Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

Lecture 5: Training and Optimization

Instructors: George Em Karniadakis, Khemraj Shukla, Lu Lu
Teaching Assistants: Vivek Oommen and Aniruddha Bora

Course Roadmap

Module-1 (Basics)

- Lecture 1: Introduction
- Lecture 2: A primer on Python, NumPy, SciPy and *jupyter* notebooks
- Lecture 3: Deep Learning Networks
- Lecture 4: A primer on TensorFlow and PyTorch
- Lecture 5: Training and Optimization
- Lecture 6: Neural Network Architectures

Module-2 (PDEs and Operators)

- Lecture 7: Machine Learning using Multi-Fidelity Data
- Lecture 8: Physics-Informed Neural Networks (PINNs)
- Lecture 9: PINN Extensions
- Lecture 10: Neural Operators

Module-3 (Codes & Scalability)

- Lecture 11: Multi-GPU Scientific Machine Learning

Contents

Problem definition: Stationary points

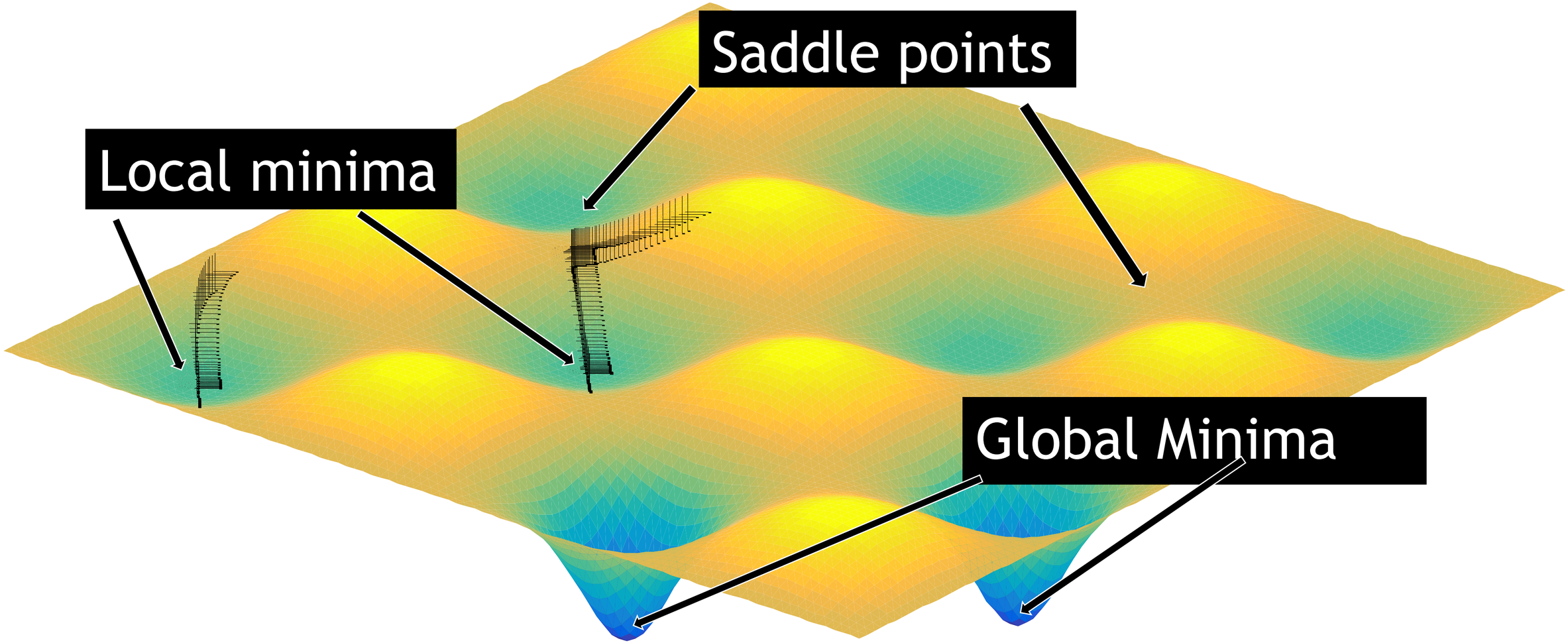
- Stationary points

$$\nabla \mathcal{L}(\theta) = 0 \text{ minimization w.r.t. } \theta$$

- $\mathcal{L}(\theta - \eta \nabla \mathcal{L}(\theta)) = \mathcal{L}(\theta) - \eta ||\nabla \mathcal{L}(\theta)||_2^2 + \mathcal{O}(\eta^2)$
- A gradient-based step decreases the loss value with sufficiently small $\eta > 0$ if $\nabla \mathcal{L}(\theta) \neq 0$
- Converge to a stationary point under the mild conditions (s.t. $\eta \not\to 0$ and not diverge)
- A gradient-based method is a greedy local search method that can avoid the curse of dimensionality (CoD)

Roughly, if every stationary point is "good", we can get a "good" point without the CoD.

Types of Stationary Points



- x-y plane = search space for optimization
- z axis = value of objective function to be minimized

Can we avoid bad local minima for Deep Nets? Yes!

This has been answered in the past 5 years in the literature:

- Deep Learning without Poor Local Minima

  [NeurIPS 2016: Kenji Kawaguchi]

- Elimination of All Bad Local Minima in Deep Learning

  [AISTATS 2020: Kenji Kawaguchi, Leslie Kaelbling]

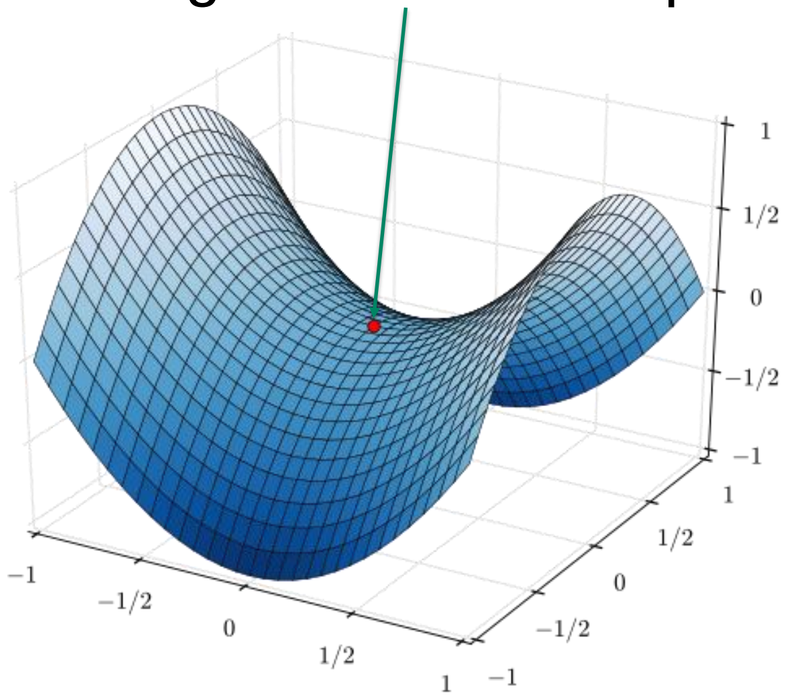
- Depth with Nonlinearity Creates No Bad Local Minima in ResNets

  [NN 2020: Kenji Kawaguchi and Yoshua Bengio]

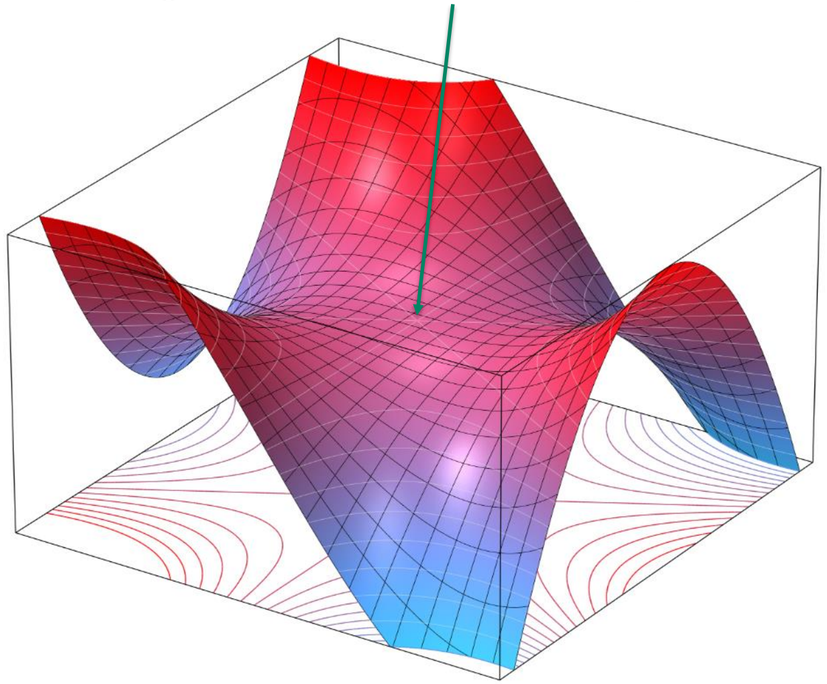
How about avoiding saddle points?

Types of Saddle Points

non-degenerate saddle point



degenerate saddle point



- Degenerate: Hessian is positive semidefinite and has 0 eigenvalue.
- Degenerate saddle points are often more difficult to escape from than non-degenerate saddle points in general

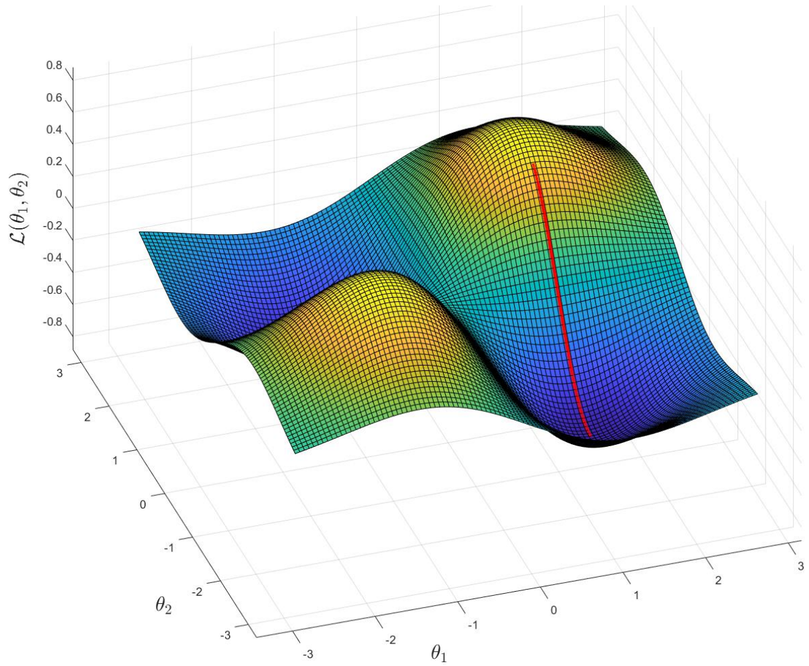
Can we find global minima (by avoiding both bad local minima and saddle points) for Deep Nets? Yes!

- Using the results of no bad local minima, this has been also answered recently in the literature:
  - Allerton 2019: Kenji Kawaguchi and Jiaoyang Huang
  - ICLR 2021: Kenji Kawaguchi
  - ICML 2021: Keyulu Xu, Mozhi Zhang, Stefanie Jegelka, Kenji Kawaguchi
  - AAAI 2021: Kenji Kawaguchi and Qingyun Sun

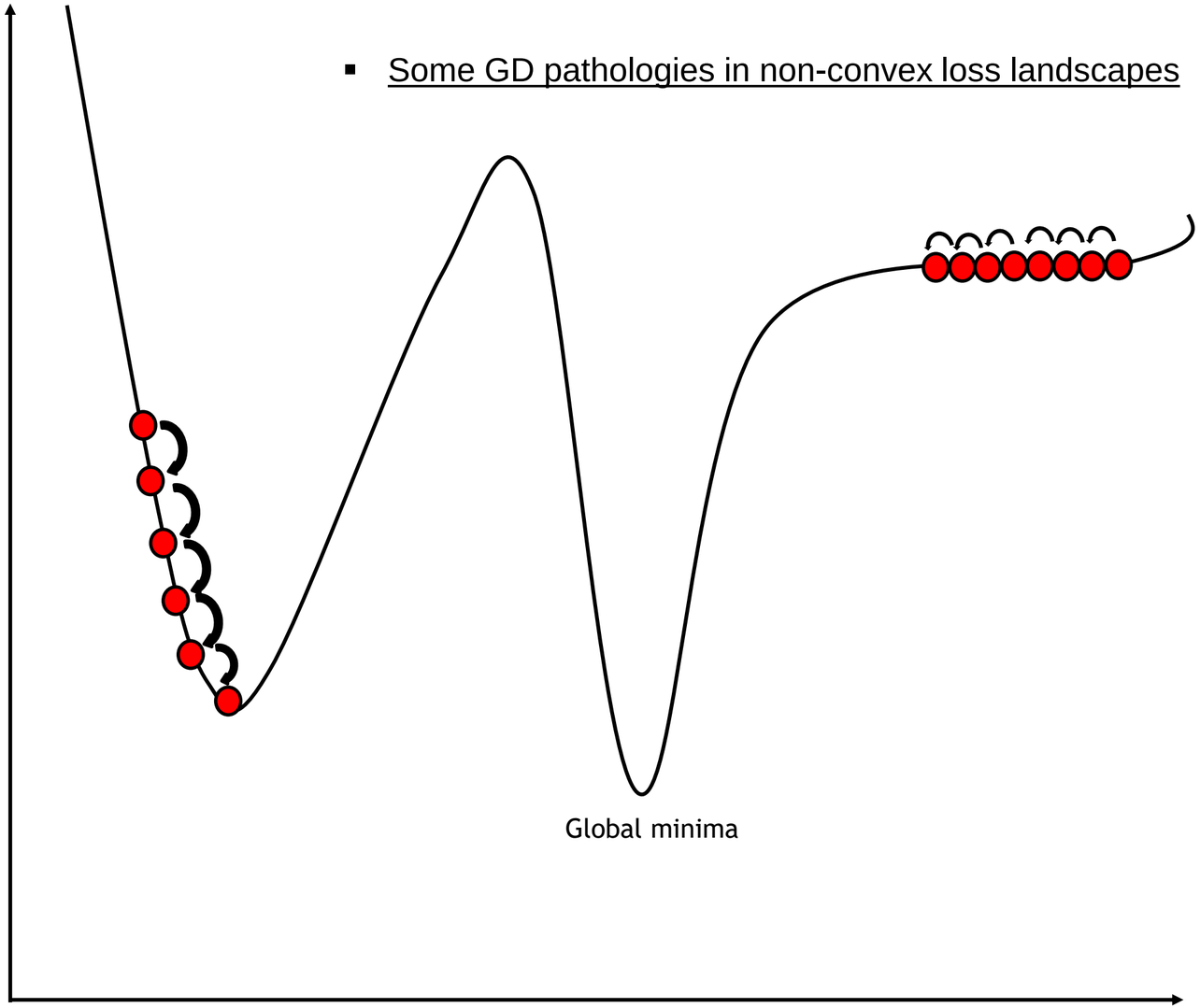
Gradient Descent (GD)

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$



- Some GD pathologies in non-convex loss landscapes



```
tf.keras.optimizers.SGD(learning_rate=0.01)
```

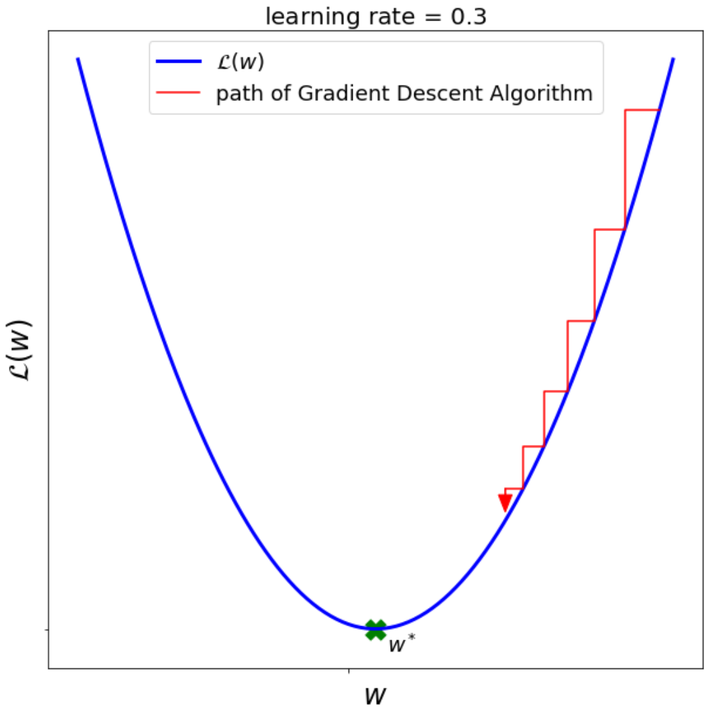
```
torch.optim.SGD(params, lr=0.01)
```

Gradient Descent (GD) vs Stochastic GD (SGD)

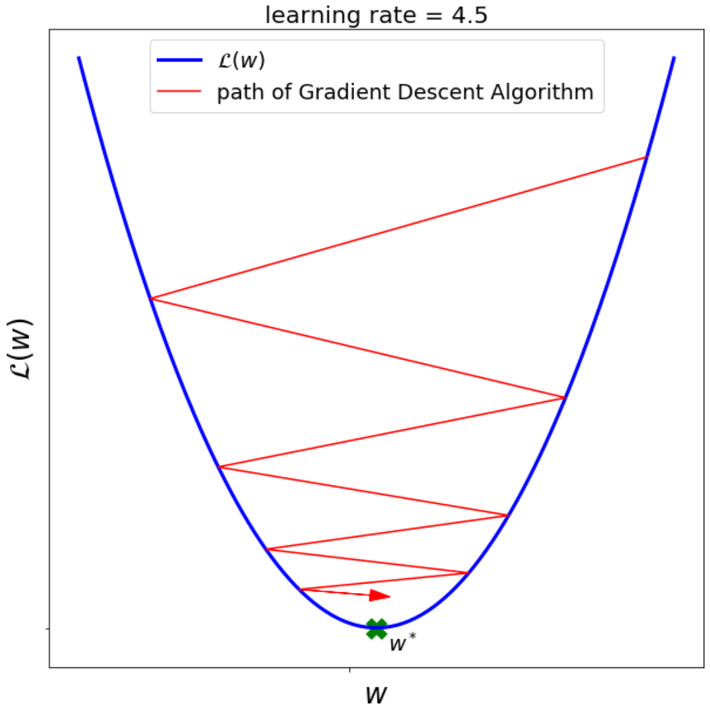
- Unlike batch/full GD that involves computations over the entire training set X, SGD takes only one instance at a time. The training instances must be i.i.d. (independent and identically distributed) to ensure convergence to global minimum. Need to pick each instance randomly or shuffle the training set at each epoch in order to reach the global minimum.
- SGD is analogous to ADI in Computational Fluid Dynamics that involves direction splitting.
- The randomness in SGD arising from weight update after every instance can help escape from bad local minima of irregular and non-convex loss functions.
- However, this randomness is also bad as SGD will not fully converge, it will wander around the minimum.
- An effective solution in practice is to introduce **decaying learning rate** via a learning schedule (like simulated annealing).
- Another solution is **mini-batch GD**: weight update based on a loss computed from a random subset of instances (not just one, like SGD!).
- Mini-batch GD is computationally friendly to GPUs (for matrix operations) so it has good performance.

Effect of Learning Rate

❑In linear regression we have convexity (hence global minimum) but still we should scale all features for faster convergence

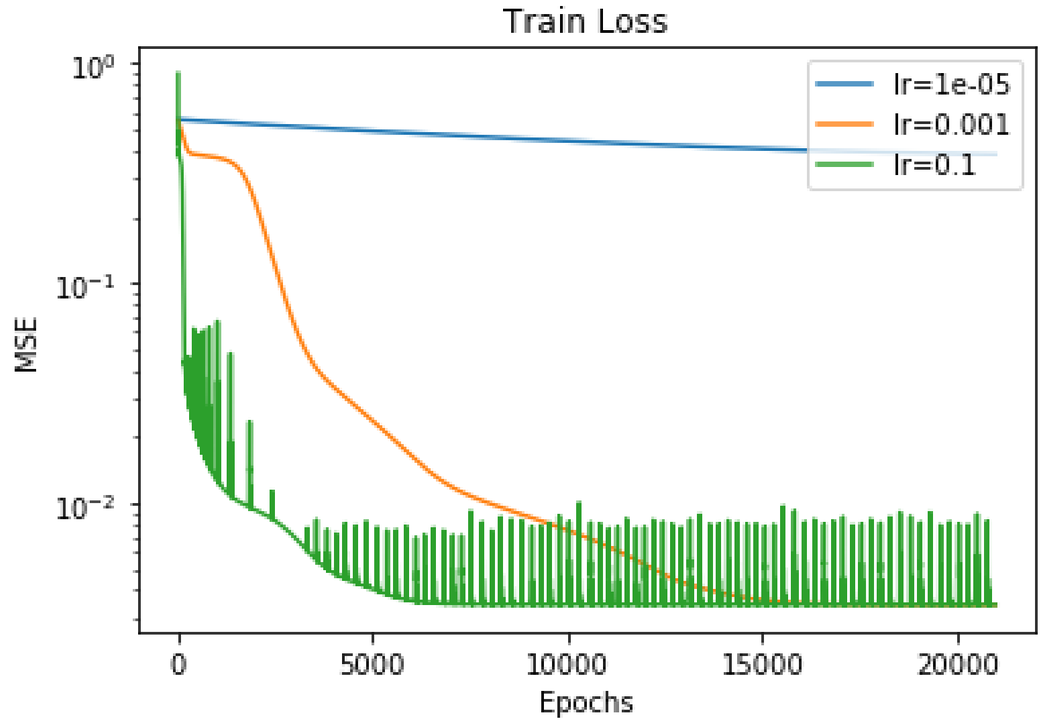


Learning rate is too small



Learning rate is too large

Loss plot associated with learning the sine function with noise using a fully connected neural network



Convergence depends strongly on the lr

❑ **An effective strategy is to use a variable/decaying learning rate**

Practical tips in training a DNN

- Vanilla mini-batch gradient descent does not guarantee good convergence unless several challenges are resolved:

- Choosing a proper learning rate can be difficult.

- Learning rate schedules (i.e., adjusting the learning rate during training) has to be defined in advance and it is thus unable to adapt to a dataset's characteristics.

- The same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent but perform a larger update for rarely occurring features.

- Another key challenge of minimizing highly non-convex loss functions for neural networks is avoiding getting trapped in their numerous suboptimal local minima. *Dauphin et al*. argue that the difficulty arises not from the local minima but from the saddle points. These saddle points are usually surrounded by a plateau of the same error, which makes it hard for SGD to escape as the gradient is close to zero in all directions.

- Ruder, S. (2016).An overview of gradient descent optimization algorithms. arXiv preprint arXiv: 1609.04747
- Dauphin, Y., et al. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1-14

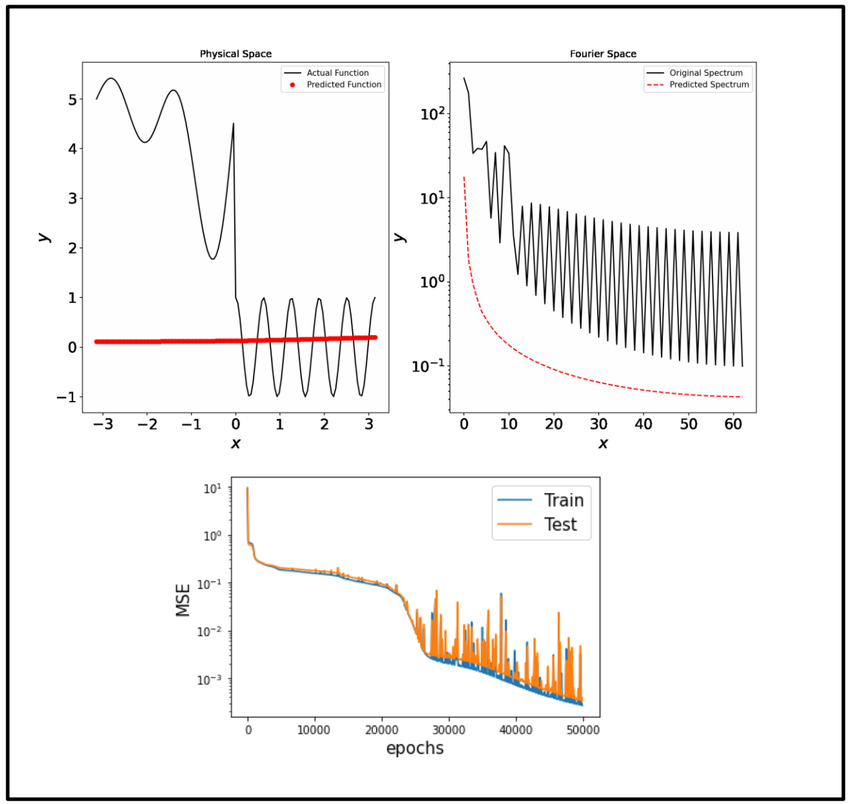
Practical tips in training a DNN

- ❑ Not enough training data or noisy data that can lead to overfitting.
- ❑ An overparametrized DNN may lead to overfitting – regularization is crucial!
- ❑ Training becomes too slow, trapped in a local minimum.
- ❑ Vanishing/Exploding gradients, hence deep networks are difficult to train.
- ❑ Spectral bias that can make multi-scale problems extremely hard to learn.
- ❑ Dying ReLU in deep but narrow networks.

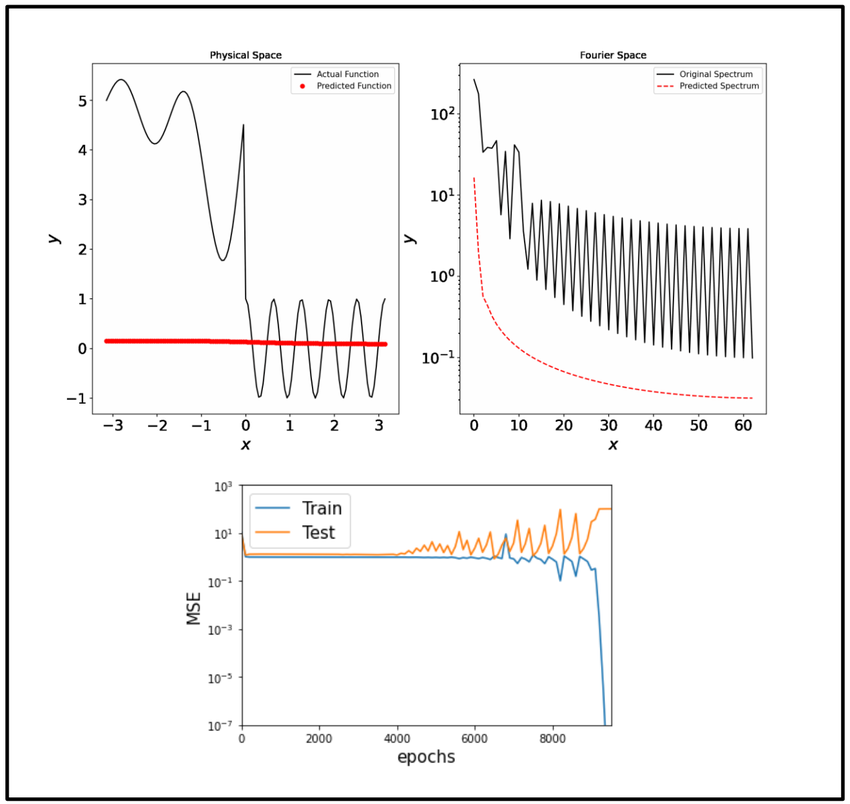
Occam's razor (c. 1287-1347). This principle states that among competing hypotheses that explain known observations equally well, one should choose the "simplest" one. Statistical learning theory provides upper bounds on generalization error; see also information bottleneck theory.

Underfitting vs. Overfitting

- Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set (low-capacity models).
- Overfitting occurs when the gap between the training error and test error is too high (high-capacity models).



Overfitting

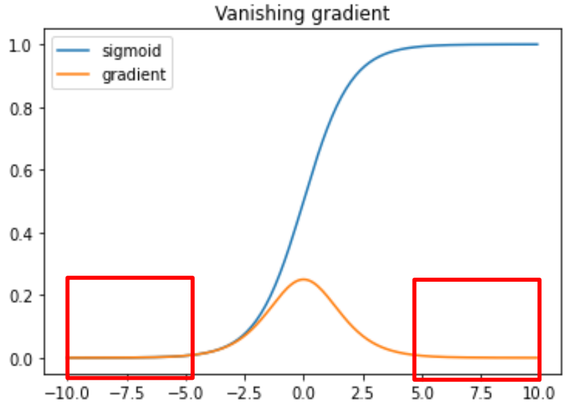


- The neural network is forced to overfit by considering only 10 training points
- The predicted function passes through the 10 training points, making training loss $\sim 10^{-7}$
- Model fails to learn the underlying function

Vanishing and Exploding Gradients

- Different layers may learn at hugely different rates: for most NN architectures the gradients become smaller and smaller in back propagation, hence leaving the weights of lower layers unaffected (vanishing gradients).

- **Examples : Vanishing and exploding gradient**



Exploding gradients: Multiply 100 Gaussian random matrices
(All linear layers)

```
A single matrix
 tensor([[ 0.7948, -0.5345,  2.2011, -1.2147, -0.8642],
         [-1.0132, -0.3668,  1.3064, -0.3739,  0.3137],
         [-0.7110,  0.0634, -3.0735, -0.8933, -0.2504],
         [ 0.2037, -0.6473,  1.2173,  0.6089, -1.1243],
         [ 0.7627,  0.8086, -0.9196, -1.1723,  0.4238]])
After multiplying 100 matrices
 tensor([[-5.9421e+28,  2.5019e+28, -4.7395e+27,  9.2365e+28, -9.0353e+28],
         [-4.1826e+28,  1.7611e+28, -3.3361e+27,  6.5015e+28, -6.3598e+28],
         [ 8.4344e+28, -3.5513e+28,  6.7274e+27, -1.3111e+29,  1.2825e+29],
         [-2.4234e+28,  1.0204e+28, -1.9328e+27,  3.7670e+28, -3.6850e+28],
         [ 2.1614e+28, -9.1011e+27,  1.7237e+27, -3.3597e+28,  3.2865e+28]])
```

- This was the main obstacle in training DNNs until the early 2000s.

- 2010: Breakthrough paper of Xavier Glorot & Youshua Bengio "*Understanding the difficulty of training Deep neural nerworks*", Proc 13th Int. Conf. on AI and Statistics pp. 249-256.

- The main reasons were the then popular sigmoid activation function and the normal distribution of initialized weights $\mathcal{N}(0,1)$. The variance of each layer from input to output increases monotonically and then the activation function saturates at 0 and 1 in the deep layers. Note that the mean of this activation function is 0.5.

Xavier and He - Weight Initializations

- Variance of output of each layer = Variance of inputs to that layer.
- Gradients should have equal variance before and after flowing through a layer in the reverse direction (fan-in/fan-out) – this led to *Xavier (or Glorot) initialization.*

$$w \sim \mathcal{N}\left(0, \sqrt{\frac{1}{\text{fan}_{\text{avg}}}}\right)$$

$$\text{fan}_{\text{avg}} = 0.5(\text{fan}_{\text{in}} + \text{fan}_{\text{out}})$$

- He Normal
- *He initialization* is similar with ReLU $\mathcal{N}(0, \sqrt{\frac{2}{\text{fan}_{\text{in}}}})$

Glorot Initialization
`torch.nn.init.xavier_normal_(w)`

Example:
```
w = torch.empty(5, 5)
nn.init.xavier_normal_(w)
```

He Initialization
`torch.nn.init.kaiming_normal_(w)`

Example:
```
w = torch.empty(5, 5)
torch.nn.init. kaiming_normal_(w)
```

Glorot Initialization
`tf.initializers.GlorotNormal()`

Example:
```
init = tf.initializers.GlorotNormal()
w = initializer(shape=(4, 4))
```

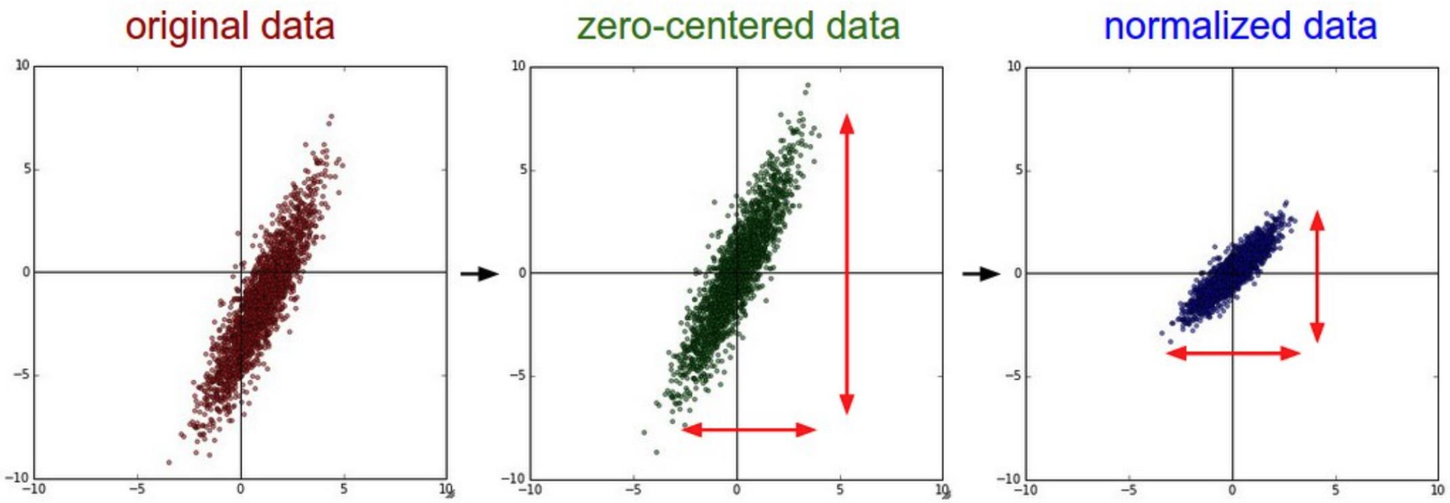
He Initialization
`tf.initializers.HeNormal()`

Example:
```
init = tf.initializers.HeNormal()
w = initializer(shape=(4, 4))
```

"Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", 2015, IEEE Conference on Computer Vision, pp. 1026-1034

Data Normalization

Ref: https://zaffnet.github.io/batch-normalization



Common data preprocessing pipeline. **Left**: Original toy, 2-dimensional input data. **Middle**: The data is zero-centered by subtracting the mean in each dimension. The data cloud is now centered around the origin. **Right**: Each dimension is additionally scaled by its standard deviation. The red lines indicate the extent of the data - they are of unequal length in the middle, but of equal length on the right.

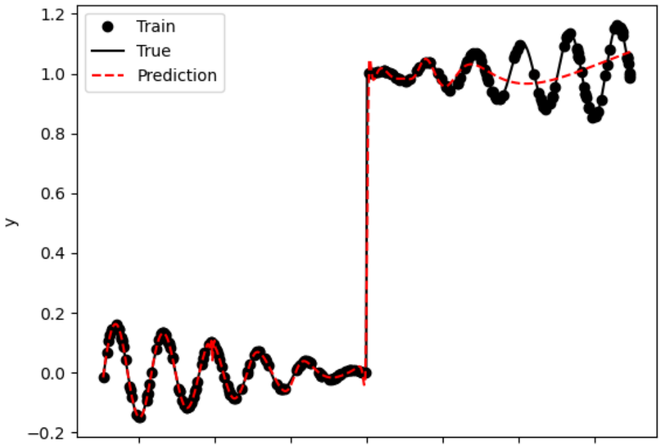
In 1998, Yan LeCun in his paper, Effiecient BackProp, highlighted the importance of normalizing the inputs. Preprocessing of the inputs using normalization is a standard machine learning procedure and is known to help in faster convergence. Normalization is done to achieve the following objectives:

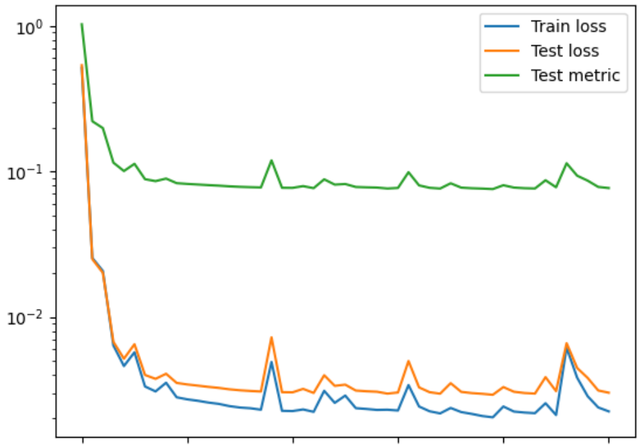
- ❑ The average of each input variable (or feature) over the training set is close to zero (Mean subtraction).
- ❑ Covariances of the features are same (Scaling).
- ❑ Decorrelate the features (Whitening – not required for CNNs).

Data Normalization - Example

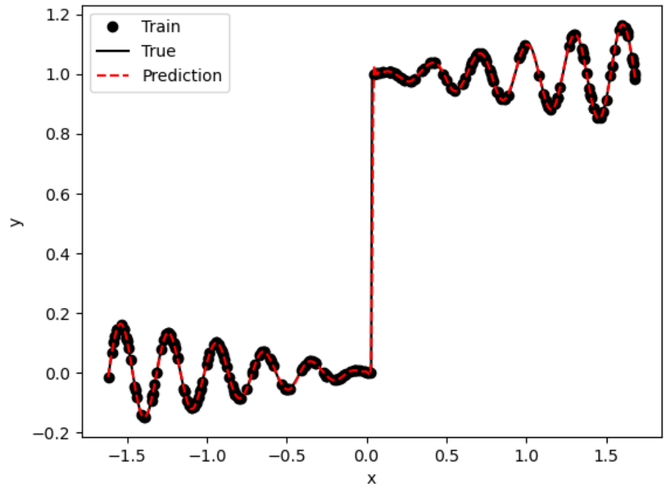
Without data-normalization

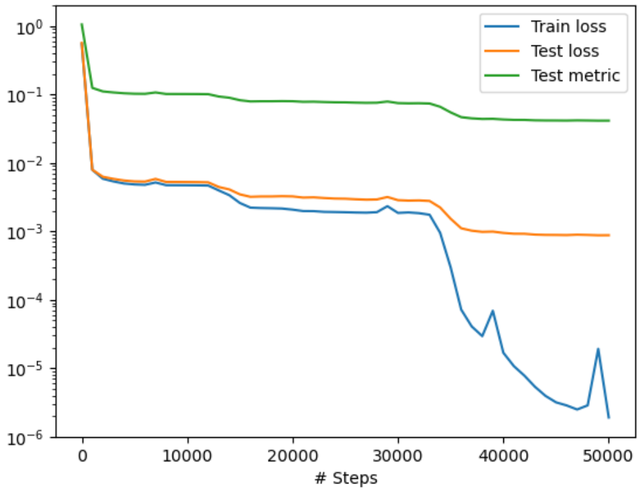
$$y = \begin{cases} \frac{1}{10} x sin(20x) & \text{if } x < 0 \\ 0.5 & \text{if } x = 0.5 \\ 1 + \frac{1}{10} x sin(20x) & \text{if } x > 0 \end{cases}$$



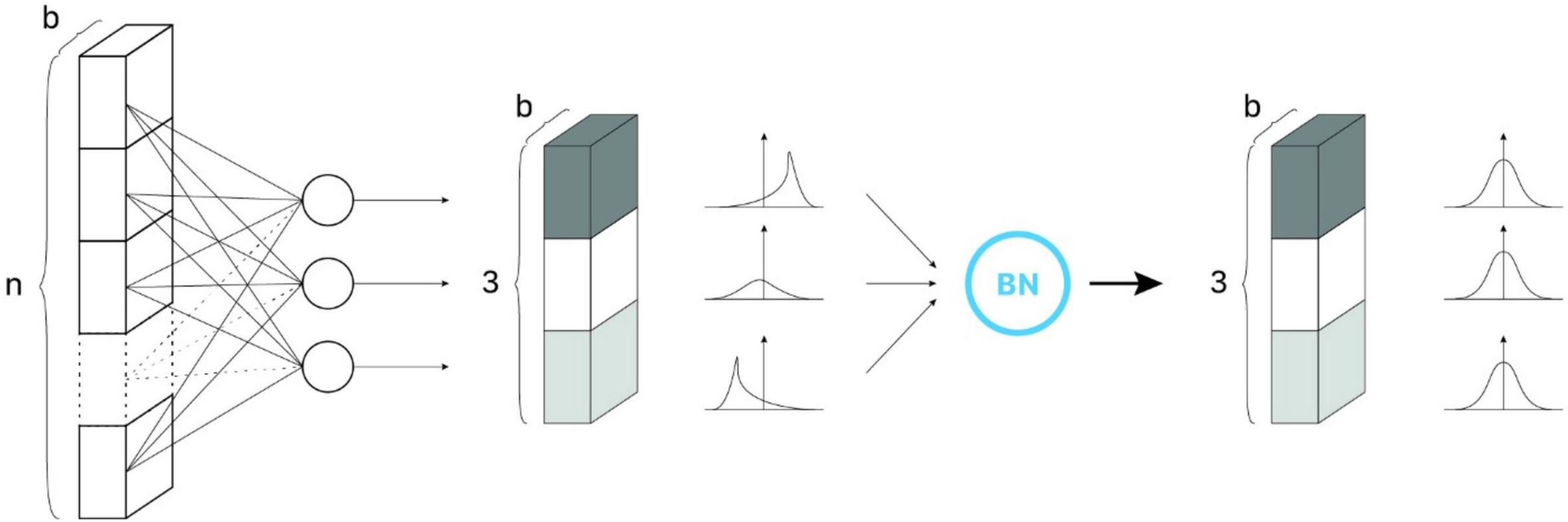


With data-normalization





Batch Normalization -



Batch Normalization - Algorithm

Algorithm:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;

Parameters to be learned: γ, β

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

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

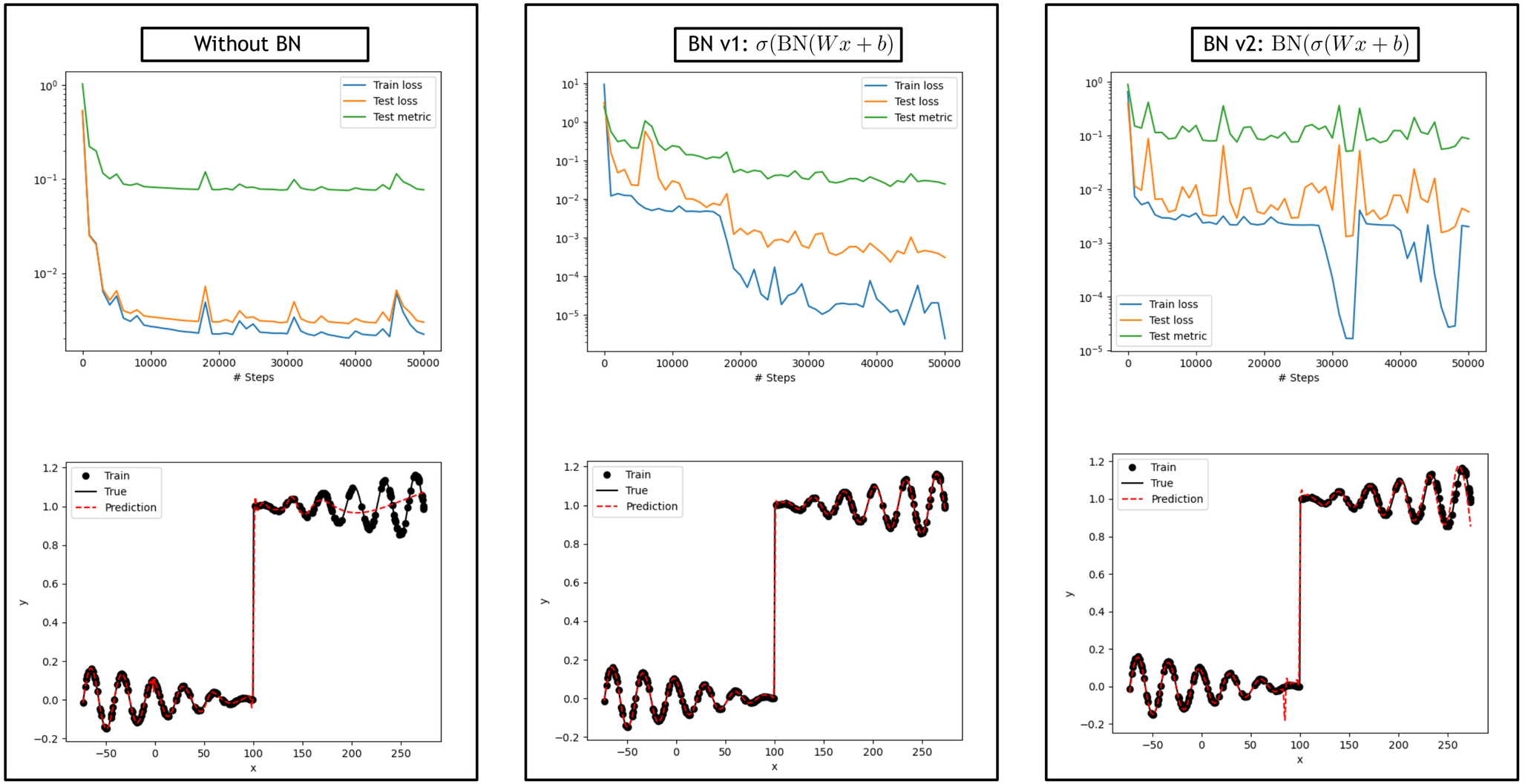
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

- Helps in faster convergence.
- Improves gradient flow through the network (and hence mitigates the *vanishing gradient* problem).
- Allows higher learning rate and reduces high dependence on initialization.
- Acts as a form of regularization and reduces the need for Dropout.
- The learned affine transformation $y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$ helps in preserving the identity mapping $\gamma(i) = \sqrt{(\sigma[y(i)])}$ and $\beta(i) = E[y(i)]$, if the network finds this optimal.
- The Batch Normalization transformation is differentiable and hence can be added comfortably in a computational graph. Nowadays we can get the backward pass for free using automatic differentiation (AD).

Batch Normalization - Example



Batch Normalization pays off! (BNv1 provides the most effective treatment)

What Optimizer to Use?

- Géron A. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media; 2019 Sep 5.

1. Momentum optimizer

- To simulate some sort of friction mechanism and prevent the momentum from growing too large.
- The algorithm introduces a new hyperparameter β, called the *momentum*, and $\beta \in [0, 1]$ with:

$$\beta = 0 \text{ High friction}$$
$$\beta = 1 \text{ No friction.}$$

- A typical momentum value is 0.9.
- Momentum optimization suppresses the oscillations of GD and helps to escape from plateaus much faster than GD.
- In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot.
- It can also help roll past local optima.

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

```
# Import Optimizers bundle
opt_class=torch.optim
# Momentum optimizer
mnt= opt_class.SGD(model.parameters(),lr=0.01,momentum=0.9)
```

```
# Import Optimizers bundle
opt_class=tf.keras.optimizers
# Momentum optimizer
opt=opt_class.SGD(lr=0.01,momentum=0.9)
```

What Optimizer to Use?

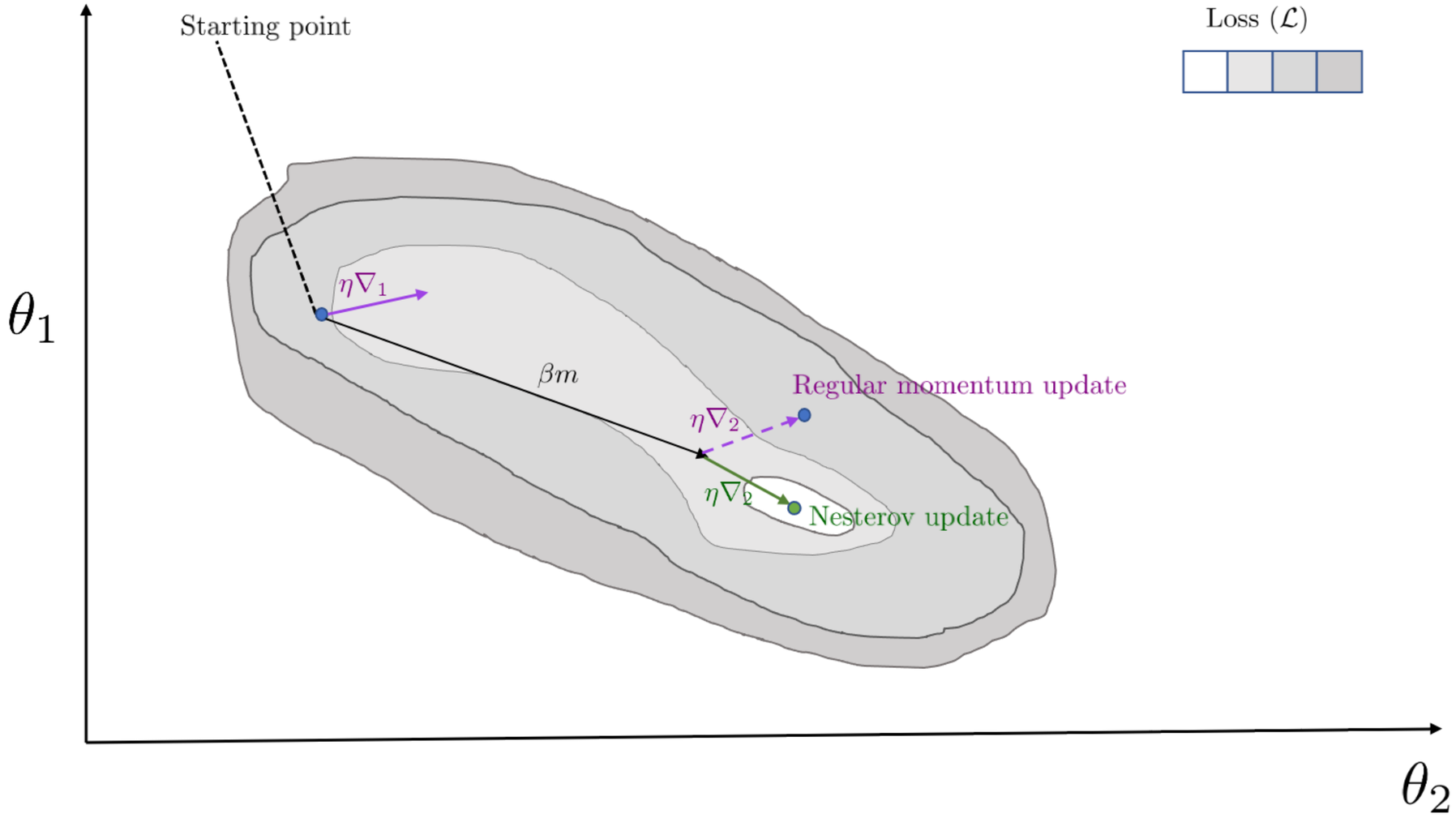
2. Nesterov momentum optimizer or Nesterov Accelerated Gradient (NAG) :

- A variant of momentum optimizer.
- NAG measures the gradient of the loss function not at the local position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, at $\boldsymbol{\theta} + \beta \boldsymbol{m}$.
- NAG is generally faster than regular momentum optimization.

1. $\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta \boldsymbol{m})$
2. $\boldsymbol{\theta} \leftarrow \theta + \boldsymbol{m}$
	`# Import Optimizers bundle`
	`opts=torch.optim`
	`# Momentum optimizer`
	`opt= opts.SGD(model.parameters(),lr=0.01,momentum=0.9,nesterov=True)`
		`# Import Optimizers bundle`
		`opt_class=tf.keras.optimizers`
		`# Momentum optimizer`
		`opt=opt_class.SGD(lr=0.01,momentum=0.9,nesterov=True)`

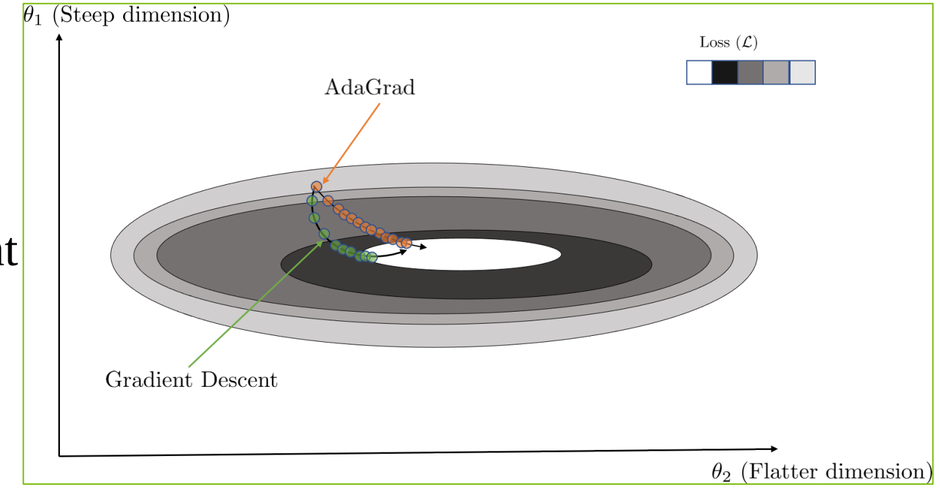
What Optimizer to Use?

2. Momentum vs Nesterov



What Optimizer to Use?

3. AdaGrad Optmizer



- AdaGrad achieves a correction due to anisotropy by scaling down the gradient vector along the steepest dimensions.

- The first step accumulates the square of the gradients into the vector $\boldsymbol{s}$.

- The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{(\boldsymbol{s}+\epsilon)}$. ($\boldsymbol{\epsilon}$ is a smoothing term to avoid division by zero, and typically $\boldsymbol{\epsilon = 10^{-10}}$).

- AdaGrad decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum.

- Moreover, it requires much less tuning of the learning rate hyperparameter η. But it may never converge to global minimum (not used often in practice).

$\{\otimes : \text{element wise product}, \oslash : \text{element wise division}\}$

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

```
# Import Optimizers bundle
opt_class=torch.optim
# Momentum optimizer
opt= opt_class.AdaGrad(model.parameters(),lr=0.01, eps=1e-10)
```

```
# Import Optimizers bundle
opt_class=tf.keras.optimizers
# Momentum optimizer
opt=opt_class.Adagrad(lr=0.01, epsilon=1e-10)
```

What Optimizer to Use?

4. RMSProp

- AdaGrad has risk of slowing down a bit too fast and never converging to global optimum and RMSProp comes to rescue.
- Instead of accumulating gradients from the beginning, RMSProp algorithm accumulates gradients from the most recent iterations.
- It does so by using exponential decay in the first step. It introduces a new hyperparameter *β, which* is typically set to 0.9.
- Except on very simple problems, this optimizer almost always performs much better than AdaGrad.
- It was the preferred optimization algorithm of many researchers until Adam optimization came around.

f_x		

1. $\boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1-\beta)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \bigotimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \theta - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\boldsymbol{s} + \epsilon}$
	`# Import Optimizers bundle`
	`opt_class=torch.optim`
	`# Momentum optimizer`
	`opt= opt_class.RMSprop(model.parameters(),lr=0.01, alpha=0.9)`
		`# Import Optimizers bundle`
		`opt_class=tf.keras.optimizers`
		`# Momentum optimizer`
		`opt=opt_class.RMSprop(lr=0.001, rho=0.9)`

*β in algorithm is set as setting alpha in PyTorch and *rho* in TensorFlow.

What Optimizer to Use?

5. Adam Optimizer: Adaptive moment based optimizer

- Adam = *adaptive moment estimation* is a hybrid method and combines the ideas of momentum optimization and RMSProp.
- Similar to momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients.
- Steps 1, 2, and 5 in algorithm (below) reveal Adam's close similarity to both momentum optimization and RMSProp.

1. $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^{\top}}$
4. $\widehat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2^{\top}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\boldsymbol{m}} \oslash \sqrt{\widehat{\boldsymbol{s}} + \epsilon}$

```
# Import Optimizers bundle
opt_class=torch.optim
# Momentum optimizer
opt= opt_class.Adam(model.parameters(),lr=0.01,betas=(0.9,0.999))
```

```
# Import Optimizers bundle
opt_class=tf.keras.optimizers
# Momentum optimizer
opt=opt_class.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

$\top$ denotes iteration number starting at 1.

What Optimizer to Use?

5. Adam Optimizer: Adaptive moment based optimizer

- Steps 3 and 4 can be explained as follows: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

- The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. The smoothing term ε is usually initialized to a small number such as 10^{-7}.

- Since Adam is an adaptive learning rate algorithm, it requires less tuning of the learning rate hyperparameter η. We can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

1. $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^T}$
4. $\widehat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2^T}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\widehat{\boldsymbol{m}} \oslash \sqrt{\widehat{\boldsymbol{s}} + \epsilon}$

```
# Import Optimizers bundle
opt_class=torch.optim
# Momentum optimizer
opt= opt_class.Adam(model.parameters(),lr=0.01,betas=(0.9,0.999))
```

```
# Import Optimizers bundle
opt_class=tf.keras.optimizers
# Momentum optimizer
opt=opt_class.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

What Optimizer to Use?

6. Nadam Optimizer: Nesterov Adaptive moment based optimizer

- Nadam = *Adam + Nesterov trick.*
- Nadam will often converge slightly faster than Adam.
- Nadam generally outperforms Adam but is sometimes outperformed by RMSProp.

Algorithm 1: NADAM

Input : initial $\alpha_0, \ldots, \alpha_T$; $\mu_0, \ldots, \mu_T$; ν;
Input : initial ϵ: Hyperparametrs
$\boldsymbol{m}_0; \boldsymbol{n}_0 \leftarrow$ (first/second moment vectors)
do
  $\boldsymbol{g}_t \leftarrow \nabla_{\theta_{t-1}} f_t(\theta_{t-1})$
  $\boldsymbol{m}_t \leftarrow \mu_t \boldsymbol{m}_{t-1} + (1-\mu_t)\boldsymbol{g}_t$ Nesterov trick
  $\boldsymbol{n}_t \leftarrow \nu \boldsymbol{n}_{t-1} + (1-\nu)\boldsymbol{g}_t^2$
  $a \leftarrow (\mu_{t+1}\boldsymbol{m}_t / (1 - \prod_{i=1}^{t+1} \mu_i))$
  $b \leftarrow ((1-\mu_t)\boldsymbol{g}_t / (1 - \prod_{i=1}^{t} \mu_i))$
  $\widehat{\boldsymbol{m}} \leftarrow a + b$
  $\widehat{\boldsymbol{n}} \leftarrow \nu \boldsymbol{n}_t / (1-\nu^t)$
  $\theta_t \leftarrow \theta_{t-1} - \frac{\alpha_t}{\sqrt{\widehat{\boldsymbol{n}}_t + \epsilon}} \widehat{\boldsymbol{m}}_t$
while *until* θ_t *is not converged;*

```
# Import Optimizers bundle
opt_class=torch.optim
# Momentum optimizer
opt= opt_class.Nadam(model.parameters(),lr=0.01,betas=(0.9,0.999))
```

```
# Import Optimizers bundle
opt_class=tf.keras.optimizers
# Momentum optimizer
opt=opt_class.NAdam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

What Optimizer to Use?

Adam vs Nadam

$$y = 5 + \sum_{k=1}^{4} \sin(kx), \ x < 0$$

$$y = \cos(10x), \ x \geq 0$$

Error for Adam : 0.281

Error for Nadam : 0.394

Time for Adam : 4.307 sec

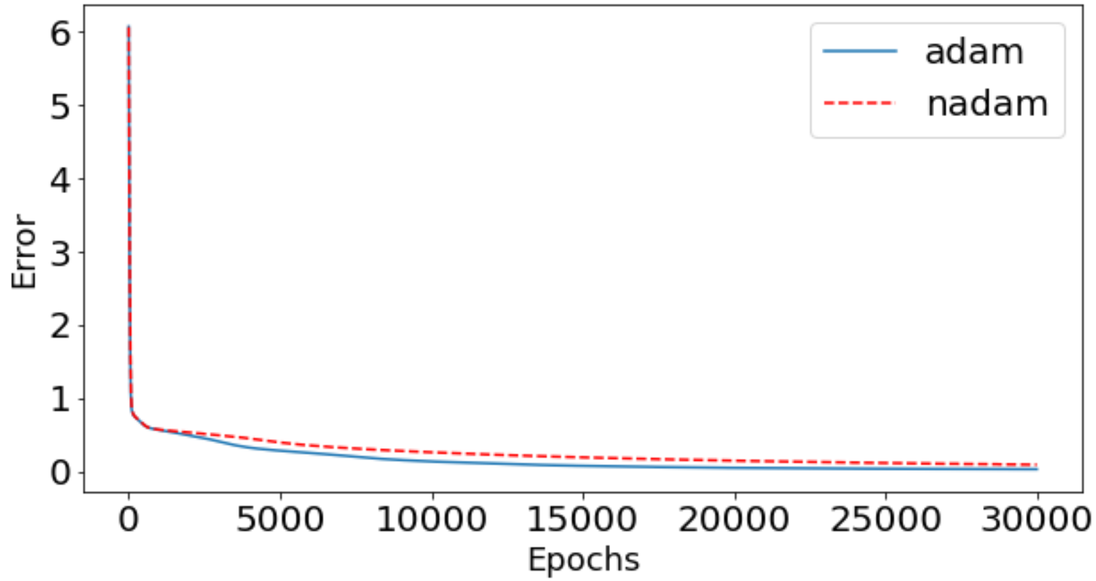
Time for Nadam : 4.224 sec

Number of Hidden layers: 2
Number of Hidden Units: 100
Activation function: tanh
Learning rate: 0.001

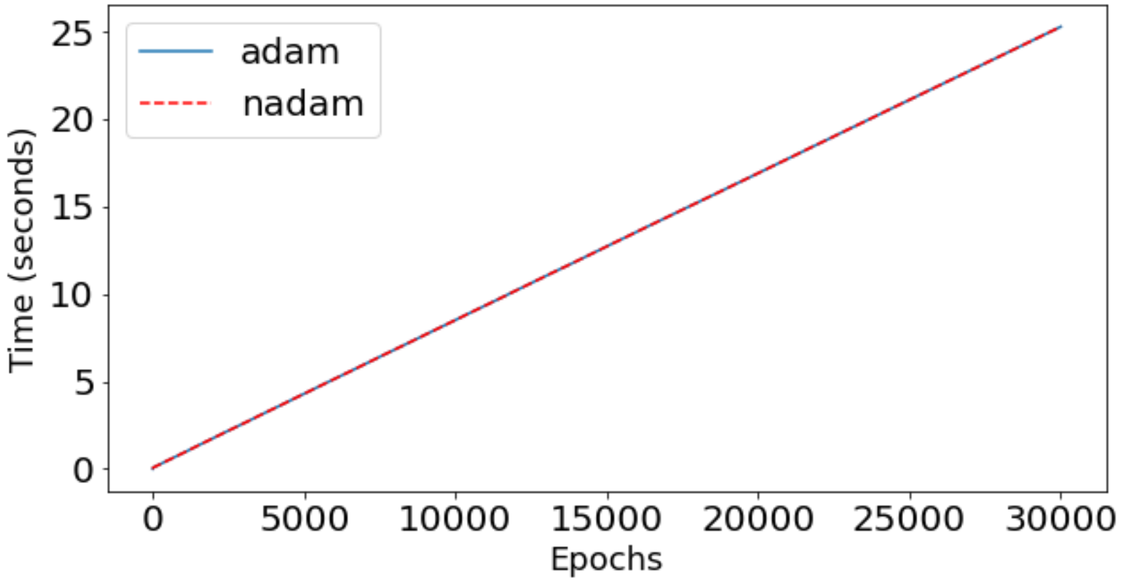
I. Error for Adam always better

II. Time is almost the same

MSE Error



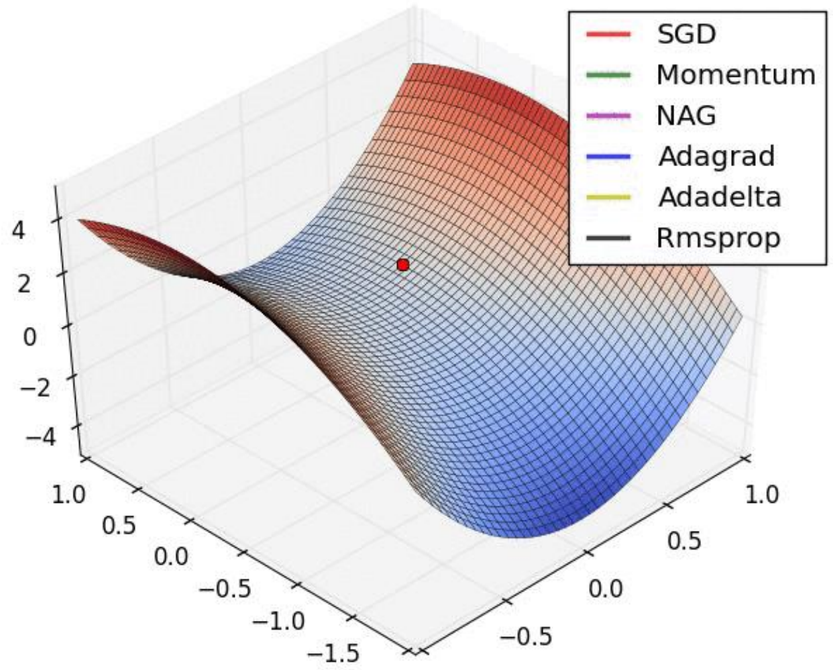
Time



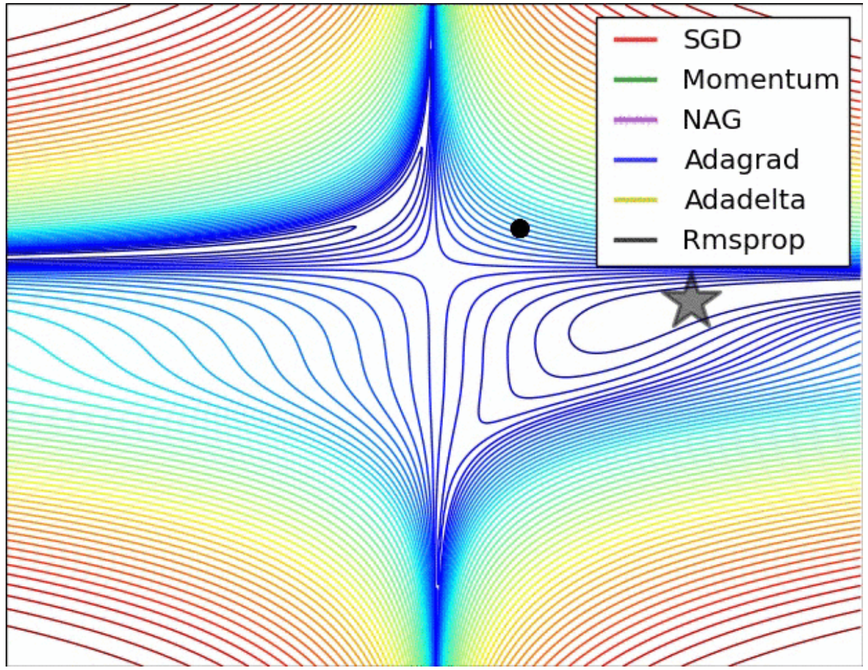
An Overview of Gradient Descent Optimization Algorithms

https://arxiv.org/abs/1609.04747

- This post explores how many of the most popular gradient-based optimization algorithms actually work.



A. This movie shows the **behaviour of the algorithms at a saddle point** Notice that SGD, Momentum, and NAG find it difficulty to break symmetry, although the two latter eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope.



B. In this movie, we see their **behavior on the contours of a loss surface (the Beale function) over time**. Note that Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge similarly fast, while Momentum and NAG are led off-track, evoking the image of a ball rolling down the hill. NAG, however, is quickly able to correct its course due to its increased responsiveness by looking ahead and heads to the minimum.

- These two animations (Images credit: Alec Radford) provide some intuitions towards the optimization behavior of most of the presented optimization methods.

What Optimizer to Use?

➢ Even faster optimizers : **second-order**

$$\nabla^2_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1^2} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2^2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d^2} \end{bmatrix}$$

Hessian based optimizers

❑ Optimization technique discussed so far is based on first-order partial derivatives (Jacobians).

❑ There are optimizers which are based on the second-order partial derivatives (the Hessian).

❑ These algorithms are very expensive as there are n^2 Hessians to be computed for each output (n is number of parameters) in comparison to n Jacobians per output.

❑ Next, we will discuss a couple of Hessian-based optimizer and their implementation in TensorFlow and PyTorch.

What Optimizer to Use?

Iterative algorithm

❑ Optimization uses iterative algorithms, which consider a sequence of $\boldsymbol{x}_n$ converging to $\boldsymbol{x}^*$. Therefore, $\forall\ f''(\boldsymbol{x}) \neq 0$, and current estimate being $\boldsymbol{x}_n$, the next estimate $\boldsymbol{x}_{n+1}$ should satisfy

$$f(\boldsymbol{x}_{n+1}) < f(\boldsymbol{x}_n)$$

❑ Using Taylor Approximation

$$f(\boldsymbol{x} + \Delta x) \approx f(\boldsymbol{x}) + \Delta x^\top \nabla f(x) + \frac{1}{2}\Delta x^T(\nabla^2 f(x))\Delta x$$

❑ Iterative algorithm produces a sequence of such quadratic approximations $\boldsymbol{h}_n$ and considers $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta \boldsymbol{x}$;

$$h_n(\Delta \boldsymbol{x}) = f(\boldsymbol{x}_n) + \Delta x^\top \boldsymbol{g}_n + \tfrac{1}{2}\Delta x^\top \boldsymbol{H}_n \Delta x$$

Where $\boldsymbol{g}_n$ and $\boldsymbol{H}_n$ are gradient and Hessian of f at x_n

❑ Minimality condition

$$\frac{\partial h_n(\Delta \boldsymbol{x})}{\partial \Delta x} = \boldsymbol{g}_n + \boldsymbol{H}_n \Delta x = 0$$

$$\Delta x = -\boldsymbol{H}_n^{-1}\boldsymbol{g}_n$$

$$\boxed{\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha(\boldsymbol{H}_n^{-1}\boldsymbol{g}_n)(\boldsymbol{x}_n)}$$

What Optimizer to Use?

Caveats: Iterative algorithm

- The main issue with iterative algorithm is that it needs to compute the inverse Hessian matrix.
- In DL applications, the dimensionality of the input corresponds to model parameters. It is quite common to have hundreds of millions of parameters.
- For these reasons, computing the Hessian or its inverse is often impractical.
- Therefore, Iterative algorithm in this form is rarely used in practice to optimize functions corresponding to large problems.
- Luckily, the above algorithm can still work even if $\boldsymbol{H}_n^{-1}$ does not correspond to the exact inverse hessian at $\boldsymbol{x}_n$, but is instead a good approximation. These are handled by **quasi-Newton methods and are implemented through BFGS and L-BFGS methods.**

What Optimizer to Use?

Quasi-Newton Methods

- General form of quasi-Newton methods is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \boldsymbol{H}_n^{-1} \nabla f(\boldsymbol{x}_n)$$

where α is line search parameter and is $\boldsymbol{H}_n^{-1}$ Hessian or some approximation to the Hessian.

- Quasi-Newton methods lose their quadratic convergence of Newton's method, but super linear convergence is often achieved.

- Here we will discus two forms of quasi-Newton methods:
  1) Broyden-Fletcher-Goldfarb-Shannon (BFGS) method

  2) Limited-memory Broyden-Fletcher-Goldfarb-Shannon (L-BFGS) method

What Optimizer to Use?

1. BFGS Method

❑ General form of quasi-Newton methods is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \boldsymbol{H}_n^{-1} \nabla f(\boldsymbol{x}_n)$$

where α is line search parameter and is $\boldsymbol{H}_n^{-1}$ Hessian or some approximation to the Hessian.

Assume

$$\boldsymbol{H}_n \boldsymbol{s}_n = -\nabla f(\boldsymbol{x}_n), \qquad \boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \alpha \boldsymbol{s}_n$$
$$\boldsymbol{y}_n = \nabla f(\boldsymbol{x}_{n+1}) - \nabla f(\boldsymbol{x}_n)$$

❑ The updated approximated Hessian should satisfy the following under-determined system

$$\boldsymbol{H}_{n+1} \boldsymbol{s}_n = \boldsymbol{y}_n$$

Here the objective is to adjust $\boldsymbol{H}_n$ minimally to satisfy the above condition by ensuring $\boldsymbol{H}_n$ remains symmetric-positive-definite matrix.

What Optimizer to Use?

1. BFGS Method

❑ BFGS proposed

$$\boldsymbol{H}_{n+1} = \boldsymbol{H}_n + \beta \boldsymbol{u}\boldsymbol{u}^\top + \gamma \boldsymbol{v}\boldsymbol{v}^\top$$

❑ Substitute $\boldsymbol{u} = \boldsymbol{y}_n$ and $\boldsymbol{v} = \boldsymbol{H}_n \boldsymbol{s}_n$ and $\boldsymbol{H}_n^\top = \boldsymbol{H}_n$ to obtain

$$\boldsymbol{H}_{n+1} = \boldsymbol{H}_n + \beta \boldsymbol{y}_n \boldsymbol{y}_n^\top + \gamma \boldsymbol{H}_n \boldsymbol{s}_n \boldsymbol{s}_n^\top \boldsymbol{H}_n$$

❑ Since

$$\boldsymbol{y}_n = \boldsymbol{H}_{n+1}\boldsymbol{s}_n = \boldsymbol{H}_n \boldsymbol{s}_n + \beta \boldsymbol{y}_n \boldsymbol{y}_n^\top \boldsymbol{s}_n + \gamma \boldsymbol{H}_n \boldsymbol{s}_n \boldsymbol{s}_n^\top \boldsymbol{H}_n \boldsymbol{s}_n$$

$$\beta = \frac{1}{\boldsymbol{y}_n^\top \boldsymbol{s}_n} \qquad \gamma = -\frac{1}{\boldsymbol{s}_n^\top \boldsymbol{H}_n \boldsymbol{s}_n}$$

❑ Hence the update approximation to Hessian is

$$\boldsymbol{H}_{n+1} = \boldsymbol{H}_n + \frac{\boldsymbol{y}_n \boldsymbol{y}_n^\top}{\boldsymbol{y}_n^\top \boldsymbol{s}_n} - \frac{\boldsymbol{H}_n \boldsymbol{s}_n \boldsymbol{s}_n^\top \boldsymbol{H}_n}{\boldsymbol{s}_n^\top \boldsymbol{H}_n \boldsymbol{s}_n}$$

What Optimizer to Use?

- Liu DC, Nocedal J. On the limited memory BFGS method for large scale optimization. Mathematical programming. 1989 Aug;45(1):503-28.

2. L-BFGS optimizer

❑ BFGS is the most popular of all Quasi-Newton methods and have storage complexity.

❑L-BFGS (Limited memory BFGS), which does not require to explicitly store H^{-1} but instead stores the previous data $\{(x_i, \nabla f(x_i))\}_{i=1}^{k}$ and manages to compute $d = H^{-1}\nabla f(x)$ directly from this data. L-BFGS has storage complexity of $\mathcal{O}(n)$.

❑ L-BFGS implementation is not straightforward in PyTorch and TF2. A detailed implementation will be discussed in Lecture 4, but here we provide a simple API for both.

Algorithm 1: BFGS Algorithm

Input : initial $\boldsymbol{x}_0 \in \mathbb{R}^n$, functions $f(\boldsymbol{x})$, $\nabla f(\boldsymbol{x})$. tolerance θ
Output: x

initialize $\boldsymbol{H}^{-1} \leftarrow \mathbb{I}$
do
  compute $\boldsymbol{s} = \boldsymbol{H}^{-1}\nabla f(\boldsymbol{x})$
  perform a line search $\min_\alpha f(x + \alpha d)$
  $\boldsymbol{s} \leftarrow \alpha \boldsymbol{d}$
  $\boldsymbol{y} \leftarrow \nabla f(\boldsymbol{x} + \boldsymbol{s}) - \nabla f(\boldsymbol{x})$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{s}$
  update $\boldsymbol{H}^{-1} \leftarrow \left(\mathbb{I} - \frac{\boldsymbol{y}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}\right)\boldsymbol{H}^{-1}\left(\mathbb{I} - \frac{\boldsymbol{y}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}\right) + \frac{\boldsymbol{s}\boldsymbol{s}^\top}{\boldsymbol{s}^\top\boldsymbol{y}}$
while *until* $||s||_\infty < \theta$;

```
torch.optim.LBFGS(params, lr=1, max_iter=2
0, max_eval=None, tolerance_grad=1e-
07, tolerance_change=1e-
09, history_size=100, line_search_fn=None)
```

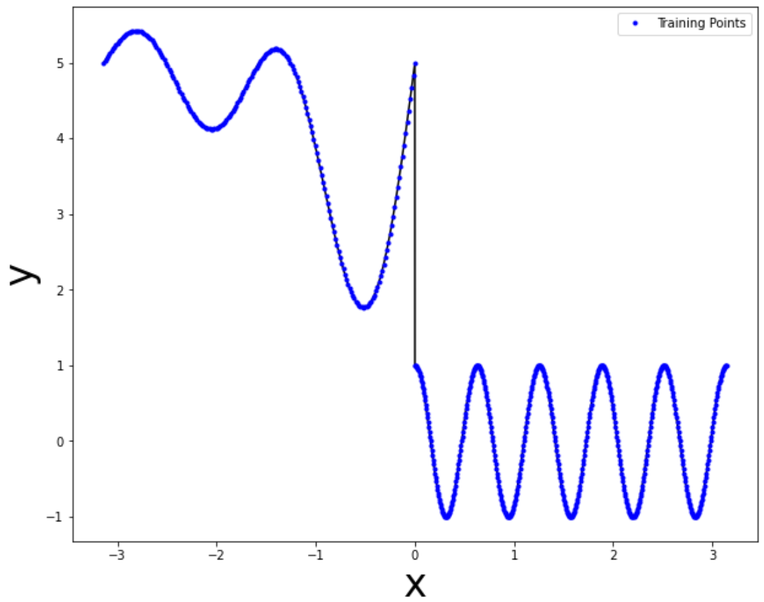
L-BFGS implementation in TensorFlow is provided through TF Probability package

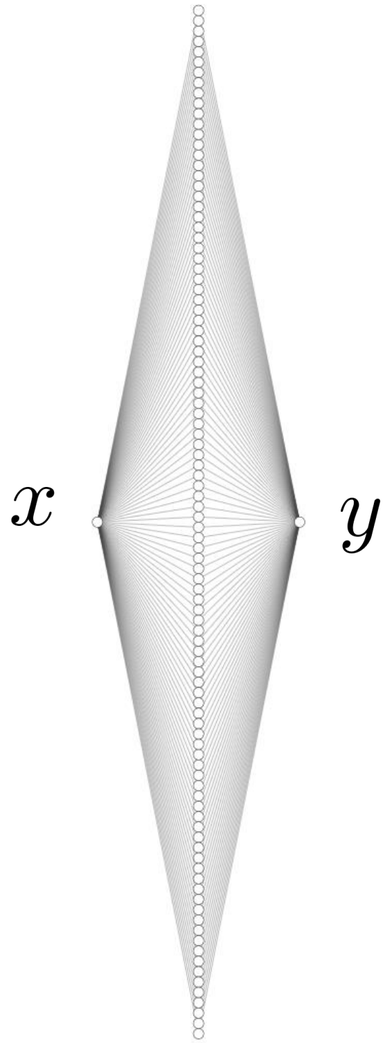
```
tfp.optimizer.lbfgs_minimize(f,
initial_position=self.get_weights(),
num_correction_pairs=50,
max_iterations=2000)
```

NN + Function Approximation + Adam + L-BFGS

$$y = 5 + \sum_{k=1}^{4} \sin(kx), \;\; x < 0$$

$$y = \cos(10x), \;\; x \geq 0$$





Function Approximations + Adam + LBFGS

1) Data Generation

```
import numpy as np
import imageio
import torch
import torch.nn.functional as F
import torch.utils.data as Data
from torch.autograd import Variable
import matplotlib.pyplot as plt
from torch.optim import SGD, LBFGS, Adam
from torch.utils.data import Dataset, DataLoader


%matplotlib inline
torch.manual_seed(1234)

### Input data
x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)
# torch.unsqueeze: Returns a new tensor
# with a dimension of size one inserted at the specified position.
y = torch.square(x)
# Add Random Noise
y = y + 0.1*torch.rand(y.size())

```

2) NN Model

```
Net = torch.nn.Sequential(
      torch.nn.Linear(1, 100),
      torch.nn.ReLU(),
      torch.nn.Linear(100, 1))
```

3) Optimizers Initialization

```
optimizer_adam = torch.optim.Adam(Net.parameters(), lr = 0.001)
optimizer_lbfgs = LBFGS(Net.parameters(), history_size=8, max_iter=500000)
```

4) Adam Optimization

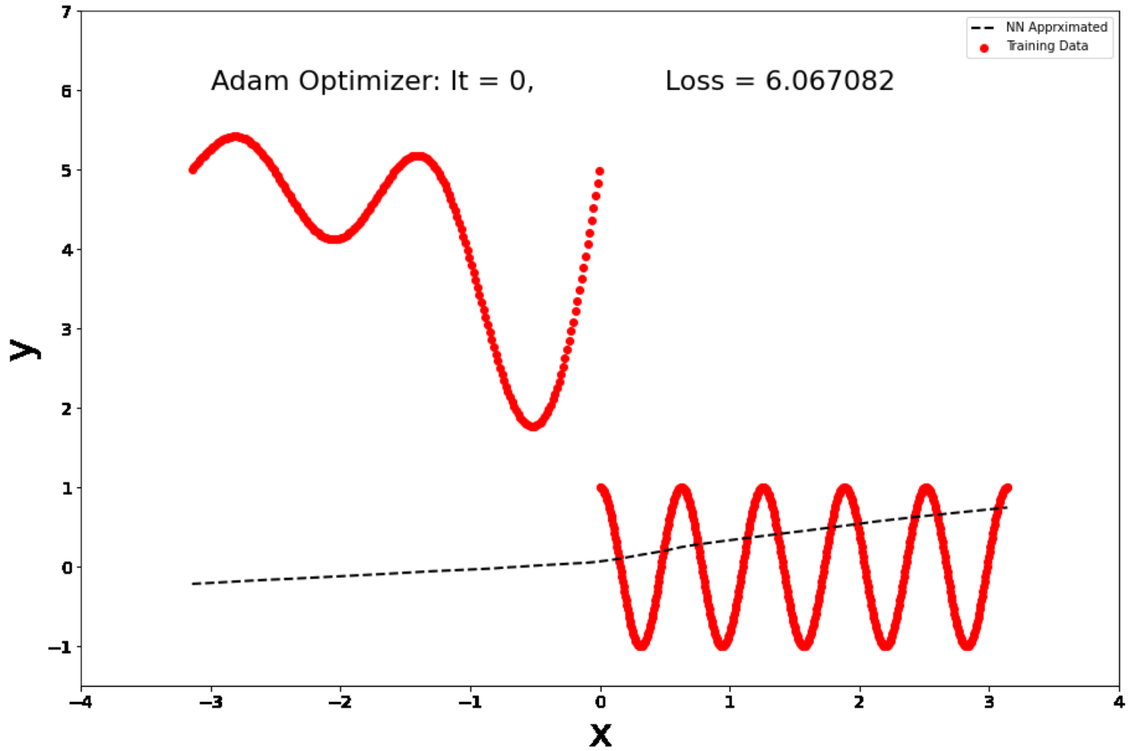
```
for it in range(Niter_Adam):
    y_pred = Net(x)
    loss = loss_function(y_pred, y)
    optimizer.zero_grad()    # Zero (
    loss.backward()
    optimizer_adam.step()
    optimizer.step(closure)
    loss = closure()
    adam_loss.append(loss)
```

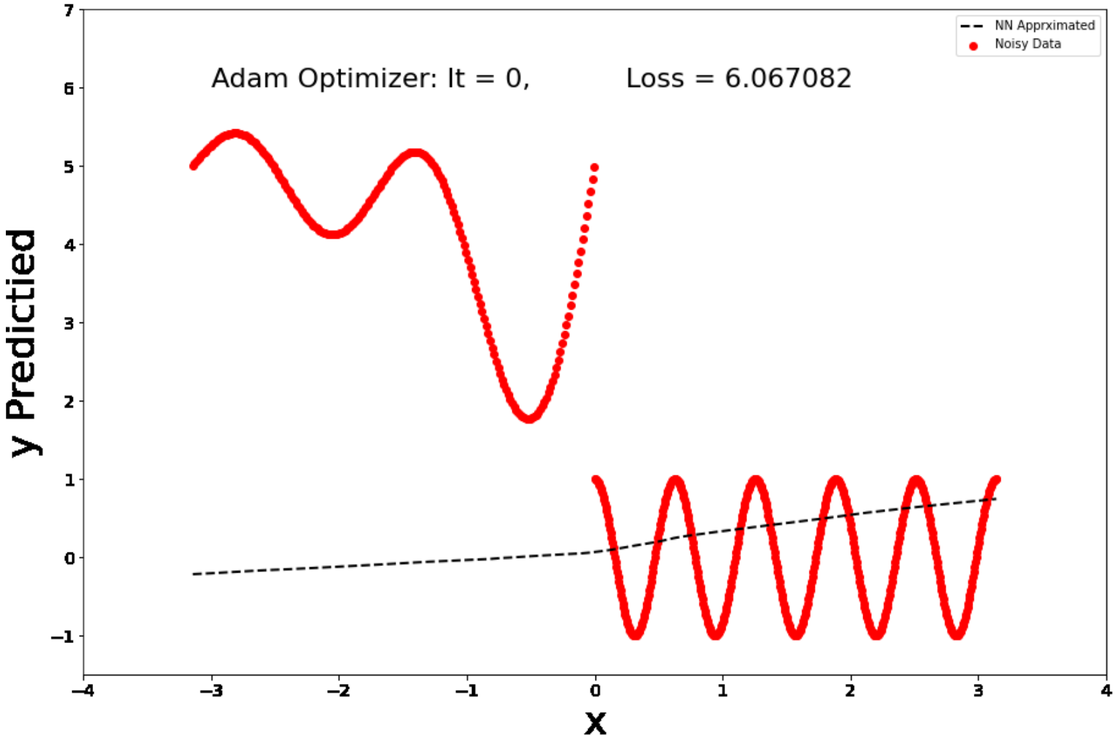
5) L-BFGS Optimization

```
for it in range(Niter_LBFGS):
    def closure():
        optimizer_lbfgs.zero_grad()
        y_pred = Net(x)
        loss = loss_function(y_pred, y)
        loss.backward()
        return loss

    optimizer_lbfgs.step(closure)
    loss = closure()
```

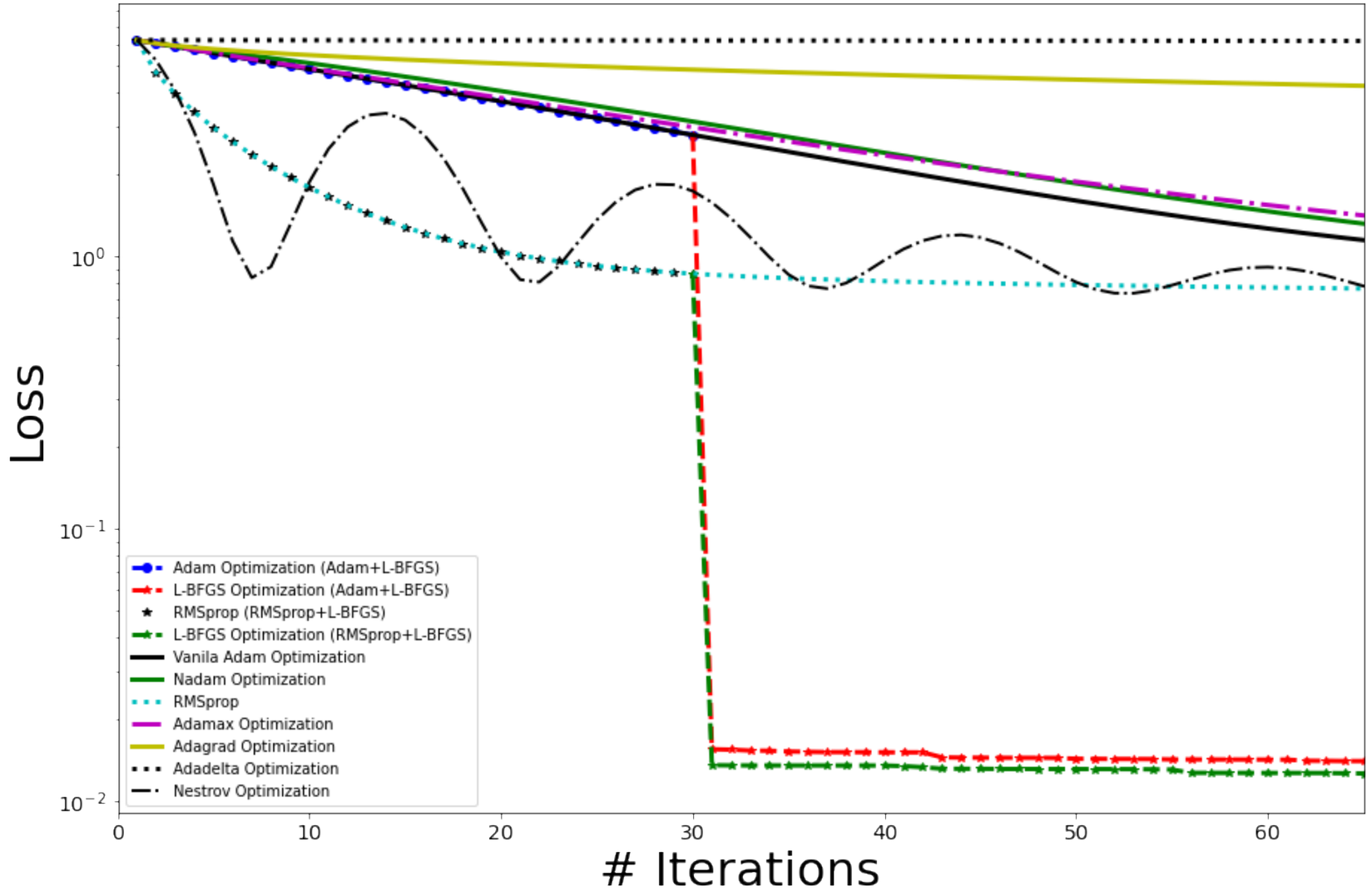
NN + Function Approximations + Adam + L-BFGS

Adam (30 Iterations) + 50 L-BFGS (50 Iterations)





Loss: X + L-BFGS


Loss
Iterations
10^0
10^{-1}
10^{-2}
0
10
20
30
40
50
60
Adam Optimization (Adam+L-BFGS)
L-BFGS Optimization (Adam+L-BFGS)
RMSprop (RMSprop+L-BFGS)
L-BFGS Optimization (RMSprop+L-BFGS)
Vanila Adam Optimization
Nadam Optimization
RMSprop
Adamax Optimization
Adagrad Optimization
Adadelta Optimization
Nestrov Optimization

Recap: Optimizers

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \nabla f(\boldsymbol{x}_n)$$

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \boldsymbol{H}_n^{-1} \nabla f(\boldsymbol{x}_n)$$

First Order	Second Order
Mometum	Newton Method
Nesterov	Quasi-Newton methods: BFGS
Adagrad	Quasi-Newton methods: L-BFGS x
RMSProp	
Adam x	
Nadam	

- Géron A. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media; 2019 Sep 5.

Learning Rate Scheduling

- We can find a good learning rate by training the model for a few hundred iterations, exponentially increasing learning rate from a very small value to a very large value.

- Then looking at the learning curve and picking a learning rate slightly lower than the one at which the learning curve starts shooting back up.
- We can then reinitialize the model and train it with that learning rate.

- We can do better than a constant learning rate: if we start with a large learning rate and then reduce it once training stops making fast progress, we can reach a good solution faster than with the optimal constant learning rate. There are few approaches to scheduling learning rate:

Dynamic scheduling:

$$\eta(t) = \eta_i \qquad \text{if} \quad t_i \leq t \leq t_{i+1} : \text{Piecewise decay}$$
$$\eta(t) = \eta_0 \exp(-\lambda t) \qquad : \text{Exponential decay}$$
$$\eta(t) = \eta_0 (\beta t + 1)^{-\alpha} : \text{Polynomial or Power decay}$$

Learning Rate Scheduling- Constant Learning

Consider the objective function

$$f(x) : x_1^2 + 2x_2^2$$

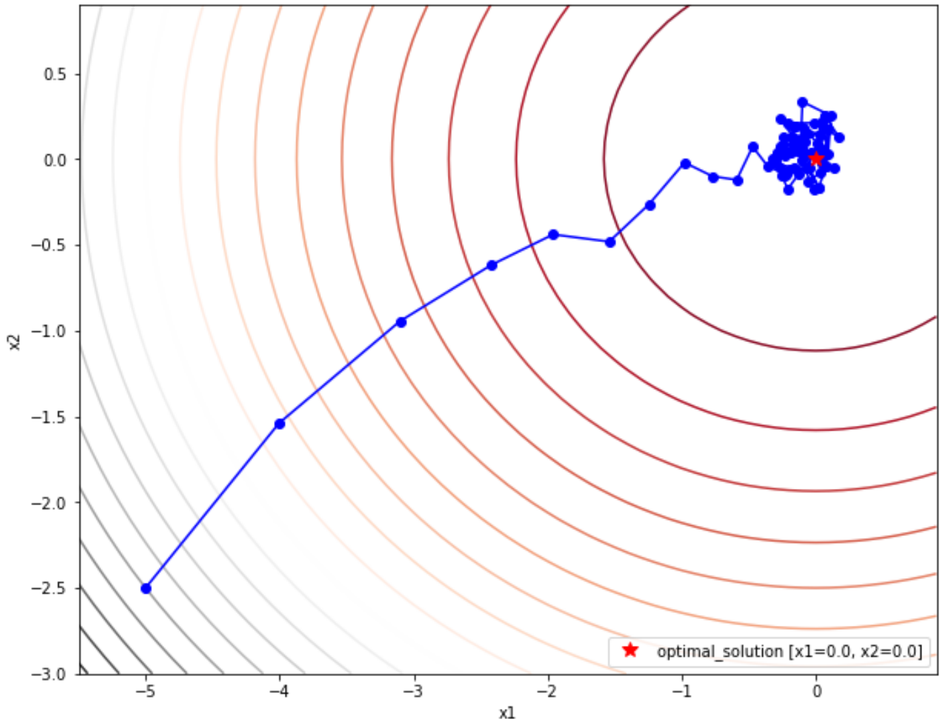
1. Descent trajectory is noisy.

2. Optimal solution has higher variance.

3. Solution will not improve even running it for longer epochs.

3. Small learning rate will lead to solution very slow.

Rescue: Dynamic Learning Rate




```
epoch 70, x1: 0.020621, x2: -0.171758
```

```python
def f(x1, x2):
    return x1**2 + 2*x2**2


def sgd(x1, x2, f_grad):
    g1, g2 = f_grad(x1, x2)
    g1 += torch.normal(0.0, 1, (1,)) # Gradient with noise
    g2 += torch.normal(0.0, 1, (1,)) #Gradient with noise
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2)

def f_grad(x1, x2):
    return 2 * x1, 4 * x2

def train(trainer,f_grad, steps=50):
    x1, x2 = -5, -2.5
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2 = trainer(x1, x2, f_grad)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

def constant_lr():
    return 1

eta = 0.1
lr = constant_lr
results = train(sgd, steps=70, f_grad=f_grad)
```

Learning Rate Scheduling- Exponential Learning

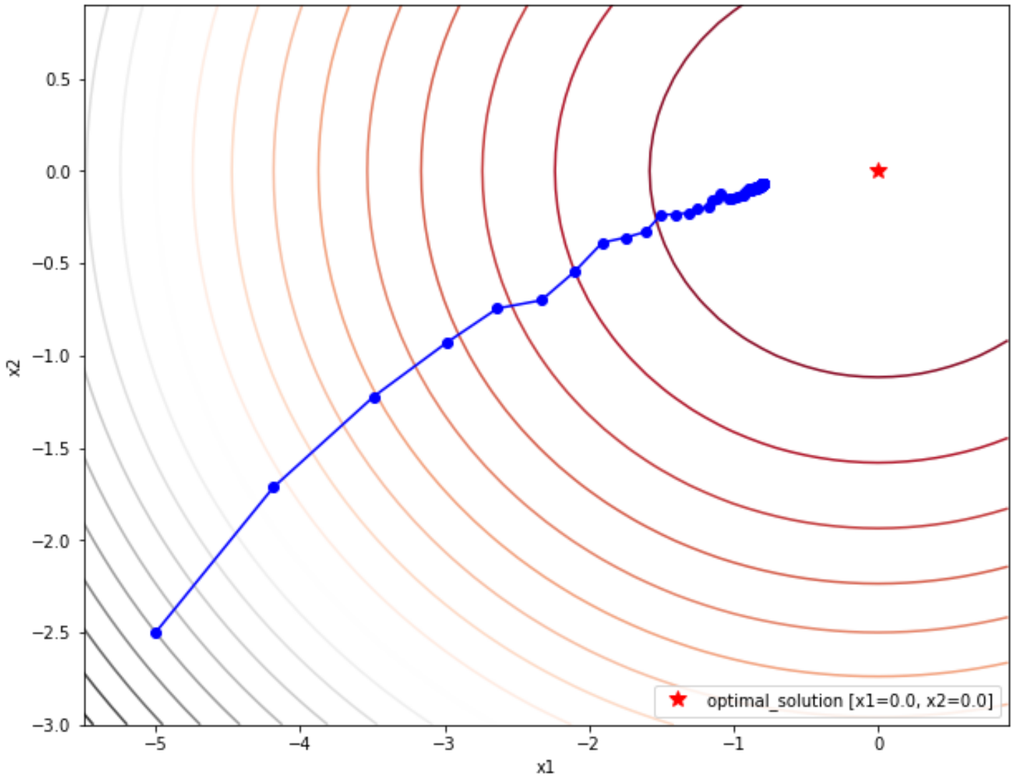
Consider the objective function

$$f(x) : x_1^2 + 2x_2^2$$

1. Descent trajectory is clean in comparison to constant learning rate.

2. Optimal solution has less variance.

3. Fail to converge to optimal solution.




```
epoch 2500, x1: -0.793612, x2: -0.070806
```

```python
def f(x1, x2):
    return x1**2 + 2*x2**2


def sgd(x1, x2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # Simulate noisy gradient
    g1 += torch.normal(0.0, 1, (1,))
    g2 += torch.normal(0.0, 1, (1,))
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2)

def f_grad(x1, x2):
    return 2 * x1, 4 * x2

def train_2d(trainer,f_grad, steps=50):
    x1, x2 = -5, -2.5
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2 = trainer(x1, x2, f_grad)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

def exponential_lr():
    global it
    it += 1
    return np.exp(-0.1 * it)
it =1
eta = 0.1
lr = exponential_lr
results = train_2d(sgd, steps=2500, f_grad=f_grad)
```

Learning Rate Scheduling- Polynomial Learning

Consider the objective function

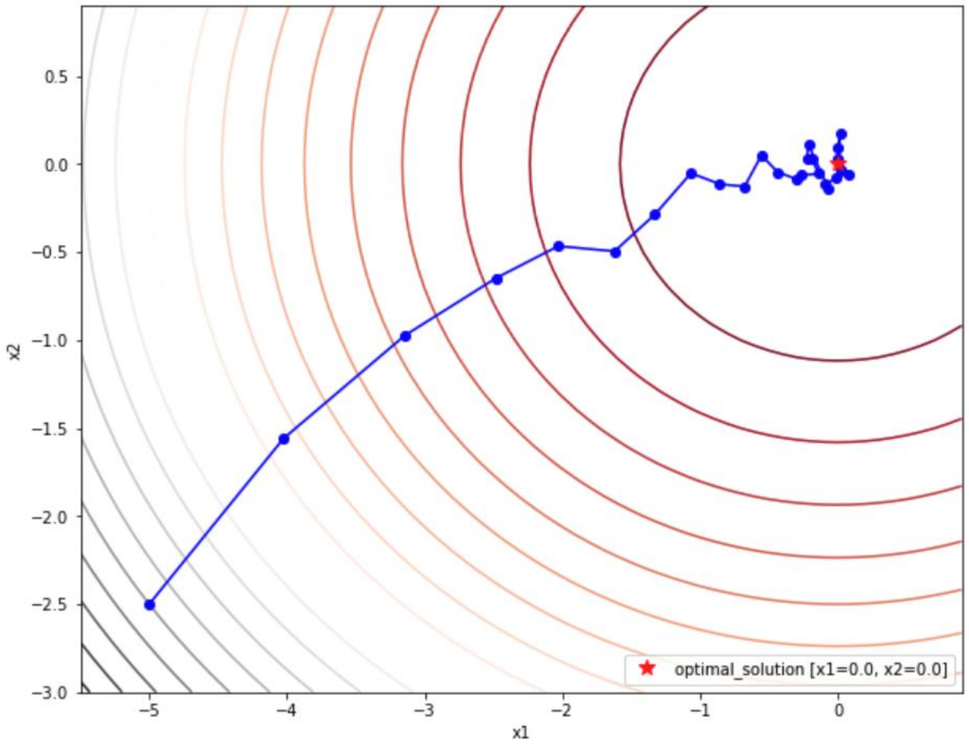
$$f(x) : x_1^2 + 2x_2^2$$

1. Descent trajectory is clean in comparison to constant learning rate.

2. Optimal solution has less variance.

3. Converges very fast with less variance in solution.

4. Therefore the adaptive learning rate helps in convergence and uncertainty quantification.



epoch 25, x1: 0.000743, x2: 0.088306

```python
def f(x1, x2):
    return x1**2 + 2*x2**2


def sgd(x1, x2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # Simulate noisy gradient
    g1 += torch.normal(0.0, 1, (1,))
    g2 += torch.normal(0.0, 1, (1,))
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2)

def f_grad(x1, x2):
    return 2 * x1, 4 * x2

def train_2d(trainer,f_grad, steps=50):
    x1, x2 = -5, -2.5
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2 = trainer(x1, x2, f_grad)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

def poly_lr():
    global it
    it += 1
    return (1 + 0.02 * it)**(-0.5)

it = 1
eta = 0.1
lr = poly_lr
results = train_2d(sgd, steps=25, f_grad=f_grad)
```

➢ Next we will discuss the Dynamic scheduling on Neural Network for function approximation using Power and 1 cycle scheduling.

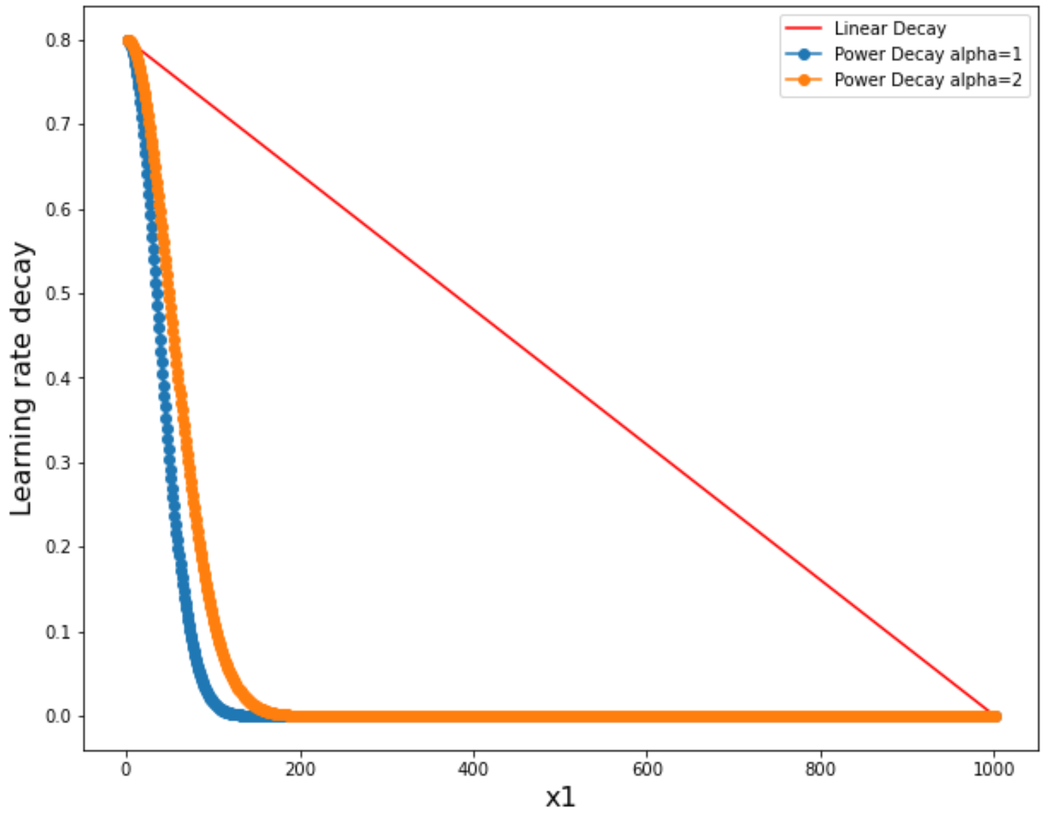
Learning Rate Scheduling in DNN

Power scheduling

- Set the learning rate η to a function of the iteration number t.

$$\eta(t) = \frac{\eta_0}{(1+\beta t)^{\alpha}}$$

- A popular choice for α is 0.5.




```python
class lr_decay():
    def __init__(self, epochs, initial_lr, power):
        self.epochs = epochs
        self.initial_lr = initial_lr
        self.power = power

    def linear_decay(self, epoch):
        decay = (1 - (epoch / float(self.epochs)))
        eta_updated = self.initial_lr * decay
        return float(eta_updated)

    def power_decay(self, epoch, lr, alpha):
        decay = self.initial_lr/float((self.epochs))
        return float(lr * 1 / (1 + decay * epoch)**alpha)

    def power_deacay_hist(self, alpha=1):
        lr = self.initial_lr
        power_decay_list = []
        for ep in range(0, self.epochs + 1):
            lr = self.power_decay(ep, lr, alpha)
            power_decay_list.append(lr)
        return power_decay_list



initial_lr = 0.8
epochs = 1000
lr_sch = lr_decay(epochs, initial_lr, power)
lr_linear = np.array([lr_sch.linear_decay(ep) for ep in range(0, epochs + 1)])
lr_power_alpha_1 = lr_sch.power_deacay_hist(alpha=1)
lr_power_alpha_2 = lr_sch.power_deacay_hist(alpha=0.5)

```

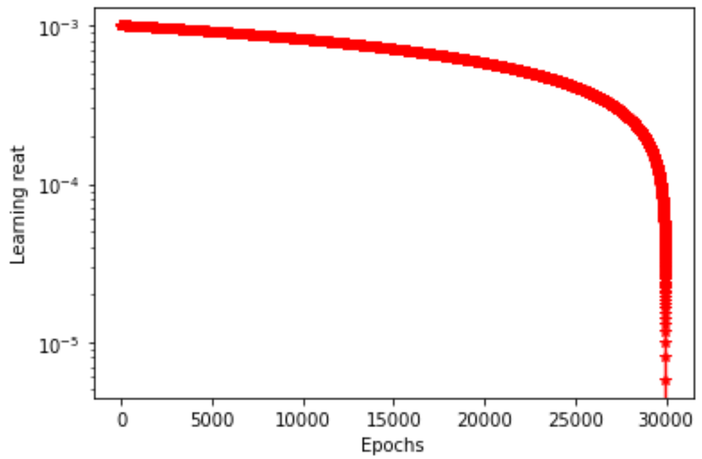
Learning Rate Scheduling in DNN

Power scheduling: user defined

1.
```
def lr_poly(base_lr, iter, max_iter, power):
    return base_lr*((1-float(iter)/max_iter)**(power))
```

2.
```
optimizer = SGD(Net.parameters(), lr = lr_intial)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, \
                                              lr_lambda=lambda it: lr_poly(lr_initial, it, Niter_Adam, 0.5))
```

3.
```
for it in range(Niter_Adam):
    y_pred = Net(x)
    loss = loss_function(y_pred, y)  # Notice the order: NN Pred first and then actual value
    optimizer.zero_grad()    # Zero Out the gradient
    loss.backward()
    optimizer.step()
    print(f"Epoch: {it + 1:02}/{Niter_Adam} Loss: {loss:.5e}")
    y_train_pred = Net(x)
    scheduler.step()
    lr_pow.append(optimizer.param_groups[0]['lr'])
```



Some PyTorch in-built learning rate scheduler

i) torch.optim.lr_scheduler.MultiplicativeLR
ii) torch.optim.lr_scheduler.LinearLR
iii) torch.optim.lr_scheduler.ExponentialLR

Learning Rate Scheduling in DNN

1. Power scheduling: user defined

```
def lr_poly(base_lr, iter, max_iter, power):
    return base_lr*((1-float(iter)/max_iter)**(power))
```

```
1  optimizer = tf.keras.optimizers.SGD(learning_rate=lr_poly)
```

Some TF2.0 in-built learning rate scheduler

i) tf.keras.optimizers.schedules.ExponentialDecay

ii) tf.keras.optimizers.schedules.CosineDecay

iii) tf.keras.optimizers.schedules.PolynomialDecay

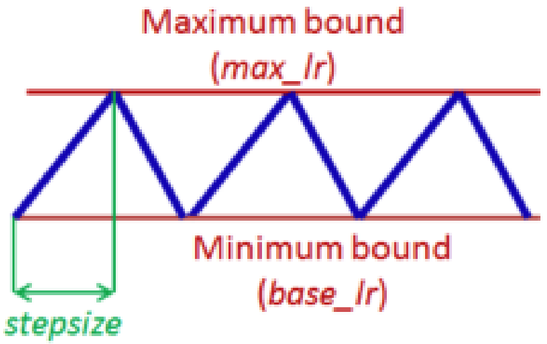
Learning Rate Scheduling- 1 Cycle Scheduling

❑ A minimum and maximum bound is set and the learning rate cyclically varies between them.

❑ The function according to which the rate changes does not affect significantly the performance and thus select a triangular window.

❑ The reason for doing it is to avoid tuning the learning rate and reduce the number of training epochs for optimization purposes.

❑ Stepsize is taken as 2 −10 times the number of iterations in an epoch.

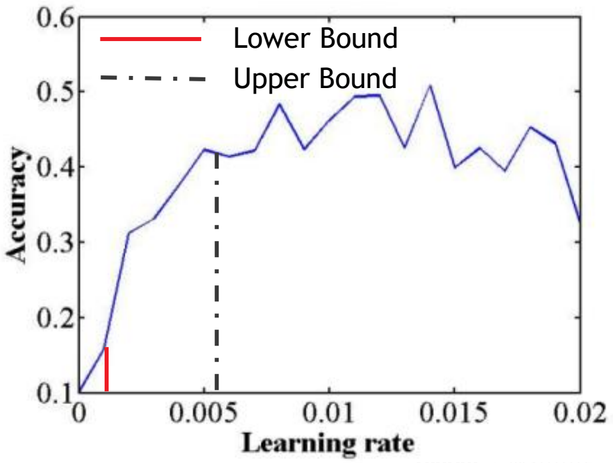


❑ Bounds are selected by running the optimization algorithm for a few epochs with linearly increasing learning rate.

❑ The lower bound can be selected as the rate at which an increase to the accuracy is observed.

❑ The upper bound when accuracy starts getting ragged or decreasing, e.g,

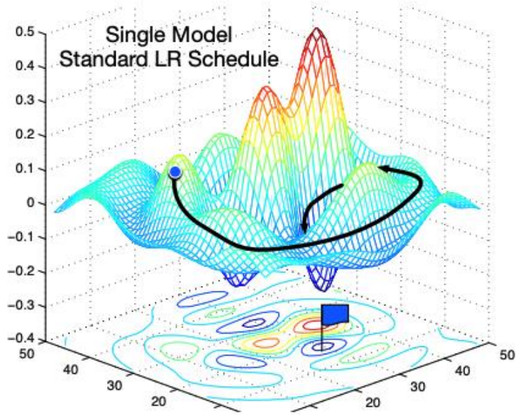
Lower Bound on learning rate = 0.001
Upper bound on learning rate = 0.006



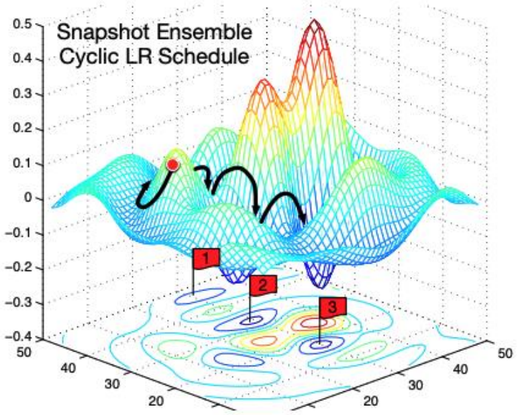
- Smith LN. Cyclical learning rates for training neural networks. In2017 IEEE winter conference on applications of computer vision (WACV) 2017 Mar 24 (pp. 464-472). IEEE.

Learning Rate Scheduling- 1 Cycle Scheduling

- SGD optimization with a typical learning rate schedule converges to a minimum at the end of training.



- 1 Cycle Scheduling let model undergoes several learning rate annealing cycles, converging to and escaping from multiple local minima.

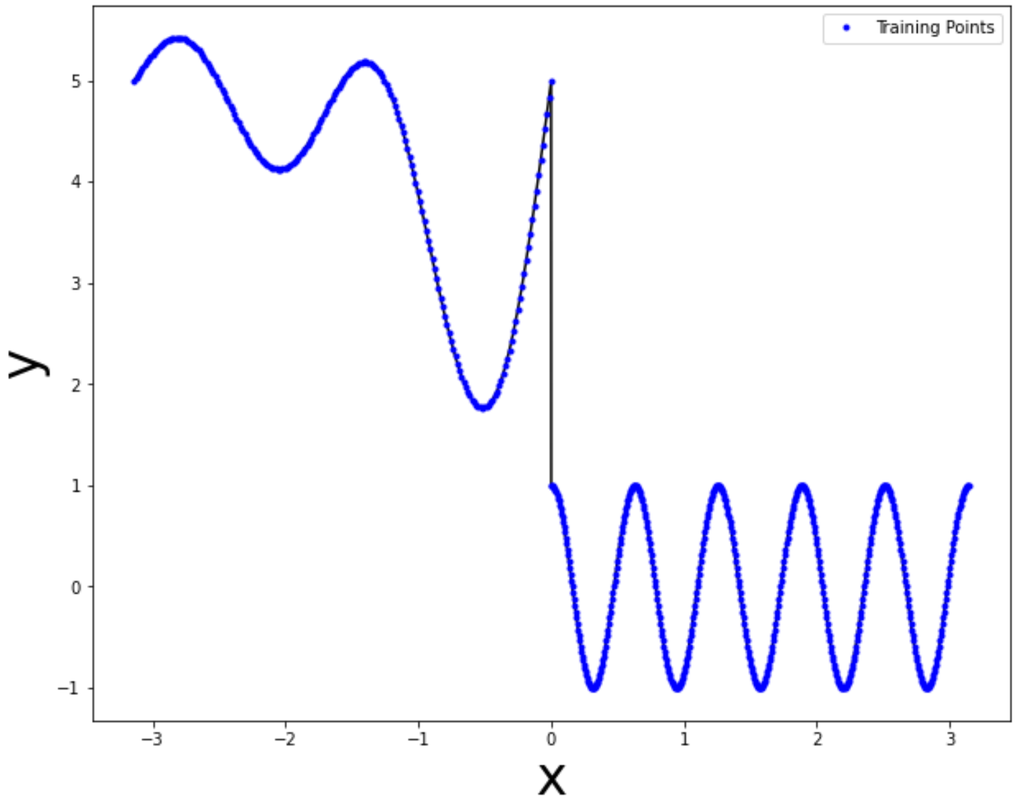


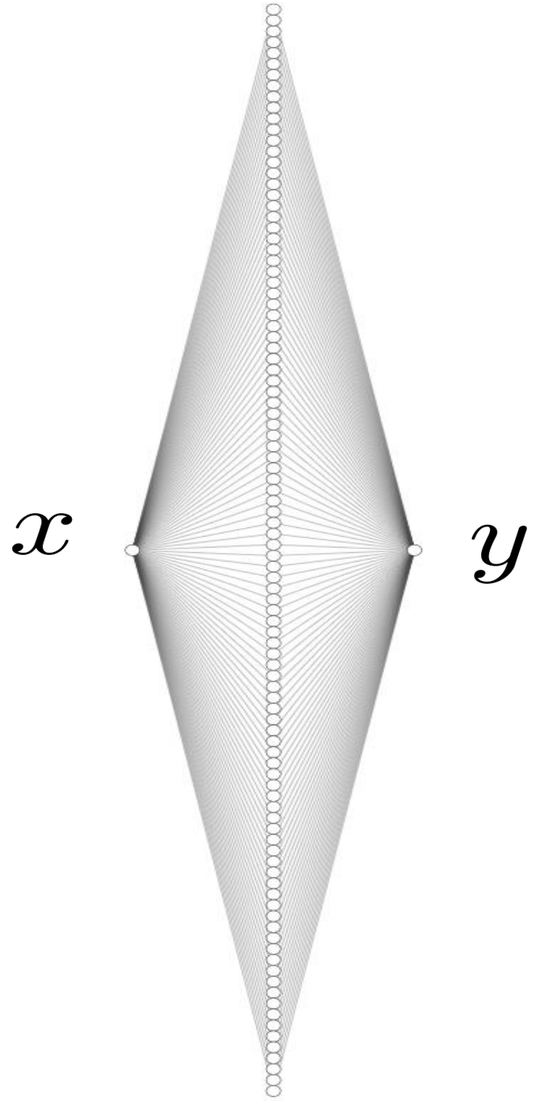
Huang, Gao, et al. "Snapshot ensembles: Train 1, get m for free." *arXiv preprint arXiv:1704.00109* (2017).

Learning Rate Scheduling- 1 Cycle Scheduling

$$y = 5 + \sum_{k=1}^{4} \sin(kx), \ x < 0$$

$$y = \cos(10x), \ x \geq 0$$





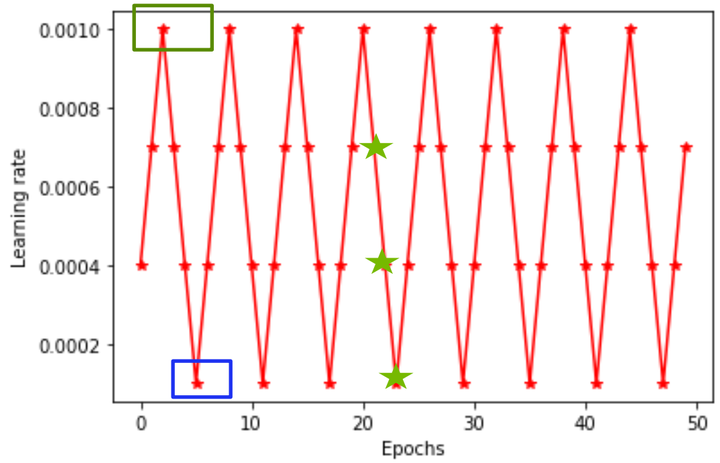
Learning Rate Scheduling- 1 Cycle Scheduling

API + Arguments

1.

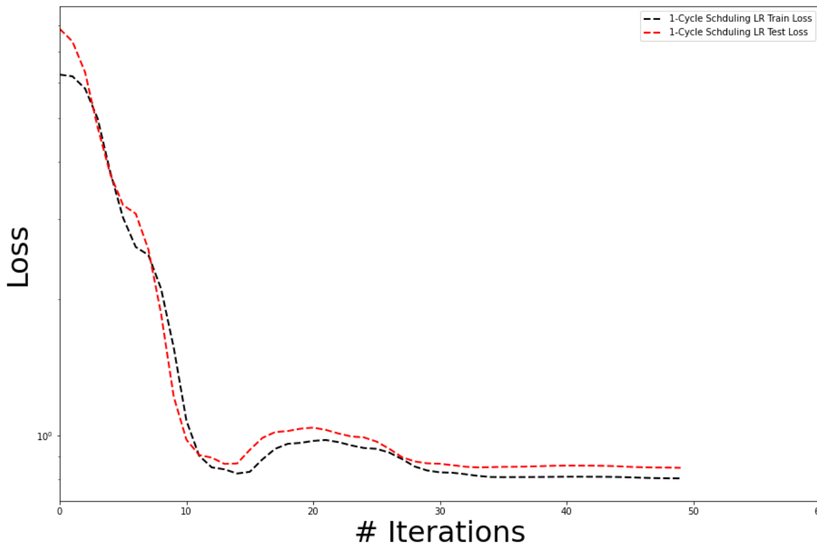
```
optimizer = SGD(Net.parameters(), lr = lr_initial)

scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.0001,\
                        max_lr=0.001,step_size_up=3,mode="triangular")
```

★ ★ ★



Training Code

```
loss_function = torch.nn.MSELoss()
Niter_Adam =  50

Net = torch.nn.Sequential(
      torch.nn.Linear(1, 100),
      torch.nn.ReLU(),
      torch.nn.Linear(100, 1))

optimizer = SGD(Net.parameters(), lr = lr_initial)
scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.0001,\
                        max_lr=0.001,step_size_up=3,mode="triangular")
train_loss_list = []
test_loss_list = []

acc_train = []
acc_test = []
x = Variable(x)
y = Variable(y)
lr_lcs = []



for it in range(Niter_Adam):
    y_pred = Net(x)
    loss = loss_function(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    print(f"Epoch: {it + 1:02}/{Niter_Adam} Loss: {loss:.5e}")
    y_train_pred = Net(x)
    scheduler.step()
    lr_lcs.append(optimizer.param_groups[0]['lr'])
```



Learning Rate Scheduling- 1 Cycle Scheduling

- In TF2.0 the 1 Cycle scheduling is available through *tensorflow_addons*
- To install and import it please do:

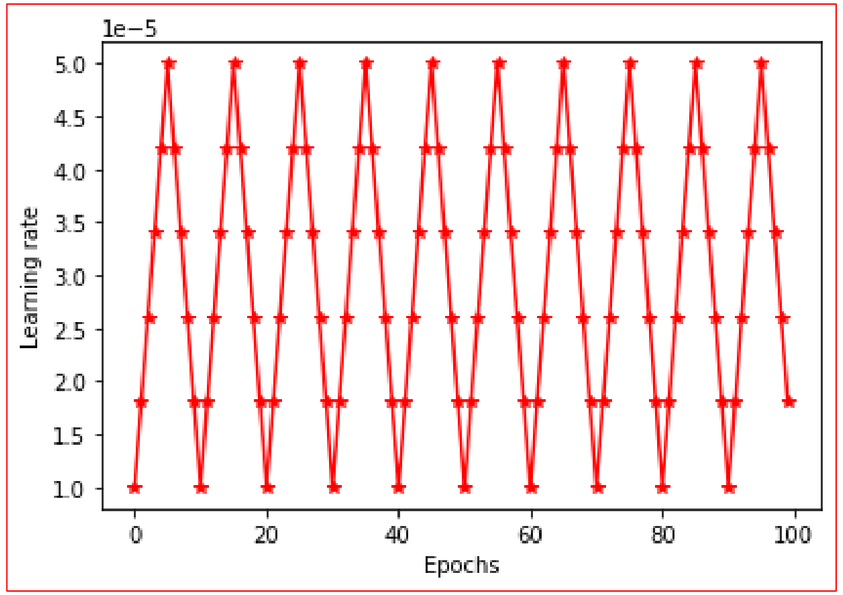
1.

```
!pip install -q -U tensorflow_addons
```

```
import tensorflow_addons as tfa
```

2.

```
start_lr = 0.00001
min_lr = 0.00001
max_lr = 0.00005
rampup_epochs = 5
Nepoch = 100

clr = tfa.optimizers.CyclicalLearningRate(initial_learning_rate=start_lr,
    maximal_learning_rate=max_lr,
    scale_fn=lambda x: 1,
    step_size=rampup_epochs)

ep = np.arange(0, Nepoch)
lr = clr(ep)
plt.plot(ep, lr, "-*r")
plt.xlabel("Epochs")
plt.ylabel("Learning rate")
```



Finally pass this learning rate scheduler (`clr`) to optimizer

3.

```
import tensorflow as tf
opt = tf.optimizers.Adam(learning_rate = clr)
```

Loss Regularizers L^2

- Regularization strategy is used to reduce test errors for new inputs but may increase the training errors.
- Loss function $\mathcal{L}$ with L^2 regularizer is expressed as

$$\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- After taking the gradient

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, y) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, y)$$

- Single gradient step to update the weights is expressed as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta(\alpha \boldsymbol{w} + \nabla_w \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}])$$

- After rearranging the term in above expression

$$\boldsymbol{w} \leftarrow (1 - \eta\alpha)\boldsymbol{w} - \eta \nabla_w \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

Loss Regularizers L^2: Alternative View

- The sum of squared error between the output of neural networks and actual value is expressed as

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{Y})^\top (\boldsymbol{X}\boldsymbol{w} - y)$$

- After applying the L^2- regularizers

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{Y})^\top (\boldsymbol{X}\boldsymbol{w} - y) + \frac{1}{2}\alpha \boldsymbol{w}^\top \boldsymbol{w}$$

- This changes the normal equation of solution

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

to

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

Loss Regularizers L^2: Alternative View

1. Linear Regression analysis: The matrix $\boldsymbol{X}^\top \boldsymbol{X}$ is proportional to the covariance matrix.

2. Using L^2 regularization replaces this matrix with $(\boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{\alpha} \boldsymbol{I})^{-1}$.

3. The new matrix is the same as the original one, but with the addition of $\boldsymbol{\alpha}$ to the diagonal.

4. The diagonal entries of this matrix correspond to the variance of each input feature.

5. We can see that L^2 regularization causes the learning algorithm to "perceive" the input $\boldsymbol{X}$ as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added variance.

6. L^2 regularization (weight decay) is also known as ridge regression or Tikhonov regularization.

7. L^2 regularization is equivalent to MAP Bayesian inference with a Gaussian prior on the weights.

Loss Regularizers: L^2

```
optimizer_adam = torch.optim.Adam(Net.parameters(), lr = 0.01, weight_decay=1e-5)
```

```
tf.nn.l2_loss(v)

loss = (tf.reduce_mean(tf.squared_difference(prediction, Y))) + 1e-05*tf.nn.l2_loss(hidden_weights) + 1e-05*tf.nn.l2_loss(out_weights))
```

weight_decay=1e-5 = $\frac{\alpha}{2}\boldsymbol{w}^\top\boldsymbol{w}$ with $\alpha = 10^{-5}$

Note: A detailed implementation in both frameworks is provided in jupyter notebook associated with the module.

Loss Regularizers: L^1

- The L^1 regularization model parameter $\boldsymbol{w}$ is defined as

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1$$

- Thus, the regularized loss function $\boldsymbol{\theta}$ is expressed as

$$\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha||\boldsymbol{w}1||_1$$

- The gradient of L^1 regularized loss function is expressed as

$$\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{X}, y; \boldsymbol{w})$$

  where $\text{sign}(\boldsymbol{w})$ is sign $\boldsymbol{w}$ applied element-wise.

Loss Regularizers L^1: Alternative View

- We see that the effect of L^1 regularization is quite different from that of L^2 regularization.
- The regularization contribution to the gradient no longer scales linearly with each w_i; instead it is a constant factor with a sign equal to $\text{sign}(w_i)$.
- One consequence of this form of the gradient is that we will not necessarily see clean algebraic solutions to quadratic approximations of $\mathcal{L}(X, y; w)$ as we did for L^2 regularization.
- In comparison to L^2 regularization, L^1 regularization results in a solution that is more sparse. Sparsity in this context refers to the fact that some parameters have an optimal value of zero.
- The sparsity property induced by L^1 regularization has been used extensively as a feature selection mechanism.

Loss Regularizers: L^1

There is no direct API but we can define the function and pass it to loss.

```
loss_function = torch.nn.MSELoss()
Niter_Adam = 30000

Net = torch.nn.Sequential(
      torch.nn.Linear(1, 100),
      torch.nn.ReLU(),
      torch.nn.Linear(100, 1))

optimizer_adam = torch.optim.Adam(Net.parameters(), lr = 0.01, weight_decay=1e-5)


x = Variable(x)
y = Variable(y)

def l1_penalty(model, l1_lambda=1e-05):
    l1_norm = sum(p.abs().sum() for p in model.parameters())
    return l1_lambda*l1_norm


# Implementation of Adam
print('Adam Optimization')

for it in range(Niter_Adam):
    y_pred = Net(x)
    loss = loss_function(y_pred, y) + + l1_penalty(Net, l1_lambda=1e-05)
    optimizer_adam.zero_grad()     # zero out the gradient
    loss.backward()
    optimizer_adam.step()
    print(f"Epoch: {it + 1:02}/{Niter_Adam} Loss: {loss:.5e}")
    y_train_pred = Net(x)
```

sub-routine

Loss + L1 - term

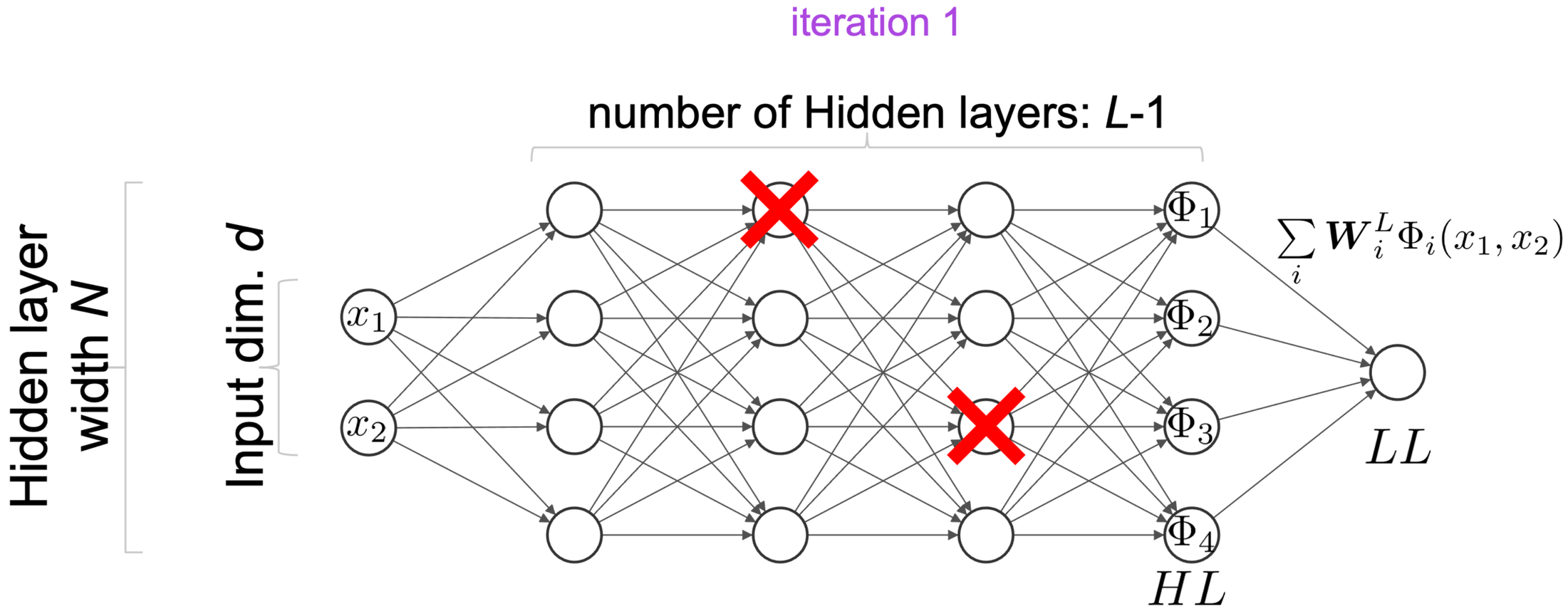
```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
reg_constant = 0.01 # Choose an appropriate one.
loss = my_normal_loss + reg_constant * sum(reg_losses)
```

Note: A detailed implementation in both frameworks is provided in jupyter notebook associated with the module.

Dropout is an Effective Regularizer

- Dropout was proposed in a paper by Hinton in 2012 and it has proven to be very effective.
- Even the state-of-the-art neural networks could get a 2% accuracy boost by Dropout.
- Dropout creates ensembles of many different neural networks, hence it trains multiple models, and evaluates multiple models on each test case.
- This is typically impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of GPU time and memory, but Dropout provides an inexpensive method for training and evaluating an ensemble of exponentially many neural networks.
- Srivastava *et al.* (2014) demonstrated that Dropout is more effective than weight decay and sparse activity regularization. Dropout may also be combined with other forms of regularization to yield a further improvement.
- Wager *et al.* (2013) demonstrated that for *linear regression*, Dropout is equivalent to *L2* weight decay, with a different weight decay coefficient for each input feature. The magnitude of each feature's weight decay coefficient is determined by its variance. However, for deep learning models, Dropout is not equivalent to weight decay.

Dropout


iteration 1
number of Hidden layers: L-1
Hidden layer width N
Input dim. d
x_1
x_2
Φ_1
Φ_2
Φ_3
Φ_4
HL
$\sum_i \boldsymbol{W}_i^L \Phi_i(x_1, x_2)$
LL

Dropout Training

- To train a DNN with Dropout, we use a mini-batch learning algorithm that makes small steps, e.g., SGD.
- When we load an example into a mini-batch, we randomly sample a different binary mask to apply to all of the input units and hidden units in the DNN.
- The mask for each unit is sampled independently from all of the others. The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter p fixed before training begins – this is the *dropout rate*. It is typically set between 0.1 and 0.5, closer to 0.2–0.3 in recurrent neural nets, and closer to 0.4–0.5 in convolutional neural networks.
- Since each neuron can be either present or absent, there are a total of 2^N possible networks (where *N* is the total number of droppable neurons). This is a huge number that it is impossible for the same DNN to be sampled twice.
- Once we have run 10,000 training steps, it is like we trained 10,000 different DNN, each with just one training instance.

Dropout Training

- These neural networks are obviously not independent because they share many of their weights, but they are all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.
- In practice, we usually apply Dropout only to the neurons in the higher 1-3 layers but not to the output layer.
- Dropout is only active during training, hence we should not try to compare the training loss and the validation loss. We should evaluate the training loss without Dropout after training.
- If we see that the model is overfitting, we can increase the Dropout rate or decreasing the Dropout rate if the model underfits the training set.
- It can also help to increase the Dropout rate for the top layers while reducing it for lower ones or use Dropout partially for some layers only.
- Dropout tends to slow down convergence, but it usually results in a more accurate model when tuned properly, hence it is worth the extra cost.

Example: Regularizers- 2 Hidden Layers

Hyperparametrs:

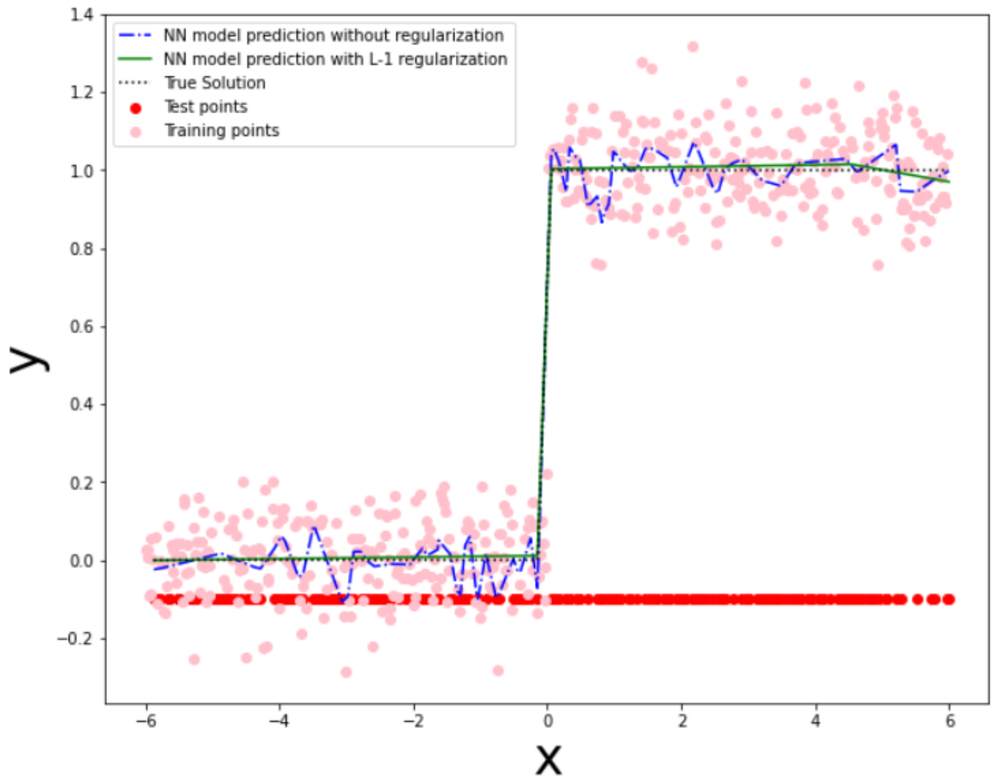
Number of Layers: 2
Number of neurons in each layer: 100
Nepochs: 30000
Learning rate: 0.01
Activation: ReLu

Heaviside function

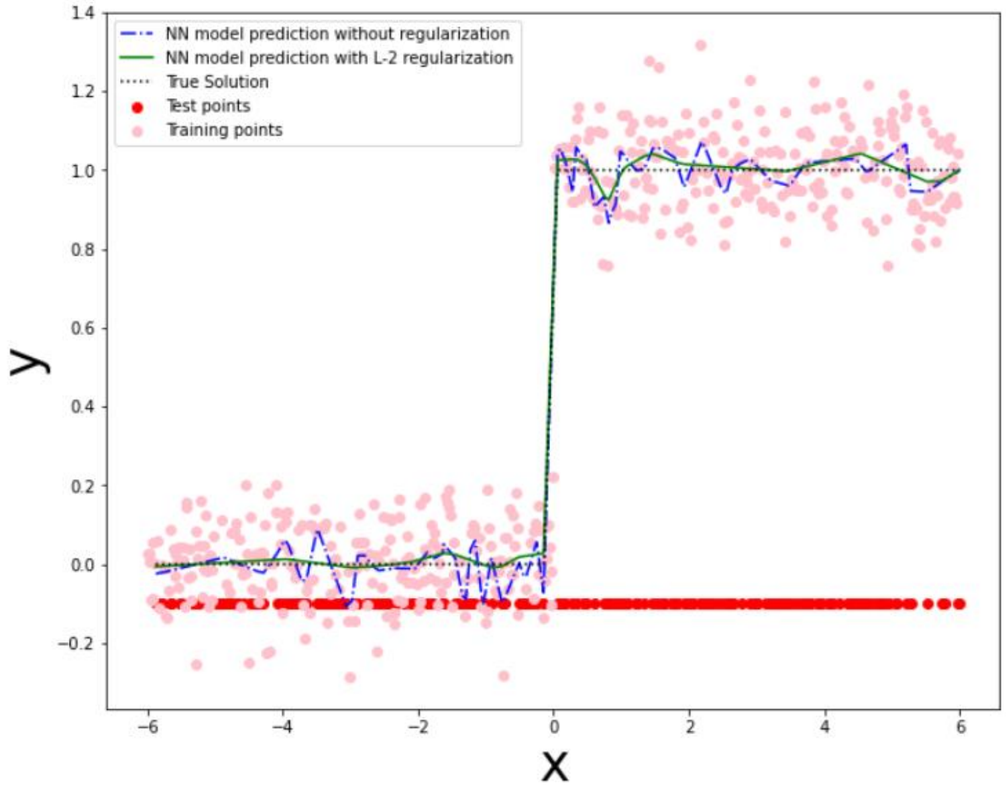
$$f(x) = 0, \qquad x < 0$$
$$= 1, \qquad x \geq 0$$

L1 seems to work better here

L_1- regularizer- $\alpha = 1\text{e} - 04$



L_2-regularizer $\alpha = 1\text{e} - 04$



Example: Regularizers- 3 Hidden Layers

Hyperparametrs:

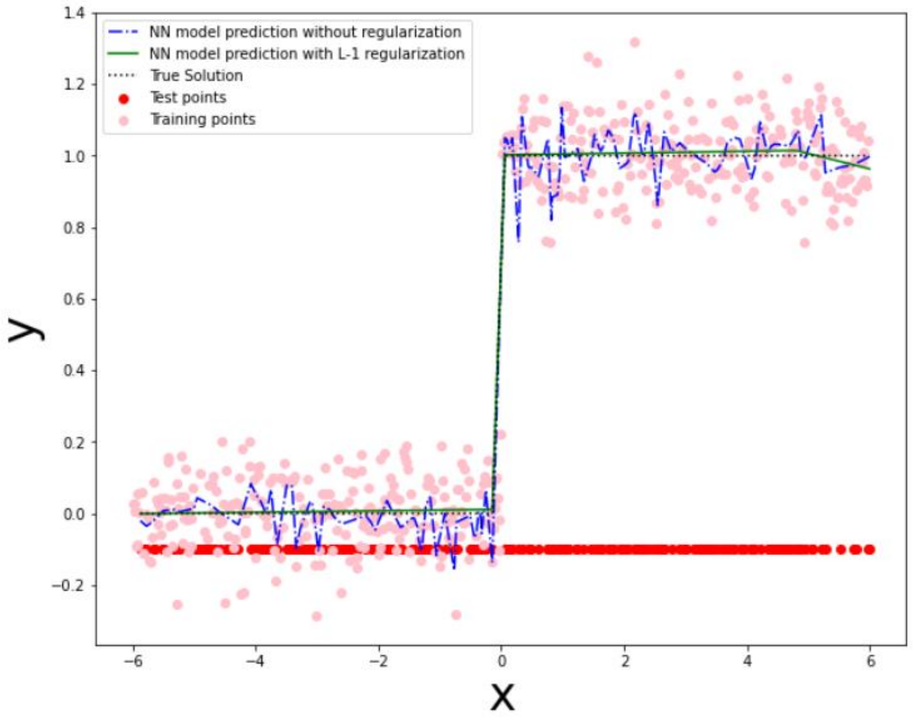
Number of Layers: 3
Number of neurons in each layer: 100
Nepochs: 30000
Learning rate: 0.01
Activation: ReLu

1. Heaviside function

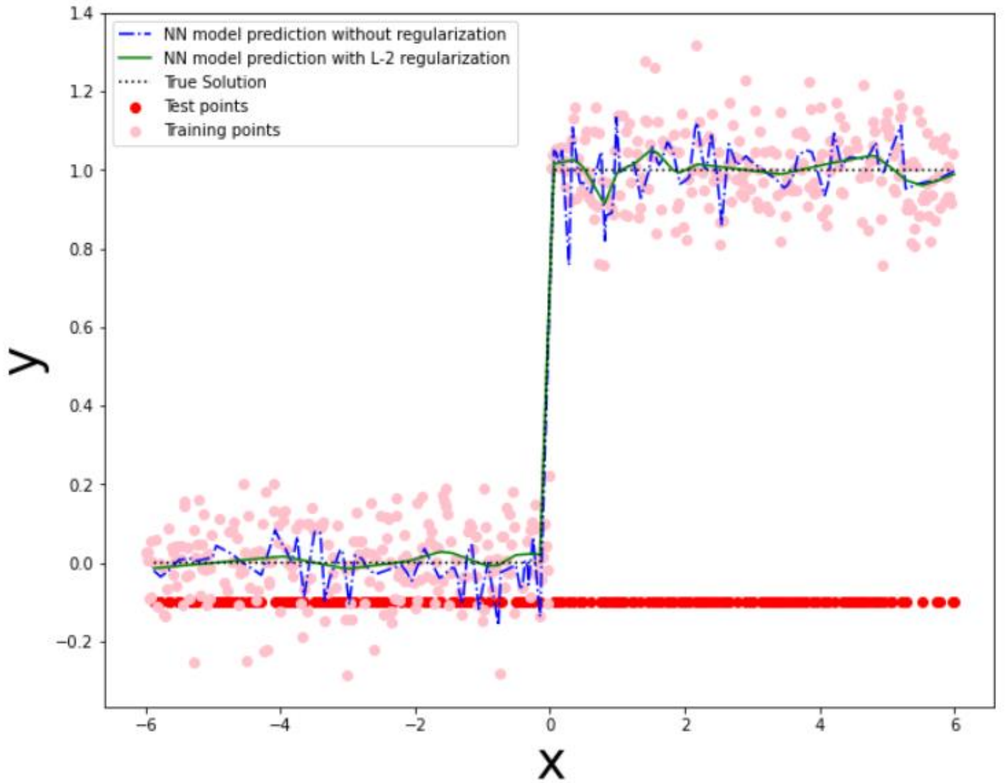
$$f(x) = 0, \qquad x < 0$$
$$= 1, \qquad x \geq 0$$

L1 seems to work better here

L_1- regularizer



L_2-regularizer



Example: Dropout Regularization

Heaviside function

$$f(x) = 0, \qquad x < 0$$
$$= 1, \qquad x \geq 0$$

Hyperparametrs:

Number of Layers: 2(Left) and 3 (Right)
Number of neurons in each layer: 100
Nepochs: 30000
Learning rate: 0.01
Activation: ReLu

Dropout Probabilities

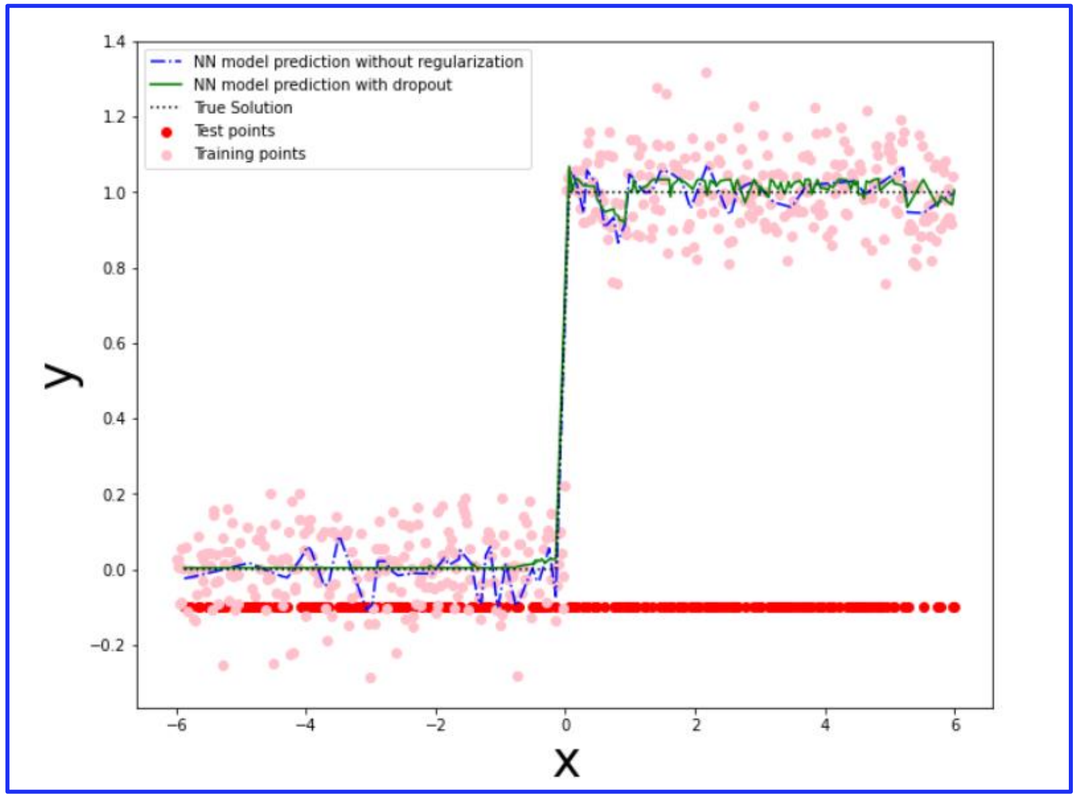
Layer 1: 0.75
Layer 2: 0.01

3 layers seem to work better!

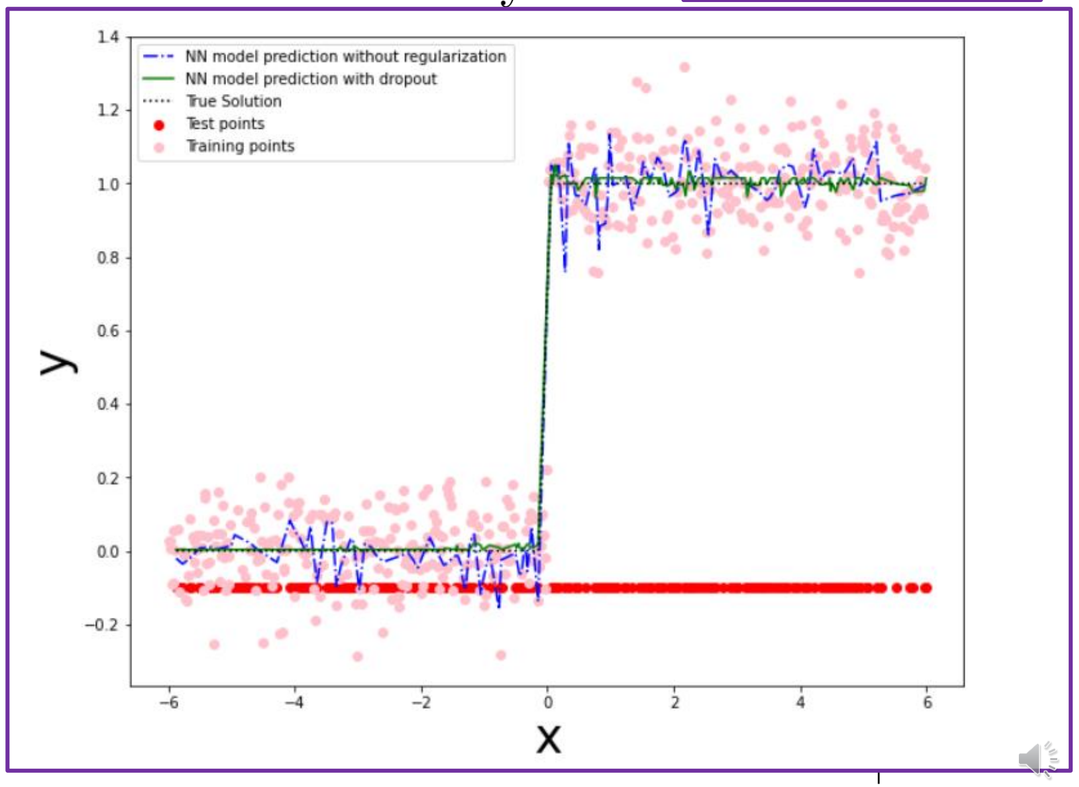
Dropout Probabilities

Layer 1: 0.75
Layer 2: 0.6
Layer 3: 0.01

2 Layers



3 Layers



Collapse of Deep and Narrow ReLU Neural Networks

Training of NNs

- NP-hard [Sima, 2002]
- Local minima [Fukumizu & Amari, 2002]
- Bad saddle points [Kawaguchi, 2016]

ReLU

- Dying ReLU neuron: stuck in the negative side

Deep ReLU nets?

Dying ReLU network

NN is a constant function after initialization

Collapse

NN converges to the "mean" state of the target function during training

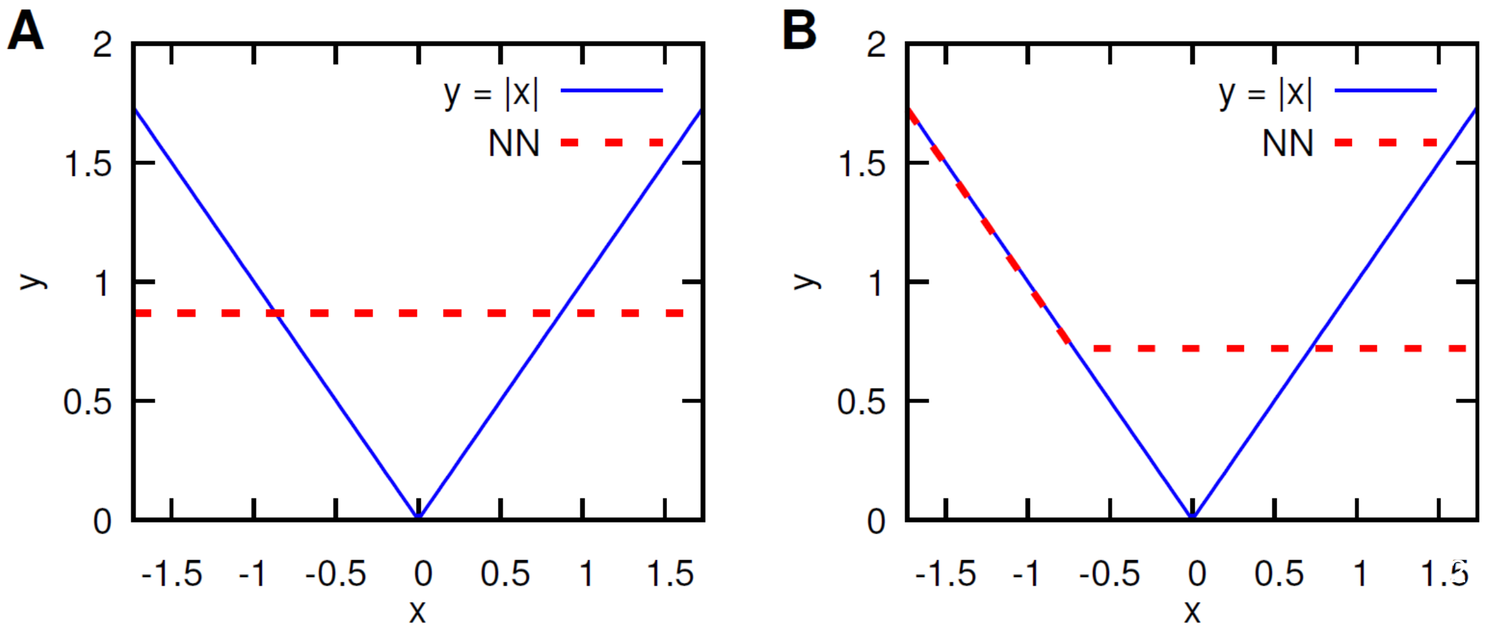
ReLU Collapse: One-Dimensional Examples

$$f(x) = |x|$$

- $|x| = \text{ReLU}(x) + \text{ReLU}(-x) = \begin{bmatrix} 1 & 1 \end{bmatrix} \text{ReLU}(\begin{bmatrix} 1 \\ -1 \end{bmatrix} x)$
- 2-layer with width 2

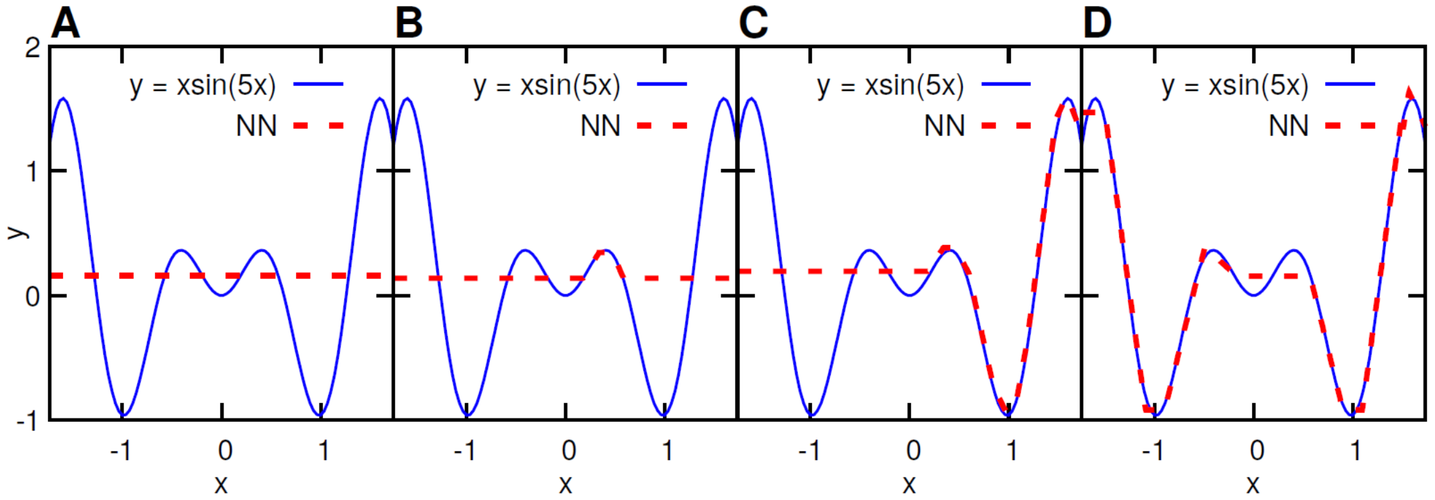
Train a 10-layer ReLU NN with width 2 (MSE loss, whatever optimizer)

- Collapse to the mean value (A): $\sim$93%
- Collapse partially (B)

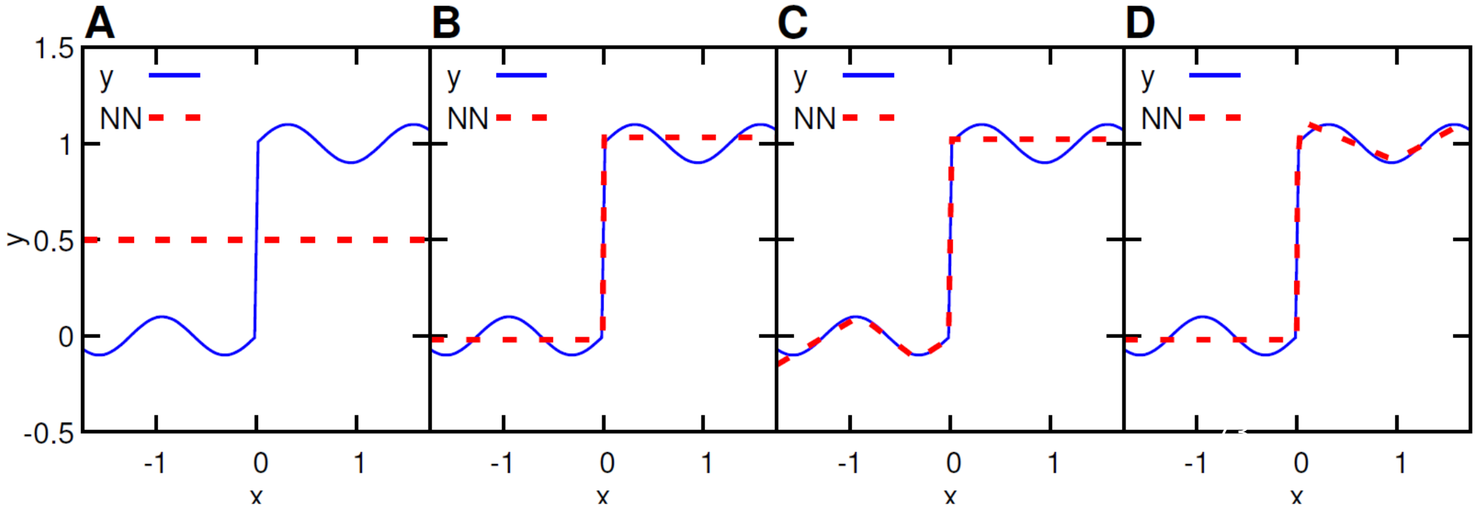


ReLU Collapse: One-Dimensional Examples

$$f(x) = x \sin(5x)$$

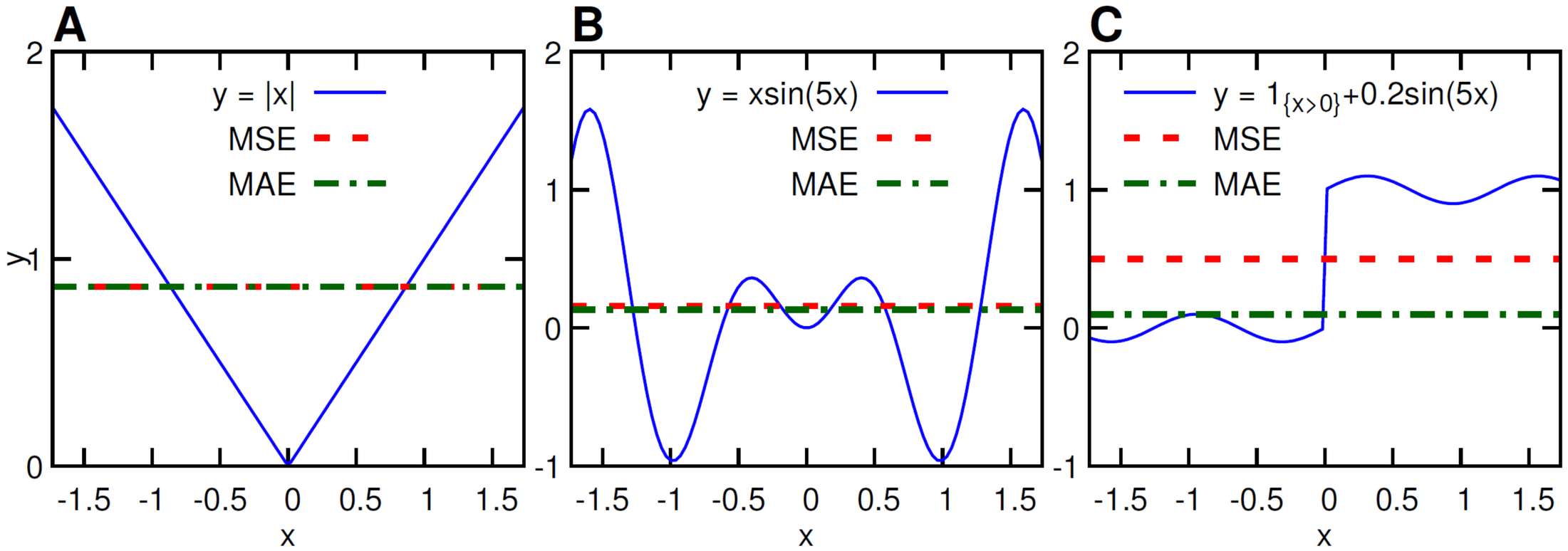


$$f(x) = 1_{\{x>0\}} + 0.2 \sin(5x)$$



Does the Loss Type Matter?

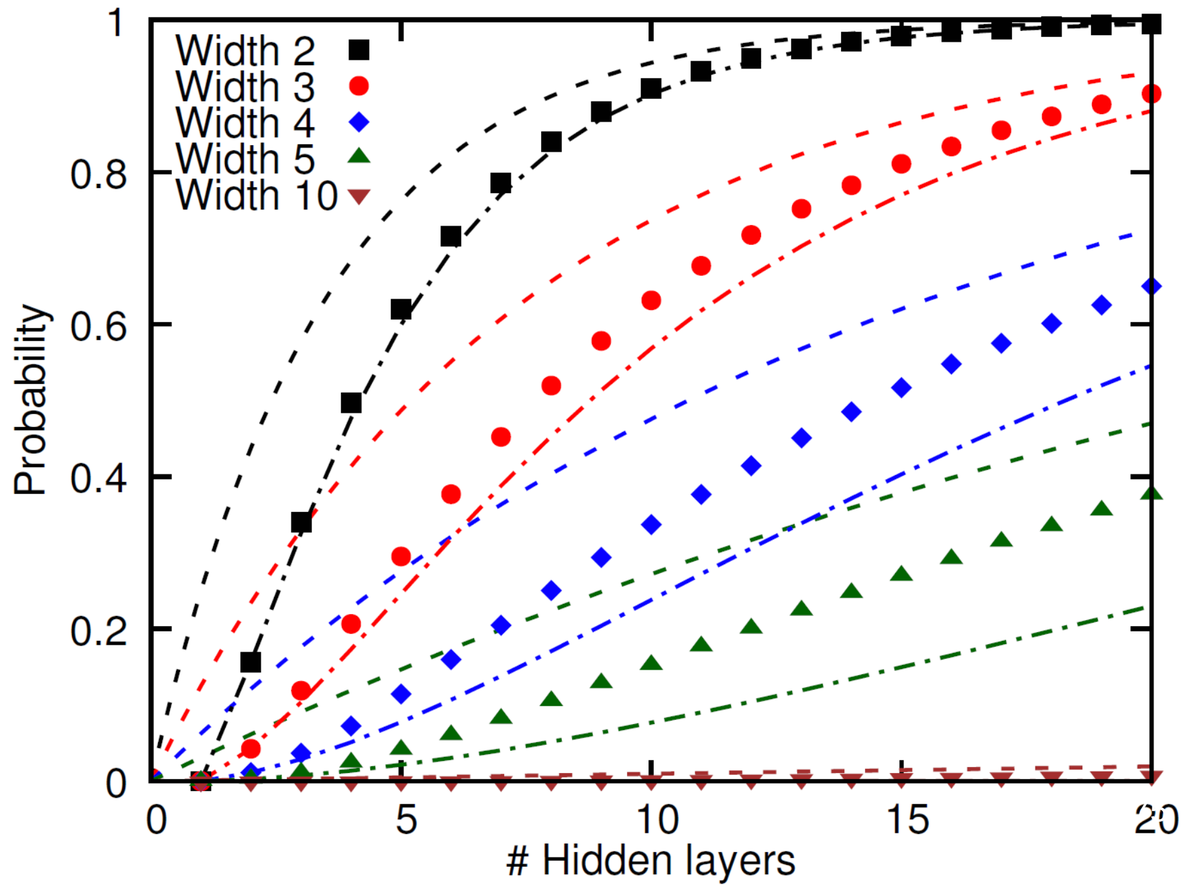
- Mean squared error (MSE) $\Rightarrow$ mean
- Mean absolute error (MAE) $\Rightarrow$ median



Theoretical versus Numerical Results

- A ReLU NN with $d_{in} = 1$
- Weights randomly initialized from symmetric distributions
- Biases are initialized to 0

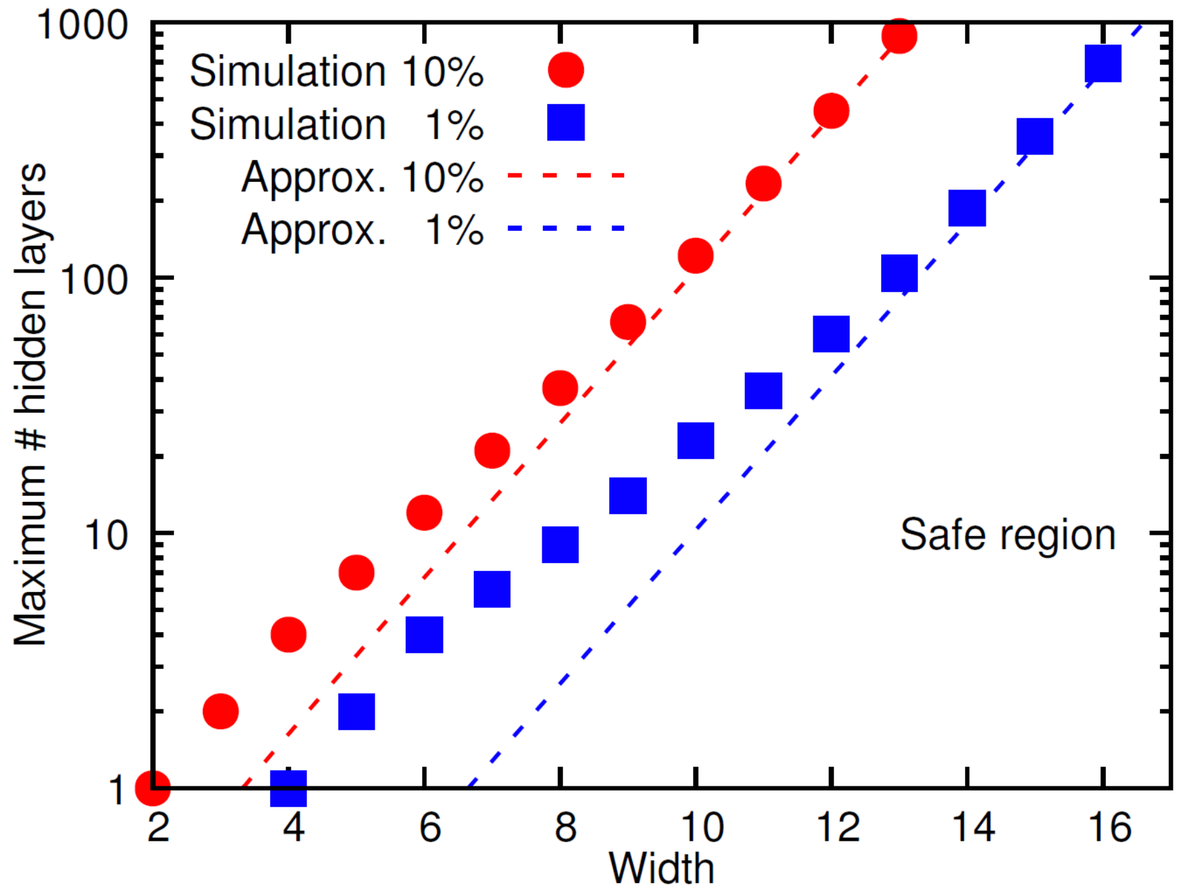
More likely to die when it is deeper and narrower



Theoretical versus Numerical Results

Safe Operating Region for a ReLU NN

Keep the dying probability $< 10\%$ or 1%



Summary

- The main difficulty in training a NN is due to back propagation and the need to minimize a high-dimensional non-convex function.
- Stochastic gradient descent (SGD) and mini-batch GD with decaying learning rate are effective and practical methods.
- Underfitting occurs for a low-capacity model while overfitting occurs for a high-capacity model.
- The Xiavier and He initialization as well as the data normalization are key components of effective NN training.
- Batch normalization can lead to faster convergence, avoid vanishing gradients, allows higher learning rates and act as regularizer.
- There are many optimizers but the combination of Adams first, followed by L-BFGS is usually the winner.
- Decaying learning rate and cyclical learning rate are important in accelerating convergence and avoiding bad minima.
- L2 regularization leads to better generalization while L1 encourages sparsity; which one to use is problem-dependent.
- Dropout regularization is effective but depends stronlgly on the dropout rate, which should be variable across the layers.
- Information bottleneck theory suggests a phase transition in deep learning, from fitting to early layers to true learning in the higher layers.
- Deep neural networks are expressive but they can collapse during training yielding erroneous results, e.g. a deep and narrow ReLu network.

References

Cyr EC, Gulian MA, Patel RG, Perego M, Trask NA. Robust training and initialization of deep neural networks: An adaptive basis viewpoint. In Mathematical and Scientific Machine Learning 2020 Aug 16 (pp. 512-536). PMLR.

Dauphin Y., et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1-14, 2014.

Fukumizu K., Amari S.I. Local minima and plateaus in hierarchical structures of multilayer perceptrons, Neural networks, 13(3), 317-327, 2000.

Géron A. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media; 2019 Sep 5.

Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics 2010 Mar 31 (pp. 249-256). JMLR Workshop and Conference Proceedings.

Goldfeld Z, van den Berg E, Greenewald K, Melnyk I, Nguyen N, Kingsbury B, Polyanskiy Y. Estimating information flow in neural networks. In Proceedings of the International Conference on Machine Learning (ICML-2019), vol. Long Beach, California, USA, Jun. 2019.

He K, Zhang X, Ren S, Sun J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision 2015 (pp. 1026-1034).

Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov RR. Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580. 2012 Jul 3. Wager S, Wang S, Liang PS. Dropout training as adaptive regularization. Advances in neural information processing systems. 2013;26:351-9.

Huang G, Li Y, Pleiss G, Liu Z, Hopcroft JE, Weinberger KQ. Snapshot ensembles: Train 1, get m for free. arXiv preprint arXiv:1704.00109. 2017 Apr 1.

Kawaguchi K. Deep learning without poor local minima. arXiv preprint arXiv:1605.07110. 2016 May 23.

Kawaguchi K. On the Theory of Implicit Deep Learning: Global Convergence with Implicit Layers. arXiv preprint arXiv:2102.07346. 2021 Feb 15.

Kawaguchi K, Bengio Y. Depth with nonlinearity creates no bad local minima in ResNets. Neural Networks. 2019 Oct 1;118:167-74.

References

Kawaguchi K, Huang J. Gradient descent finds global minima for generalizable deep neural networks of practical sizes. In2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton) 2019 Sep 24 (pp. 92-99). IEEE.

Kawaguchi K, Kaelbling L. Elimination of all bad local minima in deep learning. In International Conference on Artificial Intelligence and Statistics 2020 Jun 3 (pp. 853-863). PMLR.

Kawaguchi K, Sun Q. A Recipe for Global Convergence Guarantee in Deep Neural Networks. In Proceedings of the AAAI Conference on Artificial Intelligence 2021 Apr 12 (Vol. 35, No. 9, pp. 8074-8082).

Liu DC, Nocedal J. On the limited memory BFGS method for large scale optimization. Mathematical programming. 1989 Aug;45(1):503-28.

Lu, L., Su, Y., & Karniadakis, G. E. (2018), Collapse of deep and narrow neural nets, arXiv preprint arXiv:1808.04947.

Ruder, S. An overview of gradient descent optimization algorithms. arXiv preprint arXiv: 1609.04747, 2016.

Saxe, M, Bansal, Y, Dapello, J., Advani, M, Kolchinsky, A, Tracey, BD and Cox, D. On the information bottleneck theory of deep learning, in Proceedings of the International Conference on Learning Representations (ICLR), 2018.

Shwartz-Ziv R, Tishby N. Opening the black box of deep neural networks via information. arXiv preprint arXiv:1703.00810 (2017).

Sima J. Training a single sigmoidal neuron is hard, Neural computation. 14(11), 2709-2728, 2002.

Smith LN. Cyclical learning rates for training neural networks. In2017 IEEE winter conference on applications of computer vision (WACV) 2017 Mar 24 (pp. 464-472). IEEE.

Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research. 2014 Jan 1;15(1):1929-58.

Tishby N, Zaslavsky N. Deep learning and the information bottleneck principle. Information Theory Workshop (ITW), 2015 IEEE. IEEE, 2015.

Xu K, Zhang M, Jegelka S, Kawaguchi K. Optimization of graph neural networks: Implicit acceleration by skip connections and more depth. arXiv preprint arXiv:2105.04550. 2021 May 10.

Deep Learning for Science and Engineering Teaching Kit

Thank You