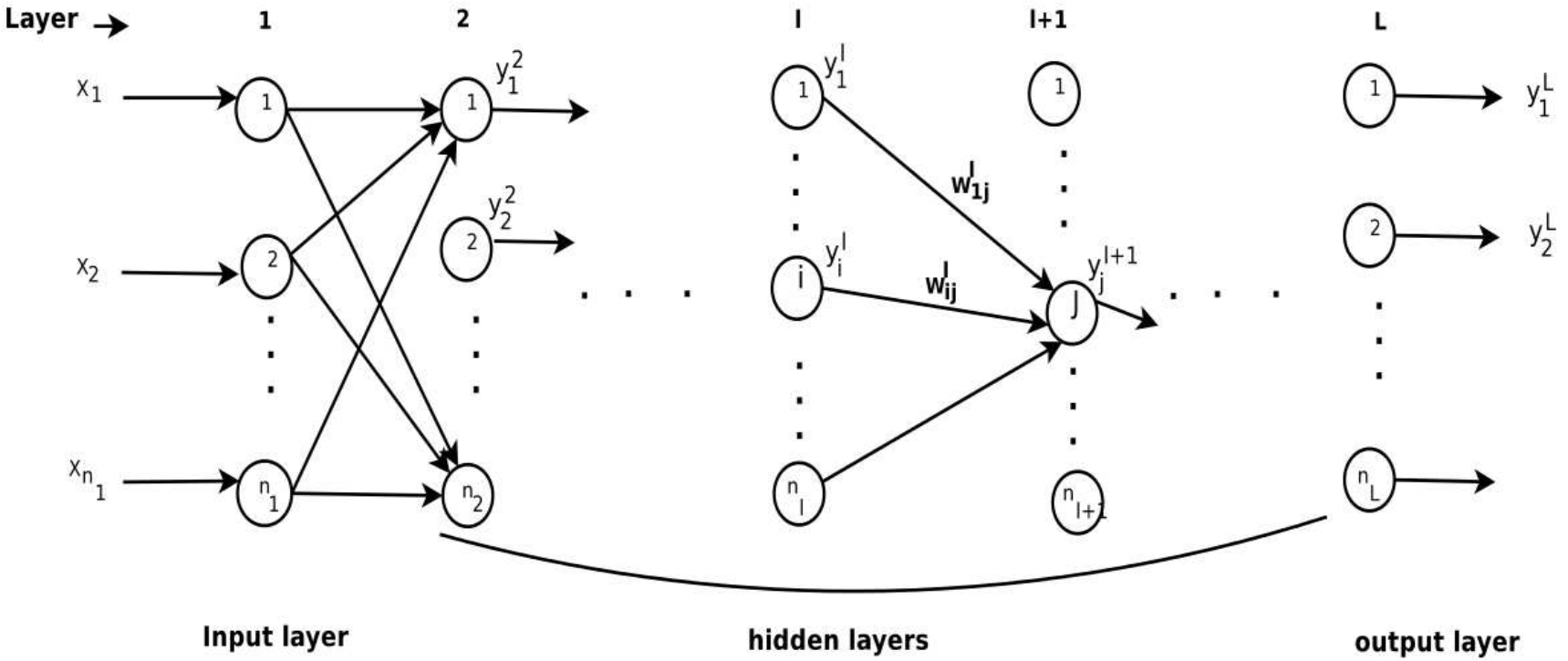
Recap

- We are discussing neural network models.
- We are looking at multilayer feedforward networks.
- These provide a parameterized class of nonlinear functions.

Recap – Multilayer feedforward networks



- We have an input layer, an output layer and one or more hidden layers.

Recap – Notation

- L – number of layers
- n_ℓ – number of nodes in layer ℓ, $\ell = 1, \cdots, L$.
- y_i^ℓ – output of i^{th} node in layer ℓ,
$$i = 1, \cdots, n_\ell, \ \ell = 1, \cdots, L.$$
- w_{ij}^ℓ – weight of connection from node-i, layer-ℓ to node-j, layer-$(\ell + 1)$.
- η_i^ℓ – net input of node-i in layer-ℓ
- Our network represents a function from $\Re^{n_1}$ to $\Re^{n_L}$.

- The network represents functions from $\Re^{n_1}$ to $\Re^{n_L}$.
- To get a specific function we need to learn appropriate weights.
- Thus,
  $w_{ij}^{\ell},\ i = 0, \cdots, n_{\ell},\ j = 1, \cdots, n_{\ell+1},\ \ell = 1, \cdots, L-1,$
  are the parameters to learn.
- Let W represent all these parameters.
- We learn W through empirical risk minimization with squared error loss function.

Recap – Learning the weights

- We want to learn w_{ij}^{ℓ} to minimize

$$J(W) = \sum_{i=1}^{N} J_i(W) = \sum_{i=1}^{N} \frac{1}{2} \left(\sum_{j=1}^{n_L} (y_j^L(X^i, W) - d_j^i)^2 \right)$$

- Using gradient descent for the minimization, we have

$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) - \lambda \sum_{s=1}^{N} \frac{\partial J_s}{\partial w_{ij}^{\ell}}$$

- For each training sample $(X^s,\ d^s)$, we first compute the output of network.
- Then we use the desired output to find errors at all nodes through backpropagation.
- Then we use these partial derivatives of J_s to update all weights (depending on batch or incremental mode of operation).

Computing output of network

- For the input layer: $y_i^1 = x_i^s,\ i = 1, \cdots, n_1$.
- For $\ell = 2, \cdots, L$, we now compute

$$\begin{aligned} \eta_j^\ell &= \sum_{i=1}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1} \\ y_j^\ell &= f(\eta_j^\ell) \end{aligned}$$

- The $y_1^L, \cdots, y_{n_L}^L$ form the final outputs of the network.
- Once we have the output of the network, we then need to compute the 'errors' δ_j^ℓ.

Backpropagation of Errors

- At the output layer

$$\delta_j^L = (y_j^L - d_j^s) f'(\eta_j^L)$$

- Now, for layers $\ell = (L-1), \cdots, 2$, we compute

$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} \, w_{js}^\ell \right) \, f'(\eta_j^\ell)$$

- Once all δ_j^ℓ are available, we update the weights by

$$w_{ij}^\ell(t+1) = w_{ij}^\ell(t) \; - \; \lambda \, \delta_j^{\ell+1} \, y_i^\ell$$

Recap

- Feedforward networks with one hidden layer are capable of approximating continuous functions on compact sets.
- The representation can be viewed as a linear model in terms of basis functions which are themselves learned from data.
- One can view learning a neural network as learning of proper (internal) representations.

- Neural network models are seen to be quite effective for both classification and regression.
- The backpropagation algorithm is quite effective in learning good representations.
- But to learn the appropriate weights, there are many parameters of the network that need to be chosen.
- Also, gradient descent can get stuck in local minima and the initialization could be crucial.
- We look at a few of the issues that are important for learning neural networks to be effective..

Art of Backpropagation

- To use a network for learning a function, we have to decide on many 'hyperparameters':
  - Number of hidden layers and hidden nodes (Network structure)
  - Activation function for nodes
  - The initial values for weights
  - Online or batch mode for learning
  - Learning algorithm
  - Loss function to be used

- We need to fix the structure of network before we can learn weights using backpropagation.
- The theorem we had says that one hidden layer is enough.
- But how many hidden nodes?
- In practice how many hidden layers, nodes?
- The VC-dimension of these models is of the order of number of weights plus nodes.
- Structure should not be too complicated relative to the number of examples we have.

- Our motivation is an analogy with the architecture of the brain.
- Most sensory information processing in Brain involves tens of layers.
- So, many hidden layers may be needed for good performance.
- The theorem does not say anything about the 'complexity' of representation with one layer.
- In recent times there is large interest in 'deep networks' that have many hidden layers.

Activation function

- What kind of activation functions to use?
- Theoretically we need smooth monotonically increasing functions.
- For gradient descent to work we need differentiability of activation function.
- Both sigmoid and tanh are suitable.
- There are other choices – RELU in deep networks

Online Vs Batch mode

- Should we do online or batch mode updates?
- If we have large number of examples, online is more convenient.
- Otherwise, we may need to do too much computation for small changes in weights.
- Also, we can alter the order of presentation of examples from epoch to epoch.
- Often helps in finding good minima.
- As we have seen, this is called stochastic gradient descent.
- Since we use constant step-size gradient descent, with small step-size online would also have good converge properties.

- In general we use minibatch. Then the hyperparameter is the size of minibatch.
- The minibatch method results in reduced variance of the stochastic gradient.
- The minibatch size is a balance between computation and variance in the gradient.

Normalizing the Inputs

- We also generally normalize the input (or feature) vectors.
- If different components of the input vectors, X^s, in the training set have widely differing range of values, we will get into numerical problems.
- We can use a linear transform to bring each feature value to $[-1,\ 1]$.
- Or we can transform each feature to be a zero-mean unit variance random variable.

- Let $X = (x_1, \cdots, x_m)^T$ be the feature vector.
- The training examples,
  $X^s = (x_1^s, \cdots, x_m^s)^T,\ s = 1, \cdots, N,$
  can be taken to be *iid*.
- We can estimate mean, μ_j, and variance, σ_j^2 of j^{th} feature as

$$\mu_j = \frac{1}{N} \sum_{s=1}^{N} x_j^s \quad \text{and} \quad \sigma_j^2 = \frac{1}{N} \sum_{s=1}^{N} (x_j^s - \mu_j)^2$$

- Now we can transform each example X^s to $\tilde{X}^s = (\tilde{x}_1^s, \cdots, \tilde{x}_m^s)^T$ by

$$\tilde{x}_j^s = \frac{x_j^s - \mu_j}{\sigma_j}$$

- This will make each component of $\tilde{X}$, to be a zero-mean unit variance random variable.
- This is often better than simply transforming the range of each feature component into $[-1, 1]$.

- We can also use some linear transform to decorrelate the feature components.
- Let $S = \frac{1}{n} \sum_{i=1}^{n} (X^i - \bar{X})(X^i - \bar{X})^T$ be the data covariance matrix (whose dimension is $m \times m$).
- Let $\lambda_1, \cdots, \lambda_m$ be the eigenvalues of S arranged in a decreasing order.
- Let $U_1, \cdots, U_m$ be the corresponding eigen vectors (which are orthonormal).
- Let L be a diagonal matrix with λ_i being the diagonal entries.
- Let $\tilde{U}$ be a $m \times m$ matrix whose columns are U_j.

- Now the eigen value equations for S are

$$S\tilde{U} = \tilde{U}L$$

- Now define a transformation of the data vectors given by

$$Z^i = L^{-0.5}\tilde{U}^T(X^i - \bar{X})$$

  where $\bar{X}$ is the mean of the data vectors.
- It is easy to see that mean of Z^i is zero:

- The covariance matrix for the data Z^i is now given by

$$\begin{aligned} S_Z &= \frac{1}{n}\sum_{i=1}^{n} Z^i \, (Z^i)^T \\ &= \frac{1}{n}\sum_{i=1}^{n} L^{-0.5} \, \tilde{U}^T \, (X^i - \bar{X}) \, (X^i - \bar{X})^T \, \tilde{U} \, L^{-0.5} \\ &= L^{-0.5} \, \tilde{U}^T \, S \, \tilde{U} \, L^{-0.5} \\ &= L^{-0.5} \, \tilde{U}^T \, \tilde{U} \, L \, L^{-0.5} \\ &= I \end{aligned}$$

- Thus, if we transform X^i to Z^i by

$$Z^i = L^{-0.5}\tilde{U}^T(X^i - \bar{X})$$

  then the transformed data are zero-mean, unit variance and uncorrelated.
- This may be useful in many pattern classification problems (e.g., for naive Bayes).
- This is called the **whitening transform**.

The Learning Algorithm

- Backpropagation is a gradient descent in a very high dimensional space.
- Hence it has all problems associated with such gradient descent.
- It gets stuck in local minima. It is also generally slow.
- Good initialization helps a lot.
- One can use 'multiple starts'
- There are some more ways to improve this.

Backpropagation with momentum term

- One often uses a so called momentum term and writes the algorithm as

$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) - \lambda \frac{\partial J}{\partial w_{ij}^{\ell}}(t) \; + \; \gamma \, \Delta w_{ij}^{\ell}(t-1)$$

  where $\Delta w_{ij}^{\ell}(t-1) = w_{ij}^{\ell}(t) - w_{ij}^{\ell}(t-1)$.
- At each iteration, we add a small term which is proportional to the direction in which we moved in the previous iteration.

- Gives a flavour of conjugate gradient search to the algorithm.
- If λ, γ are sufficiently small, this algorithm also converges to local minima.
- In practice, this considerably speeds up the algorithm.
- One normally always uses backpropagation with momentum.

- We can rewrite the momentum based algorithm as follows.
- Let g_t denote the gradient vector at iteration t. This would be gradient of J_t if we are using stochastic gradient descent.
- Now we write the gradient descent as

$$W_{t+1} = W_t - \eta g_t$$

- When we want to write it component wise we will use g_t^{ij}.

- Now we can write the algorithm with momentum term as follows.

$$\begin{aligned} m_t &= \beta m_{t-1} + (1-\beta) g_t \\ W_{t+1} &= W_t - \eta m_t \end{aligned}$$

- Note that m_t contains some 'history' of all past gradients.
- This can make the learning more focussed.
- Using the general structure of the above equations, many variations of backpropagation are possible.

ADAM Algorithm

- This is a recent method that uses the 'moving averages' of the gradient.
- Essentially we use some average of recent gradients for finding the descent direction and use recent averages of squares of gradient to scale the step-size.

ADAM Algorithm

- As earlier, let g_t be the gradient vector at iteration t with g_t^{ij} being gradient with respect to w_{ij}.
- We iteratively compute

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\nu_t = \beta_2 \nu_{t-1} + (1 - \beta_2) g_t^2$$

  where g_t^2 denotes element-wise squaring of the gradient vector.
- These are essentially 'moving averages'.
- We start with $m_t = \nu_t = 0$. The β_i are parameters.

ADAM Algorithm

- We now do gradient descent as

$$w_{ij}(t) = w_{ij}(t-1) - \frac{\eta}{\sqrt{\nu_t^{ij} + \epsilon}} m_t^{ij}$$

- We are essentially using the 'averaged gradient' as in the momentum algorithm.
- The step-size is automatically scaled.
- It is invariant to scaling the objective function by a constant.
- Improves speed of convergence.

Weight Decay

- Another simple strategy that one uses in the learning algorithm is the so called weight decay.
- From time to time we replace each weight w_{ij}^{ℓ} with $w_{ij}^{\ell}(1-\epsilon)$ where ϵ is a small positive number.
- We can do this every iteration, or after a fixed number of iterations and so on.
- This is essentially implementing gradient descent on a 'regularized' risk.

- We are trying to minimize J, sum of squared errors.
- Suppose we decide to minimize

$$\tilde{J} = J \ + \ 0.5\,\epsilon\,||W||^2$$

- This is empirical risk minimization with a regularization term
- Now gradient descent would be

$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) \ - \ \lambda \frac{\partial J}{\partial w_{ij}^{\ell}} \ - \ \epsilon w_{ij}^{\ell}$$

- This corresponds to the so called weight decay.

- Sometimes this weight-decay regularization is also implemented as a constraint of the magnitude of weights.
- We choose some constant K and for each node constrain the norm of the weight-vector consisting of all incoming links to that node, to be less than K.
- That is, after update, we project the weight vector into the feasible region.
- This allows one to often use a higher learning rate (step-size)

drop-out

- Another important regularization used with deep networks is the so called drop-out.
- In drop out regularization, one keeps dropping a random subset of nodes in the network from being considered.
- In each iteration, for each node (except possibly for the input nodes) we independently decide with probability, p, whether that node (along with all its incoming and outgoing links) would be present.
- The backpropagation would update only those links that are present.
- After learning, we use the full net.

Step-size in the Learning Algorithm

- The step-size used in backpropagation is crucial in achieving a proper trade-off between accuracy and speed.
- Ideally, one should use a step-size based on line search for gradient descent to work well.
- But generally, backpropagation uses constant step-size gradient descent. (Can be choosen using cross-validation).
- There are also heuristics for changing step-size as learning progresses. (Normally, one decreases step-size as learning progresses).

Optimization methods

- To learn the weights in a feedforward network we are minimizing empirical risk under squared error loss.
- Gradient descent and hence backpropagation is only one method to do such optimization.
- We can use other methods, such as Newton's method or quasi-Newton methods.
- We need second order partial derivatives here.
- A backpropagation like procedure can be derived for getting all second order partial derivatives.
- Many such optimization techniques are used to learn appropriate weights in these network models.

Choice of loss function

- So far we have only considered squared error loss.
- We can consider other loss functions also.
- We can look at it as choice of a 'discriminative model'

- The feedforward networks we are considering are discriminative models.
- We are using them for nonlinear regression.
- We saw that linear least squares regression can be viewed as ML estimation of discriminative models.
- Similarly, here also we are estimating discriminative models.
- The model we are fitting, depends on the loss function.

- Our training data is $\{(X^i, d^i)\}$
- For a discriminative model, we want to learn $p(d|X)$.
- We can use ML estimation from the given data, by choosing a suitable model for $p(d|X)$.

- For a network with weights W, let $y^L(W, X)$ denote the output for input X.
  (From now on we will omit superscript, L).
- Suppose we assume a density model

$$p(d|X, W) = \mathcal{N}(y(W, X), \sigma I)$$

- We are assuming independent zero-mean Gaussian noise in measured data.
- Then it is easy to see that the ML estimate would be minimizer of squared error loss.

- The likelihood is (for data $\{(X^i, d^i)\}$)

$$l(W|\text{data}) = K \exp\left(\sum_i \frac{-1}{2\sigma^2}||y(X^i, W) - d^i||^2\right)$$

- Maximizing this is same as minimizing squared error.
- As discussed earlier, in a regression problem Gaussian noise is a reasonable assumption.
- For a classification problem, we can use a different probability model and hence a different loss function.

- Let d_j denote the j^{th} component of the output.
- In the training data, the desired output is a vector with one component 1 and all others zero.
- Hence a good model for conditional density is

$$p(d|X) = \prod_j (q_j(X))^{d_j}$$

where $q_j(X)$ is posterior probability of class-j for X.

- The network output, $y(W, X)$, is the (estimated) posterior probability vector for any given weights W.
- Hence, a parameterized class of discriminative models is

$$p(d|X, W) = \prod_j (y_j(W, X))^{d_j}$$

- Hence the log-likelihood function given the data is

$$l(W|\text{data}) = \ln \left(\prod_i p(d^i|X^i, W) \right) = \ln \left(\prod_i \prod_j (y_j(W, X^i))^{d_j^i} \right)$$

- We need to maximize it over W.
- We can minimize negative log-likelihood

- For this to be a proper model, the output, $y(W, X)$ should be a probability vector.
- This would be so, if we have a softmax output layer.
- Neural network nodels for classification use a softmax output layer.

- The negative log-likelihood, say, $J(W)$, is

$$J(W) = -\ln\left(\prod_i \prod_j (y_j(W, X^i))^{d_j^i}\right) = -\sum_i \sum_j d_j^i \ln(y_j(W, X^i))$$

- This is empirical risk under the loss function

$$L(h(X), d) = L(y, d) = -\sum_j d_j \ln(y_j)$$

- This is called cross entropy loss.
- This is the loss that is often used for classification problems.

- The cross entropy loss is given by

$$L(y, d) = -\sum_j d_j \ln(y_j)$$

- Recall that d is a 'one-hot' vector.
- Hence, on a training sample of class k, if output is the vector y then the loss is $-\ln(y_k)$.

- How does the backpropagation algorithm change when one chages the loss function?
- As we saw earlier, only the computation of 'error' at the output nodes changes.
- That is, only computation of δ_j^L changes.
- Once all δ_j^L are calculated, rest of the backpropagation is same.

- We know that ML estimation is related to minimizing KL divergence.
- By using a softmax output layer, we are considering the output of the netwrk as an estimate of posterior probability.
- This is a discriminative model parameterized by W.
- The risk minimizer under cross entropy loss would be a distribution, parameterized by W, that minimizes KL divergence from the true posterior probabilities.

- Consider a 2-class case.
- Instead of 2-node softmax output layer, we can have a single output node with sigmoidal activation.
- we can have $d \in \{0,\ 1\}$.
- The output represents (estimated) posterior probability for class-1.

- For the two class case, cross entropy loss is

$$L(y(W,X),d) = -d\log(y(W,X)) - (1-d)\log(1-y(W,X))$$

- Consider two distributions of binary random variables: $p(1) = q(X)$ and $p'(1) = y(W,X)$
- So, KL divergence between them would be

$$-p(1)\ln\left(\frac{p'(1)}{p(1)}\right) - p(0)\ln\left(\frac{p'(0)}{p(0)}\right)$$

which is same as

$$-q(X)\ln(y(W,X)) - (1-q(X))\ln(1-y(W,X)) + K$$

where K is a term independent of W.

- We can see the utility of cross entropy loss from an optimization point of view also.
- Let $\eta(W, X)$ be net input to the output node that uses sigmoid activation. $(y(W, X) = f(\eta(W, X)))$
- If L is squared error loss, then

$$\frac{\partial L(y(W,X),d)}{\partial w_{ij}} = (y(W,X)-d)y(W,X)(1-y(W,X))\frac{\partial \eta(W,X)}{\partial w_{ij}}$$

(For a sigmoid function, f, $f'(t) = f(t)(1 - f(t))$).

- Suppose for a specific X, $d = 1$ and the current W is such that $y(W, X)$ is close to zero.
- Then the error $(y(W, X) - d)$ is large but the gradient is very small.
- Makes learning very slow and difficult.

- Now consider the cross entropy loss

$$L(y(W,X),d) = -d\log(y(W,X)) - (1-d)\log(1-y(W,X))$$

- Here we get

$$\frac{\partial L(y(W,X),d)}{\partial w_{ij}} = [-d(1-y(W,X)) - (1-d)y(W,X)]\frac{\partial \eta}{\partial w_{ij}}$$

- As is easy to see, here when we have large error, the gradient magnitude is also large.
- This is another way of looking at utility of cross entropy loss.

Art of Backpropagation

We started with the following list of issues

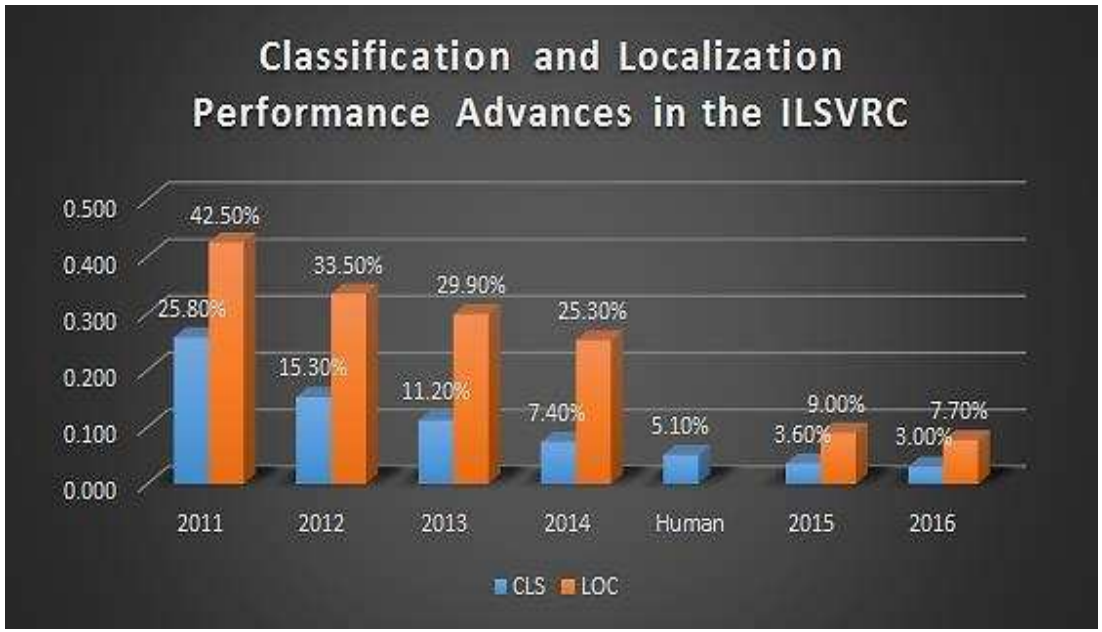
- Number of hidden layers and hidden nodes (Network structure)
- The initial values for weights
- Activation function for nodes
- Online or batch mode for learning
- Normalization of inputs
- Learning algorithm
- Loss function to be used

Network Structure

- Our motivation is an analogy with the architecture of the brain.
- Most sensory information processing in Brain involves tens of layers.
- So, many hidden layers may be needed for good performance.
- In the last 5-10 years, many interesting neural networks with large number of hidden layers are investigated.
- There is large interest in such 'deep networks' that have many hidden layers.

Performance of deep networks

Over the years deep networks have delivered very good performance:[1]



[1] Taken from: Kaplanoglou, Pantelis. (2017). Content-Based Image Retrieval using Deep Learning. 10.13140/RG.2.2.29510.16967

Deep Networks

- The theory says that a network with one hidden layer with 'sufficiently large' number of nodes can approximate any continuous function.
- However, theory does not say anything about complexity of this representation or efficiency of learning.
- Deep networks are seen to be much more effective in many applications.
- However, there are difficulties in training deep networks.

Deep Networks

- Backpropagation is gradient descent in high dimensional space.
- With deep networks, the number of weights to be learnt becomes huge.
- So, we need some way to control increase of weights.
- Or we need some way to ensure that we reach good local minima.

- So far, we looked at networks that are fully connected.
- Thus, weights would scale as square of the nodes in layers.
- With many hidden layers number of weights also becomes very huge.
- One way to control this is to use only 'local connections'.
- Convolutional Neural Networks (CNNs) represent this approach which has proved to be highly successful in many pattern recognition tasks, especially in image processing.

Initial Values of Weights

- Another factor that affects the performance of gradient descent is the initialization of weights.
- Good initial values of weights can take us to better local minima
- Small random values for initialization is generally better.
- Normally one uses random initial weights drawn from a distribution with mean zero and variance $1/m$ where m is the indegree of the node to which this weight connects.

Initialization of Weights

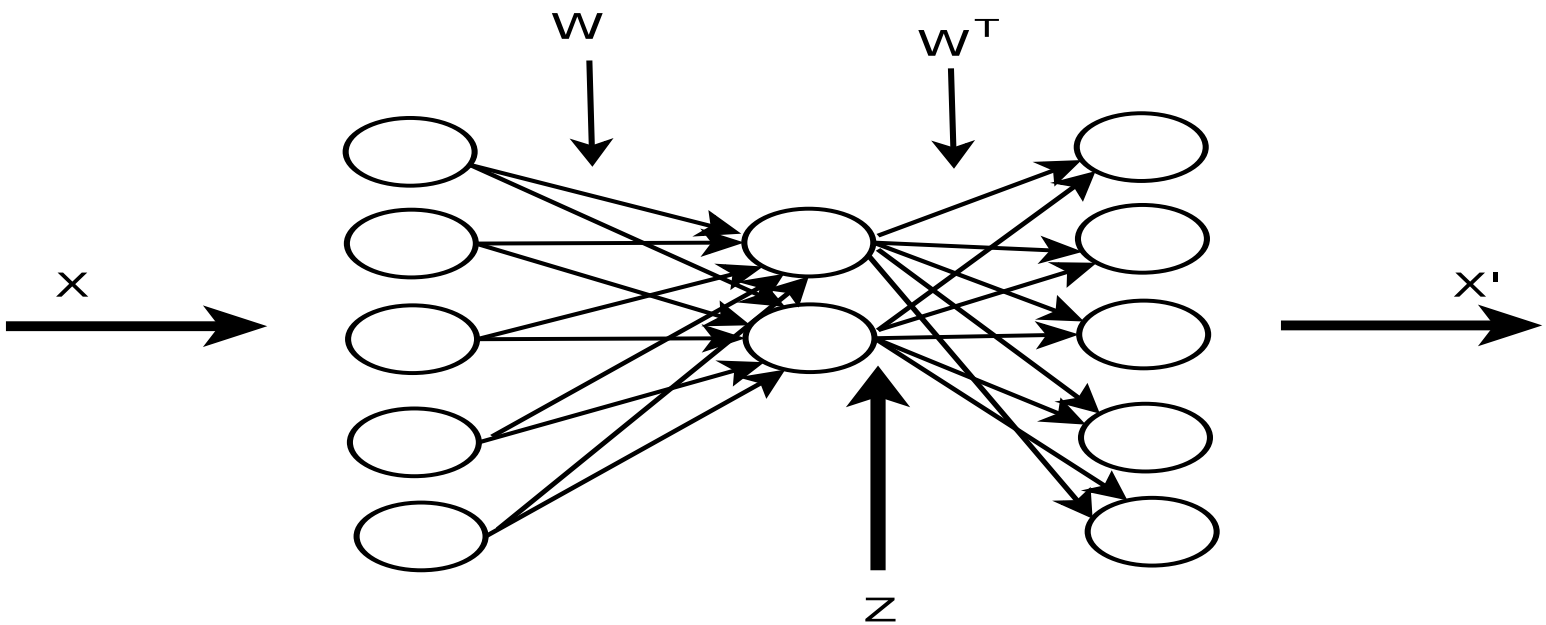
- Net input to a node: $\sum_{i=1}^{m} w_i x_i$.
- Suppose inputs are zero-mean unit variance. Suppose initial weights are also independent zero-mean random variables with variance σ_w^2.
- Then, variance of net input scales as: $m\sigma_w^2$.
- So, if $\sigma_w^2 = 1/m$, then $[-1,\ 1]$ will span one standard deviation from mean for net input.
- This will keep the sigmoid function away from saturation and thus gives good values for intial gradients.

Unsupervised Learning to Initialize Weights

- One of the insights that contributed to current success of deep networks is that we need good initialization **based on data**.
- One possible way to realize deep learning is to obtain good initial weights through unsupervised learning.
- Two important developments here gave thrust to deep learning:
  - Autoencoders to initialize multilayer feedforward nets.
  - Restricted Boltzman machines to initialize Deep belief networks.

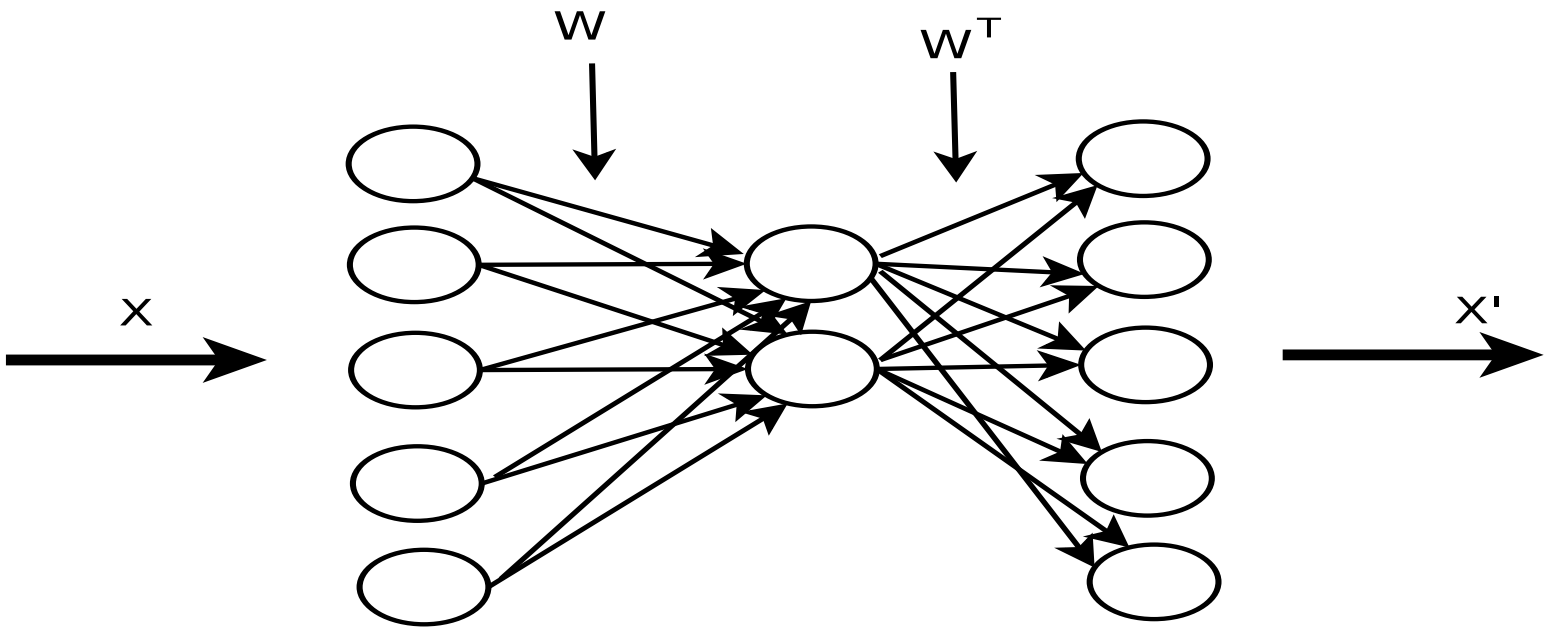
Autoencoder Network

It is a 3-layer feedforward network with very few hidden nodes.



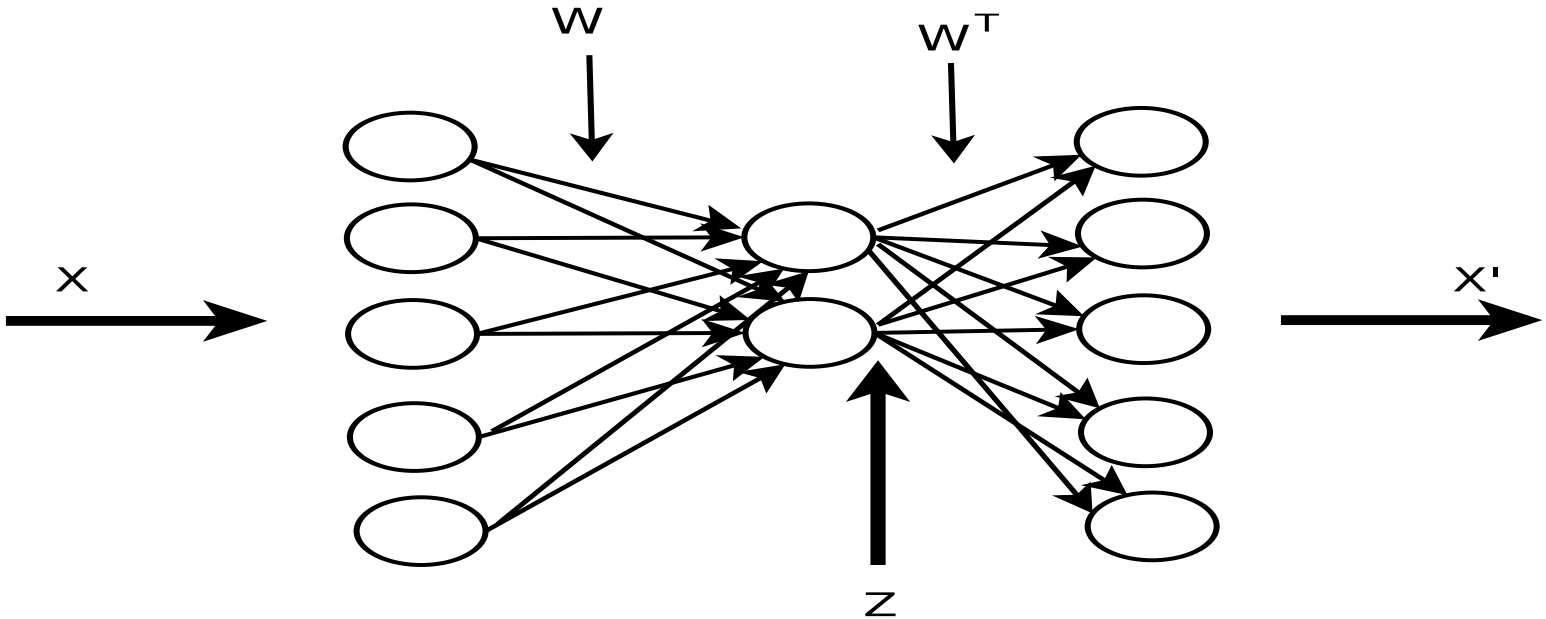
$$
\begin{aligned}
Z &= f(WX + b) \\
X' &= f(W^T X + b')
\end{aligned}
$$

Autoencoder Network



- We want to learn W such that $X' = X$.
- The W is good for transforming the representation.
- The motivation is from Minimum Description Length principle

Autoencoder Network



- Given data, $\{X^1, \cdots X^m\}$, we can learn W through backpropagation.
- This is unsupervised learning.
- We learn a 'compressed' representation.
- The representation may be 'good' because we can recreate original X.
- Like dimensionality reduction. (Using W and W^T is motivated by the linear case)

- There are many variations possible on this basic theme.

Denoising Autoencoder

- We would give noise-corrupted X at input but want X' to be the 'clean' X.
- Can add independent noise to each component of X.
- But, what is often done is to make a few randomly selected components of X zero.
- If we can learn W to create X at output, then that W can capture dependences among components of X.
- Hence, W is a good set of weights to transform X into a useful representation.

Another Variation: Sparse Autoencoder

- We are learning a 'compressed' representation by having only few nodes in hidden layer.
- Alternately, only a few of the hidden nodes should be 'active' for any given X.
- Then we need not have any restriction on the number of hidden layer nodes
- By making representation 'sparse' we achieve similar 'compression'
- Sparsity can be incorporated into the objective function.

Ensuring Sparsity

- Let X^i be the i^{th} example, $i = 1, \cdots, m$. Let

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} y_j^2(X^i)$$

  (y_j^2 is the output of j^{th} node in the hidden layer)
- $\hat{\rho}_j$ is the fraction of input for which j^{th} hidden neuron is 'ON'.
- We want $\hat{\rho}_j = \rho$, $\forall j$, where ρ is the sparsity parameter.
- Typically ρ is very small (e.g., 0.05).

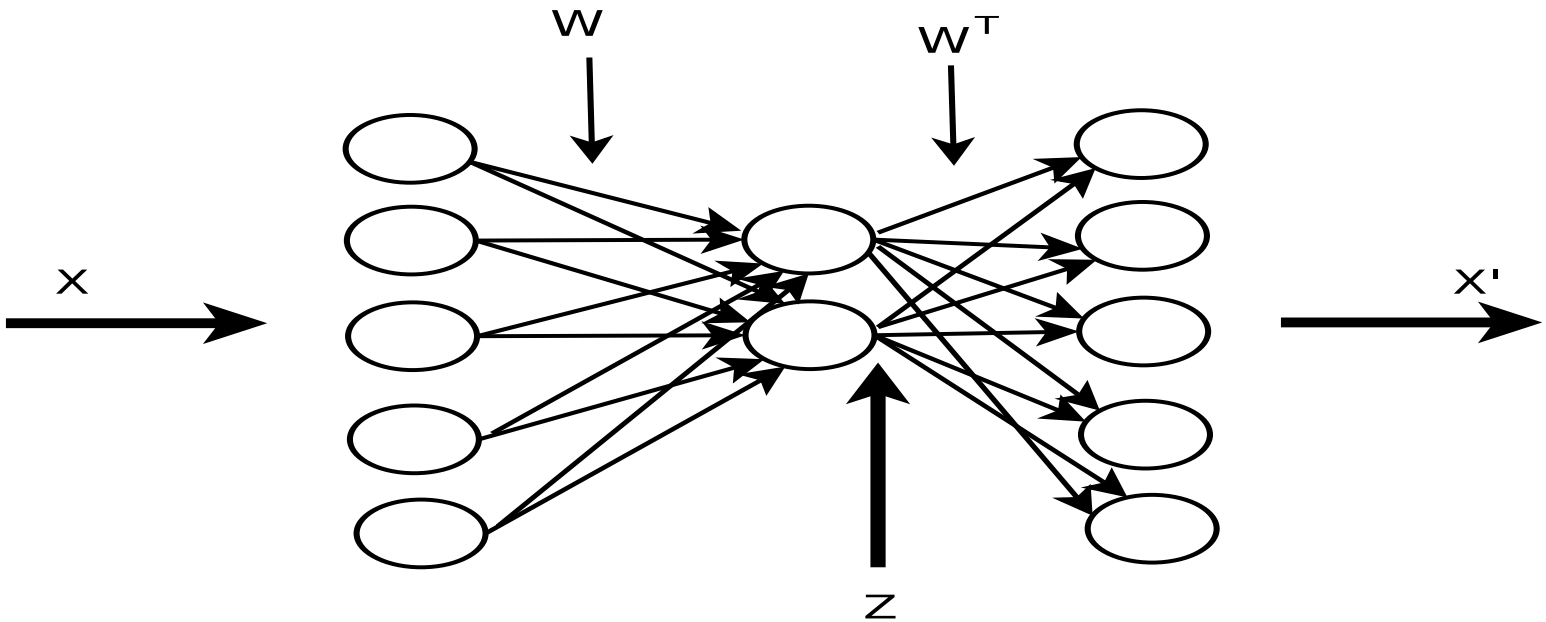
- We can think of ρ as a bernoulli parameter.
- We can use KL divergence to measure deviation of ρ_j from ρ.

$$KL(\rho||\rho_j) = \rho \log\left(\frac{\rho}{\rho_j}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\rho_j}\right)$$

- We can use the following objective function

$$J(W) = \sum_{i=1}^{m} ||y^3(X^i) - X^i||^2 + \beta \sum_{j=1}^{n_2} KL(\rho||\rho_j)$$

$$J = \sum_{i=1}^{m} ||y^3(X^i) - X^i||^2 + \beta \sum_{j=1}^{n_2} \rho \log\left(\frac{\rho}{\rho_j}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\rho_j}\right)$$



- The ρ_j depend on y_j^2 but not on w_{ij}^2.
- Hence δ_j^3 would be same as earlier.
- We only need to calculate $\frac{\partial J}{\partial w_{ij}^1}$.

- It can be shown that

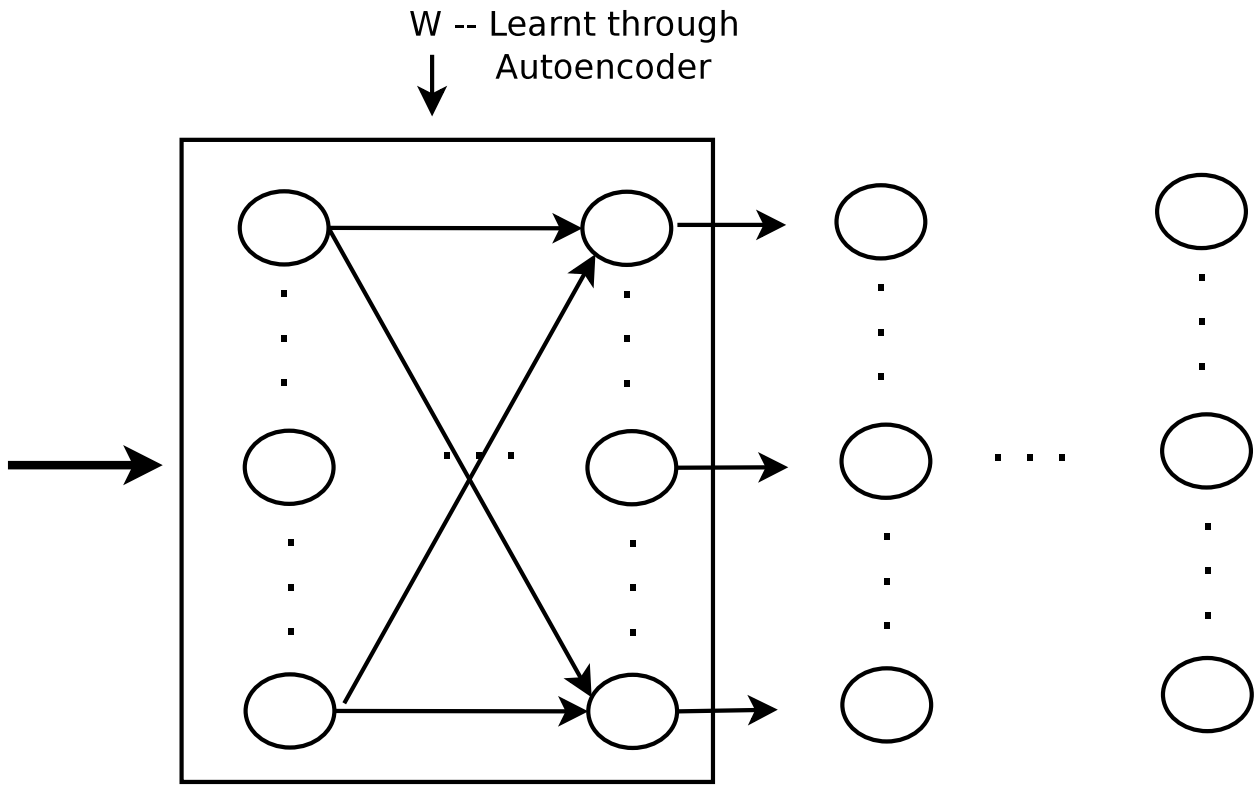
$$\delta_i^2 = \left(\sum_{j=1}^{n_3} w_{ij}^2 \delta_j^3 + \beta \left(\frac{-\rho}{\rho_i} + \frac{1-\rho}{1-\rho_i} \right) \right) f'(\eta_i^2)$$

- Note that we have $\frac{\partial J}{\partial w_{ij}^1} = \delta_j^2 y_i^1$.
- We need current value of ρ_j for weight update – need $y_j^2(X^i)$ for all i.
- Need one pass over examples before starting weight update.

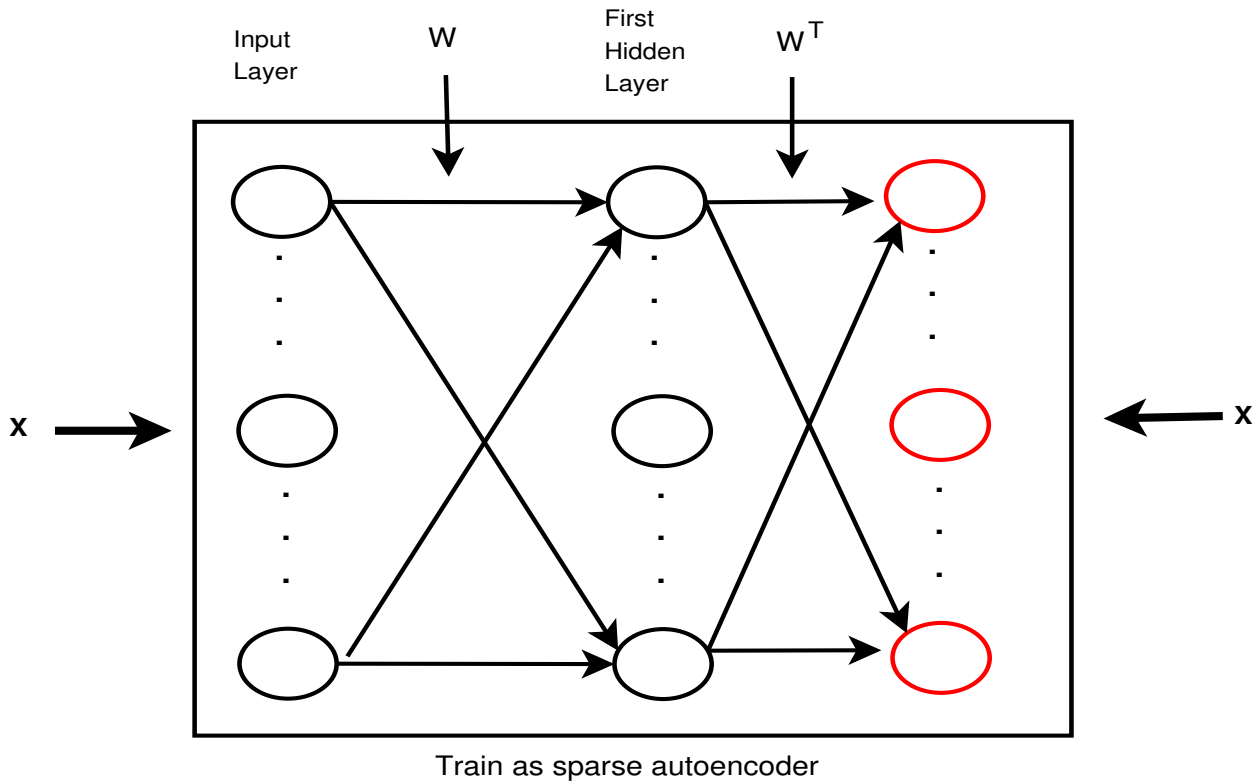
- We can have the sparsity penalty in a denoising autoencoder too.
- So, we get sparse denoising autoencoders.
- We can use the autoencoder network to initialize weights successively in each layer of a feedforward network.

Autoencoder for initializing weights

- We will first learn the weights from input layer to first hidden layer using (sparse denoising) autoencoder. (This is unsupervised learning)

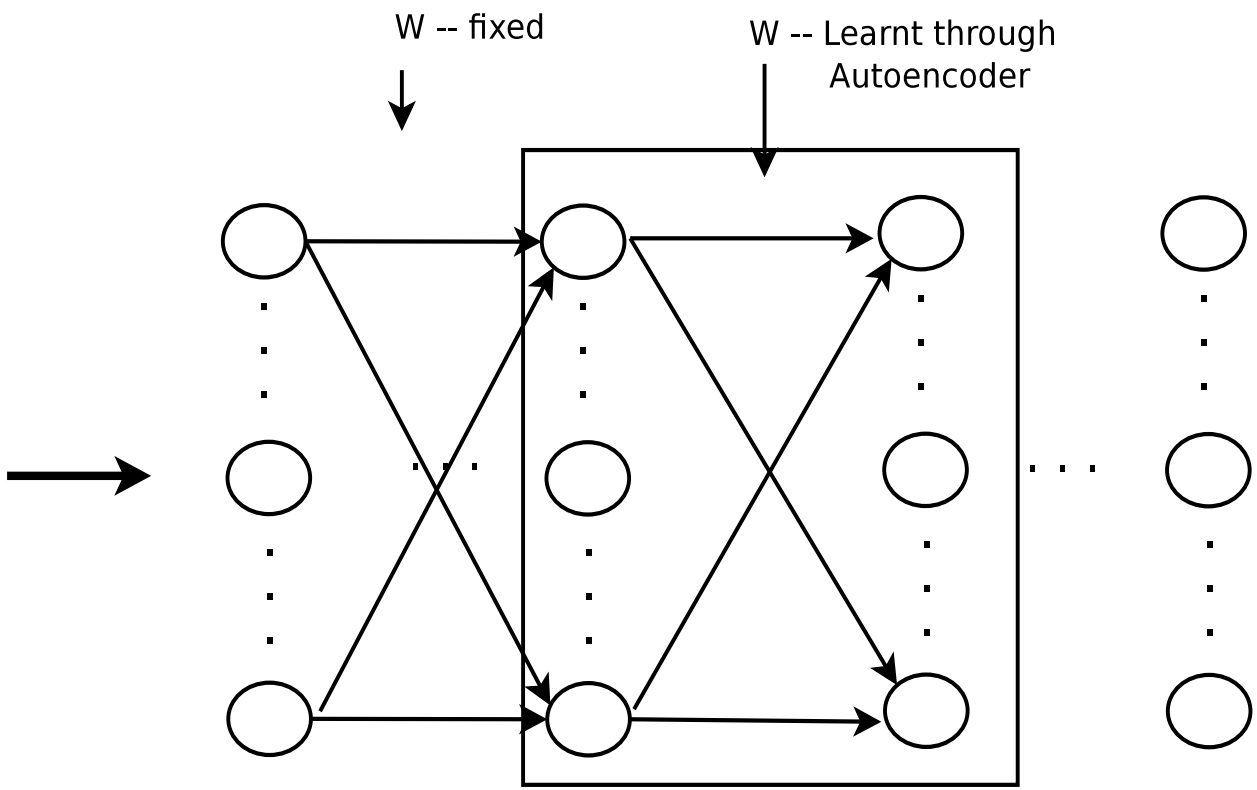


We learn the first layer weights as below:



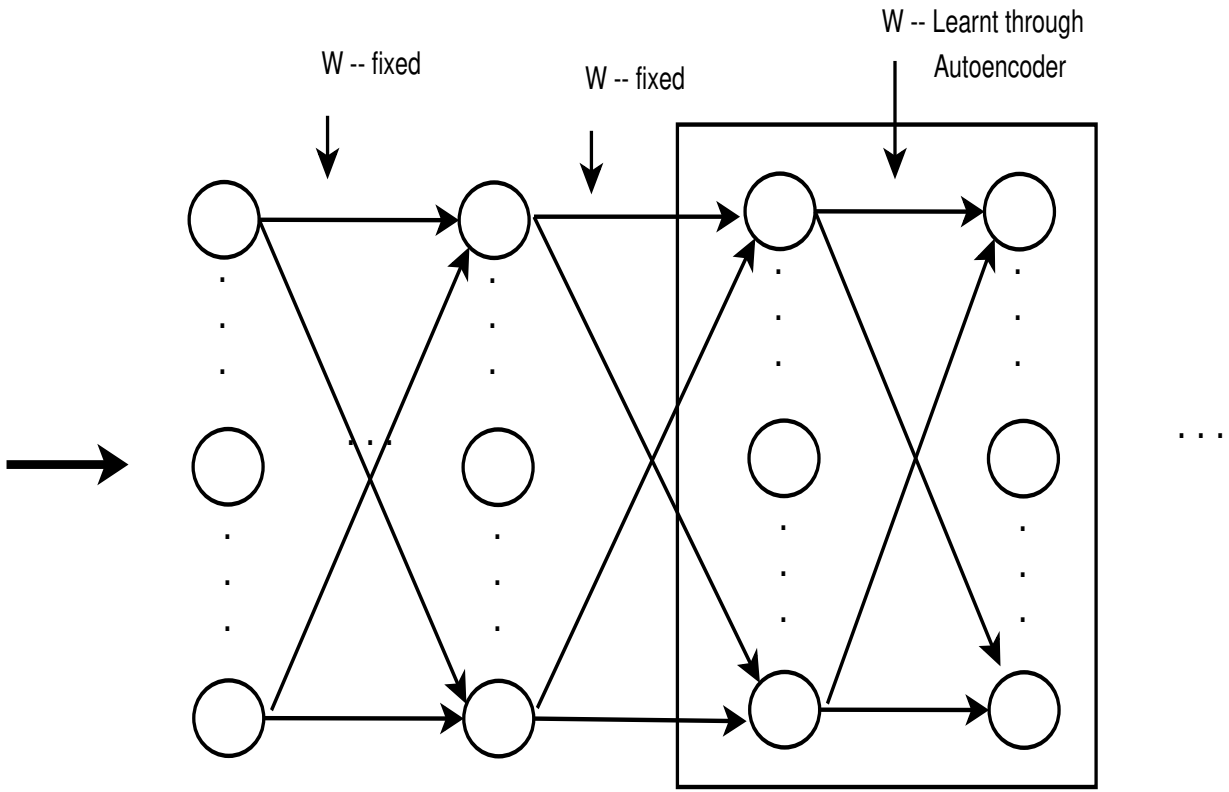
- Here we have no control on the number of hidden nodes in the autoencoder network,
- Hence we use sparse autoencoder.

Next we learn the weights from first hidden layer to second hidden layer using autoencoder.



- Input to autoencoder are trainig data X^i transformed through the W that is already learnt.

Following this procedure we learn weights for each successive layer.



- The basic idea here is the following.
- Using one hidden layer at a time we learn weights to generate a new representation of the input.
- The weights are learnt to recreate the input through a sparse autoencoder.
- Such weights form a good (and data-dependent) initialization of weights in the network.
- Starting with this initialization, we use supervised learning through backpropagation to learn all the weights in the network.
- This idea of using autoencoder for initializing all weights for deep network is very effective.
- In many applications this allows backpropagation to learn 'good local minima' even for very deep networks.