UNIT - II

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, Learning vs Pure Optimization, Challenges in Neural Network Optimization, Basic Algorithms, Parameter Initialization Strategies, Algorithms with Adaptive Learning Rates.

Regularization for Deep Learning :

Regularization : A central problem in machine learning and deep learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning and deep learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as **regularization.**

Regularization techniques are essential for preventing overfitting and improving the generalization capability of machine learning models.

"any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias.

There are many regularization strategies.

Parameter Norm Penalties :

Parameter norm penalties are regularization techniques applied in machine learning and statistical models to prevent overfitting by penalizing large parameter values. By adding a penalty term based on the norm of the model parameters to the loss function, these methods encourage simpler models with smaller parameter values, leading to better generalization.

Before delving into the regularization behavior of different norms, we note that for neural networks, we typically choose to use a parameter norm penalty Ω that penalizes **only the weights** of the affine transformation at each layer and leaves the biases unregularized. The biases typically require less data to fit accurately than the weights.

1. **L1regularization:** L1 regularization on the model parameter w is defined as:

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

It is also known as **Lasso Regression (Least Absolute Shrinkage and Selection Operator),** adds **"Absolute value of magnitude"** of coefficient, as penalty term to the loss function.

Lasso shrinks the less important feature's coefficient to zero; thus, removing some feature altogether. So, this works well for feature selection in case we have a huge number of features.

The loss function for linear regression with L1 regularization is given by:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b$$

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 + \frac{\lambda}{2} ||\mathbf{w}||_1$$

Where:

- m is the number of samples.
- y^ (i) is the predicted output for the i^{th} sample.
- $y^{(i)}$ is the actual output for the i^{th} sample.
- λ is the regularization parameter (also known as the regularization strength).
- ||W||1 is the L1 norm of the weight vector **w**

2. **L2 regularization:** one of the simplest and most common kinds of parameter norm penalty: the *L2* parameter norm penalty commonly known as ***weight decay*.** This regularization strategy drives the weights closer to the origin by adding a regularization term

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

to the objective function. *L2* regularization is also known as ***ridge regression*** **or Tikhonov regularization.**

L2 regularization adds a penalty term to the loss function proportional to the squared magnitude of the model's coefficients. It prevents the weights from becoming too large, effectively reducing the impact of less important features.

The loss function for linear regression with L2 regularization is given by:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b$$

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 + \frac{\lambda}{2} ||\mathbf{w}||_2^2$$

Where:

- m is the number of samples.
- y^ (i) is the predicted output for the i^{th} sample.
- $y^{(i)}$ is the actual output for the i^{th} sample.
- λ is the regularization parameter (also known as the regularization strength).

$||\mathbf{w}||_2^2$ is the L2 norm of the weight vector w, squared.

3. **Elastic Net regularization:** Elastic Net combines both L1 and L2 regularization to provide a balance between feature selection (sparsity) and parameter shrinkage.

In Elastic Net regularization, the cost function is a combination of L1 and L2 regularization terms. The total cost function for Elastic Net regression can be expressed as:

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}(X^{(i)}w - y^{(i)})^2 + \lambda_1\|w\|_1 + \frac{\lambda_2}{2}\|w\|_2^2$$

Where:

- **J(w)** is the cost function.
- $X^{(i)}$ is the feature vector of the i-th training example.
- $y^{(i)}$ is the target variable of the i-th training example.
- w is the weight vector.
- $\|w\|_1$ is the L1 norm of the weight vector w.
- $\|w\|_2^2$ is the L2 norm of the weight vector w.
- λ1 and λ2 are the regularization parameters for L1 and L2 regularization, respectively.

Norm Penalties as Constrained :

Norm penalties as constraints refer to a method used in optimization problems, particularly in machine learning and statistics, where regularization terms (norm penalties) are used to enforce constraints on the parameters of a model. These penalties help to control the complexity of the model, preventing overfitting by restricting the size of the model parameters.

Norm Penalties:

- **L1 Norm (Lasso):** Imposes a penalty proportional to the absolute values of the parameters. This often results in sparse models (many parameters become zero).
- **L2 Norm (Ridge):** Imposes a penalty proportional to the square of the parameters, leading to smaller but non-zero parameter values.

Constrained Optimization:

- Norm penalties can be viewed as adding constraints to an optimization problem, where the goal is to minimize the loss function subject to a constraint on the norm of the parameters.

For example, minimizing a loss function L(θ) subject to $\|\theta\|^2 \leq c$ (where c is a constant) can be equivalently solved by adding an **L2 norm penalty** $\lambda\|\theta\|^2_2$ to the loss function.

Regularization and Under- Constrained Problems

In some machine learning tasks, like linear regression or Principal Component Analysis (PCA), we need to work with a matrix (think of it as a table of numbers) that represents our data. To solve certain problems, we need to "invert" this matrix, which is a bit like flipping it inside out. However, this only works if the matrix is not meaning"singular"— it's well-behaved and doesn't have any weird issues like having rows or columns that are too similar.

These linear problems have closed form solutions when the relevant matrix is invertible. It is also possible for a problem with no closed form solution to be under-

determined. An example is logistic regression applied to a problem where the classes are linearly separable. If a weight vector $\boldsymbol{w}$ is able to achieve perfect classification, then $\boldsymbol{w}$ will also achieve perfect classification and higher likelihood. An iterative optimization procedure like stochastic gradient descent will continually increase the magnitude of $\boldsymbol{w}$ and, in theory, will never halt. In practice, a numerical implementation of gradient descent will eventually reach sufficiently large weights to cause numerical overflow, at which point its behavior will depend on how the programmer has decided to handle values that are not real numbers.

Most forms of regularization are able to guarantee the convergence of iterative methods applied to under-determined problems. **For example,** weight decay will cause gradient descent to quit increasing the magnitude of the weights when the slope of the likelihood is equal to the weight decay coefficient.

The idea of using regularization to solve under-determined problems extends beyond machine learning.

Dataset Augmentation

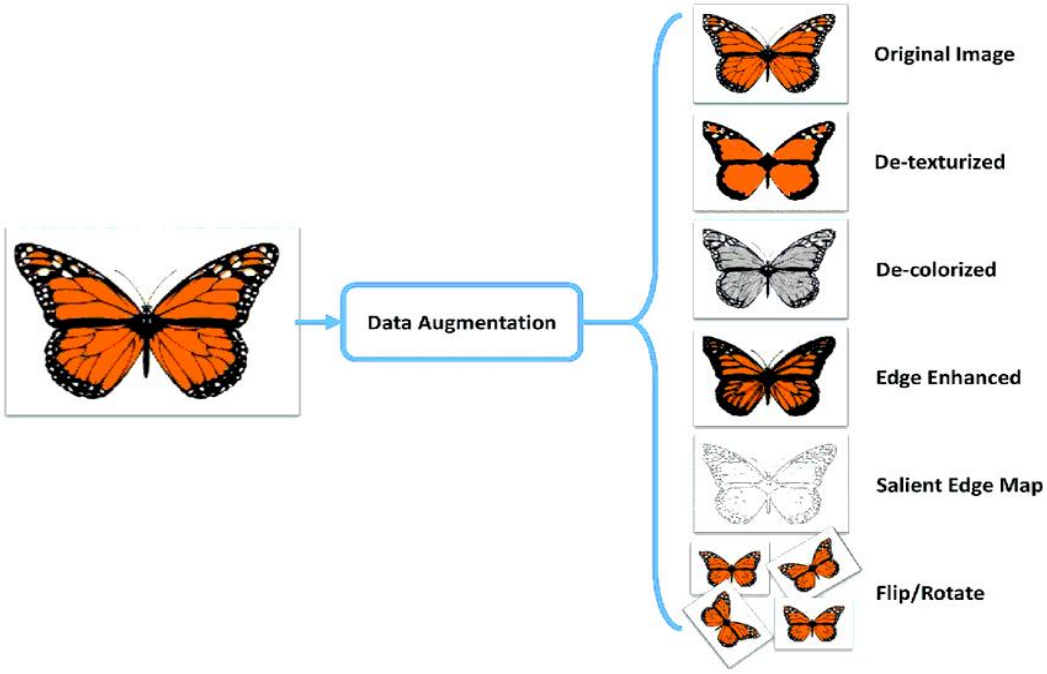
The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. For some machine learning tasks, it is reasonably straightforward to create new fake data.

This approach is easiest for classification. A classifier needs to take a complicated, high dimensional input $\boldsymbol{x}$ and summarize it with a single category identity y. This means that the main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new $(\boldsymbol{x}, y)$ pairs easily just by transforming the $\boldsymbol{x}$ inputs in our training set.

Dataset augmentation has been a particularly effective technique for a specific classification problem: object recognition. Images are high dimensional and include an enormous variety of factors of variation, many of which can be easily simulated.

Operations like translating the training images a few pixels in each direction can often greatly improve generalization, even if the model has already been designed to be partially translation invariant by using the convolution and pooling techniques. Many other operations such as rotating the image or scaling the image have also proven quite effective.

In Keras this can be done via : **keras.preprocessing.image.ImageDataGenerator class.**



Noise Robustness

The use of noise applied to the inputs as a dataset augmentation strategy. For some models, the addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights.

Another way that noise has been used in the service of regularizing models is by adding it to the weights. This technique has been used primarily in the context of recurrent neural networks. This can be interpreted as a stochastic implementation of a Bayesian inference over the weights. Adding noise to the weights is a practical, stochastic way to reflect this uncertainty.

This can also be interpreted as equivalent (under some assumptions) to a more traditional form of regularization. Adding noise to the weights has been shown to be an effective regularization strategy in the context of Recurrent Neural Networks.

Train a function $\hat{y}(\boldsymbol{x})$ that maps a set of features $\boldsymbol{x}$ to a scalar using the least-squares cost function between the model predictions $\hat{y}(\boldsymbol{x})$ and the true values y:

$$J = \mathbb{E}_{p(x,y)} \left[(\hat{y}(\boldsymbol{x}) - y)^2 \right]. \tag{7.30}$$

The training set consists of m labeled examples $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

We now assume that with each input presentation we also include a random perturbation $\boldsymbol{W} \sim N(\mathrm{E}; \mathbf{0}, \eta \boldsymbol{I})$ of the network weights.

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

This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output. In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions.

Semi-Supervised Learning

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

In the context of deep learning, semi-supervised learning usually refers to learning a representation $\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x})$. The goal is to learn a representation so that examples from the same class have similar representations. Unsupervised learning can provide useful cues for how to group examples in representation space. Examples that cluster tightly in the input space should be mapped to similar representations. A linear classifier in the new space may achieve better generalization in many cases. A long-standing variant of this approach is the application of principal components analysis as a pre-processing step before applying a classifier (on the projected data).

Instead of having separate unsupervised and supervised components in the model, one can construct models in which a generative model of either $P(\mathbf{x})$ or $P(\mathbf{x}, \mathbf{y})$ shares parameters with a discriminative model of $P(\mathbf{y} \mid \mathbf{x})$.

Instead of having separate unsupervised and supervised components in the model, one can construct models in which a generative model of either $P(\mathbf{x})$ or $P(\mathbf{x}, \mathbf{y})$ shares parameters with a discriminative model of $P(\mathbf{y} \mid \mathbf{x})$.

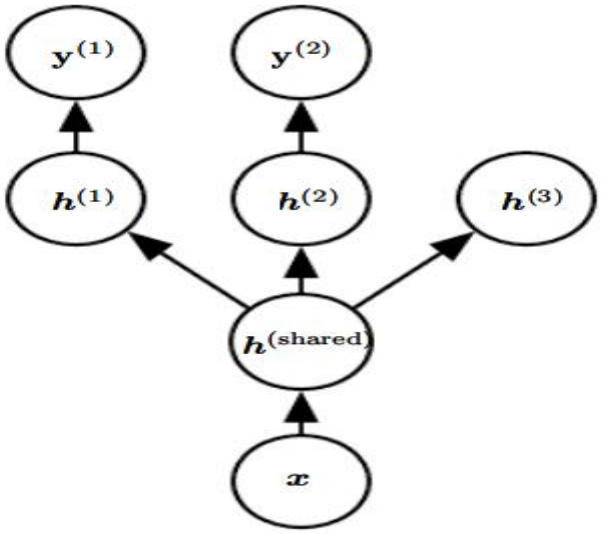
Multi- Task Learning

Multi-task learning is a way to improve generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks. In the same way that additional training examples put more pressure on the parameters of the model towards values that generalize well, when part of a model is shared across tasks, that part of the model is more constrained towards good values (assuming the sharing is justified), often yielding better generalization.

A very common form of multi-task learning, in which different supervised tasks (predicting $\mathbf{y}^{(i)}$ given $\mathbf{x}$) share the same input $\mathbf{x}$, as well as some intermediate-level representation $\boldsymbol{h}^{(\text{shared})}$ capturing a common pool of factors. 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.



Parameter Tying and Parameter Sharing

A common type of dependency that we often want to express is that certain parameters should be close to one another. Consider the following scenario: we have two models performing the same classification task (with the same set of classes) but with somewhat different input distributions. Formally, we have model A with parameters $\boldsymbol{w}^{(A)}$ and model B with parameters $\boldsymbol{w}^{(B)}$. The two models map the input to two different, but related outputs: $\hat{y}^{(A)} = f(\boldsymbol{w}^{(A)}, \boldsymbol{x})$ and $\hat{y}^{(B)} = g(\boldsymbol{w}^{(B)}, \boldsymbol{x})$.

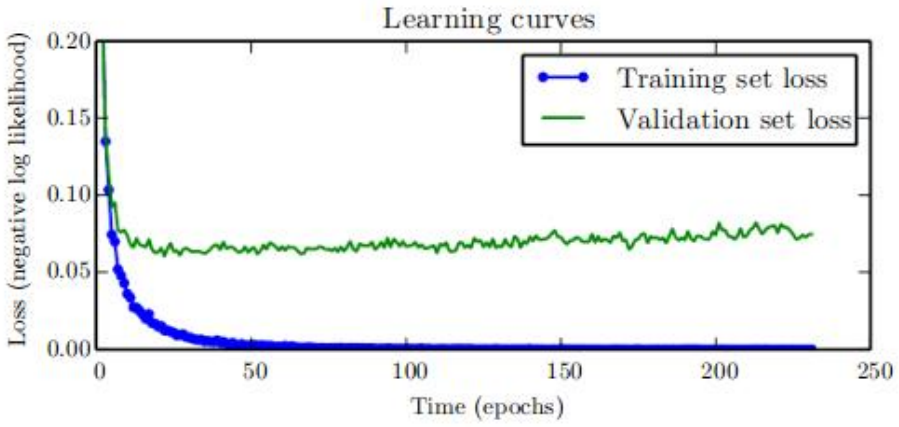
Let us imagine that the tasks are similar enough (perhaps with similar input and output distributions) that we believe the model parameters should be close to each other: $\forall i$, $w_i^{(A)}$ should be close to $w_i^{(B)}$. We can leverage this information through regularization. Specifically, we can use a parameter norm penalty of the form: $\Omega(\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)}) = ||\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}||^2_2$. Here we used an L^2 penalty, but other choices are also possible.

This kind of approach was proposed, who regularized the parameters of one model, trained as a classifier in a supervised paradigm, to be close to the parameters of another model, trained in an unsupervised paradigm. The architectures were constructed such that many of the parameters in the classifier model could be paired to corresponding parameters in the unsupervised model.

While a parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: **to force sets of parameters to be equal**. This method of regularization is often referred to as ***parameter sharing***, where we interpret the various models or model components as sharing a unique set of parameters.

Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again.



This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Instead of running our optimization algorithm until we reach a (local) minimum of validation error, we run it until the error on the validation set has not improved for some amount of time.

Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters.

Early Stopping Algorithm:

- The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.
- A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.
- Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Example : A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

1. Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
2. Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into ($\boldsymbol{X}^{(\text{subtrain})}$, $\boldsymbol{X}^{(\text{valid})}$) and ($\boldsymbol{y}^{(\text{subtrain})}$, $\boldsymbol{y}^{(\text{valid})}$) respectively.
3. Run early stopping starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This
4. returns i^* , the optimal number of steps.
5. Set $\boldsymbol{\theta}$ to random values again.
6. Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for i^* steps.

Early stopping requires a validation set, which means some training data is not fed to the model. To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed.

Early stopping is also useful because it reduces the computational cost of the training procedure.

Early stopping therefore has the advantage over weight decay that early stopping automatically determines the correct amount of regularization while weight decay requires many training experiments with different values of its hyperparameter.

Sparse Representations

Weight decay acts by placing a penalty directly on the model parameters. Another strategy is to place a penalty on the activations of the units in a neural network, encouraging their activations to be sparse. This indirectly imposes a complicated penalty on the model parameters.

Representational sparsity, on the other hand, describes a representation where many of the elements of the representation are zero (or close to zero). A simplified view of this distinction can be illustrated in the context of linear regression:

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ $\boldsymbol{x} \in \mathbb{R}^n$

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\boldsymbol{B} \in \mathbb{R}^{m \times n}$ $\boldsymbol{h} \in \mathbb{R}^n$

In the first expression, we have an example of a sparsely parametrized linear regression model. In the second, we have linear regression with a sparse representation $\boldsymbol{h}$ of the data $\boldsymbol{x}$. That is, $\boldsymbol{h}$ is a function of $\boldsymbol{x}$ that, in some sense, represents the information present in $\boldsymbol{x}$, but does so with a sparse vector.

Bagging and Other Ensemble Methods

Bagging (short for *bootstrap aggregating*) is a 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. This is an example of a general strategy in machine learning called *model averaging*. Techniques employing this strategy are known as *ensemble methods*.

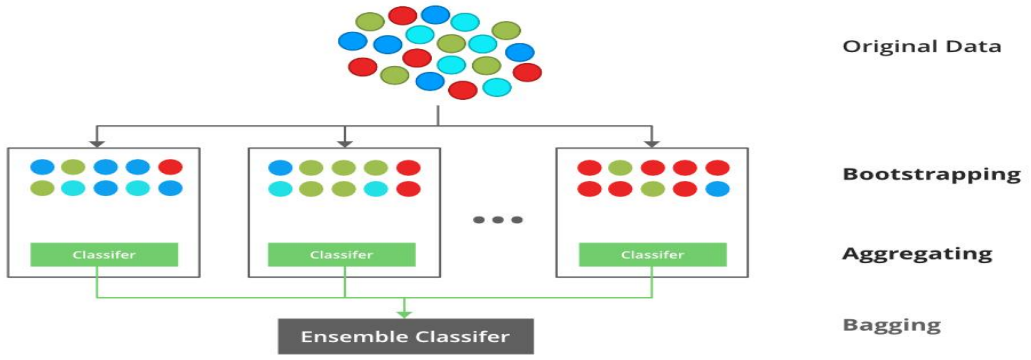
Consider for example a set of k regression models. Suppose that each model makes an error $\in$i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $E[\in^2_i] = v$ and covariances $E[\in_i \in_j] = c$. Then the error made by the average prediction of all the ensemble models is $1/k\ Sum(\in_l)$. The expected squared error of the ensemble predictor is

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right]$$
$$= \frac{1}{k}v + \frac{k-1}{k}c.$$

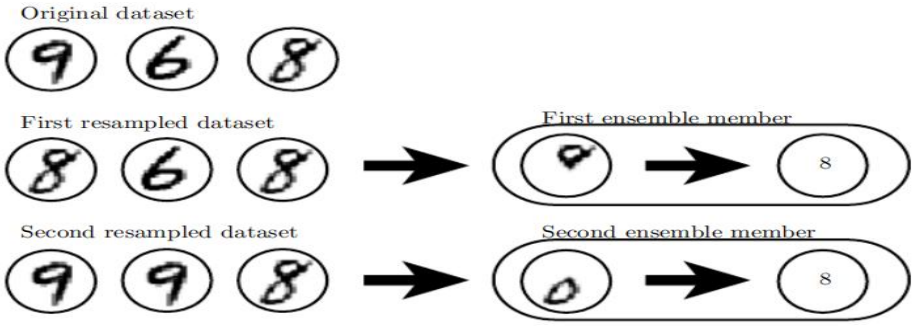
- In the case where the errors are perfectly correlated and $c = v$, the mean squared error reduces to v, so the model averaging does not help at all.
- In the case where the errors are perfectly uncorrelated and $c = 0$, the expected squared error of the ensemble is only $1\ k_v$.
- This means that the expected squared error of the ensemble decreases linearly with the ensemble size.

Implementation Steps of Bagging

Step 1: Multiple subsets are created from the original data set with equal tuples, selecting observations with replacement.
Step 2: A base model is created on each of these subsets.
Step 3: Each model is learned in parallel with each training set and independent of each other.
Step 4: The final predictions are determined by combining the predictions from all the models.



Example :



Example : 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.

Bagging involves constructing *k* different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset. This means that, with high probability, each dataset is missing some of the examples from the original dataset and also contains several duplicate examples.

Other Ensemble Methods : A technique called ***boosting*** constructs an ensemble with higher capacity than the individual models. Boosting has been applied to build ensembles of neural networks by incrementally adding neural networks to the ensemble. Boosting has also been applied interpreting an individual neural network as an ensemble, incrementally adding hidden units to the neural network.

Dropout

Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making **bagging** practical for ensembles of very many large neural networks.

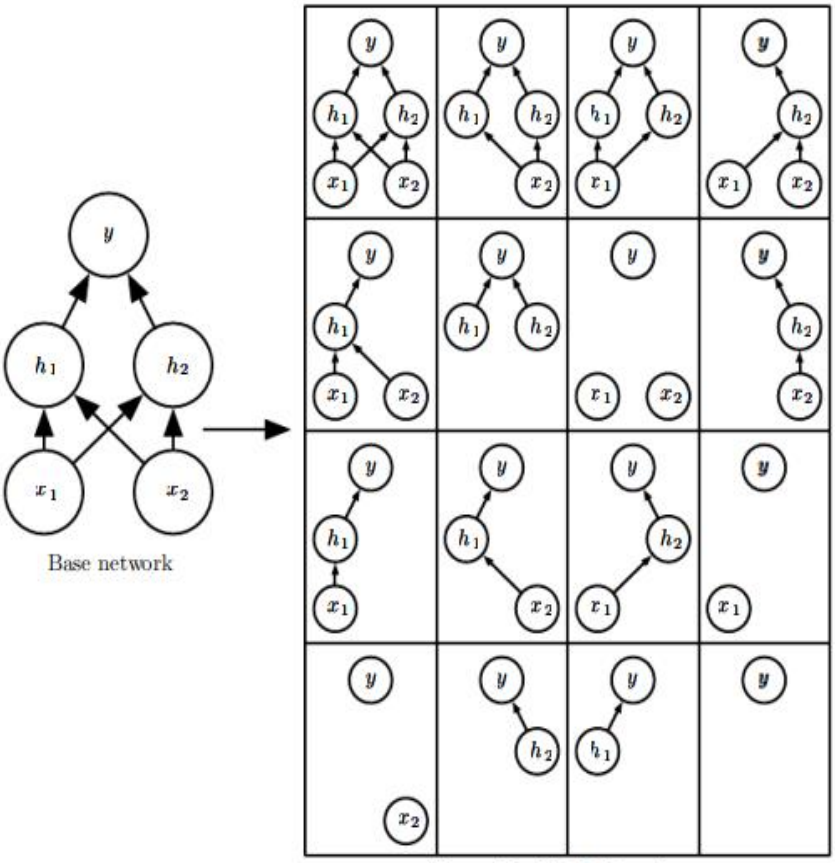
Bagging involves training multiple models, and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory.

Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero.

Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent. In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network.

In the case of dropout, typically most models are not explicitly trained at all—usually, the model is large enough that it would be infeasible to sample all possible sub-networks within the lifetime of the universe. Instead, a tiny fraction of the possible sub-networks are each trained for a single step, and the parameter sharing causes the remaining sub-networks to arrive at good settings of the parameters.



The prediction of the ensemble is given by the arithmetic mean of all of these distributions, each model *I* produces a probability distribution $p^{(i)}(y \mid \boldsymbol{x})$

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y \mid \boldsymbol{x}).$$

In the case of dropout, each sub-model defined by mask vector $\boldsymbol{\mu}$ defines a probability distribution $p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$. The arithmetic mean over all masks is given by

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$$

where $p(\boldsymbol{\mu})$ is the probability distribution that was used to sample $\boldsymbol{\mu}$ at training time.

Adversarial Training

Adversarial Training is a technique that has been developed to protect Machine Learning models from Adversarial Examples. Let's briefly recall what Adversarial Examples are. These are inputs that are very slightly and cleverly perturbed (such as an image, text, or sound) in a way that is imperceptible to humans but will be misclassified by a machine learning model.

In many cases, $\boldsymbol{x'}$ can be so similar to $\boldsymbol{x}$ that a human observer cannot tell the difference between the original example and the ***adversarial example***, but the network can make highly different predictions.

Adversarial examples have many implications, for example, in computer security, that are beyond the scope of this chapter. However, they are interesting in the context of regularization because one can reduce the error rate on the original i.i.d. test set via *adversarial training*—training on adversarially perturbed examples from the training set.

Adversarial training: This is a brute force solution where we simply generate a lot of adversarial examples and explicitly train the model not to be fooled by each of them.

Example of Adversarial Example :

A demonstration of adversarial example generation applied to GoogLeNet on ImageNet. By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet's classification of the image.

Tangent Distance

Many machine learning algorithms aim to overcome the curse of dimensionality by assuming that the data lies near a low-dimensional manifold.

One of the early attempts to take advantage of the manifold hypothesis is the ***tangent distance* algorithm.** It is a non-parametric nearest-neighbor algorithm in which the metric used is not the generic Euclidean distance but one that is derived from knowledge of the manifolds near which probability concentrates.

Tangent distance refers to the shortest distance between a point and a curve (or a surface) along a line that is tangent to the curve at a specific point.

Formula:

For a curve f(x), the equation of the tangent line at a point $(x_0,f(x_0))$ can be expressed as:

$$\mathbf{y-f(x_0)=f'(x_0)(x-x_0)}$$

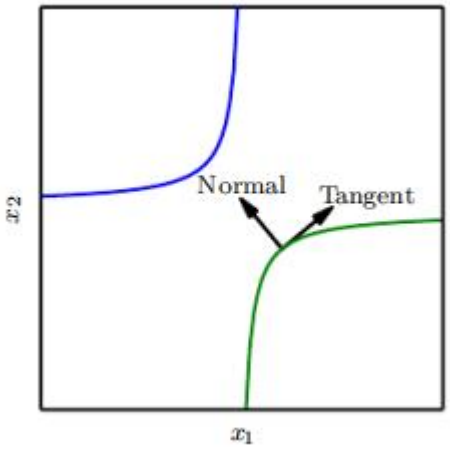
Here:

- $(x_0,f(x_0))$ is the point of tangency on the curve.
- $f'(x_0)$ is the derivative of the curve at x_0, which gives the slope of the tangent line.

Tangent Prop and Manifold Tangent Classifier

The ***tangent prop*** **algorithm** trains a neural net classifier with an extra penalty to make each output $f(\boldsymbol{x})$ of the neural net locally invariant to known factors of variation. These factors of variation correspond to movement along the manifold near which examples of the same class concentrate. Localinvariance is achieved by requiring $\nabla_x f(\boldsymbol{x})$ to be orthogonal to the known manifold tangent vectors $\boldsymbol{v}^{(i)}$ at $\boldsymbol{x}$, or equivalently that the directional derivative of f at $\boldsymbol{x}$ in the directions $\boldsymbol{v}^{(i)}$ be small by adding a regularization penalty Ω:

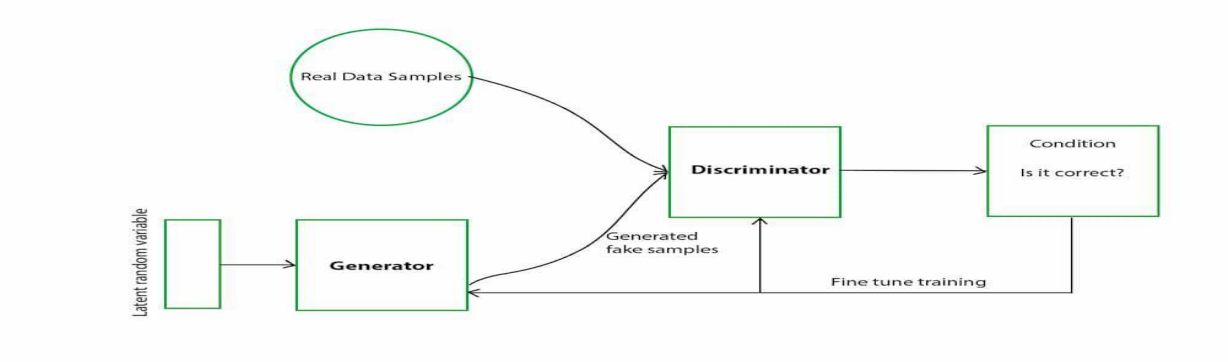
$$\Omega(f) = \sum_i \left((\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^\top \boldsymbol{v}^{(i)} \right)^2.$$



- Figure Illustration of the main idea of the tangent prop algorithm and manifold tangent classifier, which both regularize the classifier output function $f(\boldsymbol{x})$.
- 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 $f(\boldsymbol{x})$ to not change very much as $\boldsymbol{x}$ 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.

Generative Adversarial Neural Networks : Generative Adversarial Networks (GANs) are a powerful class of neural networks that are used for an unsupervised learning. GANs are made up of two neural networks, **a discriminator and a generator.** They use adversarial training to produce artificial data that is identical to actual data.

- The Generator attempts to fool the Discriminator, which is tasked with accurately distinguishing between produced and genuine data, by producing random noise samples.
- Realistic, high-quality samples are produced as a result of this competitive interaction, which drives both networks toward advancement.
- GANs are proving to be highly versatile artificial intelligence tools, as evidenced by their extensive use in image synthesis, style transfer, and text-to-image synthesis.
- They have also revolutionized generative modeling.



Optimization for Training Deep Models

Deep learning algorithms involve optimization in many contexts. For example, performing inference in models such as PCA involves solving an optimization problem.

Of all of the many optimization problems involved in deep learning, the most difficult is neural network training. It is quite common to invest days to months of time on hundreds of machines in order to solve even a single instance of the neural network training problem. Because this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it.

Learning vs Pure Optimization

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable.

We therefore optimize P only indirectly. We reduce a different cost function $J(\boldsymbol{\theta})$ in the hope that doing so will improve P. This is in contrast to pure optimization, where minimizing J is a goal in and of itself.

Typically, the cost function can be written as an average over the training set, such as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y),$$

where L is the per-example loss function, $f(\boldsymbol{x}; \boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$, $\hat{p}$data is the empirical distribution.

We would usually prefer to minimize the corresponding objective function where the expectation is taken across **the data generating distribution** $p$data rather than just over the finite training set:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y).$$

1. **Empirical Risk Minimization**

The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\boldsymbol{x}, y)$ with the empirical distribution $\hat{p}(\boldsymbol{x}, y)$ defined by the training set. We now minimize the *empirical risk*

$$\mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)})$$

where m is the number of training examples. The training process based on minimizing this average training error is known as *empirical risk minimization*.

2. **Surrogate Loss Functions and Early Stopping**

one typically optimizes a *surrogate loss function* instead, which acts as a proxy but has advantages. For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss. The negative log-likelihood allows the model to estimate the conditional probability of the classes, given the input, and if the model can do that well, then it can pick the classes that yield the least classification error in expectation.

Typically the early stopping criterion is based on the true underlying loss function, such as 0-1 loss measured on a validation set, and is designed to cause the algorithm to halt whenever overfitting begins to occur. Training often halts while the surrogate loss function still has large derivatives, which is very different from the pure optimization setting, where an optimization algorithm is considered to have converged when the gradient becomes very small.

3. **Batch and Mini-batch Algorithms**

Optimization algorithms that use the entire training set are called *batch* or *deterministic* gradient methods, because they process all of the training examples simultaneously in a large batch.

Most algorithms used for deep learning fall somewhere in between, using more than one but less than all of the training examples. These were traditionally called *minibatch* or *minibatch stochastic* methods and it is now common to simply call them *stochastic* methods.

Challenges in Neural Network Optimization

The most prominent challenges involved in optimization
for training deep models.

1. **Ill-Conditioning**

The ill-conditioning problem is generally believed to be present in neural network training problems. Ill-conditioning can manifest by causing SGD to get “stuck” in the sense that even very small steps increase the cost function.

2. **Local Minima**

One of the most prominent features of a convex optimization problem is that it can be reduced to the problem of finding a local minimum. Any local minimum is guaranteed to be a global minimum.

Some convex functions have a flat region at the bottom rather than a single global minimum point, but any point within such a flat region is an acceptable solution.

With non-convex functions, such as neural nets, it is possible to have many local minima. Indeed, nearly any deep model is essentially guaranteed to have an extremely large number of local minima.

Local minima can be problematic if they have high cost in comparison to the global minimum. One can construct small neural networks, even without hidden units, that have local minima with higher cost than the global minimum. If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms.

3. **Plateaus, Saddle Points and Other Flat Regions**

For many high-dimensional non-convex functions, local minima (and maxima) are in fact rare compared to another kind of point with zero gradient: a saddle point. Some points around a saddle point have greater cost than the saddle point,
while others have a lower cost.

There may also be wide, flat regions of constant value. In these locations, the gradient are all zero. Such degenerate locations pose major problems for all numerical optimization algorithms. In a convex problem, a wide, flat region must consist entirely of global minima, but in a general optimization problem, such a region could correspond to a high value of the objective function.

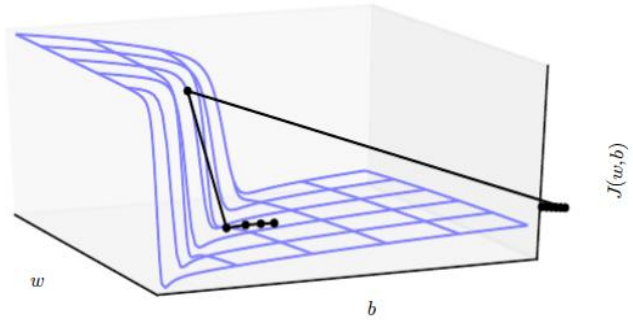
4. **Cliffs and Exploding Gradients**

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. The cliff can be dangerous whether we approach it from above or from below, but fortunately its most serious consequences can be avoided using the *gradient clipping* heuristic.

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. Which construct very deep computational graphs by repeatedly applying the same operation at each time step of a long temporal sequence. Repeated application of the same parameters gives rise to especially pronounced difficulties.

Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.



Basic Algorithms

1. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variant of the **Gradient Descent** algorithm that is used for optimizing machine learning models. It addresses the computational

inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.

Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to fitting linear classifiers and regressors under convex loss functions such as (linear) Support Vector Machines and Logistic Regression.

In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, hence the term "stochastic" in stochastic Gradient Descent.

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

where θ are the model parameters, α is the learning rate, and $J(\theta)$ is the loss function.

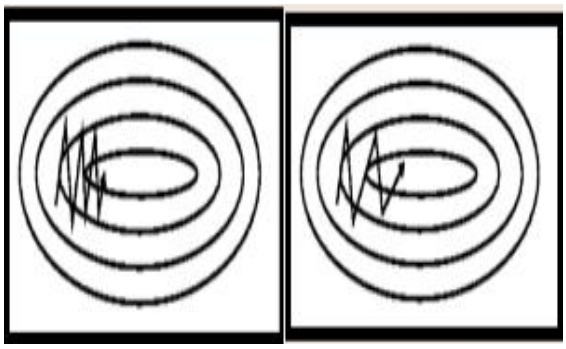
Stochastic Gradient Descent Algorithm

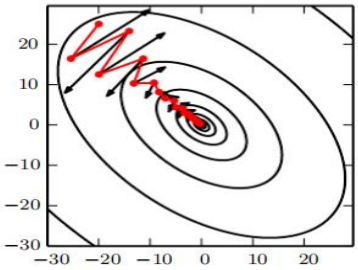
- **Initialization**: Randomly initialize the parameters of the model.
- **Set Parameters**: Determine the number of iterations and the learning rate (alpha) for updating the parameters.
- **Stochastic Gradient Descent Loop**: Repeat the following steps until the model converges or reaches the maximum number of iterations:
    1. Shuffle the training dataset to introduce randomness.
    2. Iterate over each training example (or a small batch) in the shuffled order.
    3. Compute the gradient of the cost function with respect to the model parameters using the current training example (or batch).
    4. Update the model parameters by taking a step in the direction of the negative gradient, scaled the learning rate.
    5. Evaluate the convergence criteria, such as the difference in the cost function between iterations of the gradient.
- **Return Optimized Parameters**: Once the convergence criteria are met or the maximum number of iterations is reached, return the optimized model parameters.

In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm.

2. Momentum

Momentum is an enhancement to SGD that helps accelerate convergence, especially in the presence of high curvature or noisy gradients. What the momentum does is helps in faster convergence of the loss function.adding a fraction of the previous update to the current update will make the process a bit faster. One thing that should be remembered while using this algorithm is that the learning rate should be decreased with a high momentum term.





It introduces a momentum term β that accumulates the gradient updates over time.
The update rule for momentum is:

$$v_{t+1} = \beta v_t + (1-\beta)\nabla J(\theta_t)$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

where v_t is the momentum term and β is the momentum parameter.

Parameter Initialization Strategies

- Training algorithms for deep learning models are usually iterative in nature and thus require the user to specify some initial point from which to begin the iterations.
- Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization.
- The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.
- When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.
- Modern initialization strategies are simple and heuristic.
- Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood.
- Most initialization strategies are based on achieving some nice properties when the network is initialized.
- Perhaps the only property known with complete certainty is that the initial parameters need to “break symmetry” between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.
- We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution.
- Larger initial weights will yield a stronger symmetry breaking effect, helping to avoid redundant units. They also help to avoid losing signal during forward or back-propagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication. Initial weights that are too large may, however, result in exploding values during forward propagation or back-propagation.

Algorithms with Adaptive Learning Rates

Neural network researchers have long realized that the learning rate was reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance.
More recently, a number of incremental (or mini-batch-based) methods have been introduced that adapt the learning rates of model parameters.

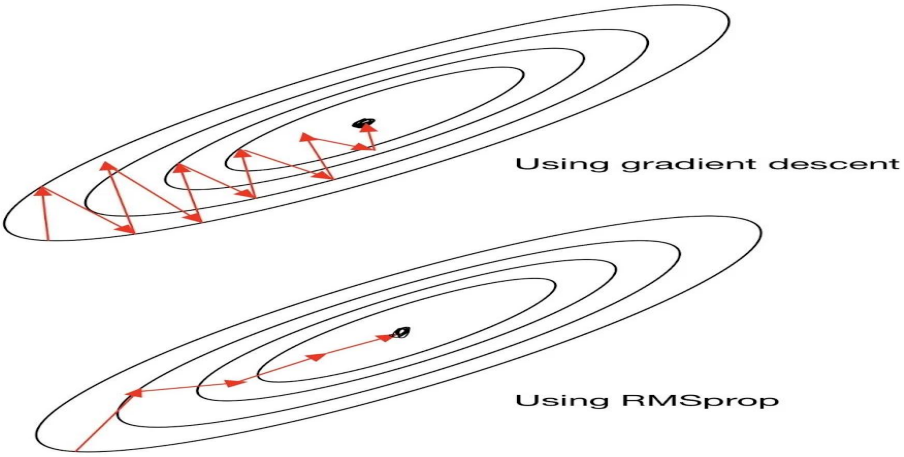
1. **AdaGrad :** The AdaGrad algorithm, individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values.
The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.

Algorithm : 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

2. **RMSProp (Root Mean Square Propagation):** RMS prop is one of the popular optimizers among deep learning enthusiasts. It resolves the problem of varying gradients.The problem with the gradients is that some of them were small while others may be huge. So, defining a single learning rate might not be the best idea.

Adapting the step size individually for each weight. In this algorithm, the two gradients are first compared for signs. If they have the same sign, we're going in the right direction, increasing the step size by a small fraction. If they have opposite signs, we must decrease the step size. Then we limit the step size and can now go for the weight update.



Algorithm 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

3. **Adam (Adaptive Moment Estimation optimizer):** Adam optimizer, short for Adaptive Moment Estimation optimizer, is an optimization algorithm commonly used in deep learning. It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training.
Its ability to adaptively adjust the learning rate for each network weight individually. Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments.
By incorporating both the first moment (mean) and second moment (uncentered variance) of the gradients, Adam optimizer achieves an adaptive learning rate that can efficiently navigate the optimization landscape during training. This

adaptivity helps in faster convergence and improved performance of the neural network.

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 $\boldsymbol{s} = \boldsymbol{0}, \boldsymbol{r} = \boldsymbol{0}$
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