Machine Learning

by ambedkar@IISc

- Feed Forward Neural Networks
- Backpropagation Algorithm
- CNNs and RNNs

Agenda

Introduction

A Snap Shot of Deep Learning

Features

- Go beyond the curve fitting...
- Amazing results with raw data...
- Pay and get the tagged data...

A Snap Shot of Deep Learning (cont...)

Popular Models

- Feed Forward Neural Networks
- Convolutional Neural Networks
- Recurrent Neural Networks and Long Short Term Memory Networks
- Restricted Boltzmann Machines and Deep Boltzmann Machines
- Autoencoders
- Generative Adversarial Networks,
- Variational Autoencoders
- a lot more....

A Snap Shot of Deep Learning (cont...)

Tools

- PyTorch
- Theano (outdated)
- Caffe
- TensorFlow

A Snap Shot of Deep Learning (cont...)

Consequences

- Brought "AI" back into computer science ***forey***
- If it works, we accept....we do not mind waiting for "why"

Shallow Vs. Deep

Until recently most machine learning and signal processing techniques had exploited "Shallow-Structured Architectures"

- Shallow: typically contain at most one or two layers of nonlinear function transformations
  - Gaussian mixture models
  - Conditional random fields
  - Linear or nonlinear dynamical systems
  - Maximum entropy models
  - Support vector machines
  - Logistic regression
  - Kernel regression
  - Multilayer perceptron with single hidden layers.
- Deep: More nonlinear "hidden" layers

Shallow Vs. Deep (Cont...)

- Shallow architectures have been effective in solving many simple or well constrained problems
- Shallow architectures have limited modeling and "representation power" can cause difficulties when dealing with more complicated real world applications involving natural signals:
  - human speech,
  - natural language,
  - natural images and scenes

Shallow Vs. Deep (Cont...)

- These shallow architectures work well given very good "hand crafted features" (may require signal processing techniques)
- Advantage is that training is easy and may end up mostly with a "convex optimization problem".

Human Perception and Evidence for layered hierarchical systems

Human information processing mechanism (eg. vision and audio) suggest the need of deep architectures for extracting complex structures and building internal representations from rich sensory inputs.

- Human speech production and perception systems are equipped with "layered hierarchical structures" in transforming the information from the waveform level to the linguistic level.
- Human visual systems on the perception side is hierarchical but also "generative."

Can we also emulate the same?

- The concept of deep learning is obtained from neural networks
- Feedforward neural networks or MLPs with many hidden layers, which are often referred to as deep neural networks (DNNs)
- Back propagation is popularized in 1980 and has been well known algorithm for learning parameters.

Can we also emulate the same? (Cont...)

- Unfortunately BP alone did not work because nonconvex nature of resulting optimization problems
- The bigger problem: **Vanishing gradient problem**
- This has steered away most of the ML researchers from natural networks to shallow models that have convex loss functions like
  - support vector machines (SVM),
  - conditional random fields (CRF) and
  - maximum entropy models (MaxEnt)

  for which global optimum can be efficiently obtained at the cost of reduced modeling power.

What has changed now?

The optimization difficulties associated with the deep models was empirically alleviated when a “reasonably efficient” unsupervised learning algorithms were introduced by Hinton (2006)

What has changed now? (Cont...)

- DBN is composed of a stack of restricted Boltzmann machines (RBM)
- A greedy, layer by layer algorithm optimizes DBM weights at the at the time complexity linear to the size and depth of the networks.
- DBMs can be used to initialize the training of Deep Neural Networks (DNN)
- Advantages of DBMs:
  - Supply of good initialization for DNNs
  - Learning algorithm makes effective use of unlabeled data

What has changed now? (Cont...)

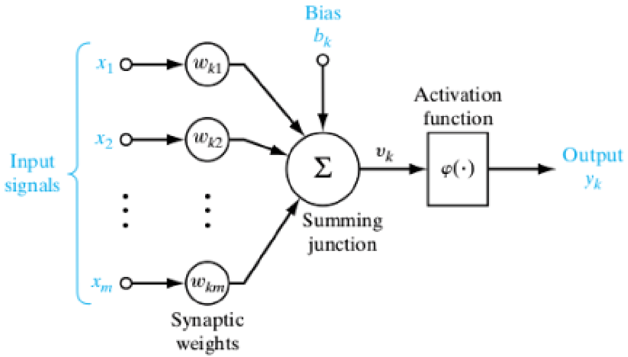
Most importantly GPUs and Tools like Torch and TensorFlow

Perceptron: History

History:

- McCulloch and Pitts (1943) introduced idea of neural networks as computing machines
- Hebb (1949) postulated the first rule for self organizing maps
- Rosenblatt (1958) invented perceptron that algorithmically described neural networks

Perceptron: History



Deep Learning: Advantages

- Nonlinearity
- Input-Output mapping
- Adaptivity
- Fault Tolerance
- VLSI implementability

Where do we start?

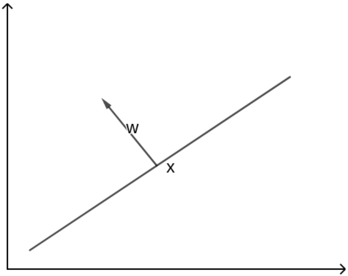
- Perceptron (Recall)
- Feed forward deep networks and Back propagation algorithm
- later CNNs, LSTMs (if time permits)

Perceptron (Recall)

Hyperplanes

- Seperates a d-dimensional space into two half spaces(positive and negative)
- Equation of the hyperplane is

$$w^{\intercal}x = 0$$



- By adding bias $b \in \mathbb{R}$

$$w^{\intercal}x + b = 0 \quad \begin{array}{ll} b > 0 & \text{moving the hyperplane parallely along } w \\ b < 0 & \text{opposite direction} \end{array}$$

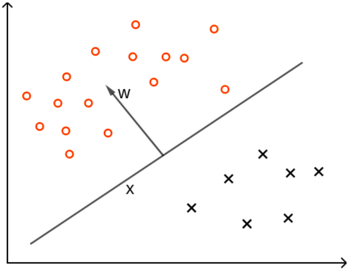
Hyperplane based classification

- Classification rule

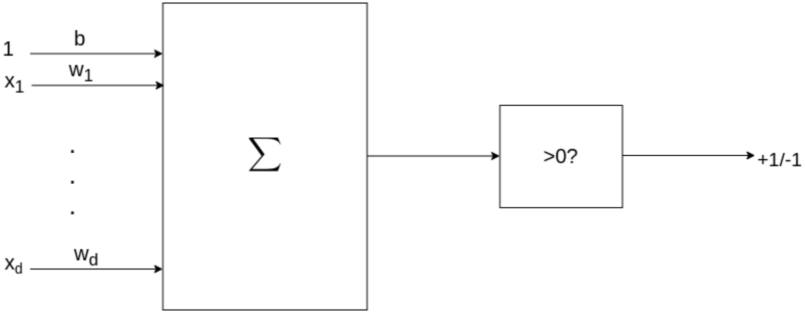
$$y = \text{sign}(w^\intercal x + b)$$

-

$$w^\intercal x + b > 0 \implies y = +1$$
$$w^\intercal x + b < 0 \implies y = -1$$



Hyperplane based classification



The Perceptron Algorithm (Rosenblatt, 1958)

- Aim is to learn a linear hyperplane to separate two classes.
- Mistake drives online learning algorithm
- Guaranteed to find a separating hyperplane if data is linearly separable.
- If data is not linearly separable
  - Make linearly separable using kernel methods.
  - (Or) Use multilayer perceptron.

Perceptron Algorithm

- Given training data $\mathcal{D} = \{(x_1, y_1), ..., (x_n, y_n)\}$
- Initialize $w_{old} = [0, ..., 0], \ \ b_{old} = 0$
- Repeat until convergence.
  - For a random $(x_n, y_n) \in \mathcal{D}$
  - If $y_n(w^\intercal x_n + b) \leq 0$
    [Or $\text{sign}(w^\intercal x + b) \neq y_n$ i.e mistake mode]
  - $w_{new} = w_{old} + y_n x_n$
  - $b_{new} = b_{old} + y_n$

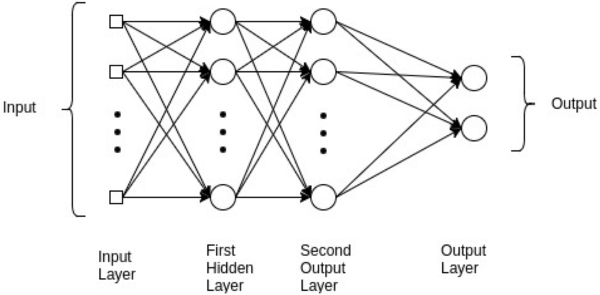
Perceptron Convergence Theorem (Block and Novikoff)

"Roughly" : If the data is linearly separable perceptron algorithm converges.

Feed Forward Neural Networks

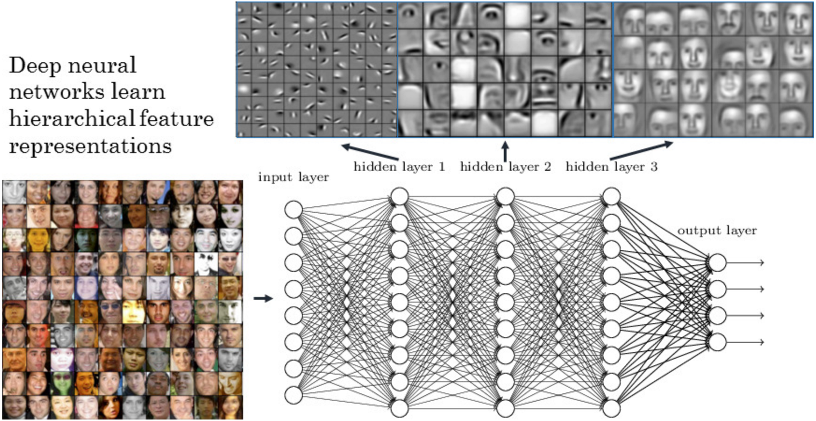
Some Basic Features of Multilayer Perceptrons (or Feed-forward Deep Neural Networks

- Network will have hidden layers.
- Since perceptron works only for linearly separable data, each neuron has a non-linear activation function: Activation function is differentiable.
- Network exhibit a high degree of connectivity.



Why hidden Layers?

Deep neural networks learn hierarchical feature representations



Why hidden Layers? (Cont...)

- Hidden layers can automatically learn features from data
- The bottom-most hidden layer captures very low level features (e.g., edges). Subsequent hidden layers learn progressively more high-level features (e.g., parts of objects) that are composed of previous layer's features

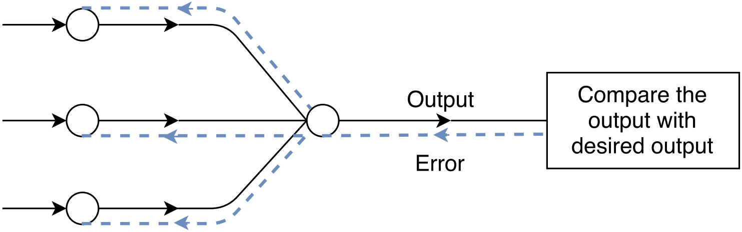
Two important steps in training neural network

1. **Forward step:**
   - Input is fed to the first layer.
   - Input signal is propagated through the network layer by layer.
   - Synaptic weights of the network are fixed i.e. no learning happens in this step.
   - Error is calculated at the output layer by comparing the observed output with "desired output" (Ground truth)
2. **Backward step:**
   - The observed error at the output layer is propagated "backwards", layer by layer. (How?)
   - Error is propagated "backwards", layer by layer.
   - In this step, successive adjustments are made to the synaptic weights.

Propagation of information in neural network

Two kinds of signals:

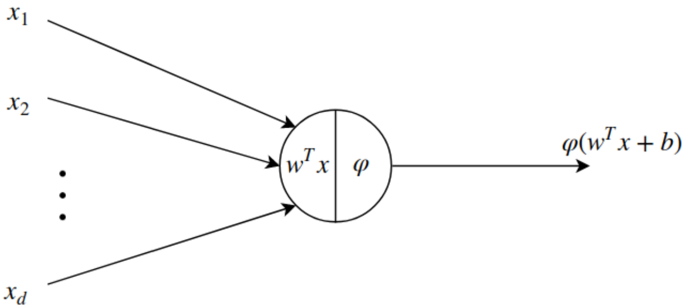
1. Function signal (leads to observed error)
2. Error signal (leads to updation of weights or parameters)



Propagation of information in neural network

Computation of signals

1. Computation of function signal (in the forward step)



2. Computation of gradient
   - gradients of the "error surface" w.r.t weights (we will see this later how)

Error

- $\mathcal{D} = \{(x(n), z(n))\}_{n=1}^{N}$ be a training sample where x(n) is an input and z(n) is the desired output.
  - $x(n) \in \mathbb{R}^D$, we write $x(n) = (x_1(n), ..., x_D(n)$.
  - $z(n) \in \mathbb{R}^M$, we write $z(n) = (z_1(n), ..., z_M(n))$.
- Suppose the output of the network is $y(n) = (y_1(n), ..., y_M(n)$ when x(n) is the input.
- Error at the j^{th} output neuron is

$$e_j(n) = |z_j(n) - y_j(n)|, \qquad j = 1, 2, ..., M$$

Error (contd. . .)

- The total error per sample is

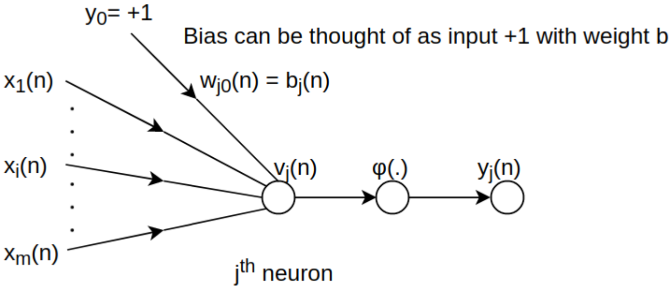
$$\mathcal{E}(n) = \frac{1}{2}\sum_{j=1}^{M}(z_j(n) - y_j(n))^2$$

- Average error for the training data or empirical risk

$$\overline{\mathcal{E}} = \frac{1}{N}\sum_{n=1}^{N}\mathcal{E}(n) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j=1}^{M}(z_j(n) - y_j(n))^2$$

Backpropagation Algorithm: 1

The Backpropagation Algorithm



The Backpropagation Algorithm

- $x_1(n), \ldots, x_i(n), \ldots, x_m(n)$: Function signals that are produced by the previous layer which is an input to the j^{th} neuron.
- $v_j(n) = \sum_{i=0}^{m} w_{ji}(n) x_i(n)$
  - $v_j(n)$ is the induced local field.
  - m is the size of the input (i.e. in the previous layer there are m neurons)
- $y_j(n) = \varphi(v_j(n))$
  - Function signal appearing at the output of neuron j.

The Backpropagation Algorithm (cont...)

- BPA applies a correction $\Delta w_{ji}(n)$ to the synaptic weight proportional to $\dfrac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}, \qquad i = 1, 2, \ldots, m.$

  - Note that we are trying to update j^{th} neuron out of all M neurons.

  - For n^{th} data point, the error is

$$\mathcal{E}(n) = \frac{1}{2}\sum_{j=1}^{M}\left(z_j(n) - y_j(n)\right)^2 = \frac{1}{2}\sum_{j=1}^{M} e_j^2$$

The Backpropagation Algorithm (cont...)

We compute the derivative of

$$\mathcal{E}(n) = \frac{1}{2}\sum_{j=1}^{M}(z_j(n) - y_j(n))^2$$

w.r.t $w_{ji}(x)$ (apply chain rule)

The Backpropagation Algorithm (cont...)

The derivative

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Since

- $\mathcal{E}$ is a function of e_j
- e_j is a function of y_j (y_j is the output)
- y_j is a function of v_j (v_j is the local field)
- v_j is a function of w_{ji}

The Backpropagation Algorithm (cont...)

- The derivative is

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \quad \frac{\partial e_j(n)}{\partial y_j(n)} \quad \frac{\partial y_j(n)}{\partial v_j(n)} \quad \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

  - $\mathcal{E}(n) = \frac{1}{2}\sum_{j=1}^{M} e_j^2(n) \implies \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n)$
  - $e_j(n) = z_j(n) - y_j(n) \implies \frac{\partial e_j(n)}{\partial y_j(n)} = -1$
  - $y_j(n) = \varphi(v_j(n)) \implies \frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$
  - $v_j(n) = \sum_{i=0}^{m} w_{ji}(n)x_i(n) \implies \frac{\partial v_j(n)}{\partial w_{ji}} = x_i(n)$
- $\implies$

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))x_i(n)$$

The Backpropagation Algorithm (contd. . .)

Update rule for jth output neuron

- We have
$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))x_i(n)$$

- Hence, the update rule is
$$\begin{aligned} w_{ji}(n+1) &= w_{ji}(n) - \eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \\ &= w_{ji}(n) + \eta e_j(n)\varphi_j'(v_j(n))x_i(n) \end{aligned}$$

Local Gradient

- Define local gradient $\delta_j(n)$ for j^{th} neuron as

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)}$$
$$= -\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \quad \frac{\partial e_j(n)}{\partial y_j(n)} \quad \frac{\partial y_j(n)}{\partial v_j(n)} \quad = e_j(n)\varphi_j'(v_j(n))$$

- $w_{ji}(n+1) = w_{ji}(n) + \eta \underbrace{\delta_j(n)}_{\text{Local Gradient}} x_i(n)$

BPA: Case 1: Neuron j is an output node

- Output neuron has an "easy" access to the error

$$e_j(n) = d_j(n) - y_j(n)$$

$$\mathcal{E}(n) = \frac{1}{2} \sum_{j=1}^{M} e_j^2(n)$$

BPA: Case 1: Neuron j is an output node (Cont...)

- Update rule

$$
\begin{aligned}
w_{ji}(n+1) &= w_{ji}(n) - \eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \\
&= w_{ji}(n) - \eta \underbrace{e_j(n)\varphi'(v_j(n))}_{\text{Local Gradient}} x_i(n) \\
&= w_{ji}(n) - \eta \delta_j(n) x_i(n)
\end{aligned}
$$

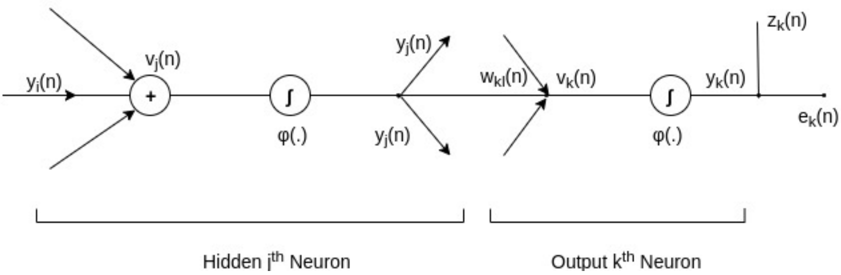
BPA: Case 1: Neuron j is an output node (Cont...)

- Update rule

$$\begin{aligned} w_{ji}(n+1) &= w_{ji}(n) - \eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \\ &= w_{ji}(n) - \eta \underbrace{e_j(n)\varphi'(v_j(n))}_{\text{Local Gradient}} x_i(n) \\ &= w_{ji}(n) - \eta \delta_j(n) x_i(n) \end{aligned}$$

BPA: Case 2: Neuron j is a hidden node

- Unlike in the case of output neuron, hidden neuron does not have a direct access to the "error".
- **TRICK**
  - Error signal for a hidden neuron will be determined recursively.
  - They expect the next hidden neuron (or output neuron), they are connected to, to share "some" error.
  - Error propagates by working backwards.



Strategy

- First compute the local gradient $\delta_j(n)$ for j^{th} hidden neuron (How we will see...)
- Then use the update that is similar to output neuron

$$\Delta w = \text{Learning rate} \times \text{Local gradient} \times \text{Input}$$
$$= \eta\delta_j(n)x_i(n)$$

BPA: Case 2: Neuron j is a hidden node (contd...)

Local field of j^{th} hidden neuron:

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \quad \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \quad \varphi_j'(v_j(n)) \qquad \because y_j(n) = \varphi_j(v_j(n))$$

Note: If this had been the output neuron, we would have had

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \quad \frac{\partial e_j(n)}{\partial y_j(n)} = -e_j(n)$$

Since j is hidden, it does not have access to the error.

BPA: Case 2: Neuron j is a hidden node (contd...)

- We are trying to compute local gradient

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n))$$

- Let us compute $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}$
  - We have $\mathcal{E}(n) = \frac{1}{2} \underbrace{\sum_{k \in C} e_k^2(n)}_{\text{Summation over all the output neurons}}$
  - Then

$$\begin{aligned}\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} &= \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial y_j(n)} \\ &= \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \quad \frac{\partial v_k(n)}{\partial y_j(n)}\end{aligned}$$

BPA: Case 2: Neuron j is a hidden node (contd...)

We are computing $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \quad \frac{\partial v_k(n)}{\partial y_j(n)}$

- We have

$$\begin{aligned} e_k(n) &= z_k(n) - y_k(n) \\ &= z_k(n) - \varphi_k(v_k(n)) \end{aligned}$$

- $\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi'(v_k(n))$
- We have $v_k(n) = \sum_{l=1}^{m} w_{kl}(n) y_l(n)$
  Note that $j \in \{1, 2, ..., m\}$ and j^{th} neuron output along with the other neurons in that layer are fed to the k^{th} output neuron.
  $\implies \frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$

BPA: Case 2: Neuron j is a hidden node (contd...)

We are computing $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \quad \frac{\partial v_k(n)}{\partial y_j(n)}$

$$\begin{aligned} \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} &= -\sum_{k \in C} e_k(n) \varphi_k(v_k(n)) w_{kj}(n) \\ &= -\sum_{k \in C} \delta_k(n) w_{kj}(n) \end{aligned}$$

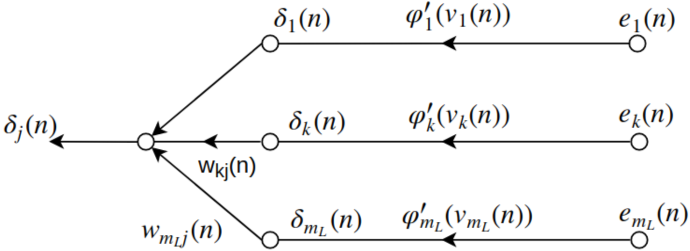
where $\delta_k(n) = e_k(n)\varphi_k(v_k(n))$ is the local gradient of the k^{th} neuron.

BPA: Case 2: Neuron j is a hidden node (contd...)

$$\text{We have } \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_k \delta_k(n) w_{kj}(n)$$

$$\text{and } \frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$$

$$Hence, \delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$



BPA: Case 2: Neuron j is a hidden node (contd...)

- Now we have local gradient at the j^{th} hidden node i.e. $\delta_j(n) = \varphi'_j(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$ where $\delta_k(n) = e_k(n)\varphi_k(v_k(n))$ is the local field at k^{th} output layer.

- Hence,

$$\begin{aligned} w_{ji}(n+1) &= w_{ji}(n) + \eta \delta_j(n) x_i(n) \\ &= w_{ji}(n) + \eta \left[\varphi'_j(v_j(n)) \sum_{k \in C} \delta_k(n) w_{kj}(n) \right] x_i(n) \end{aligned}$$

BPA: Update Rule Summary

Case 1: j^{th} neuron us a output neuron

$$\Delta w_{ji}(n) = \eta \underbrace{e_j(n)\varphi_j'(v_j(n))}_{\text{Local gradient at the } j^{th} \text{ neuron}} x_i(n)$$

Case 2: j^{th} neuron is a hidden neuron

$$\Delta w_{ji}(n) = \eta \underbrace{\left(\sum_{k \in C} \delta_k(n) w_{kj}(n)\right) \varphi_j'(v_j(n))}_{\text{Local gradient at the } j^{th} \text{ neuron}} x_i(n)$$

BPA: Update Rule Summary

Case 1: j^{th} neuron us a output neuron

$$\Delta w_{ji}(n) = \eta \underbrace{e_j(n)\varphi_j'(v_j(n))}_{\text{Local gradient at the } j^{th} \text{ neuron}} x_i(n)$$

Case 2: j^{th} neuron is a hidden neuron (see Bishop Section 5.3))

$$\Delta w_{ji}(n) = \eta \underbrace{\left(\sum_{k \in C} \delta_k(n) w_{kj}(n)\right) \varphi_j'(v_j(n))}_{\text{Local gradient at the } j^{th} \text{ neuron}} x_i(n)$$

Online Vs Batch Learning

Batch Learning

- Each adjustment to the weights is performed after the presentation of all the N examples in the training samples are presented.
- That is, cost function is average error or empirical risk.

$$\overline{\mathcal{E}} = \frac{1}{N}\sum_{n=1}^{N}\mathcal{E}(n) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j=1}^{M} e_j^2(n)$$
$$= \frac{1}{2N}\sum_{n=1}^{N}\sum_{j=1}^{M}(z_j(n) - y_j(n))^2$$

Online Vs Batch Learning (Cont...)

Batch Learning

- This constitutes one epoch of training.
- In each epoch of training, samples are randomly shuffled.
- The learning curve in this case is $\overline{\mathcal{E}}$ vs epoch number.
- Advantage: It can be easily parallelized.
- Disadvantage: Memory requirements are very high.

Online Vs Batch Learning

Online Learning

- Each adjustment to the weights is performed example by example in the training data.
- The cost function is error obtained in each sample.

$$\mathcal{E} = \frac{1}{2}\sum_{j=1}^{M} e_j^2(n) = \frac{1}{2}\sum_{j=1}^{M}(z_j(n) - y_j(n))^2$$

- The learning curve in this case is $\mathcal{E}(n)$ vs epoch.
- Learning curve is significantly different from that of batch learning.
- Online learning take advantage of redundant data (multiple copies of data).
- Online learning is simple to implement.

Activation Function

Activation function needs to be differentiable

1. **Logistic function:**

$$\varphi_j'(v_j(n)) = \frac{1}{1 + exp(-av_j(n))} \text{ a>0}$$

where v_j is induced local field and a is a parameter.

2. **Hyperbolic Tangent Function:**

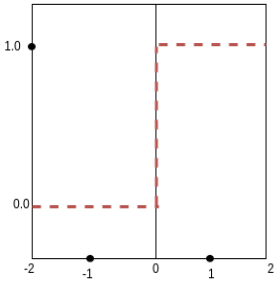
$$\varphi_j'(v_j(n)) = a \tanh(bv_j(n))$$

where a and b are positive constants.

Activation Functions

Heaviside step function:

$$\varphi(x) = 0 \qquad \text{if x<0}$$
$$\varphi(x) = 1 \qquad \text{if x>0}$$



This is useful
in the case of perceptron which works
only when the data is linearly separable.

Heaviside step function (contd.)

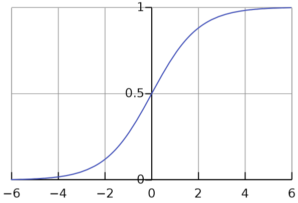
The reasons why we cannot use Heaviside step function in feedforward neural networks:

- We train neural network using backpropagation algorithm which requires differential activation function. For Heaviside step function, it is not differentiable at x=0 and it has 0 derivation everywhere else.
  $\implies$ The gradient descent will not be able to make progress in updating weights.
- We want our neural network weights to be modified continuously so that predictions can be as close as real values. Having a function that can only generate either 0 or 1 will not help to achieve this objective.

Activation Functions (contd...)

Sigmoid Function

- Sigmoid function is also known as the logistic function.
- It non-linearly squashes a number to a value between 0 and 1.
- $\text{Sigmoid}(x) = \frac{1}{1+e^{-z}}$
- Activations are bounded between 0 and 1.



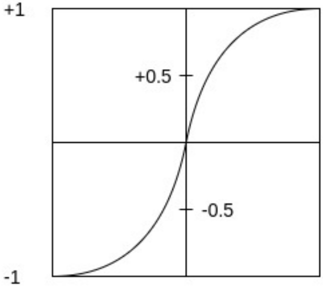
Sigmoid Function (contd...)
Disadvantages:

1. When the input is too small (towards -∞) the gradient is zero.
   - Hence while executing the backpropagation algorithm weights will not get updated i.e. there is no learning.
   - Vanishing gradient problem.
2. Though computing activation functions are less computationally expensive than matrix multiplication or convolutions, still computing exponential is expensive.

Tanh Function (or Hyperbolic tangent)

- It is similar to sigmoid function but squashes the values, non-linearly between -1 and 1.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} + 1$$

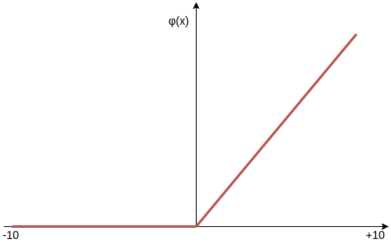


- Shares some disadvantages of sigmoid.

Rectified Linear Unit (ReLU)

- Given an input if it is negative or zero, it outputs zero. Otherwise it outputs the number same as input.

$$\mathrm{ReLU}(x) = \max(0, x)$$



Rectified Linear Unit (ReLU)

- Is it non-linear? Yes.
  - A linear function should satisfy the property that

$$f(x+y) = f(x) + f(y)$$

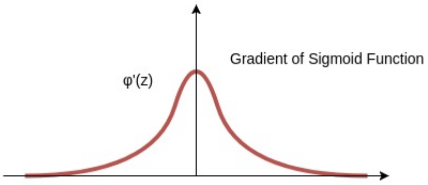
$$\text{But } \varphi(-1) + \varphi(+1) \neq \varphi(0)$$

  - But it is piece-wise linear.

Activation Functions (contd. . .)

ReLU (contd...)

- It is a non-bounded function.
- Change from sigmoid to ReLU as an activation function in hidden layer is possible - hidden neurons need not have bounded values.
- The issue with sigmoid function is that it has very small values (near zero) everywhere except near 0.



ReLU (contd...)

- At the j^{th} neuron which is a hidden layer

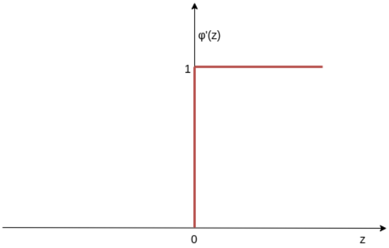
$$\Delta w_{ji}(n) = \eta \underbrace{\left(\sum_{k \in C} \delta_k(n) w_{kj}(n) \right)}_{\text{Local gradient from above}} \underbrace{\varphi_j'(v_j(n))}_{\substack{\text{Derivation of} \\ \text{sigmoid function}}}$$

  We multiply the gradient from the layer above with partial derivative of the sigmoid function.

- Since in the case of sigmoid, it has very small values everywhere except for when the input has values close to 0. $\implies$ Lower layers will likely have smaller gradients in terms of magnitude compared to higher layers.

ReLU (contd...)

- The reason is in the case of sigmoid $\varphi'(.)$ is always less than 1 with most values being 0.
  $\implies$ This imbalance in gradient magnitude makes it difficult to change the parameters of the neural networks with stochastic descent.



Relu gradient

ReLU (contd...)

- This problem can be adjusted by the use of rectified linear activation function because dervative of ReLU can have many non-zero values.
  $\implies$ Which in turn means that magnitude of the gradient is more balanced throughout the network.
- Dying ReLU: A neuron in the network is permanently dead due to inability to fire in forward pass.
  $\implies$ When activation is zero in the forward pass all the weights will get zero gradient.
  $\implies$ In backpropagation, the weights of neurons never get updated.
- Using ReLU in RNNs can blow up the computations to infinity as activations are not bounded.

Rate of Learning

"Backpropagation algorithm provides an 'approximation' to the trajectory in weight space computed by the method of steepest gradient."

Case 1:- Smaller the learning rate $\Rightarrow$ smaller the change to the synoptic weights in the network

$\Downarrow$

Smoother the learning will be

Case 2:- Larger learning rate $\Rightarrow$ The network may become unstable or oscillatory

Rate of Learning (contd...)

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

where α is the momentum constant

$\alpha = 0$ gives us original delta rule

Stopping Criteria

"In general backpropagation algorithm cannot be shown to converge"

$$\Downarrow$$

Hence this is not a well defined criteria for stopping its operation.

Some good Criteria:

- Euclidean norm of $\frac{\partial \mathcal{E}}{\partial w}$ reaches a sufficiently small gradient threshold.
  $\because$ The necessary condition for w^* to be global maximum or local minimum is $\left.\frac{\partial \mathcal{E}}{\partial w}\right|_{w^*} = 0$
- Absolute rate of change is the average squared error per epoch is sufficiently small.
- Stop when "generalizing performance" is adequate or it apparent that generalization performance is peaked.

Summary of Backpropagation Learning Algorithm

$$w_{ji}(n+1) = w_{ji}(n) + \eta\delta_j(n)x_i(n)$$

- η : learning rate
- $\delta_j(n)$: local gradient
- $x_i(n)$: Input to the j^{th} neuron

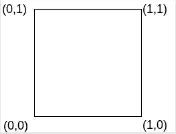
Local gradient:

$$\begin{aligned} \delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) && \text{if } j \text{ is a output neuron} \\ &= \Big[\sum_k \underbrace{\delta_k(n)}_{\substack{\text{local gradient} \\ \text{of } k^{th} \text{ neuron} \\ \text{at the output}}} w_{kj}(n)\Big]\varphi_j'(v_j(n)) && \text{if } j \text{ is a hidden neuron} \end{aligned}$$

XOR Problem

- Rosenblatt's single layer perceptron has no hidden layer hence it cannot classify input pattern that are NOT linearly separable.
  Consider XOR Problem:

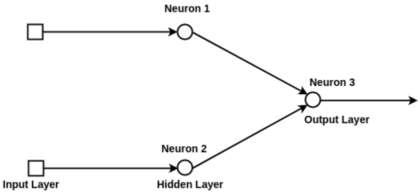
$$0 \oplus 0 = 0$$
$$1 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$



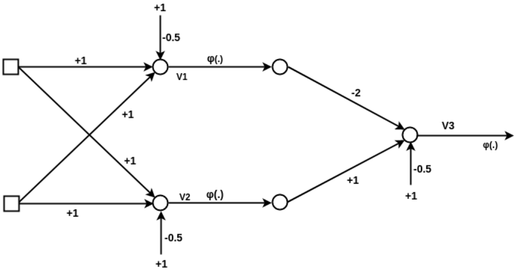
- $\{(0,0),(1,1)\}$ and $\{(0,1),(1,0)\}$ are not linearly separable. Hence single layer neural network cannot solve this problem.

XOR Problem (contd...)

- We use a hidden layer



- Consider the following neural network



XOR Problem (contd. . .)

- The function of output neuron : construct a linear combination of decision boundaries formed by the two hidden neurons.
  For various inputs :
  - For input (1,1): $v_1 = (1)(+1) + (1)(+1) + (+1)(\text{-}1.5)$
    $= 1 + 1 \text{ - } 1.5 = 0.5 \Rightarrow \varphi(v_1) = 1$
    $v_2 = (1)(+1) + 1(+1) + (+1)(\text{-}0.5)$
    $= 1 + 1 \text{ - } 0.5 = 1.5 \Rightarrow \varphi(v_2) = 1$
    $v_3 = 1(\text{-}2) + 1(+1) + (+1)(\text{-}0.5)$
    $= \text{-}2 + 1 \text{ - } 0.5 = \text{-}1.5 \Rightarrow \varphi(v_3) = 0$

XOR Problem (contd. . .)

For various inputs :

- For input (0,0): $v_1 = (+1)(\text{-}1.5) = \text{-}1.5 \Rightarrow \varphi(v_1) = 0$
  $v_2 = (+1)(\text{-}0.5) = \text{-}0.5 \Rightarrow \varphi(v_2) = 0$
  $v_3 = (+1)(\text{-}0.5) = \text{-}0.5 \Rightarrow \varphi(v_3) = 0$

- For input (1,0): $v_1 = (1)(+1) + (+1)(\text{-}0.5)$
  $= 1 \text{ - } 1.5 = \text{-}0.5 \Rightarrow \varphi(v_1) = 0$
  $v_2 = 1(+1) + (+1)(\text{-}0.5)$
  $= 1 \text{ - } 0.5 = 0.5 \Rightarrow \varphi(v_2) = 1$
  $v_3 = 1(+1) + (+1)(\text{-}0.5)$
  $= 1 \text{ - } 0.5 = 0.5 \Rightarrow \varphi(v_3) = 1$

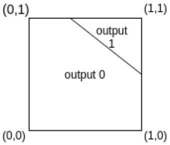
XOR Problem (contd...)

For various inputs :

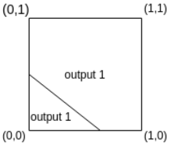
- For input (0,1):
$$v_1 = (1)(+1) + (+1)(\text{-}1.5)$$
$$= 1 \text{ - } 1.5 = \text{-}0.5 \Rightarrow \varphi(v_1) = 0$$
$$v_2 = 1(+1) + (+1)(\text{-}0.5)$$
$$= 1 \text{ - } 0.5 = 0.5 \Rightarrow \varphi(v_2) = 1$$
$$v_3 = 1(+1) + (+1)(\text{-}0.5)$$
$$= 1 \text{ - } 0.5 = 0.5 \Rightarrow \varphi(v_3) = 1$$

XOR Problem (contd. . .)

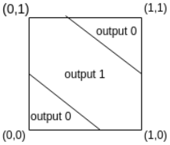
- Decision Boundary of neuron 1



- Decision Boundary of neuron 2



- Decision Boundary of neuron 3



Universal Approximation Theorem

Let $\varphi(.)$ be a non-constant, bounded and monotonic-increasing function. Let I_{m_0} denotes the m_0 -dimensional unit hypercube $[0,1]^{m_0}$. Let the space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, there exists an integer m_1 and sets of real numbers α_i , b_i and w_i, where $i = 1, 2, ...m$ and $j = 1, 2...m_0$.
Such that

$$F(x_1, ...x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi(\sum_{j=1}^{m_0} w_{ij} x_j + b_j)$$

and F arbitrarily approximates f(.). That is

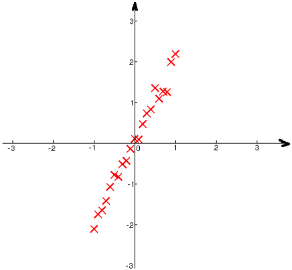
$$|F(x_1, ..x_{m_0}) - f(x_1, ...x_{m_0})| < \epsilon_0 \qquad \forall x_1, ...x_{m_0} \in I_{m_0}$$

Autoencoders

Introduction

- The origin of deep learning (post neural networks) since early 2000 was the use of Deep Belief Nentworks to "pretrain" deep networks.
- This approach is based on the observation that random initialization is not a good idea, and that pretraining each layer with an unsupervised learning algorithm can allow for better initial weights.
- Examples such unsupervised algorithms are
  - Deep Belief Networks based on Restricted Boltzmann Machines
  - Deep autoencoders

Compression



- Aim is to transmit this data: that is we have to send both the first and second dimension
- If we observe carefully, value at the second dimension is just twice the first dimension
- Hence we can just transmit first dimension (can be thought of as encoding of the data) and compute the value of the second dimension (can be thought of as decoding the data)

Compression (cont...)

The process...

- Encoding: Map the data x_n by means of some method to compressed data z_n
- Transmit
- Decoding: Map from compressed data z_n to $\tilde{x}_n$

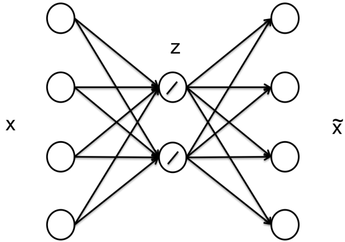
Autoencoder

A linear encoding and decoding

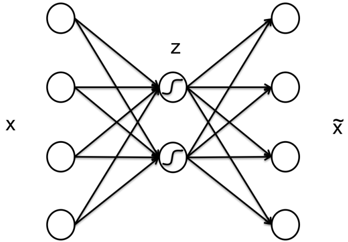
- Encoding: $z_n = W_1 x_n + b_1$
- Decoding: $\tilde{x}_n = W_2 z_n + b_2$

Objective function:

$$J(W_1, b_1, W_2, b_2) = \sum_{i=1}^{N} (\tilde{x}_n - x_n)^2$$



Autoencoder



- If the data lie on a nonlinear surface, we use nonlinear activation functions.
- If the data is highly nonlinear, one could add more hidden layers to the networks to have a deep encoder.
- Note that this is an unsupervised learning.

Convolutional Neural Networks

Convolutional Neural Network(Introduction)

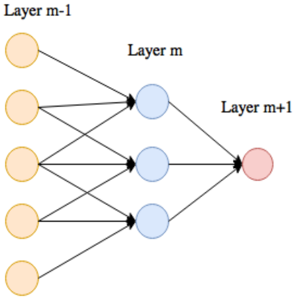
- Convolutional Neural Network (CNN) came into limelight in 2012
  - Alex Krizhevsky used CNN to win 2012 Imagenet competition.
  - The classification error has been improved from 20% to 15%.
  - **Paper:** Krizhevsky, Sutskever and Hinton : Imagenet classification with Deep Convolutioinal Neural Network, NIPS 2012.
- CNN were first proposed in the paper by Lecun, Bottou, Bengio, Haffner : Gradient based Learning Applied to Document Recognition, 1998 (*Proceedings of IEEE*)

Biological Connection

- Experiment by Hubel and Wiesel (1962)
- Some individual neuronal cells in the brain fire only in the presence of edges of certain Orientation.
- For example, Some neurons fired when exposed to vertical and some fired when exposed to horizontal edges.
- Hubel and Wiesel found that all these neurons were organized in a column architecture and that together they were able to produce visual perception.

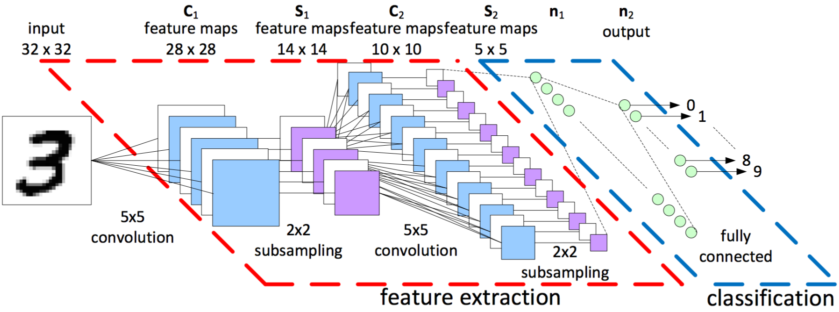
CNN

- CNN is a fixed feed forward Neural Network with special structure.
- Sparse "Local" connectivity between layer except the last output layer ⇒ Reduces the number of parameters.



Local connectivity of CNN

- Shared weights (like a global filter) Helps to capture the local properties of the signal (useful for the images)



input
32 x 32
C1
feature maps
28 x 28
S1
feature maps
14 x 14
C2
feature maps
10 x 10
S2
feature maps
5 x 5
n1
n2
output
0
1
8
9
5x5
convolution
2x2
subsampling
5x5
convolution
2x2
subsampling
fully
connected
feature extraction
classification

CNN

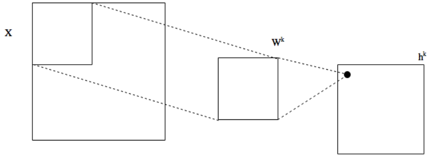
- **Convolution:** Extract "Local" properties of the signal, using "filters" that have to be learned.

- **Pooling:** Down Samples the output to reduce the size of representation.

- **Nonlinearity:** Non-linearity is used after the convolution layer.

Convolution

- This operation extracts local spacial properties of input



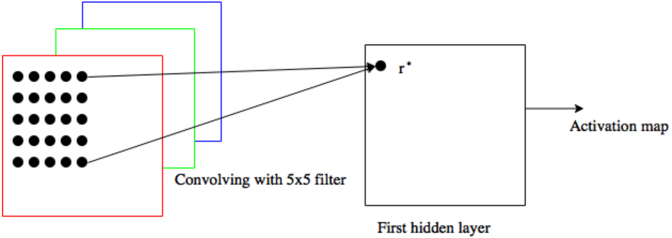
- The operation is defined as

$$h_{ij}^k = f((W^k \odot X)_{ij} + b_k)$$

  where W^k is a filter, $\odot$ is the convolution operation and f is a nonlinear function.
- Second filter $W^k, k = 1, 2, 3, ...$ are applied which need to be learned. Size of filter also need to be specified.

Convolution Layer

- A small portion of the image that we look at the image from a "lens".
- Suppose size of this lens is $5 \times 5 \times 3$.



$$r^* = \sum_{i=1}^{5} \sum_{j=1}^{5} a_{ij} b_{ij}$$

Convolution Layer (cont....)

- **Example:**

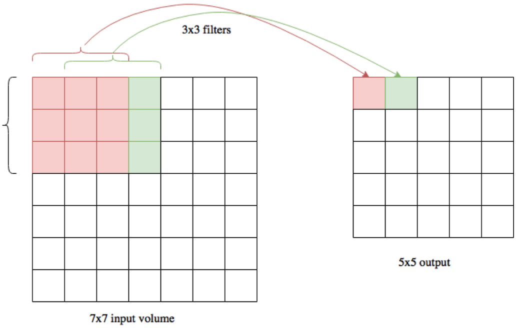
Input	Filter	Output
$32 \times 32 \times 3$	$5 \times 5 \times 3$	28×28
$32 \times 32 \times 32$	$5 \times 5 \times 3$ (2 nos.)	$28 \times 28 \times 2$

- Each filter can be thought of as a "Feature identifier".
- **Intuition:** In the input image, if there is a shape that generally resembles the curve that the particular filter is representing thus all the multiplications summed together will result in a large Value.

Convolution Layer (cont....)

Stride: Stride is size of the shift of the filter across the image (preciously we kept stride as 1).

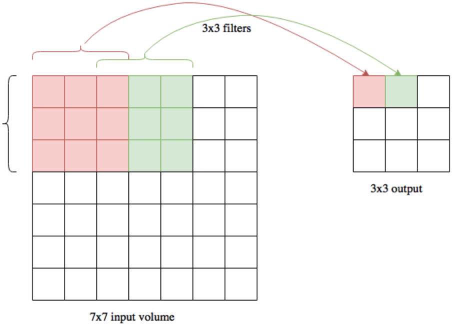
Ex.



3 × 3 convolution with a stride of 1

Convolution Layer (contd. . .)

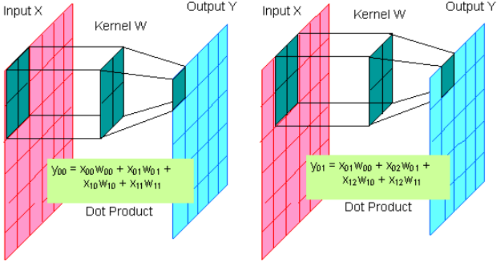
Stride (contd...) Example :



Convolution with stride of 2

- $$\text{Size of output} = \frac{\text{Size of input} - \text{Size of filter}}{\text{Stride}} + 1$$

Convolution with stride

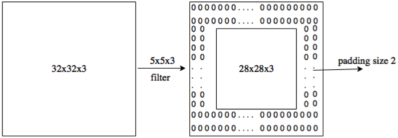


Filer is moved along the image and at each position the dot product is computed

Image taken from Poczos's notes

Convolution Layer (contd...)

Padding: If we want output to be same size as input then we pad the output with zeros.



Padding of two to the output

- To enforce size of input and output to be same we need the padding size to be

$$\text{size of padding} = \frac{\text{size of filter} - 1}{2}$$

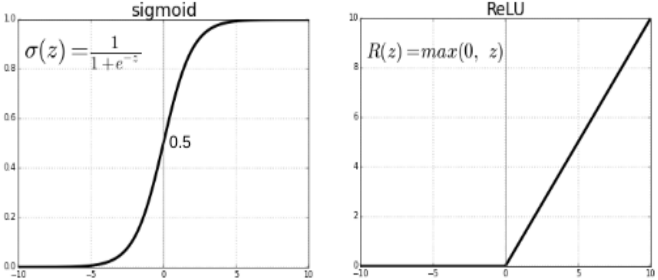
Convolution Layer (contd...)

In general,

$$\text{size of output} = \frac{\text{size of input} - \text{size of filter} + 2 * \text{size of padding}}{\text{size of stride}} + 1$$

Rectified Linear Unit OR ReLU

- A recent advance (not very recent) : Use ReLU, $y(z) = \max(0, z)$ as the activation function instead of traditional sigmoid function.

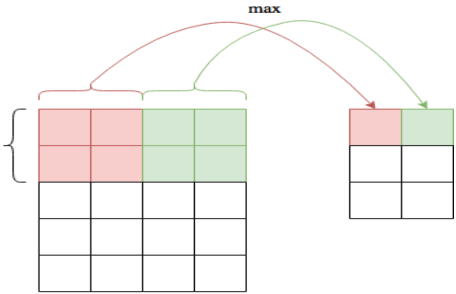


Activation functions

- ReLU improves performance of many networks.

Pooling or Down Sampling Layer

Divide layer into partitions and get a max or average of each partition



Maxpooling

- Max Pooling
- Average Pooling
- L_2 norm Pooling

Pooling or Down Sampling Layer (contd...)

- Advantages
  - Reduces the dimension of representation
  - Controls overfitting

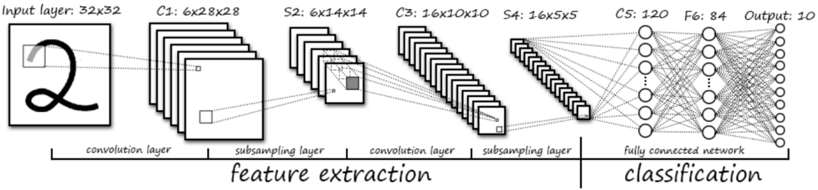
Lookout : If you have 99% to 100% accuracy on training set and only 40% to 50% of test accuracy it is a cause of concern.

Dropout Layer

- This layer drops random set of activation in that layer by setting them to zero.

- Helps as a Regularizer.

Architecture of LeNet-5 (LeCun et al, 1986)



Architecture of LeNet-5

- This is one of the first convolutional neural network
- This was designed to classify images of handwritten digits.
- Here the activation function used it *tanh*, but now the usual choice is *ReLU*.

Some Popular CNNs

Alex Net (2012)

- Trained on 15 million images.
- Achieved test error 15.4% (The next lost was only 26%).
- 5 convolution layer, max-pooling later, dropout layer and 3 fully connected layer.
- Used ReLU for activation.
- Used data augmentation techniques that consists of image translation, horizontal reflection and patch extraction.
- Implemented dropout layer in order to control overfitting to the training data.

Some popular CNNs (contd...)

Alex Net (2012) (contd...)

- Trained the model using batch stochastic gradient descent with specific values of momentum and decay.
- Trained on two GTx580 GPU's for five to six days.

ZF Net (2013) Zieler and Fergus

- Error of 11.2%.
- More of a fine tuning of Alex Net.
- Provided visualizations which provided better intuitions.
- ZF trained using 1.3 million images.

Some popular CNNs (contd. . .)

- Used 7x7 filters instead of 11x11 filters (as in Alex Net) also with decreased stride value.
  - Smaller filters in convolution layer help retain a lot of original pixel information in input image.
- ReLU for activation, cross entropy loss, training using batch stochastic descent.
- Trained on a GTx580 GPU for 12 days
- Deconvolutional network helps to visualize the feature maps.

Some popular CNN's (contd...)

VGG Net (2014) Simonyan and Zisserman

- Error 7.3%.
- 19 layers of convolutional layer, $3x3$ filters, padding of 2, max pooling with stride of 2.
- Trained for two to three weeks.

Google Net (2015)

- Error 6.7%
- 22 layer CNN.
- User Inception modules.

Some Popular CNN's (contd. . .)

Microsoft ResNet (2015)

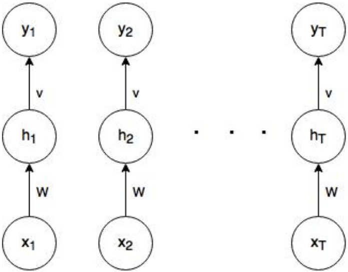
- Error 3.6%, (Human accuracy is $5 - 10\%$).
- Residual blocks.
- 152 layers.
- Trained on 8 GPU machines for 2 to 3 weeks.

Recurrent Neural Networks

What we have been doing so far?

Feed Forward Neural Networks

- Consists of input,hidden and output layers
- Given sequential data, FF networks does not take sequential structure in the data
- Given a sequence of observations, $x_1, \ldots, x_T$, then corresponding hidden units are $h_1, \ldots, h_T$ are assumed independent of each other (i.i.d data?)

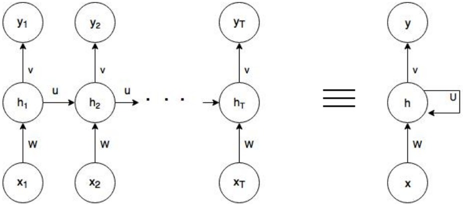


What we have been doing so far? (cont. . .)

- How can we use feed forward neural networks for sequential data like text, audio, video?
- Can we modify feed forward neural networks such a way that, it "remembers" the previous example?
- The answer is Recurrent Neural Networks or RNNs

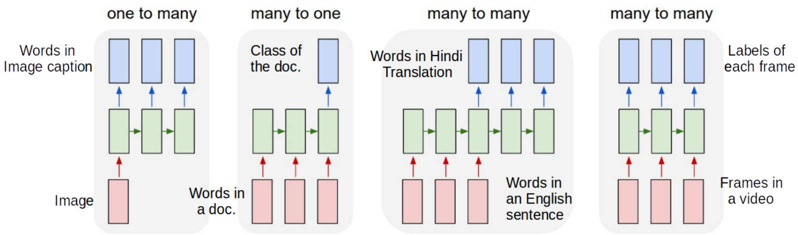
RNN (Introduction)

- Since we have sequential data,hidden state at each step depends on the hidden states of the previous



- Hence, $h_t = \varphi(Wx_t + Uh_{t-1})$ where U acts as a transition matrix and φ is a nonlinear activation function
- h_t acts as a memory
- RNNs can be considered as multiple copies of the same network, each passing a message to a successor.

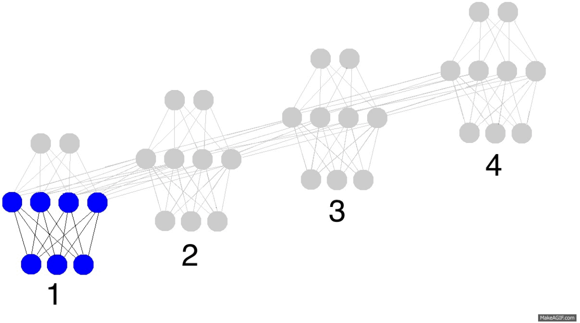
RNN Application



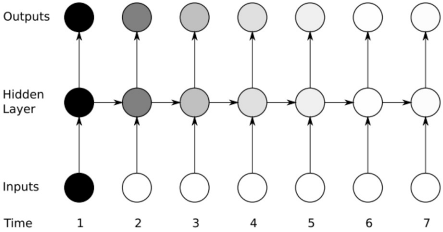
- RNN have many applications in modeling the sequential data
  - Input,Output or both can be sequences (possibly of different lengths)
  - Different inputs and different outputs need not be of the same length
- Regardless of the length of the input, RNN will learn fixed sized embedding for the input

RNN Training

- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



Vanishing Gradient Problem



- Learnability of hidden states and outputs become weaker as we move away from them along the sequence ⇒ Weak Memory
- New inputs "overwrite" the activations of the previous hidden states
- Repeated multiplications can cause the gradients to vanish or explode (with ReLU)

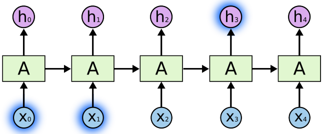
RNNs are really useful for sequential data?

- The whole idea of feedback loop is to able to connect previous information to the present task.
- For example, previous video frames may inform the understanding of the present frame.
- How much past information RNNs can remember so that they can be used for the present task?

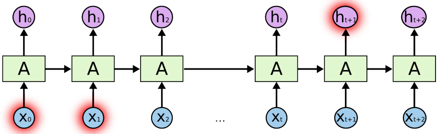
RNNs are really useful for sequential data? (cont...)

- Consider a language model trying to predict the next word based on the previous words.
- If the model is trying to predict the word `sky` in the setence `the clouds are in the sky`, that model do not require very old context.
- If we consider `I grew up in France...I speak fluent French`. There is a huge gap between the relevant information.
- If this gap is too much RNNs will not be able to connect the information.

RNNs are really useful for sequential data? (cont...)



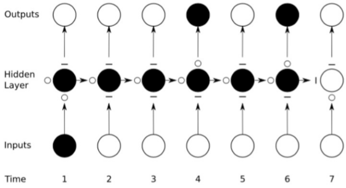
Short Range Dependencies



Long Range Dependencies

The solution is Long Short Term Memory Netowrks (Hochreiter & Schmidhuber, 1997)

Capturing the Long range Dependencies



- Augment hidden states with gates
  - The gates involves some parameters which needs to be learned
- These gates will help the model to remember and target the information selectively
- The hidden states has three types of gates
  - Input(bottom), Forget(left) and Output(top)
- Open 'o', close '-'

Some images and material on CNNs and RNNs is taken from Piyush Rai's Lecture Notes.

Homework:

Go through Colah's blog on LSTM networks.

http://colah.github.io/posts/2015-08-Understanding-LSTMs/