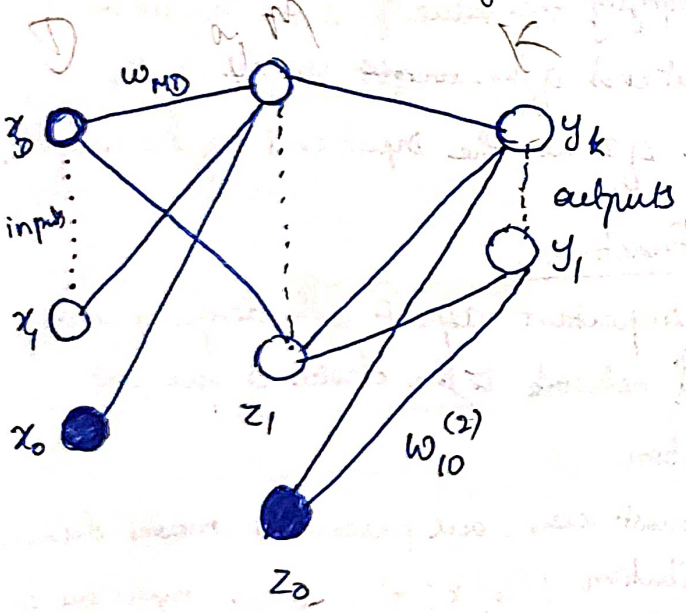
Backpropagation Algorithm

- Choose random weights for the ~~propagation~~ network.
- Feed in an example and obtain a result.
- Calculate the error for each node (starting from the last stage and propagating the error backwards)
- Update the weights.
- Repeat with other examples until the network converges on the target output.

Backdrop Algorithm Explanation - Neural network with one hidden layer



Neural network with one hidden layer

D input variables $x_1, \ldots x_D$

M hidden unit activations.

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \quad \text{where } j = 1, \ldots M$$

Hidden unit activation functions

$$z_j = h(a_j)$$

K output activations.

$$a_k = \sum_{i=1}^{M} w_{ki}^{(2)} z_i + w_{k0}^{(2)} \quad \text{where } k = 1, \ldots K$$

Output activation functions

$$y_k = \sigma(a_k)$$

$$y_k(x, w) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h \left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right)$$

- Each layer is a function of layer that preceeded it.
- First layer is given by

$$z = h\left(W^{(1)T} x + b^{(1)} \right)$$

- Second layer is

$$y = \sigma\left(W^{(2)T} z + b^{(2)} \right)$$

Gradient descent

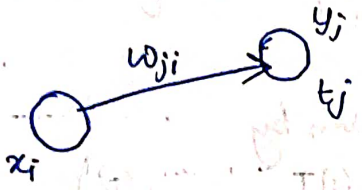
- Gradient descent is used to determine weights w from labelled set of training samples.
- Learning procedure has two stages
  1. Evaluate derivatives of loss $\nabla E(w)$ with respect to weights $w_1, \ldots w_7$
  2. Use derivative ~~vector~~ to compute adjustments to weights.

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)})$$

η - learning rate.

$$\nabla E(w) = \begin{bmatrix} \frac{\partial E}{\partial w_0} \\ \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_T} \end{bmatrix}$$

Gradient of Error function wrt a weight

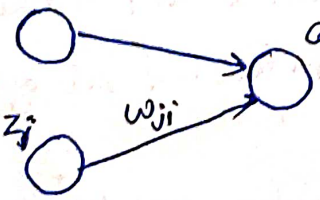


- For a particular input x and weight w, squared error is:

$$E = \frac{1}{2}(y(x,w) - t)^2$$

$$\frac{\partial E}{\partial w} = (y(x,w) - t) = \delta \cdot x$$

Evaluation of Derivative wrt a weight



$$\frac{\partial E_n}{\partial w_{ji}} = \partial E_n$$

By chain rule for partial derivatives

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}}$$

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j}$$

$$a_j = \sum_i w_{ji} z_i$$

we have $\frac{\partial a_j}{\partial w_{ji}} = z_i$

Evaluating of Derivative E_n wrt a weight w_{ji}

- Substituting we get: $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$

- i.e required derivative is obtained by multiplying the value of δ for the unit at the output end of the ~~weight~~ weight by the value of z at the input end of the weight.

Cost Functions

- An important aspect of the design of a deep neural network is the choice of the cost function.
- In most cases, our parametric model defines a distribution $p(y|x;\theta)$ and we simply use the principle of maximum likelihood
- This means we use the cross entropy between the training data and the model's predictions as the cost function.
- Cross entropy is a way to calculate distances between two functions or probability distributions.
- Cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label.

- This cost function is given by

$$J(\theta) = -E_{x,y \sim \hat{p}_{data}} \log p_{model}(y|x)$$

Learning Conditional Distributions with Maximum likelihood

- We would like to have some principle from which we can derive specific functions that are good estimators for different models.
- The most common such principle is the maximum likelihood principle
- Consider a set of m examples $X = \{x^{(1)}, \ldots, x^{(m)}\}$ drawn independently from the true but unknown data generating distribution $p_{data}(x)$.

Maximum Likelihood Estimation (MLE)

- MLE is just a way to measure how good or bad a model is.
- A good estimator is a function whose output is close to the true underlying θ that generated the training data.
- Our goal is, for any input, output value should be as close as ideal.
  - This is so called maximum likelihood.
- Let $p_{model}(x; \theta)$ maps any configuration x to a real number estimating the true probability $p_{data}(x)$.
- The maximum likelihood estimator for θ is then defined as

$$\theta_{ML} = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^{(i)}; \theta)$$

Logarithm of the likelihood

- This product over many probabilities can be inconvenient.
- We observe that taking the logarithm of the likelihood does not change its arg max but does conveniently transform a product into a sum:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x^{(i)}; \theta)$$

log transform

- Doing the log transform of the likelihood function makes it easier to handle (multiplication becomes sums) and this is also numerically more stable.
- This is because the magnitude of the likelihoods can be very small.
- Doing a log transform converts these small numbers to large negative values which a finite precision machine can handle better.

KL Divergence

- One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{data}$ defined by the training set and the model distribution.
- With the degree of dissimilarity between the two distributions measured by the Kullback-Leibler (KL) divergence.

Kullback - Leibler (KL) divergence

- The KL divergence is given by

$$D_{KL}(\hat{P}_{data} \| P_{model}) = E_{x \sim \hat{P}_{data}}\left[\log \hat{P}_{data}(x) - \log P_{model}(x)\right]$$

- The term on the left is a function only of the data generating process, not the model. This means when we train the model to minimize the KL divergence, we need only minimize.

$$-E_{x \sim \hat{P}_{data}}\left[\log P_{model}(x)\right]$$

KL divergence

- Minimizing this KL divergence corresponds exactly to minimizing the cross entropy ~~beta~~ between the distributions.
- Any loss consisting of a negative log likelihood is a cross entropy between the empirical distribution defined by the training set and the model.

Maximum Likelihood vs KL divergence

- We can thus see the maximum likelihood as an attempt to make the model distribution match the empirical distribution $\hat{P}_{data}$ (training set).
- While the optimal θ is the same regardless of whether we are maximizing the likelihood or minimizing the KL divergence.
- Maximum likelihood thus becomes minimization of the negative log-likelihood (NLL) or equivalently minimization of cross entropy.

Learning Conditional Distributions with Maximum Likelihood

- Most modern neural networks are trained using maximum likelihood.
- This means that the cost-function is simply the negative log likelihood, ~~or~~ equivalently described as the cross entropy between the training data and the model distribution.

KL divergence

- The perspective of maximum likelihood as minimum KL divergence becomes helpful in this case ~~be~~ because the KL divergence has a known minimum value of zero.
- Relative entropy
- KL divergence quantifies the difference between 2 problems distance; such as true distance of an approximation of that distribution.

~~Cate~~

- Calculate a score that measures the divergence of one probability distribution from another.

- Provide shortcuts for calculating scores such as mutual information (information gain).

For discrete probability ~~P~~ and Q defined on the same probability space, X the relative entropy from Q to P is defined to be

$$D_{KL}(P\|Q) = \sum_{x \in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

which is equivalent to

$$D_{KL}(P\|Q) = -\sum_{x \in X} P(x) \log\left(\frac{Q(x)}{P(x)}\right)$$

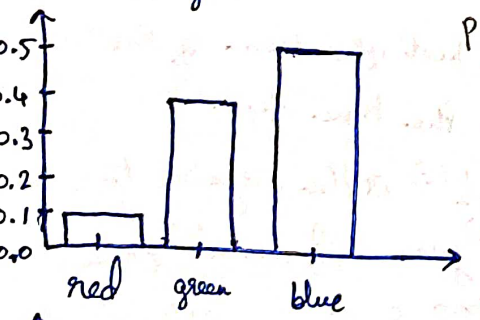
- When the score is 0, it suggests that both distributions are identical, otherwise the score is positive.
- Define distributions

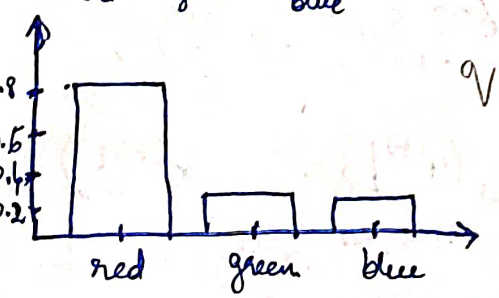
events = ['red', 'green', 'blue']

P = [0.10, 0.40, 0.50]

q = [0.80, 0.15, 0.05]

- Plot histogram





Find KL divergence KL(P||Q)

- KL ~~diverg~~ divergence KL(P || Q)
- 0.1 (log [0.1/0.8]) + 0.4 (log [0.4/0.15]) + 0.5 (log [0.5/0.05])
- KL (P||Q) : 1.927 bits
- KL (Q||P) : 2.022 bits
- There is more divergence in this second case.

- The log can be base-2 to give units in "bits" of the natural logarithm base-e with units in "nats"
- ~~The~~ The KL divergence is the average number of extra bits needed to encode the data - due to the fact that we used distribution q to encode the data instead of the true distribution p.
- The intuition for the KL divergence score is that ~~we~~ when the probability for an event from P is large, but the probability for the same event in Q is small, there is a large divergence.
- When the probability from P is small and the probability from Q is large, ~~there~~ there is also a large divergence, but not as large as the first case.

Recurrent Neural Network (RNN)

- Recurrent neural network or RNNs are a family of ~~n~~ neural networks for processing sequential data.
- Consider the classical form of a dynamical system driven by an external

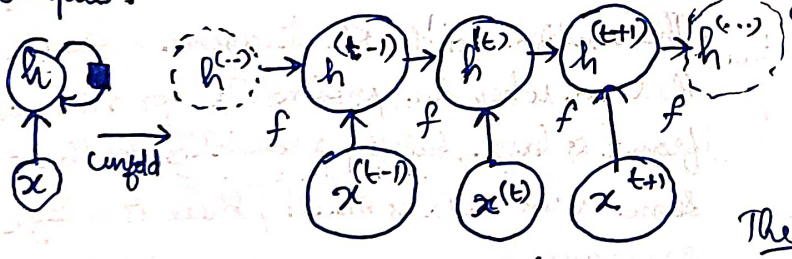
System $x^{(t)}$, $s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta)$

S is called state of the system.

- This equation is recurrent because the definition of s at time refers back to the same definition at time $t-1$.
- Essentially any function involving recurrence can be considered a recurrent neural network.
- To indicate that the state is the hidden units of the network, we now rewrite above Eq. using the variable h to represent the state.

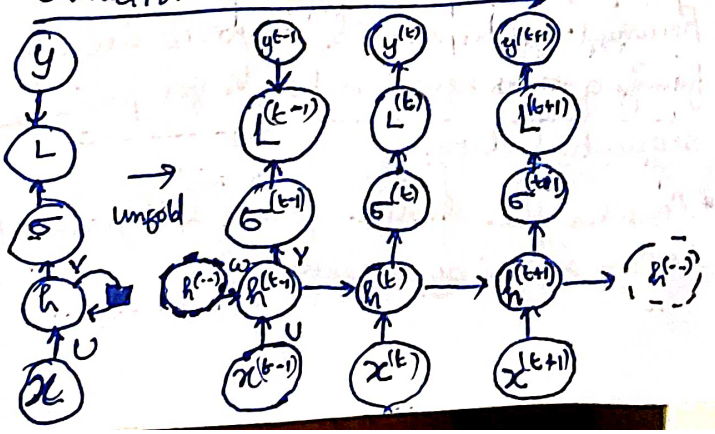
$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

A recurrent network with no outputs.



The network seen as an unfolded computational graph, where each node is now associated with the particular time instant.

Recurrent networks that produce an output at each time step and have recurrent connections between hidden units



- For the hidden units here we assume the hyperbolic tangent activation function.
- We can then apply the softmax operation as a post-processing step to obtain a vector $\hat{y}$ of normalized probabilities over the output.
- Forward propagation begins with a specification of the initial state $h^{(0)}$.

Then for each time step from $t=1$ to $t=\tau$, we apply the following update equations:

$$a^{(t)} = b + W h^{(t-1)} + U x^{(t)}$$
$$h^{(t)} = \tanh(a^{(t)})$$
$$o^{(t)} = c + V h^{(t)}$$
$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

* where the parameters are the bias vectors b and c along with weight matrices U, V and W, respectively for input-to-hidden, hidden-to-output and hidden-to-hidden connections.

The total loss

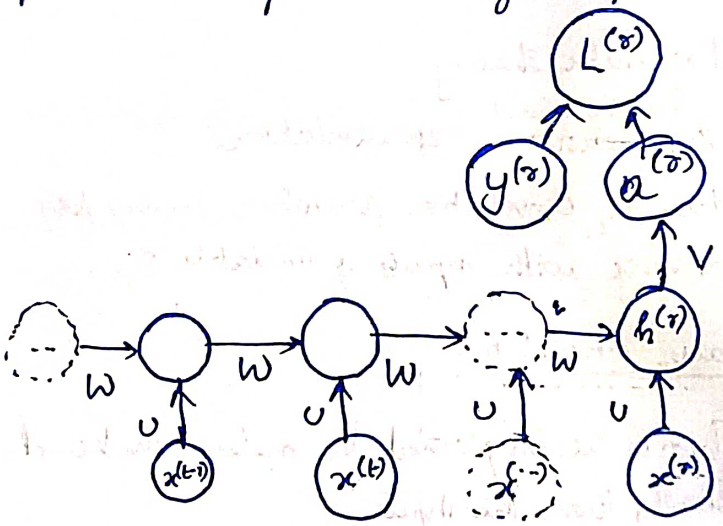
- The total loss for a given sequence of x values paired with a sequence of y values would then be just the sum of the losses over all the time steps.
- For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $x^{(1)}, \ldots x^{(t)}$ then

$$L(\{x^{(1)}, \ldots x^{(\tau)}\}, \{y^{(1)}, \ldots, y^{(\tau)}\})$$
$$= \sum_t L^{(t)}$$
$$= -\sum_t \log P_{\text{model}}(y^{(t)} \mid \{x^{(1)} \ldots x^{(t)}\})$$

where $P_{\text{model}}(y^{(t)} \mid \{x^{(1)}, \ldots x^{(t)}\})$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{y}^{(t)}$.

- The network with recurrence bfo hidden units is very powerful but also expensive to train.

Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output.



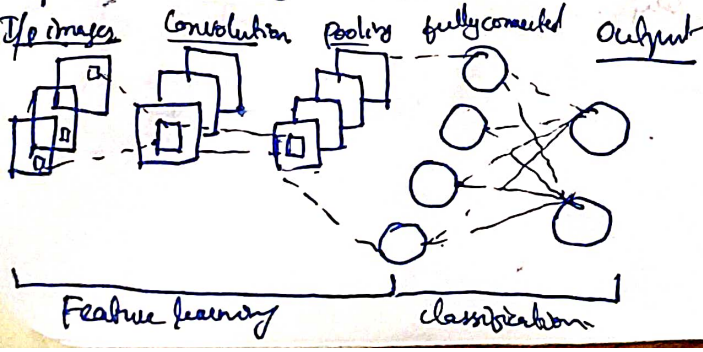
- Above fig shows the time unfolded recurrent NN with a single o/p at the end of the sequence

- Such a network can be used to summarize a sequence and produce a fixed size representation used as input for further processing.

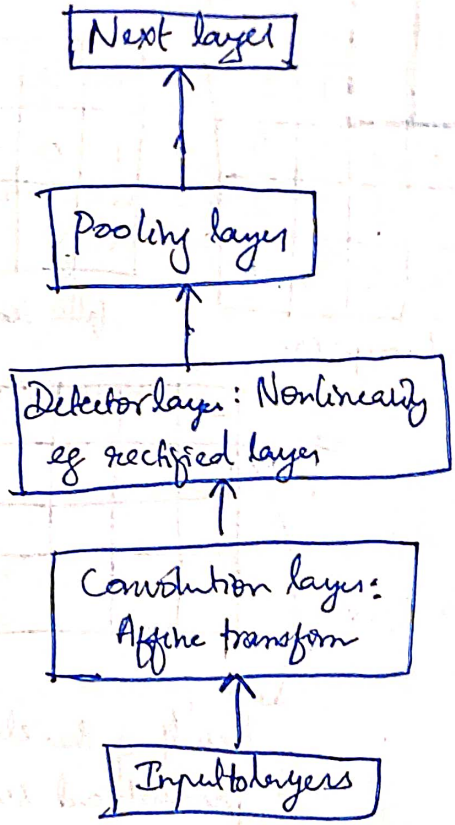
Convolutional Neural Networks (CNN)

- Convolutional neural networks (LeCun, 1989), are a specialized kind of neural network for processing data that has a known, grid like topology.

- The network employs a mathematical operation called convolution.



The components of a typical convolutional neural network layer



Discrete convolution

- If we now assume that x and w are defined only on integer t, we can define the discrete convolution:

$$S(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)\, w(t-a)$$

- Continuous form

$$S(t) = \int x(a)\, w(t-a)\, da$$

Two dimensional convolutions

- Finally we often use convolutions over more than one axis at a time.

- For ~~time~~ example, if we use a two-dimens... image I as our input, we probably also want to use a two dimensional kernel K.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)\, K(i-m, j-n)$$

Convolution Example

3	1	1	2	8	4
1	0	7	3	2	6
2	3	5	1	1	3
1	4	1	2	6	5
3	2	1	3	7	2
9	2	6	2	5	1

Original image 6×6

×

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

filter ~~test~~ 3×3

=

-7			

Result of the element wise product and sum of the filter matrix and the original image.

An example of 2D convolution

Input

a	b	c	d
e	f	g	h
i	j	k	l

Kernel

w	x
y	z

$aw+bx+ey+fz$	$bw+cx+fy+gz$	$cw+dx+gy+hz$
$ew+fx+iy+jz$	$fw+gx+jy+kz$	$gw+hx+ky+lz$

How convolution improves a machine learning system?

- Sparse interactions
  - also refered to as
  - Sparse connectivity or
  - Sparse weights
- Parameter sharing
- ~~Equivalent~~ Equivariant representations
- Moreover, convolution provides a means for working with inputs of variable size.

Sparse interactions

→ This is accomplished by making the kernel smaller than the input.

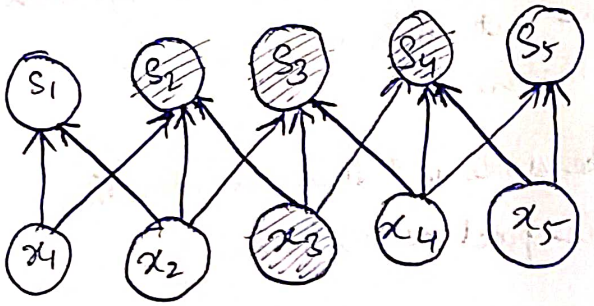
→ For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels.

→ This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency.

→ It also means that computing the output requires fewer operations.

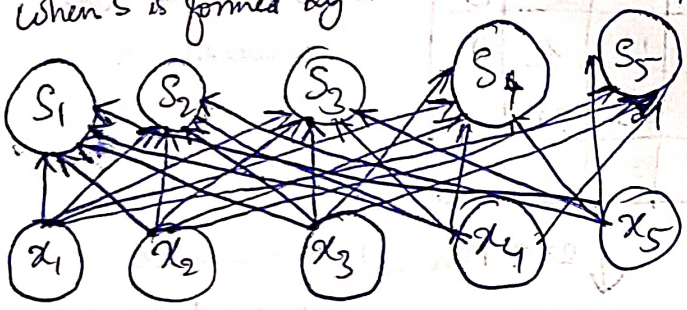
→ These improvements in efficiency are usually quite large.

One input unit, x3, and the output units in s that are affected by this unit.



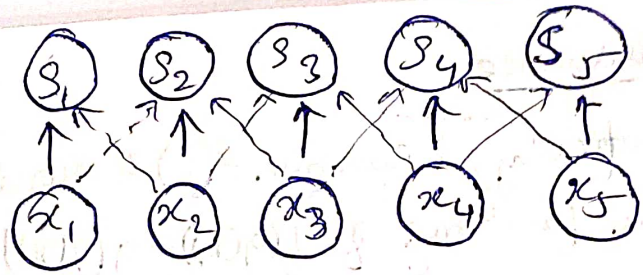
~~When s is formed by~~

When s is formed by matrix multiplication,



Parameter sharing

- Parameter sharing refers to using the same parameter for more than one function in a model
- In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer.
- It is multiplied by one element of the i/p and then never revisited
- The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.



- The black arrows indicate uses of central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations.
- In a CNN, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels).
- The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set.
- Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirement and statistical efficiency.

Equivalent representation

- In the case of convolution, the particular form of parameter sharing causes the layer to have a property called equivalence to translation.
- To say a function is equivariant means that if the input changes, the output changes in the same way.

Equivariance

- Specifically, a function $f(x)$ is equivalent to a function g if $f(g(x)) = g(f(x))$
- In the case of convolution, if we let g be any function that translates the input, i.e shifts it, then the convolution function is equivariant to g.
- With images, convolution creates a 2D map of where certain features appear in the input.
- If we move the object in the input, its representation will move the same amount
- Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image.

Pooling

- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs.
- For example, the max pooling (Zhou and Chellappa 1988) operation reports the max o/p within a rectangular neighbourhood.
- In all cases, pooling helps to make the representation become approximately invariant to small translations of the input.

Stride

- Stride is the number of pixels by which we slide our filter matrix over the input matrix.
- When the stride is 1 then we move the filters one pixel at a time.

Max pooling example

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

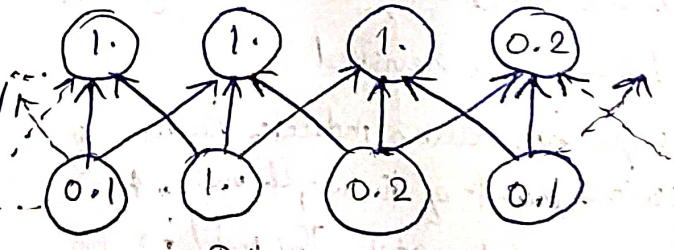
↓ 2×2 Max Pool

20	30
112	37

- Pooling is basically "downscaling" the image obtained from the previous layers.
- It can be compared to shrinking an image to reduce its pixel density.

- Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.

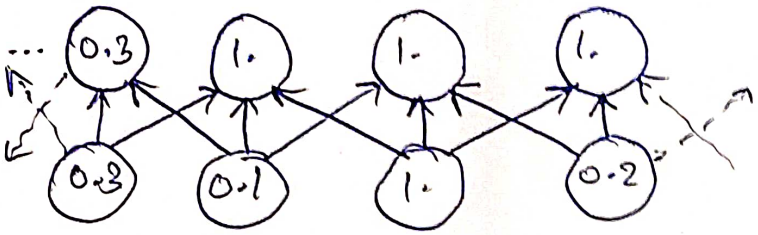
Max pooling introduces invariance



Detector stage.

- Bottom row shows o/ps of non linearity.
- Top row shows the o/ps of max pooling with a stride of 1 pixel b/w pooling regions and a pooling region width of three pixels.

After ~~that~~ the input has been shifted to the right by one pixel



Detector stage

- Every value in the bottom row has changed, but only half of the values in the top row has changed, because
- The max pooling unit are only sensitive to the maximum value in the neighbourhood, not its exact location.