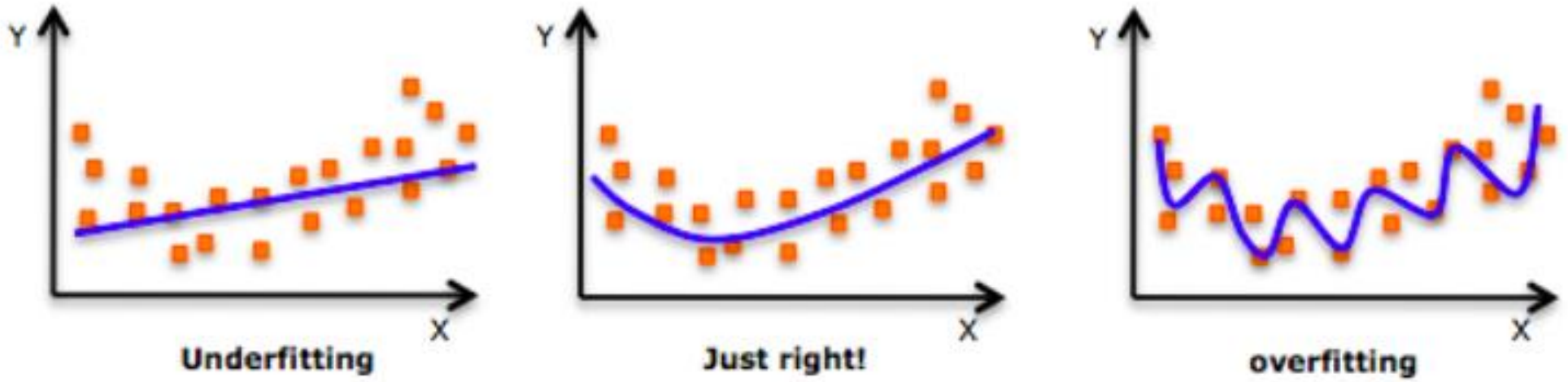
Ch-3-Regularization and Optimization for Training Deep Models

Lecture slides by

Prof. Sujata Pathak

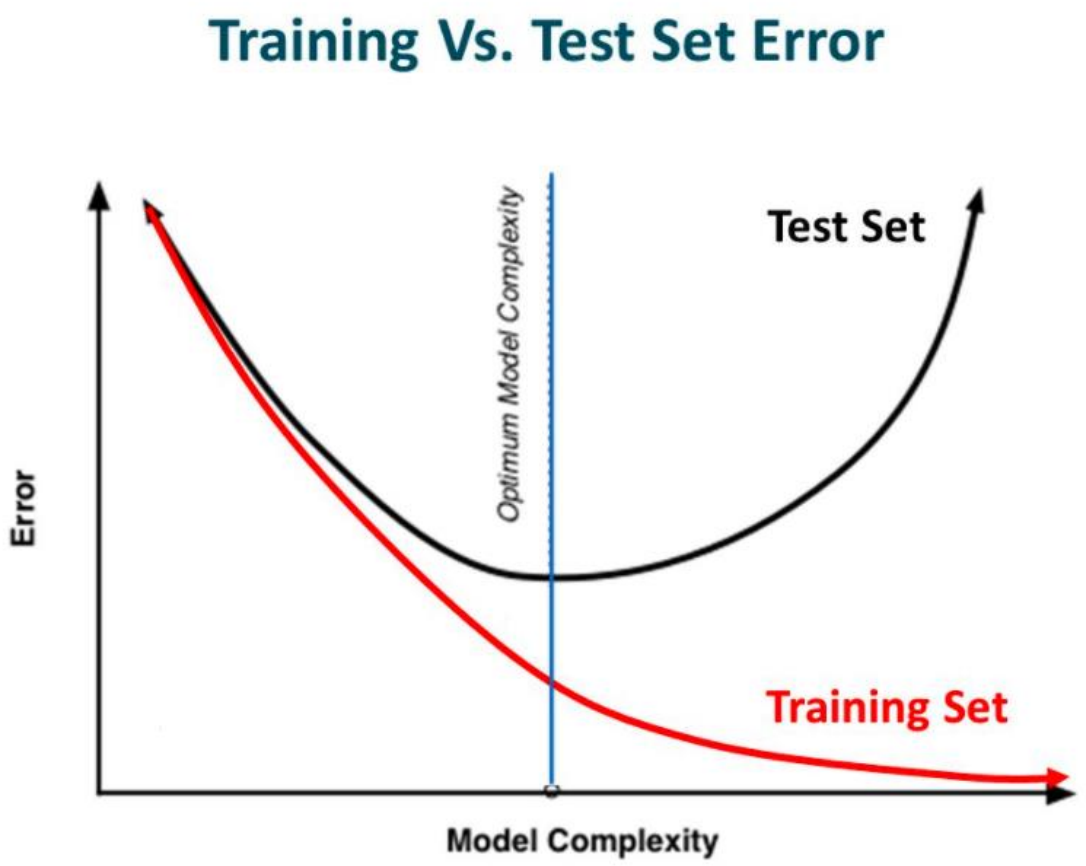
IT, KJSCE

What is regularization?



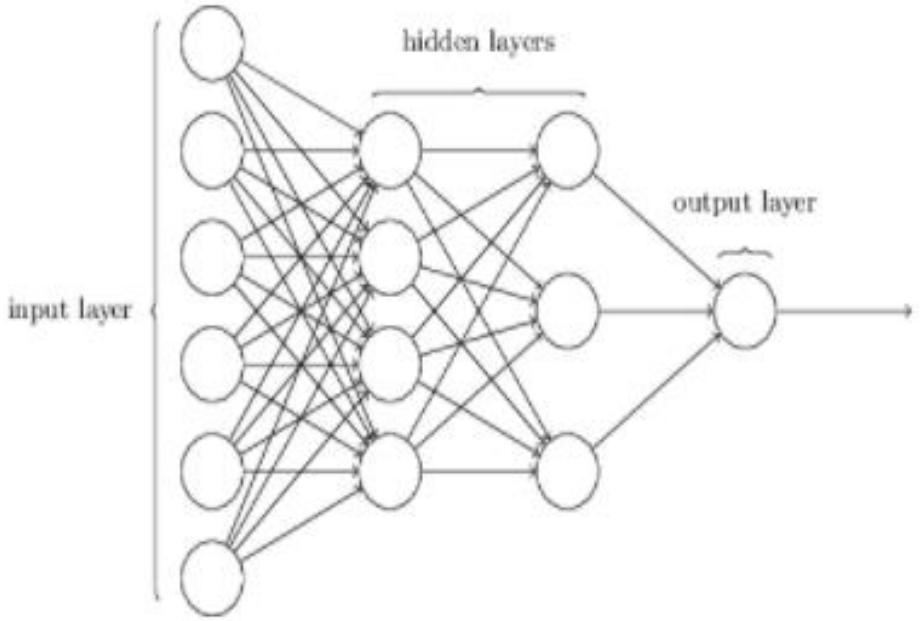
Overfitting causes the neural network model to perform very well during training, but the performance gets much worse during inference time when faced with brand new data.

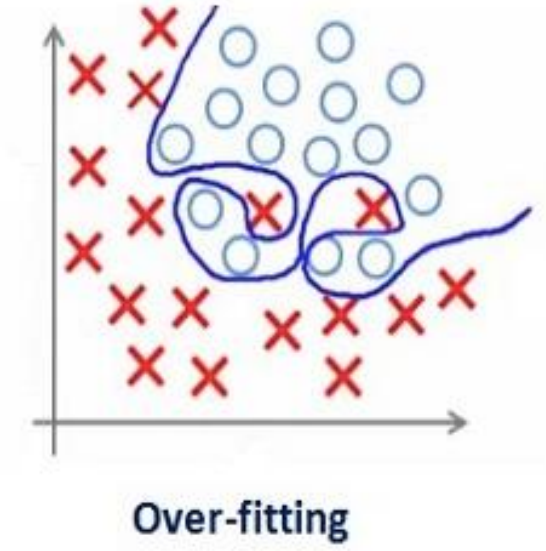
What is regularization?



- Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better.
- Regularization refers to a set of different techniques that lower the complexity of a neural network model during training, and thus prevent the overfitting.
- This in turn improves the model's performance on the unseen data as well.

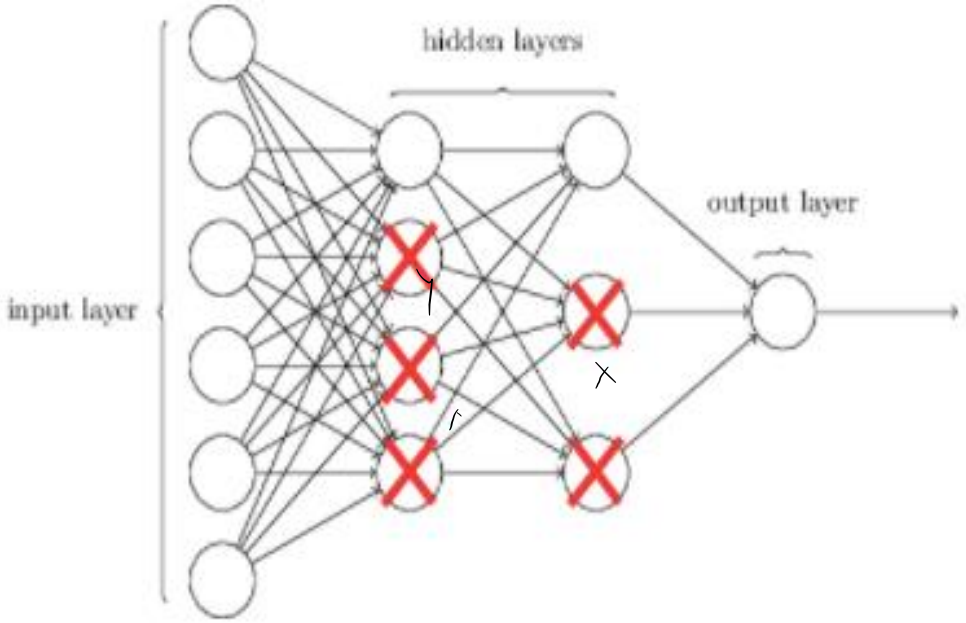
How does Regularization help reduce Overfitting?

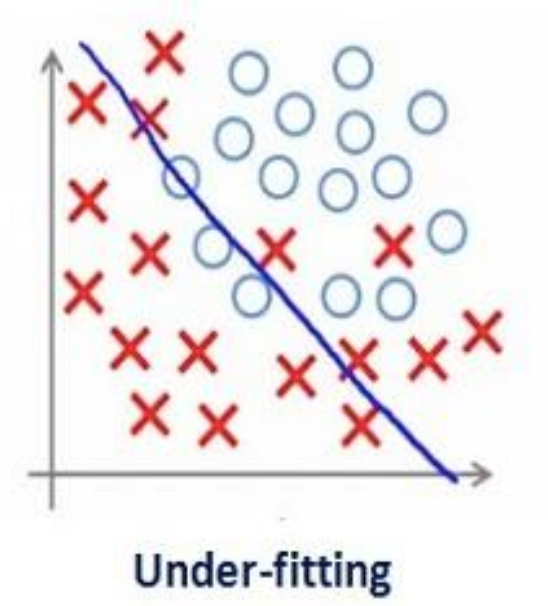




- Regularization penalizes the coefficients.
- In deep learning, it actually penalizes the weight matrices of the nodes.

How does Regularization help reduce Overfitting?





- Assume that our regularization coefficient is so high that some of the weight matrices are nearly equal to zero.
- This will result in a much simpler linear network and slight underfitting of the training data
- Optimize the value of regularization coefficient in order to obtain a well-fitted model

Different Regularization techniques in Deep Learning

- L2 and L1 regularization
- Dropout
- Data augmentation
- Early stopping

L2 Regularization

- Most common type of all regularization techniques
- Also commonly known as weight decay or ridge regression or Tikhonov regularization.
- During the L2 regularization the loss function of the neural network as extended by a so-called regularization term, which is called here Ω

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$$

L2 Regularization

- The regularization term Ω is defined as –
  - Euclidean Norm (or L2 norm) of the weight matrices, which is the sum over all squared weight values of a weight matrix.
- The regularization term is weighted by the scalar alpha divided by two and added to the regular loss function that is chosen for the current task.
- This leads to a new expression for the loss function:

$$\hat{\mathcal{L}}(W) = \frac{\alpha}{2}||W||_2^2 + \mathcal{L}(W) = \frac{\alpha}{2}\sum_i \sum_j w_{ij}^2 + \mathcal{L}(W)$$

Regularization loss during L2 regularization

L2 Regularization

- Alpha is sometimes called as the regularization rate.
- Additional hyperparameter we introduce into the neural network and determines how much we regularize our model.
- Compute the gradient of the new loss function and put the gradient into the update rule for the weights:

$$\nabla_W \hat{\mathcal{L}}(W) = \alpha W + \nabla_W \mathcal{L}(W)$$

$$W_{new} = W_{old} - \epsilon(\alpha W_{old} + \nabla_W \mathcal{L}(W_{old}))$$

Gradient Descent during L2 Regularization.

L2 Regularization

- Reformulate the update rule leading to the expression similar to update rule for the weights during regular gradient descent:

$$W_{new} = (1 - \epsilon\alpha)W_{old} - \epsilon\nabla_W \mathcal{L}(W_{old})$$

- The only difference is that by adding the regularization term we introduce an additional subtraction from the current weights (first term in the equation).
- In other words independent of the gradient of the loss function we are making our weights a little bit smaller each time an update is performed.

L1 Regularization

- also knows as Lasso regression
- Simply use another regularization term Ω.
- This term is the sum of the absolute values of the weight parameters in a weight matrix:

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

- Multiply the regularization term by alpha and add the entire thing to the loss function.

$$\hat{\mathcal{L}}(W) = \alpha||W||_1 + \mathcal{L}(W)$$

Loss function during L1 Regularization

L1 Regularization

- The derivative of the new loss function leads to the following expression, which the sum of the gradient of the old loss function and sign of a weight value times alpha.

$$\nabla_W \hat{\mathcal{L}}(W) = \alpha sign(W) + \nabla_W \mathcal{L}(W)$$

Gradient of the loss function during L1 Regularization.

What does Regularization achieve?

- Performing L2 regularization encourages the weight values towards zero (but not exactly zero)
- Performing L1 regularization encourages the weight values to be zero

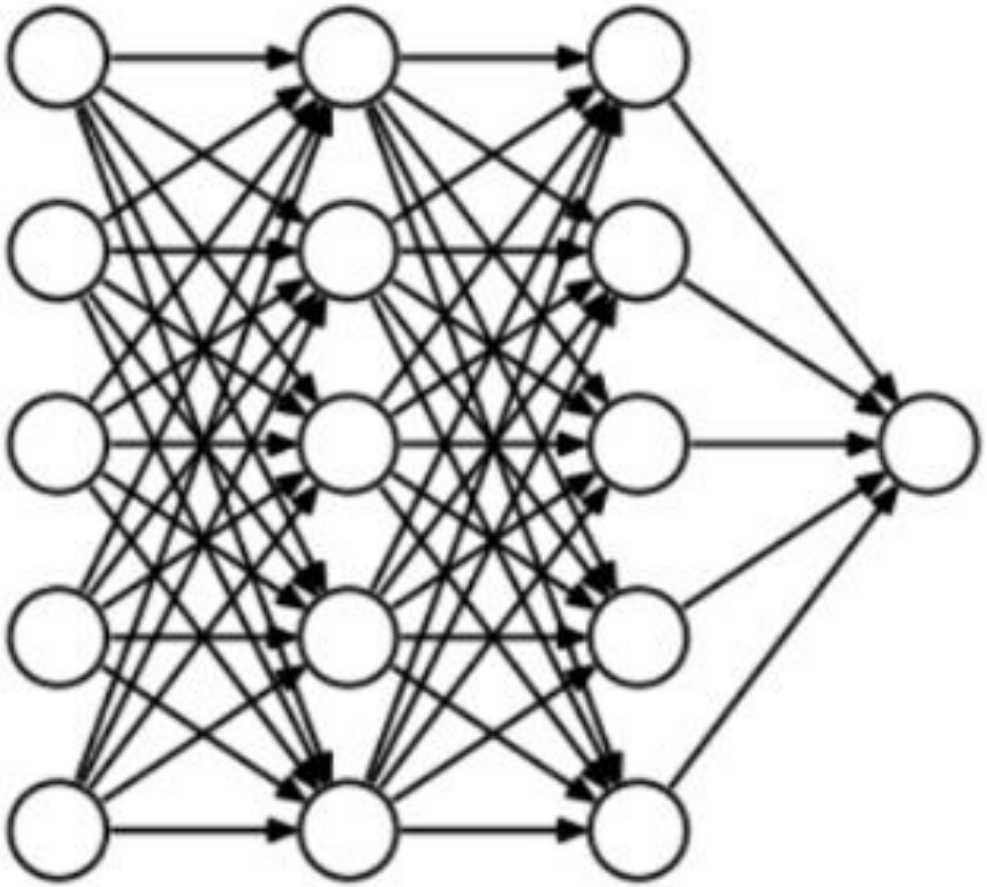
Selecting regularization term **α.**

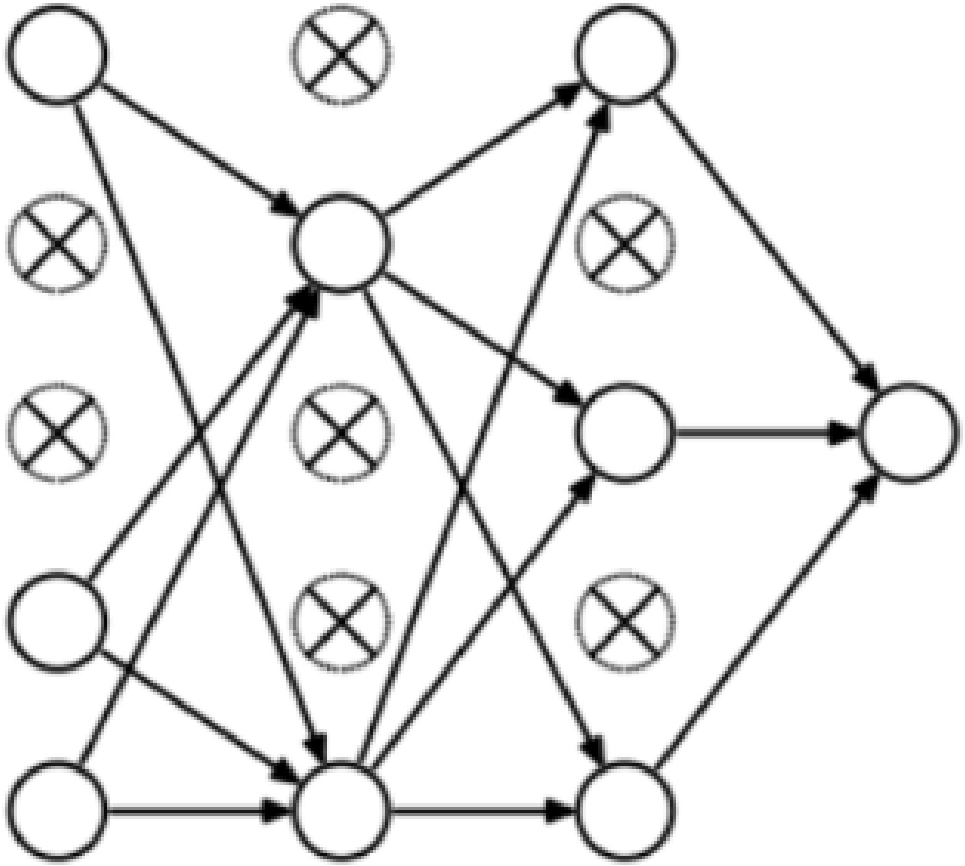
- If alpha value is too high, model will be simple, but you run the risk of underfitting your data. Your model won't learn enough about the training data to make useful predictions.
- If alpha value is too low, model will be more complex, and you run the risk of overfitting your data. Your model will learn too much about the particularities of the training data, and won't be able to generalize to new data.

Dropout

- Produces very good results
- Most frequently used regularization technique in the field of deep learning.
- During training with some probability P a neuron of the neural network gets turned off.
- Each iteration has a different set of nodes and this results in a different set of outputs.
- **It can also be thought of as an ensemble technique in machine learning.**

Dropout





Dropout

- Probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function.
- Dropout can be applied to both the hidden layers as well as the input layers.
- Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.
- During dropout, some neurons get deactivated with a random probability P → Neural network becomes less complex → less overfitting

Dropout

- In keras, implement dropout using the keras core layer.

```
from keras.layers.core import Dropout

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu')
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softma
 ])
```

Data Augmentation

- The simplest way to reduce overfitting is to increase the size of the training data.
- In machine learning- labeled data too costly.
- rotating the image, flipping, scaling, shifting, etc

- This technique is known as data augmentation.
- Provides a big leap in improving the accuracy of the model.
- Can be considered as a mandatory trick in order to improve our predictions.

Data Augmentation

- In *keras*, perform all of these transformations using ImageDataGenerator.

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

Early stopping

- Kind of cross-validation strategy where we keep one part of the training set as the validation set.
- When we see that the performance on the validation set is getting worse, we immediately stop the training on the model.
- This is known as early stopping.
- In keras, apply early stopping using the callbacks function


```
from keras.callbacks import EarlyStopping

EarlyStopping(monitor='val_err', patience=5)
```

Multi-Task Learning

Multi-task learning (MTL) is a learning paradigm where a model is trained on multiple tasks simultaneously. This improves generalization by leveraging shared knowledge across tasks.

- Way to improve generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks.
- The model can generally be divided into two kinds of parts and associated parameters:
  - Task-specific parameters (which only benefit from the examples of their task to achieve good generalization). These are the upper layers of the neural network
  - Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks). These are the lower layers of the neural network

  By sharing parameters and pooling data, the model can achieve better generalization for all tasks.

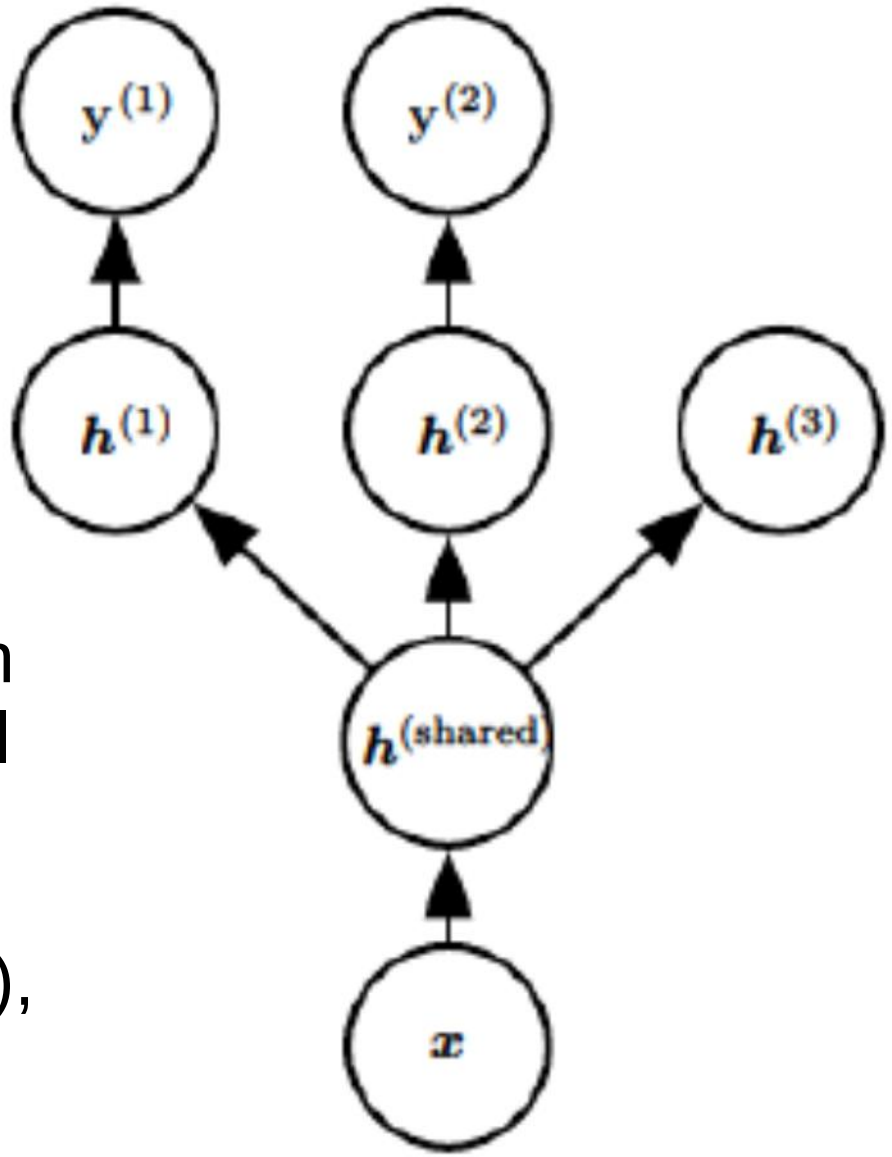
Multi-Task Learning

This slide illustrates the architecture of a neural network used for multi-task learning:

Input Node: x, representing the input data shared across tasks.

Shared Hidden Layer: h(shared), which learns generic features applicable to all tasks.

Task-Specific Hidden Layers: h(1), h(2), h(3), each designed to learn features specific to Task 1, Task 2, and Task 3, respectively.



Parameter Norm Penalties

- Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty Ω(θ) to the objective function J .
- We denote the regularized objective function by J˜:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

where α ∈ [0, ∞) is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J.

Setting α to 0 results in no regularization.

Larger values of α correspond to more regularization.

Regularization and Under-Constrained Problems

What Are Under-Constrained Problems?
These arise when there are more unknowns (parameters) than equations (constraints) in a system.
In machine learning, this often happens when the model has too much capacity (e.g., too many layers or neurons in a neural network) compared to the amount of training data.
Result: The model can overfit, memorizing the training data instead of generalizing to new data.

Role of Regularization
Regularization techniques prevent overfitting by adding penalties or constraints to the model:
L1 Regularization (Lasso):
- Adds a penalty proportional to the absolute value of weights.
- Encourages sparsity by driving some weights to zero.
L2 Regularization (Ridge):
- Adds a penalty proportional to the square of weights.
- Encourages small weights, reducing the complexity of the model.
Dropout:
- Randomly deactivates a fraction of neurons during training.
- Mimics ensemble learning by training multiple "sub-networks."
Early Stopping:
- Halts training when the performance on validation data stops improving.
- Prevents the model from overfitting to the training data.

Noise Robustness

- Noise applied to the inputs as a dataset augmentation strategy.
- Noise injection can be much more powerful than simply shrinking the parameters, especially when the noise is added to the hidden units
- Another way that noise has been used in the service of regularizing models is by adding it to the weights.
  - Used primarily in the context of recurrent neural networks
- Injecting Noise at the Output Targets-

  - Adding noise to the output targets during training makes the model less sensitive to small perturbations in the labels.
  - This technique is especially beneficial in situations where labels may be noisy or uncertain.

Parameter Tying and Parameter Sharing

Parameter Tying:

Definition: A mechanism where certain parameters in the model are forced to have the same value.

Purpose: Reduces the number of independent parameters, simplifying the model and improving efficiency.

Example: In Recurrent Neural Networks (RNNs), parameters for each time step are tied, ensuring consistency across the sequence.

Parameter Sharing:

Definition: A technique where multiple parts of the model reuse the same set of parameters.

Purpose: Reduces memory usage and allows the model to generalize better.

Example: In Convolutional Neural Networks (CNNs), the same filter (kernel) is applied across different parts of the image.

Key Benefit: Both techniques help reduce overfitting by limiting model capacity and ensure better generalization with fewer parameters.

Sparse Representations

Key Insight: Sparse representations help create models that are both efficient and effective in extracting essential patterns from data.

What is Sparse Representation?
A way to encode data where most of the elements are zero, focusing only on the most important features.

Advantages:
- Efficiency: Saves memory and computation time, as only significant features are stored or processed.
- Interpretability: Enhances understanding of the model's outputs by focusing on the most critical components.
- Biological Inspiration: Mimics how the human brain processes information, by activating only a small number of neurons at any time.

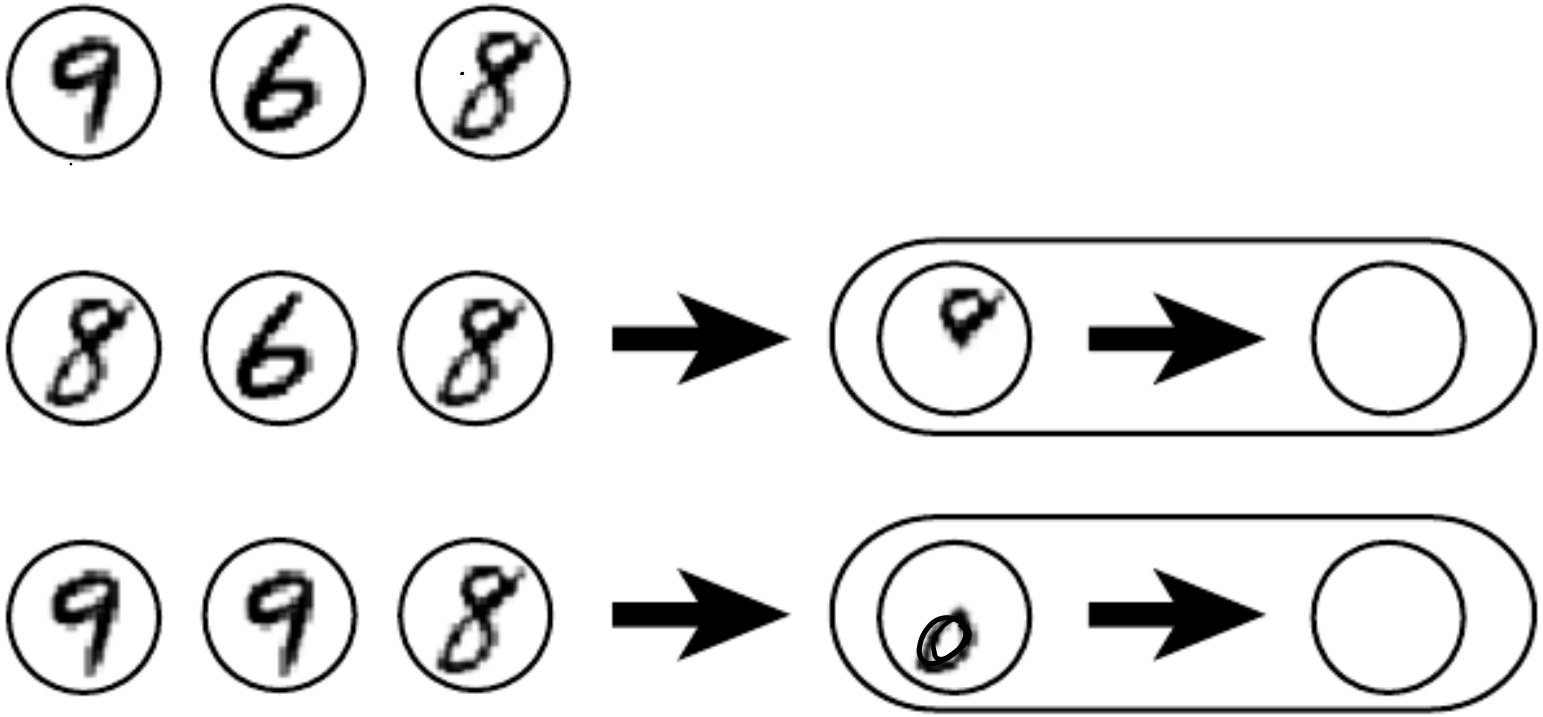
Applications:
- Used in autoencoders to learn compressed, sparse representations.
- Common in image processing and natural language processing (NLP) to extract meaningful patterns.

Bagging and Other Ensemble Methods

- Bagging (short for bootstrap aggregating) - technique for reducing generalization error by combining several models.
- The idea is to train several different models separately, then have all of the models vote on the output for test examples.
- Example of a general strategy in machine learning called model averaging.
- Techniques employing this strategy are known as ensemble methods
- Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times

Why it works: Combining different models helps to reduce variance, as each model might make different errors, and voting helps smooth out these errors.
Example: Random Forest is an ensemble method that heavily relies on Bagging.

Bagging and Other Ensemble Methods



Adversarial Training

- Training on adversarially disturbed examples from the training set
- Adversarial training helps to illustrate the power of using a large function family in combination with aggressive regularization.

What is Adversarial Training?
Definition: Adversarial training is a technique where models are exposed to intentionally perturbed (adversarial) examples during training. This improves their robustness to malicious attacks or noisy data.
Adversarial Examples: These are inputs modified with small, crafted changes that trick the model into making incorrect predictions. For Example: An image of a dog with imperceptible noise added might cause the model to misclassify it as a cat.
Benefits:
- Helps the model generalize better by learning to handle perturbed data.
- Leverages a large function family (capable of modeling complex relationships) with aggressive regularization to balance model complexity and robustness.
Use Cases: Enhancing the security of machine learning models in areas like fraud detection, cybersecurity, and autonomous systems.

Tangent Distance, Tangent Prop, and Manifold Tangent Classifier

Tangent Distance:
- Definition: A distance measure that accounts for transformations (e.g., rotations or translations) of input data, making it more robust to such variations.
- Example: In handwritten digit recognition, tangent distance ensures that small rotations or shifts in a digit (like '6') don't affect its classification.

Tangent Prop:
- Definition: A learning technique that penalizes the model for being sensitive to variations along a manifold (a lower-dimensional representation of data).
- Purpose: Encourages the network to focus on invariant features (e.g., recognizing the same object even when it appears at a slightly different angle).

Manifold Tangent Classifier:
- Definition: A classifier trained to model the local structure of data on a manifold, ensuring invariance to small perturbations.
- Benefit: This approach improves generalization by focusing on the intrinsic data structure.

Optimization for Training Deep Models

Challenges:

1. Ill-conditioning
2. Local Minima
3. Plateaus, Saddle Points and other Flat Regions
4. Cliffs and exploding gradients
5. Long-term Dependencies
6. Inexact Gradients

Challenges in Neural Network Optimization

1. Ill-Conditioning-

- Ill-conditioning can manifest by causing SGD to get "stuck" in the sense that even very small steps increase the cost function.
- From the literature if the below mentioned condition is True then we face the challenge of ill-conditioning in convex optimization.

$$(1/2) * learningrate^2 * g^T * H * g > learningrate * g^T * g$$

where g is first order gradient,

H is second order derivative or $Hessian$ matrix

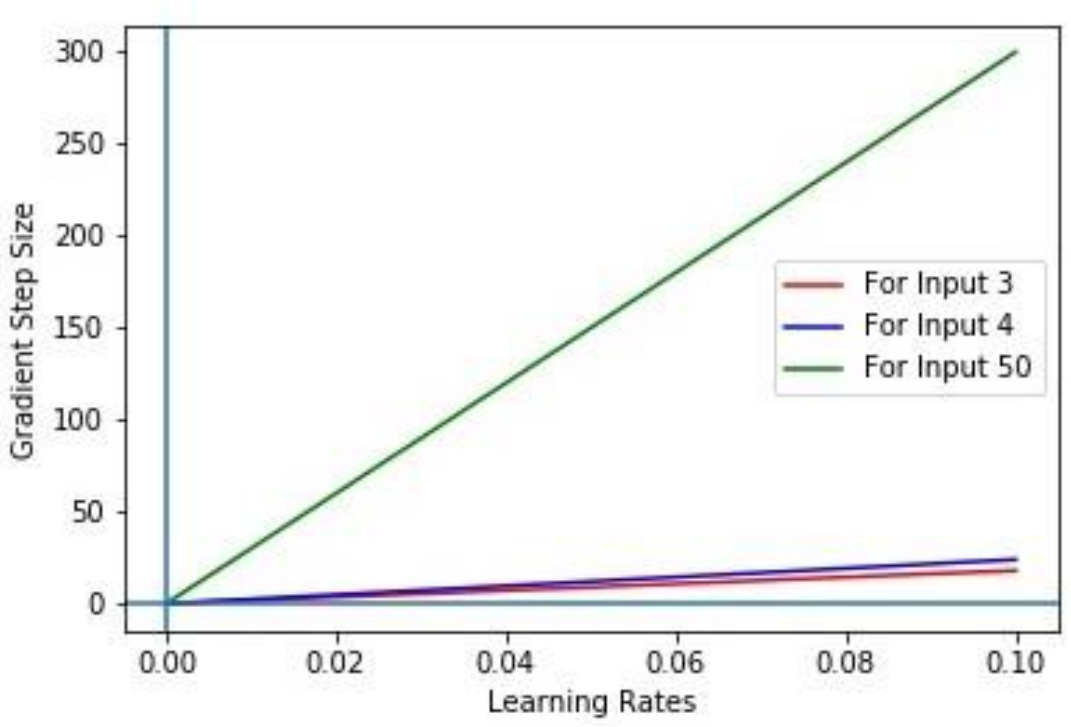
Challenges in Neural Network Optimization

1.Ill-Conditioning-

- Let us consider 3 inputs for x as 3.0, 4.0 and 50.0 and a weight function $f(x) = 30x^2$.
- From the function ,the first order derivative to be $f'(x) = 60x$ and second order derivative to be $f''(x) = 60$.

Challenges in Neural Network Optimization

1.Ill-Conditioning-

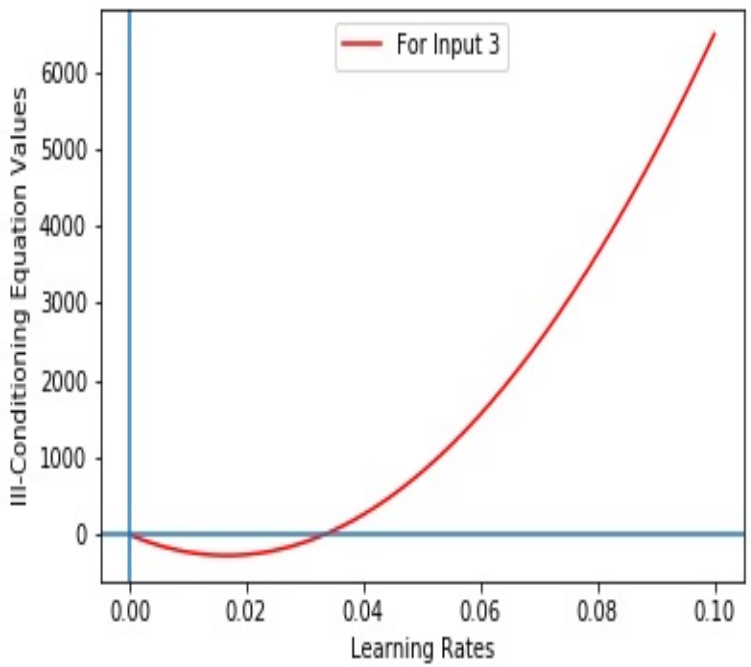


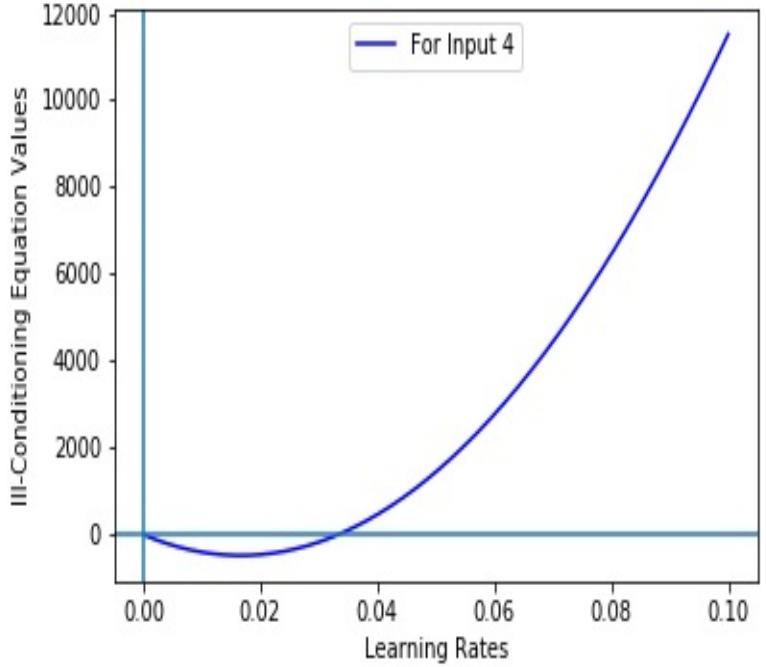
Weight Updates w.r.t Learning Rates

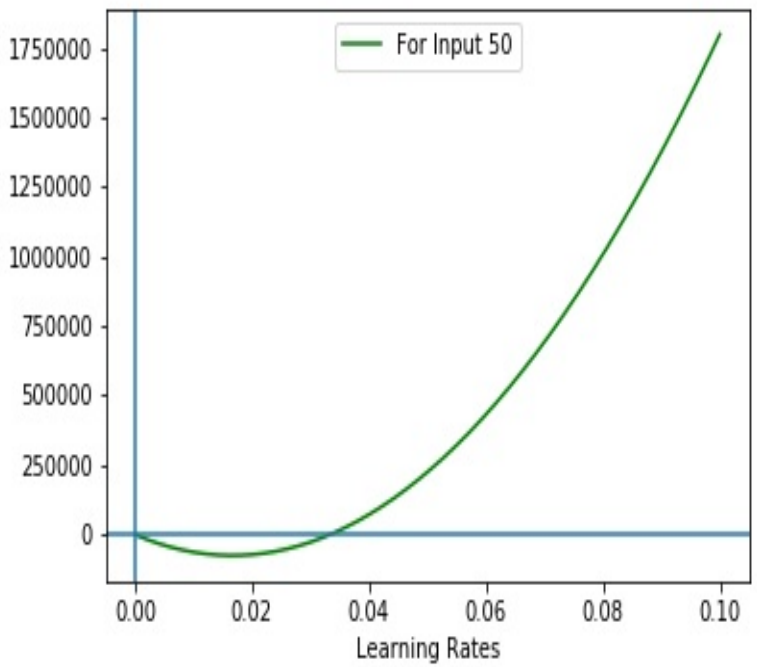
- Larger the size of the input the larger is the gradient step size.
- Find out the approximate learning rate which may lead to ill-conditioning.
- Obtain the values for the ill-conditioning equation and plot it.

Challenges in Neural Network Optimization

1.Ill-Conditioning-







*Ill-Conditioning Equation Values (**if positive it leads to ill-conditioning**)*

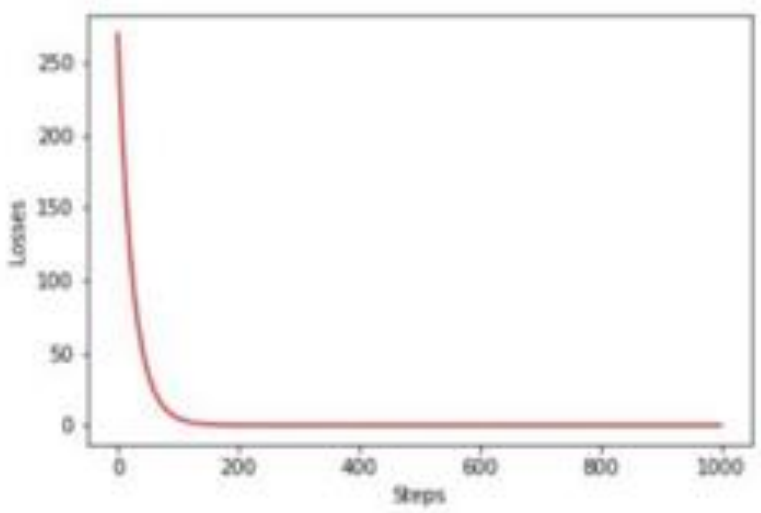
Challenges in Neural Network Optimization

Ill-Conditioning-

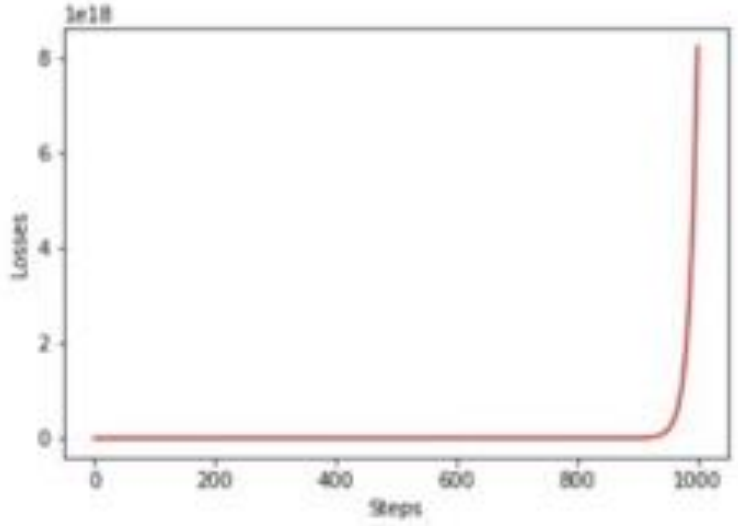
- Proof of Ill-Conditioning-
- To prove ill-conditioning take three scenarios.
  - No ill-conditioning learning rate
  - Learning rates on the border which may cause ill-conditioning if we let it run for a long time
  - Learning rate which will definitely cause ill-conditioning

Challenges in Neural Network Optimization

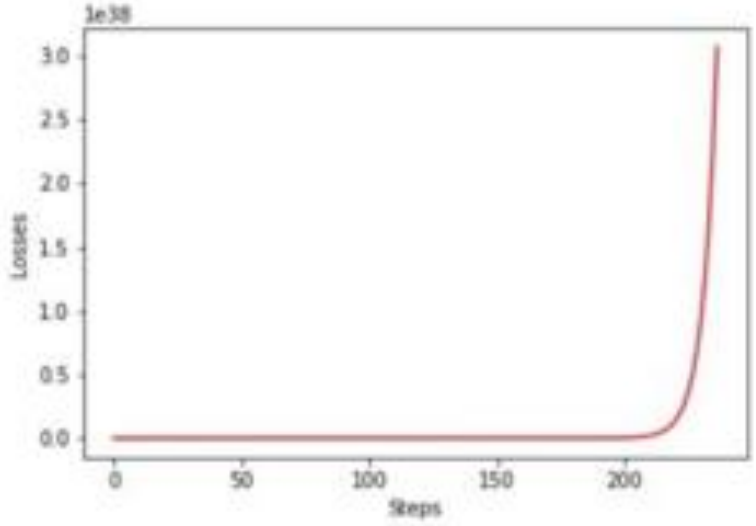
1. Ill-Conditioning-



For Input 3 and Learning Rate 0.033



For Input 3 and Learning Rate 0.034



For Input 3 and Learning Rate 0.035

Challenges in Neural Network Optimization

1. Ill-Conditioning-

Identifiers that there is Ill-conditioning-

- Loss values do not seem to decrease. This could mean that the loss function performs larger updates which leads the gradient to bounce around in the function.
- Sometimes NaN values can be observed in the loss values. One explanation for this is that the loss values could have exploded due to larger learning rates or larger weights. In addition to ill-conditioning, this can also be an indicator for exploding gradients problem.

Challenges in Neural Network Optimization

1. Ill-Conditioning-

How to solve Ill-conditioning

- **Using adaptive learning methods or SGD with annealing-**SGD with a constant learning rate may or may not cause ill-conditioning as the gradient approaches the local minima. Adaptive learning rates like Adam and Adadelta have been observed to perform better than SGD.
- **Pre-processing of the inputs**-Even something simple as pre-processing could be a solution to the ill-conditioning problem.
  - normalization

Challenges in Neural Network Optimization

2. Local Minima

- In convex optimization, problem is one of finding a local minimum
- Some convex functions have a flat region rather than a global minimum point
- Any point within the flat region is acceptable
- With non-convexity of neural nets many local minima are possible
- Many deep models are guaranteed to have an extremely large no. of local minima

Challenges in Neural Network Optimization

2. Local Minima-

- **Model Identifiability-**
- Model is identifiable if large training sample set can rule out all but one setting of parameters
  - Models with latent variables are not identifiable
  - Because we can exchange latent variables
    - If we have m layers with n units each there are $n!^m$ ways of arranging the hidden units
  - This non-identifiability is weight space symmetry
- Another is scaling incoming weights and biases
  - By a factor α and scale outgoing weights by $1/\alpha$
- Even if a neural net has uncountable no. of minima, they are equivalent in cost
  - So not a problematic form of non-convexity

Challenges in Neural Network Optimization

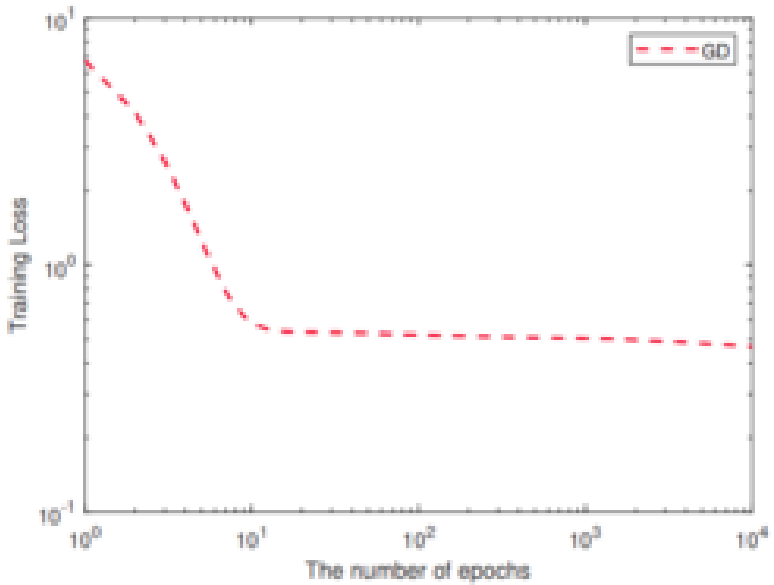
3. Plateaus, Saddle Points and Other Flat Regions-

More common than local minima/maxima are:

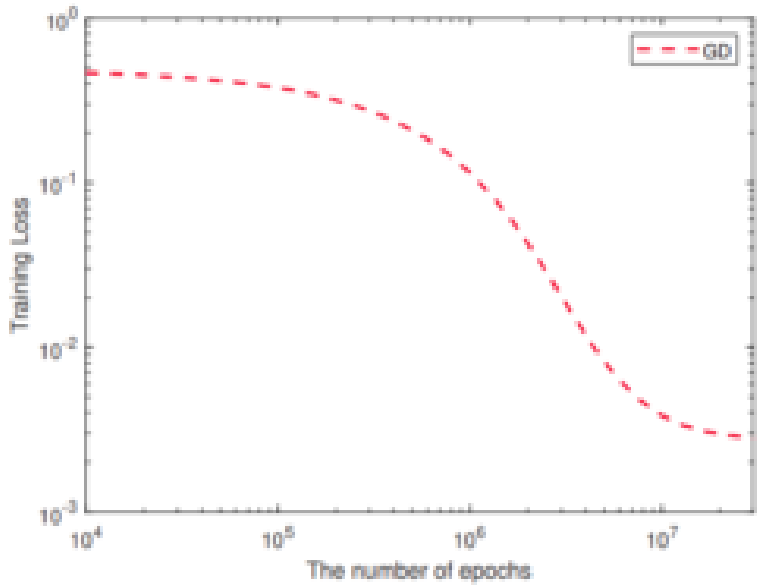
- Another kind of zero gradient points: saddle points
  - At saddle, Hessian has both positive and negative values
    - Positive: cost greater than saddle point
    - Negative values have lower value
  - In low dimensions:
    - Local minima are more common
  - In high dimensions:
    - Local minima are rare, saddle points more common

Challenges in Neural Network Optimization

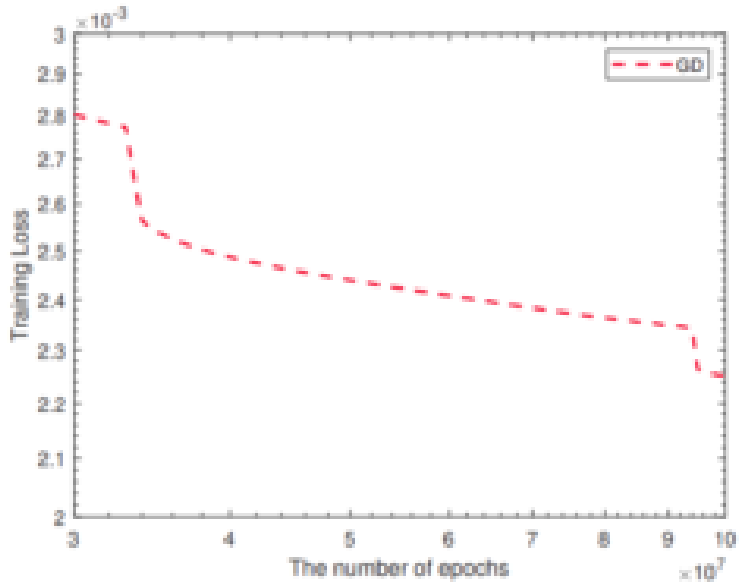
- Plateaus



(a)



(b)

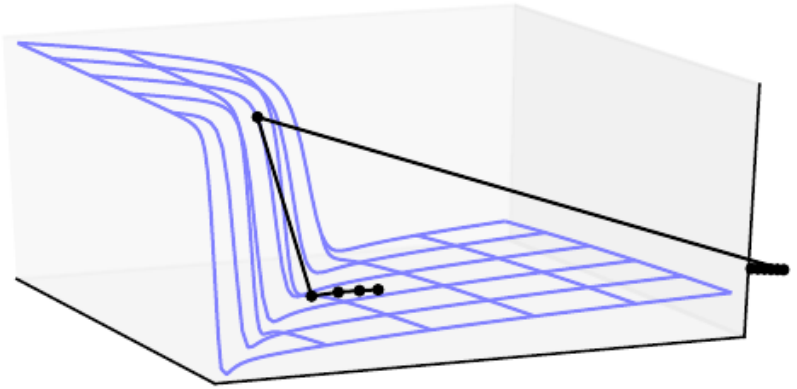


(c)

Challenges in Neural Network Optimization

4. Cliffs and Exploding Gradients-

- Neural networks with many layers often have extremely steep regions resembling cliffs
- These result from the multiplication of several large weights together.
- On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off of the cliff structure altogether.



Challenges in Neural Network Optimization

5. Long-Term Dependencies-

- Another difficulty that neural network optimization algorithms must overcome arises when the computational graph becomes extremely deep.
- Feedforward networks with many layers have such deep computational graphs.
- Recurrent networks, construct very deep computational graphs by repeatedly applying the same operation at each time step of a long temporal sequence

Challenges in Neural Network Optimization

6. Inexact Gradients-

- Most optimization algorithms are designed with the assumption that we have access to the exact gradient or Hessian matrix
- Use of surrogate loss function

Basic Algorithms

Basic Algorithms

- Stochastic Gradient Descent-

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
end while

Basic Algorithms

- Momentum-

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Basic Algorithms

- Nesterov Momentum-

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Parameter Initialization Strategies

Why Is Initialization Important?
In neural networks, the weights (parameters) are the backbone of the model.
Initializing these parameters effectively is critical for:
- Faster convergence during training.
- Avoiding issues like vanishing or exploding gradients.
- Ensuring the optimizer performs well.

Types of Parameter Initialization
Zero Initialization:
- All weights are set to zero.
- Issue: It causes all neurons to behave identically during training, leading to poor learning (symmetry problem).
Random Initialization:
- Parameters are given random small values to break symmetry.
- However, improper scaling can cause instability, such as gradients exploding or vanishing.
Xavier Initialization:
- Aims to keep the variance of the weights balanced across layers.
- Primarily designed for sigmoid or tanh activations.
- Prevents gradients from shrinking or growing excessively.

He Initialization:
- Specifically tailored for activation functions like ReLU.
- Provides slightly larger initial weights than Xavier Initialization to better handle deep networks.
Orthogonal Initialization:
- Ensures weights are initialized in an orthogonal fashion.
- Helps maintain stability in deep architectures by preserving variance through layers.

Algorithms with Adaptive Learning Rates

- Delta-bar-delta algorithm

1. AdaGrad-

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

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

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

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

end while

Algorithms with Adaptive Learning Rates

2. RMSProp-

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

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

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

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

end while

Algorithms with Adaptive Learning Rates

- 2. RMSProp-

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.

Initialize accumulation variable $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$

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

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

Algorithms with Adaptive Learning Rates

3. Adam

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Optimization Strategies and Meta-Algorithms

- Batch Normalization

What is Batch Normalization?
- A technique to improve training speed, stability, and performance in deep neural networks. It normalizes the input of each layer within a minibatch, ensuring that the mean is 0 and the variance is 1.

Benefits:
- Stabilizes Training: Reduces the risk of exploding/vanishing gradients.
- Faster Convergence: Allows the use of higher learning rates, leading to quicker optimization.
- Regularization: Acts as a slight form of regularization, reducing the need for dropout in some cases.

How It Helps:
- By ensuring a consistent distribution of inputs at each layer, Batch Normalization mitigates the internal covariate shift, leading to more efficient training.

Optimization Strategies and Meta-Algorithms

- Coordinate Descent

What is Coordinate Descent?
- An optimization algorithm that updates one parameter (or coordinate) at a time while keeping others fixed.

Advantages:
1. Efficient: Works well for problems with high-dimensional data or sparsity.
2. Simple to Implement: No need to compute the gradient over all variables simultaneously.

Applications:
- Used in regression tasks (e.g., Lasso, ElasticNet) and other models where partial optimization over one variable at a time is computationally efficient.

Limitation:
- May converge slowly for non-convex problems compared to methods like gradient descent.

Optimization Strategies and Meta-Algorithms

Gradient Clipping

What is Gradient Clipping?
- A technique to handle exploding gradients by setting an upper threshold for the gradient values during backpropagation.

Why It's Important:
- Exploding gradients can destabilize the learning process, especially in deep networks or recurrent architectures.

How It Works:
- If a computed gradient exceeds a pre-defined threshold, it's scaled down to fall within the range.

Benefits:
- Prevents the optimizer from making overly large updates.
- Stabilizes training in architectures like RNNs.

Optimization Strategies and Meta-Algorithms

Adaptive Learning Rates

What Are Adaptive Learning Rates?
- Techniques where the learning rate changes dynamically during training to suit the needs of the optimization process.

Popular Methods:
- Adagrad: Adjusts learning rates based on the frequency of parameter updates.
- RMSProp: Smoothens Adagrad by introducing a decay factor.
- Adam: Combines the benefits of momentum and adaptive learning.

Why It Matters:
- Ensures efficient convergence by reducing learning rates for parameters that are updated frequently while maintaining sufficient updates for others.

Optimization Strategies and Meta-Algorithms

Meta-Optimization Techniques

What is Meta-Optimization?
Refers to methods that optimize the optimization process itself (also called hyperparameter optimization).

Approaches:
- Grid Search: Tries all combinations of hyperparameters exhaustively.
- Random Search: Samples hyperparameters randomly, saving computation time.
- Bayesian Optimization: Uses probabilistic models to find the best hyperparameters efficiently.

Key Benefit:
- Improves the performance of machine learning models by systematically identifying optimal settings for learning rates, regularization strength, etc.

References

- Ian Goodfellow, Yoshua Bengio, Aaron Courville, “Deep Learning”, MIT Press 2017
- Josh Patterson and Adam Gibson, “Deep Learning A Practitioner’s Approach”, O’Reilly Media, 2017
- https://towardsdatascience.com/regularization-an-important-concept-in-machine-learning-5891628907ea
- https://www.analyticsvidhya.com/blog/2021/05/complete-guide-to-regularization-techniques-in-machine-learning/
- https://www.geeksforgeeks.org/regularization-in-machine-learning/