Recap – Some Issues in learning neural networks

- Activation function for nodes
- Online or batch mode for learning: fixing minibatch size
- Normalization of inputs
  - convert individual features to mean zero variance 1,
  - whitening transform
- Learning algorithm
  - BP with momentum
  - ADAM algorithm
  - Weight decay, dropout, step-size selection
  - other optimization methods (e.g., Hessian based)
- Loss functions (e.g., cross entropy loss)
- Network structure
  - Number of hidden layers and hidden nodes
  - The initial values for weights

Recap

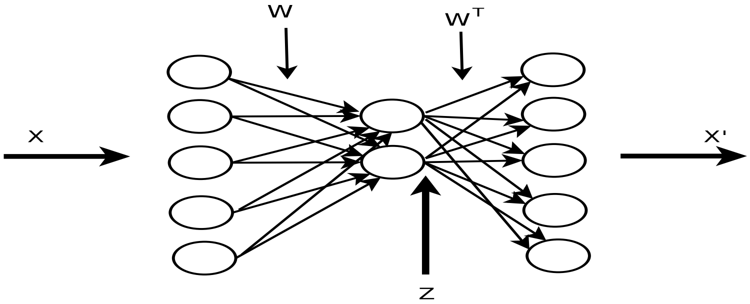
- Over the last 5-10 years there is increasing interest in deep networks – networks with large number of hidden layers.
- Deep networks are shown to be highly effective in applications.
- Learning deep networks through gradient descent can be challenging because of the large number of weights.
- One approach is to use 'local' connectivity patterns (e.g., CNNs)
- Another approach is to have good data-dependent initializations of weights.

Unsupervised Learning to Initialize Weights

- One of the insights that contributed to the sucesses of learning 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.

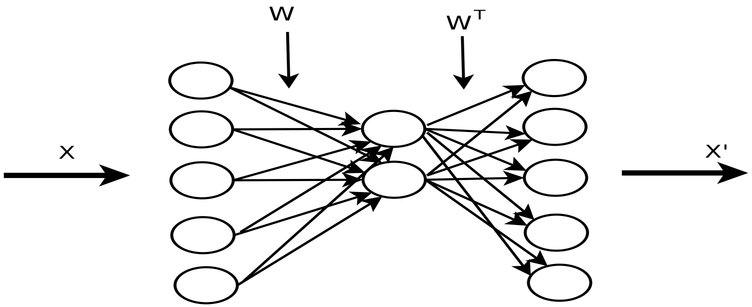
Autoencoder Network

- It is a feedforward network that was investiagted in 1980's.
- It is a 3-layer feedforward network with very few hidden nodes.



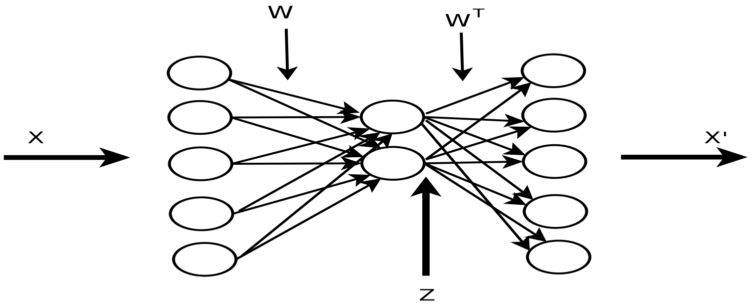
$$
\begin{aligned}
Z &= f(WX + b) \\
X' &= f(W^T X + b')
\end{aligned}
$$

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).

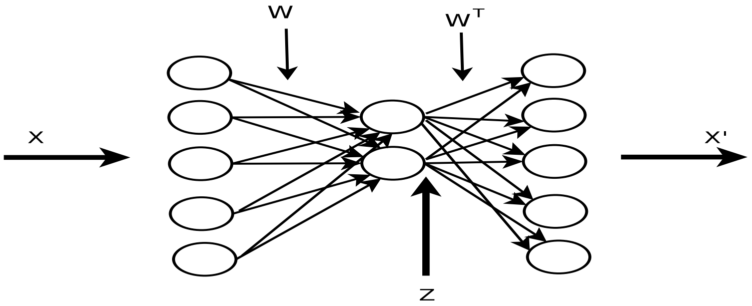
- We can think of ρ as a bernoulli parameter.
- We can use KL divergence to measure deviation of ρ_j from ρ.

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

- 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

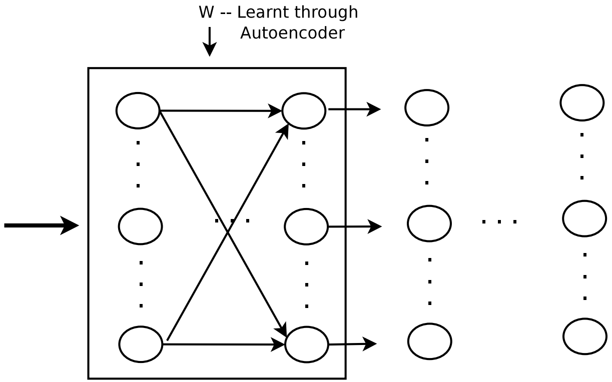
$$\delta_i^2 = \left(\sum_{j=1}^{n_3} w_{ij}^2 \delta_j^3 + \beta \left(\frac{-\rho}{\rho_i} + \frac{1-\rho}{1-\rho_i} \right) \right) f'(\eta_i^2)$$

- 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.

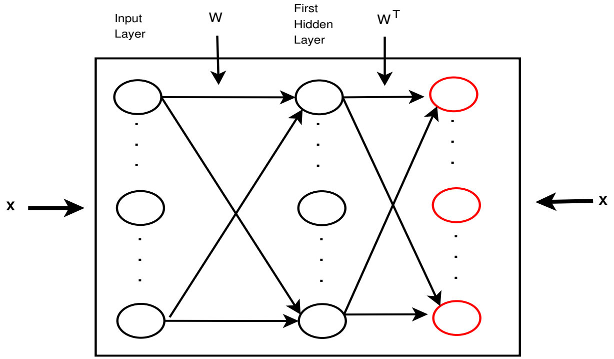
- We can have the sparsity penalty in a denoising autoencoder too.
- So, we get sparse denoising autoencoders.
- We can use the autoencoder network to initialize weights successively in each layer of a feedforward network.

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)



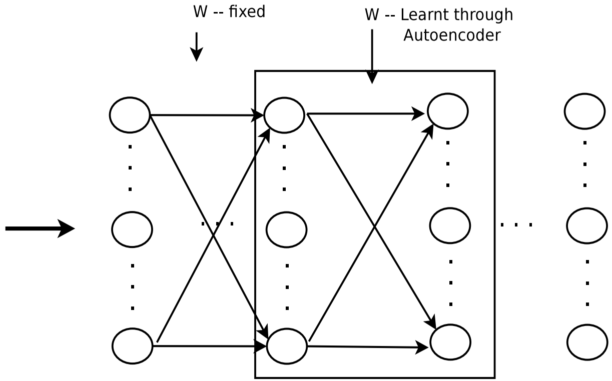
We learn the first layer weights as below:



Train as sparse autoencoder

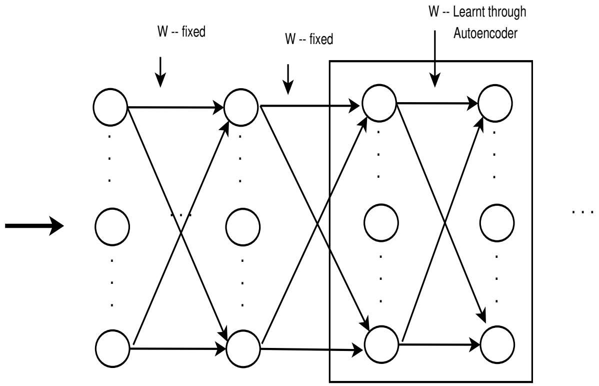
- Here we have no control on the number of hidden nodes in the autoencoder network,
- Hence we use sparse autoencoder.

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.

Convolutional Neural Networks (CNNs)

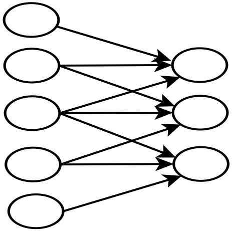
- CNNs represent a deep network model that has revolutionized image-recognition
- CNNs are largely responsible for starting the current wave of interest in deep neural networks.

Convolutional Neural Networks (CNNs)

- CNNs are deep neural networks that are originally proposed for image recognition.
- The input layer of a CNN is 2-dimensional because it is an image.
- There are many features of CNNs that make them much more efficient compared to normal feedforward nets for image based pattern recognition.
- They use local connectivity, weight sharing, multiple 'feature planes' to learn appropriate features from data.
- We start by looking at the ideas undelying CNNs first in a one dimensional context.

Connectivity is local

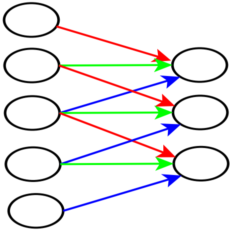
- In fully connected networks weights in a layer grow as square of number of nodes.
- We can reduce the number of weights by making connections local.



- Now weights scale only linearly with number of neurons.

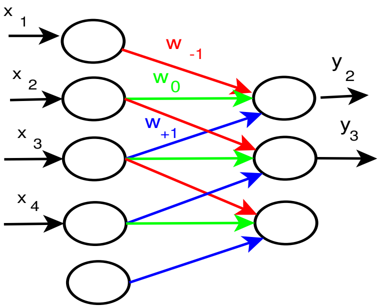
Weight-Sharing

- We can reduce weights further by 'sharing' of weights.



- Now number of weights per layer is a constant independent of the number of nodes.
- Such layers are called convolutional layers.

Output of a convolutional node



- The output can be calculated as

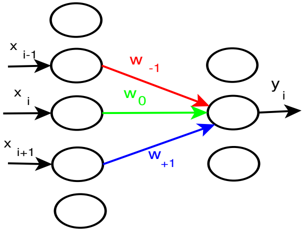
$$y_2 = f\left(w_{-1}x_1 + w_0x_2 + w_{+1}x_3\right)$$

$$y_3 = f\left(w_{-1}x_2 + w_0x_3 + w_{+1}x_4\right)$$

- Essentially a convolution type computation

Output of a convolutional node

- The general case.



$$y_i = f\left(\sum_{k=-1}^{+1} w_k x_{i+k}\right)$$

- Weight vector is a 3-point filter.
- We are computing the filter output at each point by sliding over the input.

CNNs

- Each layer of a network with such connectivity is called a convolutional layer.
- Convolutional neural networks (CNNs) are feedforward networks that contain many such convolutional layers.

- In a CNN, the output of the convolutional layer is passed through a non-linear activation function.
- The often used activation function is ReLU.

CNNs

- What are typical problems where such connectivity is natural?
- Example: Image-based pattern recognition
- Useful features in an image (e.g., edges, corners) are computed using such local convolution through so called masks.
- We apply the same operation at all points in an image to detect the feature wherever it exists.

- For example, this is a simple edge-detector mask

-1	0	1
-1	0	1
-1	0	1

- We do this masking (or convolution) operation at each point in the image

- Traditionally, in Pattern Recognition, feature extraction and classification are viewed as two separate steps.
- Often features are designed separtely based on the knowledge of the problem.
- For example, the SIFT or HOG features used in image recognition problems.
- After transforming the input image into a representation using the chosen features, one learns a (linear or nonlinear) classifier.
- As discussed earlier, the philosophy underlying the neural networks approach is that we should **automatically learn** the relevant features based on the data.

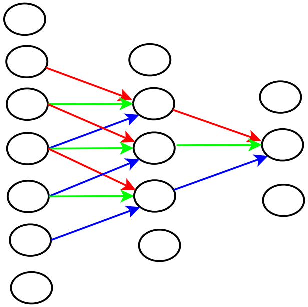
- Each convolutional layer is essentially detecting a feature.
- Since the weights would be learnt, we are learning the ‘proper features’ automatically using the training data.
- So far we are considering 1D layers.
- Image is two dimensional and hence all layers as well as the filters need to be two dimensional.
- But this is a straight-forward extension as we shall see.

Multiple Convolutional Layers

- We need to combine simple features into more complex features to achieve object recognition.
- We may combine edge pixels at different locations into lines before we can recognize shapes.
- This is easily achieved by having many convolutional layers.

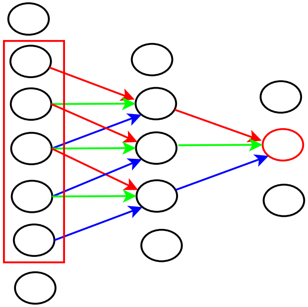
Receptive Field of a convolutional layer node

- The second convolutional layer gets its output from the first, through local connectivity.
- Eventhough connectivity is local, later filters are effectively looking at larger portion of the input. (We do not explicitly show the nonlinear activation)



Receptive Field of a convolutional layer node

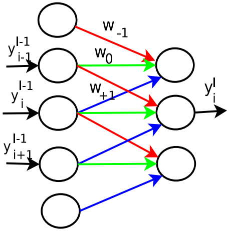
- The second convolutional layer gets its output from the first, through local connectivity.
- Eventhough connectivity is local, later filters are effectively looking at larger portion of the input. (We do not explicitly show the nonlinear activation)



- Each node in any convolutional layer has an effective receptive field in the input.

Output of a Convolutional Layer

- Since we have many layers let us revert to earlier notation.



$$y_i^l = f\left(\sum_{k=-1}^{+1} w_k^l y_{i+k}^{l-1}\right)$$

where w_k^l is the weight connecting any node in layer l to a node with offset k in layer $l-1$.

- When we generalize to 2-D case, y_i^l would become y_{ij}^l, w_k^l would become w_{sk}^l and so on.

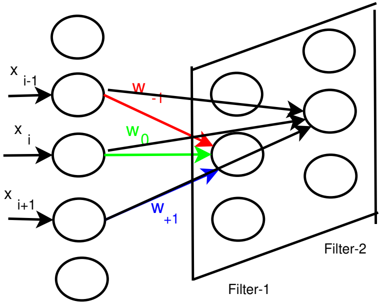
- The convolutional layer we saw so far is completely specified by one weight vector (of small dimension).
- It represents a single feture detector or filter.
- Our convolutional layer essentially detects this feature wherever it exists in the image.
- The output of the convolutional layer can be thought of as the representation of the input in terms of this feature (feature plane).

Multiple Filters

- A single feature would not be sufficient for many pattern recognition tasks.
- We would need multiple filters.
- For example, at each pont we may want to detect edges in different orientations.
- Further, through multiple layers we need to combine multiple simple features into multiple complex features.
- Hence, every convolutional layer should have multiple filters.

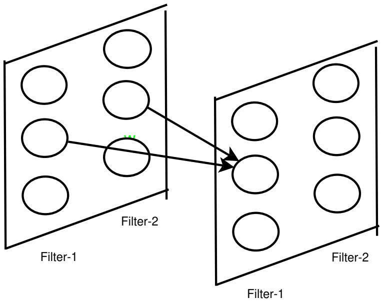
Convolutional layer with multiple filters

- As earlier let us take input as 1D.
- Now the convolutional layer would be 2D because of multiple filters



Multiple convolutional layers with multiple filters

- Consider two convolutional layers with multiple filters.
- Each layer is 2D (space dimension and filter dimension)



- Connectivity is full in the filter dimension

Notation for multiple filters

- Each output would now have three 'indices' – layer, position in the layer, and filter.
- $y_i^{l,m}$ – output of node-i, layer-l, filter-m.
- The connectivity is local in space but is full in filter domain (it needs to combine all features at that point).
- Each weight has four 'indices' – layer and filter number in that layer; the space-offset and filter coordinates of the input it is multiplying.
- $w_{k,m'}^{l,m}$ – weight for filter m in layer l connecting to node with 'offset' k in space and filter m' in layer $l-1$.

Output of a convolutional layer node with multiple features

- $y_i^{l,m}$ – output of node-i, layer-l, filter-m.
- $w_{k,m'}^{l,m}$ – weight for filter m in layer l connecting to node with 'offset' k in space and filter m' in layer $l-1$.
- The outputs are now calculated as

$$y_i^{l,m} = f\left(\sum_{m'}\sum_{k=-q}^{q} w_{k,m'}^{l,m} y_{i+k}^{l-1,m'}\right)$$

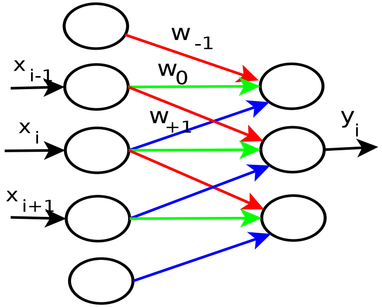
  $W^{l,m}$ – the 'weight matrix' associated with filter m in layer l.
- Compare with the single filter case

$$y_i^l = f\left(\sum_{k=-1}^{+1} w_k^l y_{i+k}^{l-1}\right)$$

Notation for multiple filters

- Each output now has three ‘indices’ – layer, position in the layer, and filter.
- Each weight has four ‘indices’ – layer and filter number in that layer; the space-offset and filter coordinates of the input it is multiplying.
- Essentially now all these are hyper-matrices (also called tensors).
- When we move to 2D, the space part would be a pair of coordinates / offsets.

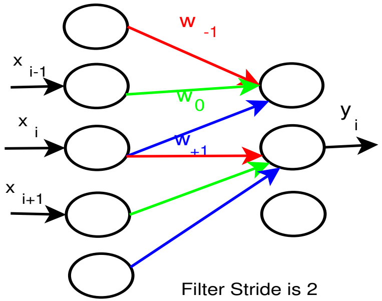
- Consider a single filter dimension.
- As we said earlier, there is local connectivity and weight sharing. That is what defines the filter



- There is one more characteristic for the filter definition.

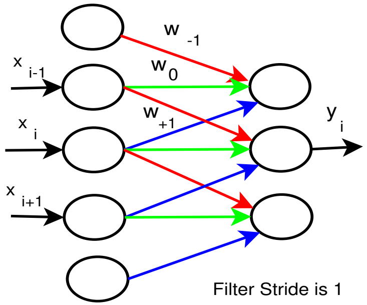
Stride of a filter

- In the connectivity there can be an 'off-set'
- The filter shown below is said to have stride 2.

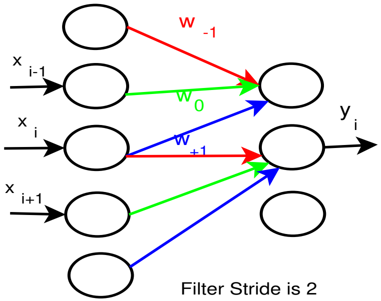


Stride of a filter

Filter with stride 1 and stride 2.



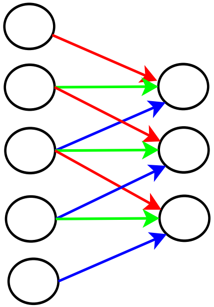
Filter Stride is 1

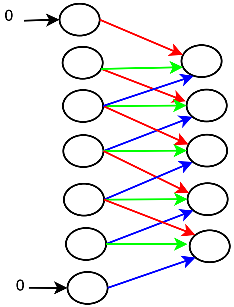


Filter Stride is 2

Zero Padding

- In all the figures shown so far, successive layers seem to have fewer nodes.
- We need not have that. Can use ‘zero-padding’.





Numbers of nodes in successive layers

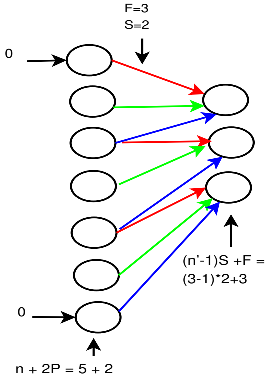
- In general, we can have different numbers in successive layers.
- Suppose we have n nodes in one layer and use P number of nodes for zero padding on either side. Suppose filter width is F and its **stride** is S.
- Then every node but the last one in the next layer would have S nodes exclusively to it and last node would have F. If this layer has n' nodes, then

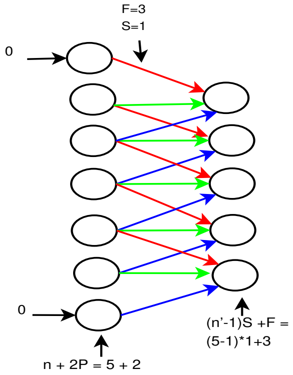
$$(n'-1)S + F = n + 2P$$

Numbers of nodes in successive layers

- Suppose n and n' are nodes in successive layers, P is number of nodes for zero padding on either side, filter width is F and its **stride** is S.

$$(n' - 1)S + F = n + 2P$$





Numbers of nodes in successive layers

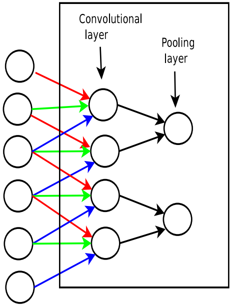
- Suppose we have n nodes in one layer and use P number of nodes for zero padding on either side. Suppose filter width is F and its **stride** is S. Then, number of nodes in the next layer, n' satisfies

$$(n' - 1)S + F = n + 2P$$

- Note that all quantities here have to be integers.
- If you do not use zero padding, number of nodes in successive layers decreases.
- Even otherwise, we may want sizes of convolutional layers successively reduced.

Pooling layers

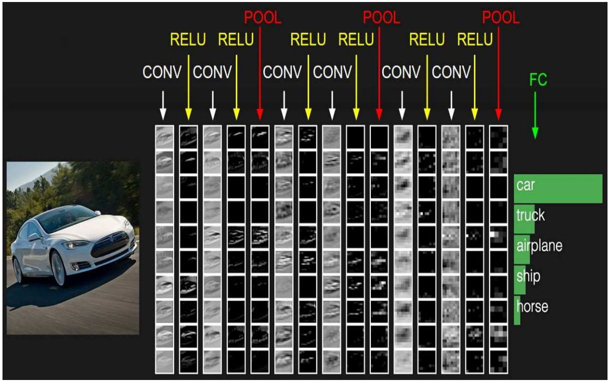
- One also reduces the size of the successive convolutional layers by using what is known as pooling.
- For example, we can reduce size to half by taking average or Max of successive elements.



- Often, Max pooling is used.

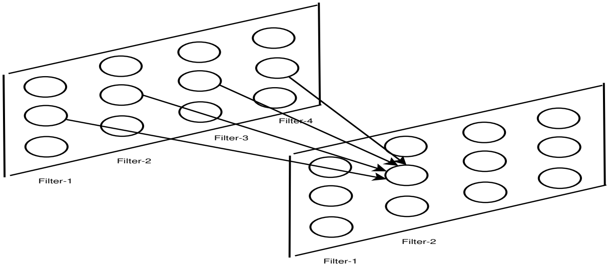
- Each convolutional layer has many filters and each extracts some feature.
- Through a series of convolutional layers, the original input (image) is transformed to a new feature representation.
- Then we need a classifier to classify it.
- For this, we have one or more ‘fully-connected’ layers after all convolutional layers.
- Finally, we may have a soft-max layer for multi-class classification.

A Typical CNN



- We need to extend all our notation to the case of images which are two dimensional in space.
- Each convolutional layer can be thought of as three dimensional – two space dimensions and one filter dimension.
- So, we can think of it as processing a ‘volume’ and producing a ‘volume’
- To keep notation uniform, we can think of input image also as 3-dimensional – e.g., colour being the third dimension.
- The third dimension in each layer, the **filters**, are also called the **channels**.

- Let us sligthly modify our notation for better readability.
- $y_r^\ell(i, j)$ – the output of the node at spatial location $(i,\ j)$ corresponding to filter r in (convolutional) layer ℓ.
- Now the node corresponding to filter r in layer ℓ would be associated with a weight tensor that connects this node to a limited spatial region and all filters in the previous layer. Let us look at 1D case:



- Let us sligthly modify our notation for better readability.
- $y_r^\ell(i, j)$ – the output of the node at spatial location $(i,\ j)$ corresponding to filter r in (convolutional) layer ℓ.
- Now the node corresponding to filter r in layer ℓ would be associated with a weight tensor that connects this node to a limited spatial region and all filters in the previous layer.
- $W_r^\ell(a, b; c)$ – weight connecting any node in layer ℓ and filter r to a node in the previous layer at a spatial offset given by $(a,\ b)$ and filter index in the previous layer, c.

- Now we can write the expression for output of a convolutional layer:

$$y_r^{\ell}(i,j) = \sum_c \sum_{a=-q}^{q} \sum_{b=-q}^{q} y_c^{\ell-1}(i+a, j+b) W_r^{\ell}(a,b;c)$$

  where c ranges over filters in layer $\ell - 1$.
- In the above, r ranges over number of filters in that layer and i, j range over the spatial extent of that layer.
- Here we have taken the spatial offset to go from $-q$ to q. (A notation)
- Hence filter size is $(2q+1) \times (2q+1)$.

- As we discussed earlier, after the convolution operation, we actually pass the output through a non-linearity.
- Supoose after the nonlinear activation function the output is $\bar{y}_r^\ell(i,j)$.
- Now the equations become

$$y_r^\ell(i,j) = \sum_c \sum_{a=-q}^{q} \sum_{b=-q}^{q} \bar{y}_c^{\ell-1}(i+a, j+b) W_r^\ell(a,b;c)$$

$$\bar{y}_r^\ell(i,j) = f(y_r^\ell(i,j)) = \max\{0, y_r^\ell(i,j)\}$$

where we are assuming ReLU activation.

- There are many important details that we have ignored in writing these equations.
- Recall that the number of nodes in successive layers are related by

$$(n_2 - 1) * S + F = n_1 + 2P$$

  (This holds for both space dimensions)
- In our equations we asumed $S = 1$.
- Our convention of taking $F = 2q + 1$ and letting the offset variable range over $-q$ to q is convenient only when we have $n_2 = n_1$ by taking P as large as needed.
- In such cases, we simply take the value to be zero when index is negative.

- Numbers of nodes are related by

$$(n_2 - 1) * S + F = n_1 + 2P$$

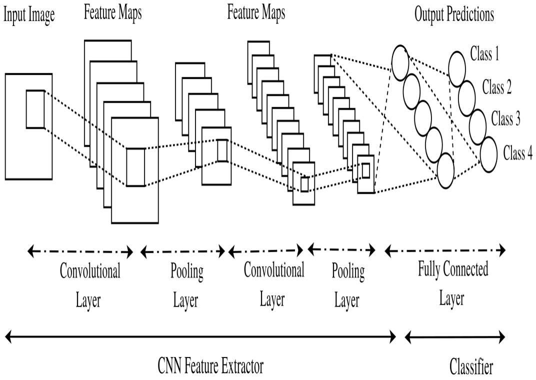
- Another special case is $S = 1$ and $P = 0$. Then $n_1 = n_2 - 1 + F$
- This would reduce the spatial size of the next convolutional layer
- Then we number nodes as 1 to n_1 and 1 to n_2; and let the offset variable run from 0 to $F - 1$.

- Our convention for numbering nodes as well as the ‘offset’ variables depends on how the number of nodes in successive layers changes.
- It also depends on the stride of the filter.
- We also need to properly take care of pooling layers.
- In general, writing these equations is a little more involved compared to those for regular feedforward networks.

- After a number of such convolutional layers, we have fully connected layers.
- This part is like a standard feedforward network we considered earlier.
- The input to the first fully connected layer would be a 'vectorized version' of the last convolutional layer.
- If the output of a convolutional layer needs to be an image, then we do not have the fully connected layers. ("Fully Convolutional Networks")

A Typical CNN Architecture

- Thus, we have the following structure for a CNN.



- Fixing a CNN architecture involves many issues.
- Each convolutional layer is characterized by number of filters, size of filters and stride.
- Each such layer has one weight tensor.
- The spacial extents of different convolutional layers is now determined by whether or not to we use zero-padding.
- We also need to fix details of fully connected layers.
- There are many standard architectures (e.g., Alexnet, VGGNet, Resnet etc).

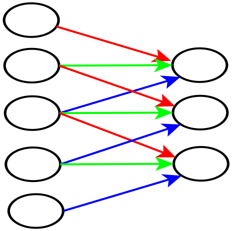
Training a CNN

- To use a CNN as a classifier, we need to learn all weights.
- One normally uses the cross entropy loss.
- We need to learn all the filters (weights in convolutional layers) in addition to weights in fully connected layers.
- Fully connected layers are same as the earlier feedforward networks.

Training a CNN

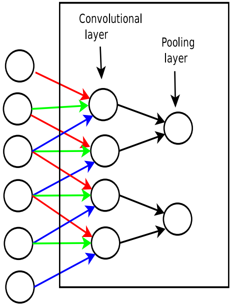
- We can learn the weights in convolutional layers (i.e., the filters) also using the backpropagation algorithm.
- Since convolutional layers do only linear summation followed by a nonlinear transformation, the backpropagation is essentially same as what we derived earlier.
- There is just one simple modification.

- In a CNN weights are shared.



- The final update for each weight is sum of all the updates.

- We can similarly backpropagate through pooling layer also.



- In case of max-pooling, we transfer the 'error at the node' to that node which was the maximum.

Regularization for learning CNNs

- Many different regularizations are used.
- We have discussed weight decay and L_2 regularization.
- Often also implemented as a constraint on norm of each weight vector
- Essentially, large values for weights is an indicator of overfitting.

dropout

- We had mentioned dropout earlier
- In dropout 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.
- Another variant is dropconnect.

- For a general network, dropout can be represented as:

$$y_j^l = f(\eta_j^l) = f\left(\sum_i w_{ij}^{l-1} \xi_i^{l-1} y_i^{l-1}\right)$$

  where $\xi_i^{l-1} \sim$ Bernouli(p).
- This can be particularized for any network structure, e.g., CNN
- In the equations, output of a node is multiplied by a Bernoulli random variable.
- The ξ_j^l are independent.
- This is effective in guarding against spurious 'co-adaptation' of weights.
- It essentially 'averages' many low-complexity networks and hence is effective as a regularization technique.

- Consider a single logistic unit (in some layer) with n inputs. Because of dropout it gets different subsets of inputs with different probabilities.
- Let $S_1, \cdots, S_m$ be the net input to this unit under these different subnetworks with probabilities $P_1, \cdots, P_m$ and let $O_1, \cdots, O_m$ be the outputs. (We would have $m = 2^n$).
- Note that the subscripts here do not refer to different units.
  (That is why we are using different symbols).

- Define weighted geometric mean and weighted geometric mean of the complement as

$$G = \prod_i (O_i)^{P_i} \quad G' = \prod_i (1 - O_i)^{P_i}$$

- Define Normalized weighted geometric mean as

$$NWGM(O_1, \cdots, O_m) = \frac{G}{G + G'}$$

- Then one can show that

$$NWGM(O_1, \cdots, O_m) = \frac{1}{1 + ce^{-\beta \sum_i P_i S_i}} = f(ES_i)$$

- We can generalize this to a full network

$$y_j^l = f(\eta_j^l) = f\left(\sum_i w_{ij}^{l-1}\xi_i^{l-1}y_i^{l-1}\right)$$

  we can show (under some approximation)

$$E[\eta_j^l] = \sum_i w_{ij}^{l-1}p_i^{l-1}E[y_i^{l-1}]$$

  where $p_j^l = E[\xi_j^l]$ $(= p,$ normally).
- This is essentially the averaging effect that dropout provides.

Batch Normalization

- In a neural network, we assume the input distribution to be constant. We learn a proper classifier for that distribution.
- As mentioned earlier, it helps to normalize the input distribution to have zero-mean and unit variance (or use a whitening transform)
- We can represent the output of a single hidden layer network as
$$y = F_2(F_1(x, \theta_1), \theta_2)$$
- Note that $x' = F_1(x, \theta_1)$ is the output of hidden layer and is input to the final layer.
- But the distribution of x' keeps changing as we are learning θ_1.

Batch Normalization

- Batch normalization is a heuristic method that attempts to normalize the distribution of outputs at each layer in a deep network.
- Such normalization is seen to make learning more stable under SGD.
- This allows one to use larger step-sizes and hence achieve faster convergence.

- Suppose x is input to some layer (output of previous layer). The normalized version would be

$$\tilde{x} = \text{Normalize}(x, \mathcal{S})$$

  where $\mathcal{S}$ is the training set.
- But this defeats the idea of SGD because now gradient depends on all examples.
- Hence the normalization is done only on a minibatch.

- Let $\mathcal{B} = \{x_1, \cdots, x_m\}$ be the inputs to a layer in a minibatch.
- These are all vectors. But we would normalize each component separately to have mean 0 and variance 1. Hence, we show equations as if these are scalars.
- The normalization could be:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\tilde{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

- The idea is to give $\tilde{x}_i$ as input to next layer.

- In a deep network, each layer is tranforming the representation.
- We do not know how the normalization would affect learning of the representation.
- Hence we compute the batch-normalization as

$$y_i = \mathsf{BN}_{\gamma,\beta}(x_i) = \gamma \tilde{x}_i + \beta$$

  and supply y_i as input to next layer.
- the parameters γ, β would also be learnt. (They would be different for different layers)
- If $\gamma = 1, \beta = 0$ we are using usual normalization.
- If $\gamma = \sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$, $\beta = \mu_{\mathcal{B}}$, then there is no normalization.
- We would learn the appropriate normalization for each layer.

- In this notation, at some layer, x_i is the input to that layer in the original network which is now transformed into y_i.
- If L is the loss function, then we need $\frac{\partial L}{\partial x_i}$ for updating weights connecting to this layer whose output is x_i.
- Since now y_i is input this layer, backpropagation can compute $\frac{\partial L}{\partial y_i}$.
- So, we need to express $\frac{\partial L}{\partial x_i}$ in terms of $\frac{\partial L}{\partial y_i}$ and other parameters of the batch-normalization transformation.
- We also need $\frac{\partial L}{\partial \gamma}$ and $\frac{\partial L}{\partial \beta}$.
- All these can be obtained using chain rule of differentiation.

- Now, in the final learnt network there is a batch-normalization transformation after each layer:

$$y_i = \gamma \tilde{x}_i + \beta$$

- We are learning γ and β for each layer. But to compute $\tilde{x}_i$ we need statistics of minibatch.
- What do we do during test time (regular operation)?

- After learning the network, with the learnt parameters fixed, we compute $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ over many random minibatches from training set, all of size m.
- Let
$$\bar{\mu} = E[\mu_{\mathcal{B}}] \quad \text{and} \quad \bar{\sigma}_{\mathcal{B}}^2 = \frac{m}{m-1} E[\sigma_{\mathcal{B}}^2]$$
where $E[\cdot]$ denotes average over all random minibatches.
- Let γ and β be learnt values for this layer. Then we use
$$y = \frac{\gamma}{\sqrt{\bar{\sigma}_{\mathcal{B}}^2 + \epsilon}} \; x \; + \; \left(\beta - \frac{\gamma \; \bar{\mu}}{\sqrt{\bar{\sigma}_{\mathcal{B}}^2 + \epsilon}} \right)$$
- This will be the final network that we use.

Regularization for CNNs

- We have mentioned three methods of regularization.
- These are the main ones used.
- One may use any subset of them.
- These can be used with all feedforward networks (and also with recurrent networks)
- One also progressively decreases step-size as learning proceeds to promote more stable learning.

Convolutional neural Networks

- CNNs are seen to achieve very high accuracies in a large number of applications involving classification of images, speech, text etc.
- The convolutional layer structure is very effective in extracting good feature representations using training data.
- CNNs can take a large part of the credit for the unprecedented interest in deep learning now.