
- Random initialization
  - Gaussian
  - Xavier
    - uniform
    - normal
  - Kaiming
    - uniform
    - normal

```python
import torch

w = torch.empty(3, 5)
torch.nn.init.kaiming_uniform_(w, mode='fan_in', nonlinearity='relu')
```

Summary of Training (Recap)

- Backpropagation Algorithm
  - Gradient Descent
  - Non-Convex Optimization (in NN)
  - Generic Algorithm
- Applicable for
  - MLP
  - CNN
  - RNN
  - A wide class of networks with differentiable units

Blank Slide

Training

1) choose arch

2) Initialize

3) FP, BP

4) Update

5) Terminate/End

Summary of Training (Rcap)

- Challenges
  - Non-Convexity, Plateau, Vanishing and Exploding Gradients
- Better Optimizers
  - SGD
  - Momentum
  - Adam
- Learning rate Scheduling
- Better/Smarter Initializations

Blank Slide

* Loss Function

many possibilities

"Regularize"

prefer some over others

NN → give high accuracy with many weights as zero

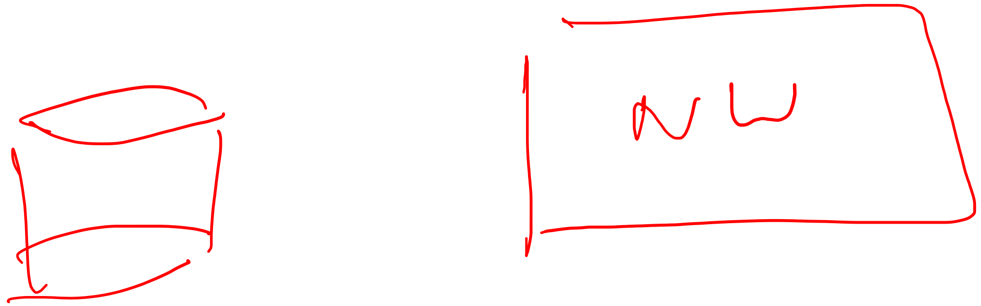
sparse

nonzero weights <<

⇒ Model is compact

Blank Slide

↑ Data ⟷ # parameters



Regularization of the Network

- Input Level
  - Data Augmentation

- Activation level
  - Dropout
  - Dropconnect

- Feature statistics level
  - Batch normalization
  - Layer normalization
  - Group normalization

- Decision level
  - Ensemble

- Constraining network weights
  - $\ell_1$norm, ℓ_2 norm

- Terminating early based of the performance on validation set
  - Early stopping

Regularization at the Input Level

A trick to increase the training data

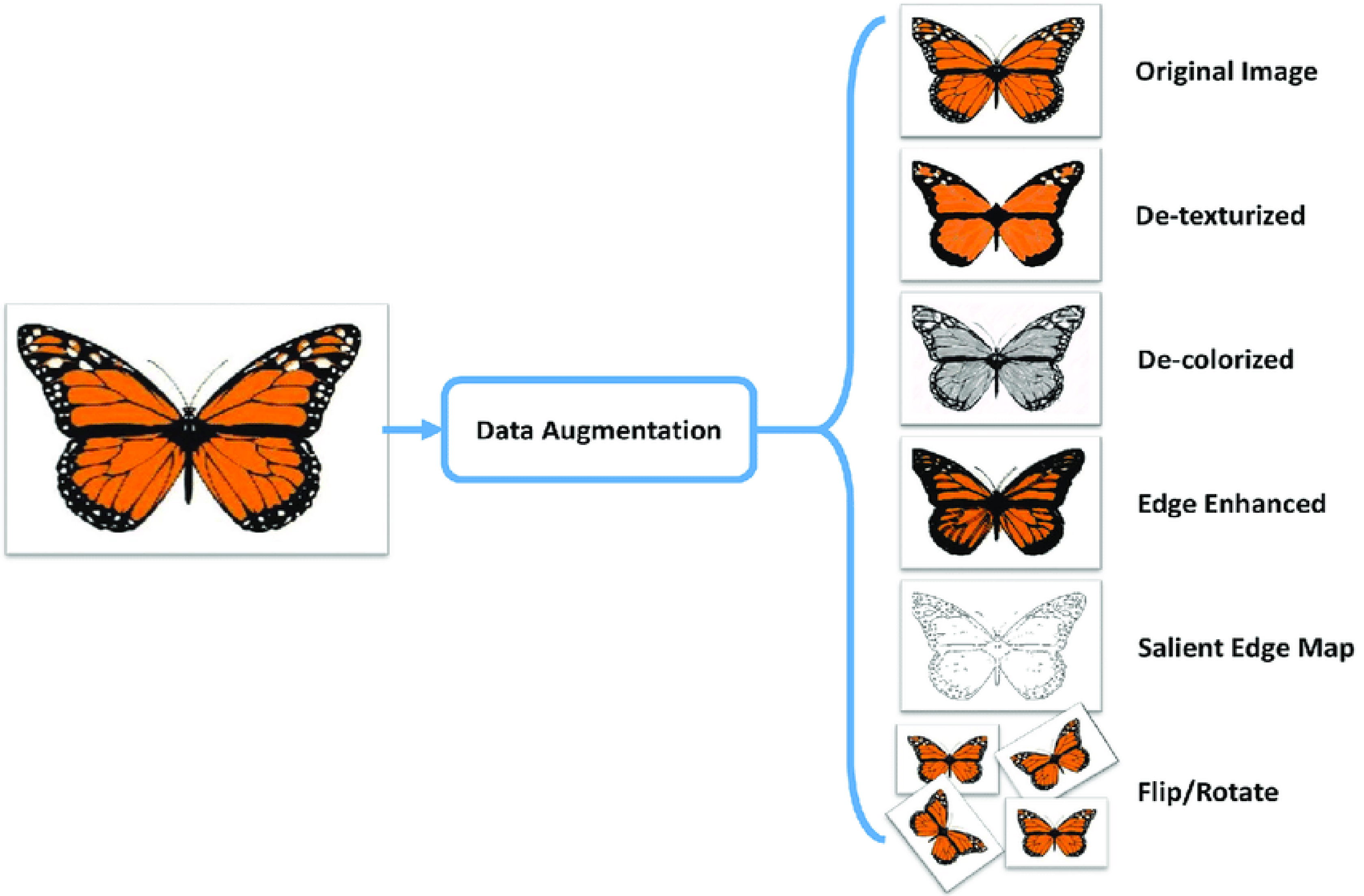
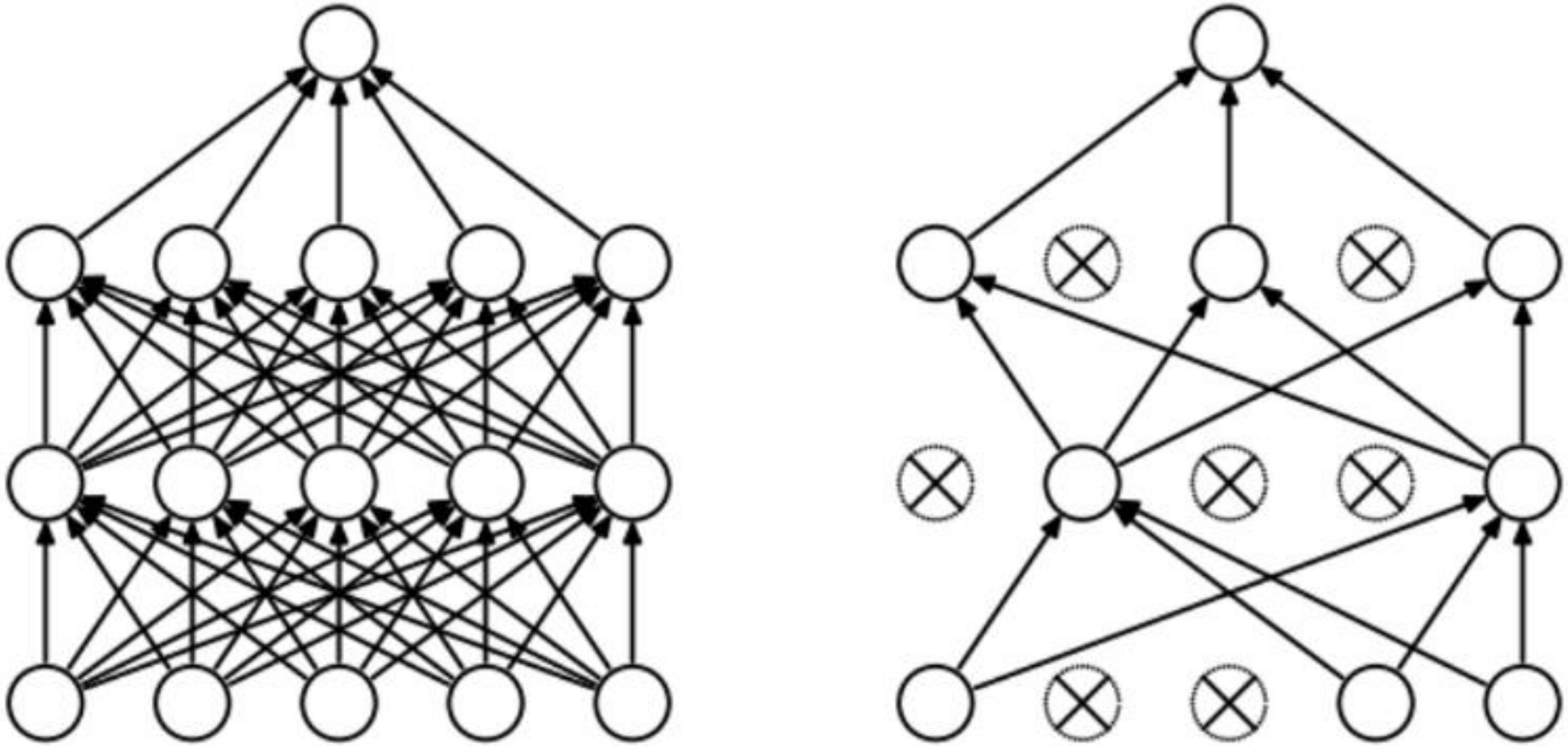


Image Credit: medium.com

Dropout

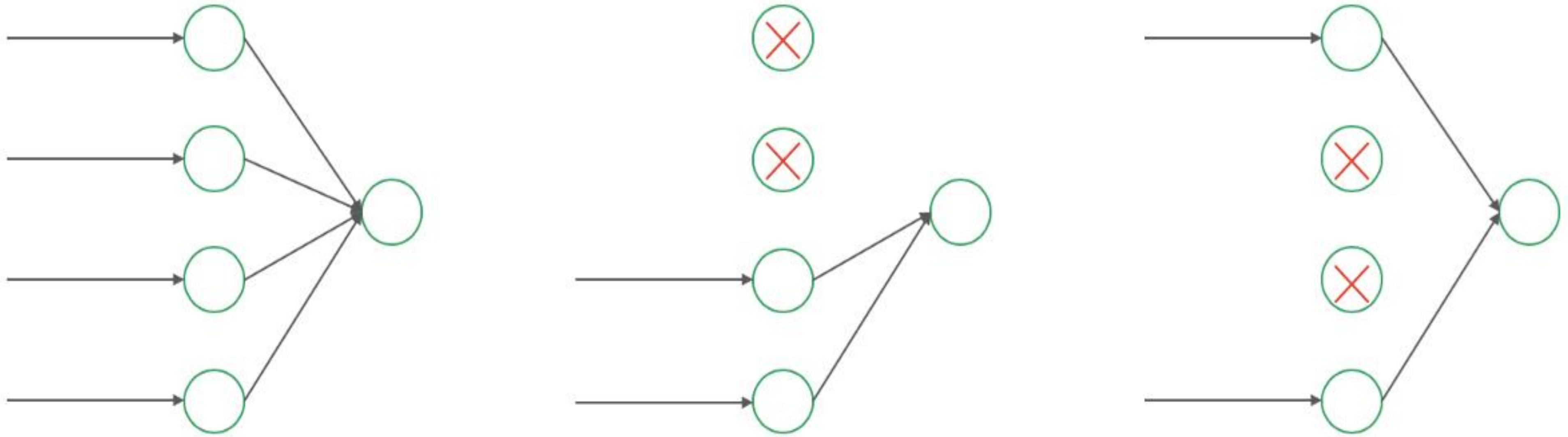


(a) Standard Neural Net

(b) After applying dropout.

Regularization at the Activation Level

Dropout:



a. Full network

b. Partial learning of weights over iterations

At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p

Image Credit: towrdsdatascience.com

Blank Slide

Dropout

- Ignore/delete/mask certain neurons while training.
  - Get a simpler network.
  - Eg. Multiply the outputs by 0 or 1 at random.
- Equivalent to creating many neural networks.
- Reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons.
- Force to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

Dropout

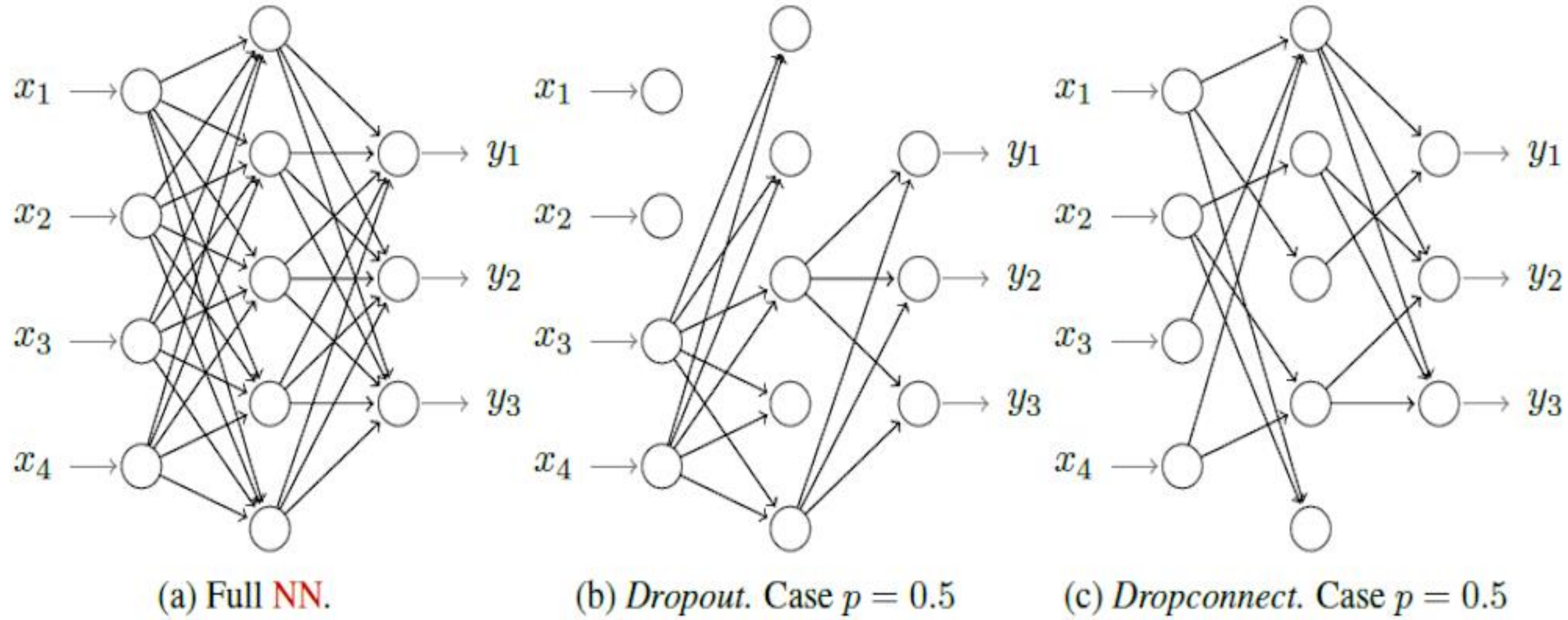
- Set the output of each hidden neuron to zero with a probability of p (say 0.5).
- The neurons which are “dropped out” in this way do not contribute to the forward pass and do not participate in back propagation.
- Thus neural network samples a different architecture, but all these architectures share weights.
- At test time, scale outputs by probability p.

Drop-Connect

- Drop-connect (Wan et al 2013) is very similar to dropout.
- Randomly deactivates the network weights instead of randomly reducing the neuron activations to zero.
- Performs mask out operations on weights (instead of outputs).

Regularization at the Activation Level

Dropconnect:



At each training stage, disable individual weights (i.e., set them to zero), instead of nodes, so a node can remain partially active

Image Credit: jaronsanders.nl

Regularization at the Feature Statistics Level

BatchNorm:

- We normalize the input data before feeding to the NNs
- What about the input distribution at each hidden layer?
    - the distribution changes with iterations
    - internal co-variate shift
    - Solution: Normalize the layer's input over a mini-batch

```python
import torch
import torch.nn as nn

m = nn.BatchNorm2d(100, affine=False)
input = torch.randn(20, 100, 35, 45)
output = m(input)
```

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

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

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Weight Decay, Constraining Weights

- Add a term corresponding to weights into the objective function.
- Smaller the weight (or even zero), the better.

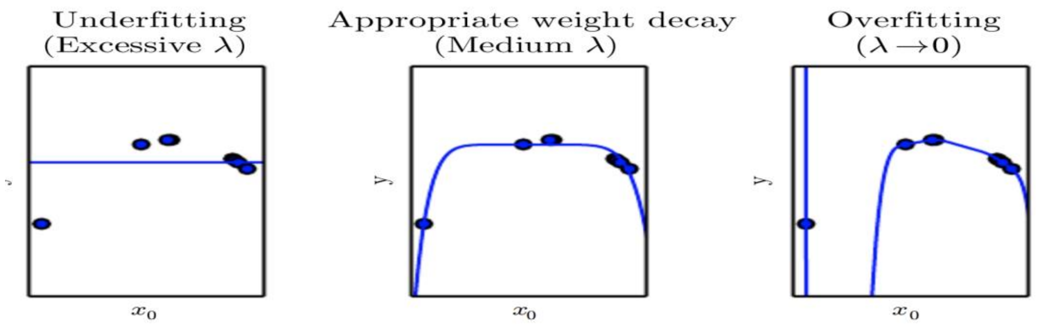
$$J(w) = \mathcal{L} + \lambda ||w||_p$$

L1 Regularization

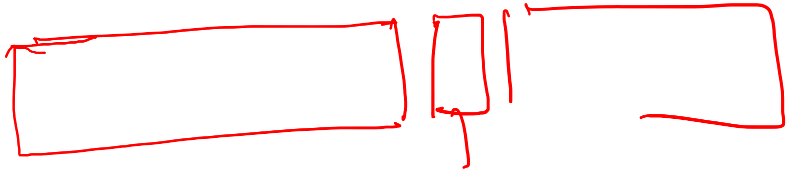
$$\text{Cost} = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

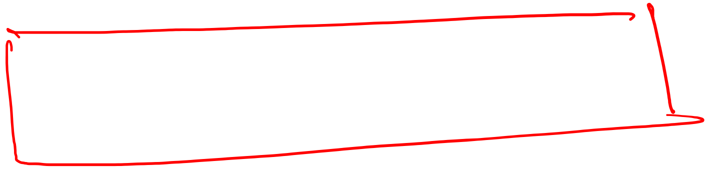
L2 Regularization

$$\text{Cost} = \underbrace{\sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2}_{\text{Loss function}} + \lambda \underbrace{\sum_{j=0}^{M} W_j^2}_{\text{Regularization Term}}$$



Blank Slide





a Compute mean

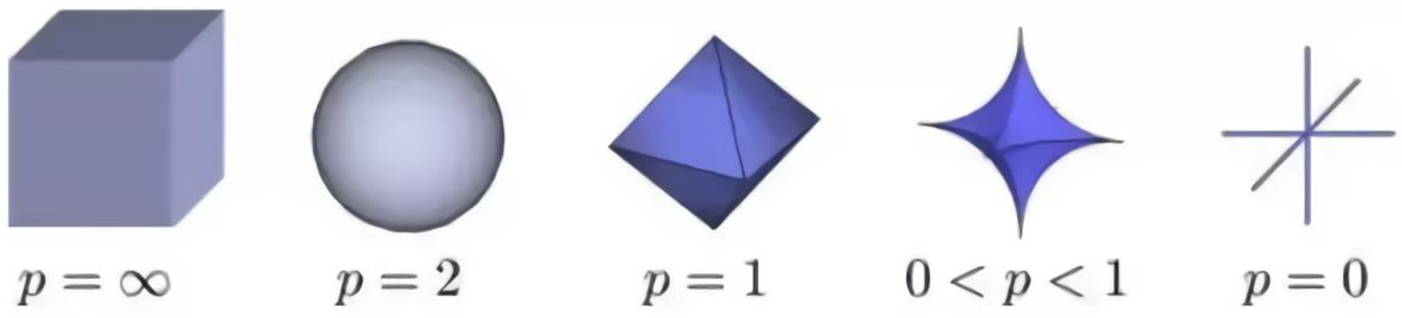
b Subtract mean

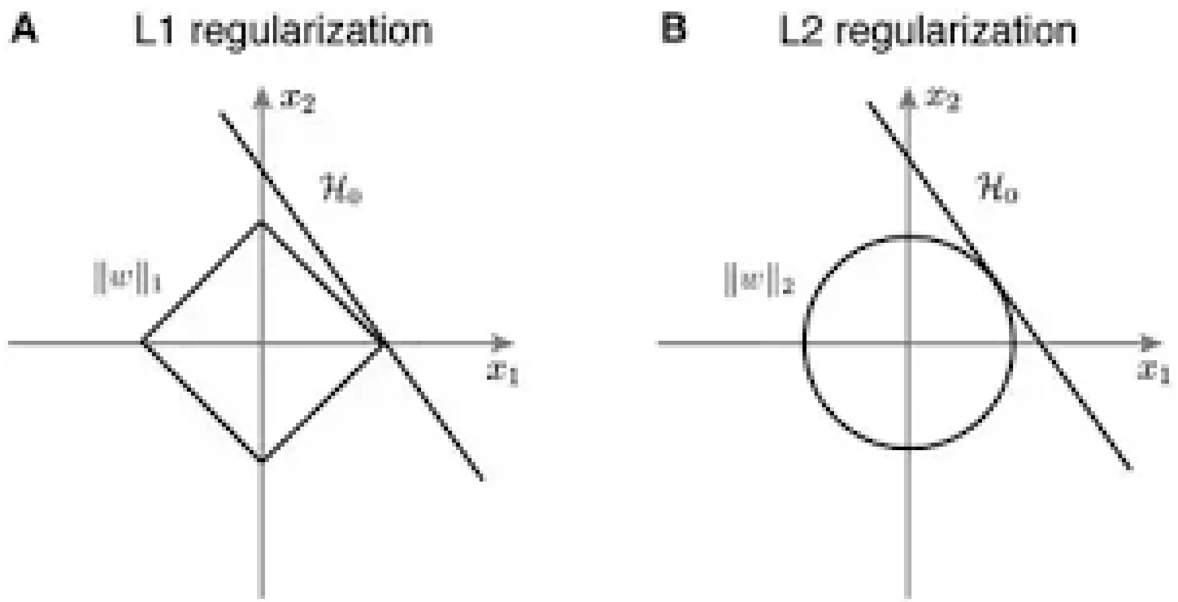
* Train & get weights → Allow to decay

* make small wts → 0

* Retrain → sparse n/w

Norm and Sparsity





L1 is likely to yield sparse solutions

Blank Slide

Early Stopping

- Stop the training if validation loss/accuracy not improving
- Avoids overfitting on the training data

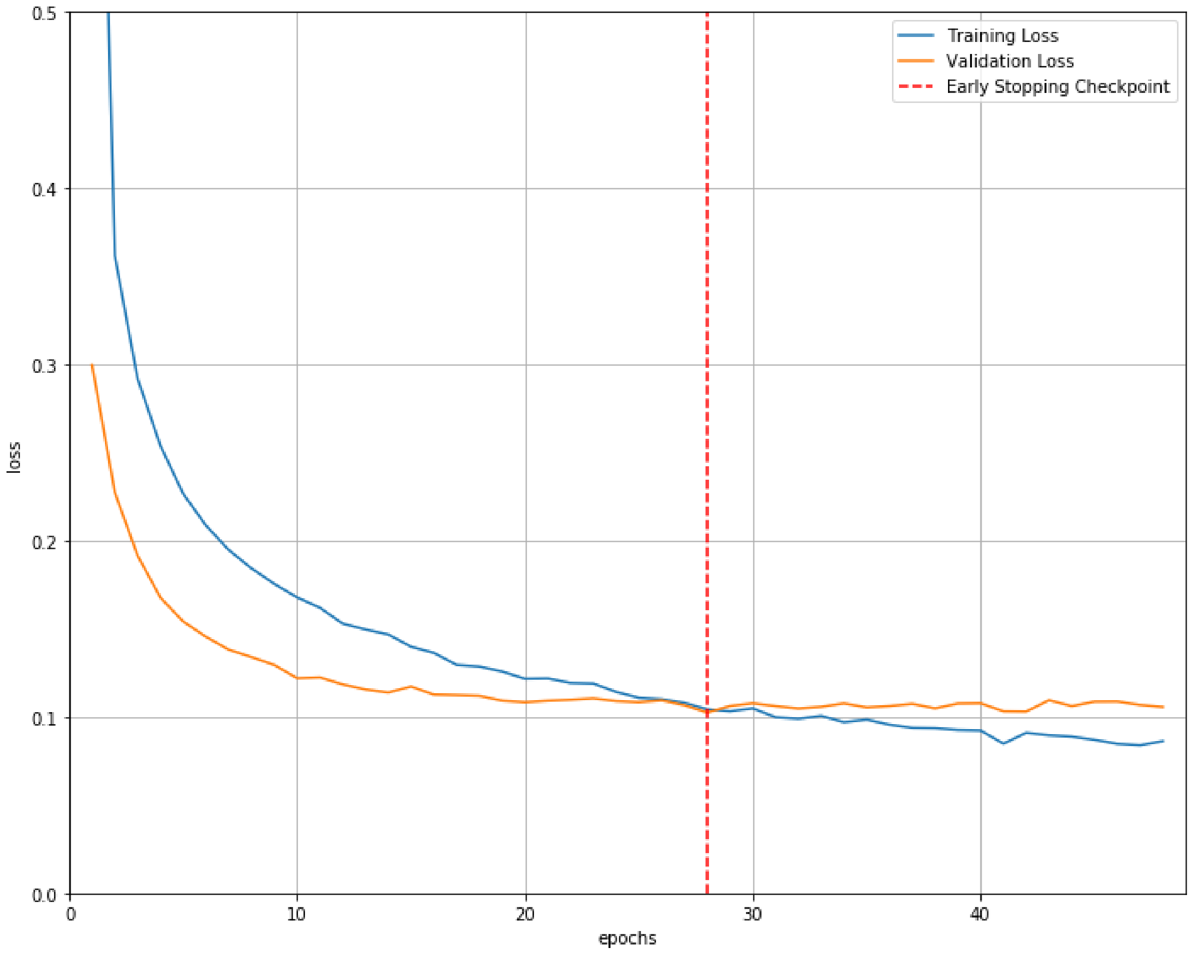


Image Credit: stats.stackexchange.com

Blank Slide

Reg minimizes overfitting

- * More data (Add noise, crop, & ...)
- * normalize
- * Robustify (Dropout)
- * Early stop
- * Add extra ~~constr~~ terms in $L + \lambda \|w\|_p$

Summary

- Data Normalization
- Data Augmentation
- Weight Initialization
- Optimization Algorithms
- Regularizer
- Batch Norm
- Dropout

Many Improvements

- Better Learning/Optimization
- Better Generalization/Regularization
- Better Loss Functions

Thanks!!

Questions?