CSCE 633: Machine Learning

Lecture 28: Neural Networks

Texas A&M University

10-28-19

Last Time

- Perceptron
- PROJECTS!

Goals of this lecture

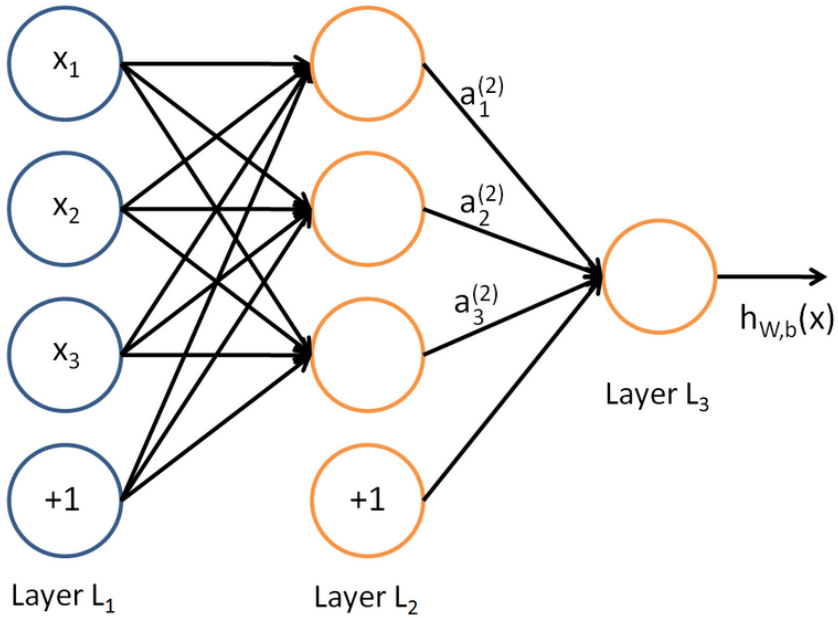
- Deep Neural Networks

Exam Comments

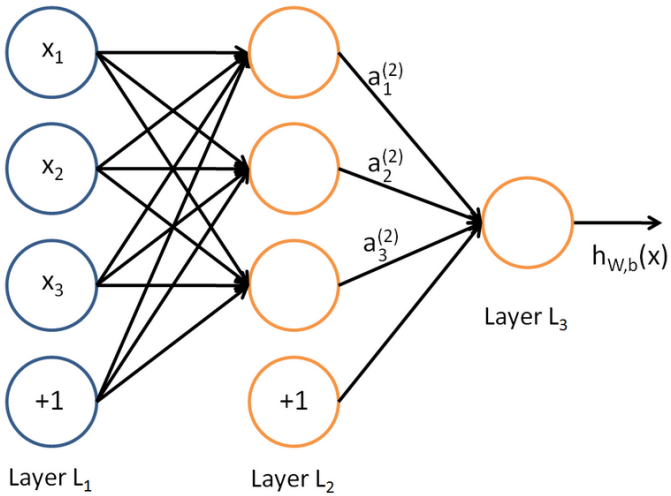
- AVG: 72.3, STD DEV: 14.6
- Q1a: 11.6, 4.0 ** READ THE QUESTIONS!
- Q1b: 12.3, 3.4
- Q1c: 16.9, 4.1
- Q2a: 6.3, 2.4
- Q2b: 10.6, 4.3
- Q2c: 9.3, 4.7
- Q2d: 5.8, 3.3

Multilayer Perceptron

- Type of feedforward neural network
- Can model non-linear associations
- "Multi-level combination" of many perceptrons



Multilayer Perceptron: Representation



$$\alpha_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$

$$\alpha_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$

$$\alpha_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \alpha_1^{(3)} = f(W_{11}^{(2)}\alpha_1^{(2)} + W_{12}^{(2)}\alpha_2^{(2)} + W_{13}^{(2)}\alpha_3^{(2)} + b_1^{(2)})$$

Terminology

$W_{ij}^{(l)}$: connection between unit j in layer l to unit i in layer $l+1$

$\alpha_i^{(l)}$: activation of unit i in layer l

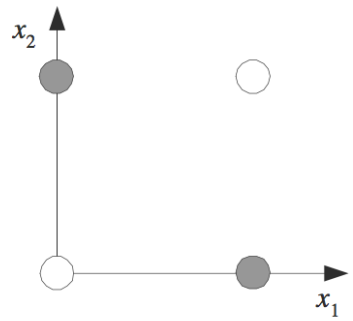
$b_i^{(l)}$: bias connected with unit i in layer $l+1$

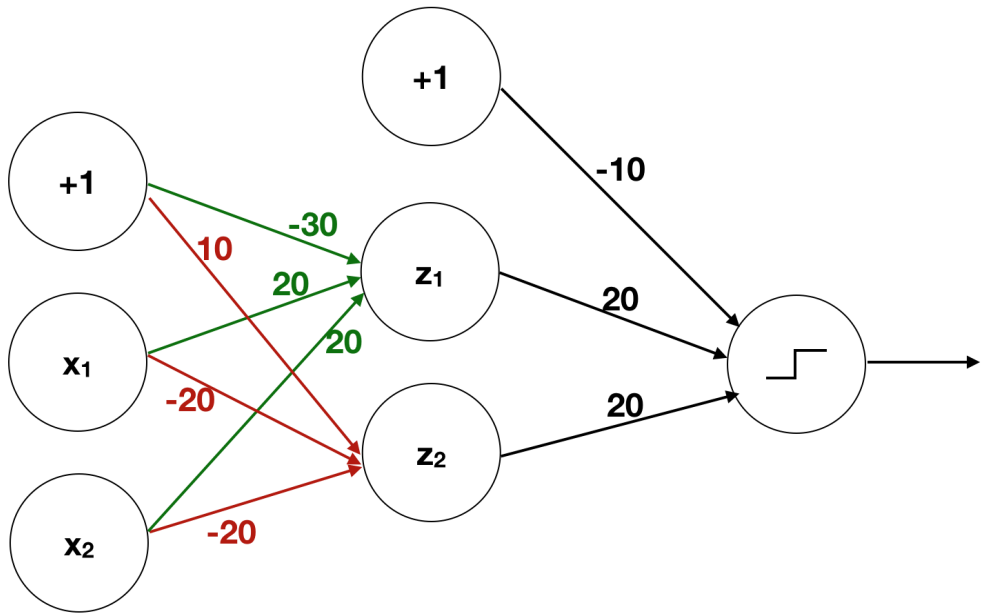
Forward propagation: The process of propagating the input to the output through the activation of inputs and hidden units to each node

Multilayer Perceptron: Approximating non-linear functions

Example: Boolean XOR with multilayer perceptrons

x_1	x_2	z_1	z_2	r
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1





Backpropagation

Multilayer Perceptron: Representation

- Input: $\mathbf{x} \in \mathbb{R}^D$
- Output:
  $y \in \{0, 1\}$ or $y \in \{1, \ldots, K\}$ (classification)
  $y \in \mathbb{R}$ or $y \in \mathbb{R}^K$ (regression)
- Training data: $\mathcal{D}^{train} = \{(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_N}, y_N)\}$
- Model: $h_{\mathbf{W},\mathbf{b}}(\mathbf{x})$
  represented through forward propagation (see previous slides)
- Model parameters: weights $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(\mathbf{L})}$ and biases $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(\mathbf{L})}$

Multilayer Perceptron: Evaluation criterion

$J(\mathbf{W}, \mathbf{b}, \mathcal{D}^{train}) = \frac{1}{2} \| h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) - y \|_2^2$ (regression)

$J(\mathbf{W}, \mathbf{b}, \mathcal{D}^{train}) = y \log h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) + (1 - y) \log(1 - h_{\mathbf{W},\mathbf{b}}(\mathbf{x}))$ (classification)

Multilayer Perceptron: Evaluation criterion

Regression

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M}\frac{1}{2}\|h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) - y_n\|_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

Classification

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M}(y_n \log h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) + (1-y_n)\log(1-h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n})))$$
$$+ \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

We will perform gradient descent

Backpropagation

Gradient descent for regression

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M}\frac{1}{2}\|h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) - y_n\|_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

$$W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$$

$$b_i^{(l)} := b_i^{(l)} - \alpha\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$$

Note: Initialize the parameters randomly $\rightarrow$ symmetry breaking

Use backpropagation to compute partial derivatives $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$ and $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$

Backpropagation

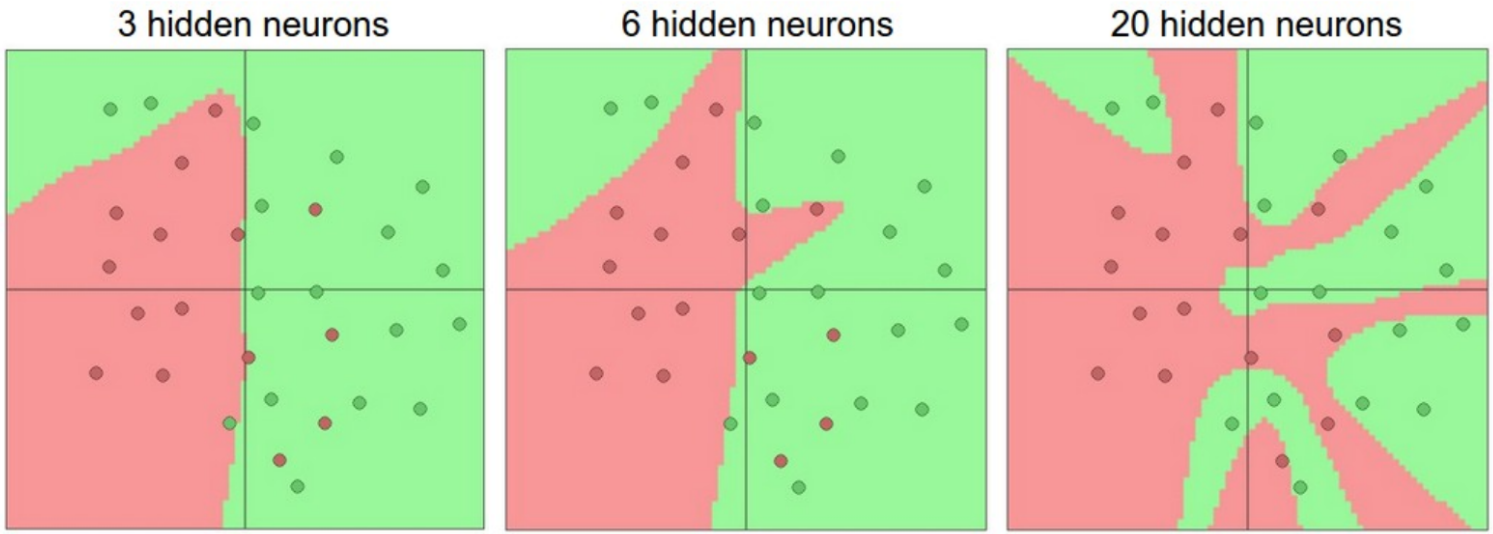
Implementation

- For each node i in output layer L
  - $\delta_i^{(L)} = (\alpha_i^{(L)} - y_n) f'(z_i^{(L)})$
- For each node i in layer $l = L-1, L-2, \ldots, 2$
  - Hidden nodes: $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$
- Compute the desired partial derivatives as:
  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}} = \alpha_j^{(l)} \delta_i^{(l+1)}$
  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}} = \delta_i^{(l+1)}$
- Update the weights as:
  $W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$
  $b_i^{(l)} := b_i^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$

Determining number of layers and their sizes

Implementation

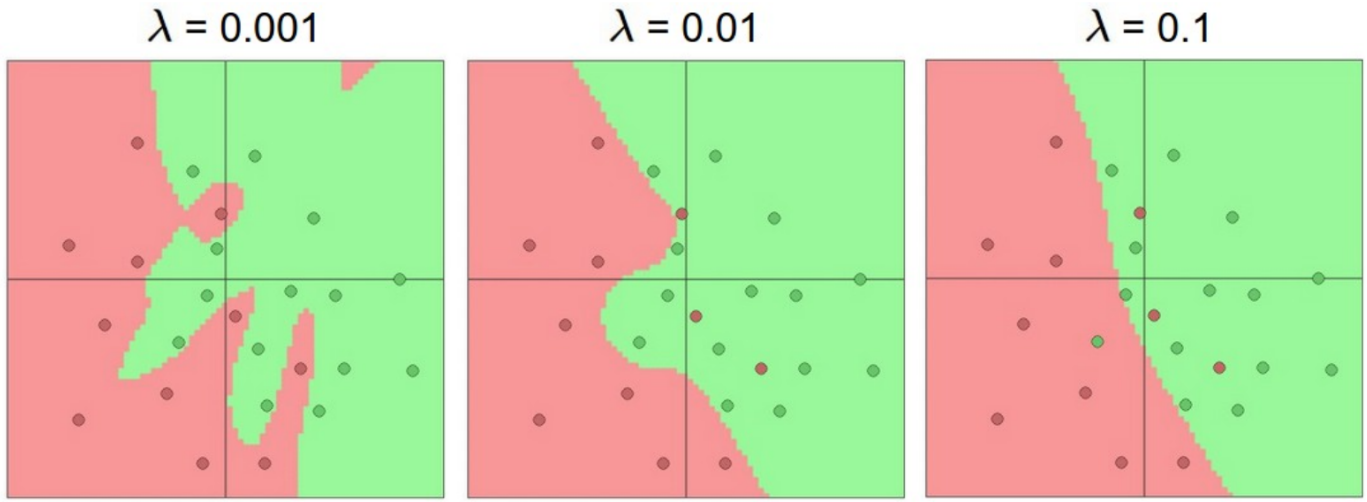
- The capacity of the network (i.e. the number of representable functions) increases as we increase the number of layers
- How to avoid overfittting?



Determining number of layers and their sizes

How to avoid overfitting

- Limit # layers and #hidden units per layers
- Early stopping: start with small weights and stop learning early
- Weight decay: penalize large weights (regularization)
- Noise: add noise to the weights
- Add constraints to the weights



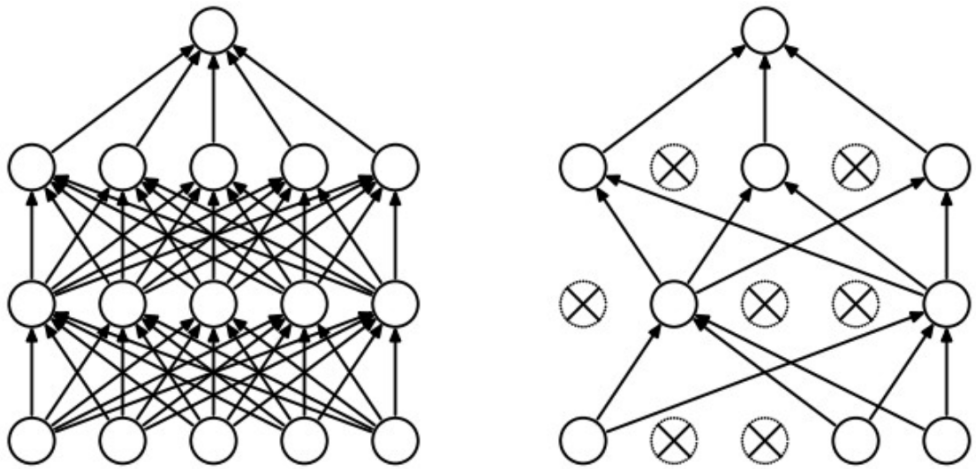
The effects of regularization strength: Each neural network above has 20 hidden neurons, but changing the regularization strength makes its final decision regions smoother with a higher regularization. You can play with these examples in this ConvNetsJS demo.

`http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html`

Determining number of layers and their sizes

How to avoid overfitting

- An alternative method that complements the above is dropout
- While training, dropout keeps a neuron active with some probability p (a hyperparameter), or sets it to zero otherwise



(a) Standard Neural Net

(b) After applying dropout.

Determining number of layers and their sizes

How to chose the number of layers and nodes

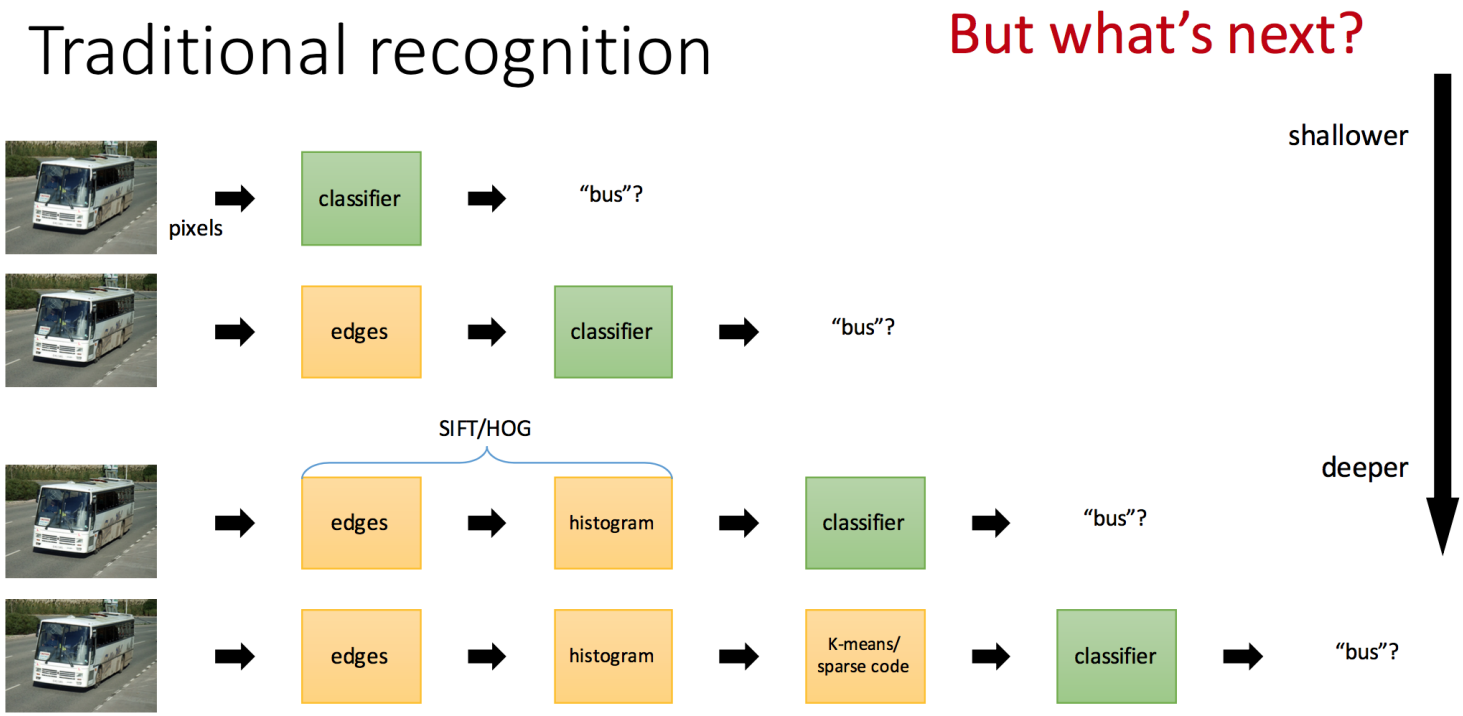
- No general rule of thumb, this depends on:
  - Amount of training data available
  - Complexity of the function that is trying to be learned
  - Number of input and output nodes
- If data is linearly separable, you don't need any hidden layers at all
- Start with one layer and hidden nodes proportional to input size
- Gradually increase

Activation Function

Transforms the activation level of a node (weighted sum of inputs) to an output signal

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent: $s(x) = \tanh(x) = 2\sigma(2x) - 1$
- Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$
- Leaky ReLU: $f(x) = (ax) \cdot \mathbb{I}(x < 0) + (x) \cdot \mathbb{I}(x \geq 0)$ (e.g. $a = 0.01$)

Deep neural networks: Motivation



Deep neural networks: Motivation

Deep Learning

Specialized components, domain knowledge required

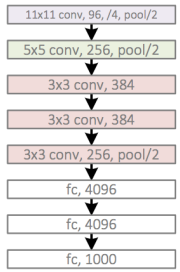
Generic components ("layers"), less domain knowledge

Repeat elementary layers => Going deeper

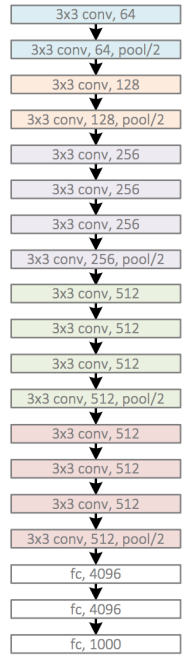
- End-to-end learning
- Richer solution space

Deep neural networks: Motivation

AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)

Deep neural networks: Motivation

Why go deep?

- Deep Representations might allow for a hierarchy or representation
  - Non-local generalization
  - Comprehensibility
- Multiple levels of latent variables allow combinatorial sharing of statistical strength
- Deep architectures work well (vision, audio, NLP, etc.)!

[Contents for the following slides have been summarized from the NIPS 2010, CVPR 2012, and Stanford Deep Learning Tutorials]

Deep neural networks: Motivation

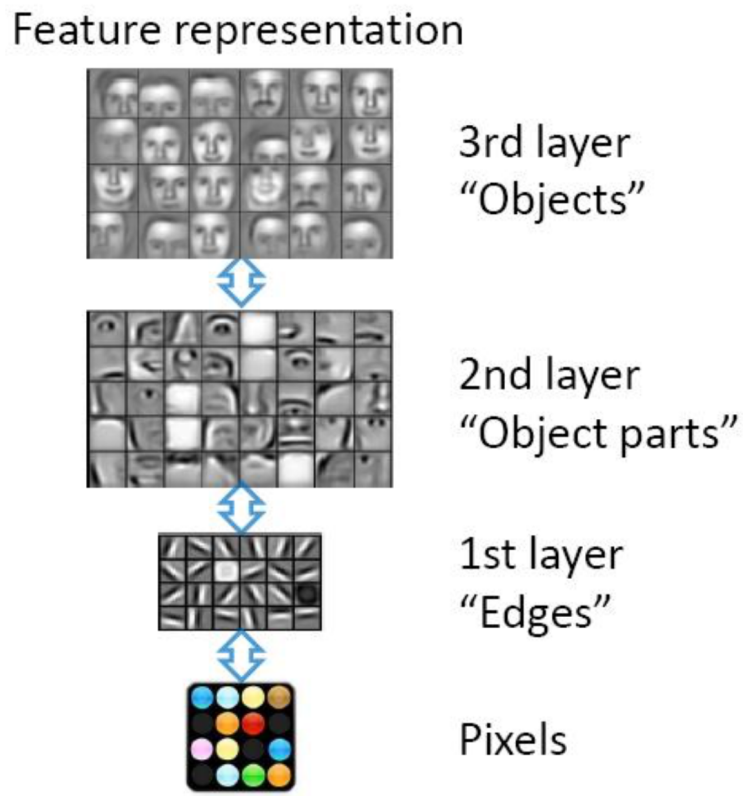
Key ideas of (deep) neural networks

- Learn features from data
- Use differentiable functions that produce features efficiently
- End-to-end learning: no distinction between feature extractor and classifier
- "Deep" architectures: cascade of simpler non-linear modules

Deep neural networks: Motivation

Different levels of abstraction:Hierarchical learning

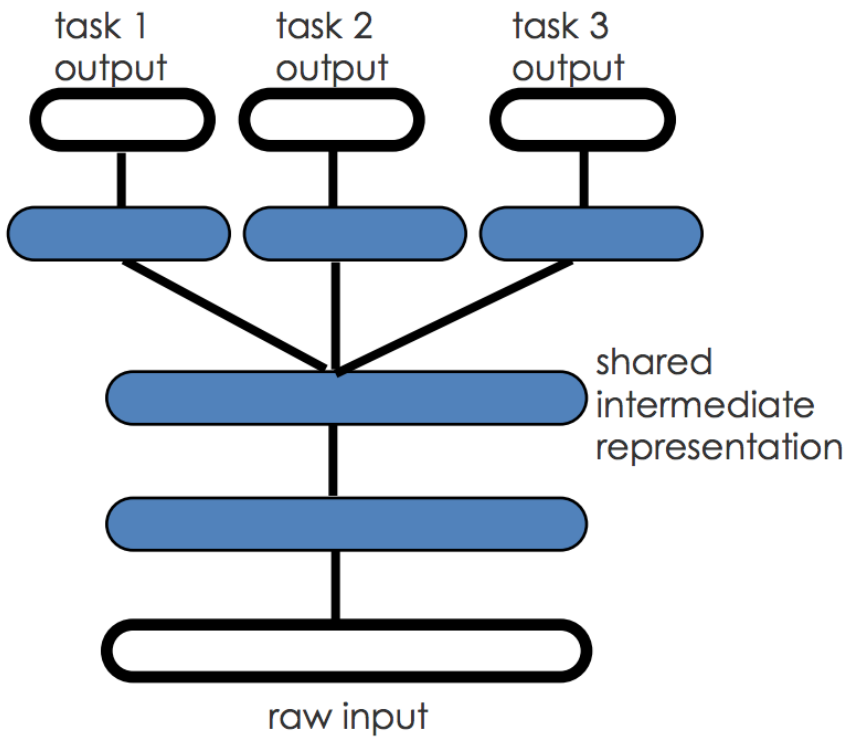
- Natural progression from low level to high level structure as seen in natural complexity
- Easier to monitor what is being learnt and to guide the machine to better subspaces
- A good lower level representation can be used for many distinct tasks



Deep neural networks: Motivation

Shared low-level representations

- Multi-task learning
- Unsupervised training



Deep neural networks: Challenges

High memory requirements

- Memory is used to store input data, weight parameters and activations as an input propagates through the network
- Activations from a forward pass must be retained until they can be used to calculate the error gradients in the backwards pass
- Example: 50-layer neural network
  - 26 million weight parameters, 16 million activations in the forward pass
  - 168MB memory (assuming 32-bit float)

Parallelize computations with GPU (graphics processing units)

Deep neural networks: Challenges

Backpropagation does not work well

- Deep networks trained with backpropagation (without unsupervised pretraining) perform worse than shallow networks
- Gradient is progressively getting more dilute
  - Weight correction is minimal after moving back a couple of layers
- High risk of getting "stuck" to local minima
- In practice, a small portion of data is labelled

Perform pretraining to mitigate this issue

	train.	valid.	test
DBN, unsupervised pre-training	0%	1.2%	1.2%
Deep net, auto-associator pre-training	0%	1.4%	1.4%
Deep net, supervised pre-training	0%	1.7%	2.0%
Deep net, no pre-training	.004%	2.1%	2.4%
Shallow net, no pre-training	.004%	1.8%	1.9%

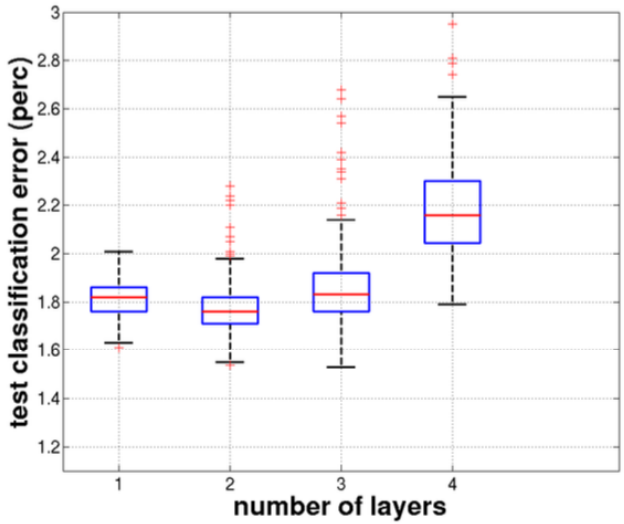
(Bengio et al., NIPS 2007)

Deep neural networks: Unsupervised Pretraining

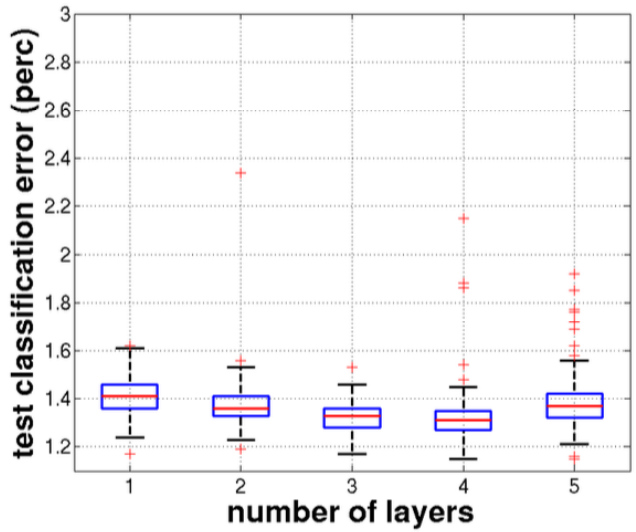
- This idea came into play when research studies found that a DNN trained on a particular task (e.g. object recognition) can be applied on another domain (e.g. object subcategorization) giving state-of-the-art results
- **1st part: Greedy layer-wise unsupervised pre-training**
  - Each layer is pre-trained with an unsupervised learning algorithm
  - Learning a nonlinear transformation that captures the main variations in its input (the output of the previous layer)
- **2nd part: Supervised fine-tuning**
  - The deep architecture is fine-tuned with respect to a supervised training criterion with gradient-based optimization
- We will examine the deep belief networks and stacked autoencoders

Unusual form of regularization: minimizing variance and introducing bias towards configurations of the parameter space that are useful for unsupervised learning

Deep neural networks: Unsupervised Pretraining



Without pre-training



With pre-training

[Source: Erhan et al., 2010]

Deep neural networks: Unsupervised Pretraining

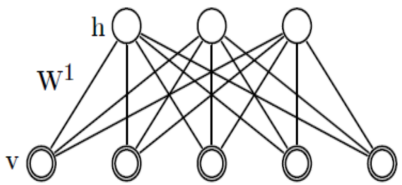
Deep Belief Networks (Hinton et al. 2006)

- Pretraining is implemented by stacking several layers of Restricted Boltzmann Machines (RBM) in a greedy manner
- Assuming joint distribution between hidden h_i and observed variables x_j with parameters $\mathbf{W}, \mathbf{b}, \mathbf{c}$
  $P(\mathbf{x}, \mathbf{h}) \propto \exp(\mathbf{h}^T \mathbf{W} \mathbf{x} + \mathbf{b}^T \mathbf{x} + \mathbf{c}^T \mathbf{h})$
  $P(\mathbf{x}|\mathbf{h}) = \prod_j P(x_j|\mathbf{h}),\ P(x_j = 1|\mathbf{h}) = \text{sigmoid}(b_j + \sum_i W_{ij} h_i)$
  $P(\mathbf{h}|\mathbf{x}) = \prod_i P(h_i|\mathbf{x}),\ P(h_i = 1|\mathbf{x}) = \text{sigmoid}(c_i + \sum_j W_{ij} x_j)$
- RBM trained by approximate stochastic gradient descent
- This representation is extended to all hidden layers
- The RBM parameters correspond to the parameters of the feed-forward multi-layer neural network

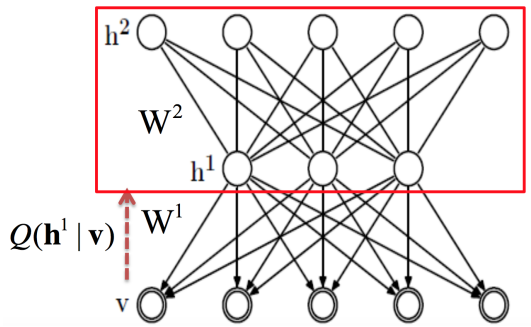
Deep neural networks: Unsupervised Pretraining

Deep Belief Networks (Hinton et al. 2006)

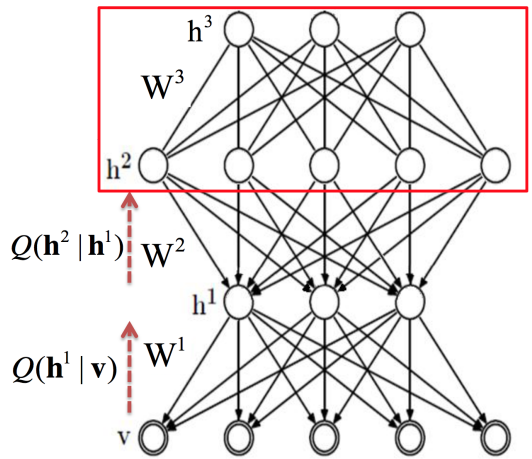
- Step 1: Construct an RBM with an input and hidden layer and train to find $\mathbf{W^{(1)}}$
- Step 2: Stack another hidden layer on top of the RBM to form a new RBM
  - Fix $\mathbf{W^1}$. Assume $\mathbf{h^{(1)}}$ as input. Train to find $\mathbf{W^{(2)}}$.
- Step 3: Continue to stack layers and find weights $\mathbf{W^{(3)}}$, etc.



Step 1



Step 2

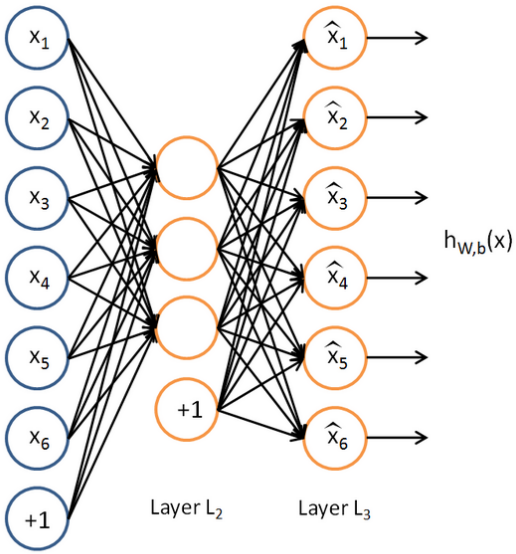


Step 3

Deep neural networks: Unsupervised Pretraining

Autencoder

- Unsupervised algorithm that tries to learn an approximation of the identity function $h_{\mathbf{W},\mathbf{b}}(x) \approx \mathbf{x}$
- Trivial problem unless we place constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data
  e.g. if some of the input features are correlated, then this algorithm will be able to discover some of those correlations

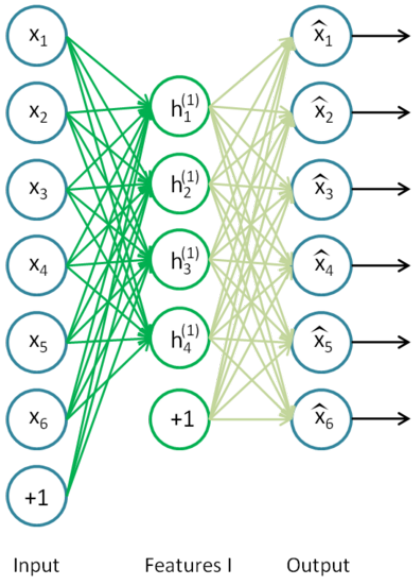


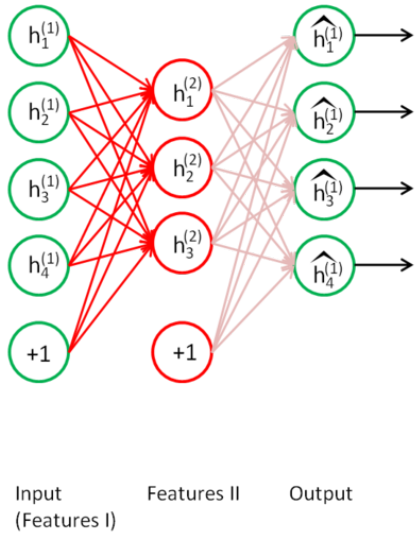
- $\alpha_j^{(i)} = f(W_{i1}^{(1)} x_1 + W_{i2}^{(1)} x_2 + \ldots + b_i^{(1)})$
- Trained using back-propagation and additional sparsity constraints
- Can be also used for feature transformation

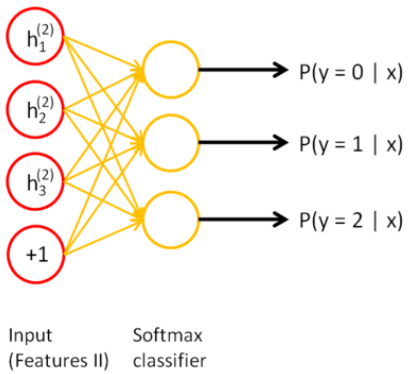
[http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity]

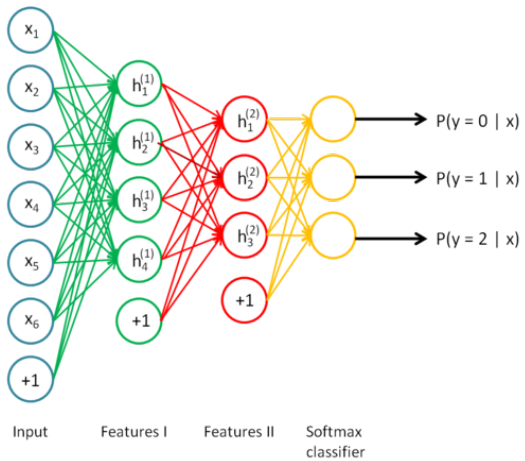
Deep neural networks: Unsupervised Pretraining

Stacked Autencoders for pretraining









Deep neural networks: Unsupervised Pretraining

Stacked Autencoders for pretraining

- Capture a "hierarchical grouping" of the input
- First layer learns a good representation of input features (e.g. edges)
- Second layer learns a good representation of the patterns in the first layer (e.g. corners), etc.

`http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders`

Convolutional neural networks

- Similar to regular neural networks
  - made up of neurons, each with an input and an activation function
  - have weights and biases to be learned
  - have a loss function on the last (fully-connected) layer
- Explicit assumption that the inputs are images
  - explicit assumption that the inputs are images
  - vastly reduce the amount of parameters in the network

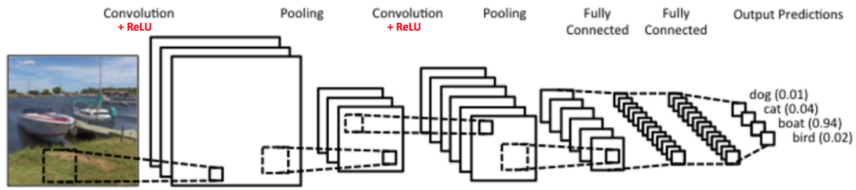
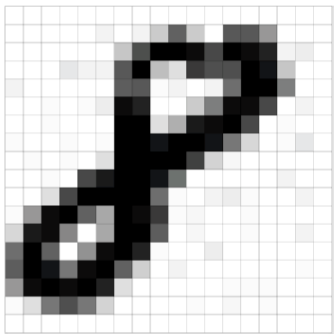
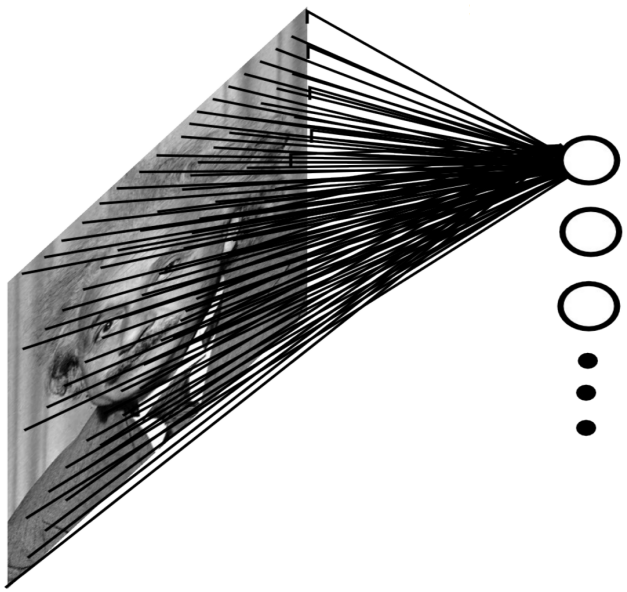


Image representation

- Grayscale image (1-channel)
  - 2d-matrix
  - each pixel ranges from 0 to 255 - 0: black, 255: white
- Color image (3-channel, RGB)
  - three 2d-matrices stacked over each other
  - each with pixel values ranging between 0 and 255



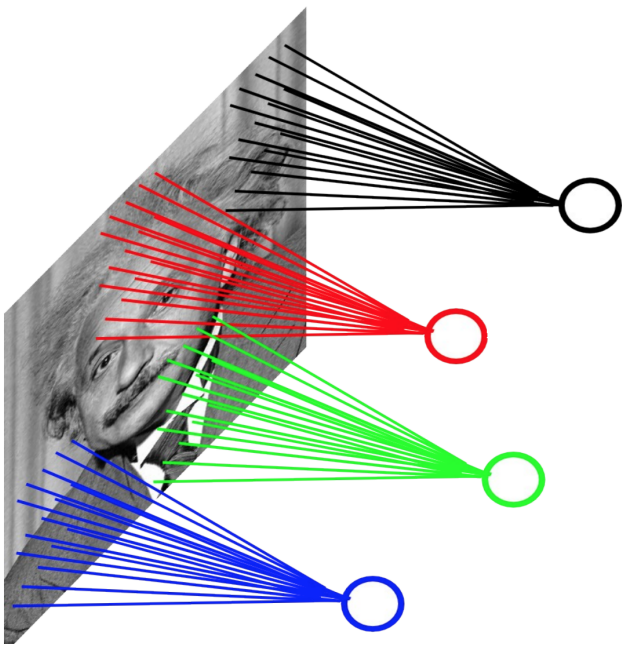
A **fully** connected neural network with image input



- 1000 $\times$ 1000 image, 1M hidden units $\rightarrow$ 10^{12} parameters
- Since spatial correlation is local, we can significantly simplify this

Idea 1: Convolution

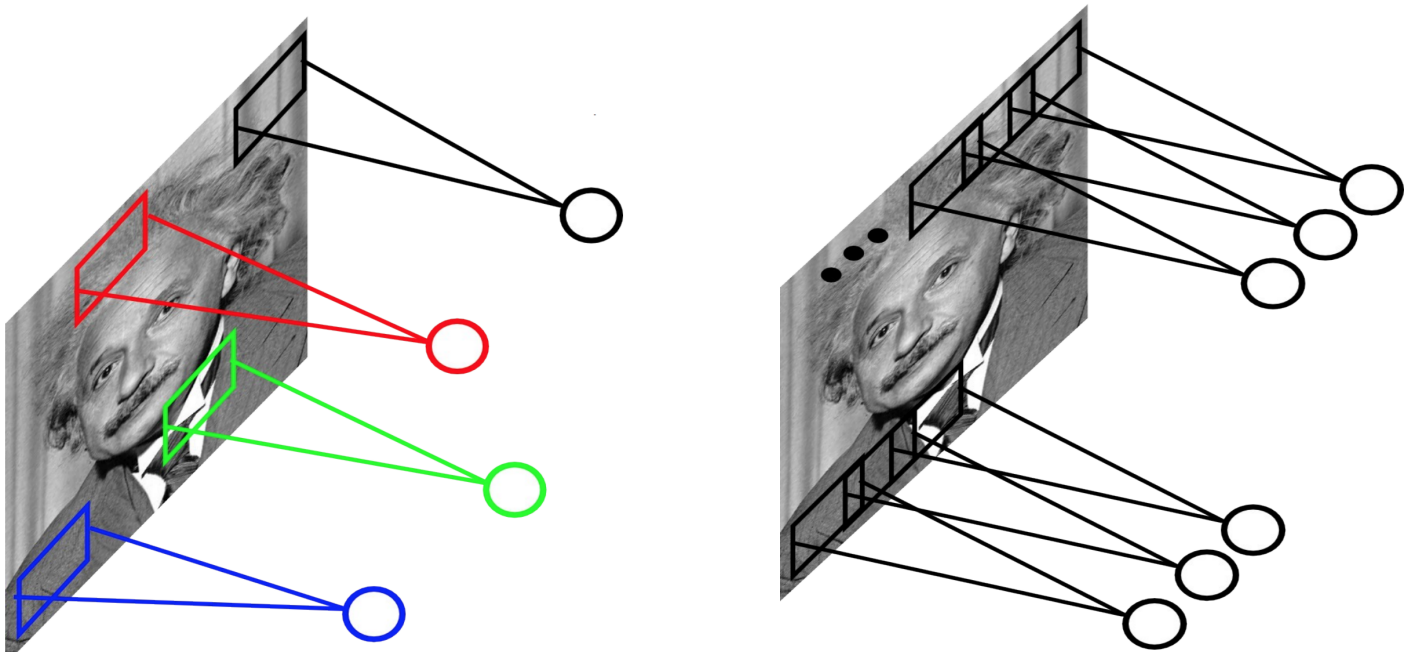
A **locally** connected neural network with image input



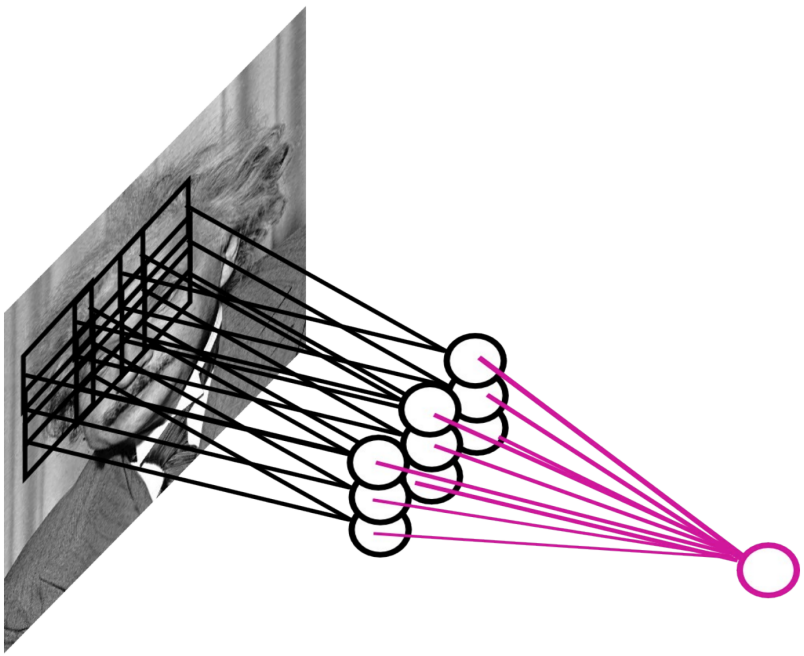
- 1000×1000 image, 1M hidden units, 10×10 filter size $\rightarrow$ 10^8 parameters
- Since spatial correlation is local, we can significantly simplify this

Idea 2: Weight sharing

- Stationarity: Statistics are similar at different locations
- Share the same parameters across different locations



Idea 3: Max-pooling

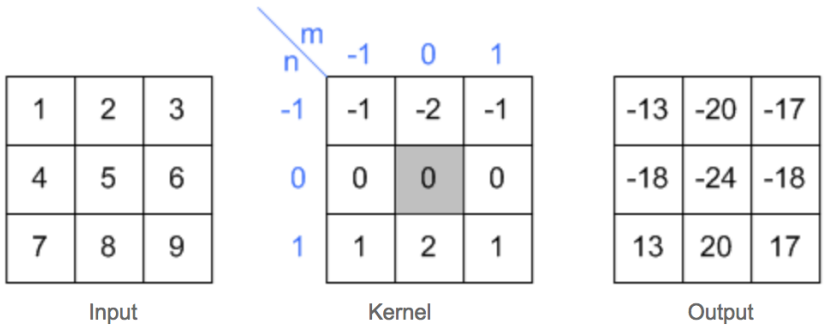


- Let us assume filter is an "eye" detector
- How can we make the detection robust to the exact location of the eye?
- By pooling (e.g., max or average) filter responses at different locations we gain robustness to the exact spatial location of features

Convolution 2d-example

- Convolution is the mathematical operation that implements filtering
- Given an input image $x[m, n]$ and an impulse response $h[m, n]$ (filter or kernel), the convolution output can be written as

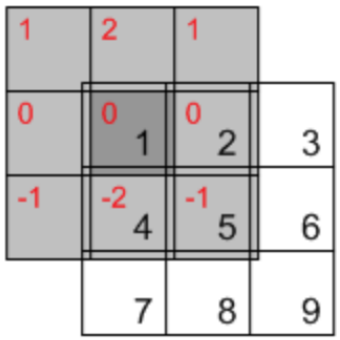
$$y[m, n] = x[m, n] * h[m, n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i, j] h[m - i, n - j]$$



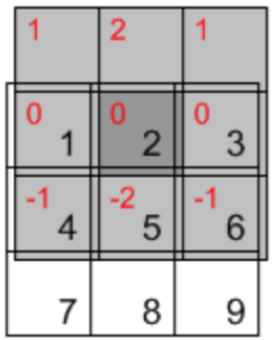
http://www.songho.ca/dsp/convolution/convolution.html#convolution_2d

Convolutional neural networks: The convolution operation

Convolution 2d-example



$$y[0,0] = x[-1,-1]\cdot h[1,1] + x[0,-1]\cdot h[0,1] + x[1,-1]\cdot h[-1,1]$$
$$+ x[-1,0]\cdot h[1,0] + x[0,0]\cdot h[0,0] + x[1,0]\cdot h[-1,0]$$
$$+ x[-1,1]\cdot h[1,-1] + x[0,1]\cdot h[0,-1] + x[1,1]\cdot h[-1,-1]$$
$$= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 0\cdot 0 + 1\cdot 0 + 2\cdot 0 + 0\cdot(-1) + 4\cdot(-2) + 5\cdot(-1) = -13$$

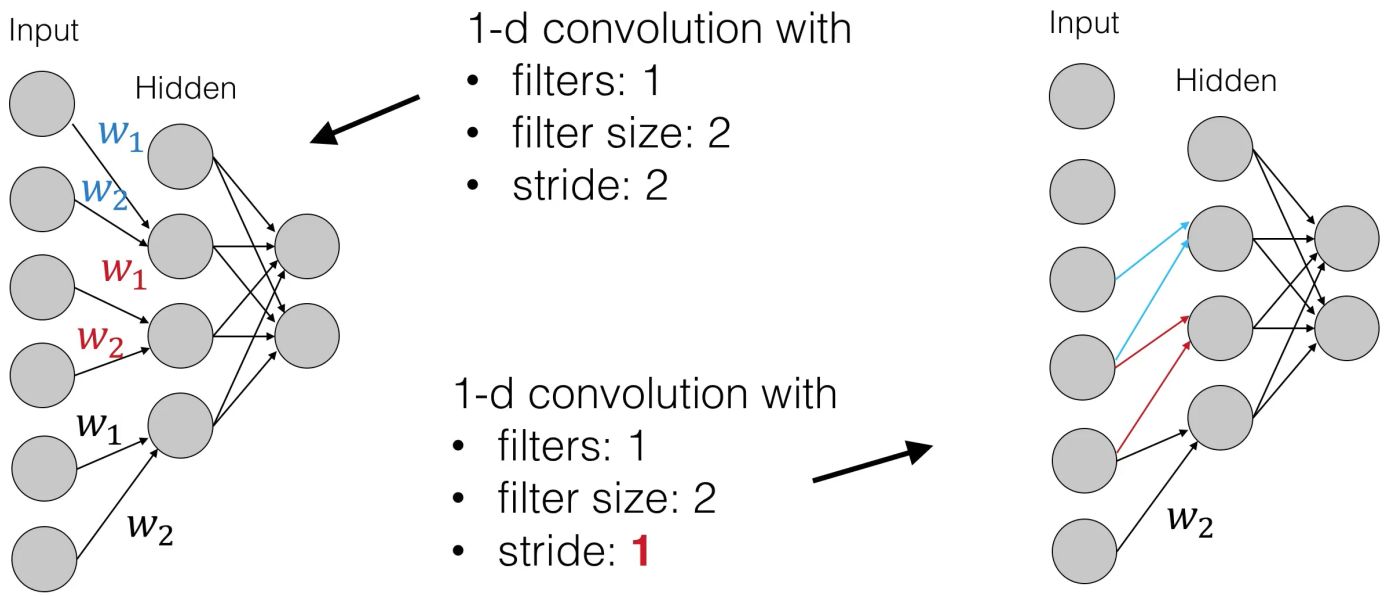


$$y[1,0] = x[0,-1]\cdot h[1,1] + x[1,-1]\cdot h[0,1] + x[2,-1]\cdot h[-1,1]$$
$$+ x[0,0]\cdot h[1,0] + x[1,0]\cdot h[0,0] + x[2,0]\cdot h[-1,0]$$
$$+ x[0,1]\cdot h[1,-1] + x[1,1]\cdot h[0,-1] + x[2,1]\cdot h[-1,-1]$$
$$= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 1\cdot 0 + 2\cdot 0 + 3\cdot 0 + 4\cdot(-1) + 5\cdot(-2) + 6\cdot(-1) = -20$$

http://www.songho.ca/dsp/convolution/convolution.html#convolution_2d

Convolutional neural networks: The convolution operation

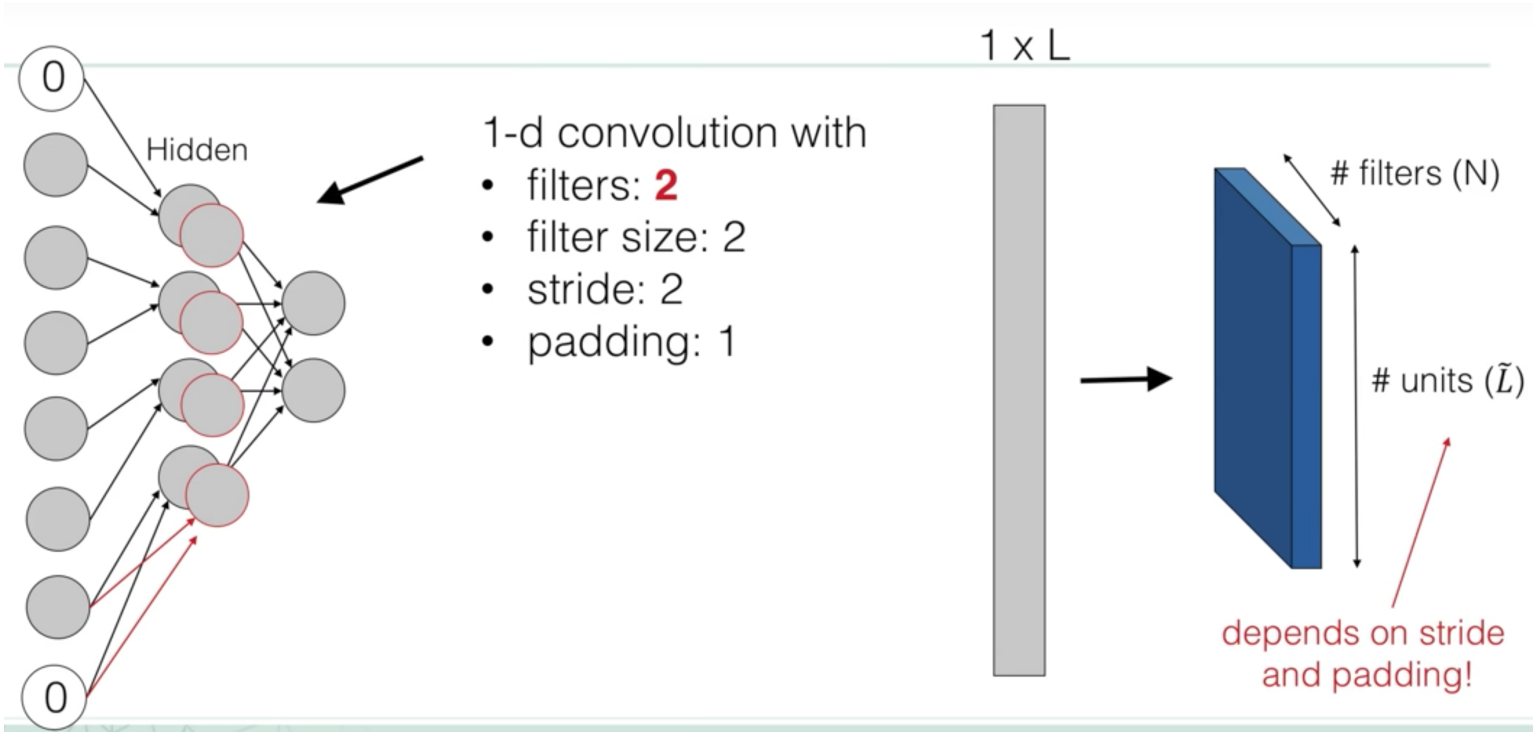
Image 1d-convolution



https://www.nervanasys.com/convolutional-neural-networks/

Convolutional neural networks: The convolution operation

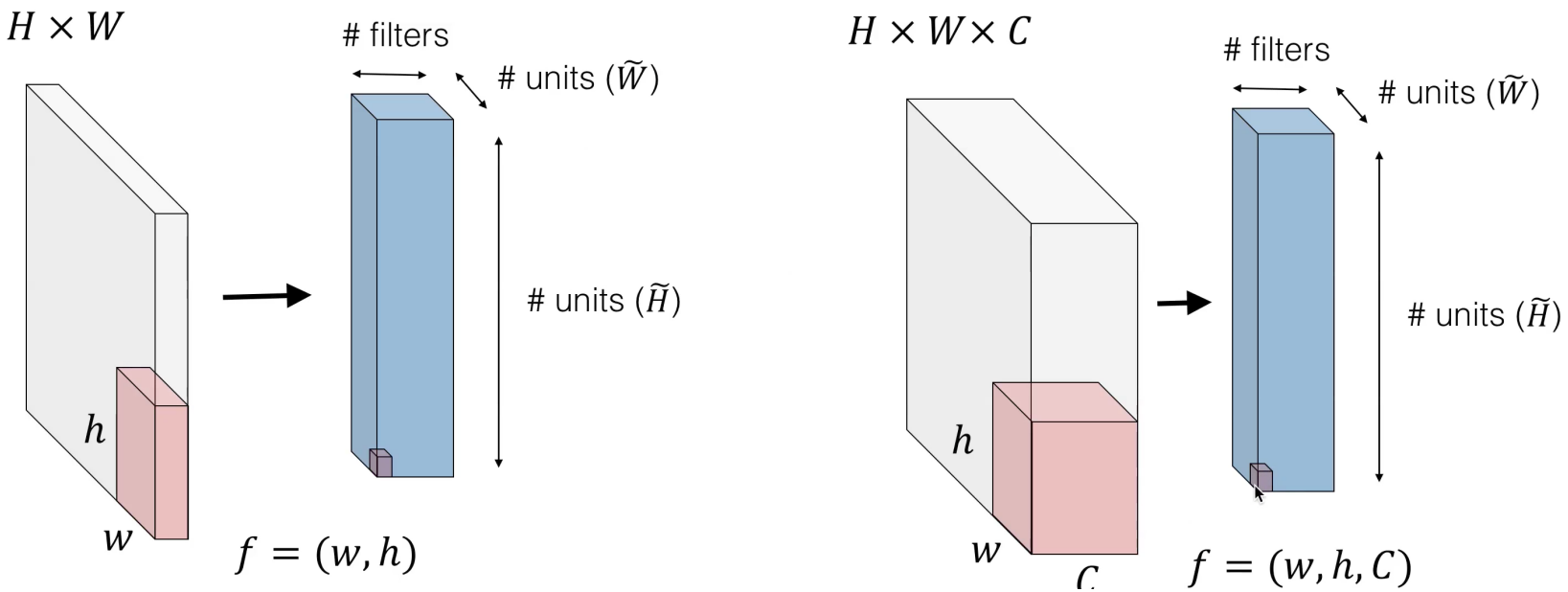
Image 1d-convolution



https://www.nervanasys.com/convolutional-neural-networks/

Convolutional neural networks: The convolution operation

Image 2d-convolution



https://www.nervanasys.com/convolutional-neural-networks/

Also check:

http://cs231n.github.io/assets/conv-demo/index.html

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/ (figure 6)

Convolutional neural networks: The convolution operation

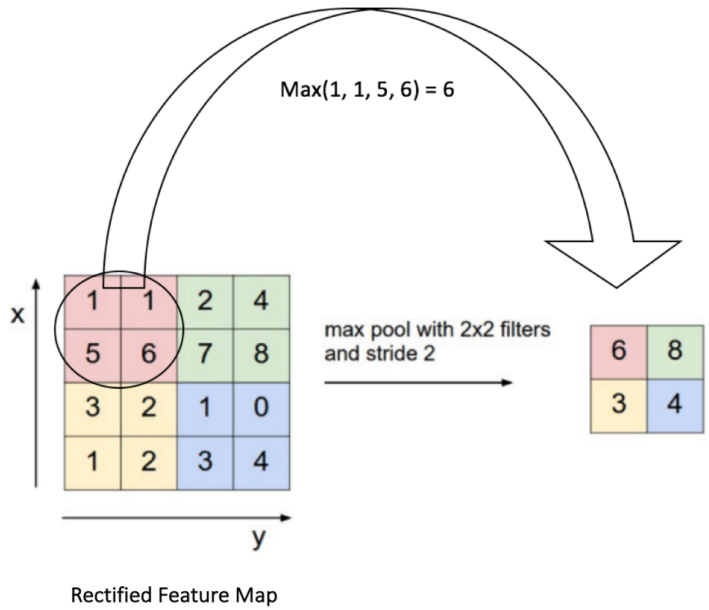
Image 2d-convolution hyperparameters

- Depth: the number of filters we use for the convolution operation
- Stride: the number of pixels by which we slide our filter matrix over the input
- Zero-padding: padding the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix

Convolutional neural networks: Max-pooling

Spatial Pooling (also called subsampling or downsampling)

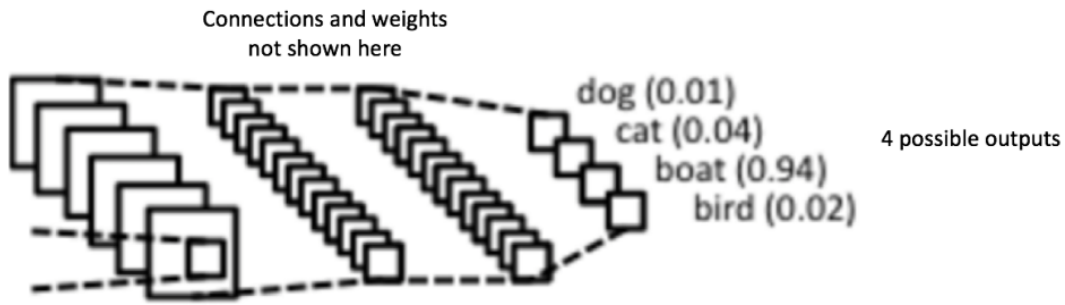
- Reduces the dimensionality of each feature map but retains the most important information
- Can be of different types: Max, Average, Sum etc.
- Makes the input representations (feature dimension) smaller and more manageable
- Promotes an almost scale invariant representation of the image



Convolutional neural networks: Final fully connected layer

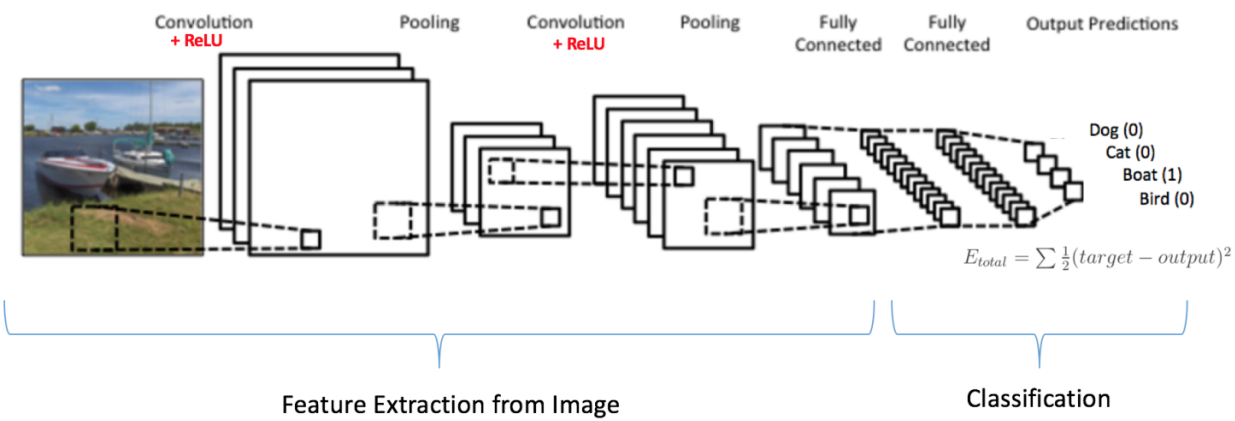
Final fully connected layer

- Traditional multilayer perceptron
- Yields the classification/regression result

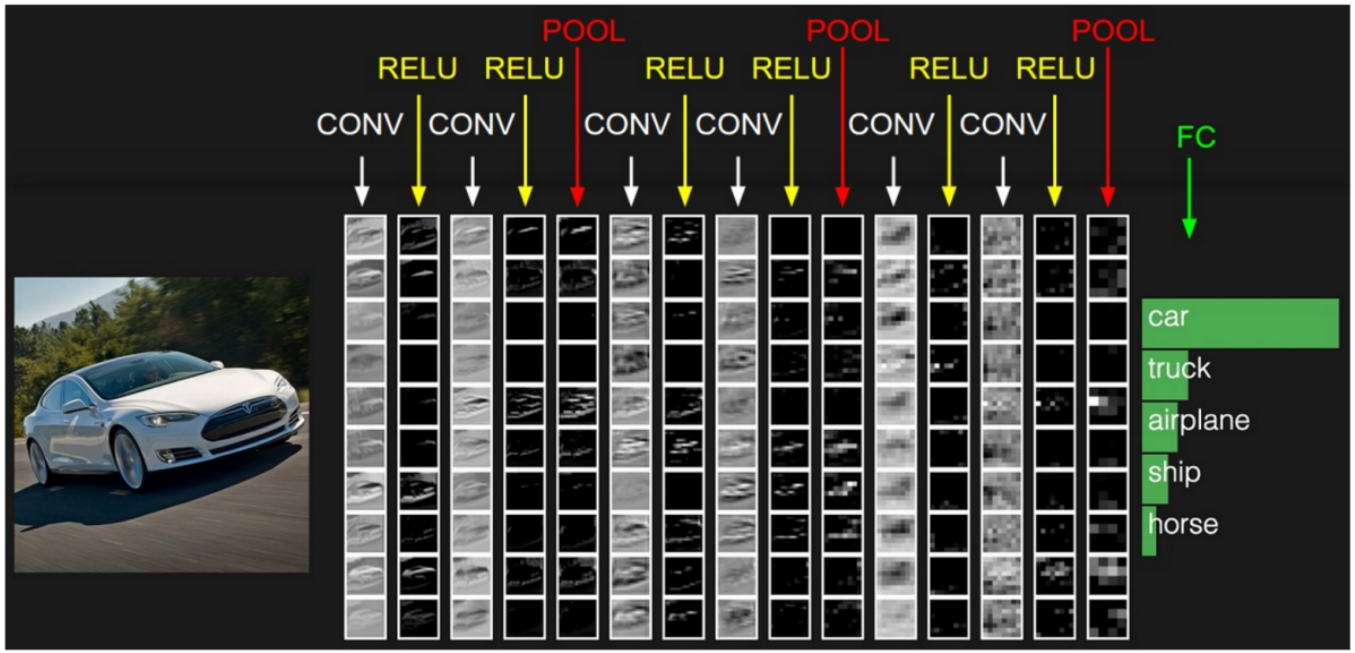


Convolutional neural networks: Putting it all together

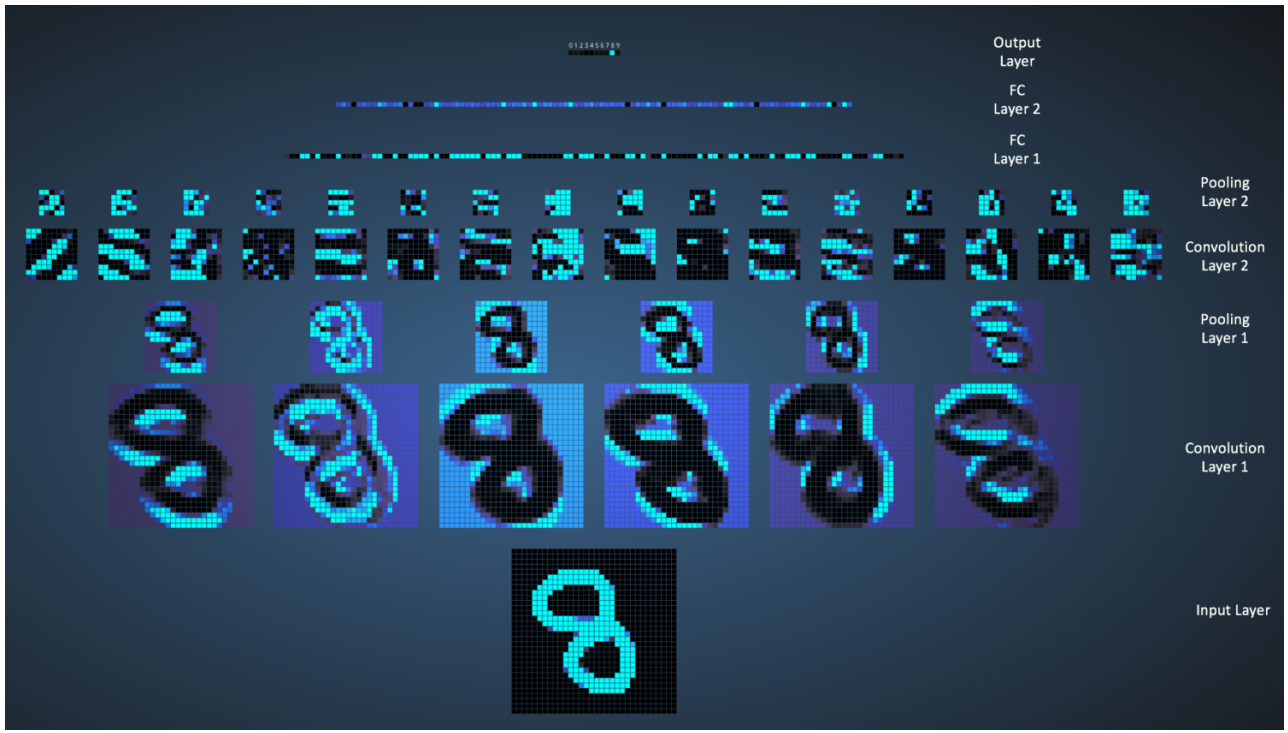
- **Step 1:** Initialize weights
- **Step 2:** Take first image as input and go through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the fully connected layer) and finds the output probabilities for each class
- **Step 3:** Calculate the total error at the output layer
- **Step 4:** Use backpropagation to update the weights, which are adjusted in proportion to their contribution to the total error
- **Step 5:** Repeat Steps 1-4 for all train images



Convolutional neural networks: Examples



Convolutional neural networks: Examples



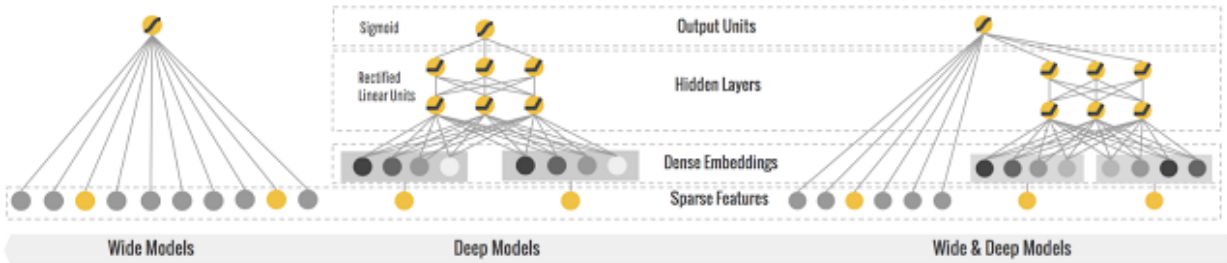
Convolutional neural networks: Hyperparameter tuning

- Learning rate: how much to update the weight during optimization
- Number of epochs: number of times the entire training set pass through the neural network
- Batch size: the number of times the entire training set pass through the neural network
- Activation function: the function that introduces non-linearity to the model (e.g. sigmoid, tanh, ReLU, etc.)
- Number of hidden layers and units
- Weight initialization: Uniform distribution usually works well
- Dropout for regularization: probability of dropping a unit

We can perform grid or randomized search over all parameters

Wide and deep models

- Google: Wide & Deep Learning (2016)
- generic large-scale regression and classification problems with sparse inputs (categorical features with a large number of possible feature values), such as recommender systems, search, and ranking problems
- The wide model takes into account a wide set of cross-product feature transformations to capture how the co-occurrence of a query-item feature pair correlates with the target label
- The deep feed-forward neural network learns lower-dimensional dense representations for every query and item.



Takeaways and Next Time

- Deep Neural Networks
- Next Time: Discussion on Neural Networks