

## Unit 3

**Regularization for Deep Learning:** Parameter Norm Penalties, Norm Penalties as Constrained Optimization, Regularization and Under-Constrained Problems, Dataset Augmentation, Noise Robustness, Semi-Supervised Learning, Multi-Task Learning, Early Stopping, Parameter Tying and Parameter Sharing, Sparse Representations, Bagging and Other Ensemble Methods, Dropout, Adversarial Training, Tangent Distance, Tangent Prop and Manifold Tangent Classifier.

**Optimization for Training Deep Models:** Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, Approximate Second-Order Methods, Optimization Strategies and Meta-Algorithms.

<https://studyglance.in/dl/display.php?tno=12&topic=Early-stopping>

### What is Regularization?

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs

Before we deep dive into the topic, take a look at this image:

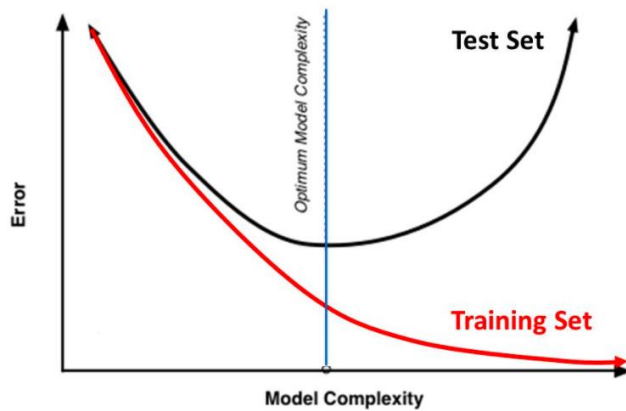


Have you seen this image before? As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, which ultimately results in poor performance on the unseen data.

In other words, while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't. This is shown in the image below.

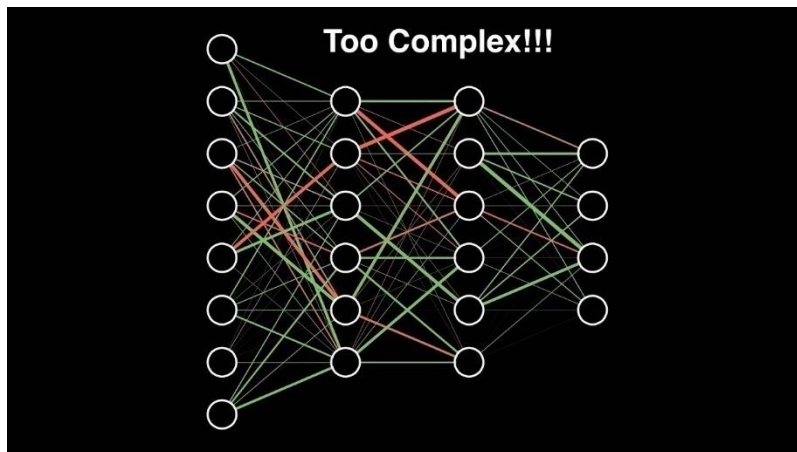
## Unit 3

### Training Vs. Test Set Error



*Source: Slideplayer*

If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting.



Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.

## Unit 3

### 1. Parameter Norm Penalties

Parameter Norm Penalties are regularization methods that apply a penalty to the norm of parameters in the objective function of a neural network.

#### Parameter Norm Penalty

- Limits the model's capacity by adding norm penalty  $\Omega(\theta)$  parameter to objective function  $J$ .

$$\text{Regularized objective function} \quad \tilde{J}(\theta; X, y) = \text{Original objective function} \quad J(\theta; X, y) + \text{Penalty term} \quad \alpha\Omega(\theta)$$

- Does not modify the model in inference phase, but adds penalties in learning phase.
- Norm penalty penalizes only weights, leaving biases unregularized.
- Also known as **Weight Decay**.

Methods: i. Weight decay ii. L1 Regularization

### Weight Decay

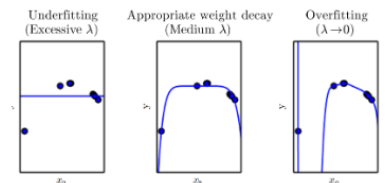
 Edit

**Weight Decay**, or  $L_2$  **Regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function comprising both the primary loss function and a penalty on the  $L_2$  Norm of the weights:

$$L_{new}(w) = L_{original}(w) + \lambda w^T w$$

where  $\lambda$  is a value determining the strength of the penalty (encouraging smaller weights).

Weight decay can be incorporated directly into the weight update rule, rather than just implicitly by defining it through to objective function. Often weight decay refers to the implementation where we specify it directly in the weight update rule (whereas  $L_2$  regularization is usually the implementation which is specified in the objective function).



i.

### L1 Regularization

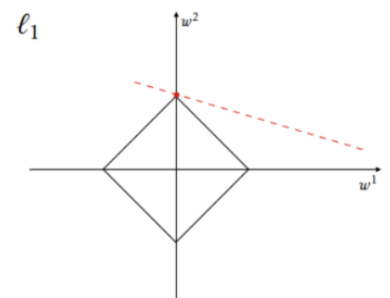
 Edit

$L_1$  **Regularization** is a regularization technique applied to the weights of a neural network. We minimize a loss function comprising both the primary loss function and a penalty on the  $L_1$  Norm of the weights:

$$L_{new}(w) = L_{original}(w) + \lambda ||w||_1$$

where  $\lambda$  is a value determining the strength of the penalty. In contrast to **weight decay**,  $L_1$  regularization promotes sparsity; i.e. some parameters have an optimal value of zero.

Image Source: [Wikipedia](#)



ii.

## Unit 3

The idea here is to limit the capacity (the space of all possible model families) of the model by adding a parameter norm penalty,  $\Omega(\theta)$ , to the objective function,  $J$ :

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

Here,  $\theta$  represents only the weights and not the biases, the reason being that the biases require much less data to fit and do not add much variance.

### 2. Norm penalties as constrained optimization

we can construct a generalized Lagrangian function containing the objective function along with the penalties. Suppose we wanted  $\Omega(\theta) < k$ , then we could construct the following Lagrangian:

$$\mathcal{L}(\theta, \alpha; X, y) = J(\theta; X, y) + \alpha(\Omega(\theta) - k).$$

We get optimal  $\theta$  by solving the Lagrangian. If  $\Omega(\theta) > k$ , then the weights need to be compensated highly and hence,  $\alpha$  should be large to reduce its value below  $k$ . Likewise, if  $\Omega(\theta) < k$ , then the norm shouldn't be reduced too much and hence,  $\alpha$  should be small. This is now similar to the parameter norm penalty regularized objective function as both of them encourage lower values of the norm. Thus, parameter norm penalties naturally impose a constraint, like the  $L^2$ -regularization, defining a constrained  $L^2$ -ball. Larger  $\alpha$  implies a smaller constrained region as it pushes the values really low, hence, allowing a small radius and vice versa. The idea of constraints over penalties is important for several reasons. Large penalties might cause non-convex optimization algorithms to get stuck in local minima due to small values of  $\theta$ , leading to the formation of so-called *dead cells*, as the weights entering and leaving them are too small to have an impact.

## Unit 3

Constraints don't enforce the weights to be near zero, rather being confined to a constrained region.

Another reason is that constraints induce higher stability. With higher learning rates, there might be a large weight, leading to a large gradient, which could go on iteratively leading to numerical overflow in the value of  $\theta$ . Constraints, along with reprojection (to the corresponding ball), prevent the weights from becoming too large, thus, maintaining stability.

A final suggestion made by Hinton was to restrict the individual column norms of the weight matrix rather than the Frobenius norm of the entire weight matrix, so as to prevent any hidden unit from having a large weight. The idea here is that if we restrict the Frobenius norm, it doesn't guarantee that the individual weights would be small, just their norm. So, we might have large weights being compensated by extremely small weights to make the overall norm small. Restricting each hidden unit individually gives us the required guarantee.

### 3. Regularized & Under-constrained problems

*Underdetermined problems* are those problems that have infinitely many solutions. A logistic regression problem having linearly separable classes with  $w$  as a solution, will always have  $2w$  as a solution and so on. In some machine learning problems, regularization is necessary. For e.g., many algorithms (e.g. PCA) require the inversion of  $X'X$ , which might be singular. In such a case, we can use a regularized form instead.  $(X'X + \alpha I)$  is guaranteed to be invertible.

## Unit 3

Regularization can solve underdetermined problems. For e.g. the Moore-Pentrose pseudoinverse defined earlier as:

$$X^+ = \lim_{\alpha \rightarrow 0} (X^T X + \alpha I)^{-1} X^T$$

This can be seen as performing a linear regression with L<sup>2</sup>-regularization.

### 4. Data augmentation

The best way to make a machine learning model generalize better is to train it on more data. Data augmentation is a way of creating fake data and adding it to training set.

Data augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data. It includes making minor changes to the dataset or using deep learning to generate new data points.

Augmented vs. Synthetic data









Augmented data is driven from original data with some minor changes. In the case of image augmentation, we make geometric and color space transformations (flipping, resizing, cropping, brightness, contrast) to increase the size and diversity of the training set.

Synthetic data is generated artificially without using the original dataset. It often uses DNNs (Deep Neural Networks) and GANs (Generative Adversarial Networks) to generate synthetic data.

Note: the augmentation techniques are not limited to images. You can augment audio, video, text, and other types of data too.

Data set augmentation very effective for the classification problem of object recognition. Images are high-dimensional and include a variety of variations, may easily simulated. translating the training images a few pixels in each direction can greatly improve performance. Many other operations such as rotating the image or scaling the image have also proven quite effective.

## Unit 3

Original	Flip	Rotation	Random crop
			
<ul style="list-style-type: none"> <li>Image without any modification</li> </ul>	<ul style="list-style-type: none"> <li>Flipped with respect to an axis for which the meaning of the image is preserved</li> </ul>	<ul style="list-style-type: none"> <li>Rotation with a slight angle</li> <li>Simulates incorrect horizon calibration</li> </ul>	<ul style="list-style-type: none"> <li>Random focus on one part of the image</li> <li>Several random crops can be done in a row</li> </ul>
Color shift	Noise addition	Information loss	Contrast change
			
<ul style="list-style-type: none"> <li>Nuances of RGB is slightly changed</li> <li>Captures noise that can occur with light exposure</li> </ul>	<ul style="list-style-type: none"> <li>Addition of noise</li> <li>More tolerance to quality variation of inputs</li> </ul>	<ul style="list-style-type: none"> <li>Parts of image ignored</li> <li>Mimics potential loss of parts of image</li> </ul>	<ul style="list-style-type: none"> <li>Luminosity changes</li> <li>Controls difference in exposition due to time of day</li> </ul>

One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between ‘b’ and ‘d’ and the difference between ‘6’ and ‘9’, so horizontal flips and 180° rotations are not appropriate ways of augmenting datasets for these tasks.

Injecting noise into the input of a neural network can be seen as data augmentation. For some regression tasks it is still possible to solve even if small random noise is added to the input. Neural networks are not robust to noise. To improve robustness, train them with random noise applied to their inputs. Input noise injection is part of some unsupervised learning algorithms such as the denoising autoencoder. Noise can also be applied to hidden units, which can be seen as doing dataset augmentation at multiple levels of abstraction.

When Should You Use Data Augmentation?

- To prevent models from overfitting.

## Unit 3

- The initial training set is too small.
- To improve the model accuracy.
- To Reduce the operational cost of labeling and cleaning the raw dataset.

### Limitations of Data Augmentation

- The biases in the original dataset persist in the augmented data.
- Quality assurance for data augmentation is expensive.
- Research and development are required to build a system with advanced applications. For example, generating high-resolution images using GANs can be challenging.
- Finding an effective data augmentation approach can be challenging.

### Data Augmentation Techniques

In this section, we will learn about audio, text, image, and advanced data augmentation techniques.

#### Audio Data Augmentation

1. **Noise injection:** add gaussian or random noise to the audio dataset to improve the model performance.
2. **Shifting:** shift audio left (fast forward) or right with random seconds.
3. **Changing the speed:** stretches times series by a fixed rate.
4. **Changing the pitch:** randomly change the pitch of the audio.

#### Text Data Augmentation

1. **Word or sentence shuffling:** randomly changing the position of a word or sentence.
2. **Word replacement:** replace words with synonyms.
3. **Syntax-tree manipulation:** paraphrase the sentence using the same word.
4. **Random word insertion:** inserts words at random.
5. **Random word deletion:** deletes words at random.

#### Image Augmentation

1. **Geometric transformations:** randomly flip, crop, rotate, stretch, and zoom images. You need to be careful about applying multiple transformations on the same images, as this can reduce model performance.



## Unit 3

2. **Color space transformations:** randomly change RGB color channels, contrast, and brightness.
3. **Kernel filters:** randomly change the sharpness or blurring of the image.
4. **Random erasing:** delete some part of the initial image.
5. **Mixing images:** blending and mixing multiple images.

### Advanced Techniques

1. **Generative adversarial networks (GANs):** used to generate new data points or images. It does not require existing data to generate synthetic data.
2. **Neural Style Transfer:** a series of convolutional layers trained to deconstruct images and separate context and style.

### Data Augmentation Applications

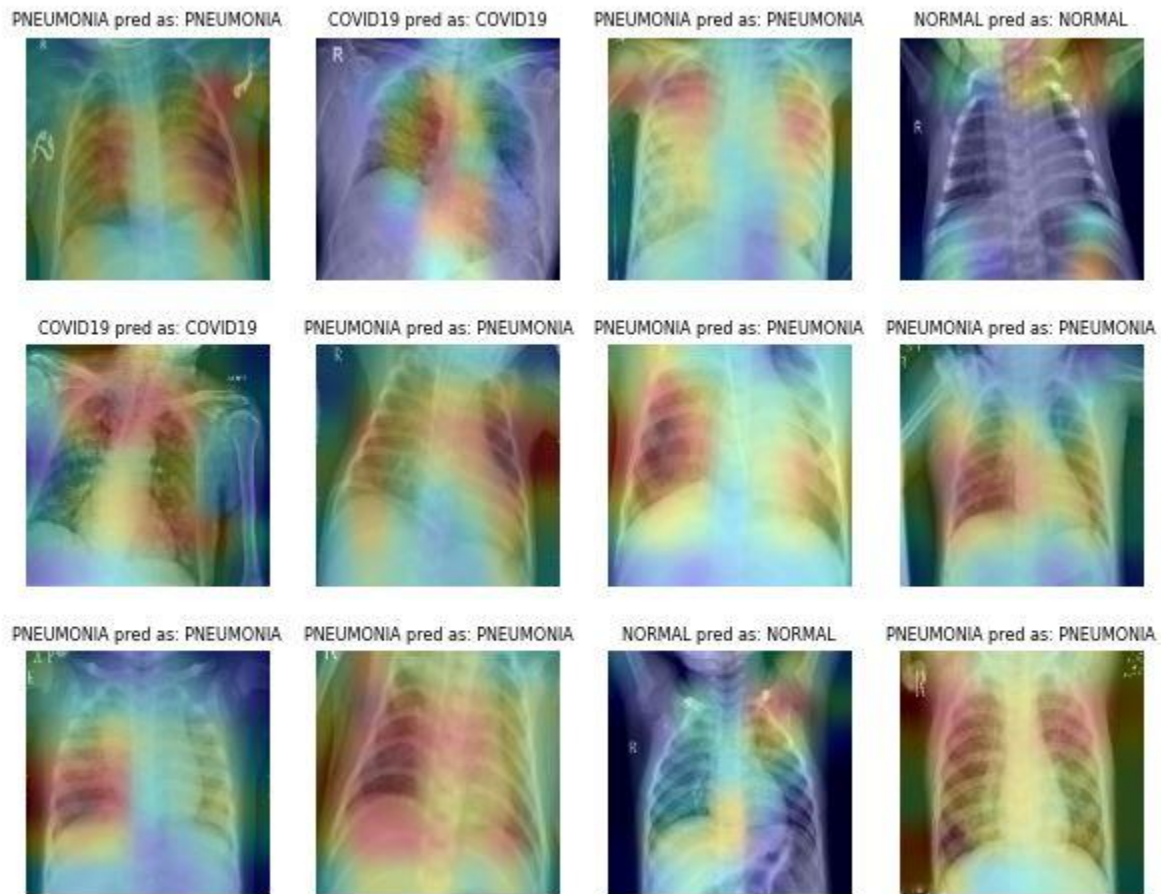
Data augmentation can apply to all machine learning applications where acquiring quality data is challenging. Furthermore, it can help improve model robustness and performance across all fields of study.

### Healthcare

Acquiring and labeling medical imaging datasets is time-consuming and expensive. You also need a subject matter expert to validate the dataset before performing data analysis. Using geometric and other transformations can help you train robust and accurate machine-learning models.

For example, in the case of Pneumonia Classification, you can use random cropping, zooming, stretching, and color space transformation to improve the model performance. However, you need to be careful about certain augmentations as they can result in opposite results. For example, random rotation and reflection along the x-axis are not recommended for the X-ray imaging dataset.

## Unit 3



### Self-Driving Cars

There is limited data available on self-driving cars, and companies are using simulated environments to generate synthetic data using reinforcement learning. It can help you train and test machine learning applications where data security is an issue.

## Unit 3



### Natural Language Processing

Text data augmentation is generally used in situations with limited quality data, and improving the performance metric takes priority. You can apply synonym augmentation, word embedding, character swap, and random insertion and deletion. These techniques are also valuable for low-resource languages.

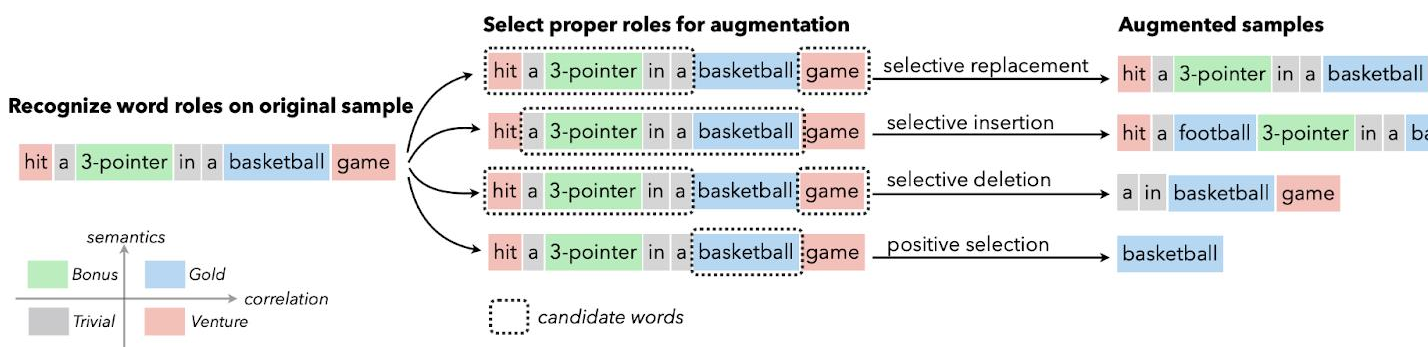


Image from [Papers With Code](#) | Selective Text Augmentation with Word Roles for Low-Resource Text Classification.

Researchers use text augmentation for the language models in high error recognition scenarios, sequence-to-sequence data generation, and text classification.

### Automatic Speech Recognition

In sound classification and speech recognition, data augmentation works wonders. It improves the model performance even on low-resource languages.

### Unit 3

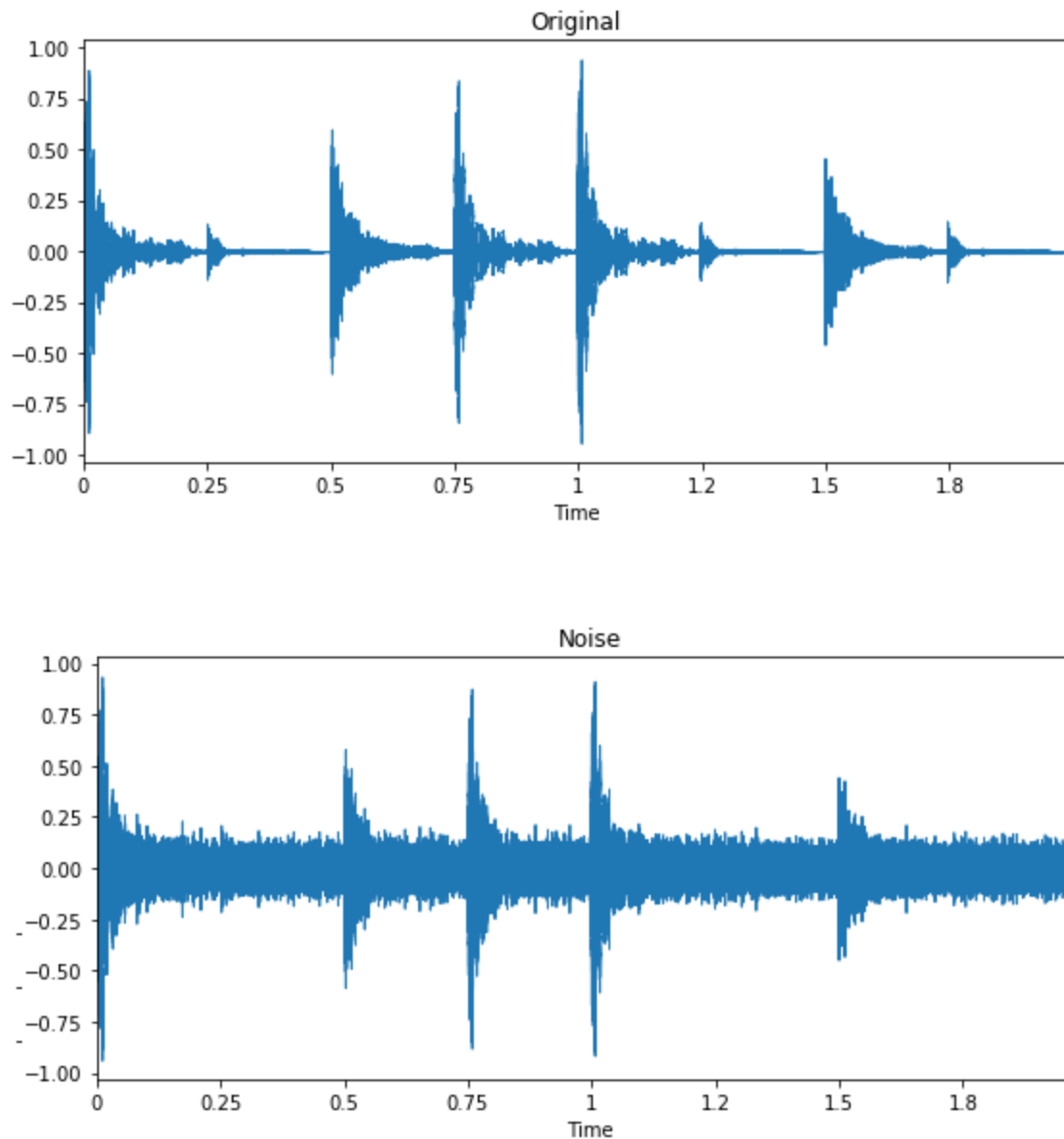


Image by [Edward Ma](#) | Noise Injection

The random noise injection, shifting, and changing the pitch can help you produce state-of-the-art speech-to-text models. You can also use GANs to generate realistic sounds for a particular application.

Having more data is the most desirable thing to improving a machine learning model's performance. In many cases, it is relatively easy to artificially generate data. For a classification

## Unit 3

task, we desire for the model to be invariant to certain types of transformations, and we can generate the corresponding  $(x,y)$  pairs by translating the input  $x$ . But for certain problems, like density estimation, we can't apply this directly unless we have already solved the density estimation problem.

However, caution needs to be maintained while augmenting data to make sure that the class doesn't change. For e.g., if the labels contain both "b" and "d", then horizontal flipping would be a bad idea for data augmentation. Adding random noise to the inputs is another form of data augmentation, while adding noise to hidden units can be seen as doing data augmentation at multiple levels of abstraction.

Finally, when comparing machine learning models, we need to evaluate them using the same hand-designed data augmentation schemes or else it might happen that algorithm A outperforms algorithm B, just because it was trained on a dataset which had more / better data augmentation.

### **5. Noise Robustness**

Noise applied to inputs is a data augmentation, For some models addition of noise with extremely small variance at the input is equivalent to imposing a penalty on the norm of the weights.

Noise applied to hidden units, Noise injection can be much more powerful than simply shrinking the parameters. Noise applied to hidden units is so important that Dropout is the main development of this approach.

Adding Noise to Weights, This technique primarily used with Recurrent Neural Networks(RNNs). This can be interpreted as a stochastic implementation of Bayesian inference over the weights. Bayesian learning considers model weights to be uncertain and representable via a probability distribution  $p(w)$  that reflects that uncertainty. Adding noise to weights is a practical, stochastic way to reflect this uncertainty.

## Unit 3

Noise applied to weights is equivalent to traditional regularization, encouraging stability. This can be seen in a regression setting, Train  $\hat{y}(x)$  to map  $x$  to a scalar using least squares between model prediction  $\hat{y}(x)$  and true values  $y$ .

$$J = \mathbb{E}_{p(x,y)} \left[ (\hat{y}(x) - y)^2 \right]$$

The training set consists of  $m$  labeled examples  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ . We perturb each input with  $\epsilon_W \sim N(\epsilon; 0, \eta I)$ . For small  $\eta$ , this is equivalent to a regularization term  $\eta \mathbb{E}_{p(x,y)} \left[ \|\nabla_w \hat{y}(x)\|^2 \right]$ . It encourages parameters to regions where small perturbations of weights have small influence on output.

Injecting Noise at the Output Targets, Most datasets have some amount of mistakes in the  $y$  labels. It can be harmful to maximize  $\log p(y | x)$  when  $y$  is a mistake. Most datasets have some amount of mistakes in the  $y$  labels. It can be harmful to maximize  $\log p(y | x)$  when  $y$  is a mistake. This can be incorporated into the cost function, Ex: Local Smoothing regularizes a model based on a softmax with  $k$  output values by replacing the hard 0 and 1 classification targets with targets of  $\epsilon/(k-1)$  and  $1-\epsilon$  respectively.

### 6. Semi-Supervised Learning

In the paradigm of semi-supervised learning, both unlabeled examples from  $P(x)$  and labeled examples from  $P(x, y)$  are used to estimate  $P(y | x)$  or predict  $y$  from  $x$ .

The most basic disadvantage of any Supervised Learning algorithm is that the dataset has to be hand-labeled either by a Machine Learning Engineer or a Data Scientist. This is a very costly process, especially when dealing with large volumes of data. The most basic disadvantage of any Unsupervised Learning is that its application spectrum is limited.

To counter these disadvantages, the concept of Semi-Supervised Learning was introduced. In this type of learning, the algorithm is trained upon a combination of labeled and unlabeled data. Typically, this combination will contain a very small amount of labeled data and a very large amount of unlabeled data.

### Working of Semi-Supervised Learning

- Firstly, it trains the model with less amount of training data similar to the supervised learning models. The training continues until the model gives accurate results.
- The algorithms use the unlabeled dataset with pseudo labels in the next step, and now the result may not be accurate.
- Now, the labels from labeled training data and pseudo labels data are linked together.
- The input data in labeled training data and unlabeled training data are also linked.
- In the end, again train the model with the new combined input as did in the first step. It will reduce errors and improve the accuracy of the model.

Sharing Parameters, Instead of separate unsupervised and supervised components in the model, construct models in which generative models of

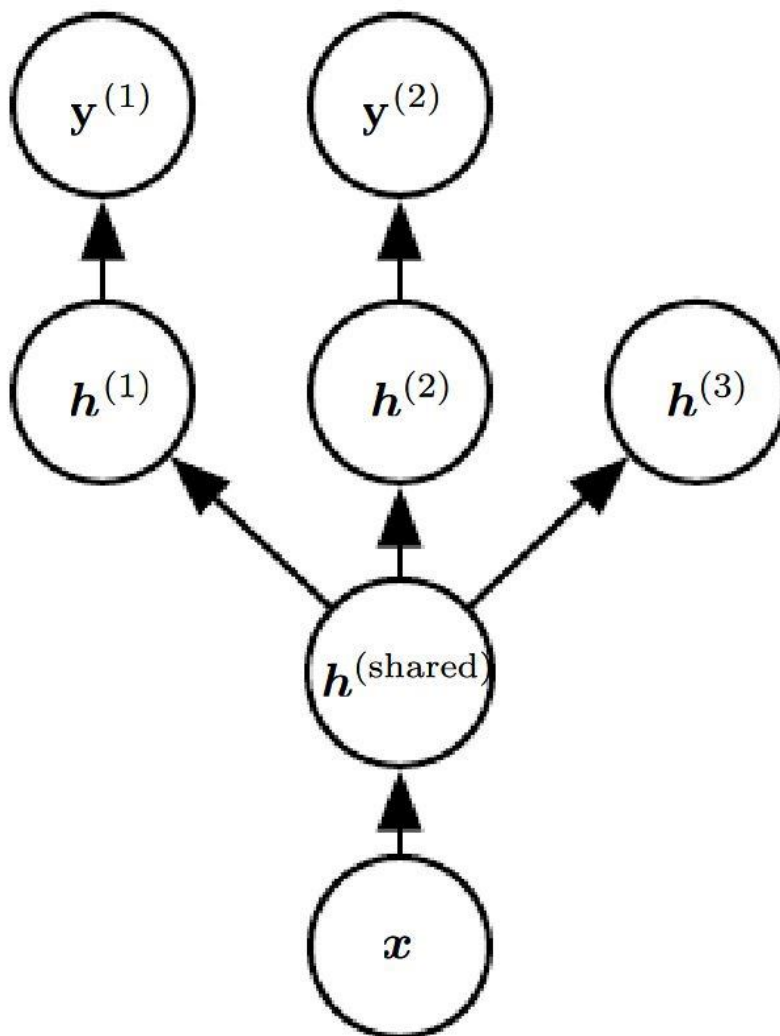
### Unit 3

either  $P(x)$  or  $P(x,y)$  shares parameters with a discriminative model of  $P(y|x)$ . One can then trade-off the supervised criterion  $-\log P(y|x)$  with the unsupervised or generative one (such as  $-\log P(x)$  or  $-\log P(x,y)$ )

#### 7. Multitask Learning

Multi-Task Learning is a sub-field of Deep Learning that aims to solve multiple different tasks at the same time, by taking advantage of the similarities between different tasks. This can improve the learning efficiency and also act as a regularizer which we will discuss in a while.

Formally, if there are  $n$  tasks (conventional deep learning approaches aim to solve just 1 task using 1 particular model), where these  $n$  tasks or a subset of them are related to each other but not exactly identical, Multi-Task Learning (MTL) will help in improving the learning of a particular model by using the knowledge contained in all the  $n$  tasks.





## Unit 3

The different supervised tasks (predicting  $y^{(i)}$  given  $x$ ) share the same input  $x$ , as well as some intermediate-level representation  $h^{(\text{shared})}$  capturing a common pool of factors (Common input but different target random variables). Task-specific parameters  $h^{(1)}$ ,  $h^{(2)}$  can be learned on top of those yielding a shared representation  $h^{(\text{shared})}$ . Common pool of factors explain variations of Input  $x$  while each task is associated with a Subset of these factors.

In the unsupervised learning context, some of the top level factors are associated with none of the output tasks  $h^{(3)}$ . These are factors that explain some of the input variations but not relevant for predicting  $h^{(1)}$ ,  $h^{(2)}$

The model can generally be divided into two kinds of parts and associated parameters:

1. 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.
2. 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.

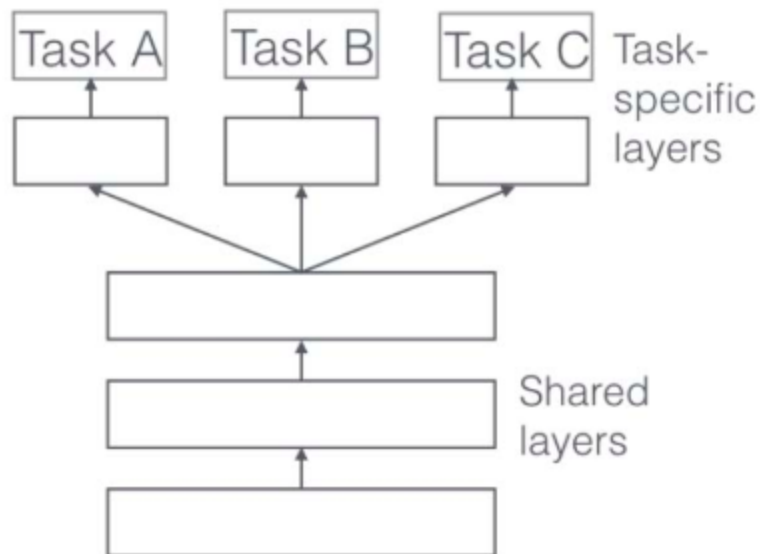
**Benefits of multi-tasking** Improved generalization and generalization error bounds. Achieved due to shared parameters for which statistical strength can be greatly improved in proportion to the increased no.of examples for the shared parameters compared to the scenario of single-task models. From the point of view of deep learning, the underlying prior belief is the following: Among the factors that explain the variations observed in the data associated with different tasks, some are shared across two or more tasks

Multitask learning leads to better generalization when there is actually some relationship between the tasks, which actually happens in the context of Deep Learning where some of the factors, which explain the variation observed in the data, are shared across different tasks.

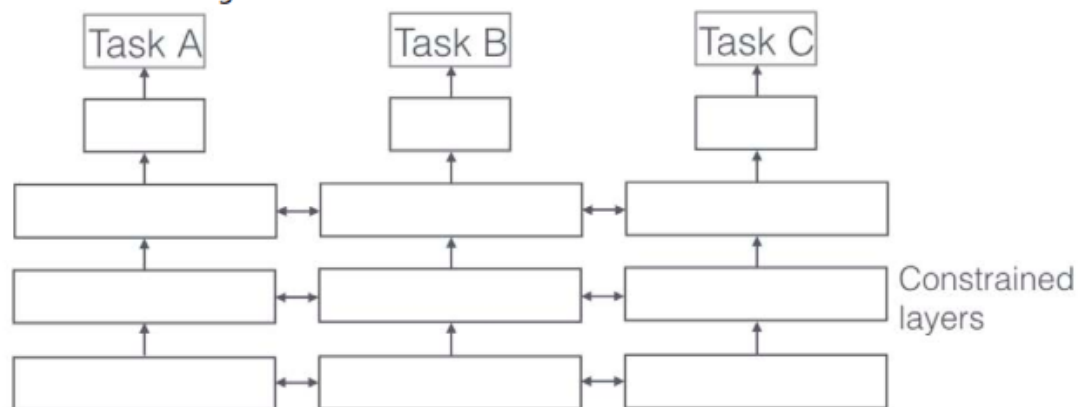


## Types of Multi-task Learning [↗](#)

- **Hard-parameter sharing :**
  - Greatly reduces risk of **over-fitting**
  - Similar concept of **bagging**.



- **Soft paramtere sharing :**
  - Take the role of **regularization**.

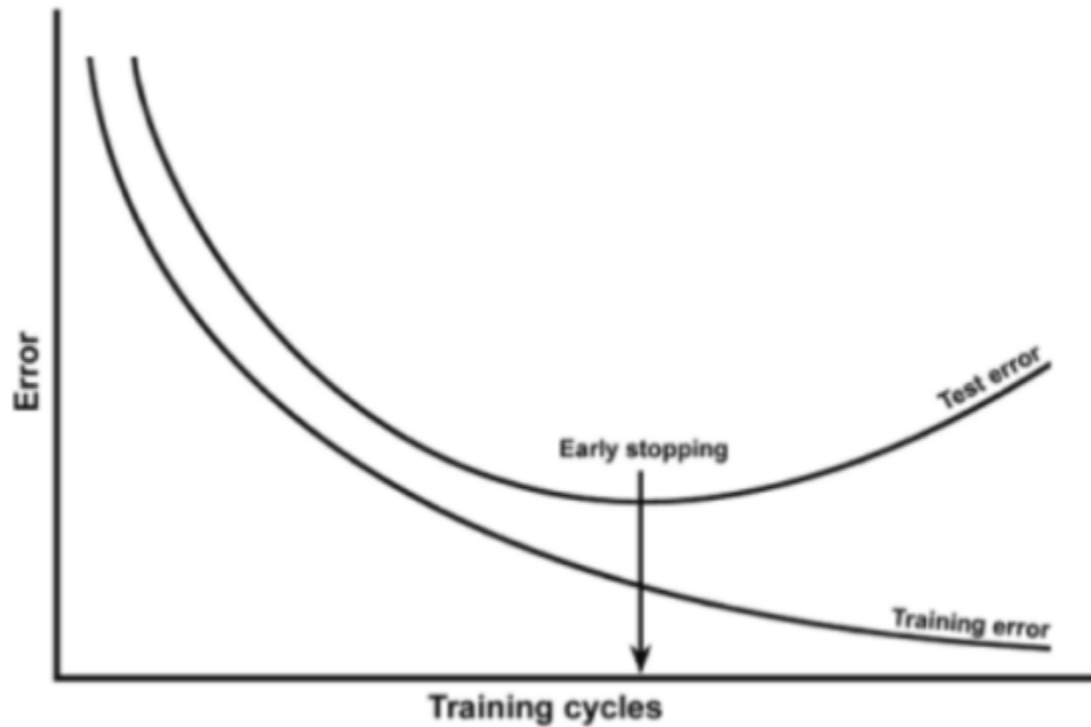


### 8. Early Stopping

- **Motivation :** When training large models with sufficient **representational capacity**, with time, **training error decreases** but **validation set error begins to rise again**.

## Unit 3

- Therefore, instead of returning latest parameters, we keep a **copy of model parameters** every time error on validation set improves (**model hits the lowest validation set error**).
- Algorithm **terminates when no parameters have improved** over the best recorded validation error **for some pre-specified number of iterations**. This is called **Early Stopping**. (effective hyper-parameter selection)
- Controls **effective capacity** of model.
- Excessive training **can cause over-fitting**.



### Advantages of Early Stopping

- Early stopping requires **no change in training procedure/objective function/set of allowable parameter values** (the learning dynamics).
- Early stopping can be used **alone or in conjunction** with other regularization strategies.
- Early stopping requires **validation data set** (extra data not included with training data). Exploiting this data requires **extra training** after initial training. Following are 2 strategies used for 2nd training -
  - Initialize model again and train all data. For 2nd training round, train data for same #steps as early-stopping predicted.
    - No good way of knowing whether to train for **same #parameter updates or same #passes** through dataset.

## Unit 3

- Keep parameters obtained from 1st training round and then continue training using all data.
  - Monitor **average loss function on validation set** and continue training till it **falls below** the value of training set **objective at which early stopping procedure halted**.
  - **Prevents high cost** of re-training model from scratch.
  - May **not ever terminate**, if objective on validation set never reaches the target value.

### Disadvantages of Early Stopping

- Expensive **cost of selecting effective hyperparameter**.
- Additional **cost to maintain copy** of model parameters.

## 9. Parameter Tying and Parameter Sharing

Till now, most of the methods focused on bringing the weights to a fixed point, e.g. 0 in the case of norm penalty. However, there might be situations where we might have some prior knowledge on the kind of dependencies that the model should encode. Suppose, two models A and B, perform a classification task on similar input and output distributions. In such a case, we'd expect the parameters for both the models to be similar to each other as well. We could impose a norm penalty on the distance between the weights, but a more popular method is to *force* the set of parameters to be equal. This is the essence behind *Parameter Sharing*. A major benefit here is that we need to store only a subset of the parameters (e.g. storing only the parameters for model A instead of storing for both A and B) which leads to large memory savings. In the example of Convolutional Neural Networks or CNNs, the same feature is computed across different regions of the image and hence, a cat is detected irrespective of whether it is at position  $i$  or  $i+1$ .

## 10. Sparse Representations

Sparse representation is a parsimonious principle that a signal can be approximated by a sparse superposition of basis functions.

## Unit 3

We can place penalties on even the activation values of the units which indirectly imposes a penalty on the parameters. This leads to representational sparsity, where many of the activation values of the units are zero. In the figure below,  $h$  is a representation of  $x$ , which is sparse. Representational sparsity is obtained similarly to the way parameter sparsity is obtained, by placing a penalty on the representation  $h$  instead of the weights.

Parameter sparsity caused by something like L1 reg.

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}_{y \in \mathbb{R}^m} = \begin{bmatrix} 1 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}_{A \in \mathbb{R}^{m \times n}} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}_{x \in \mathbb{R}^n}$$

Representational Sparsity

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}_{y \in \mathbb{R}^m} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}_{B \in \mathbb{R}^{m \times n}} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}_{h \in \mathbb{R}^n}$$

Another idea could be to average the activation values across various examples and push it towards some value. An example of getting representational sparsity by imposing hard constraint on the activation value is the **Orthogonal Matching Pursuit (OMP)** algorithm, where a representation  $h$  is learned for the input  $x$  by solving the constrained optimization problem:

$$\operatorname{argmin}_{h, \|h\|_b < k} \|x - Wh\|^2$$

where the constraint is on the the number of non-zero entries indicated by  $b$ . The problem can be solved efficiently when  $W$  is restricted to be orthogonal

## Unit 3

### 11. Bagging and Other Ensemble Methods

The techniques which train multiple models and take the maximum vote across those models for the final prediction are called *ensemble* methods. The idea is that it's highly unlikely that multiple models would make the same error in the test set.

Suppose that we have  $K$  regression models, with the model  $\#i$  making an error  $\epsilon_i$  on each example, where  $\epsilon_i$  is drawn from a zero mean, multivariate normal distribution such that:  $\mathbb{E}(\epsilon_i^2)=v$  and  $\mathbb{E}(\epsilon_i\epsilon_j)=c$ . The error on each example is then the average across all the models:  $(\sum \epsilon_i)/K$ .

The mean of this average error is 0 (as the mean of each of the individual  $\epsilon_i$  is 0). The variance of the average error is given by:

$$\begin{aligned}\mathbb{E}\left(\frac{\sum_i \epsilon_i}{K}\right)^2 &= \frac{\mathbb{E}(\sum_i \epsilon_i^2 + \sum_i \sum_{j \neq i} \epsilon_i \epsilon_j)}{K^2} \\ \Rightarrow \mathbb{E}\left(\frac{\sum_i \epsilon_i}{K}\right)^2 &= \frac{\mathbb{E} \sum_i \epsilon_i^2}{K^2} + \frac{\sum_i \sum_{j \neq i} \mathbb{E}(\epsilon_i \epsilon_j)}{K^2} \\ \Rightarrow \mathbb{E}\left(\frac{\sum_i \epsilon_i}{K}\right)^2 &= \frac{K * v}{K^2} + \frac{K * (K - 1)c}{K^2} \\ \Rightarrow \mathbb{E}\left(\frac{\sum_i \epsilon_i}{K}\right)^2 &= \frac{v}{K} + \frac{(K - 1)c}{K}\end{aligned}$$

## Unit 3

Thus, if  $c = v$ , then there is no change. If  $c = 0$ , then the variance of the average error decreases with  $K$ . There are various ensembling techniques. In the case of *Bagging* (Bootstrap Aggregating), the same training algorithm is used multiple times. The dataset is broken into  $K$  parts by sampling with replacement (see figure below for clarity) and a model is trained on each of those  $K$  parts. Because of sampling with replacement, the  $K$  parts have a few similarities as well as a few differences. These differences cause the difference in the predictions of the  $K$  models. Model averaging is a very strong technique.

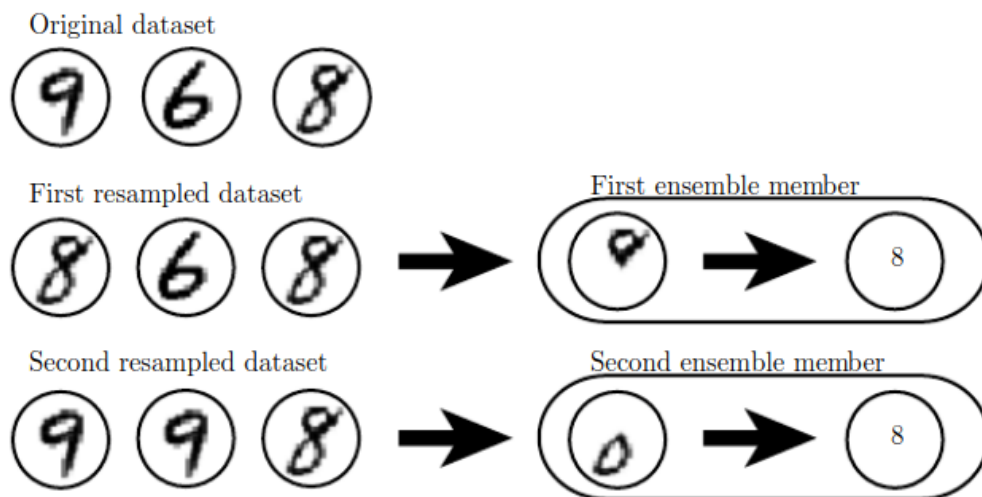


Figure 7.5: A cartoon depiction of how bagging works. Suppose we train an 8 detector on the dataset depicted above, containing an 8, a 6 and a 9. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. The first dataset omits the 9 and repeats the 8. On this dataset, the detector learns that a loop on top of the digit corresponds to an 8. On the second dataset, we repeat the 9 and omit the 6. In this case, the detector learns that a loop on the bottom of the digit corresponds to an 8. Each of these individual classification rules is brittle, but if we average their output, then the detector is robust, achieving maximal confidence only when both loops of the 8 are present.

## 12. Dropout

*Dropout* is a computationally inexpensive, yet powerful regularization technique. The problem with bagging is that we can't train an exponentially large number of models and store them for prediction later. Dropout makes bagging practical by making an inexpensive approximation. In a

## Unit 3

simplistic view, dropout trains the ensemble of all sub-networks formed by randomly removing a few non-output units by multiplying their outputs by 0. For every training sample, a mask is computed for all the input and hidden units independently. For clarification, suppose we have  $h$  hidden units in some layer. Then, a mask for that layer refers to a  $h$  dimensional vector with values either 0(remove the unit) or 1(keep the unit).

There are a few differences from bagging though:

- In bagging, the models are independent of each other, whereas in dropout, the different models share parameters, with each model taking as input, a sample of the total parameters.
- In bagging, each model is trained till convergence, but in dropout, each model is trained for just one step and the parameter sharing makes sure that subsequent updates ensure better predictions in the future.

At test time, we combine the predictions of all the models. In the case of bagging with  $K$  models, this was given by the arithmetic mean. In case of dropout, the probability that a model is chosen is given by  $p(\mu)$ , with  $\mu$  denoting the mask vector. The prediction then becomes  $\sum p(\mu)p(y|x, \mu)$ . This is not computationally feasible, and there's a better method to compute this in one go, using the *geometric mean* instead of the arithmetic mean.

We need to take care of two main things when working with geometric mean:

- None of the probabilities should be zero.
- Re-normalization to make sure all the probabilities sum to 1.

$$\hat{p}_{\text{ensemble}}(y | \mathbf{x}) = \sqrt[2^d]{\prod_{\mu} p(y | \mathbf{x}, \mu)},$$

## Unit 3

$$p_{\text{ensemble}}(y \mid \mathbf{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \mathbf{x})}.$$

The advantage for dropout is that first term can be approximated in one pass of the complete model by dividing the weight values by the keep probability (*weight scaling inference rule*). The motivation behind this is to capture the right expected values from the output of each unit, i.e. the total expected input to a unit at train time is equal to the total expected input at test time. A big advantage of dropout then is that it doesn't place any restriction on the *type* of model or training procedure to use.

### *Points to note:*

- Reduces the representational capacity of the model and hence, the model should be large enough to begin with.
- Works better with more data.
- Equivalent to  $L^2$  for linear regression, with different weight decay coefficient for each input feature.

### *Biological Interpretation:*

During sexual reproduction, genes could be swapped between organisms if they are unable to correctly adapt to the unusual features of any organism. Thus, the units in dropout learn to perform well regardless of the presence of other hidden units, and also in many different contexts.

Adding noise in the hidden layer is more effective than adding noise in the input layer. For e.g. let's assume that some unit learns to detect a nose in a face recognition task. Now, if this unit is

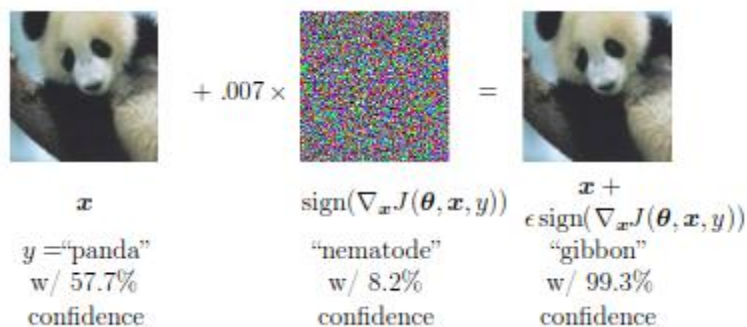


## Unit 3

removed, then some other unit either learns to redundantly detect a nose or associates some other feature (like mouth) for recognising a face. In either way, the model learns to make more use of the information in the input. On the other hand, adding noise to the input won't completely removed the nose information, unless the noise is so large as to remove most of the information from the input.

### 13. Adversarial Training

Deep Learning has *outperformed* humans in the task of Image Recognition, which might lead us to believe that these models have acquired a human-level understanding of an image. However, experimentally searching for an  $x'$  (given an  $x$ ), such that prediction made by the model changes, shows otherwise. As shown in the image below, although the newly formed image (adversarial image) looks almost exactly the same to a human, the model classifies it wrongly and that too with very high confidence:



**Adversarial training** refers to training on images which are adversarially generated and it has been shown to reduce the error rate. The main factor attributed to the above mentioned behaviour is the linearity of the model (say  $y = Wx$ ), caused by the main building blocks being primarily linear. Thus, a small change of  $\epsilon$  in the input causes a drastic change of  $W\epsilon$  in the output. The idea of adversarial training is to avoid this jumping and induce the model to be locally constant in the neighborhood of the training data.

## Unit 3

This can also be used in semi-supervised learning. For an unlabelled sample  $x$ , we can assign the label  $\hat{y}(x)$  using our model. Then, we find an adversarial example,  $x'$ , such that  $y(x') \neq \hat{y}(x)$  (an adversary found this way is called *virtual adversarial example*). The objective then is to assign the same class to both  $x$  and  $x'$ . The idea behind this is that different classes are assumed to lie on disconnected manifolds and a little push from one manifold shouldn't land in any other manifold.

### 14. Tangent Distance, Tangent Prop and manifold Tangent Classifier

Many ML models assume the data to lie on a low dimensional manifold to overcome the curse of dimensionality. The inherent assumption which follows is that small perturbations that cause the data to move along the manifold (it originally belonged to), shouldn't lead to different class predictions. The idea of the **tangent distance** algorithm to find the K-nearest neighbors using the distance metric as the distance between manifolds. A manifold  $M_i$  is approximated by the tangent plane at  $X_i$ , hence, this technique needs tangent vectors to be specified.

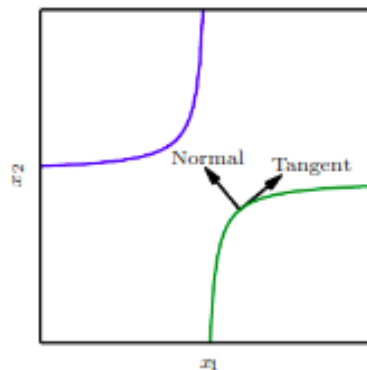


Figure 7.9: Illustration of the main idea of the tangent prop algorithm (Simard *et al.*, 1992) and manifold tangent classifier (Rifai *et al.*, 2011c), which both regularize the classifier output function  $f(\mathbf{x})$ . Each curve represents the manifold for a different class, illustrated here as a one-dimensional manifold embedded in a two-dimensional space. On one curve, we have chosen a single point and drawn a vector that is tangent to the class manifold (parallel to and touching the manifold) and a vector that is normal to the class manifold (orthogonal to the manifold). In multiple dimensions there may be many tangent directions and many normal directions. We expect the classification function to change rapidly as it moves in the direction normal to the manifold, and not to change as it moves along the class manifold. Both tangent propagation and the manifold tangent classifier regularize  $f(\mathbf{x})$  to not change very much as  $\mathbf{x}$  moves along the manifold. Tangent propagation requires the user to manually specify functions that compute the tangent directions (such as specifying that small translations of images remain in the same class manifold), while the manifold tangent classifier estimates the manifold tangent directions by training an autoencoder to fit the training data. The use of autoencoders to estimate manifolds is described in chapter 14.

## Unit 3

The **tangent prop** algorithm proposed to learn a neural network based classifier,  $f(x)$ , which is invariant to known transformations causing the input to move along its manifold. Local invariance would require that  $\nabla f(x)$  is perpendicular to the tangent vectors  $V(i)$ . This can also be achieved by adding a penalty term that minimizes the directional derivative of  $f(x)$  along each of the  $V(i)$ .

It is similar to data augmentation in that both of them use prior knowledge of the domain to specify various transformations that the model should be invariant to. However, tangent prop only resists infinitesimal perturbations while data augmentation causes invariance to much larger perturbations.

**Manifold Tangent Classifier** works in two parts:

- Use Autoencoders to learn the manifold structures using Unsupervised Learning.
- Use these learned manifolds with tangent prop

## UNIT -III

**Optimization for Training Deep Models: Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates, Approximate Second-Order Methods, Optimization Strategies and Meta-Algorithms.**

### **1. Optimization for Training Deep Models: Pure Optimization**

#### **1 Optimization in Deep Learning:**

In Deep Learning, with the help of loss function, the performance of the model is estimated/evaluated. This loss is used to train the network so that it performs better. Essentially, we try to minimize the Loss function. Lower Loss means the model performs better. The Process of minimizing any mathematical function is called Optimization.

Optimizers are algorithms or methods used to change the features of the neural network such

as weights and learning rate so that the loss is reduced. Optimizers are used to solve optimization problems by minimizing the function

The Goal of an Optimizer is to minimize the Objective Function(Loss Function based on the Training Data set). Simply Optimization is to minimize the Training Error.

#### **1.1 Need for Optimization:**

- Presence of Local Minima reduces the model performance
- Presence of Saddle Points which creates Vanishing Gradients or Exploding Gradient Issues
- To select appropriate weight values and other associated model parameters
- To minimize the loss value (Training error)

#### **2. Convex Optimization:**

Convex optimization is a kind of optimization which deals with the study of problem of minimizing convex functions. Here the optimization function is convex function.

All Linear functions are convex, so linear programming problems are convex problems. When we have a convex objective and a convex feasible region, then there can be only one optimal solution, which is globally optimal.

**Definition:** A set  $C \subseteq \mathbb{R}^n$  is convex if for  $x, y \in C$  and any  $\alpha \in [0, 1]$ ,

$$\alpha x + (1 - \alpha)y \in C$$

Convexity plays a vital role in the design of optimization algorithms. This is largely due to the fact that it is much easier to analyze and test algorithms in such a context.

Consider the given Figure 4.1 given below, select any two points in the region and join them by a straight Line. If the line and the selected points all lie inside the region then we call that region as Convex Region. Shown in following fig

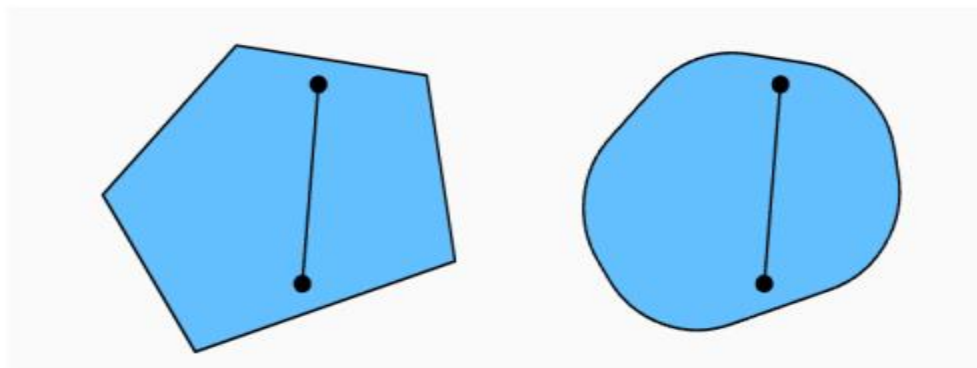


Figure 4.1: Convex Regions

A convex optimization problem is of the form:

$$\min_{x \in D} f(x)$$

subject to

$$g_i(x) \leq 0, \quad i = 1, \dots, m$$

$$h_j(x) = 0, \quad j = 1, \dots, r$$

where  $f$  and  $g_i$  are all convex, and  $h_j$  are affine. Any local minimizer of a convex optimization problem is a global minimizer.

### 3. Non-Convex Optimization:

- The Objective function is a non-convex function
- All non-linear problems can be modelled by using non-convex functions. (Linear functions are convex)
- It has Multiple feasible regions and multiple locally optimal points.
- There can't be a general algorithm to solve it efficiently in all cases

- Neural networks are universal function approximators, to do this, they need to be able to approximate non-convex functions.

### 3.1. How to solve non-convex problems?

- Stochastic gradient descent
- Mini-batching
- SVRG
- Momentum

### 3.2. Reasons For Non-Convexity:

- Presence of many Local Minima
- Presence of Saddle Points
- Very Flat Regions
- Varying Curvature



Figure 4.2: Convex Regions

## **2. Challenges in Neural Network Optimization**

Deep Neural Networks are very powerful and can do amazing things, but training them can be difficult. Training deep neural networks (DNNs) has led to impressive advances in artificial intelligence. However, it comes with hurdles like vanishing gradients, overfitting, and limited labeled data.

### **Vanishing and Exploding Gradients**

Deep learning networks can be problematic when the numbers change too quickly or slowly through many layers. This can make it hard for the network to learn and stay stable. This can cause difficulties for the network in learning and remaining stable.

**Solution:** Gradient clipping, advanced weight initialization, and skip connections help a computer learn things accurately and consistently.

## **Overfitting**

Overfitting happens when a model knows too much about the training data, so it can't make good predictions about new data. As a result, the model performs well on the training data but struggles to make accurate predictions on new, unseen data. It's essential to address overfitting by employing techniques like regularization, cross-validation, and more diverse datasets to ensure the model generalizes well to unseen examples.

**Solution:** Regularisation techniques help us ensure our models memorize the data and use what they've learned to make good predictions about new data. Techniques like dropout, L1/L2 regularisation, and early stopping can help us do this.

## **Data Augmentation and Preprocessing**

Data augmentation and preprocessing are techniques used to provide better information to the model during training, enabling it to learn more effectively and make accurate predictions.

**Solution:** Apply data augmentation techniques like rotation, translation, and flipping alongside data normalization and proper handling of missing values.

## **Label Noise**

Training data sometimes need to be corrected, making it hard for computers to do things well.

**Solution:** Using special kinds of math called "loss functions" can help ensure that the model you are using is not affected by label mistakes.

## **Imbalanced Datasets**

Datasets can have too many of one type of thing and need more of another type. This can cause models not to work very well for things not represented as much.

**Solution:** Classes can sometimes be uneven, meaning more people are in one group than another. To fix this, we can use special techniques like class weighting, oversampling, or data synthesis to ensure that all the classes have the same number of people.

## **Computational Resource Constraints**

Training deep neural networks can be very difficult and take a lot of computer power, especially if the model is very big.

**Solution:** Using multiple computers or special chips called GPUs and TPUs can help make learning faster and easier.

## Hyperparameter Tuning

Deep neural networks have numerous hyperparameters that require careful tuning to achieve optimal performance.

**Solution:** To efficiently find the best hyperparameters, such as Bayesian optimization or genetic algorithms, utilize automated hyperparameter optimization methods.

## Convergence Speed

It is important to ensure a model works quickly when using lots of data and complicated designs.

**Solution:** Adopt learning rate scheduling or adaptive algorithms like Adam or RMSprop to expedite convergence.

## Activation Function Selection

Using the proper activation function when building a machine-learning model is important. This helps ensure the model works properly and yields correct results.

**Solution:** ReLU and its variants (Leaky ReLU, Parametric ReLU) are popular choices due to their ability to mitigate vanishing activation issues.

## Gradient Descent Optimization

Gradient descent algorithms help computers solve problems but sometimes need help when it is very difficult.

**Solution:** Advanced techniques can help us navigate difficult problems better. Examples are stochastic gradient descent with momentum and Nesterov Accelerated Gradient.

## Memory Constraints

Computers need a lot of memory to train large models and datasets, but they can work properly if there is enough memory.

**Solution:** Reduce memory usage by applying model quantization, using mixed-precision training, or employing memory-efficient architectures like MobileNet or EfficientNet.

## Transfer Learning and Domain Adaptation

Deep learning networks need lots of data to work well. If they don't get enough data or the data is different, they won't work as well.

**Solution:** Leverage transfer learning or domain adaptation techniques to transfer knowledge from pre-trained models or related domains.



## Exploring Architecture Design Space

Designing buildings is difficult because there are many different ways to do it. Choosing the best way to create a building for a specific purpose can take time and effort.

**Solution:** Use automated neural architecture search (NAS) algorithms to explore the design space and discover architectures tailored to the task.

## Adversarial Attacks

Deep neural networks are unique ways of understanding data. But they can be tricked by minimal changes that we can't see. This can make them give wrong answers.

**Solution:** Employ adversarial training, defensive distillation, or certified robustness methods to enhance the model's robustness against adversarial attacks.

## Interpretability and Explainability

Understanding the decisions made by deep neural networks is crucial in critical applications like healthcare and autonomous driving.

**Solution:** Adopt techniques such as LIME (Local Interpretable Model-Agnostic Explanations) or SHAP (SHapley Additive exPlanations) to explain model predictions.

## Handling Sequential Data

Training deep neural networks on sequential data, such as time series or natural language sequences, presents unique challenges.

**Solution:** Utilize specialized architectures like recurrent neural networks (RNNs) or transformers to handle sequential data effectively.

## Limited Data

Training deep neural networks with limited labeled data is a common challenge, especially in specialized domains.

**Solution:** Consider semi-supervised, transfer, or active learning to make the most of available data.

## Catastrophic Forgetting

When a model forgets previously learned knowledge after training on new data, it encounters the issue of catastrophic forgetting.

**Solution:** Implement techniques like elastic weight consolidation (EWC) or knowledge distillation to retain old knowledge during continual learning.

### **Hardware and Deployment Constraints**

Using trained models on devices with not much computing power can be hard.

**Solution:** Scientists use special techniques to make computer models run better on devices with limited resources.

### **Data Privacy and Security**

When training computers to do complex tasks, it is essential to keep data private and ensure the computers are secure.

**Solution:** Employ federated learning, secure aggregation, or differential privacy techniques to protect data and model privacy.

### **Long Training Times**

Training deep neural networks is like doing a challenging puzzle. It takes a lot of time to assemble the puzzle, especially if it is vast and has a lot of pieces.

**Solution:** Special tools like GPUs or TPUs can help us train our computers faster. We can also try using different computers simultaneously to make the training even quicker.

### **Exploding Memory Usage**

Some models are too big and need a lot of space, so they are hard to use on regular computers.

**Solution:** Explore memory-efficient architectures, use gradient checkpointing, or consider model parallelism for training.

### **Learning Rate Scheduling**

Setting an appropriate learning rate schedule can be challenging, affecting model convergence and performance.

**Solution:** Using special learning rate schedules can help make learning easier and faster. These schedules can be used to help teach things in a better way.

### **Avoiding Local Minima**

Deep neural networks can get stuck in local minima during training, impacting the model's final performance.

**Solution:** Using unique strategies like simulated annealing, momentum-based optimization, and evolutionary algorithms can help us escape difficult spots.

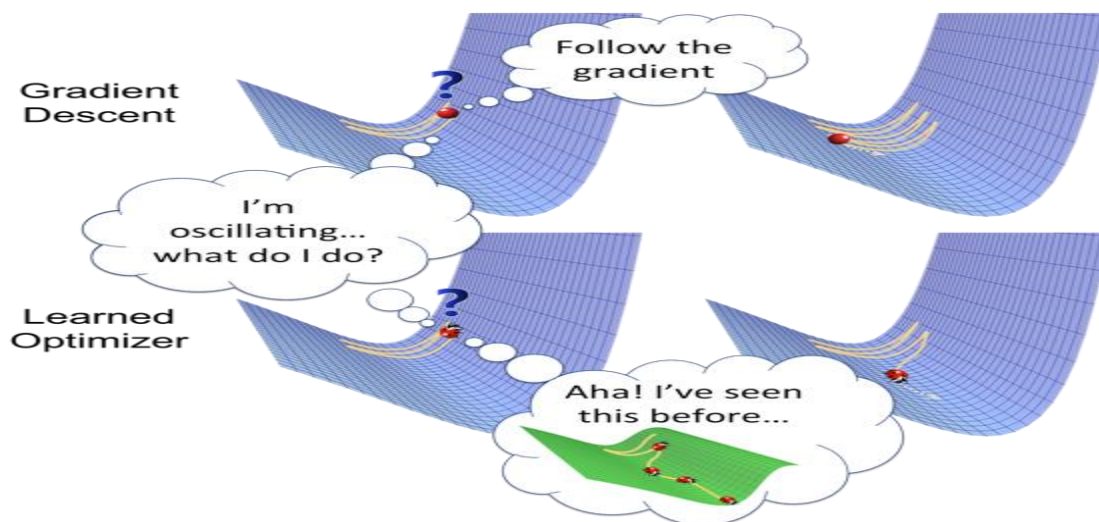
### Unstable Loss Surfaces

Finding the best way to do something can be very hard when there are many different options because the surface it is on is complicated and bumpy.

**Solution:** Utilize weight noise injection, curvature-based optimization, or geometric methods to stabilize loss surfaces.

### 3. Basic Algorithms,

Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses



Optimization algorithms or strategies are responsible for reducing the losses and to provide the most accurate results possible.

We'll learn about different types of optimizers and their advantages:

## Gradient Descent

Gradient Descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification

algorithms. Backpropagation in neural networks also uses a gradient descent algorithm.

Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function. It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized.

algorithm:  $\theta = \theta - \alpha \cdot \nabla J(\theta)$

### **Advantages:**

1. Easy computation.
2. Easy to implement.
3. Easy to understand.

### **Disadvantages:**

1. May trap at local minima.
2. Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may take years to converge to the minima.
3. Requires large memory to calculate gradient on the whole dataset.

### **Stochastic Gradient Descent**

It's a variant of Gradient Descent. It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example. So, if the dataset

contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent.

**$\theta = \theta - \alpha \cdot \nabla J(\theta; x(i); y(i))$  , where  $\{x(i), y(i)\}$  are the training examples.**

As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities.

### **Advantages:**

1. Frequent updates of model parameters hence, converges in less time.
2. Requires less memory as no need to store values of loss functions.
3. May get new minima's.

### **Disadvantages:**

1. High variance in model parameters.
2. May shoot even after achieving global minima.
3. To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

## **Mini-Batch Gradient Descent**

It's best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent. It updates the model parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated.

**$\theta = \theta - \alpha \cdot \nabla J(\theta; B(i))$ , where  $\{B(i)\}$  are the batches of training examples.**

**Advantages:**

1. Frequently updates the model parameters and also has less variance.
2. Requires medium amount of memory.

**All types of Gradient Descent have some challenges:**

1. Choosing an optimum value of the learning rate. If the learning rate is too small than gradient descent may take ages to converge.
2. Have a constant learning rate for all the parameters. There may be some parameters which we may not want to change at the same rate.
3. May get trapped at local minima.

**Momentum**

Momentum was invented for reducing high variance in SGD and softens the convergence. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by ' $\gamma$ '.

$$V(t) = \gamma V(t-1) + \alpha \cdot \nabla J(\theta)$$

Now, the weights are updated by  **$\theta = \theta - V(t)$ .**

The momentum term  $\gamma$  is usually set to 0.9 or a similar value.

### **Advantages:**

1. Reduces the oscillations and high variance of the parameters.
2. Converges faster than gradient descent.

### **Disadvantages:**

1. One more hyper-parameter is added which needs to be selected manually and accurately.

## **Nesterov Accelerated Gradient**

Momentum may be a good method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed. It is a look ahead method. We know we'll be using  $\gamma V(t-1)$  for modifying the weights so,  $\theta - \gamma V(t-1)$  approximately tells us the future location. Now, we'll calculate the cost based on this future parameter rather than the current one.

$V(t) = \gamma V(t-1) + \alpha \cdot \nabla J(\theta - \gamma V(t-1))$  and then update the parameters using  $\theta = \theta - V(t)$ .



### **Advantages:**

1. Does not miss the local minima.
2. Slows if minima's are occurring.

### Disadvantages:

1. Still, the hyperparameter needs to be selected manually.

### Adagrad

One of the disadvantages of all the optimizers explained is that the learning rate is constant for all parameters and for each cycle. This optimizer changes the learning rate. It changes the learning rate ' $\eta$ ' for each parameter and at every time step ' $t$ '. It's a type second order optimization algorithm. It works on the derivative of an error function.

$$g_{t,i} = \nabla_{\theta} J(\theta_{t,i}),$$

A derivative of loss function for given parameters at a given time  $t$ .

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

Update parameters for given input  $i$  and at time/iteration  $t$

$\eta$  is a learning rate which is modified for given parameter  $\theta(i)$  at a given time based on previous gradients calculated for given parameter  $\theta(i)$ .

We store the sum of the squares of the gradients w.r.t.  $\theta(i)$  up to time step  $t$ , while  $\epsilon$  is a smoothing term that avoids division by zero (usually on the order of  $1e-8$ ). Interestingly, without the square root operation, the algorithm performs much worse.



It makes big updates for less frequent parameters and a small step for frequent parameters.

**Advantages:**

1. Learning rate changes for each training parameter.
2. Don't need to manually tune the learning rate.
3. Able to train on sparse data.

**Disadvantages:**

1. Computationally expensive as a need to calculate the second order derivative.
2. The learning rate is always decreasing results in slow training.

**AdaDelta**

It is an extension of **AdaGrad** which tends to remove the *decaying learning Rate* problem of it. Instead of accumulating all previously squared gradients, **Adadelta** limits the window of accumulated past gradients to some fixed size **w**. In this exponentially moving average is used rather than the sum of all the gradients.

$$\mathbb{E}[g^2](t) = \gamma \cdot \mathbb{E}[g^2](t-1) + (1-\gamma) \cdot g^2(t)$$

We set  **$\gamma$**  to a similar value as the momentum term, around 0.9.

$$\mathbb{E}[g^2]_t = \gamma \mathbb{E}[g^2]_{t-1} + (1 - \gamma) g_t^2,$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} \cdot g_t.$$

### Advantages:

1. Now the learning rate does not decay and the training does not stop.

### Disadvantages:

1. Computationally expensive.

## Adam

[Adam](#) (Adaptive Moment Estimation) works with momentums of first and second order. The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search. In addition to storing an exponentially decaying average of past squared gradients like **AdaDelta**, **Adam** also keeps an exponentially decaying average of past gradients **M(t)**.

**M(t)** and **V(t)** are values of the first moment which is the **Mean** and the second moment which is the **uncentered variance** of the gradients respectively.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}.$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

Here, we are taking mean of **M(t)** and **V(t)** so that **E[m(t)]** can be equal to **E[g(t)]** where, **E[f(x)]** is an expected value of **f(x)**.

To update the parameter:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

date the parameters

The values for  $\beta_1$  is 0.9 , 0.999 for  $\beta_2$ , and  $(10 \times \exp(-8))$  for ' $\epsilon$ '.

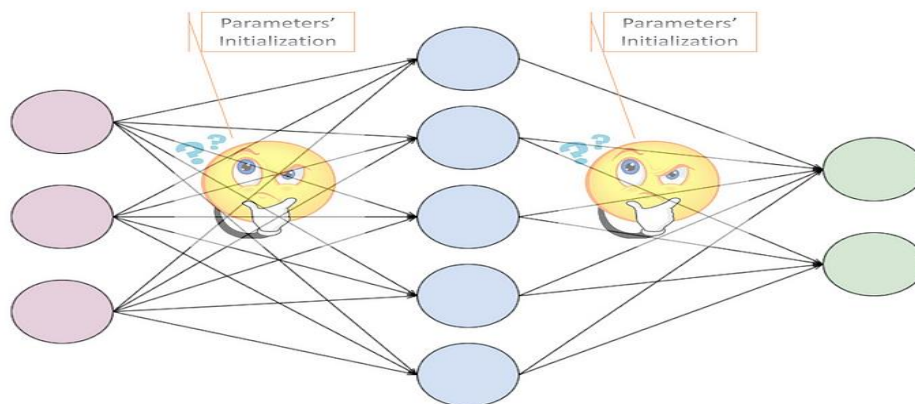
### **Advantages:**

1. The method is too fast and converges rapidly.
2. Rectifies vanishing learning rate, high variance.

### **Disadvantages:**

Computationally costly.

### **4.Parameter Initialization Strategies**



Optimization, in Machine Learning/Deep Learning contexts, is the process of changing the model's parameters to improve its performance. In other words, it's the process of finding the best parameters in the predefined hypothesis space to get the best possible performance. There are three kinds of optimization algorithms:

- Optimization algorithm that is not iterative and simply solves for one point.
- Optimization algorithm that is iterative in nature and converges to acceptable solution regardless of the parameters initialization such as gradient descent applied to logistic regression.
- Optimization algorithm that is iterative in nature and applied to a set of problems that have non-convex loss functions such as neural networks. Therefore, parameters' initialization plays a critical role in speeding up convergence and achieving lower error rates.

we'll look at three different cases of parameters' initialization and see how this affects the error rate:

1. Initialize all parameters to zero.
2. Initialize parameters to random values from standard normal distribution or uniform distribution and multiply it by a scalar such as 10.
3. Initialize parameters based on:
  - Xavier recommendation.
  - Kaiming He recommendation.

To illustrate the above cases, we'll use the cats vs dogs dataset which consists of 50 images for cats and 50 images for dogs. Each image is

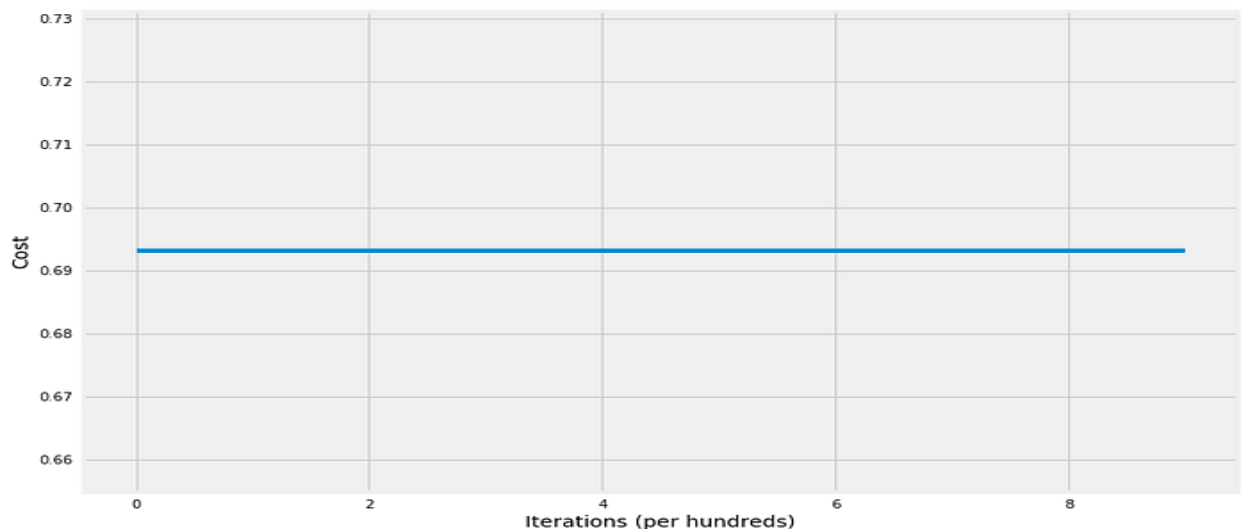
150 pixels x 150 pixels on RGB color scale. Therefore, we would have 67,500 features where each column in the input matrix would be one image which means our input data would have 67,500 x 100 dimension.

## Initializing all parameters to zero

Here, we'll initialize all weight matrices and biases to zeros and see how this would affect the error rate as well as the learning parameters.

```
# train NN with zeros initialization parameters
layers_dims = [X.shape[0], 5, 5, 1]
parameters = model(X, Y, layers_dims,
hidden_layers_activation_fn="tanh", initialization_method="zeros")
accuracy(X, parameters, Y,"tanh")The cost after 100 iterations is:
0.6931471805599453
The cost after 200 iterations is: 0.6931471805599453
The cost after 300 iterations is: 0.6931471805599453
The cost after 400 iterations is: 0.6931471805599453
The cost after 500 iterations is: 0.6931471805599453
The cost after 600 iterations is: 0.6931471805599453
The cost after 700 iterations is: 0.6931471805599453
The cost after 800 iterations is: 0.6931471805599453
The cost after 900 iterations is: 0.6931471805599453
The cost after 1000 iterations is: 0.6931471805599453 The accuracy
rate is: 50.00%.
```

Cost curve: learning rate = 0.01 and zeros initialization method



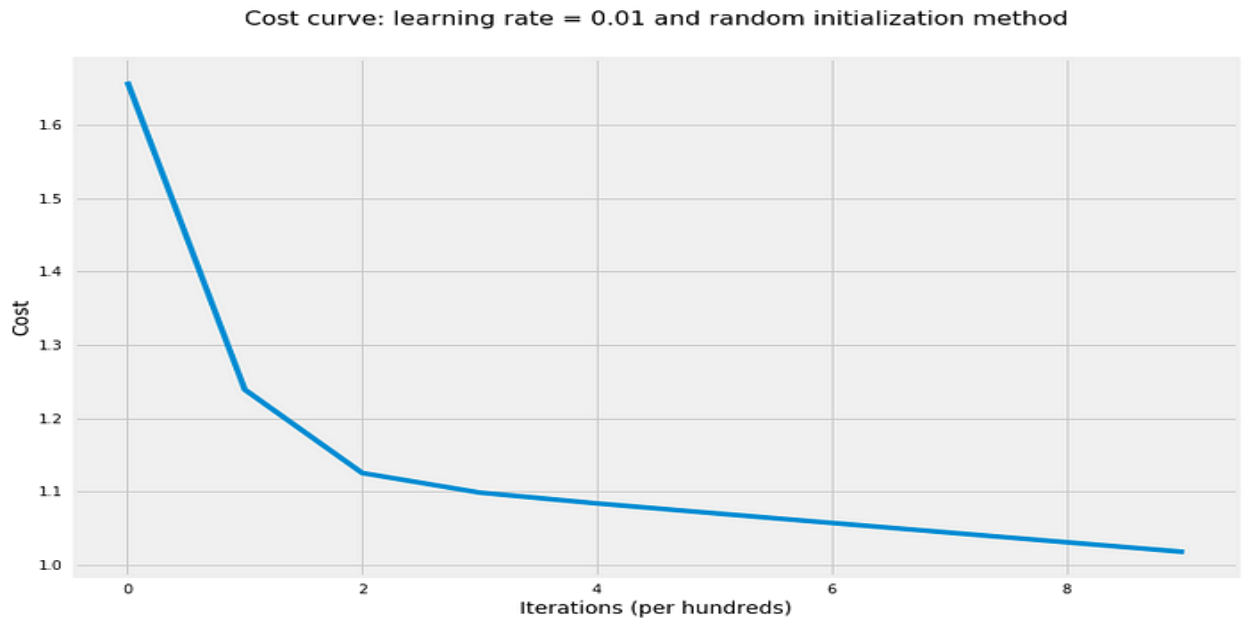
As the cost curve shows, the neural network didn't learn anything! That is because of symmetry between all neurons which leads to all neurons have the same update on every iteration. Therefore, regardless of how many iterations we run the optimization algorithms, all the neurons would still get the same update and no learning would happen. As a result, we must **break symmetry** when initializing parameters so that the model would start learning on each update of the gradient descent.

## Initializing parameters with big random values

There is no big difference if the random values are initialized from standard normal distribution or uniform distribution so we'll use standard normal distribution in our examples. Also, we'll multiply the random values by a big number such as 10 to show that initializing parameters to big values may cause our optimization to have higher error rates (and even diverge in some cases). Let's now train our neural network where all weight matrices have been initialized using the

following formula: `np.random.randn() * 10`

```
# train NN with random initialization parameters
layers_dims = [X.shape[0], 5, 5, 1]
parameters = model(X, Y, layers_dims,
hidden_layers_activation_fn="tanh", initialization_method="random")
accuracy(X, parameters, Y, "tanh")
The cost after 100 iterations is: 1.2413142077549013
The cost after 200 iterations is: 1.1258751902393416
The cost after 300 iterations is: 1.0989052435267657
The cost after 400 iterations is: 1.0840966471282327
The cost after 500 iterations is: 1.0706953292105978
The cost after 600 iterations is: 1.0574847320236294
The cost after 700 iterations is: 1.0443168708889223
The cost after 800 iterations is: 1.031157857251139
The cost after 900 iterations is: 1.0179838815204902
The cost after 1000 iterations is: 1.004767088515343 The accuracy
rate is: 55.00%.
```



Random initialization here is helping but still the loss function has high value and may take long time to converge and achieve a significantly low value.

## Initializing parameters based on He and Xavier recommendations

We'll explore two initialization methods:

- Kaiming He method is best applied when activation function applied on hidden layers is Rectified Linear Unit (ReLU). so that the weight on each hidden layer would have the following variance:  $\text{var}(W^l) = 2/n^{(l-1)}$ . We can achieve this by multiplying the random values from standard normal distribution by

$$\sqrt{\frac{2}{\text{number of units in previous layer}}}$$

- Xavier method is best applied when activation function applied on hidden layers is Hyperbolic Tangent so that the weight on each hidden layer would have the following variance:  $\text{var}(W^l) = 1/n^{(l-1)}$ . We can achieve this by multiplying the random values from standard normal distribution by

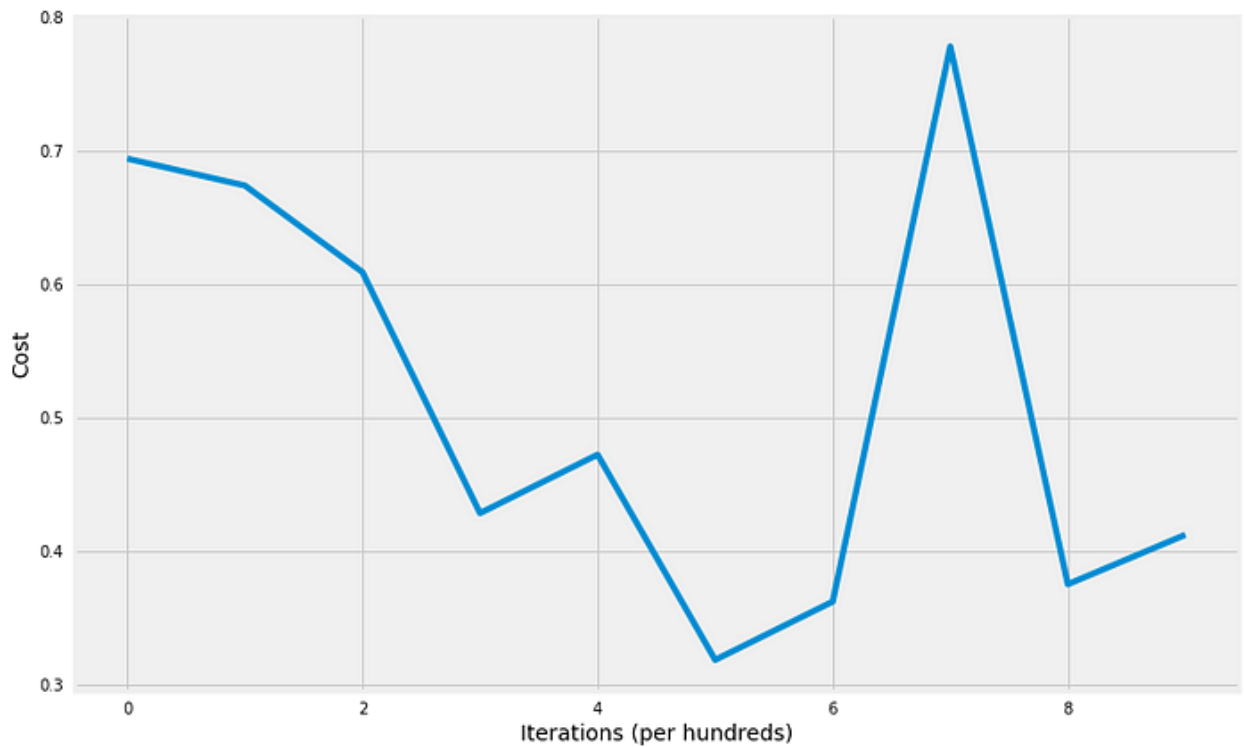
$$\sqrt{\frac{1}{\text{number of units in previous layer}}}$$

We'll train the network using both methods and look at the results.

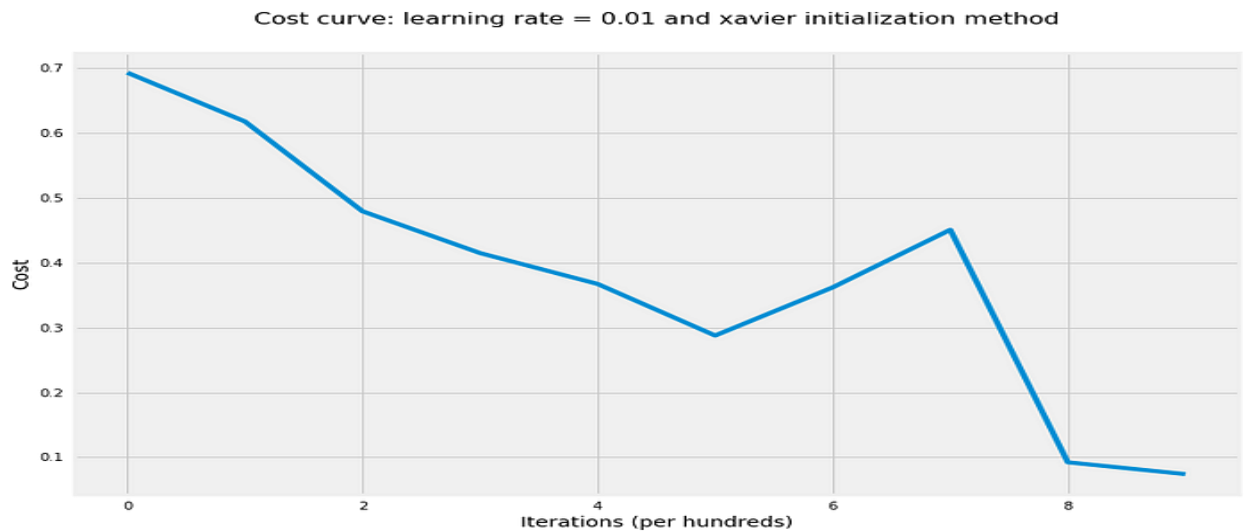
```
# train NN where all parameters were initialized based on He
recommendation
layers_dims = [X.shape[0], 5, 5, 1]
parameters = model(X, Y, layers_dims,
hidden_layers_activation_fn="tanh", initialization_method="he")
accuracy(X, parameters, Y, "tanh")The cost after 100 iterations is:
0.6300611704834093
The cost after 200 iterations is: 0.49092836452522753
The cost after 300 iterations is: 0.46579423512433943
The cost after 400 iterations is: 0.6516254192289226
The cost after 500 iterations is: 0.32487779301799485
The cost after 600 iterations is: 0.4631461605716059
The cost after 700 iterations is: 0.8050310690163623
The cost after 800 iterations is: 0.31739195517372376
The cost after 900 iterations is: 0.3094592175030812
The cost after 1000 iterations is: 0.19934509244449203The accuracy
rate is: 99.00%.
```



Cost curve: learning rate = 0.01 and he initialization method



```
# train NN where all parameters were initialized based on Xavier
recommendation
layers_dims = [X.shape[0], 5, 5, 1]
parameters = model(X, Y, layers_dims,
hidden_layers_activation_fn="tanh", initialization_method="xavier")
accuracy(X, parameters, Y, "tanh")
The cost after 100 iterations is: 0.6351961521800779
The cost after 200 iterations is: 0.548973489787121
The cost after 300 iterations is: 0.47982386652748565
The cost after 400 iterations is: 0.32811768889968684
The cost after 500 iterations is: 0.2793453045790634
The cost after 600 iterations is: 0.3258507563809604
The cost after 700 iterations is: 0.2873032724176074
The cost after 800 iterations is: 0.0924974839405706
The cost after 900 iterations is: 0.07418011931058155
The cost after 1000 iterations is: 0.06204402572328295
The accuracy rate is: 99.00%.
```



As shown from applying the four methods, parameters' initial values play a huge role in achieving low cost values as well as converging and achieve lower training error rates. The same would apply to test error rate if we had test data.

#### 5.Algorithms with Adaptive Learning Rates

there are methods which tune learning rates adaptively and work for a broad range of parameters.

### **Adagrad:**

In Adagrad, the variable  $c$ , called cache, keeps track of the per-parameter sum of the squared gradients, which in turn is used to normalize the parameter update element-wise. Essentially what happens is that the weights receiving high gradients will have their effective learning rate reduced. On the other hand, weights that receive small or infrequent updates will have their effective learning rate increased.

$$c = c + dx^2$$

$$x = x - \frac{\alpha * dx}{\sqrt{c + \epsilon}}$$

Here  $\epsilon$  is the smoothing term (usually takes value of  $1e-6$ ) needed to avoid division by zero. One issue with Adagrad is that in case of Deep Learning, the monotonic rate usually proves to be aggressive and leads to early learning stoppage.

### **RMSprop:**

This highly effective method is cited through lecture 2, slide 6 of Geoff's Hinton coursera class on [Neural Networks for Machine Learning](#). The RMSprop update counters the aggressive and monotonically decreasing learning rate behavior of Adagrad by utilizing a moving average of squared gradients (unlike squared gradients in Adagrad).

$$c = \gamma * c + (1 - \gamma) * dx^2$$
$$x = x - \frac{\alpha * dx}{\sqrt{c + \epsilon}}$$

Here  $\gamma$ , the decay rate, is another hyperparameter with typical value being 0.9–0.99. We notice, that the weights ( $x$ ) update is similar to that of the Adagrad, however the cache  $c$  is slightly different. Thus, RMSprop still provide the learning rate of each weight based on the magnitudes of its gradients. However, by using moving average of squared gradients, it ensures that the updates do not get monotonically smaller.

### **Adam:**

Adam update can be considered as RMSprop with momentum. In place of raw and noisy gradient vector  $dx$ , a “smooth” version of gradient  $m$  is used.

$$\begin{aligned}m &= \beta * m + (1 - \beta) * dx \\c &= \gamma * c + (1 - \gamma) * dx^2 \\x &= x - \frac{\alpha * m}{\sqrt{c} + \epsilon}\end{aligned}$$

Adam often works slightly better than RMSprop and considered the default update method for practitioners. However, carefully used SGD with momentum can surpass the performance of Adam and may lead to a better local minima

## **6. Approximate Second-Order Methods**

Most of the optimizers used in deep learning are (stochastic) gradient descent methods. They only consider the gradient of the loss function. In comparison, second order methods also take the curvature of the loss function into account.

second order methods require computing the Hessian matrix, which contains  $N \times N$  elements, where  $N$  is the number of parameters (weights) in the neural network. This is infeasible for most models out there. AdaHessian contains some interesting techniques to tackle that problem. However, they might look complex at first sight. This article goes through these techniques which efficiently compute an approximation of the Hessian matrix.

Gradient descent models the loss function using only its first order derivatives: the algorithm takes a small enough step (controlled by the learning rate) in the direction of the negative gradient and thereby the loss value decreases. Curvature information (second order derivatives collected in the Hessian matrix) of the loss function is ignored.

Fig. 1 shows the loss function (green) and the gradient (red) at the current position. The left plot shows the case where the gradient exactly matches the loss function locally. The right plot shows the case where the loss function turns upwards when moving in the direction of the negative gradient. While it might make sense to apply a large update step in the left plot, a smaller update step is needed in the right plot to avoid overshooting the minimum

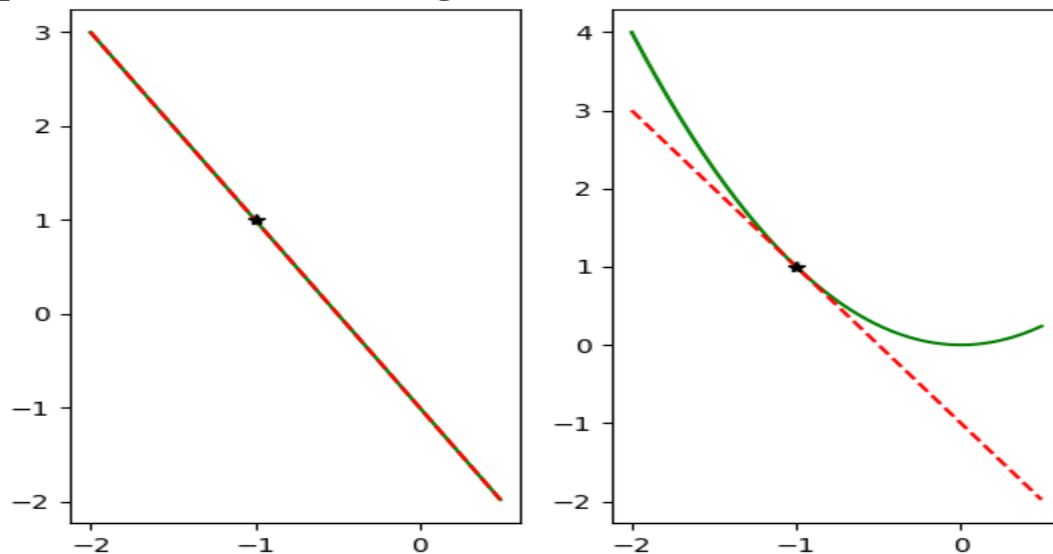


Fig. 1: Loss function  $f(w)$  (green) and its gradient at  $w=-1$  (red). (Image by author)

## The Newton update step

Newton's method handles exactly this: it takes both the gradient and the curvature at the current position into account. It models the loss function  $f(w)$  (with  $w$  the parameters or weights of the model) locally with a quadratic model. The update step is derived as follows (see Fig. 2 for formulas):

- A quadratic model  $m$  is computed using a Taylor polynomial

- $g$  is the gradient and  $H$  the Hessian matrix
  - To find the minimum of that model  $m$ , compute the derivative of the model, set it to zero, and solve for the update step  $\Delta w$
  - This gives the Newton update with the negative gradient scaled and rotated by the inverse of the Hessian
- $$m(w_0 + \Delta w) = f(w_0) + g(w_0) \cdot \Delta w + 1/2 \cdot \Delta w^T \cdot H(w_0) \cdot \Delta w$$
- $$\nabla m(w_0 + \Delta w) = g(w_0) + H(w_0) \cdot \Delta w$$
- $$\nabla m(w_0 + \Delta w) = 0 \rightarrow \Delta w = -H(w_0)^{-1} \cdot g(w_0)$$

Both gradient and Hessian could be computed with a deep learning framework like PyTorch, which is everything that is needed to apply Newton's method. However, for a neural network model with  $N$  parameters, the Hessian matrix consists of  $N \times N$  elements, which is not feasible for typical models. AdaHessian approximates the Hessian matrix with a diagonal matrix, which only consists of  $N$  elements (same size as the gradient vector).

## Compute diagonal of Hessian

We now have the Newton update formula and we restrict the Hessian approximation to a diagonal matrix. Let's see how it is computed.

### Hutchinson's method

Fig. 3 shows the formula of Hutchinson's method, which computes the diagonal elements of the Hessian:

- Create a random vector  $z$  by flipping a coin for each of its elements, and set +1 for head and -1 for tail, so in the 2D case  $z$  could be (1, -1) as an example
  - Compute matrix-vector product  $H \cdot z$
  - Multiply  $z$  element-wise (denoted by  $\odot$ ) with the result from the previous step  $z \odot (H \cdot z)$
  - Compute expected value (or the average value using multiple instances of the  $z$  vector in a real implementation)
- $$\text{diag}(H) = \mathbb{E}[z \odot (H \cdot z)]$$

Fig. 3: Hutchinson's method to compute diagonal of  $H$ . (Image by author)

This looks strange at first sight. We already need  $H$  to get the diagonal elements of  $H$  — this does not sound very clever. But it turns out that we only need to know the result of  $H \cdot z$  (a vector), which is easy to compute, so we never need to know the actual elements of the full Hessian matrix.

But how does Hutchinson's method work? When writing it down for the 2D case (Fig. 4) it is easy to see. Element-wise multiplication means multiplying the vectors row-wise. Inspecting the result of  $z \odot (H \cdot z)$  shows terms with  $z_1^2$  (and  $z_2^2$ ) and terms with  $z_1 \cdot z_2$  in it. When computing  $z \odot (H \cdot z)$  for multiple trials,  $z_1^2$  (and  $z_2^2$ ) is always +1, while  $z_1 \cdot z_2$  gives +1 in 50% of the trials and -1 for the other 50% (simply write down all possible products:  $1 \cdot 1$ ,  $1 \cdot (-1)$ ,  $(-1) \cdot 1$ ,  $(-1) \cdot (-1)$ ). When computing the mean over multiple trials, the terms containing

$z_1 \cdot z_2$  tend to zero, and we are left with a vector of the diagonal elements of  $H$ .

$$\begin{aligned}
 H \cdot z &= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} a \cdot z_1 + b \cdot z_2 \\ c \cdot z_1 + d \cdot z_2 \end{pmatrix} \\
 z \odot (H \cdot z) &= \begin{pmatrix} a \cdot z_1^2 + b \cdot z_1 \cdot z_2 \\ c \cdot z_1 \cdot z_2 + d \cdot z_2^2 \end{pmatrix} \\
 \mathbb{E}[z \odot (H \cdot z)] &= \begin{pmatrix} a \\ d \end{pmatrix} = \text{diag}(H)
 \end{aligned}$$

Fig. 4: Hutchinson's method in the 2D case. (Image by author)

## Matrix vector product

There is only one problem left: we don't have the Hessian matrix  $H$  which is used in Hutchinson's method. However, as already mentioned we don't actually need the Hessian. It is enough if we have the result of  $H \cdot z$ . It is computed with the help of PyTorch's automatic differentiation functions: if we take the gradient already computed by PyTorch, multiple it by  $z$ , and differentiate w.r.t. the parameter vector  $w$ , we get the same result as if we compute  $H \cdot z$  directly. So we can compute  $H \cdot z$  without knowing the elements of  $H$ . Fig. 5 shows why this is true: differentiating the gradient one more time gives the Hessian and  $z$  is treated as a constant.

$$\frac{\partial g(w) \cdot z}{\partial w} = \frac{\partial g(w)}{\partial w} \cdot z = H \cdot z$$

Fig. 5: Equality of what we compute with PyTorch's automatic differentiation (left) and the matrix vector product  $H \cdot z$  that we need for Hutchinson's method (right). (Image by author)

We now have the gradient and the diagonal values of the Hessian, so we could directly apply the Newton update step. However, AdaHessian



follows a similar approach as the Adam optimizer: it averages over multiple time-steps to get a smoother estimate of the gradient and the Hessian.

A single update step looks as follows:

- Compute current gradient
- Compute current diagonal Hessian approximation
- Average gradient by mixing it with earlier values
- Average Hessian by mixing it with earlier values, and take care that the diagonal elements are positive (otherwise, we could end up moving in an ascent direction)
- Compute Newton update step, including a learning rate parameter to avoid too large steps leading to divergence

In Fig. 6 a simple quadratic function is minimized with gradient descent and AdaHessian. Here, AdaHessian is used without momentum. On this toy example, when compared to gradient descent, it converges faster and does not need tuning of the learning rate. Of course, minimizing a quadratic function favors second order optimizers.

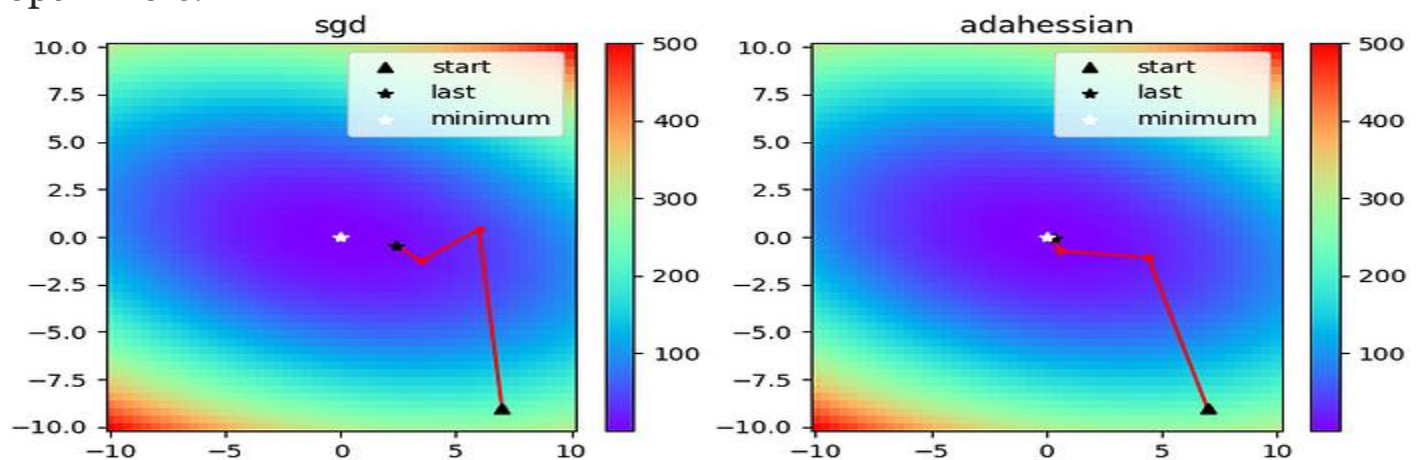


Fig. 6: 3 steps towards the minimum of a quadratic function. Left: gradient descent with an appropriate learning rate. Right: Newton's method using a diagonal Hessian approximation (right). (Image by author)

## **7.Optimization Strategies and Meta-Algorithms**

**Meta-learning** in an AI system is the capability to learn to perform a variety of complex tasks by applying the concepts it learned for one task to other activities.

Many optimization techniques are not exactly algorithms but rather general templates that can be specialized to yield algorithms, or subroutines that can be incorporated into many different algorithms

- i. **Batch normalization** is a technique for training very deep neural networks that normalizes the contributions to a layer for every mini-batch. This has the impact of settling the learning process and drastically decreasing the number of training epochs required to train deep neural networks.

In **deep learning**, preparing a **deep neural network** with many layers as they can be delicate to the underlying initial random weights and design of the learning algorithm. One potential purpose behind this trouble is the distribution of the inputs to layers somewhere down in the network may change after each mini-batch when the weights are refreshed. This can make the learning algorithm always pursue a moving target. This adjustment in the distribution of inputs to layers in the network has alluded to the specialized name **internal covariate shift**.

The challenge is that the model is refreshed layer-by-layer in reverse from the output to the input utilizing an estimate of error that accept the weights in the layers preceding the current layer are fixed. **Batch normalization** gives a rich method of parametrizing practically any deep neural network. The reparameterization fundamentally decreases the issue of planning updates across numerous layers.

- ii. **Coordinate descent** is an optimization algorithm that successively minimizes along coordinate directions to find the minimum of a function.

At each iteration, the algorithm determines a [coordinate](#) or coordinate block via a coordinate selection rule, then exactly or inexactly minimizes over the corresponding coordinate hyperplane while fixing all other coordinates or coordinate blocks. A [line search](#) along the [coordinate](#) direction can be performed at the current iterate to

determine the appropriate step size. Coordinate descent is applicable in both differentiable and derivative-free contexts.

Coordinate descent is based on the idea that the minimization of a multivariable function  $F(\mathbf{x})$  can be achieved by minimizing it along one direction at a time, i.e., solving univariate (or at least much simpler) optimization problems in a loop.<sup>[1]</sup> In the simplest case of *cyclic coordinate descent*, one cyclically iterates through the directions, one at a time, minimizing the objective function with respect to each coordinate direction at a time. That is, starting with initial variable values

$$\mathbf{x}^0 = (x_1^0, \dots, x_n^0),$$

round  $k + 1$  defines  $\mathbf{x}^{k+1}$  from  $\mathbf{x}^k$  by iteratively solving the single variable optimization problems

$$x_i^{k+1} = \arg \min_{y \in \mathbb{R}} f(x_1^{k+1}, \dots, x_{i-1}^{k+1}, y, x_{i+1}^k, \dots, x_n^k)^{[2]}$$

for each variable  $x_i$  of  $\mathbf{x}$ , for  $i$  from 1 to  $n$ .

Thus, one begins with an initial guess  $\mathbf{x}^0$  for a local minimum of  $F$ , and gets a sequence  $\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \dots$  iteratively.

By doing [line search](#) in each iteration, one automatically has

$$F(\mathbf{x}^0) \geq F(\mathbf{x}^1) \geq F(\mathbf{x}^2) \geq \dots$$

It can be shown that this sequence has similar convergence properties as steepest descent. No improvement after one cycle of [line search](#) along coordinate directions implies a stationary point is reached.

This process is illustrated below.

