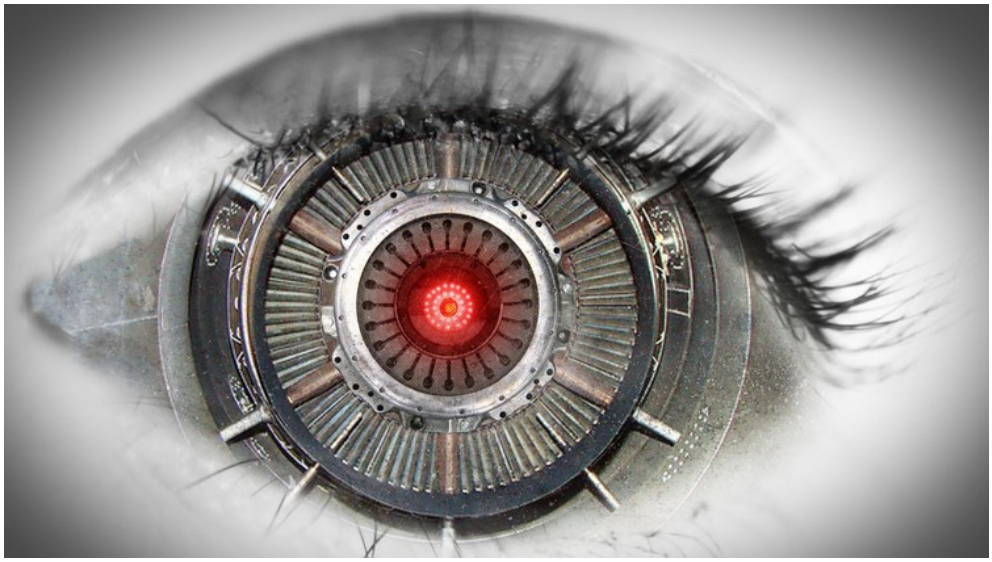
Deep Learning and Computer Vision

Shubham Lohiya
Roll no. 18D100020

June 2019



Introduction

Computer vision is the science and technology of machines that see. As a scientific discipline, computer vision is concerned with the theory and technology for building artificial systems that obtain information from images or multi-dimensional data.
The problem of computer vision appears simple because it is trivially solved by people, even very young children. Nevertheless, it largely remains an unsolved problem based both on the limited understanding of biological vision and because of the complexity of vision perception in a dynamic and nearly infinitely varying physical world. The goal of computer vision is to understand the content of digital images. Typically, this involves developing methods that attempt to reproduce the capability of human vision.
The earliest research in computer vision started way back in 1950s. Since then, we have come a long way but still find ourselves far from the ultimate objective. But with neural networks and deep learning, we have become empowered like never before. Applications of deep learning in vision have taken this technology to a different level and made possible sophisticated things like self-driven cars, auto-tagging of friends in our Facebook pictures, facial security features, gesture recognition, automatic number plate recognition, etc. This article will also introduce you to Convolution Neural Networks which form the crux of Deep Learning applications in computer vision.

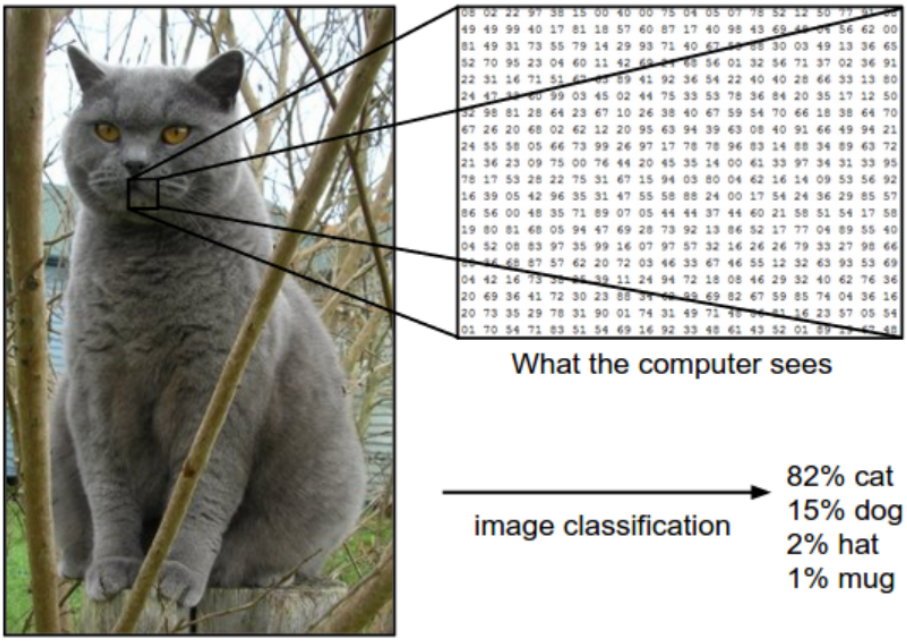
Acknowlegdements

This work was written as a part of the Summer Of Science, 2019 by the MnP Club, IIT Bombay. I have followed the course CS231n : Convolutional Neural Networks for Visual Recognition offered by Stanford. The course instructors were Fei-Fei Lee, Justin Johnson and Serena Yeung. All content here is highly inspired by this course and at several points might simply be a paraphrasing of the course content. Nevertheless, I have tried to compress the contents of the course, skipping a few of the details, while still preserving sufficient depth. Also, I would like to acknowledge the help of my mentor, Sarthak Consul, for his help and support.

Contents

1 Image Classification

This section introduces the Image Classification problem which is the task of assigning an input image one label from a fixed set of categories. This is one of the core problems in Computer Vision that, despite its simplicity, has a large variety of practical applications. Later in this article, you'll be able to see that many other seemingly distinct Computer Vision tasks (such as object detection, segmentation) can be reduced to image classification.
Here's an example -



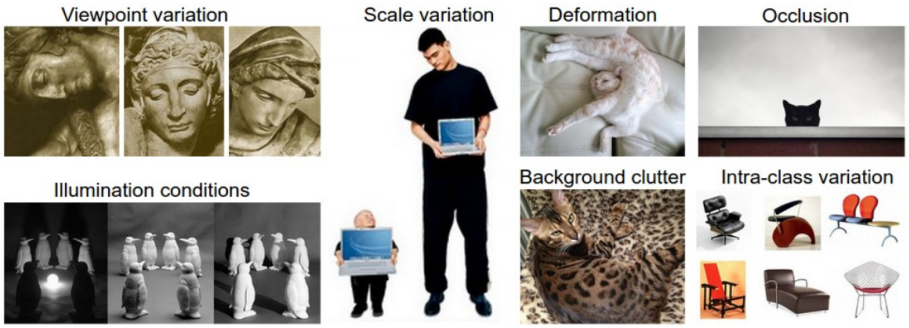
The task in Image Classification is to predict a single label (or a distribution over labels as shown here to indicate our confidence) for a given image. Images are 3-dimensional arrays of integers from 0 to 255, of size Width x Height x 3. The 3 represents the three color channels Red, Green, Blue.

1.1 Challenges in Computer Vision (CV)

Object detection is considered to be the most basic application of computer vision. Rest of the other developments in computer vision are achieved by making small enhancements on top of this. In real life, every time we(humans) open our eyes, we unconsciously detect objects. Since it is super-intuitive for us, we fail to appreciate the key challenges involved when we try to design systems similar to our eye. Lets start by looking at some of the key roadblocks:

- **Viewpoint variation :** A single instance of an object can be oriented in many ways with respect to the camera.
- **Scale variation :** Visual classes often exhibit variation in their size (size in the real world, not only in terms of their extent in the image).

- **Deformation :** Many objects of interest are not rigid bodies and can be deformed in extreme ways.
- **Occlusion :** The objects of interest can be occluded. Sometimes only a small portion of an object (as little as few pixels) could be visible.
- **Illumination conditions :** The effects of illumination are drastic on the pixel level.
- **Background clutter :** The objects of interest may blend into their environment, making them hard to identify.
- **Intra-class variation :** The classes of interest can often be relatively broad, such as chair. There are many different types of these objects, each with their own appearance.



1.2 The image classification pipeline

The image classification pipeline can be formalized as follows:

- **Input :** The input consists of a set of **N** images, each labeled with one of **K** different classes. This data is referred to as the **training set**.
- **Learning:** The task is to use the training set to learn what every one of the classes looks like. This step is referred to as **training a classifier**, or **learning a model**.
- **Evaluation:** In the end,,the quality of the classifier is evaluated by asking it to predict labels for a new set of images that it has never seen before. The true labels of these images are then compared to the ones predicted by the classifier. Intuitively, we're hoping that a lot of the predictions match up with the true answers (which we call the ground truth).

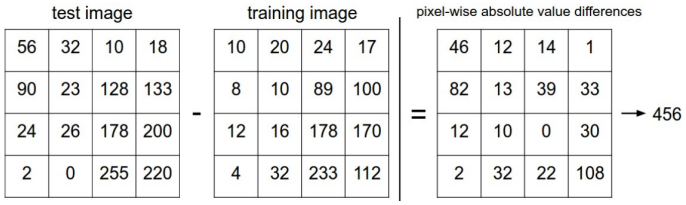
2 Nearest Neighbour Classifier

The Nearest Neighbour Classifier is a basic approach to an image classification problem.
One popular image classification dataset is the CIFAR-10 dataset. This dataset consists of 60,000 tiny images that are 32 pixels high and wide. Each image is

labeled with one of 10 classes (for example "airplane, automobile, bird, etc"). These 60,000 images are partitioned into a training set of 50,000 images and a test set of 10,000 images.
Suppose now that we are given the CIFAR-10 training set of 50,000 images (5,000 images for every one of the labels), and we wish to label the remaining 10,000. The nearest neighbor classifier will take a test image, compare it to every single one of the training images, and predict the label of the closest training image. One of the simplest possibilities is to compare the images pixel by pixel and add up all the differences. In other words, given two images and representing them as vectors I_1, I_2 , a reasonable choice for comparing them might be the L1 distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

Where the sum is taken over all pixels. Here is the procedure visualized:



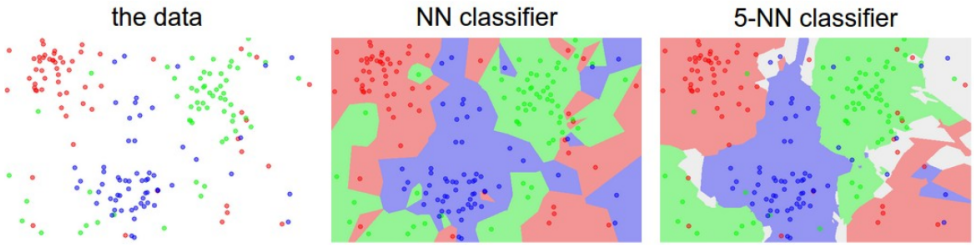
This classifier only achieves 38.6 % on CIFAR-10. That's more impressive than guessing at random (which would give 10% accuracy since there are 10 classes), but nowhere near human performance (which is estimated at about 94%) or near state-of-the-art Convolutional Neural Networks that achieve about 95%, matching human accuracy.
The choice of distance matters too. There are many other ways of computing distances between vectors. Another common choice could be to instead use the L2 distance, which has the geometric interpretation of computing the euclidean distance between two vectors. The distance takes the form:

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

If we run Nearest Neighbor classifier on CIFAR-10 with this distance, we would obtain 35.4% accuracy.

2.1 k - Nearest Neighbor Classifier

Instead of only using the label of the nearest image when you wish to make a prediction, it is almost always the case that one can do better by using what's called a **k-Nearest Neighbor Classifier**. The idea is very simple: instead of finding the single closest image in the training set, you find the top **k** closest images, and have them vote on the label of the test image. In particular, when k = 1, we recover the Nearest Neighbor classifier. Intuitively, higher values of k have a smoothing effect that makes the classifier more resistant to outliers:



2.2 Validation sets for Hyperparameter tuning

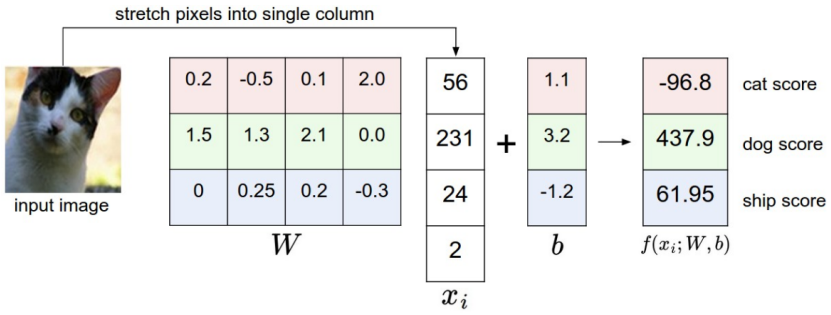
The k-nearest neighbor classifier requires a setting for k. But what number works best? Additionally, there are many different distance functions we could have used: L1 norm, L2 norm, there are many other choices weren't even considered (e.g. dot products). These choices are called hyperparameters and they come up very often in the design of many Machine Learning algorithms that learn from data. It's often not obvious what values/settings one should choose. These hyperparameters can be tuned by splitting our training set in two: a slightly smaller training set, and what we call a validation set. This validation set is essentially used as a fake test set to tune the hyper-parameters. In cases where the size of training data (and therefore also the validation data) is small, people sometimes use a more sophisticated technique for hyperparameter tuning called cross-validation.

3 Linear Classification

This is a more powerful approach to image classification that will eventually naturally extend to entire Neural Networks and Convolutional Neural Networks. The approach will have two major components: a **score function** that maps the raw data to class scores, and a **loss function** that quantifies the agreement between the predicted scores and the ground truth labels. This is then cast as an optimization problem in which the goal is to minimize the loss function with respect to the parameters of the score function. The first component of this approach is to define the score function that maps the pixel values of an image to confidence scores for each class. We start out with arguably the simplest possible function, a linear mapping:

$$f(x_i, W, b) = Wx_i + b$$

In the above equation, it is assumed that the image x_i has all of its pixels flattened out to a single column vector of shape [D x 1]. The matrix **W** (of size [K x D]), and the vector **b** (of size [K x 1]) are the **parameters** of the function. In CIFAR-10, x_i contains all pixels in the i-th image flattened into a single [3072 x 1] column, **W** is [10 x 3072] and **b** is [10 x 1], so 3072 numbers come into the function (the raw pixel values) and 10 numbers come out (the class scores). The parameters in **W** are often called the **weights**, and **b** is called the **bias vector** because it influences the output scores, but without interacting with the actual data x_i.



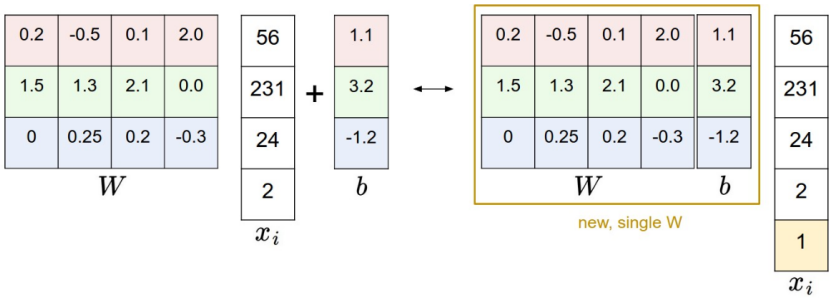
Depicted above is a simple example. It's assumed the image only has 4 pixels (4 monochrome pixels, color channels are not considered in this example for brevity), and that there are 3 classes (red (cat), green (dog), blue (ship) class).

Another interpretation for the weights **W** is that each row of **W** corresponds to a template (or sometimes also called a prototype) for one of the classes. The score of each class for an image is then obtained by comparing each template with the image using an inner product (or dot product) one by one to find the one that "fits" best. With this terminology, the linear classifier is doing template matching, where the templates are learned.

Bias trick is a common simplifying trick to representing the two parameters **W,b** as one. This is done by combining the two sets of parameters into a single matrix that holds both of them by extending the vector x_i with one additional dimension that always holds the constant **1** - *a default bias dimension*. With the extra dimension, the new score function will simplify to a single matrix multiply:

$$f(x_i, W) = Wx_i$$

With the CIFAR-10 example, x_i is now [3073 x 1] instead of [3072 x 1] - (with the extra dimension holding the constant 1), and **W** is now [10 x 3073] instead of [10 x 3072]. The extra column that **W** now corresponds to the bias **b**. Here's an illustration to help clarify:



4 Loss function

A function was defined from the pixel values to class scores, which was parameterized by a set of weights **W**. The data (x_i, y_i) cannot be controlled (it is fixed and given), but these weights can and the objective is to set them so that the predicted class scores are consistent with the ground truth labels in the training data.
For example, going back to the example image of a cat and its scores for the classes "cat", "dog" and "ship", it's seen that the particular set of weights in that example was not very good at all: The pixels that we fed in depicted a cat but the cat score came out very low (-96.8) compared to the other classes (dog score 437.9 and ship score 61.95). The unhappiness with outcomes such as this one is measured with a loss function (or sometimes also referred to as the cost function or the objective). Intuitively, the loss will be high if a poor job of classifying the training data is done.

4.1 Multiclass Support Vector Machine loss

The **Multiclass Support Vector Machine** (SVM) loss is set up so that the SVM "wants" the correct class for each image to a have a score higher than the incorrect classes by some fixed margin Δ .

Given the pixels of image x_i and the label y_i that specifies the index of the correct class, the score function takes the pixels and computes the vector $f(x_i, W)$ of class scores, which we will abbreviate to $\boldsymbol{s}$ (short for scores). For example, the score for the j-th class is the j-th element: $s_j = f(x_i, W)_j$. The Multiclass SVM loss for the i-th example is then formalized as follows:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

4.1.1 Regularization

A dataset and a set of parameters **W** that correctly classify every example (i.e. all scores are so that all the margins are met, and $L_i = 0$ for all i) can be obtained. The issue is that this set of **W** is not necessarily unique: there might be many similar W that correctly classify the examples. One easy way to see this is that if some parameters **W** correctly classify all examples (so loss is zero for each example), then any multiple of these parameters λW where $\lambda > 1$ will also give zero loss because this transformation uniformly stretches all score magnitudes and hence also their absolute differences.
To remove this ambiguity we should encode some preference for a certain set of weights **W** over others . We can do so by extending the loss function with a **regularization penalty** ***R(W)***. The most common regularization penalty is the **L2** norm that discourages large weights through an elementwise quadratic

penalty over all parameters:

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

That is, the full Multiclass SVM loss becomes:

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

Where N is the number of training examples. As you can see, we append the regularization penalty to the loss objective, weighted by a hyperparameter λ. There is no simple way of setting this hyperparameter and it is usually determined by cross-validation.

4.2 Softmax classifier

The Softmax classifier gives a slightly more intuitive output (normalized class probabilities) and also has a probabilistic interpretation . In the Softmax classifier, the function mapping $f(x_i; W) = Wx_i$ stays unchanged, but we now interpret these scores as the unnormalized log probabilities for each class and replace the hinge loss with a cross-entropy loss that has the form:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or equivalently} \qquad L_i = -f_{y_i} + \log \sum_j e^{f_j}$$

where we are using the notation f_j to mean the j-th element of the vector of class scores f. As before, the full loss for the dataset is the mean of L_i over all training examples together with a regularization term R(W). The function $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$ is called the **softmax function**: It takes a vector of arbitrary real-valued scores (in z) and squashes it to a vector of values between zero and one that sum to one.

Probabilistic Interpretation : Looking at the expression, we see that :

$$P(y_i \mid x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

can be interpreted as the (normalized) probability assigned to the correct label y_i given the image x_i and parameterized by W.

5 Optimization

Optimization is the process of finding the set of parameters W that minimize the loss function.

The strategy is to start with random weights and iteratively refine them over time to get lower loss. The best direction along which the weight vector should be changed so that is mathematically guaranteed to be the direction of the steepest descent (at least in the limit as the step size goes towards zero) can be computed. This direction will be related to the **gradient** of the loss function.
The procedure of Gradient Descent is followed which is repeatedly evaluating the gradient and then performing a parameter update. The gradient computes the direction in which the function has the steepest rate of increase, but it does not report the step size (for parameter change) in the negative of that direction. The parameter update requires a tricky setting of the step size (or the learning rate) that must be set just right: if it is too low the progress is steady but slow. If it is too high the progress can be faster, but more risky. Choosing the step size is done taking a lot of factors into account and is one of the most important (and most headache-inducing) hyperparameter settings in training a neural network.

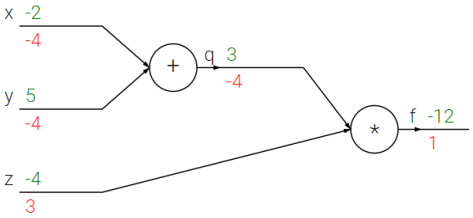
Mini-batch gradient descent: The training data can have millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data. For example, in current state of the art ConvNets, a typical batch contains 256 examples from the entire training set of 1.2 million. This batch is then used to perform a parameter update.

The SVM cost function has a bowl-shaped appearance and is an example of a convex function. Score functions f in Neural Networks are non-convex, and the visualizations do not feature bowls but complex, bumpy terrains. Hence, the gradient descent might sometimes end up in a local minima and give undesirable results.

6 Backpropagation

Backpropagation is a way of computing gradients of expressions through recursive application of chain rule. Understanding of this process and its subtleties is critical to understand, and effectively develop, design and debug Neural Networks.
Given some function $f(x)$ where x is a vector of inputs, the gradient of f at x (i.e. $\nabla f(x)$) is to be computed.
In the specific case of Neural Networks, f will correspond to the loss function (L) and the inputs x will consist of the training data and the neural network weights. For example, the loss could be the SVM loss function and the inputs are both the training data (x_i, y_i), $i = 1 \ldots N$ and the weights and biases W, b. The training data is considered given and fixed, and the weights as variables we have control over.
Here's how backpropagation works with an example:
$f(x, y, z) = (x+y)z$. This expression can be broken down into two expressions: $q = x + y$ and $f = qz$. Moreover, the derivatives of both expressions can be computed separately. f is just multiplication of q and z, so $\frac{\partial f}{\partial q} = z$, $\frac{\partial f}{\partial z} = q$, and

q is addition of x and y so $\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$. The gradient on the intermediate value q - the value of $\frac{\partial f}{\partial q}$ is not useful. Instead, the gradient of f with respect to its inputs x, y, z is desired. The chain rule allows us to "chain" these gradient expressions together through multiplication. For example, $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$.



The real-valued *"circuit"* on left shows the visual representation of the computation. The **forward pass** computes values from inputs to output (shown in green). The **backward pass** then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the way to the inputs of the circuit. The gradients can be thought of as flowing backwards through the circuit.

Backpropagation is a beautifully local process. Every gate in a circuit diagram gets some inputs and can right away compute two things: 1. its output value and 2. the local gradient of its inputs with respect to its output value. The gates can do this completely independently without being aware of any of the details of the full circuit that they are embedded in. However, once the forward pass is over, during backpropagation the gate will eventually learn about the gradient of its output value on the final output of the entire circuit. Chain rule says that the gate should take that gradient and multiply it into every gradient it normally computes for all of its inputs. Backpropagation can thus be thought of as gates communicating to each other (through the gradient signal) whether they want their outputs to increase or decrease (and how strongly), so as to make the final output value higher.

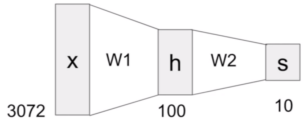
7 Neural networks - Introduction

Neural network or artificial neural network is a machine learning technique which enables a computer to learn from the observational data. Neural network in computing is inspired by the way biological nervous system process information. Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on.

Earlier, a single layer linear score function was used. Now for a Neural network in the simplest form, just stack two of these together : a linear transformation on top of another one with a non linearity in between to get a two layer Neural Network. This non linearity is important so that the two layers don't collapse into a single linear function. Here's an illustration;

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

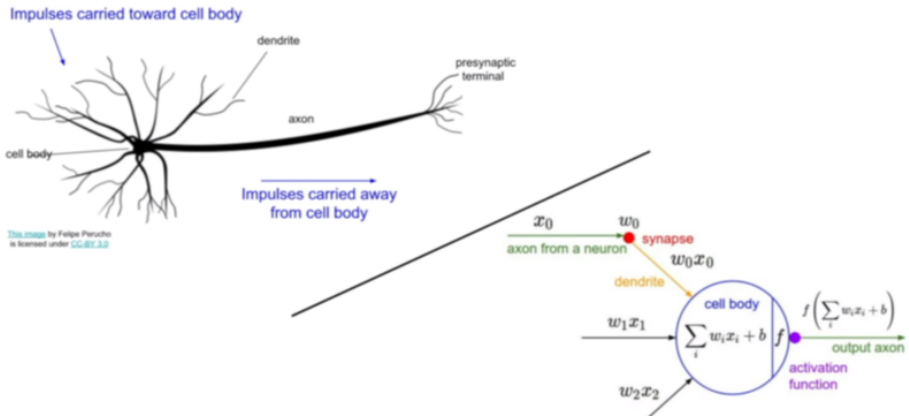


We can keep stacking more layers on top to increase the **depth** of the Neural Network.

3-layer Neural Network

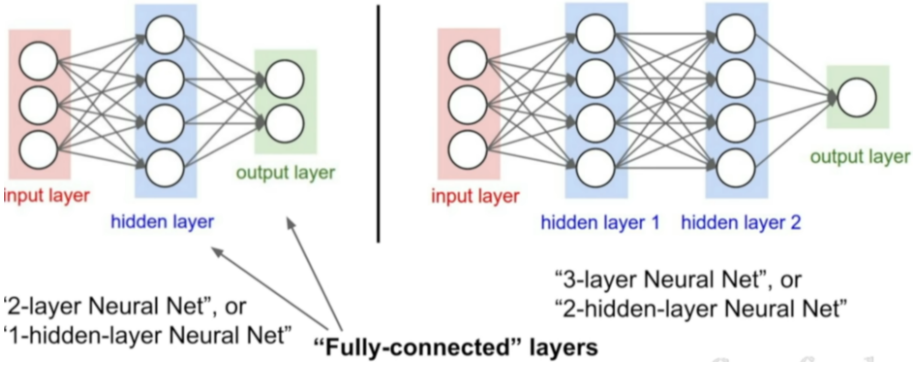
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

Taking a look at the biological analogy now :



Neural Network is divided into layers of 3 types:

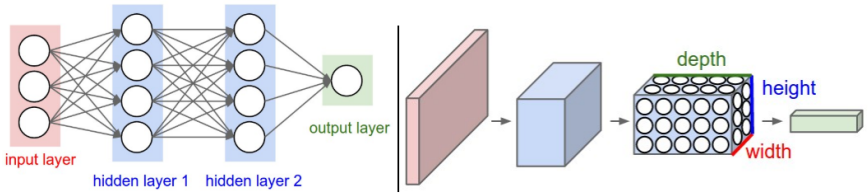
1. **Input Layer**: The training observations are fed through these neurons
2. **Hidden Layers**: These are the intermediate layers between input and output which help the Neural Network learn the complicated relationships involved in data.
3. **Output Layer**: The final output is extracted from previous two layers.



8 Convolutional Neural Networks (CNNs / ConvNets)

8.1 Architecture Overview

Regular Neural Nets don't scale well to full images. In CIFAR-10, images are only of size 32x32x3, so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights.
Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. (the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network.)
Here is a visualization:



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

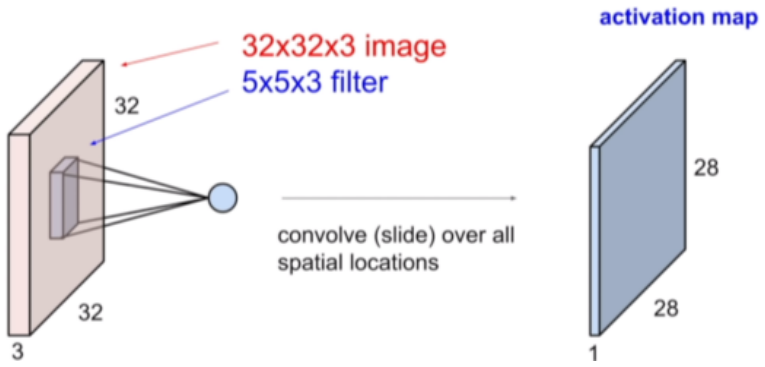
8.2 Layers used to build ConvNets

A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters.

8.2.1 Convolutional Layer (CONV)

The Convolutional layer is the core building block of a Convolutional Network and does most of the computational heavy lifting. The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume.
During the forward pass, each filter is slid (more precisely, convolved) across the width and height of the input volume and the dot products between the entries of the filter and the input at any position are computed. As the filter is slid over the width and height of the input volume it will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. There's an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. These activation maps are stacked along the depth dimension thus producing the output

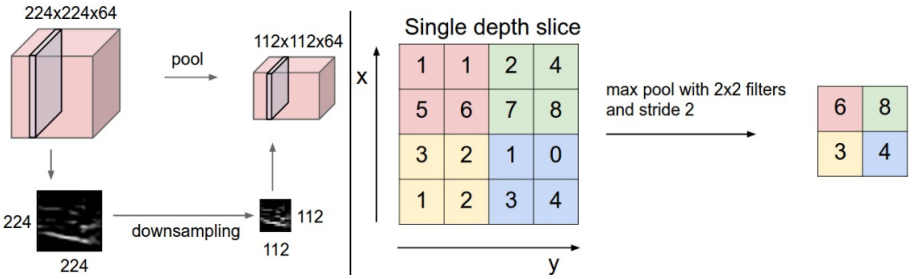
volume. Each neuron is connected only to a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume.



Three hyperparameters control the size of the output volume: the **depth**(number of filters), **stride**(number of pixels to jump at one time while sliding) and **zero-padding**.
The spatial size of the output volume can be computed as a function of the input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. The correct formula for calculating how many neurons **fit** is given by $(W - F + 2P)/S + 1$.

8.2.2 Pooling Layer (POOL)

The function of Pooling Layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, usually using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2. It downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. In addition to max pooling, the pooling units can also perform other functions, such as average pooling or even L2-norm pooling.



8.2.3 Fully-connected layer (FC)

Neurons in a fully connected layer have full connections to all activations in the previous layer, just as in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.
The only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical. Therefore, it's possible to convert between FC and CONV layers.

8.3 ConvNet Architectures

Convolutional Networks are commonly made up of only three layer types: CONV, POOL (Max pool is assumed unless stated otherwise) and FC. ReLU activation function is explicitly written as a layer, which applies elementwise non-linearity.

8.3.1 Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows the pattern:

INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC

where the ***** indicates repetition, and the **POOL?** indicates an optional pooling layer. Moreover, **N** >= **0** (and usually **N** <= **3**), **M** >= **0**, **K** >= **0** (and usually **K** < **3**).

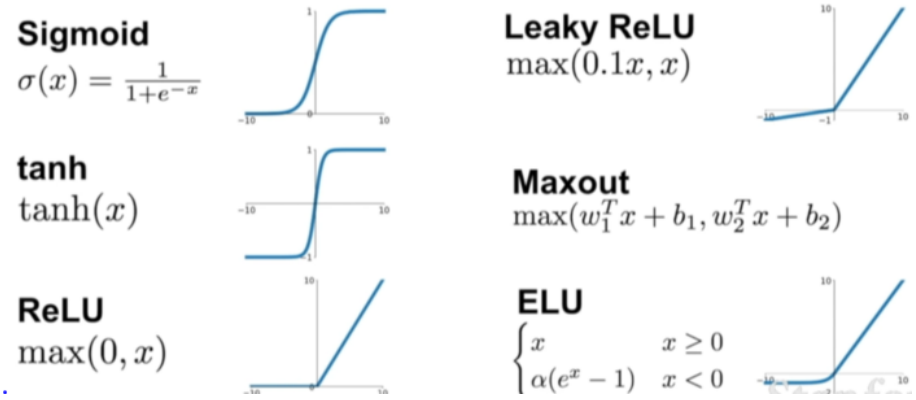
8.3.2 Layer Sizing Patterns

The common rules of thumb for sizing the architectures:

- The **input layer** (that contains the image) should be divisible by 2 many times (e.g. 32, 64, 96, 224, 384, 512).
- The **conv layers** should be using small filters (e.g. 3x3 or at most 5x5), using a stride of 1, and crucially, padding the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input.
- The **pool layers** are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields , and with a stride of 2.

9 Training Neural Networks

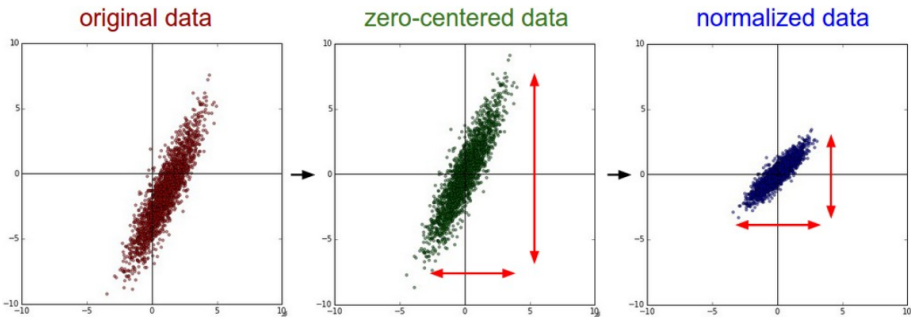
9.1 Commonly used activation functions



9.2 Data Preprocessing

The common forms of data preprocessing a data matrix **X** of size [**N x D**] (**N** is the number of data, **D** is their dimensionality) are :

- **Mean subtraction** is the most common form of preprocessing. It involves subtracting the mean across every individual feature in the data, and has the geometric interpretation of centering the cloud of data around the origin along every dimension.
- **Normalization** refers to normalizing the data dimensions so that they are of approximately the same scale. There are two common ways of achieving this normalization. One is to divide each dimension by its standard deviation, once it has been zero-centered. Another form of this preprocessing normalizes each dimension so that the min and max along the dimension is -1 and 1 respectively.



9.3 Weight Initialization

Initializing the network with the right weights can be the difference between the network converging in a reasonable amount of time and the network loss function not going anywhere even after hundreds of thousands of iterations. If the weights are too small, then the variance of the input signal starts diminishing as it passes through each layer in the network. The input eventually drops to a really low value and can no longer be useful. If the weights are too large, then the variance of input data tends to rapidly increase with each passing layer. Eventually it becomes so large that it becomes useless. (consider sigmoid activation function - saturation on either sides)
Xavier initialization is a very good method of initialization of weights. Consider a linear neuron :

$$y = w_1x_1 + w_2x_2 + ... + w_Nx_N + b$$

With each passing layer, the variance is desired to remain the same so as to keep the signal from exploding to a high value or vanishing to zero. In other words, the weights need to be initialized in such a way that the variance remains the same for x and y . This initialization process is known as Xavier initialization. The weights are picked from a Gaussian distribution with zero mean and a variance of $1/N$, where N specifies the number of input neurons.
Initializing the biases : It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights.

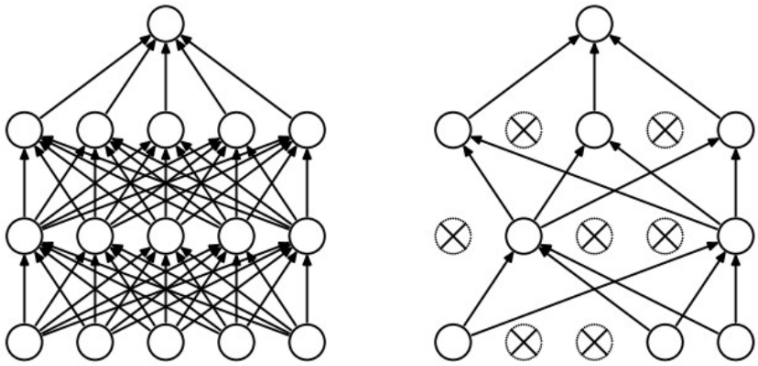
9.4 Regularization

L2 regularization is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. That is, for every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. Due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather than some of its inputs a lot.

L1 regularization is another relatively common form of regularization, where for each weight w we add the term $\lambda|w|$ to the objective. It is possible to combine the L1 regularization with the L2 regularization : $\lambda_1|w| + \lambda_2w^2$ (this is called Elastic net regularization). The L1 regularization leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero). In other words, neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs.

Max norm constraints. Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. In practice, this corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector w of every neuron to satisfy $||w||_2 < c$. Typical values of c are on orders of 3 or 4.

Dropout is an extremely effective, simple technique to Prevent Neural Networks from Overfitting that complements the other methods (L1, L2, maxnorm). While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise.



(a) Standard Neural Net (b) After applying dropout.

9.5 Parameter updates

Once the analytic gradient is computed with backpropagation, the gradients are used to perform a parameter update. There are several approaches for performing the update :

9.5.1 Stochastic Gradient Descent (SGD)

- **Vanilla update**. The simplest form of update is to change the parameters along the negative gradient direction.

```
x += - learning_rate * dx
```

- **Momentum update** is another approach that almost always enjoys better converge rates on deep networks. This update can be motivated from a physical perspective of the optimization problem. The loss can be interpreted as the height of a hilly terrain. Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location. The optimization process can then be seen as equivalent to the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape.

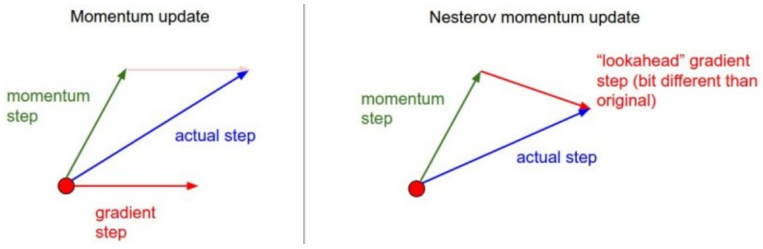
```
v = mu * v - learning_rate * dx      #integrate velocity
x += v      #integrate position
```

  The variable `v` is initialized at zero. The hyperparameter (`mu`) damps the velocity and reduces the kinetic energy of the system, or otherwise the particle would never come to a stop at the bottom of a hill. When cross-validated, this parameter is usually set to values such as [0.5, 0.9, 0.95, 0.99].

- **Nesterov Momentum** is a slightly different version of the momentum update. In practice, it consistenly works slightly better than standard

momentum.

Instead of evaluating gradient at the current position (red circle), it is known that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum, the gradient at this "looked-ahead" position is therefore evaluated instead .



9.5.2 Annealing the learning rate

In training deep networks, it is usually helpful to anneal the learning rate over time. It is good intuition that with a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function. There are three common types of implementing the learning rate decay:

- **Step decay**: Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model.
- **Exponential decay** has the mathematical form $\alpha = \alpha_0 e^{-kt}$, where α_0, k are hyperparameters and t is the iteration number (can also be units of epochs).
- **1/t decay** has the mathematical form $\alpha = \alpha_0/(1 + kt)$ where α_0, k are hyperparameters and t is the iteration number.

9.5.3 Adaptive learning rate methods

- **Adagrad** :

```
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

The variable `cache` has size equal to the size of the gradient, and keeps track of per-parameter sum of squared gradients. This is then used to normalize the parameter update step, element-wise. The weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased. The square root operation is very important and without it the algorithm performs much worse. The `eps` term (usually set somewhere in range from 1e-4 to 1e-8) is to avoid division by zero.

- **RMSprop** :
  The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. In particular, it uses a moving average of squared gradients instead, giving:
  ```
  cache = decay_rate * cache + (1 - decay_rate) * dx**2
  x += - learning_rate * dx / (np.sqrt(cache) + eps)
  ```
  `decay_rate` is a hyperparameter and typical values are [0.9, 0.99, 0.999].The `x+=` update is identical to Adagrad, but the `cache` variable is "leaky". Hence, RMSProp still modulates the learning rate of each weight based on the magnitudes of its gradients, but unlike Adagrad the updates do not get monotonically smaller.

- **Adam** :
  Adam is a recently proposed update that looks a bit like RMSProp with momentum:
  ```
  m = beta1*m + (1-beta1)*dx
  v = beta2*v + (1-beta2)*(dx**2)
  x += - learning_rate * m / (np.sqrt(v) + eps)
  ```
  Recommended values in the paper are `eps = 1e-8, beta1 = 0.9, beta2 = 0.999.` In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. The full Adam update also includes a bias correction mechanism, which compensates for the fact that in the first few time steps the vectors m,v are both initialized and therefore biased at zero. With the bias correction mechanism, the update looks as follows:
  ```
  m = beta1*m + (1-beta1)*dx
  mt = m / (1-beta1**t)
  v = beta2*v + (1-beta2)*(dx**2)
  vt = v / (1-beta2**t)
  x += - learning_rate * mt / (np.sqrt(vt) + eps)
  ```

9.6 Model Ensembles

One reliable approach to improving the performance of Neural Networks by a few percent is to train multiple independent models, and at test time average their predictions. As the number of models in the ensemble increases, the performance typically monotonically improves. Moreover, the improvements are more dramatic with higher model variety in the ensemble. There are a few approaches to forming an ensemble:

- Same model, different initializations.
- Top models discovered during cross-validation.
- Different checkpoints of a single model.
- Running average of parameters during training.

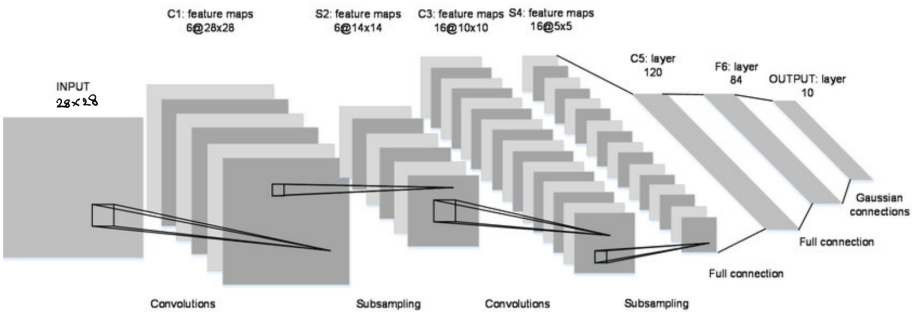
10 Implementation of learned topics

LeNet

The LeNet architecture was first introduced by LeCun et al. in their 1998 paper, **Gradient-Based Learning Applied to Document Recognition**. As the name of the paper suggests, the authors' implementation of LeNet was used primarily for OCR and character recognition in documents.

The LeNet architecture is straightforward and small, (in terms of memory footprint), making it perfect for learning the basics of constructing CNNs.

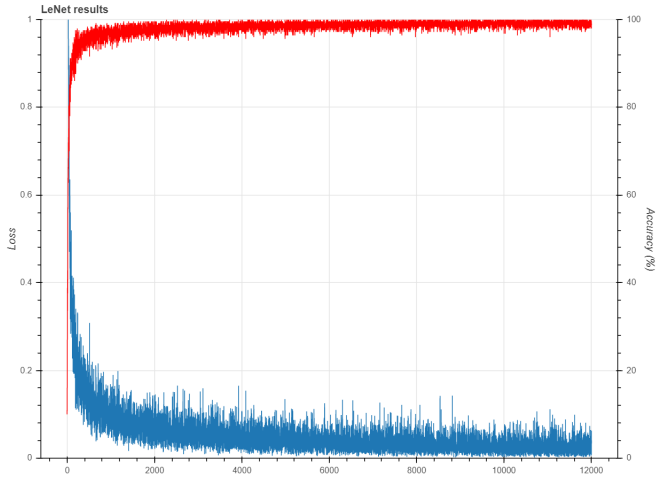
In this, I have trained the LeNet architecture on the MNIST dataset. The goal of this dataset is to classify the handwritten digits 0-9. It has a total of 70,000 images, with 60,000 images used for training and 10,000 used for evaluation. Each digit is represented as a **28 x 28 grayscale image**. The network structure is as follows:



INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => RELU => FC => RELU => FC

The python notebook for the implementation of LeNet can be found at
https://github.com/shubhlohiya/SoS_2019_LeNET

The training of the network on the MNIST dataset over 50 epochs and at batch size 250 was as follows:



11 References

- CS231n Course By Stanford
- PyTorch Tutorials
- NumPy Tutorial