Deep Learning I

CS 760: Machine Learning

Spring 2018

Mark Craven and David Page

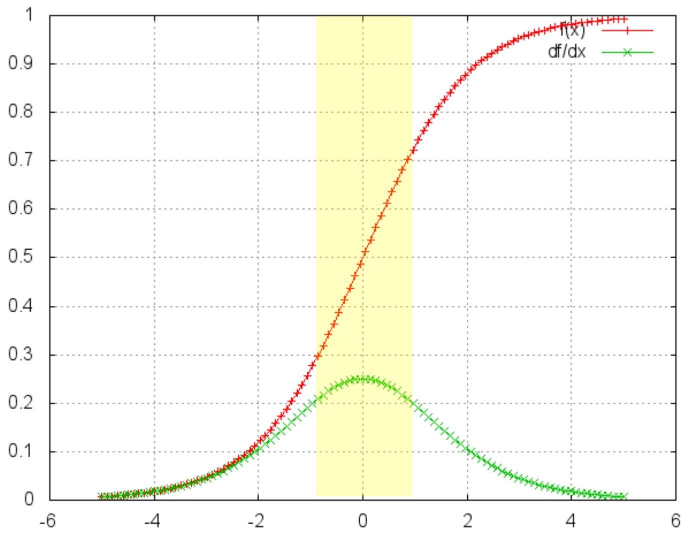
www.biostat.wisc.edu/~craven/cs760

Goals for the Lecture

- You should understand the following concepts:
  - one-hot encoding
  - autoencoders
  - denoising autoencoders
  - recurrent neural networks
  - convolutional neural networks
  - parameter tying
  - pooling
  - dropout training
  - batch normalization
  - Nesterov momentum

Wrapping Up Last Time: Initializing weights

- Weights should be initialized to
  - small values so that the sigmoid activations are in the range where the derivative is large (learning will be quicker)

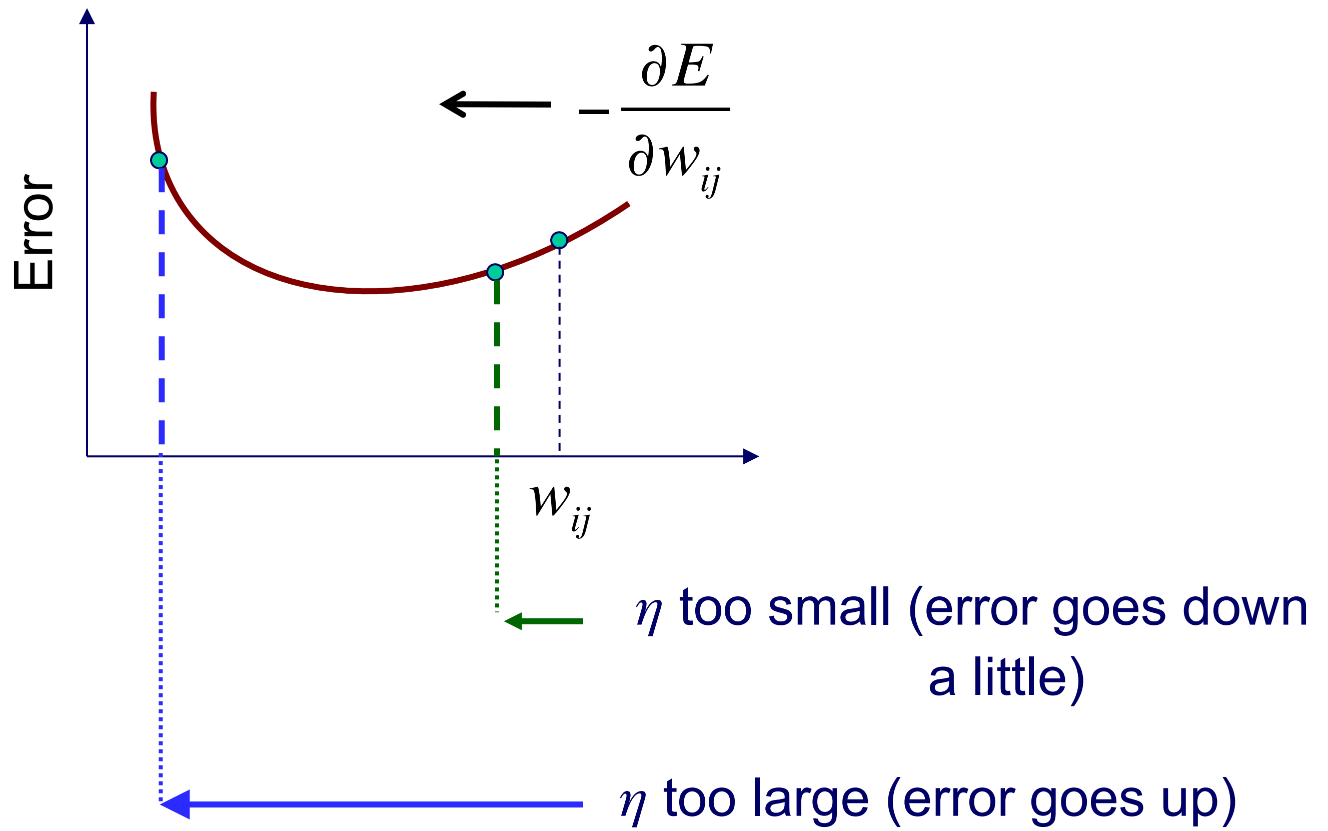


  - random values to ensure symmetry breaking (i.e. if all weights are the same, the hidden units will all represent the same thing)

- typical initial weight range [-0.01, 0.01]

Setting the learning rate

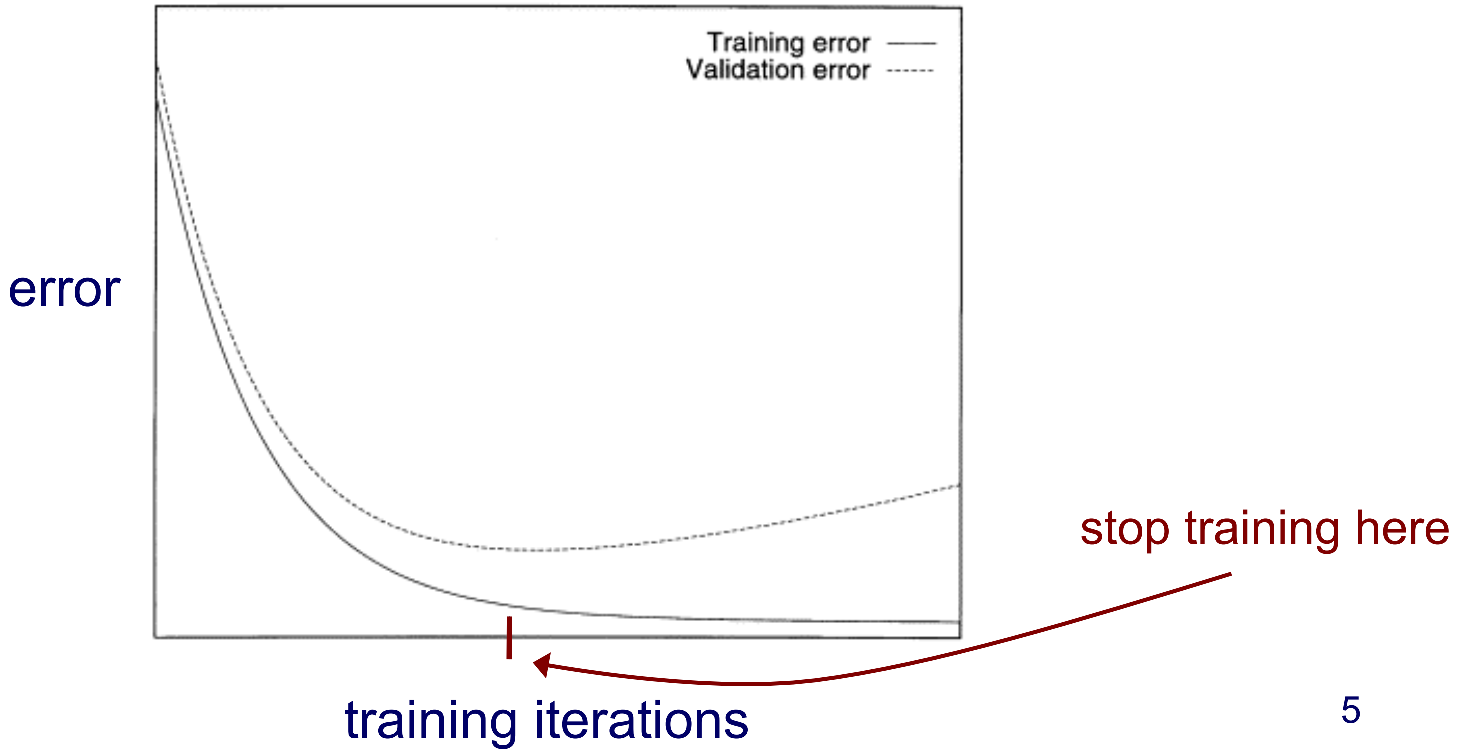
convergence depends on having an appropriate learning rate



Error
$-\frac{\partial E}{\partial w_{ij}}$
w_{ij}
η too small (error goes down a little)
η too large (error goes up)

Stopping criteria

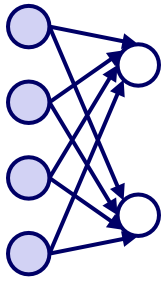
- conventional gradient descent: train until local minimum reached
- empirically better approach: *early stopping*
  - use a validation set to monitor accuracy during training iterations
  - return the weights that result in minimum validation-set error



Input (feature) encoding for neural networks

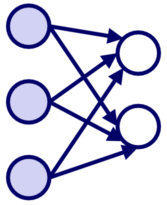
nominal features are usually represented using a *1-hot* encoding

$$A = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$



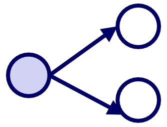
ordinal features can be represented using a *thermometer* encoding

$$small = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad medium = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad large = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$



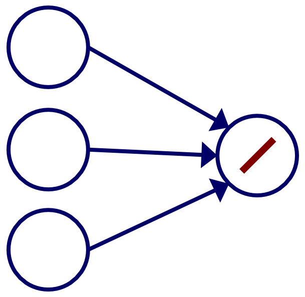
real-valued features can be represented using individual input units (we may want to scale/normalize them first though)

$$precipitation = \begin{bmatrix} 0.68 \end{bmatrix}$$

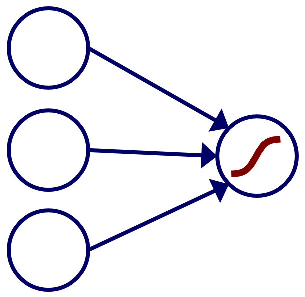


Output encoding for neural networks

regression tasks usually use output units with linear transfer functions

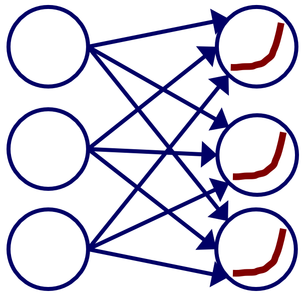


binary classification tasks usually use one sigmoid output unit



k-ary classification tasks usually use *k* sigmoid or *softmax* output units

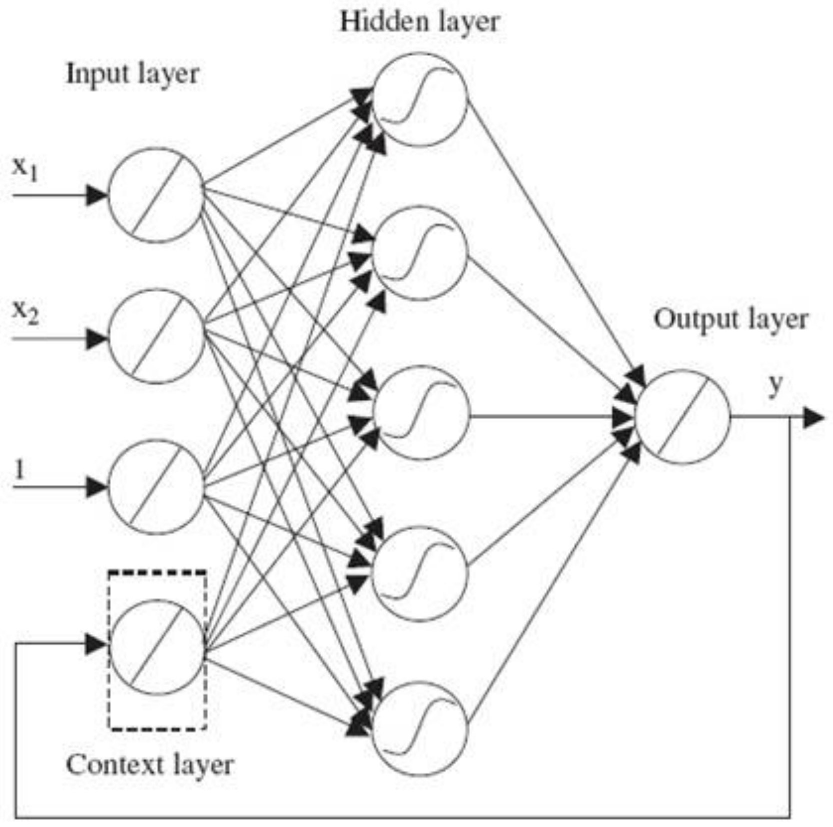
$$o_i = \frac{e^{net_i}}{\sum_{j \in outputs} e^{net_j}}$$



Recurrent neural networks

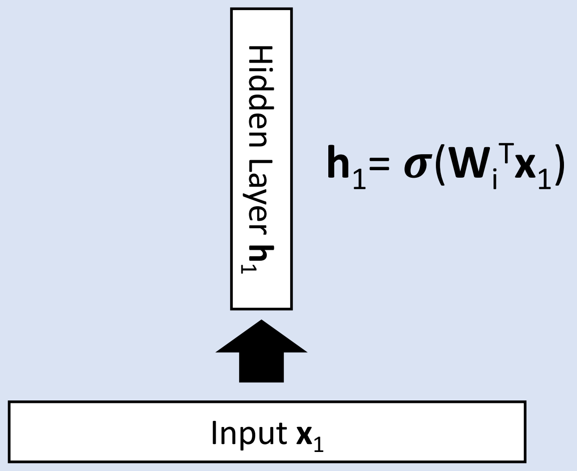
recurrent networks are sometimes used for tasks that involve making sequences of predictions

- Elman networks: recurrent connections go from hidden units to inputs
- Jordan networks: recurrent connections go from output units to inputs



Recurrent Neural Networks (thanks Ed Choi, Jimeng Sun!)

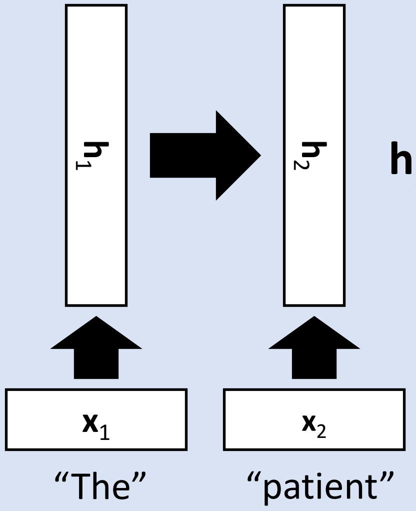
- Recurrent Neural Network (RNN)
  - Binary classification


Hidden Layer $\mathbf{h}_1$
$\mathbf{h}_1 = \boldsymbol{\sigma}(\mathbf{W}_i^T \mathbf{x}_1)$
Input $\mathbf{x}_1$

$\mathbf{x}_1$ (First element in the sequence “The”)

Recurrent Neural Networks

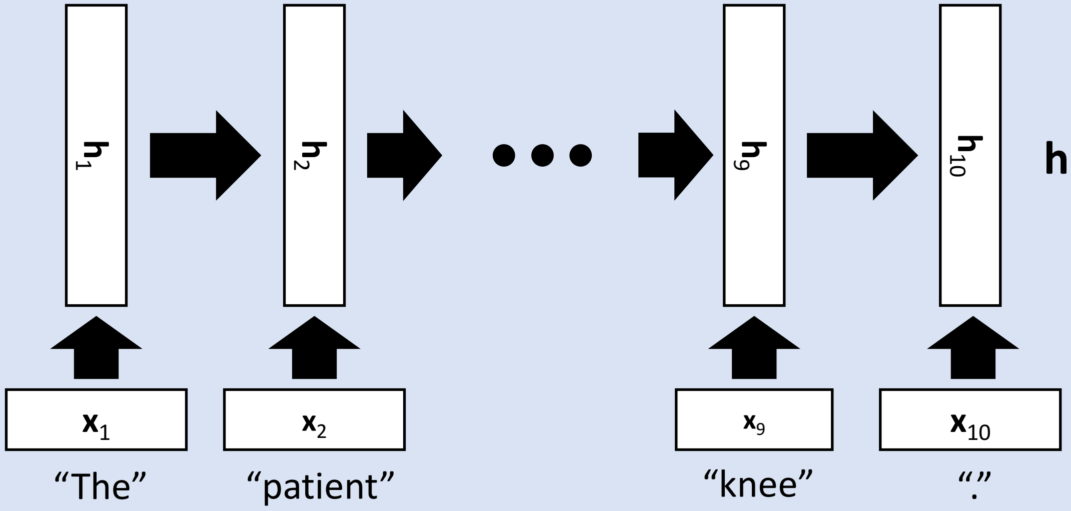
- Recurrent Neural Network (RNN)
  - Binary classification



$$\mathbf{h}_2 = \sigma(\mathbf{W}_h^T \mathbf{h}_1 + \mathbf{W}_i^T \mathbf{x}_2)$$

Recurrent Neural Networks

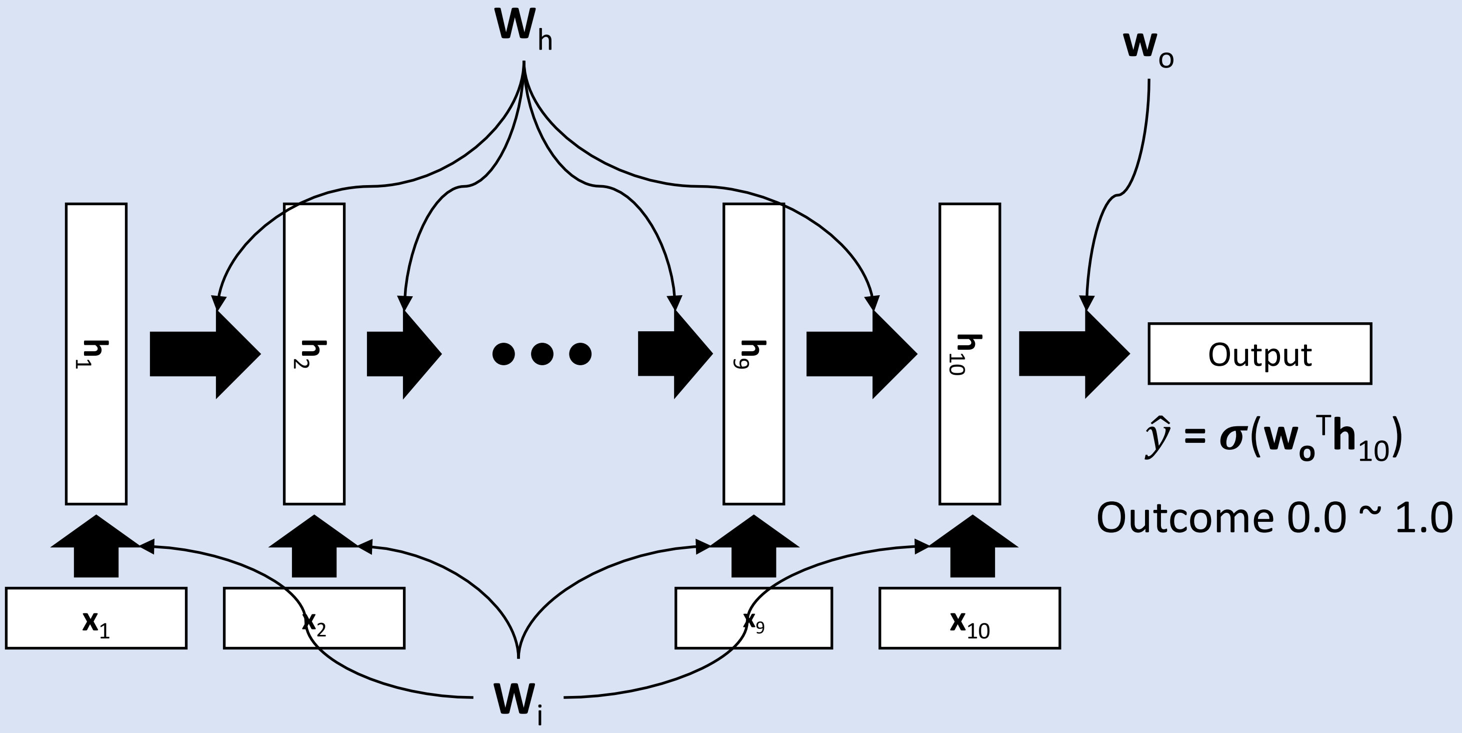
- Recurrent Neural Network (RNN)
  - Binary classification



$$\mathbf{h}_{10} = \sigma(\mathbf{W}_h^T \mathbf{h}_9 + \mathbf{W}_i^T \mathbf{x}_{10})$$

Recurrent Neural Networks

- Recurrent Neural Network (RNN)
  - Binary classification



Convention Wisdom of Last Decade

- Theorem: **one** hidden layer can represent any function
  - Number of hidden units the one hyperparameter for ANNs
  - Just tune that hyperparameter
  - Fit well into Weka and other packages
- Empirical results: learning by backpropagation doesn’t work well with more than one hidden layer
  - converge to poor solutions in practice
  - gradients either vanish or exhibit poor credit assignment in earlier hidden layers (those further from output, and hence further from the error computation)

Competing intuitions

- Only need a 2-layer network (input, hidden layer, output)
  - Representation Theorem (1989): Using sigmoid activation functions (more recently generalized to others as well), can represent any continuous function with a single hidden layer
  - Empirically, adding more hidden layers does not improve accuracy, and it often degrades accuracy, when training by standard backpropagation

- Deeper networks are better
  - More efficient representationally, e.g., can represent *n*-variable parity function with polynomially many (in *n*) nodes using multiple hidden layers, but need exponentially many (in *n*) nodes when limited to a single hidden layer
  - More structure, should be able to construct more interesting derived features

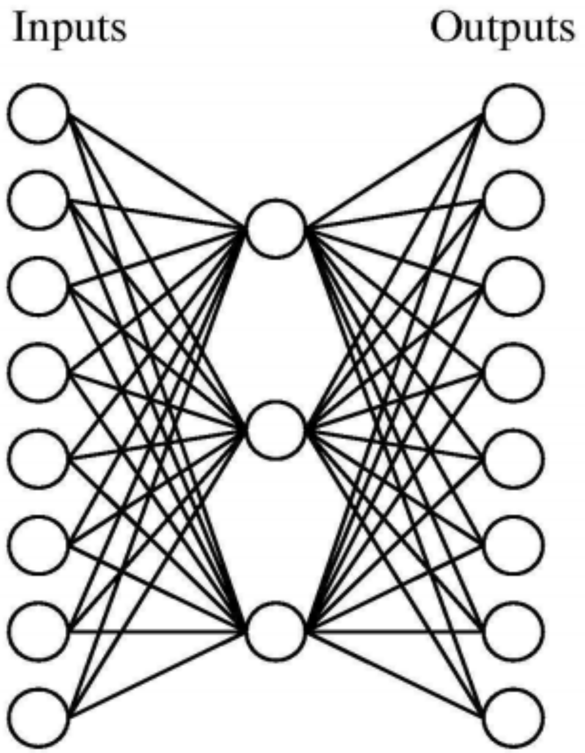
Learning representations

- the feature representation provided is often the most significant factor in how well a learning system works
- an appealing aspect of multilayer neural networks is that they are able to change the feature representation
- can think of the nodes in the hidden layer as new features constructed from the original features in the input layer
- consider having more levels of constructed features, e.g., pixels -> edges -> shapes -> faces or other objects

The role of hidden units

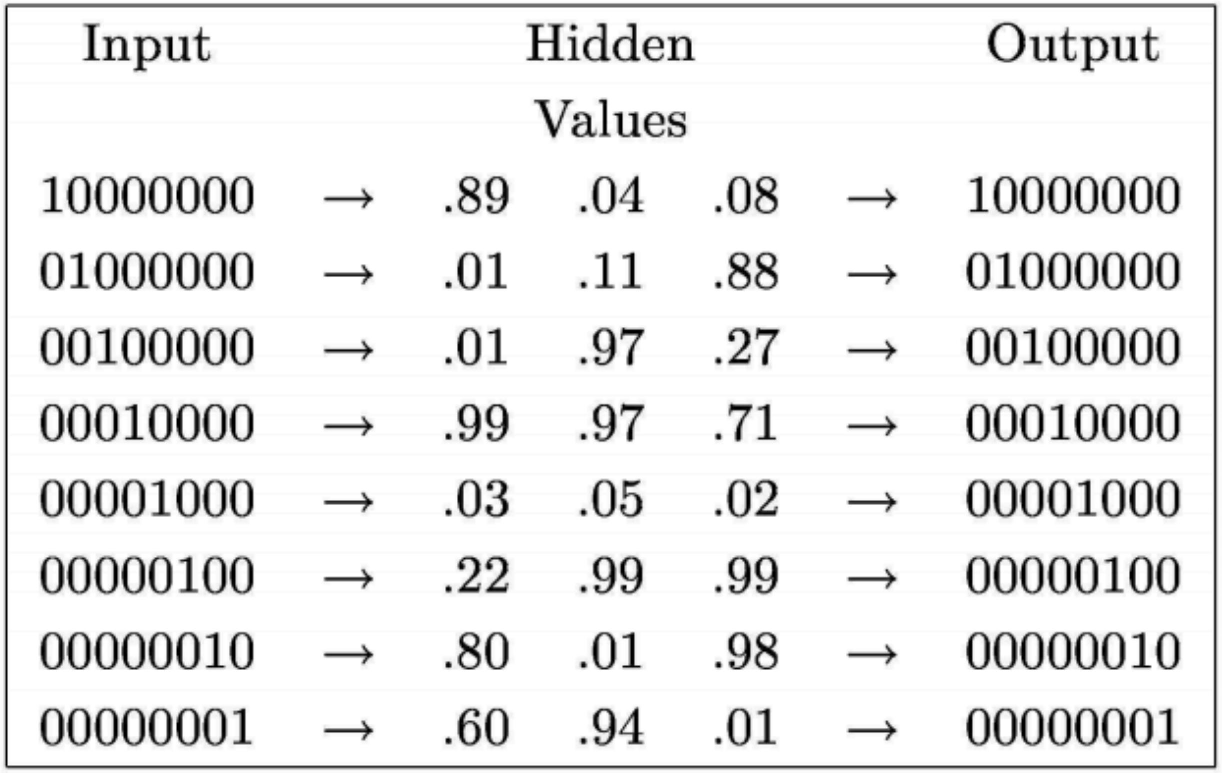
- Hidden units transform the input space into a new space where perceptrons suffice
- They numerically represent "constructed" features
- Consider learning the target function using the network structure below:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001



The role of hidden units

- In this task, hidden units learn a compressed numerical coding of the inputs/outputs



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

How many hidden units should be used?

- conventional wisdom in the early days of neural nets: prefer small networks because fewer parameters (i.e. weights & biases) will be less likely to overfit
- somewhat more recent wisdom: if early stopping is used, larger networks often behave as if they have fewer "effective" hidden units, and find better solutions

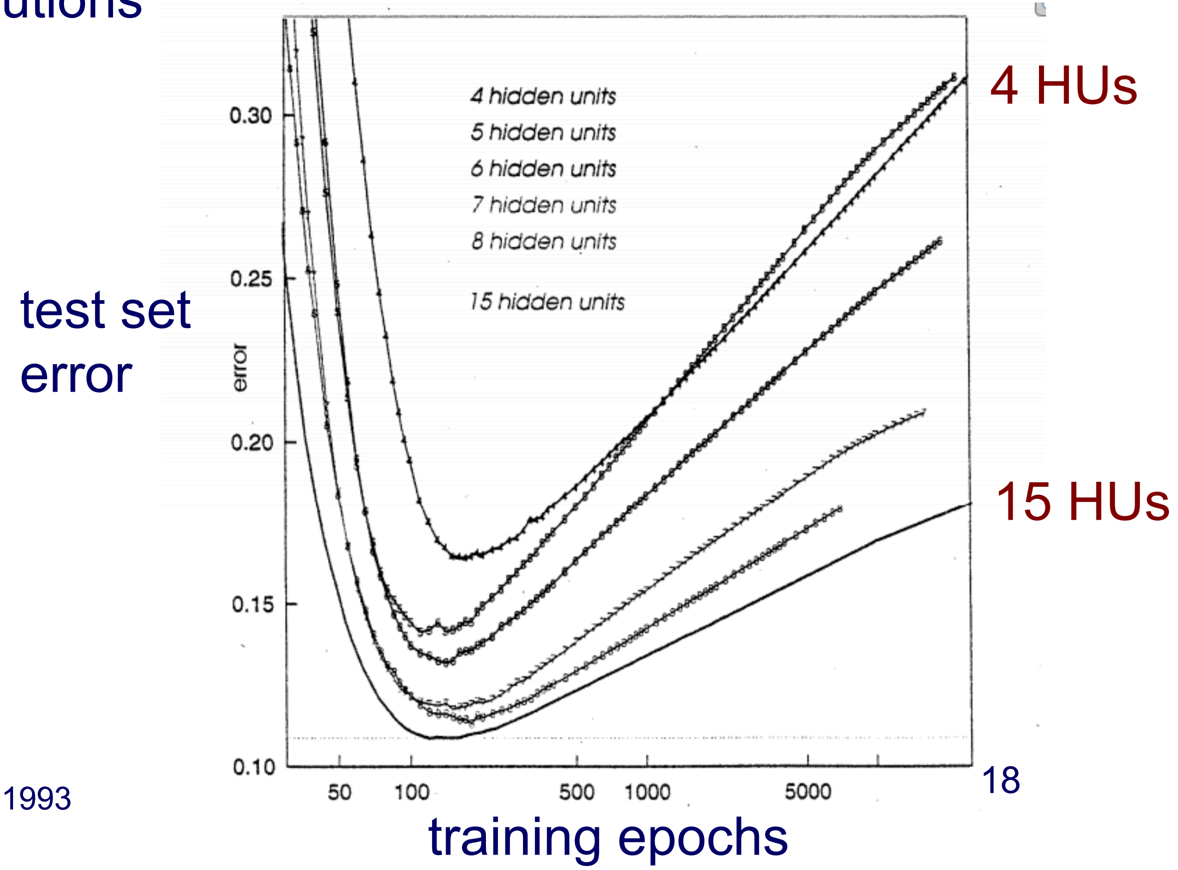


Figure from Weigend, *Proc. of the CMSS* 1993

Another way to avoid overfitting

- Allow many hidden units but force each hidden unit to output mostly zeroes: tend to meaningful concepts

- Gradient descent solves an optimization problem—add a “regularizing” term to the objective function

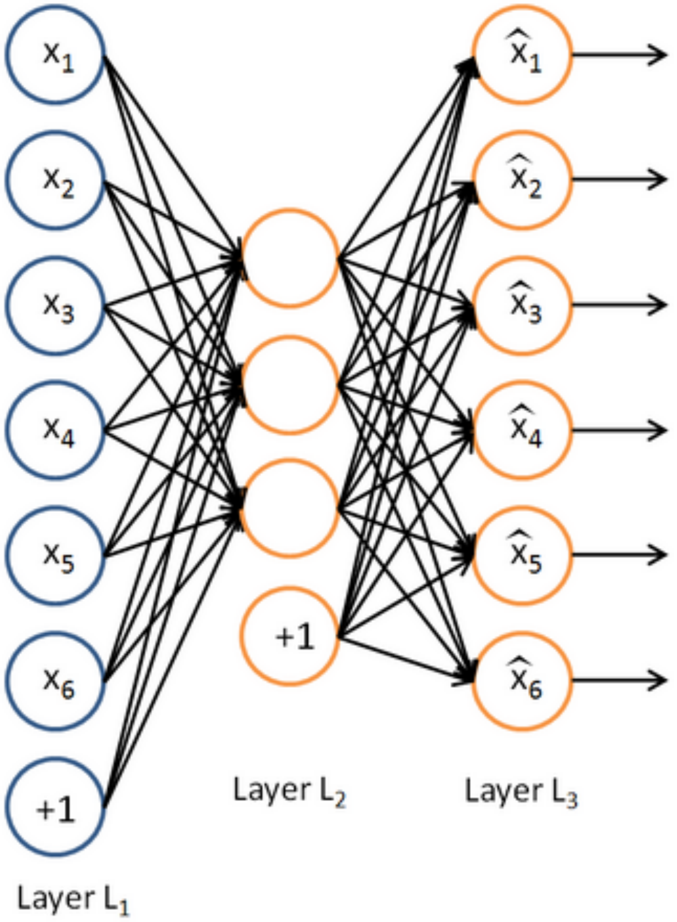
- Let $\mathbf{X}$ be vector of random variables, one for each hidden unit, giving average output of unit over data set. Let target distribution s have variables independent with low probability of outputting one (say 0.1), and let $\hat{s}$ be empirical distribution in the data set. Add to the backpropagation target function (that minimizes δ’s) a penalty of $\mathrm{KL}(s(\mathbf{X})||\hat{s}(\mathbf{X}))$

Backpropagation with multiple hidden layers

- in principle, backpropagation can be used to train arbitrarily deep networks (i.e. with multiple hidden layers)
- in practice, this doesn’t usually work well
  - there are likely to be lots of local minima
  - diffusion of gradients leads to slow training in lower layers
    - gradients are smaller, less pronounced at deeper levels
    - errors in credit assignment propagate as you go back

First Approach to Turn Things Around: Autoencoders

- one approach: use *autoencoders* to learn hidden-unit representations
- in an autoencoder, the network is trained to reconstruct the inputs

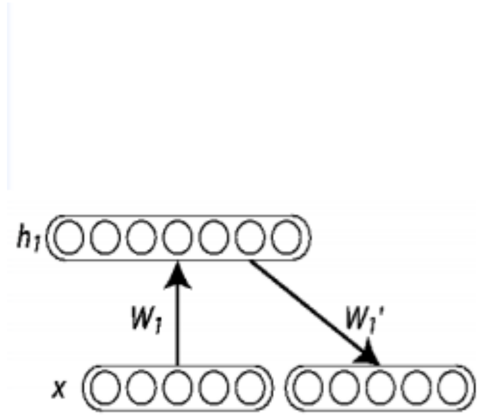


Autoencoder variants

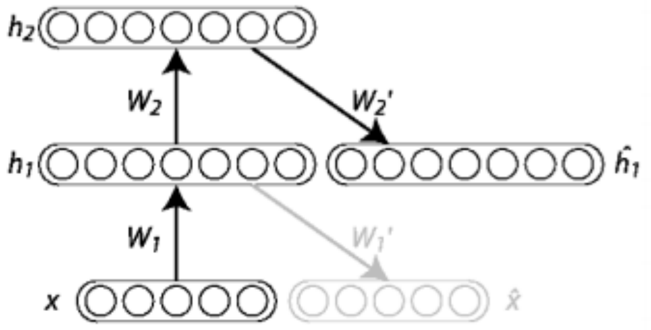
- how to encourage the autoencoder to generalize
  - *bottleneck*: use fewer hidden units than inputs
  - *sparsity*: use a penalty function that encourages most hidden unit activations to be near 0 [Goodfellow et al. 2009]
  - *denoising*: train to predict true input from corrupted input [Vincent et al. 2008]
  - *contractive*: force encoder to have small derivatives (of hidden unit output as input varies) [Rifai et al. 2011]

Stacking Autoencoders

- can be stacked to form highly nonlinear representations [Bengio et al. *NIPS* 2006]

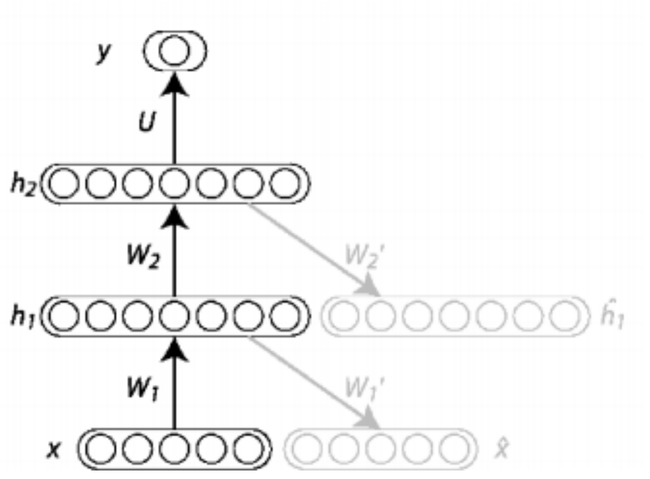


train autoencoder to represent $\boldsymbol{x}$



Discard output layer; train autoencoder to represent h_1

Repeat for k layers



discard output layer; train weights on last layer for supervised task

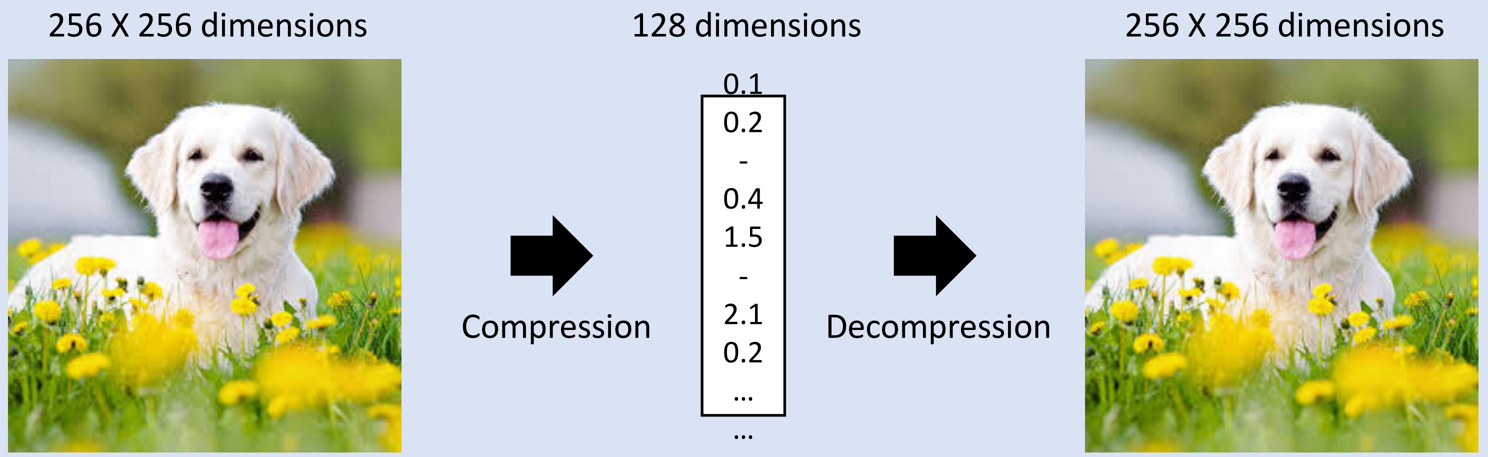
each $\boldsymbol{W}_i$ here represents the matrix of weights between layers

Fine-Tuning

- After completion, run backpropagation on the entire network to fine-tune weights for the supervised task
- Because this backpropagation starts with good structure and weights, its credit assignment is better and so its final results are better than if we just ran backpropagation initially

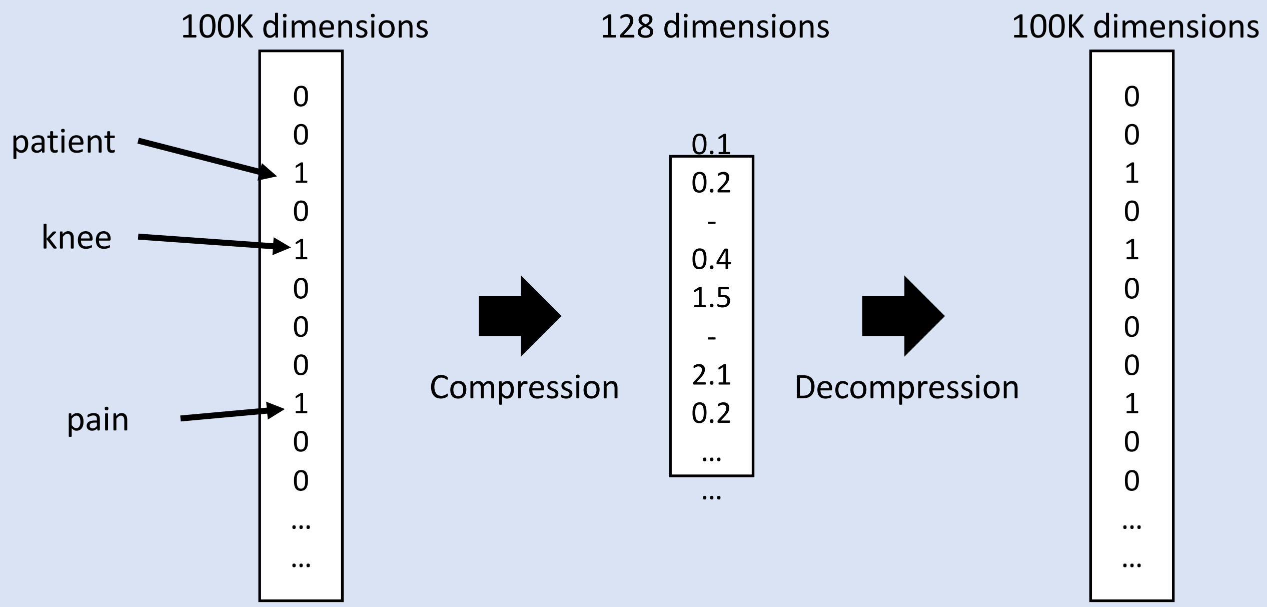
Autoencoders (thanks Ed Choi, Jimeng Sun!)

- Compression & decompression
  - Learning the latent representation of a given sample **x**



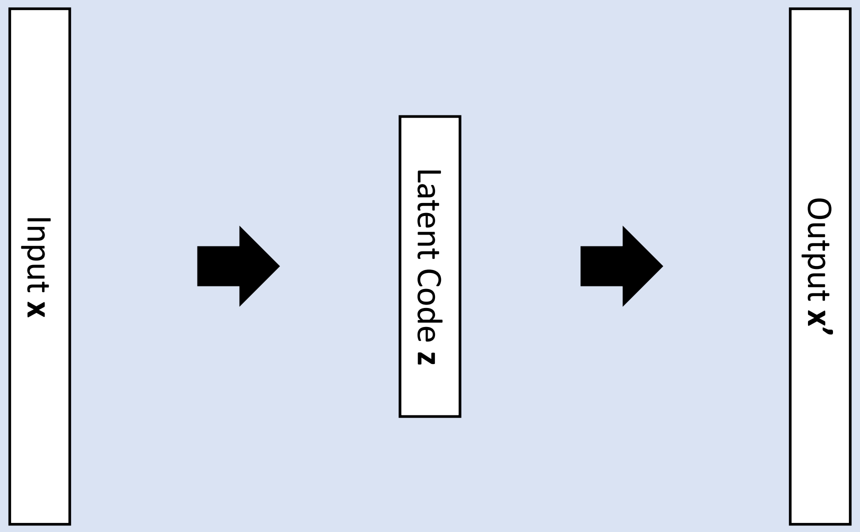
Autoencoders

- Compression & decompression
  - Learning the latent representation of a given sample **x**



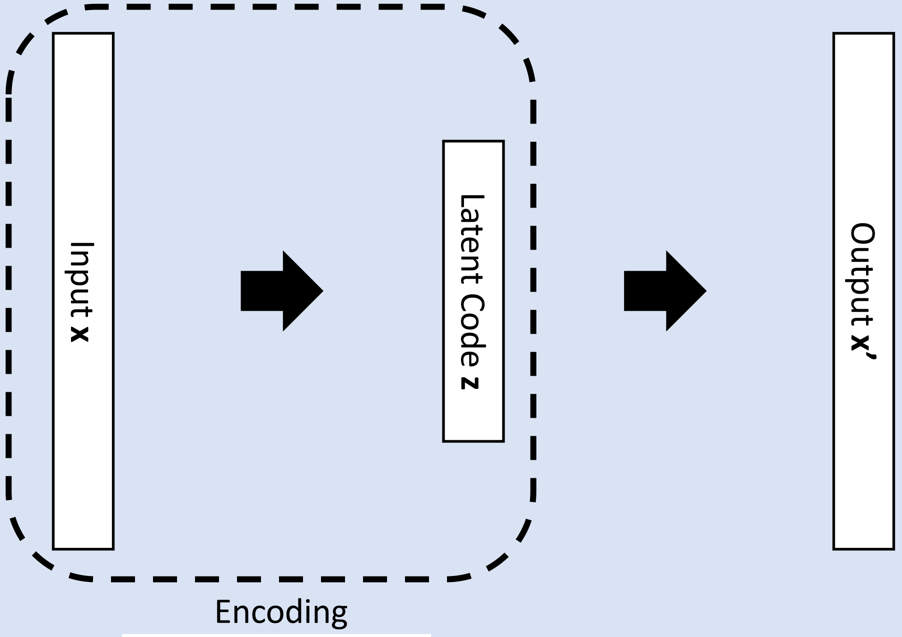
Autoencoders

- Compression & decompression
  - Learning the latent representation of a given sample **x**



Autoencoders

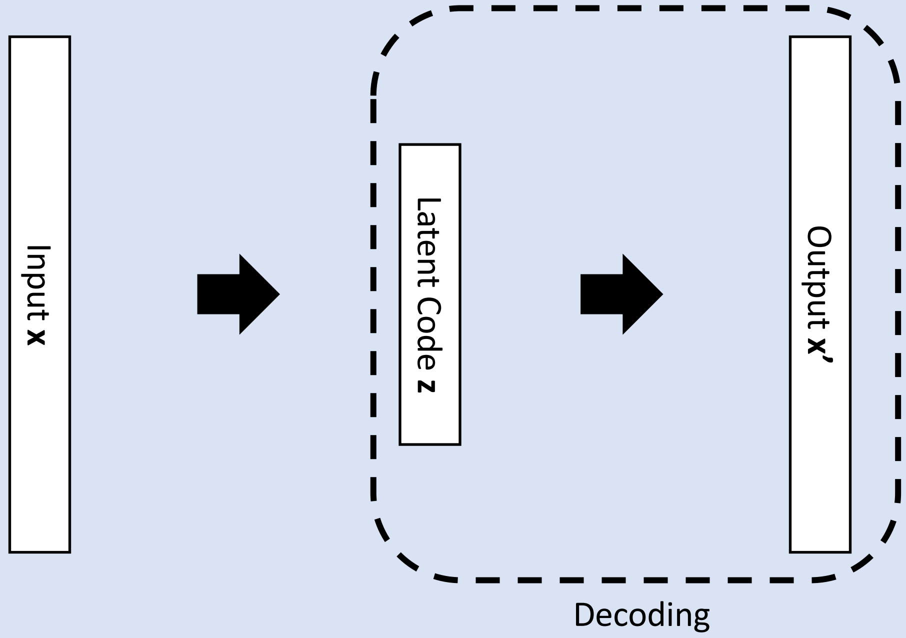
- Compression & decompression
  - Learning the latent representation of a given sample **x**



$$\mathbf{z} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Autoencoders

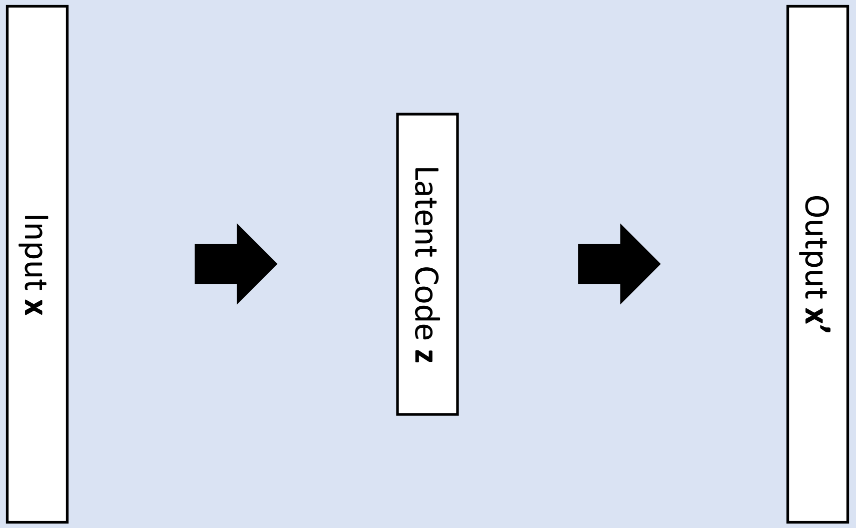
- Compression & decompression
  - Learning the latent representation of a given sample **x**



$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}')$$

Autoencoders

- Compression & decompression
  - Learning the latent representation of a given sample **x**

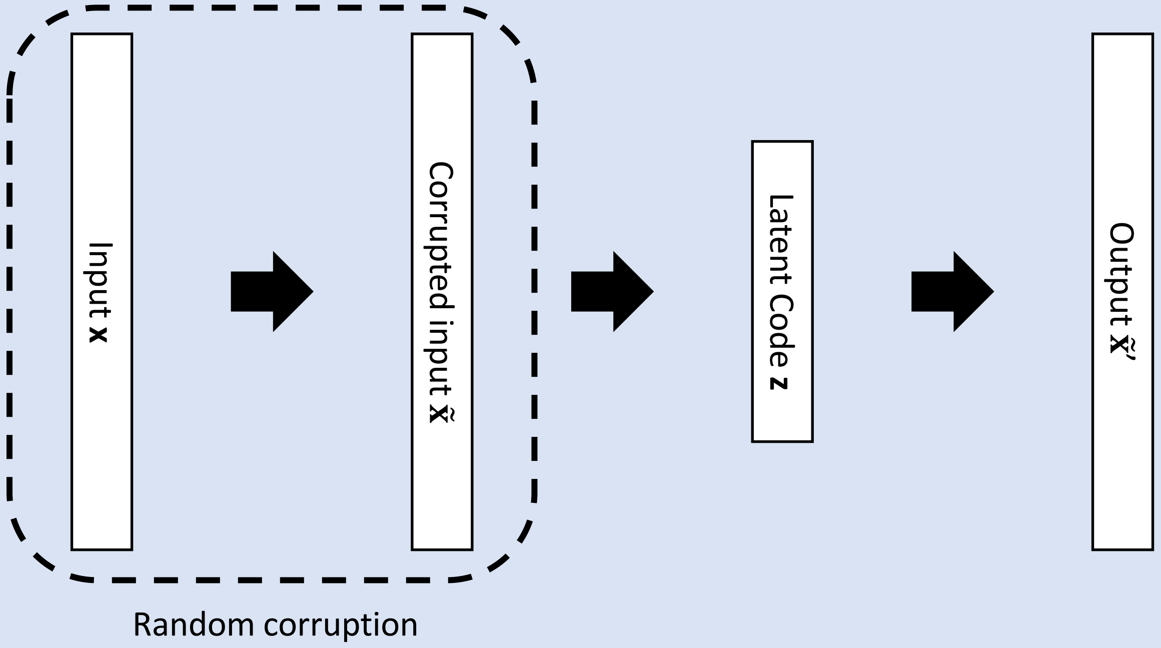


Minimize reconstruction error

$$\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2$$

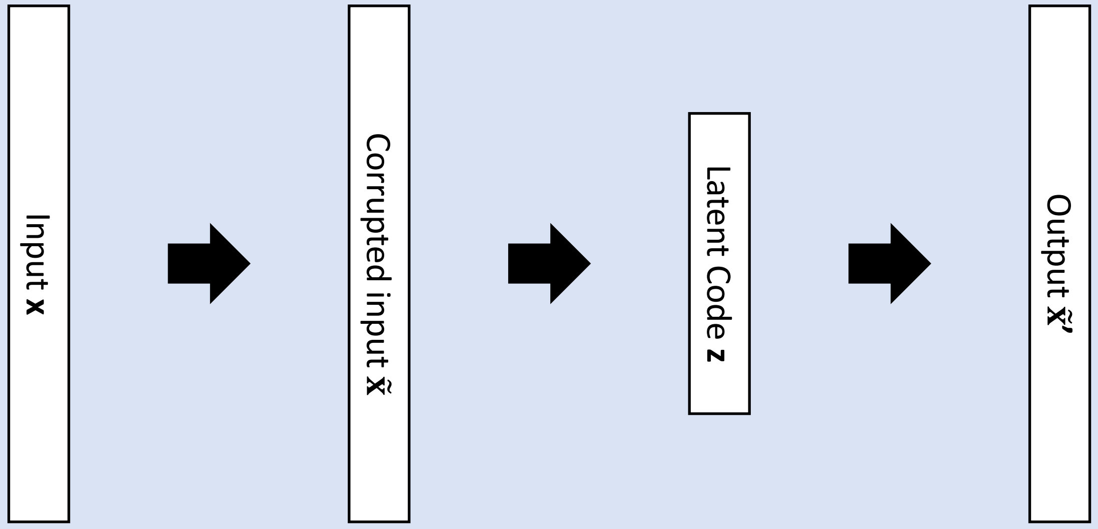
Denoising Autoencoders

- Corrupt the input sample **x**
  - To learn a robust representation of **x**
  - The model strives to learn the joint probability of the dimensions of **x**


Input **x**
Corrupted input $\tilde{\mathbf{x}}$
Latent Code **z**
Output $\tilde{\mathbf{x}}'$
Random corruption

Denoising Autoencoders

- Corrupt the input sample **x**
  - To learn a robust representation of **x**
  - The model strives to learn the joint probability of the dimensions of **x**


Input **x**
Corrupted input $\tilde{\mathbf{x}}$
Latent Code **z**
Output $\tilde{\mathbf{x}}'$

Minimize reconstruction error

$$\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}}') = \|\mathbf{x} - \tilde{\mathbf{x}}'\|^2$$

Why does the unsupervised training step work well?

- *regularization hypothesis*: representations that are good for $P(\boldsymbol{x})$ are good for $P(\mathrm{y} \mid \boldsymbol{x})$
- *optimization hypothesis*: unsupervised initializations start near better local minima of supervised training error

Deep learning not limited to neural networks

- First developed by Geoff Hinton and colleagues for *belief networks*, a kind of hybrid between neural nets and Bayes nets
- Hinton motivates the unsupervised deep learning training process by the credit assignment problem, which appears in belief nets, Bayes nets, neural nets, restricted Boltzmann machines, etc.
  - d-separation: the problem of evidence at a converging connection creating competing explanations
  - backpropagation: can’t choose which neighbors get the blame for an error at this node

The Next Big Shift that Happened

- many then argued unsupervised *pre-training* phase not really needed…
- backprop is sufficient if done better
  - wider diversity in initial weights, try with many initial settings until you get learning
  - don’t worry much about exact learning rate, but add *momentum:* if moving fast in a given direction, keep it up for awhile
  - *Need a lot of data for deep net backprop*

Problems with Backprop for Deep Neural Networks

- Overfits both training data and the particular starting point
- Converges too quickly to a suboptimal solution, even with SGD (gradient from one example or “minibatch” of examples at one time)
- Need more training data and/or fewer weights to estimate, or other regularizer

Trick 1: Data Augmentation

- Deep learning depends critically on “Big Data” – need many more training examples than features
- Turn one positive (negative) example into many positive (negative) examples
- Image data: rotate, re-scale, or shift image, or flip image about axis; image still contains the same objects, exhibits the same event or action

Trick 2: Parameter (Weight) Tying

- Normally all neurons at one layer are connected to next layer

- Instead, have only *n* features feed to one specific neuron at next level (e.g., 4 or 9 pixels of image go to one hidden unit summarizing this “super-pixel”)

- Tie the 4 (or 9) input weights across all super-pixels… more data per weight

- Here weight matrix **W** notation especially useful

Weight Tying Example: Convolution

- Have a sliding window (e.g., square of 4 pixels, set of 5 consecutive items in a sequence, etc), and only the neurons for these inputs feed into one neuron, N1, at the next layer

- Slide this window over by some amount and repeat, feeding into another neuron, N2, etc.

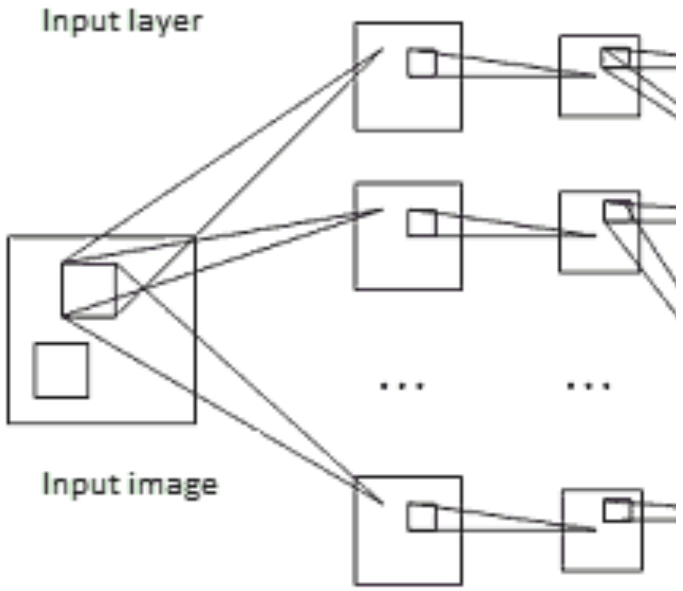
- Tie the input weights for N1, N2, etc., so they will all learn the same concept (e.g., diagonal edge)

- Repeat into new neurons N1', N2', etc., to learn other concepts.

Alternate Convolutional Layer with Pooling Layer

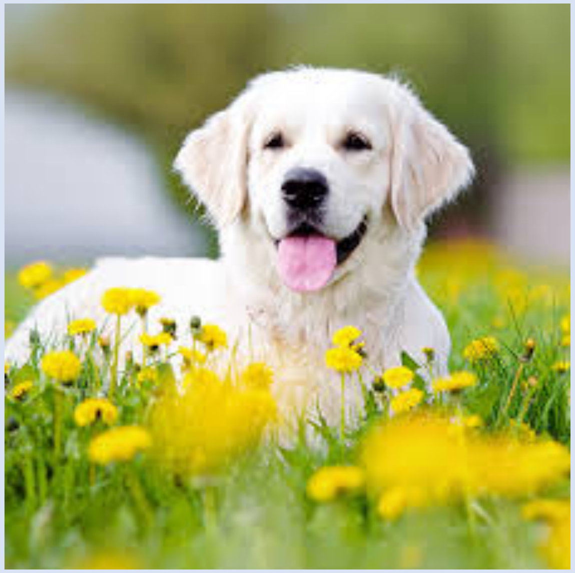
- Mean pooling: *k* nodes (e.g., corresponding to 4 pixels constituting a square in an image) are averaged to create one node (e.g., corresponding to one pixel) at the next layer.
- Max pooling: replace average with maximum
- Max pooling is like OR… true if the pattern appears anywhere in image; Min pooling is like AND.

Used in Convolutional Neural Networks for Vision Applications



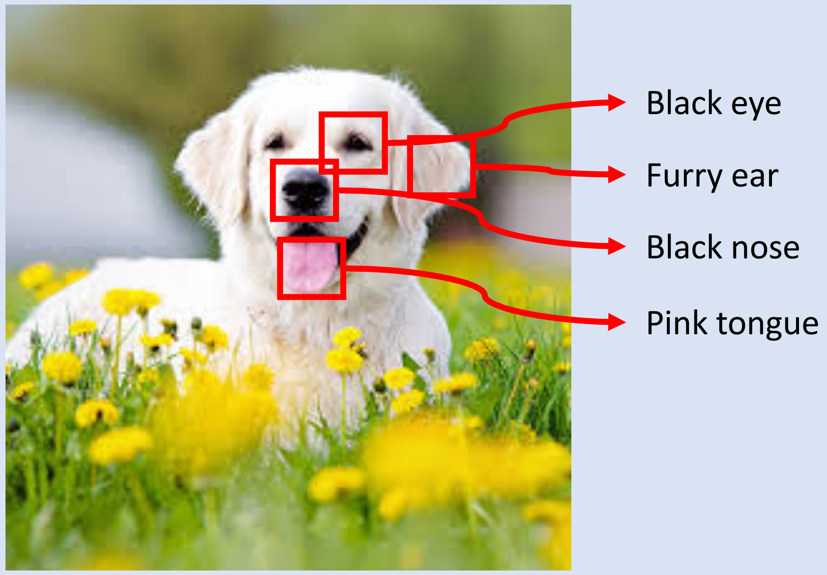
Convolutional Neural Networks (CNN)

- What makes a dog a dog?



Convolutional Neural Networks (CNN)

- What makes a dog a dog?
- Focus on the local features, build up global features



Convolution Operator (Already Learned)

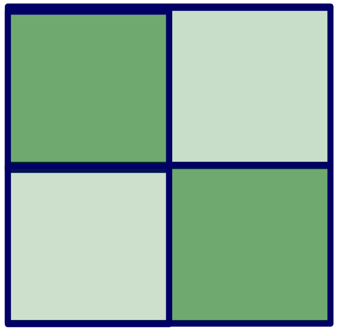
$$\mathbf{u}: \begin{array}{|c|c|c|c|}\hline 1 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 \\ \hline \end{array} \qquad \mathbf{w}: \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$$

Convolution Operator (Already Learned)

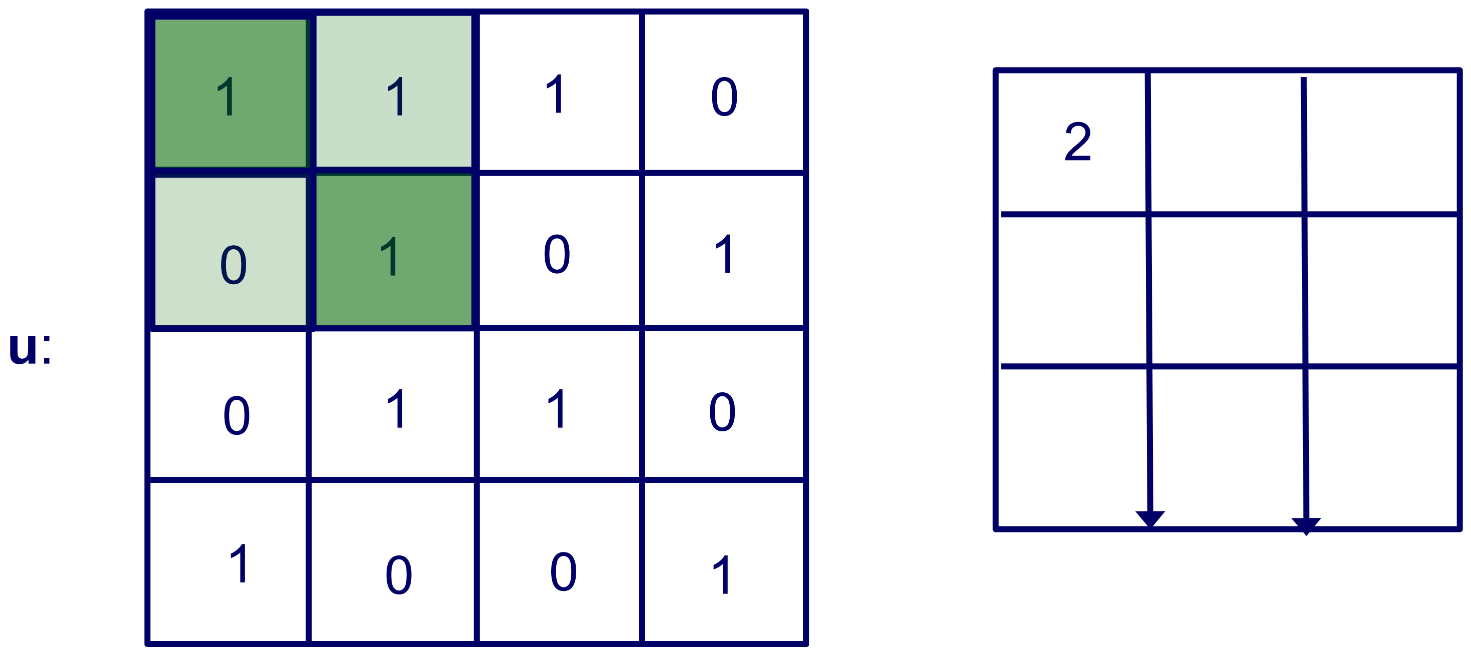
$$\mathbf{u}: \begin{array}{|c|c|c|c|}\hline 1 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 \\ \hline \end{array} \qquad \mathbf{w}: \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$$

Convolution Operator (Already Learned)

$$
\mathbf{u}: \begin{array}{|c|c|c|c|}
\hline
1 & 1 & 1 & 0 \\
\hline
0 & 1 & 0 & 1 \\
\hline
0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 \\
\hline
\end{array}
$$



Convolution Operator (Already Learned)

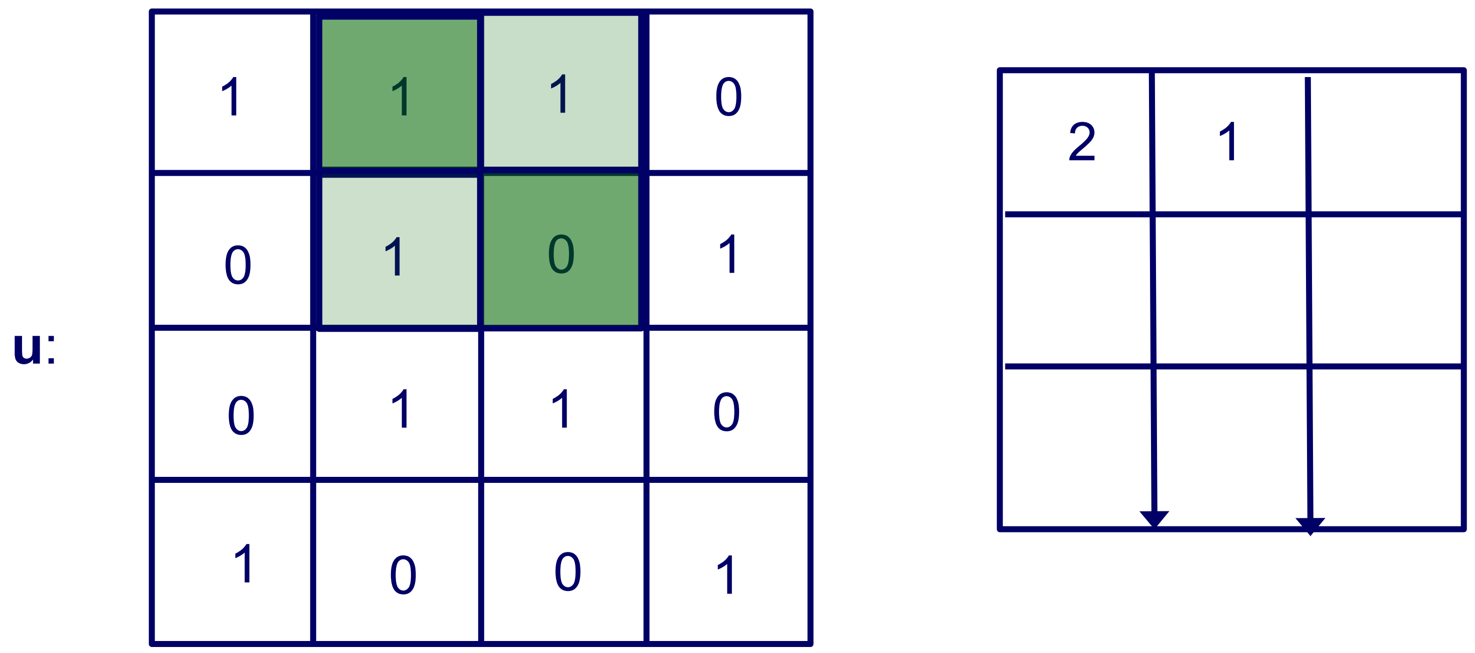


u:

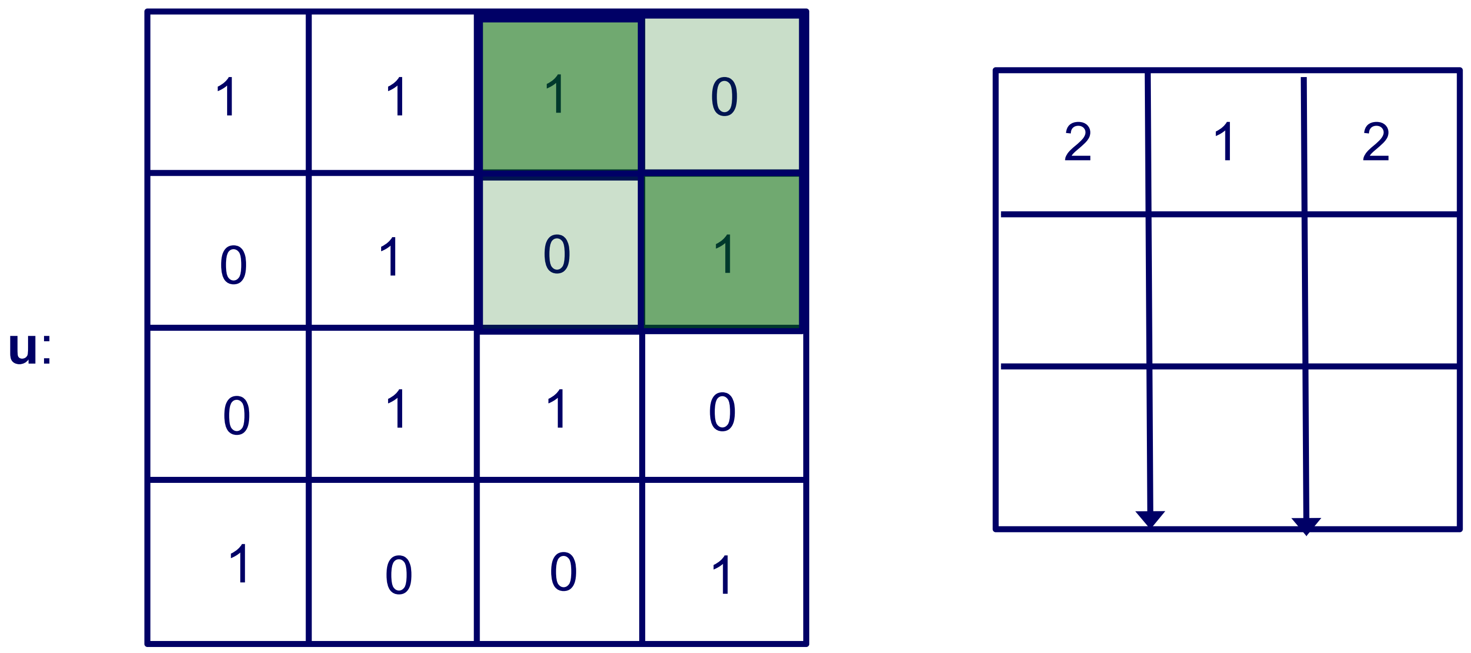
1	1	1	0
0	1	0	1
0	1	1	0
1	0	0	1

2		

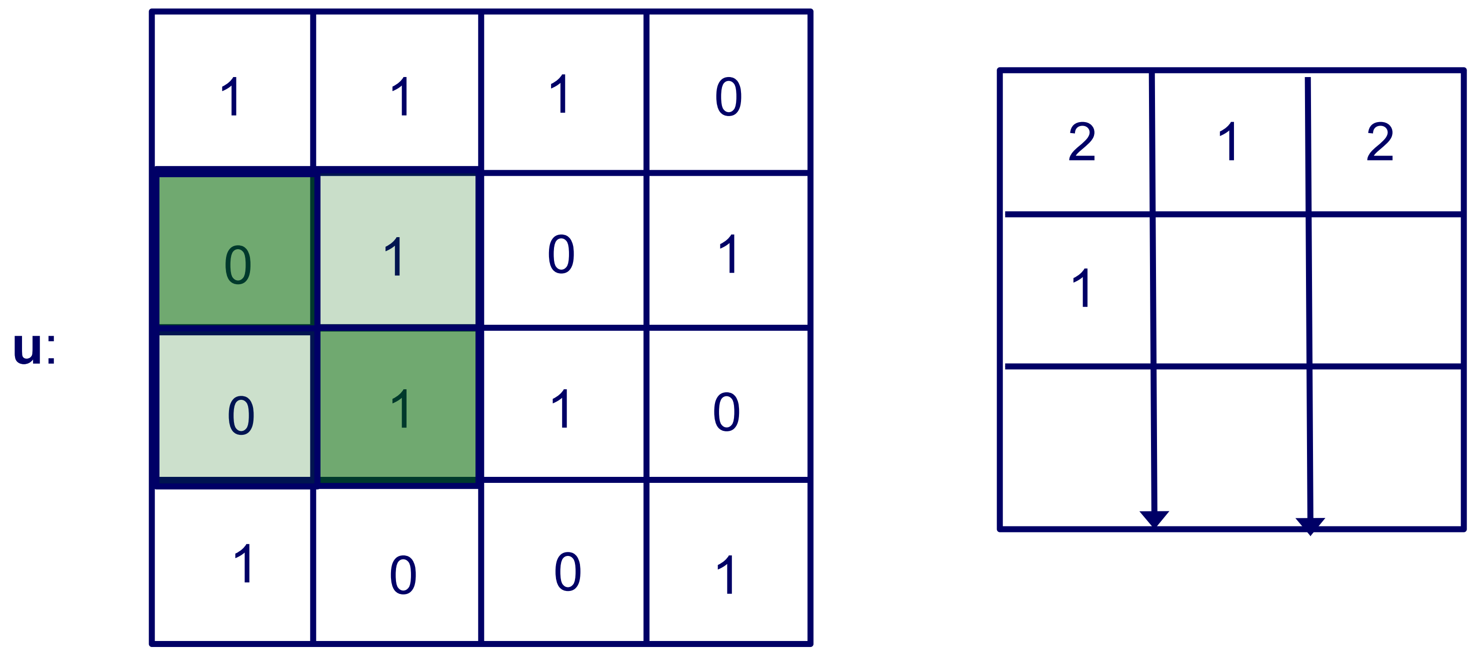
Convolution Operator (Already Learned)



Convolution Operator (Already Learned)



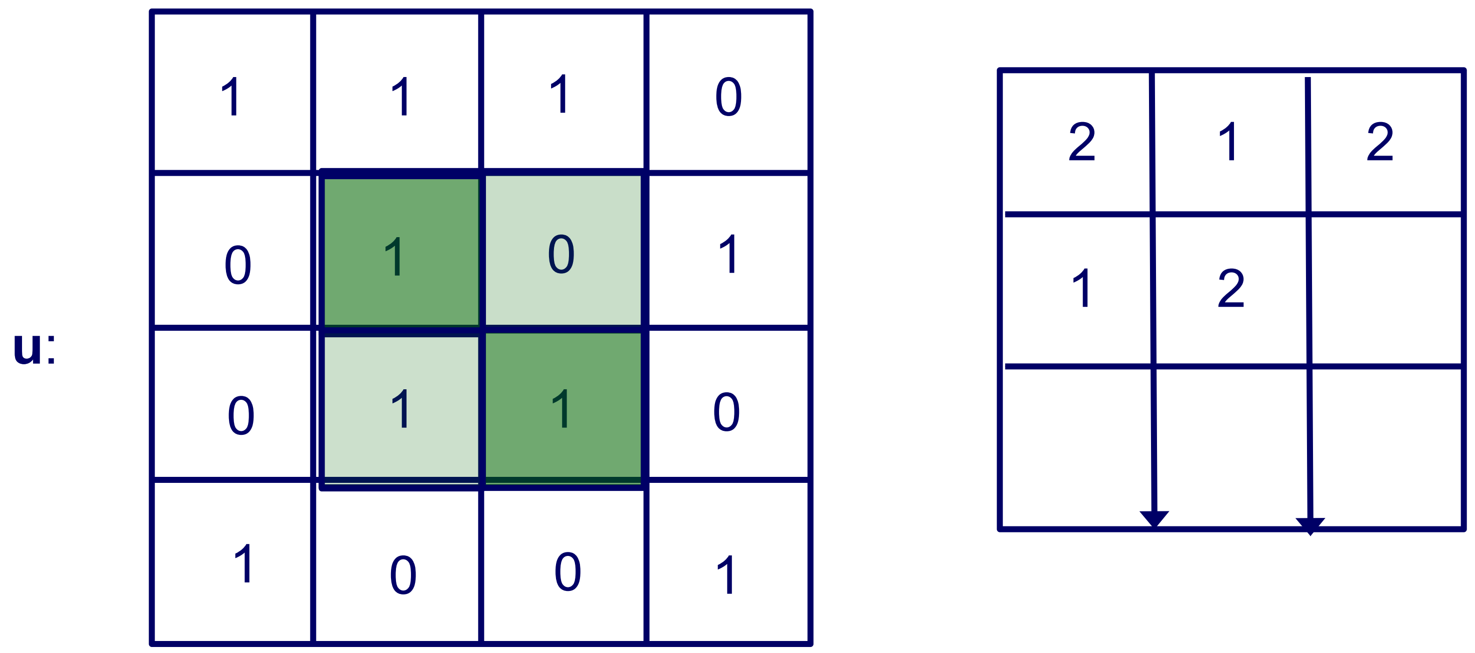
Convolution Operator (Already Learned)



1	1	1	0
0	1	0	1
0	1	1	0
1	0	0	1

2	1	2
1		

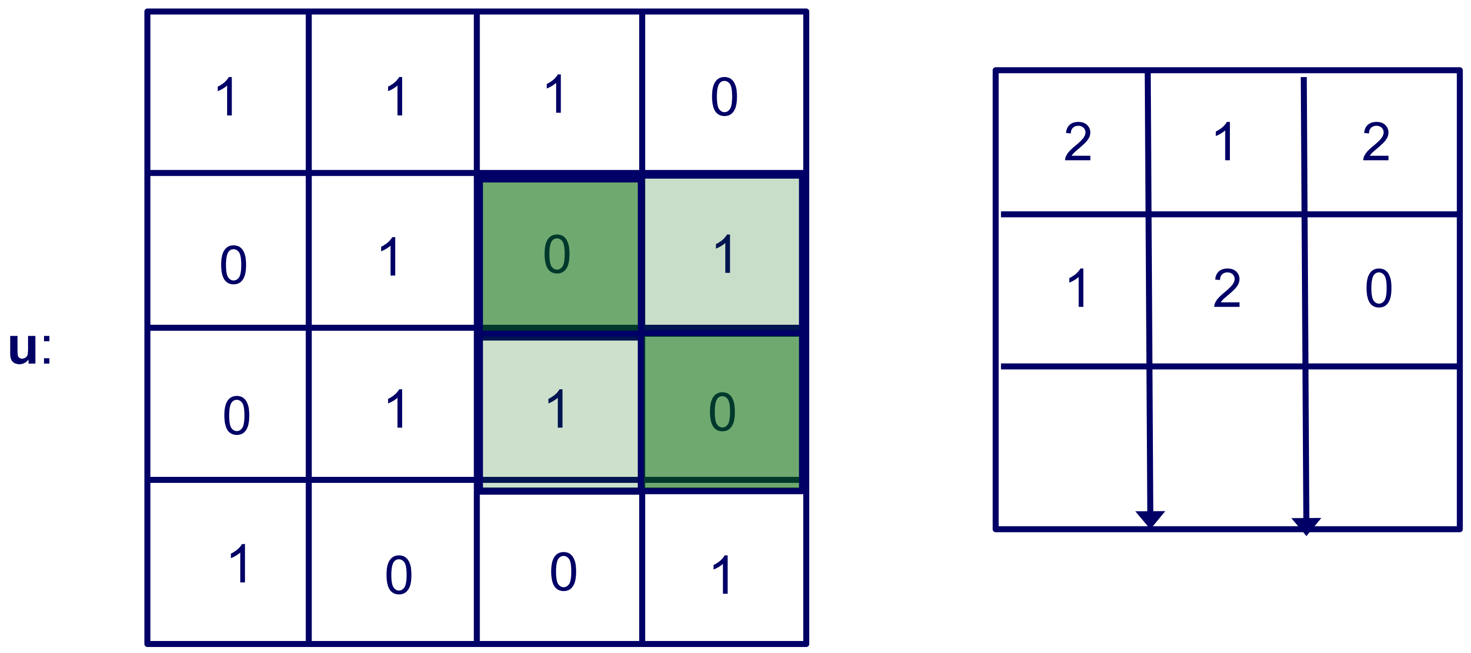
Convolution Operator (Already Learned)



1	1	1	0
0	1	0	1
0	1	1	0
1	0	0	1

2	1	2
1	2	

Convolution Operator (Already Learned)

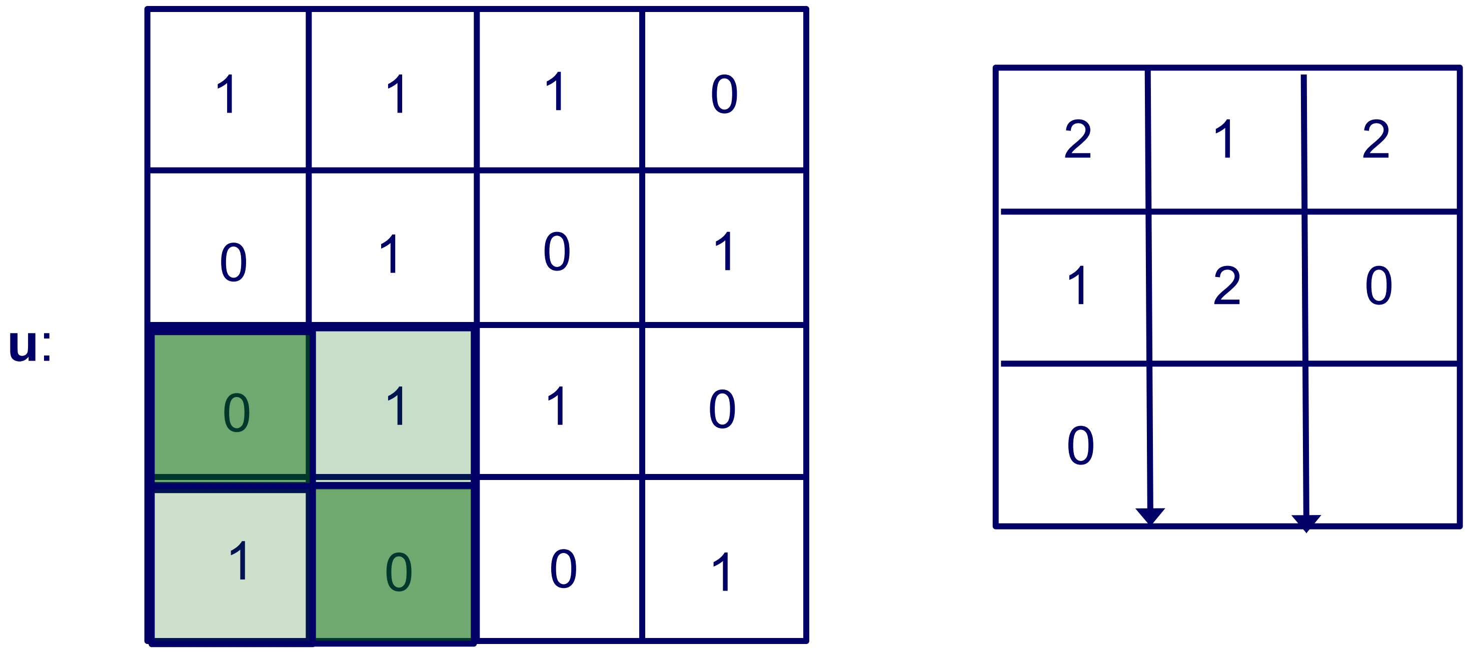


u:

1	1	1	0
0	1	0	1
0	1	1	0
1	0	0	1

2	1	2
1	2	0

Convolution Operator (Already Learned)



1	1	1	0
0	1	0	1
0	1	1	0
1	0	0	1

2	1	2
1	2	0
0		

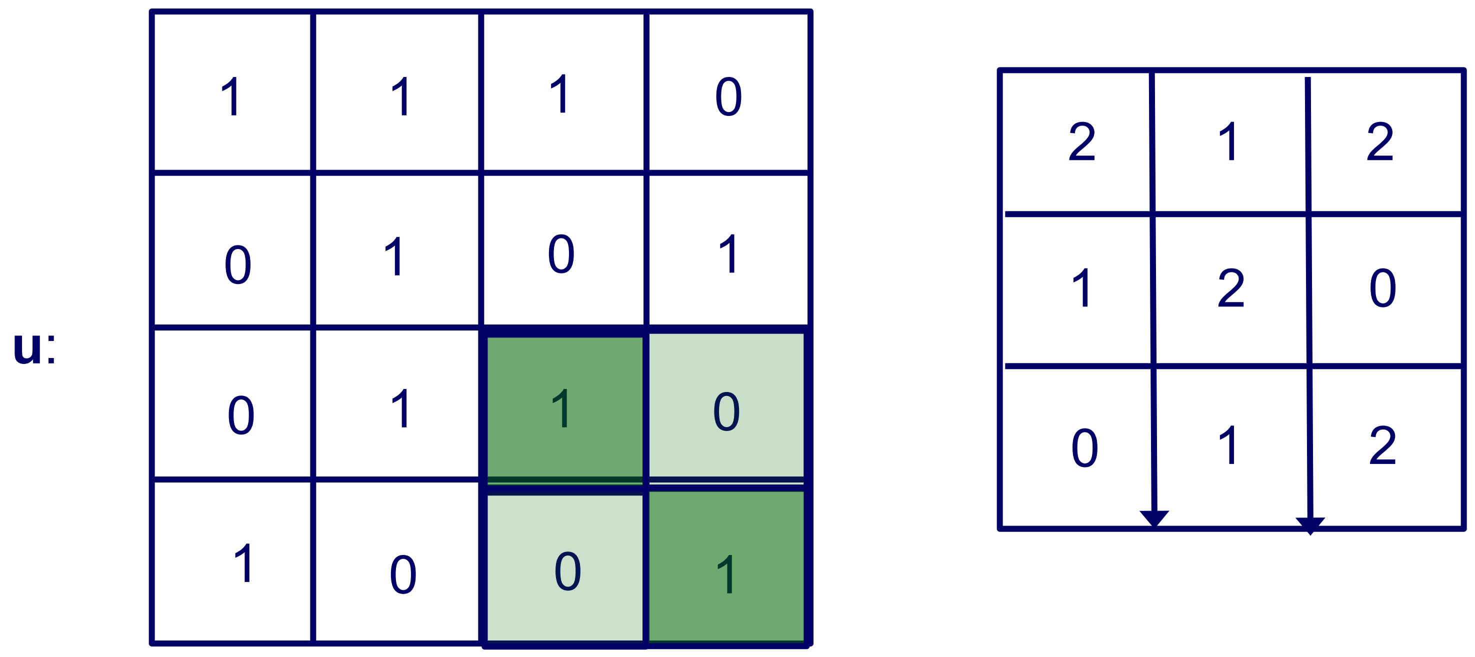
Convolution Operator (Already Learned)

u:

1	1	1	0
0	1	0	1
0	1	1	0
1	0	0	1

2	1	2
1	2	0
0	1	

Convolution Operator (Already Learned)



u:

1	1	1	0
0	1	0	1
0	1	1	0
1	0	0	1

2	1	2
1	2	0
0	1	2

Notes

- Repeating twice more and max pooling would yield an 8, which tells us there is a continuous diagonal edge (if we assume “1” is a darkened pixel)
- Many ways to define how to handle boundary cases… we ignored them (threw them away)
- We used a 2x2 *span* and a *stride* of 1 (overlapping)… what would stride of 2 give us?
- We didn’t *learn* the window template **w** (pattern or *channel*); we might even want to learn *more than one*

Same as Discrete Mathematical Convolution (Thanks to Yingyu Liang!)

- Given array u_t and w_t, their convolution is a function s_t

$$s_t = \sum_{a=-\infty}^{+\infty} u_a w_{t-a}$$

- Written as

$$s = (u * w) \quad \text{or} \quad s_t = (u * w)_t$$

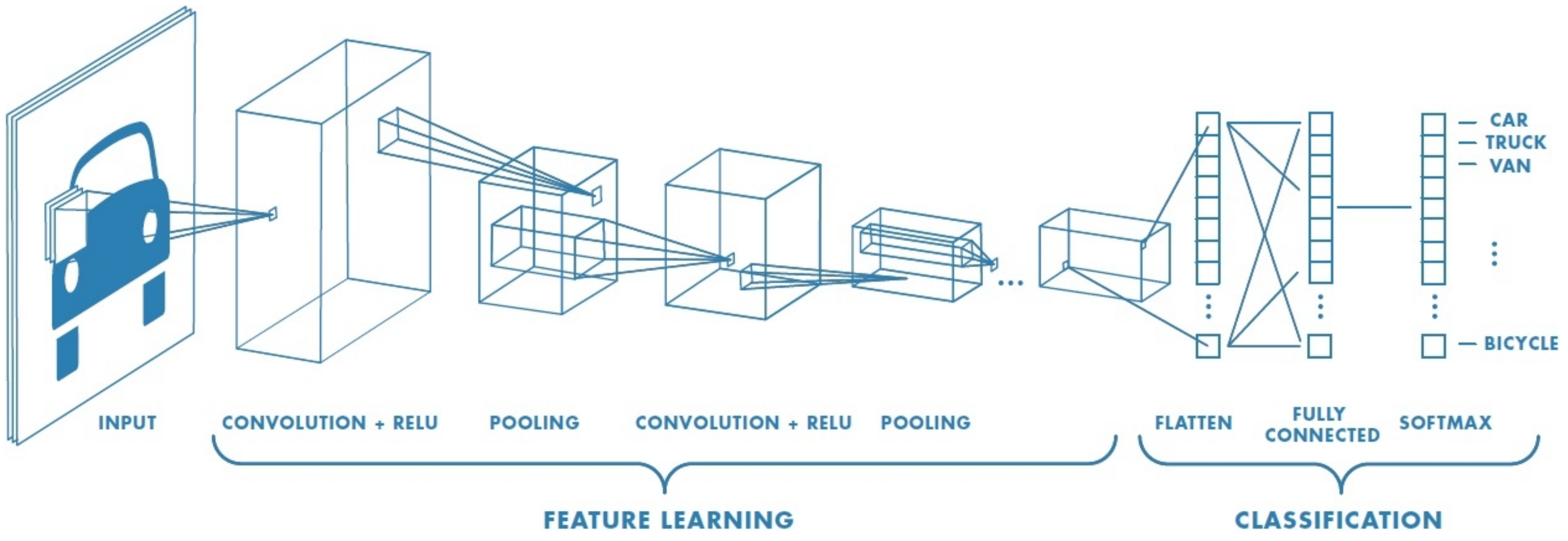
- When u_t or w_t is not defined, assumed to be 0

To *Learn* Convolution

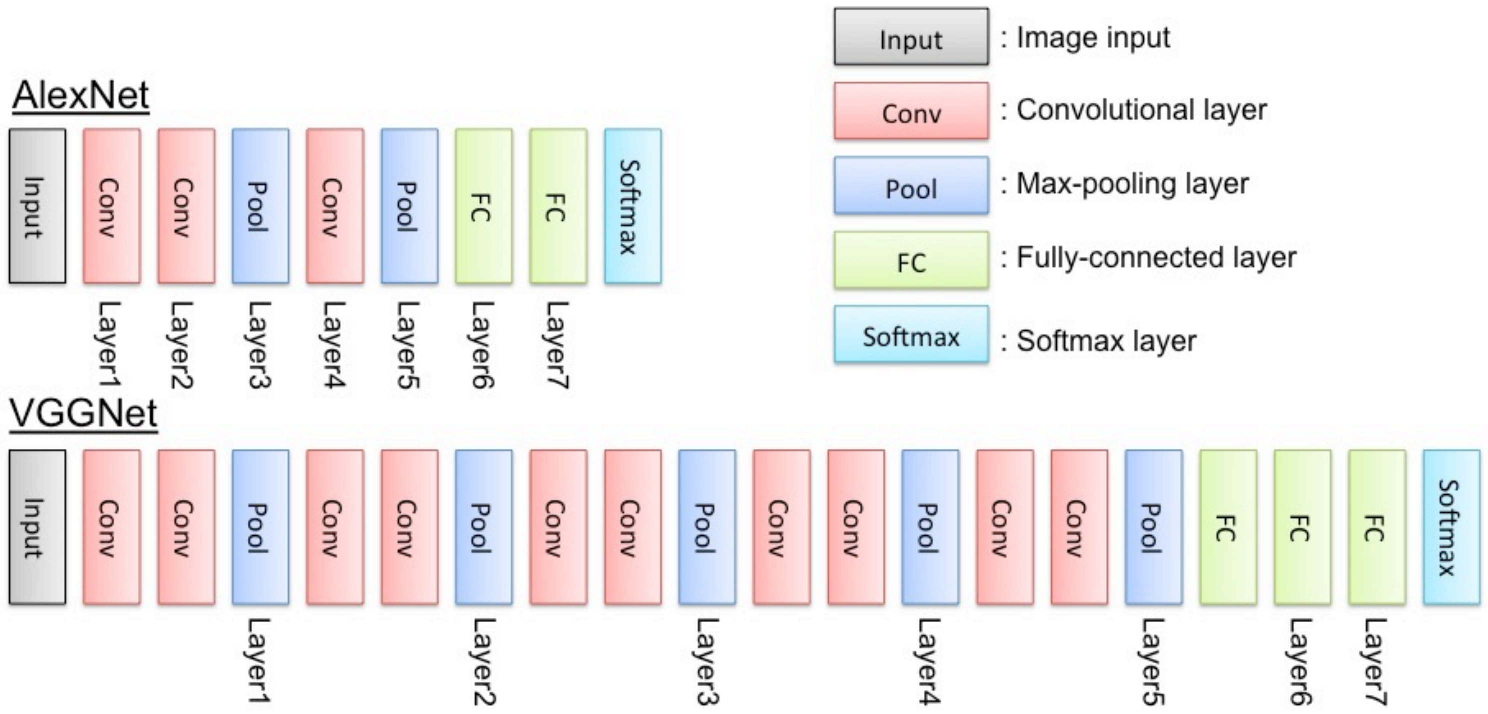
- Choose span and stride
- Choose how to handle boundaries (e.g., padding)
- Choose *how many* window patterns (channels)
- For each channel, and for each overlay on input:
  - create a hidden unit
  - tie units for the same pattern across different locations together -- their input weights will all agree (though their inputs will not)

Convolutional Neural Networks (CNN)

- Focus on the local features, build up global features



Convolutional Neural Networks


Input : Image input
Conv : Convolutional layer
Pool : Max-pooling layer
FC : Fully-connected layer
Softmax : Softmax layer
AlexNet
Input Conv Conv Pool Conv Pool FC FC Softmax
Layer1 Layer2 Layer3 Layer4 Layer5 Layer6 Layer7
VGGNet
Input Conv Conv Pool Conv Conv Pool Conv Conv Pool Conv Conv Pool Conv Conv Pool FC FC FC Softmax
Layer1 Layer2 Layer3 Layer4 Layer5 Layer6 Layer7

AlexNet: Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks.", NIPS 2012
VGGNet: Karen Simonyan, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition.", ICLR 2015

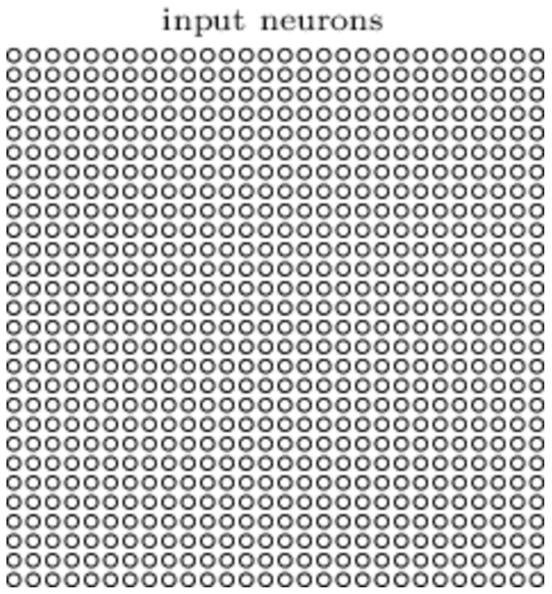
Convolutional neural nets

- well suited to tasks in which the input has spatial structure, such as images or sequences

- based on four key ideas
  - local receptive fields
  - weight sharing
  - pooling
  - multiple layers of hidden units

Convolutional neural nets

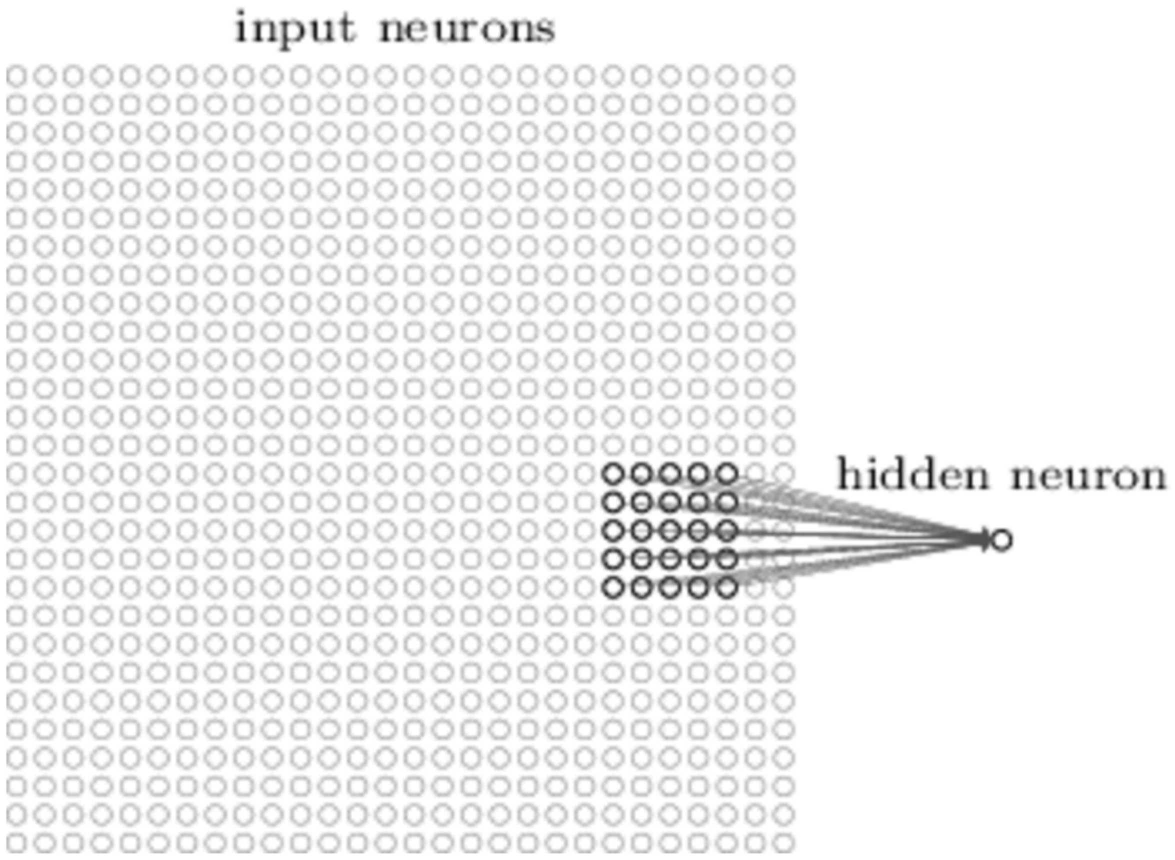
- suppose we have a task in which are instances are 28 × 28 pixel images
- we can represent each using 28 × 28 input units



[Figure from neuralnetworksanddeeplearning.com]

Convolutional neural nets

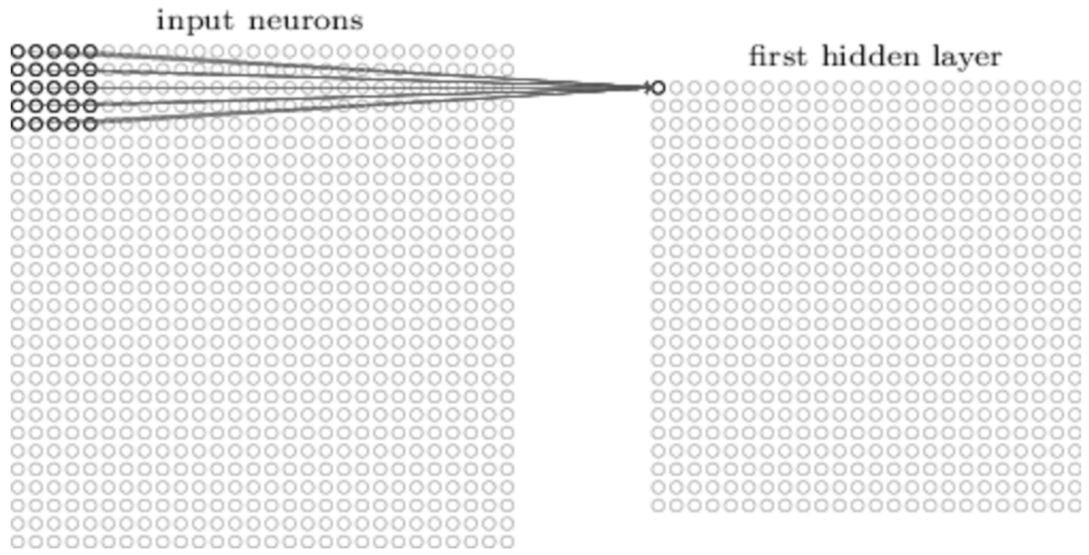
- we can connect hidden units so that each has a *local receptive field* (e.g. a 5 $\times$ 5 patch of the image).

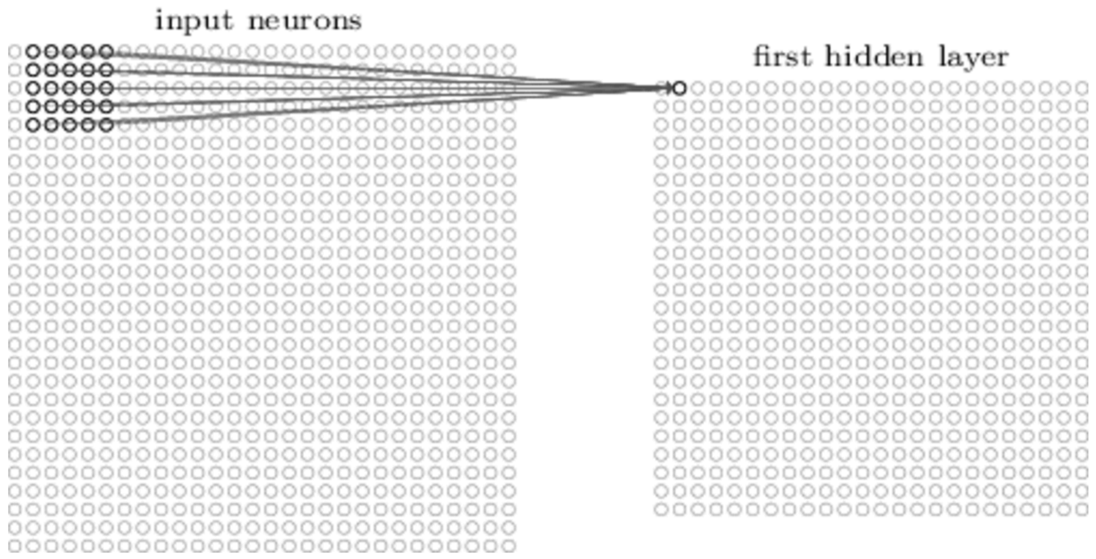


[Figure from neuralnetworksanddeeplearning.com]

Convolutional neural nets

- we can have a set of these units that differ in their local receptive field
- all of the units share the same set of weights
- so the units detect same "feature" in the image, but at different locations

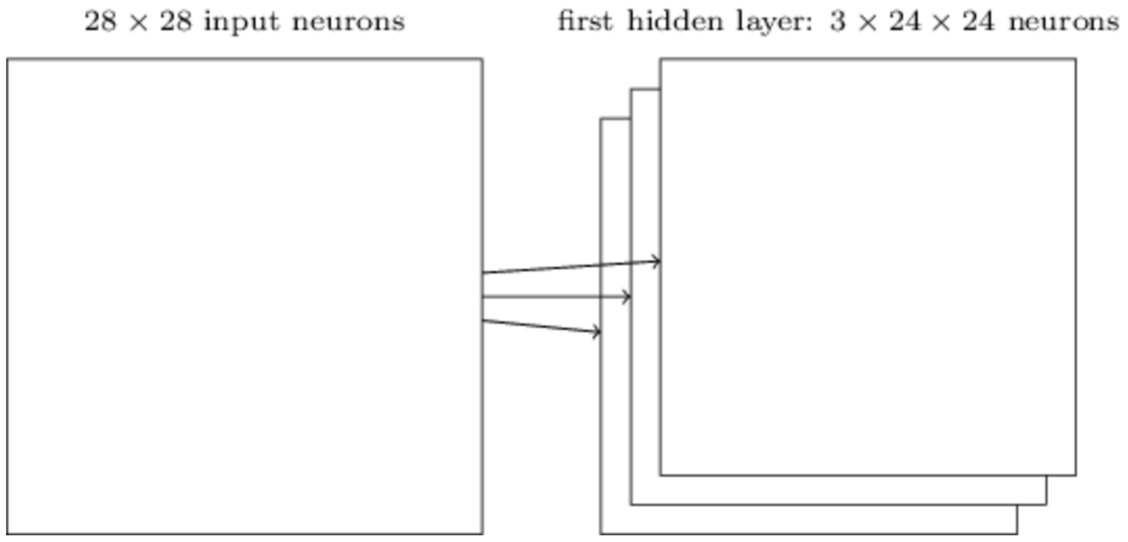




[Figure from neuralnetworksanddeeplearning.com]

Convolutional neural nets

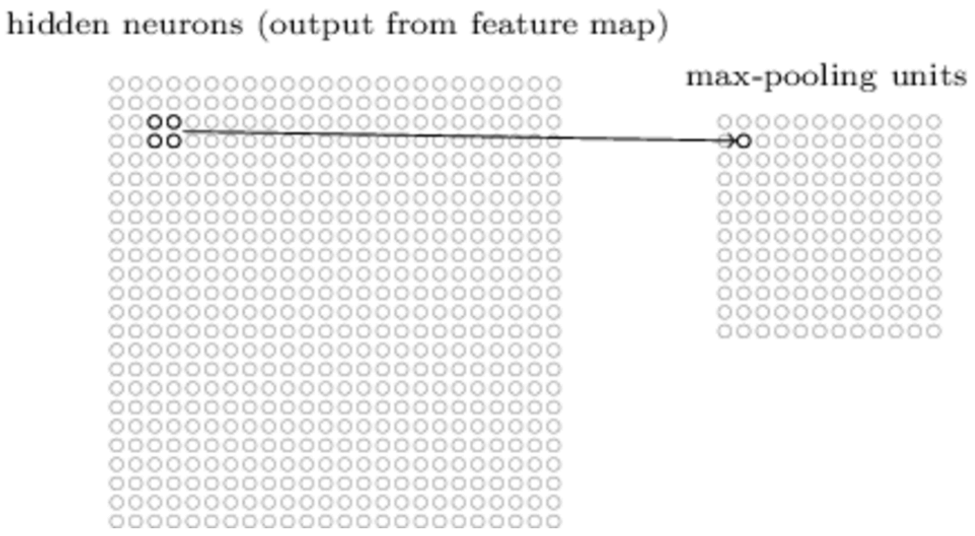
- a set of units that detect the same "feature" is called a *feature map*
- typically we'll have multiple feature maps in each layer



[Figure from neuralnetworksanddeeplearning.com]

Convolutional neural nets

- feature-map layers are typically alternated with pooling layers
- each unit in a pooling layer outputs a max, or similar function, of a subset of the units in the previous layer



$$f(\boldsymbol{x}) = max(x_1 \ldots x_i \ldots)$$

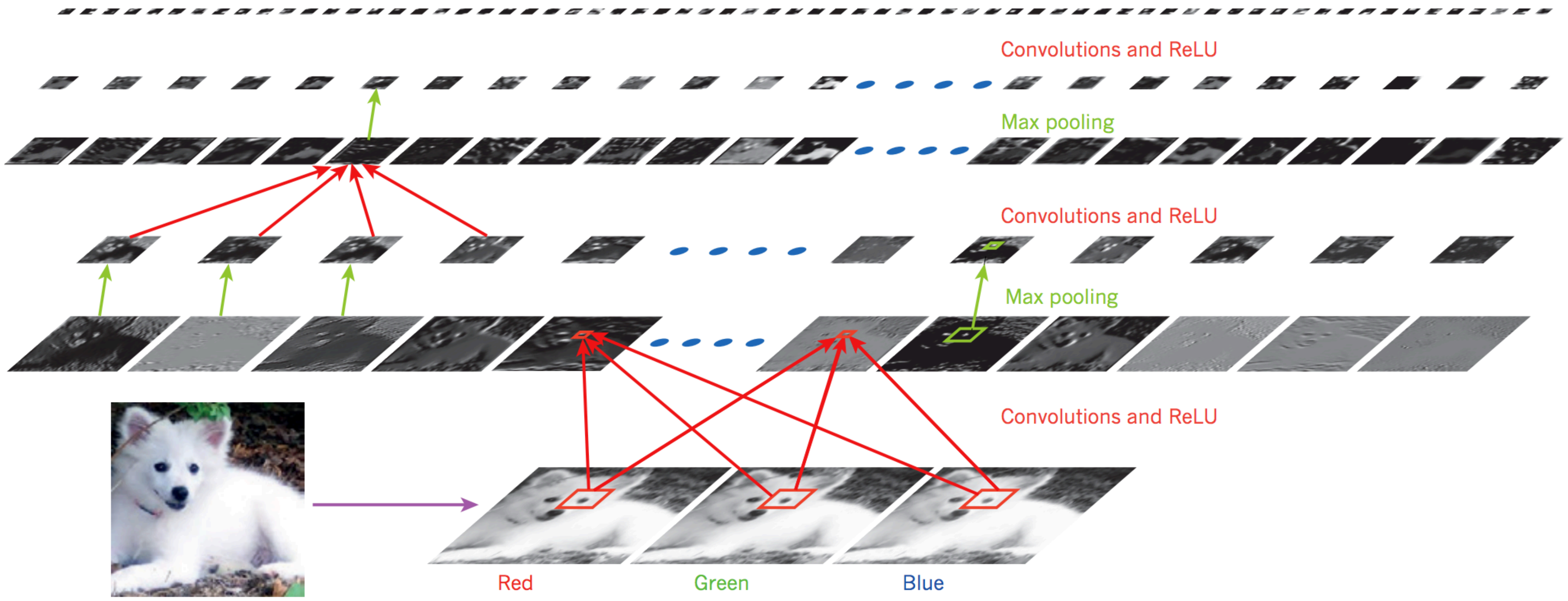
$$f(\boldsymbol{x}) = log \sum_i e^{x_i}$$

$$f(\boldsymbol{x}) = \sqrt{\sum_i x_i^2}$$

[Figure from neuralnetworksanddeeplearning.com]

Convolutional neural nets

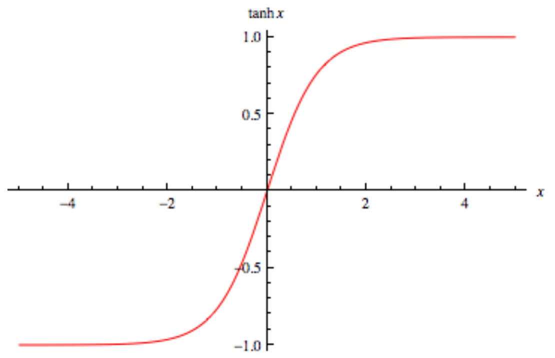
- alternating layers of convolutional and pooling layers can be stacked



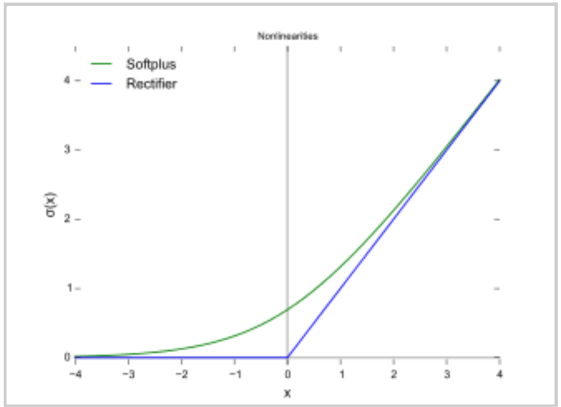
[Figure from LeCun et al., *Nature* 2015]

Trick 3: Alternative Activations

- tanh: $(e^{2x}-1)/(e^{2x}+1)$ hyperbolic tangent



- ReLU: max(0,x) or $ln(1+e^x)$ rectified linear unit or softplus



Plot of the rectifier (blue) and softplus (green) functions near $x = 0$

Trick 4: Alternative Error Function

- Example: Cross-entropy

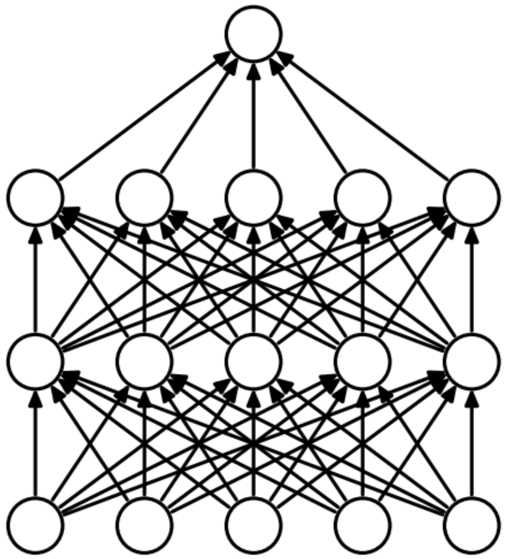
$$C = -\frac{1}{n}\sum_{x}[y \ln \mathrm{o} + (1-y)\ln(1-\mathrm{o})]$$

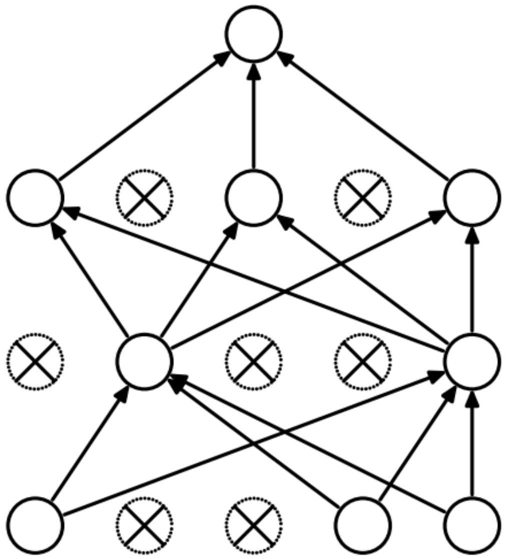
Trick 5: Dropout Training

- Build some redundancy into the hidden units
- Essentially create an “ensemble” of neural networks, but without high cost of training many deep networks
- ***Dropout training***…

Dropout training

- On each training iteration, drop out (ignore) 50% of the units (or other 90%, or other) by forcing output to 0 during forward pass
- Ignore for forward & backprop (all training)

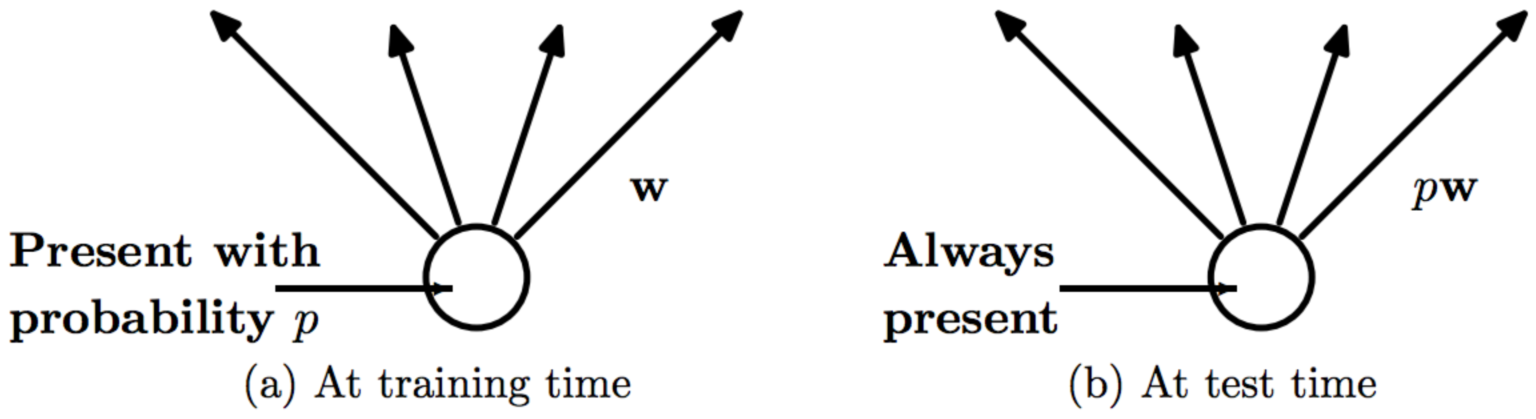

(a) Standard Neural Net


(b) After applying dropout.

Figures from Srivastava et al., *Journal of Machine Learning Research* 2014

At Test Time

- Final model uses all nodes
- Multiply each weight from a node by fraction of times node was used during training



Figures from Srivastava et al., *Journal of Machine Learning Research* 2014

Trick 6: Batch Normalization

- If outputs of earlier layers change greatly on one round for one mini-batch, then neurons at next levels can’t keep up: they output all high (or all low) values

- Next layer doesn’t have ability to change its outputs with learning-rate-sized changes to its input weights

- We say the layer has “saturated”

Another View of Problem

- In ML, we assume future data will be drawn from same probability distribution as training data
- For a hidden unit, after training, the earlier layers have new weights and hence generate input data for this hidden unit from a *new* distribution
- Want to reduce this *internal covariate shift* for the benefit of later layers

```
**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
           Parameters to be learned: $\gamma$, $\beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean

    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$          // mini-batch variance

    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                      // normalize

    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

- On forward pass
  - Process each layer one at a time
  - Each input to each neuron (activation) on each minibatch has its own μ and σ
  - Each input to each neuron (regardless of minibatch) has its own γ and β (shared across all minibatches)

- On backpropagation
  - Each γ and β are just two additional parameters in gradient, feeding into a non-linear activation such as a ReLU or Sigmoid
  - In a CNN, we tie all γs across an entire layer or a feature map, e.g., upper left corner of a 2x2 sliding window, (and same for β, μ and σ)

- At Test Time (usage time), use the trained γ and β. μ and σ are averaged over all minibatches, or as written here:

$$y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$$

Comments on Batch Normalization

- First three steps are just like standardization of input data, but with respect to only the data in mini-batch. Can take derivative and incorporate the learning of last step parameters into backpropagation.

- Note last step can completely un-do previous 3 steps

- But if so this un-doing is driven by the *later* layers, not the *earlier* layers; later layers get to “choose” whether they want standard normal inputs or not