Neural Networks

Chapter 18.6.3, 18.6.4, 18.7
and "Deep Learning" paper

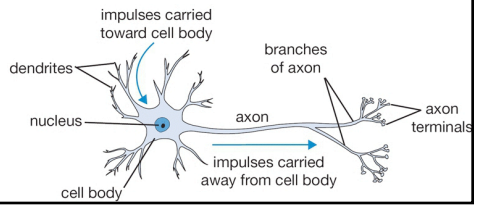
1

Introduction

- Known as
  - **Neural Networks** (NNs)
  - **Artificial Neural Networks** (ANNs)
  - **Connectionist Models**
  - **Parallel Distributed Processing** (PDP) Models
- Neural Networks are a fine-grained, parallel, distributed, computing model

4

- NNs are inspired by the brain:
  - knowledge is acquired experientially (learning)
  - knowledge stored in connections (weights)
- Brain composed of neurons:
  - dendrites collect input from ~10^4 other neurons
  - axon sends output to other neurons
  - connected at synapses that have varying strength
  - this model is greatly simplified



5

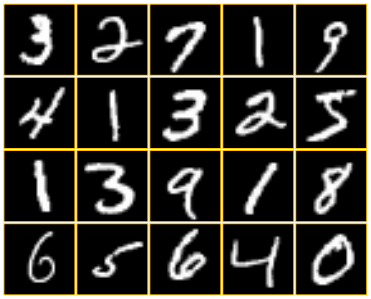
Introduction

Characteristics of human information processing

- number of neurons: ~10^{11}
- number of connections: ~10^4 per neuron
- Processing speed: ~100 Hz

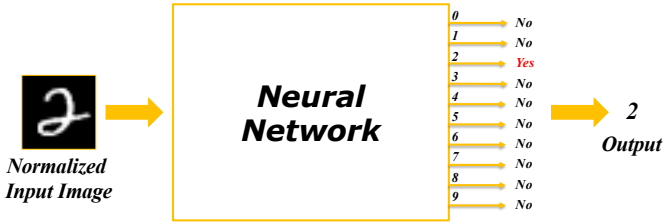
6

Recognizing Digits using a Neural Net



7

Recognizing Digits using a Neural Network as a Classifier



8

Strengths of NN Approach

- Massively parallel
  - from a large collection of simple processing elements emerges complex, global behavior
- Can do complex tasks
  - pattern recognition (handwriting, facial expressions)
  - forecasting (stock prices, power grid demand)
  - adaptive control (autonomous vehicle control, robot control)
- Robust computation
  - can handle noisy and incomplete data due to fine-grained, distributed and continuous knowledge representation

9

Strengths of NN Approach

- Fault tolerant
  - ok to have faulty elements and bad connections
  - isn't dependent on a fixed set of elements and connections
- Degrades gracefully
  - continues to function, possibly at a lower level of performance, when portions of the network are faulty
- Uses inductive learning
  - useful for a wide variety of applications

10

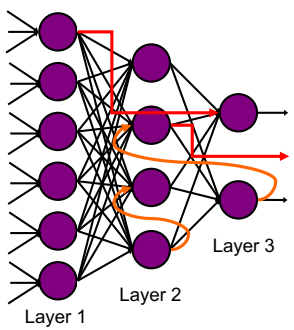
Neural Network Architecture

- large number of **units**
  - simple neuron-like processing elements
- connected by a large number of **links**
  - directed from one unit to another
- a **weight** associated with each link
  - positive or negative real values
  - means of long term storage
  - adjusted by learning
- an **activation** function associated with each unit
  - result of the unit's processing
  - unit's output

11

Neural Network Architecture

- represent as a **graph**
  - **Nodes: units**
  - **Arcs: links**
- single layer
- multi-layer
- Feedback (cycles)?
- layer skipping?
- fully connected?



12

Basics of NN

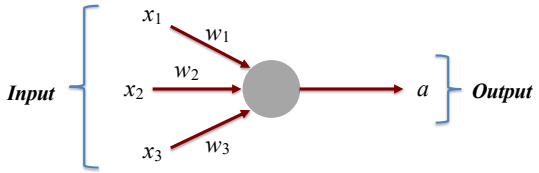
Structure of a "Unit":

- set of **input links**
  - *from* other units or features from the environment
- set of **output links**
  - *to* other units or effectors of the environment
- an **activation function**
  - computes an output scalar value using a simple non-linear function of the linear combination of its inputs

13

Computation by a Unit

Takes several inputs and computes a single scalar output value



14

Basics of NN

Given n inputs, the unit's **activation** (i.e., output) is defined by:

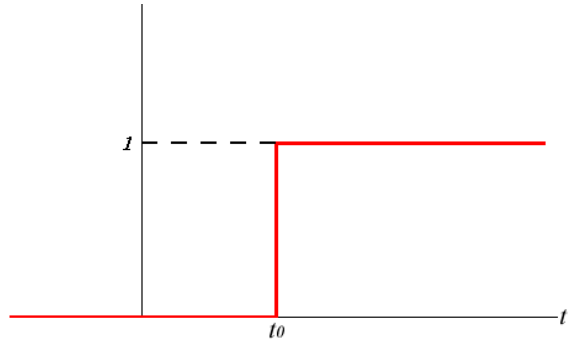
$$a = g(w_1 x_1 + w_2 x_2 + ... + w_n x_n) = g(\sum_{i=1}^{n} w_i x_i) = g(in)$$

where

- w_i are the real-valued weights
- x_i are the scalar input values (in general, real values)
- $g(\,)$ is a simple, non-linear function, commonly:
  - **Step (LTU)**: activation *flips* from 0 to 1 when $in \geq$ threshold
  - **tanh**: $g(in) = \tanh(in)$
  - **Sigmoid / Logistic**: $g(in) = 1 / (1 + \exp(-in))$
  - **Rectified Linear (ReLU)**: $g(in) = \max(0, in)$

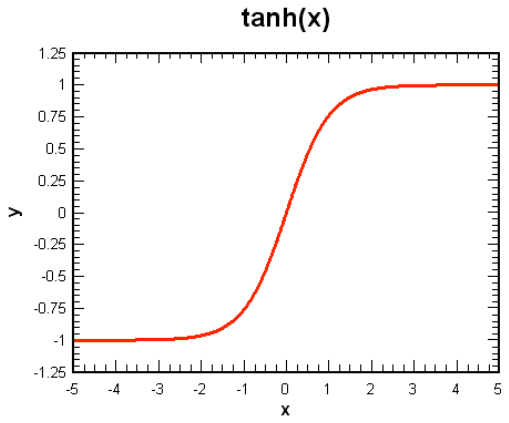
15

Step Function (aka Linear Threshold Unit) (LTU)



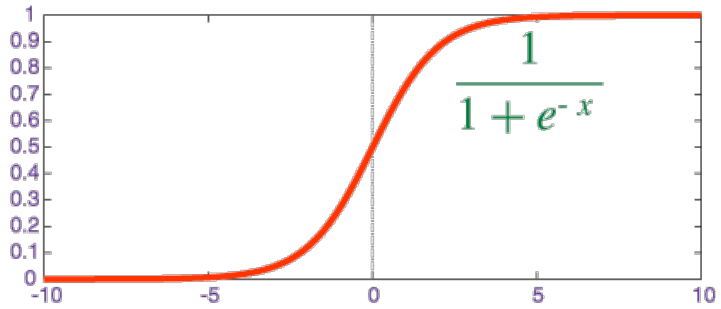
16

Hyperbolic Tangent Function



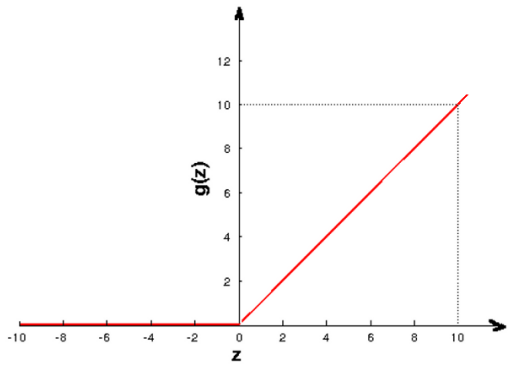
17

Sigmoid Function (aka Logistic function)



18

Rectified Linear Function (ReLU)

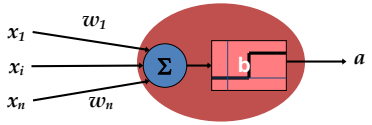


The rectified linear activation function

19

Perceptrons

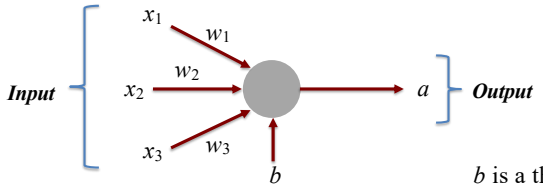
- Studied in the 1950s and 1960s as simple networks for which there was an effective learning algorithm
- "**1-layer network**": one or more *output units*
- "Input units" don't count as a layer because they don't compute anything
- Output units are all **linear threshold units** (**LTU**s)
  - a unit's inputs, x_i, are weighted, w_i, and **linearly combined**
  - **Step** function computes binary output activation value, a



20

Computation by an LTU

Takes several inputs and computes a single scalar output value



b is a threshold

$$a = \begin{cases} 0 & \text{if } \sum_i w_i x_i < b \\ 1 & \text{if } \sum_i w_i x_i \geq b \end{cases}$$

21

Linear Threshold Units (LTU)

Threshold is just another weight (called the **bias**):

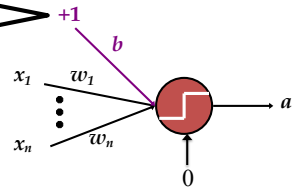
$(w_1\, x_1) + (w_2\, x_2) + \ldots + (w_n\, x_n) \geq b$

which is equivalent to

$(w_1\, x_1) + (w_2\, x_2) + \ldots + (w_n\, x_n) + (b\ (-1)) \geq 0$

Use +1 instead of -1, however

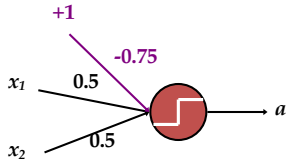
Every unit uses a fixed threshold = 0



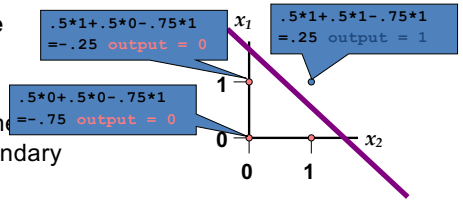
22

Perceptron Examples

- "AND" Perceptron:
  - inputs are 0 or 1
  - output is 1 when both x_1 and x_2 are 1



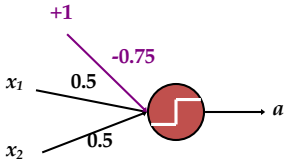
- **2D input space**
  - 4 possible data points
  - weights define decision boundary



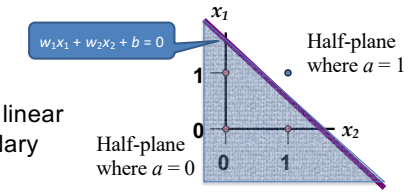
23

Perceptron is a Linear Classifier

- "AND" Perceptron:
  - inputs are 0 or 1
  - output is 1 when both x_1 and x_2 are 1



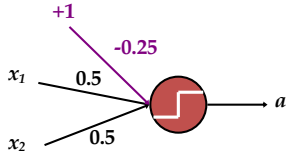
- **2D input space**
  - 4 possible data points
  - weights define linear decision boundary



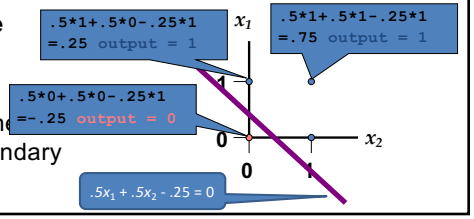
24

Perceptron Examples

- "OR" Perceptron:
  - inputs are 0 or 1
  - output is 1 when either x_1 or x_2 are 1



- **2D input space**
  - 4 possible data points
  - weights define decision boundary



25

Perceptron Learning

How are the weights learned by a Perceptron?

- Programmer specifies:
  - numbers of units in each layer
  - connectivity between units
- Only unknown is the set of weights
- Learning of weights is *supervised*
  - for each training example
    - a list of values for the input units of the network
  - the correct output is given
    - a list of desired output values for all output units

26

Perceptron Learning Algorithm

1. Initialize the weights in the network
   (usually with random values)
2. Repeat until all examples correctly classified or some other stopping criterion is met

   foreach example, ***e***, in the training set **do**

   O = neural_net_output(network, e);

   T = desired output; // **T**arget or **T**eacher output

   update_weights(***e***, O, T);

- Each pass through *all* of the training examples is called an **epoch**
- Step 2 takes *many epochs* in general!

27

Perceptron Learning Rule

How should the weights be updated?

- Determining how to update the weights is an instance of the **credit assignment problem**

Perceptron Learning Rule:

$$w_i = w_i + \Delta w_i$$

where $\Delta w_i = \alpha\, x_i\, (T - O)$

x_i is the input associated with i^{th} input unit

α is a real-valued constant between 0.0 and 1.0 called the **learning rate**

28

Perceptron Learning Rule Properties

- $\Delta w_i = \alpha\, x_i\, (T - O)$ doesn't depend on w_i
- No change in weight (i.e., $\Delta w_i = 0$) if:
  - **correct output**, i.e., $T = O$ gives $\alpha \times x_i \times 0 = 0$
  - **0 input**, i.e., $x_i = 0$ gives $\alpha \times 0 \times (T - O) = 0$
- If T=1 and O=0, *increase* the weight

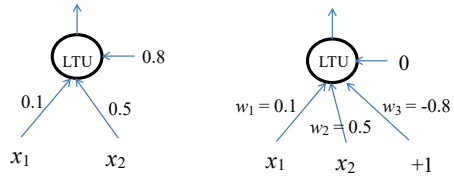
  so that maybe next time the result will exceed the output unit's threshold, causing it to be 1
- If T=0 and O=1, *decrease* the weight

  so that maybe next time the result won't exceed the output unit's threshold, causing it to be 0
- So, when error occurs, update the weight so that it changes the activation *towards* the desired output/activation value

29

Example: Learning OR

- $\Delta w_i = \alpha(T - O)x_i = 0.2(T - O)x_i$
- Initial network:



30

Example: Learning OR

bias

x1	x2	T	O	Δw1	w1	Δw2	w2	Δw3	w3
					0.1		0.5		-0.8
0	0	0	0	0	0.1	0	0.5	0	-0.8
0	1	1	0	0	0.1	0.2	0.7	0.2	-0.6
1	0	1	0	0.2	0.3	0	0.7	0.2	-0.4
1	1	1	1	0	0.3	0	0.7	0	-0.4
0	0	0	0	0	0.3	0	0.7	0	-0.4
0	1	1	1	0	0.3	0	0.7	0	-0.4
1	0	1	0	0.2	0.5	0	0.7	0.2	-0.2
1	1	1	1	0	0.5	0	0.7	0	-0.2
0	0	0	0	0	0.5	0	0.7	0	-0.2
0	1	1	1	0	0.5	0	0.7	0	-0.2
1	0	1	1	0	0.5	0	0.7	0	-0.2
1	1	1	1	0	0.5	0	0.7	0	-0.2

31

Perceptron Learning Rule (PLR)

Perceptron Convergence Theorem:

- ***If a set of examples is learnable, then PLR will find an appropriate set of weights***
  - in a finite number of steps
  - independent of the initial weights
  - Using a sufficiently small value for α
- This theorem says that if a solution exists, PLR's gradient descent is guaranteed to find an optimal solution (i.e., 100% correct classification) for any 1-layer neural network

34

Perceptron Learning Rule (PLR)

- PLR is a "local" learning rule in that only local information in the network is needed to update a weight
- **PLR performs gradient descent (hill-climbing) in "weight space"**
- Iteratively adjusts all weights so that for each training example the error decreases (more correctly, error is monotonically non-increasing)

33

Limits of Perceptron Learning

What Can be Learned by a Perceptron?

- Perceptron's output is determined by the separating hyperplane (linear decision boundary) defined by

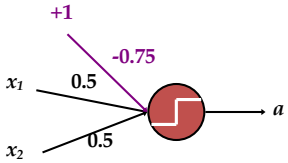
$$(w_1\ x_1) + (w_2\ x_2) + \ldots + (w_n\ x_n)\ =\ b$$

- So, Perceptrons can *only* learn functions that are **linearly separable** (in **input space**)

35

Separating Hyperplane Example

- "AND" Perceptron:
  - inputs are 0 or 1
  - output is 1 when both x_1 and x_2 are 1

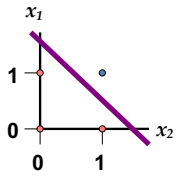


- **2D "input space"**

  decision line equation:

  $0.5x_1 + 0.5x_2 - 0.75 = 0$

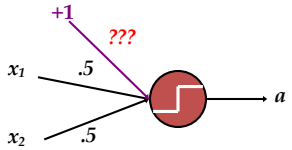
  or, in slope-intercept form: **$x_1 = -x_2 + 1.5$**



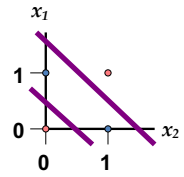
36

Limits of Perceptron Learning

- "XOR" Perceptron?
  - inputs are 0 or 1
  - output is 1 when x_1 is 1 and x_2 is 0 or x_1 is 0 and x_2 is 1



- **2D input space with 4 possible data points**



Can't separate + from – using a single straight line, so XOR is not linearly-separable and can't be learned by a Perceptron

37

Perceptron Learning Summary

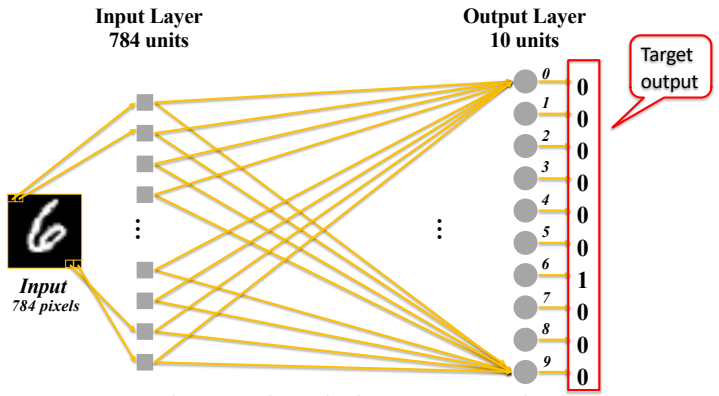
In general, the goal of learning in a 1-output-unit Perceptron is to adjust the **separating hyperplane** (i.e., linear decision boundary) that divides an *n*-dimensional "input space," where *n* is the number of input units, by modifying the weights (including the bias) until all the examples with target value 1 are on one side of the hyperplane, and all the examples with target value 0 are on the other side of the hyperplane

39

Recognizing Digits using a Neural Network as a Classifier

40

Digit Recognition using a Perceptron



Note: Figure does not show the bias input to each output unit

Classification accuracy ≈ 90% (random guessing = 10%)

41

Perceptron Summary

- 1-layer network
- 1 or more units in a single "output" layer, with each unit using an LTU activation function
- "Input" layer defines inputs from 1 example
- Complete connectivity: All inputs are connected to each output unit
- Uses the Perceptron Learning Rule to update all weights
- Can only learn linearly-separable functions

47

Perceptron Learning Summary

In general, the goal of learning in a 1-output-unit Perceptron is to adjust the **separating hyperplane** (i.e., linear decision boundary) that divides an *n*-dimensional "input space," where *n* is the number of input units, by modifying the weights (including the bias) until all the examples with target value 1 are on one side of the hyperplane, and all the examples with target value 0 are on the other side of the hyperplane

48

Beyond Perceptrons

- Perceptrons are weak computing models because they **can only learn linearly-separable functions**
- General NN's can have multiple "layers" of units, which enhance their computational power; the challenge is to find a learning rule that works for multi-layered networks

49

Beyond Perceptrons

- A ***Multi-Layer, Feed-Forward Network*** computes a function of the inputs and the weights
- Input "units"
  - Input values are given; no activation function
- Output units
  - activation value is the output result
- **Hidden units** (between input and output units)
  - cannot observe directly
- Perceptrons have input units followed by one layer of output units, i.e., no hidden units

50

Beyond Perceptrons

- NN's with one hidden layer of a sufficient number of units, can compute functions associated with convex classification regions in input space
  - And can *approximate* any continuous function
- NN's with two hidden layers are universal computing devices, although the complexity of the function is limited by the number of units
  - **If too few**, the network will be unable to represent the function
  - **If too many**, the network can memorize examples and is subject to "overfitting"

51

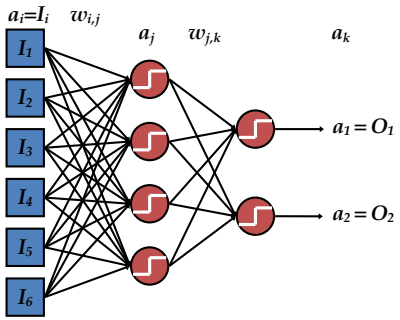
Two-Layer, Feed-Forward Neural Network

Input Units
Hidden Units
Output Units

Weights on links from input to hidden

Weights on links from hidden to output

Network Activations



52

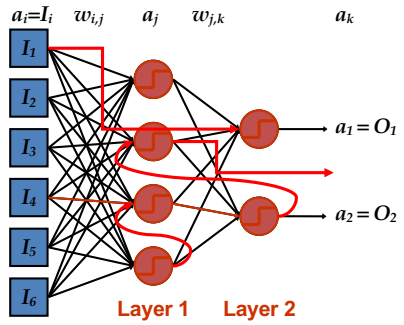
Two-Layer, Feed-Forward Neural Network

Two Layers: count layers with units computing an activation

Feed-Forward: each unit in a layer connects to ***all*** units in the next layer

no cycles
- links within the same layer
- links to prior layers

no skipping layers



53

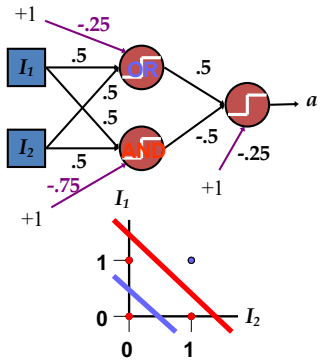
XOR Example

XOR 2-Layer Feed-Forward Network

- inputs are 0 or 1
- output is 1 when
  I_1 is 1 and I_2 is 0, or
  I_1 is 0 and I_2 is 1

Each unit in hidden layer acts like a Perceptron learning a decision line

- top hidden unit acts like an OR Perceptron
- bottom hidden unit acts like an AND Perceptron



54

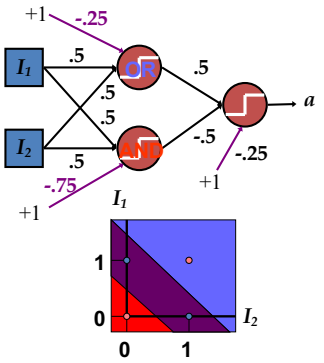
XOR Example

XOR 2-Layer Feed-Forward Network

- inputs are 0 or 1
- output is 1 when
  I_1 is 1 and I_2 is 0 or
  I_1 is 0 and I_2 is 1

To classify an example each unit in the output layer combines these decision lines by intersecting their "half-planes":

when OR is 1 and AND is 0
then output, a, is 1



55

Learning in Multi-Layer, Feed-Forward Neural Nets

- PLR doesn't work in multi-layered feed-forward nets because the desired target values for the hidden units are *not known*
- Must again solve the **Credit Assignment Problem**
  - determine which weights to credit/blame for the output error in the network, and how to update them

56

Learning in Multi-Layer, Feed-Forward Neural Nets

Back-Propogation

- Method for learning weights in these networks
- Generalizes PLR
- Rumelhart, Hinton and Williams, 1986

Approach

- **Gradient-descent algorithm** to minimize the total error on the training set
- Errors are propagated through the network starting at the output units and working *backwards* towards the input units

57

Back-Propagation Algorithm

Initialize the weights in the network (usually random values)
Repeat until stopping criterion is met **{**
  forall p,q in network, $\Delta \boldsymbol{W}_{p,q} = 0$
  foreach example ***e*** in training set **do {**
    O = neural_net_output(***network***, ***e***) // forward pass
    Calculate error (***T*** - ***O***) at the output units // ***T*** = teacher output
    Compute $\Delta w_{j,k}$ for all weights from hidden unit j to output unit k
    Compute $\Delta w_{i,j}$ for all weights from input unit i to hidden unit j
    forall p,q in network $\Delta \boldsymbol{W}_{p,q} = \Delta \boldsymbol{W}_{p,q} + \Delta w_{\boldsymbol{p,q}}$
    }
  for all p,q in network $\Delta \boldsymbol{W}_{p,q} = \Delta \boldsymbol{W}_{p,q}$ / num_training_examples
  network = update_weights(***network***, $\Delta \boldsymbol{W}_{p,q}$)
}

backward pass

Note: Uses average gradient for *all* training examples when updating weights

58

Back-Prop using Stochastic Gradient Descent (SGD)

- Most practitioners use SGD to update weights using the *average gradient computed using a small batch of examples*, and repeating this process for many small batches from the training set
- In extreme case, update after *each* example
- Called *stochastic* because each small set of examples gives a noisy estimate of the average gradient over *all* training examples

60

Updating the Weights

- Back-Propagation performs a ***gradient descent search*** in "weight space" to learn the network weights
- Given a network with n weights:
  - each configuration of weights is a vector, $\boldsymbol{W}$, of length n that defines an instance of the network
  - $\boldsymbol{W}$ can be considered a point in an n-dimensional **weight space**, where each dimension is associated with one of the connections in the network

61

Updating the Weights

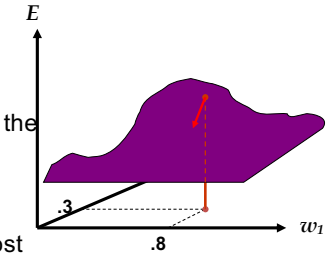
- Given a training set of m examples:
  - Each network defined by the vector $\boldsymbol{W}$ has an associated total error, $\boldsymbol{E}$, on ***all*** the training data
  - $\boldsymbol{E}$ is the sum squared error (SSE) defined by
    $$E = E_1 + E_2 + \ldots + E_m$$
    where E_i is the squared error of the network on the i^{th} training example
- Given t output units in the network:
  $$E_i = (T_{i1} - O_{i1})^2 + (T_{i2} - O_{i2})^2 + \ldots + (T_{it} - O_{it})^2$$
  - T_{ij} is the target value for the i^{th} example's j^{th} output unit
  - O_{ij} is the network output value for the i^{th} example's j^{th} output unit

62

Updating the Weights

Visualized as an error surface in "weight space"

- For example, when a network has 2 weights, each point in $w_1 w_2$ plane is a weight configuration, ***W***
- Each point has a total error E
- 2D surface represents errors for all weight configurations
- Goal is to find a lower point on the error surface (local minimum)
- Gradient descent follows the direction of steepest descent, i.e., where E decreases the most



63

Updating the Weights

- The **gradient** is defined as

$$\nabla E = [\partial E / \partial w_1, \partial E / \partial w_2, \ldots, \partial E / \partial w_n]$$

- Update the ith weight using

$$\Delta w_i = -\alpha \, \partial E / \partial w_i$$

- Can't use the Step function (LTU) because it's derivative is 0 almost everywhere and undefined at the step
- Instead, let's use (for now) the **Sigmoid function**

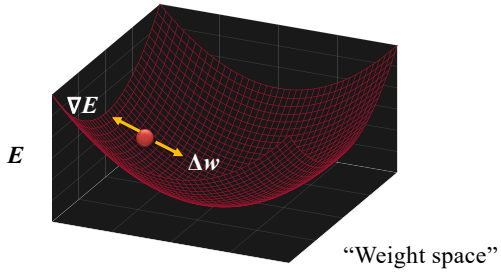
65

Back-Propagation Algorithm

1. Make prediction (Forward Pass)
2. Calculate the total Error (Loss), E
3. Calculate gradient of the loss function w.r.t. each weight, $\partial E/\partial w_i$
4. Update all weights by taking a step in the opposite direction: $\Delta w_i = -\alpha \, \partial E / \partial w_i$
5. Iterate

66

Gradient Descent

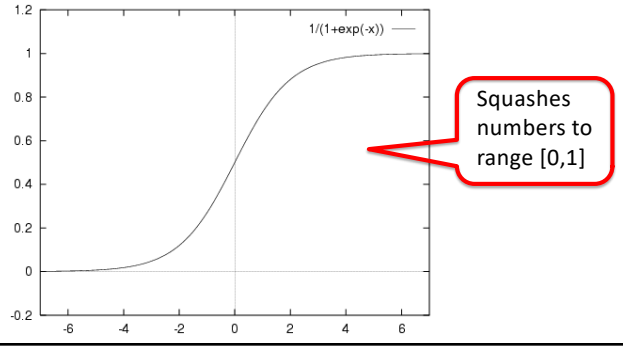


67

Sigmoid Activation Function

Use a smooth, "soft threshold" function such as **Sigmoid function** (aka **Logistic Sigmoid function**):

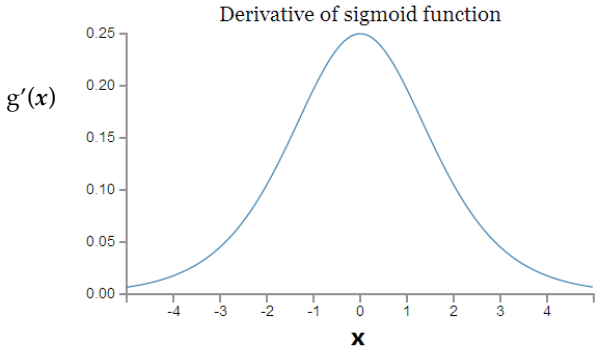
$g_w(x) = 1 \,/\, (1 + e^{-wx})$



68

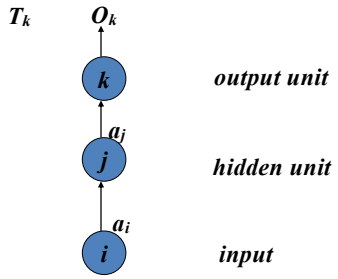
First Derivative of Sigmoid Function

$g'(x) = g(x)\,(1 - g(x))$



69

Updating the Weights



70

Perceptron Learning Rule

$w_i = w_i + \Delta w_i$

where $\Delta w_i = \alpha\, x_i\, (T - O)$

x_i is the input associated with i^{th} input "unit"

$T - O$ is the error at the output unit

α is a real-valued constant between 0.0 and 1.0 called the **learning rate**

71

Updating Weights in a 2-Layer Neural Network

For **weights between hidden and output units**, generalized PLR for ***Sigmoid activation function*** is

$$\Delta w_{j,k} = -\alpha\, \partial E / \partial w_{j,k}$$
$$= -\alpha\, -a_j\, (T_k - O_k)\, g'(in_k)$$
$$= \alpha\, a_j\, (T_k - O_k)\, O_k\, (1 - O_k)$$
$$= \alpha\, a_j\, \Delta_k$$

$\Delta_k = Err_k \times g'(in_k)$

- $w_{j,k}$ weight on link **from hidden unit *j* to output unit *k***
- α learning rate parameter
- a_j activation (i.e., output) of hidden unit j
- T_k teacher output for output unit k
- O_k actual output of output unit k
- g' derivative of the sigmoid activation function, which is $g' = g(1 - g)$
- Δ_k modified error

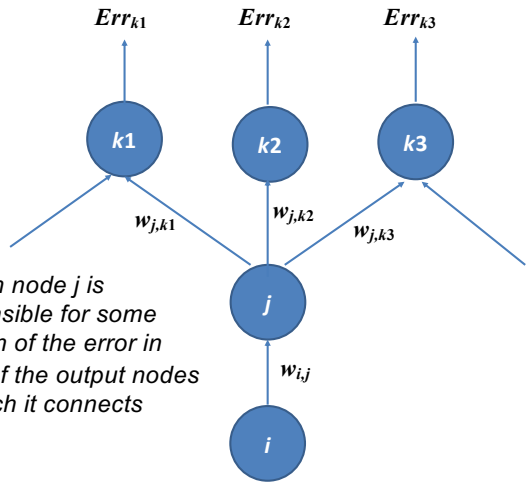
72

Updating Weights in a 2-Layer Neural Network

For **weights between input and hidden units**:

- We don't have teacher-supplied correct activation values for the hidden units
- Must infer the error at these units by "back-propagating"
- Error at an output unit is "distributed" back to each of the hidden units in proportion to the weight of the connection between them
- Each hidden unit accumulates some error from *each* of the output units to which it is connected

73



Hidden node j is responsible for some fraction of the error in each of the output nodes to which it connects

74

Updating Weights in a 2-Layer Neural Network

For **weights between inputs and hidden units**:

$$\Delta w_{i,j} = -\alpha\, \partial E / \partial w_{i,j}$$
$$= -\alpha(-a_i) g'(in_j) \sum_k w_{j,k} (T_k - O_k) g'(in_k)$$
$$= \alpha\; a_i\; a_j (1 - a_j) \sum_k w_{j,k} (T_k - O_k)\; O_k (1 - O_k)$$
$$= \alpha\; a_i\; \Delta_j \quad \text{where} \quad \Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$$

$\Delta_k = Err_k \times g'(in_k)$

- $w_{i,j}$ weight on link **from input *i* to hidden unit *j***
- $w_{j,k}$ weight on link from hidden unit j to output unit k
- α learning rate parameter
- a_j activation (i.e., output) of hidden unit j
- T_k teacher output for output unit k
- O_k actual output of output unit k
- a_i input value i
- g' derivative of sigmoid activation function, which is $g' = g(1\text{-}g)$

75

Back-Propagation Algorithm

Initialize the weights in the network (usually random values)
Repeat until stopping criterion is met
  foreach example, *e,* in training set **do**
    { O = neural_net_output(network, *e*) ← forward pass
    T = desired output, i.e., Target or Teacher's output
    calculate error (T_k - O_k) at each output unit *k*
    foreach hidden unit *j* and output unit *k* compute
      $\Delta w_{j,k} = \alpha\ a_j\ \Delta_k\ =\ \alpha\ a_j\ (T_k - O_k)\ g'(in_k)$
    foreach input unit *i* and hidden unit *j* compute
      $\Delta w_{i,j} = \alpha\ a_i\ \Delta_j = \alpha\ a_i\ g'(in_j) \sum_k w_{j,k} (T_k - O_k) g'(in_k)$
    forall *p*, *q* in network $w_{p,q} = w_{p,q} + \Delta w_{p,q}$ ← Simplistic SGD: update all weights after *each* example
  }

backward pass

76

Notes for HW #4, Problem 3

- At each **hidden unit**, use the **ReLU activation function**
- In forward pass use the ReLU activation function: $g(x) = \max(0, x)$
- In backward pass use the derivative of the ReLU function:

$$g'(x) = \begin{cases} 0, & \text{if } x \le 0 \\ 1, & \text{otherwise} \end{cases}$$

77

Multi-Class Classification with Neural Networks

- Use a number of output units equal to the number of classes
- Represent each class with 1 at a particular output unit and 0 at all other output units

$$\begin{bmatrix}1\\0\\0\end{bmatrix} \quad \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \begin{bmatrix}0\\0\\1\end{bmatrix}$$

Cat *Dog* *Toaster*

Teacher output

78

Notes for HW #4, Problem 3

- At all **output units**, use the **Softmax activation function** and define the Error (Loss) function using **Cross-Entropy**

79

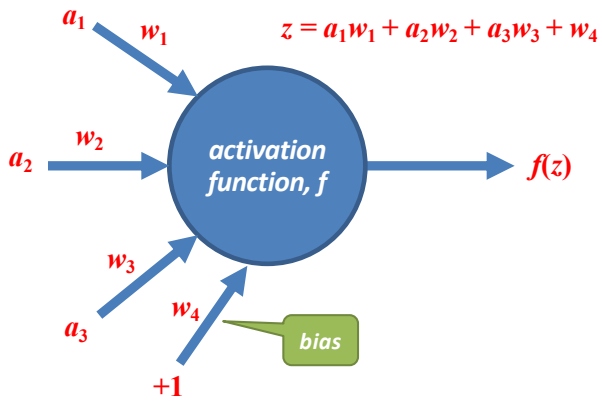
Multi-Class Classification with Neural Networks

- At each **output unit** use the **Softmax activation function**:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

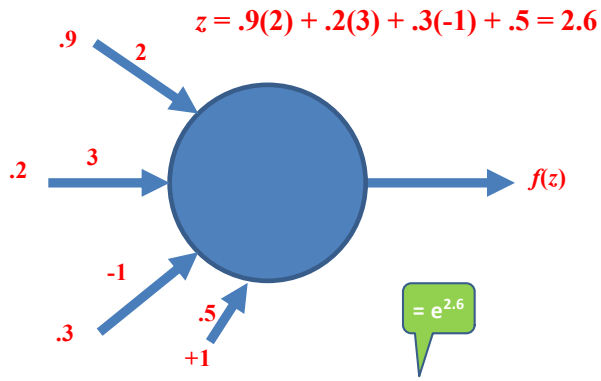
  where z_i is the weighted sum of the inputs to the i^{th} output unit, and there are K output units
- Means output units all have values between 0 and 1, and sum to 1; can be interpreted as probabilities
- Note: [3,1,1] does *not* become [.6, .2, .2] but rather [.78, .11, .11] since we are exponentiating before normalizing

80



81

Example Output Unit Computation



$f(2.6) = \text{Softmax}(2.6) = 13.46 / \sum e^{zk}$

82

Multi-class Classification with Neural Networks

- For the error function, instead of SSE, use "Cross-Entropy" loss, which for a *single* training example is:

$$C.E. = -\sum_{i=1}^{K} T_i \log(O_i)$$

Measures distance between target distribution and output distribution

  where O_i is the computed output at output unit i and T_i is the target output at unit I, and K is the number of output units
- Derivative has a nice property when used with Softmax:

$$\frac{\partial C.E.}{\partial softmax} \cdot \frac{\partial softmax}{\partial z_i} = O_i - T_i$$

83

- Also must compute the **total Cross-Entropy loss** over the *entire* training set at end of each epoch, which is defined as the average Cross-Entropy loss:

$$Total\ Loss = \frac{1}{N}\sum_{i=1}^{N} CE(example_i)$$

- Note: The entire training set is "shuffled" at the end of each epoch to aid convergence

85

Updating Weights in a 2-Layer Neural Network

For **weights between input and hidden units with ReLU at hidden units**:

$$\Delta w_{i,j} = -\alpha\, \partial E / \partial w_{i,j}$$
$$= \alpha\, a_i\, g'(in_j) \sum_{k=1}^{K} w_{j,k} \Delta_k$$

$w_{i,j}$ weight on link **from input i to hidden unit j**
$w_{j,k}$ weight on link from hidden unit j to output unit k
K number of output units
α learning rate parameter
T_k teacher output for output unit k
O_k Softmax output of output unit k
a_i input value i
$g'(in_j)$ derivative of ReLU activation function
$\Delta_k = (T_k - O_k)$

88

Updating Weights in a 2-Layer Neural Network

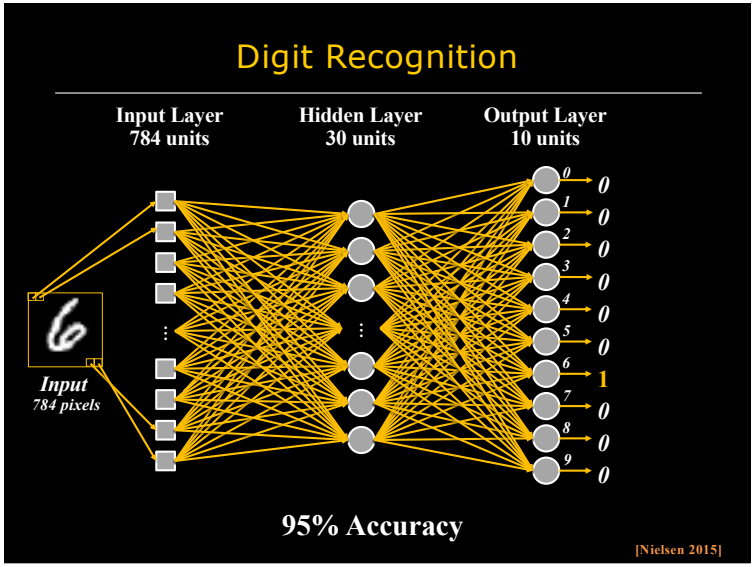
For **weights between hidden and output units**, using the **Softmax activation function** and **Cross-Entropy error function**:

$$\Delta w_{j,k} = -\alpha\, \partial E / \partial w_{j,k}$$
$$= \alpha\, a_j\, (T_k - O_k)$$
$$= \alpha\, a_j\, \Delta_k$$

where

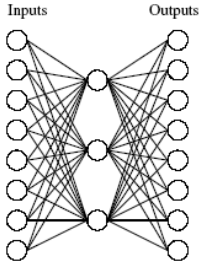
$w_{j,k}$ weight on link **from hidden unit j to output unit k**
α learning rate parameter
a_j activation (i.e., output) of hidden unit j
T_k teacher output for output unit k
O_k Softmax output of output unit k

87



89

Learning Hidden Layer Representation



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

Training set size = 8

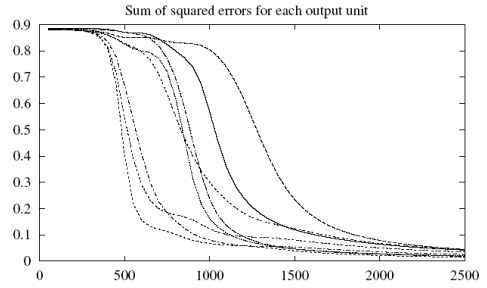
Can this be learned?

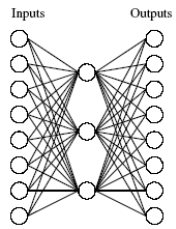
Slide by Guoping Qiu

90

Learning Hidden Layer Representation

Training



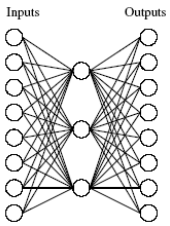


The evolving sum of squared errors for each of the eight output units

Slide by Guoping Qiu

91

Learning Hidden Layer Representation



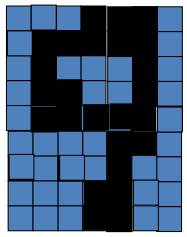
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Learned hidden layer representation (encoder)

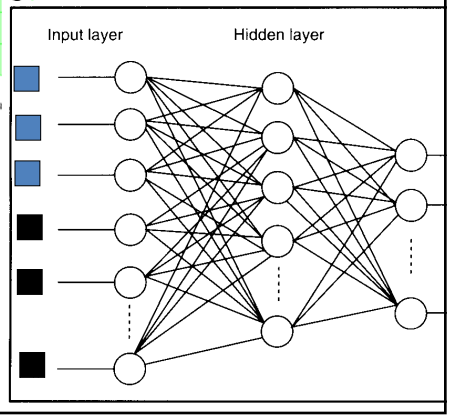
Slide by Guoping Qiu

92

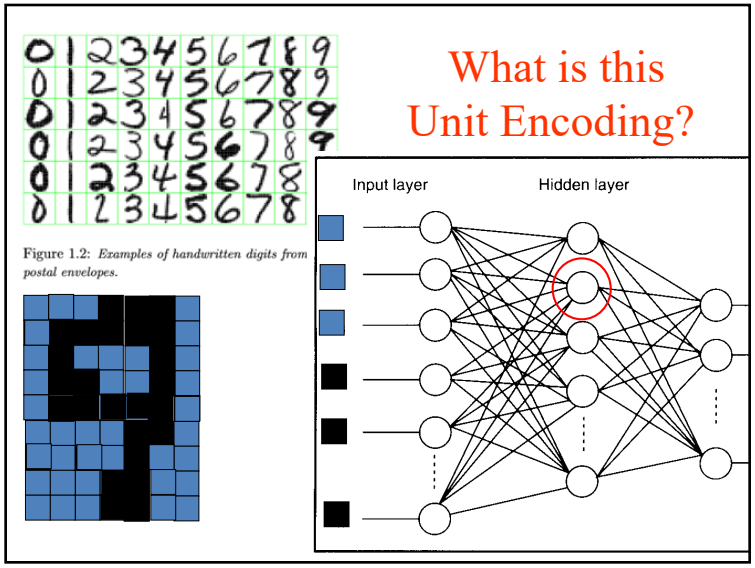
Figure 1.2: *Examples of handwritten digits from postal envelopes.*



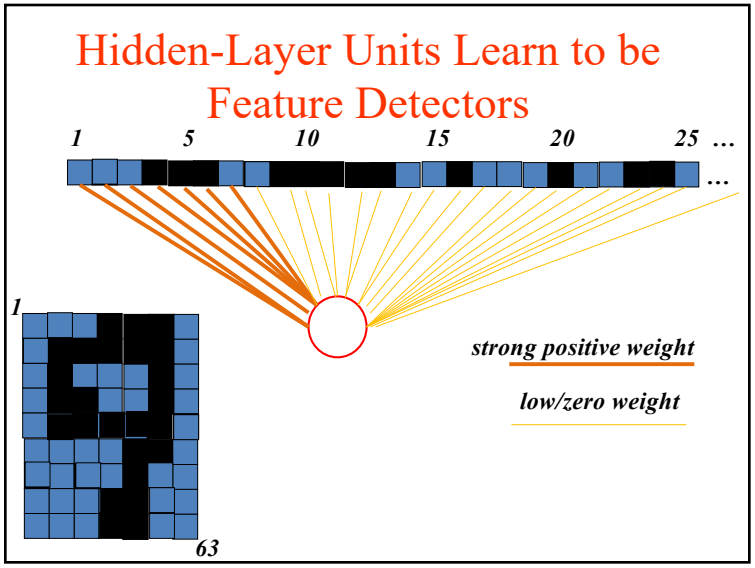
Feature Detectors



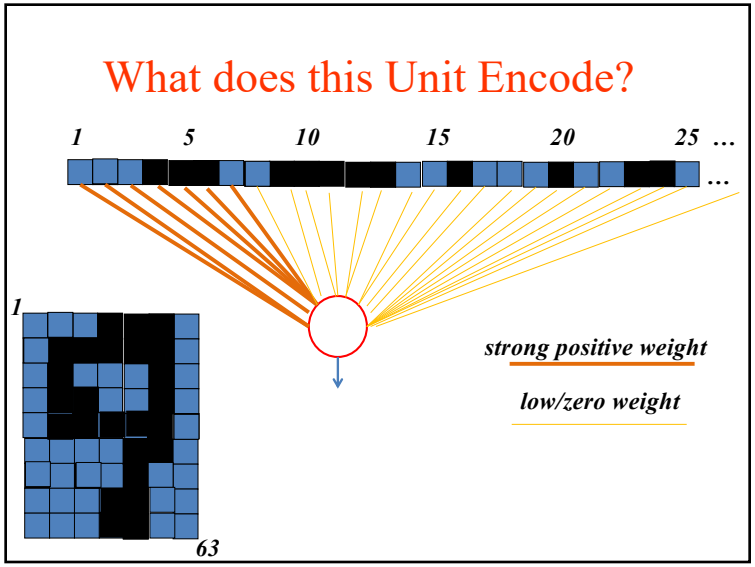
93


What is this Unit Encoding?
Figure 1.2: Examples of handwritten digits from postal envelopes.
Input layer
Hidden layer

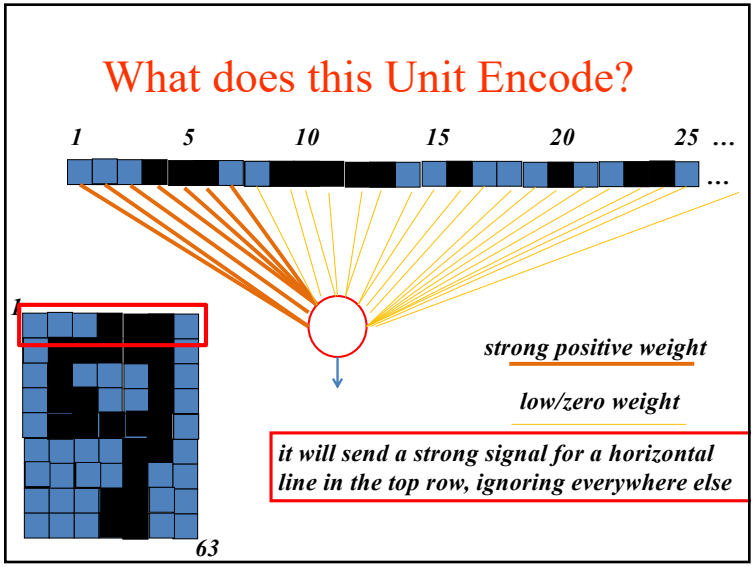
94


Hidden-Layer Units Learn to be Feature Detectors
1
5
10
15
20
25 …
…
1
63
strong positive weight
low/zero weight

95

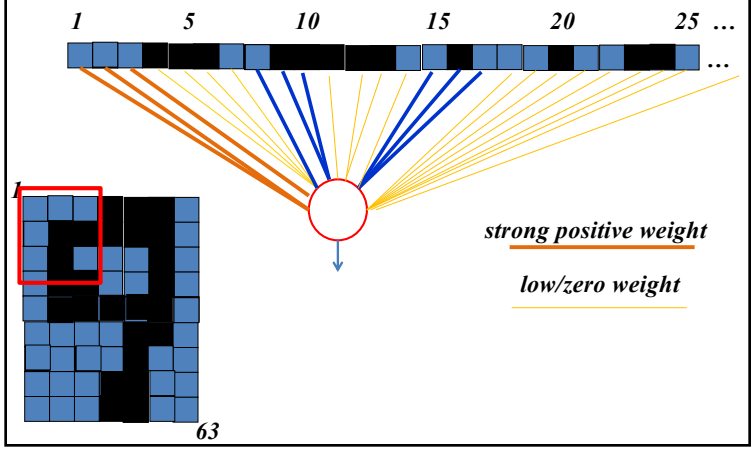

What does this Unit Encode?
1
5
10
15
20
25 …
…
1
63
strong positive weight
low/zero weight

96


What does this Unit Encode?
1
5
10
15
20
25 …
…
1
63
strong positive weight
low/zero weight
it will send a strong signal for a horizontal line in the top row, ignoring everywhere else

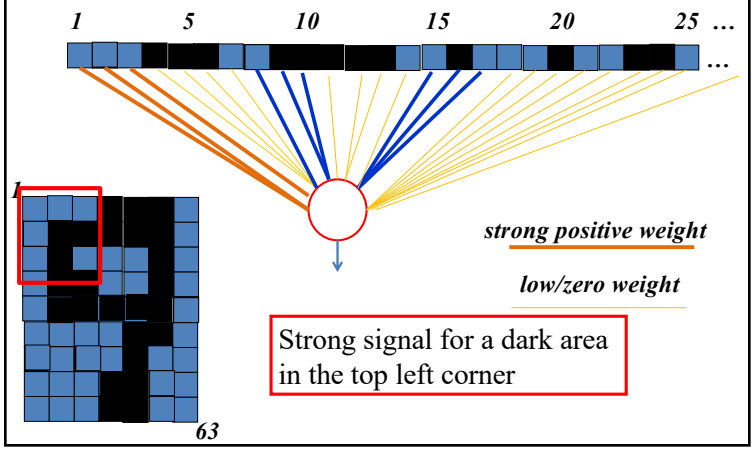
97

What does this Unit Encode?



98

What does this Unit Encode?



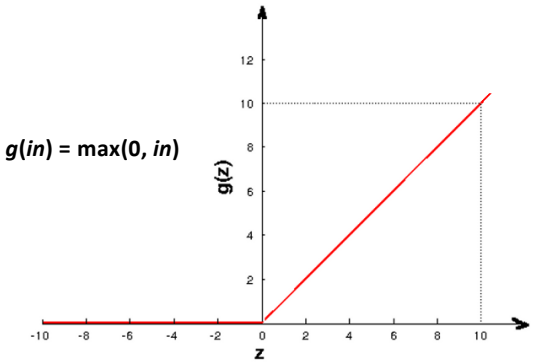
99

Multi-layer Feed-forward Networks

- Back-propagation algorithm performs gradient descent in the "weight space" of the network
- Include a bias input at *every* (non-input) unit
- In practice, use ReLU, not Sigmoid, for the activation function in order to speed up training
- Will, in general, find a **local, not global, error minimum**
- Training can take thousands of epochs, but can be computed efficiently using matrix vector operations and GPU implementation

106

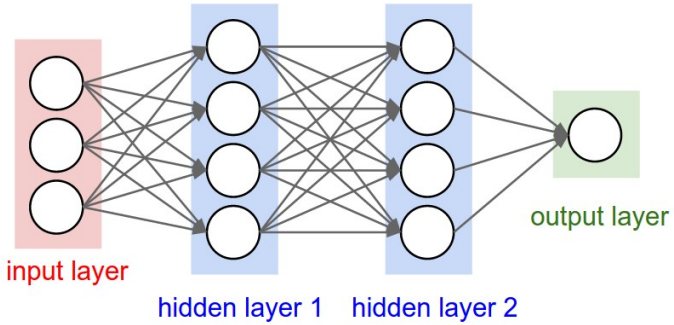
Rectified Linear Function (ReLU)



The rectified linear activation function

107

How Many Weights?



3 layers, 9 units with activation function,
((3x4)+4) + ((4x4)+4) + ((4x1)+1) = 41 weights incl. biases

109

Other Issues

How should the learning rate parameter, α, be set?

Use a **tuning set** (aka **validation set**) to train using several candidate values for α, and then select the value that gives the lowest error

110

Using Data

- **Training set** is used to *learn a "model"* (i.e., the neural network's weights)
- **Tuning set** is used to judge and *select parameters* (e.g., learning rate and the number of hidden units with best accuracy)
- **Testing set** is used to judge in a fair manner the final *model's performance accuracy* (using cross-validation or leave-one-out)
- All 3 datasets must be disjoint!

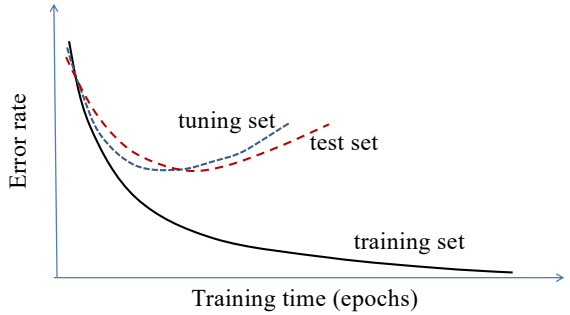
112

Other Issues

- When should training stop?
  - too soon and the concept isn't learned
  - too late:
    - overfitting, poor generalization
    - error rate will go up on the testing set
- Train the network until the error rate on a **tuning set** begins *increasing* rather than training until the error (SSE) on the training set is minimized

114

Tuning Sets



115

Other Issues

How many examples should be in the training set?

– ***The more the better, but training takes longer***

To obtain $1 - e$ correct classification on testing set, the training set should be of size approximately n/e

- n is the number of weights in the network
- e is test set error fraction between 0 and 1

– train to classify $1 - e/2$ of the training set correctly

e.g., if $n = 80$ and $e = 0.1$ (i.e., 10% error on test set)

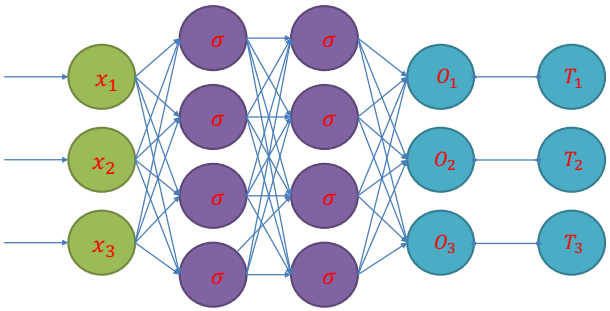
– Use training set of size is 800

– train until 95% correct classification on training set

– should produce ~90% correct classification on test set

118

Recap: Multi-Layer, Feed-Forward Neural Networks



119

Back-Propagation Recap

- Back-Propagation performs a ***gradient descent search*** in "weight space" to learn the neural network weights that minimize the total error on the training set
- Given a neural network with *n* weights:
  - each configuration of weights is a vector, W, of length *n* that defines an instance of the network
  - W can be considered a point in an *n*-dimensional **weight space**, where each dimension is associated with one of the connections in the network

120

Updating the Weights

- Given a training set of m examples:
  - Each network defined by the vector $\boldsymbol{W}$ has an associated total error, E, on ***all*** the training data
  - E is the sum squared error (SSE) defined by
    $$E = E_1 + E_2 + \ldots + E_m$$
    where E_i is the squared error of the network on the i^{th} training example
- Given t output units in the network:
  $$E_i = (T_{i1} - O_{i1})^2 + (T_{i2} - O_{i2})^2 + \ldots + (T_{it} - O_{it})^2$$
  - T_{ij} is the target value for the i^{th} example's j^{th} output unit
  - O_{ij} is the network output value for the i^{th} example's j^{th} output unit

121

Back-Propagation using SGD

Initialize the weights in the network (usually random values)
Repeat until stopping criterion is met
  foreach example, *ex,* in training set **do**
    { O = neural_net_output(network, *ex*) **forward pass**
    T = desired output, i.e., **T**arget or **T**eacher's output
    calculate error (T_k - O_k) at each output unit *k*
    foreach hidden unit *j* and output unit *k* compute $\Delta w_{j,k}$
    foreach input unit *i* and hidden unit *j* compute $\Delta w_{i,j}$
    forall *p*, *q* in network $w_{p,q} = w_{p,q} + \Delta w_{p,q}$
    }

backward pass

Simplistic SGD: update all weights after *each* example

122

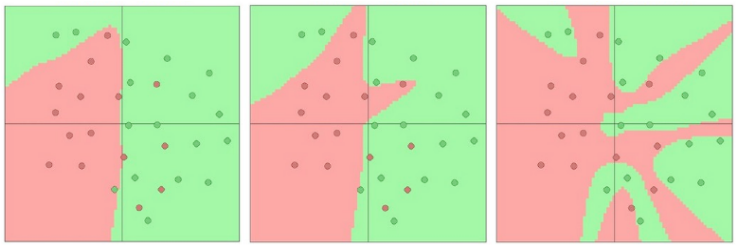
Other Issues

- How many hidden layers should be in the network?
  - often just one hidden layer is used
- How many hidden units should be in a layer?
  - too few: concept *can't* be learned
  - too many:
    - examples just memorized
    - overfitting, poor generalization

126

How Many Hidden Units?

Decision regions learned by a 2-layer network:
Larger networks can represent more complicated functions, but make it easier to overfit the training data



3 hidden units 6 hidden units 20 hidden units

127

Application: Autonomous Driving

- ALVINN (Pomerleau, 1988)
  learned to control vehicle steering to stay in the middle of its lane
- Topology: 2-layer, feed-forward network using back-propagation learning
  - **Input layer:** 480 × 512 image @ 15 frames per second
    - color image is preprocessed to obtain a 30 × 32 image
    - each pixel is one byte, an integer from 0 to 255 corresponding to the brightness of the image
    - networks has 960 input units (= 30 × 32)

132

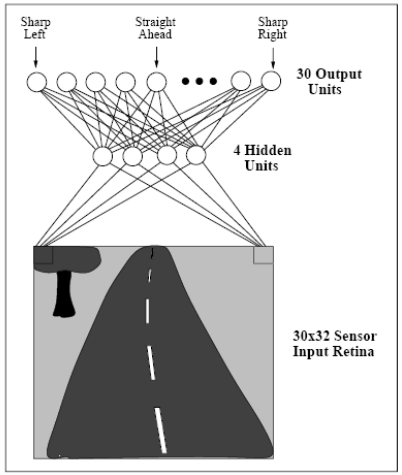
ALVINN

steering direction

[Pomerleau, 1995]

133

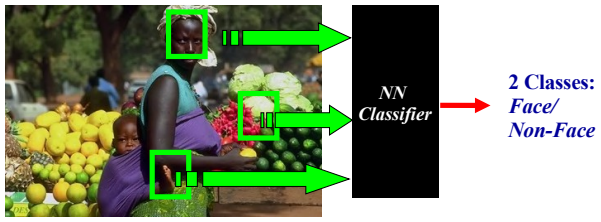
ALVINN's 2-Layer Network



[Pomerleau, 1995]

136

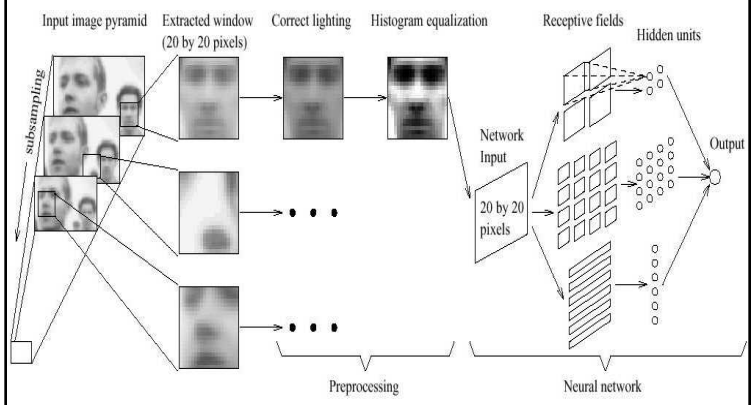
Application: Face Detection



- Input = 20 x 20 pixel window, outputs a value ranging from –1 to +1 signifying the presence or absence of a face in the region
- The window is positioned at every location of the image
- To detect faces larger than 20 x 20 pixel, the image is repeatedly reduced in size

142

2-Layer Network



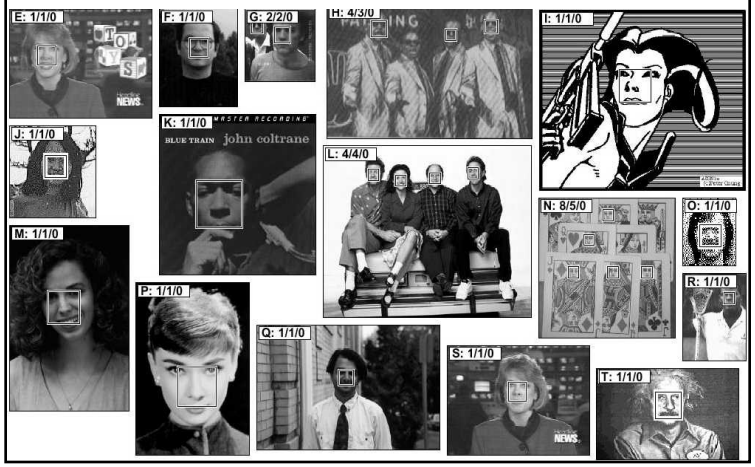
143

Application: Face Detection

- 2-layer feed-forward neural network
- Three types of hidden units
  - 4 look at 10 x 10 subregions
  - 16 look at 5 x 5 subregions
  - 6 look at 20 x 5 horizontal stripes of pixels
- Training set
  - 1,050 initial face images. More face examples generated from this set by rotation and scaling. Desired output: +1
  - Non-face training samples: 8,000 non-face training samples from 146,212,178 subimage regions! Desired output: -1

144

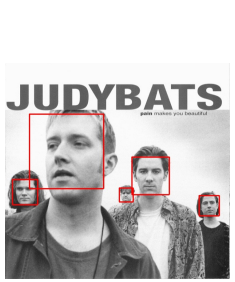
Face Detection Results



146

Results

- Notice detection at multiple scales



147

What is Wrong with Back-Propagation?

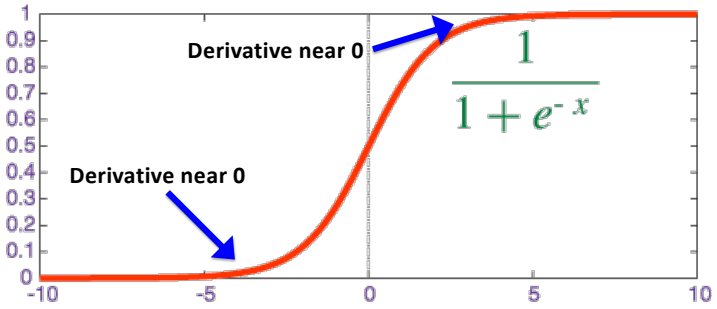
- Requires labeled training data
  - Almost all data is **unlabeled**
- Learning time does not scale well
  - It is very slow to converge in networks with multiple hidden layers
- Might get stuck at a poor local minimum
- Overfitting problem

from G. Hinton

150

Sigmoid Function

Slow learning because "saturated" units "kill" gradients

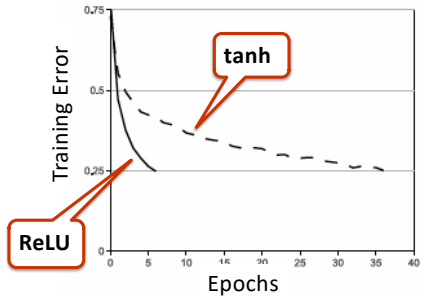


151

Speeding Up Training

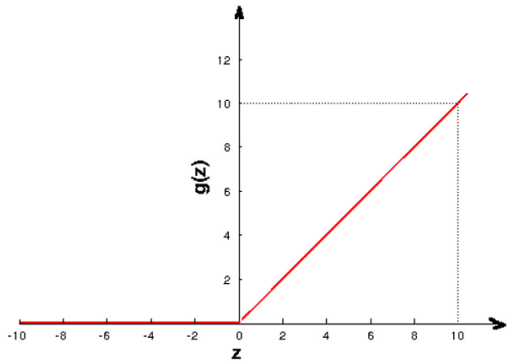
Use ReLU instead of Sigmoid (or Tanh)

- Rectified Linear Unit: $g(x) = \max(0, x)$
- ReLU reaches 25% error rate 6x faster than tanh
- Computationally efficient



152

Rectified Linear Function (ReLU)



The rectified linear activation function

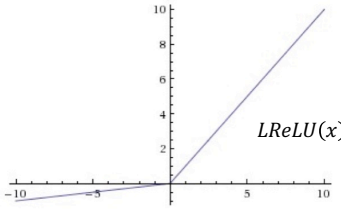
153

What's the Derivative of the ReLU Function?

$$g'(x) = \begin{cases} 0, & \text{if } x \le 0 \\ 1, & \text{otherwise} \end{cases}$$

But this means when x ≤ 0 the unit will never update because $\Delta_i = 0$

– Fix using a variation such as "**Leaky ReLU**":



$$LReLU(x) = \begin{cases} \alpha \boldsymbol{x}, & \boldsymbol{x} < 0 \\ \boldsymbol{x}, & \boldsymbol{x} \ge 0 \end{cases}$$

154

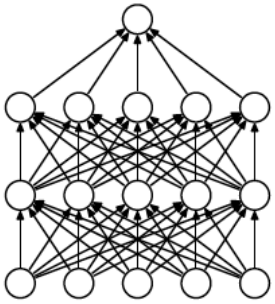
Preventing Overfitting: Dropout

To prevent overfitting the training set, "dropout" is often performed:

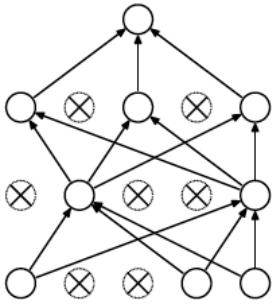
- During training: At each hidden layer, "drop out" a random set of units from that layer by setting their outputs to 0 in the forward pass
  - Each unit is retained with probability *p* (= 0.5)
  - Back-propagation is performed only on the thinned network
- During testing: Use the whole network (but weights must all be re-scaled)

155

Dropout



Original Neural Net



Neural Net after Dropping Out "X"ed nodes

156

Why Dropout?

- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data
  - But complex co-adaptations are likely to go wrong on new test data
  - Big, complex conspiracies are not robust
- If a hidden unit has to work well with combinatorially many sets of co-workers, it is more likely to do something that is individually useful
  - But it will also tend to do something that is only marginally useful given what its co-workers achieve

G. Hinton

157

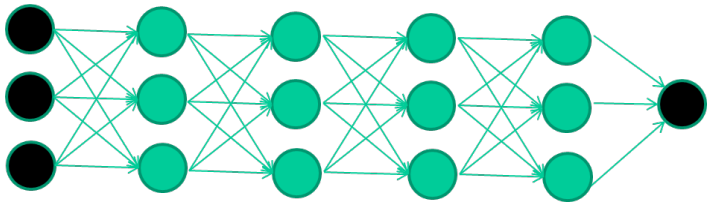
- Early NNs used only 1 or 2 hidden layers because training was slow, thought to converge to poor local minima, and overfit

- Recent results suggest that local minima are ***not*** a serious problem – regardless of initial conditions, *bigger NNs contain many more local minima but nearly all are solutions of very similar quality*. And overfitting can be avoided using regularization techniques such as dropout rather than using fewer units

158

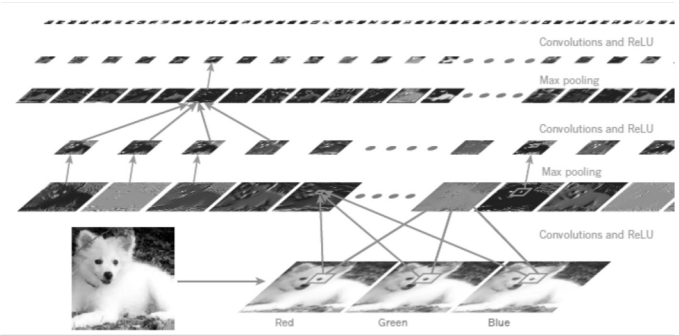
Deep Learning

- Use *many hidden layers* (often 10-20)
- Called **Deep Neural Nets** or **Deep Nets**



161

Reading: "Deep learning," Y. LeCun, Y. Bengio and G. Hinton, *Nature*, 2015



162

A Lot of Buzz about Deep Learning

Google Scoops Up Neural Networks Startup DNNresearch To Boost Its Voice And Image Search Tech

RIP EMPSON

Tuesday, March 12th, 2013 5 Comments

Similarly, big groups at Facebook, Microsoft, Baidu, Amazon, …

166

Why Deep Learning?

- Biological Plausibility – your brain works that way
- Problems that can be represented with a polynomial number of nodes with *k* layers, may require an exponential number of nodes with *k*-1 layers
- Complex functions can be efficiently represented with deep architectures
  - Fewer weights/parameters to update than a less efficient shallow representation
- Features learned in deep architecture can potentially be shared between multiple tasks
  - Type of Transfer/Multi-task learning

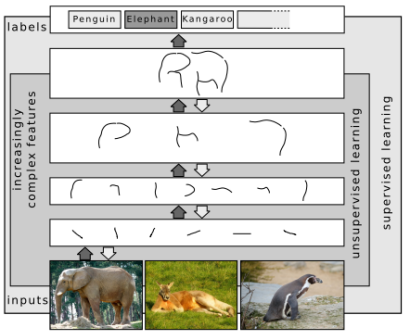
170

Why Deep Learning?

- Learn better features: **Features matter!**
- Use **context** better

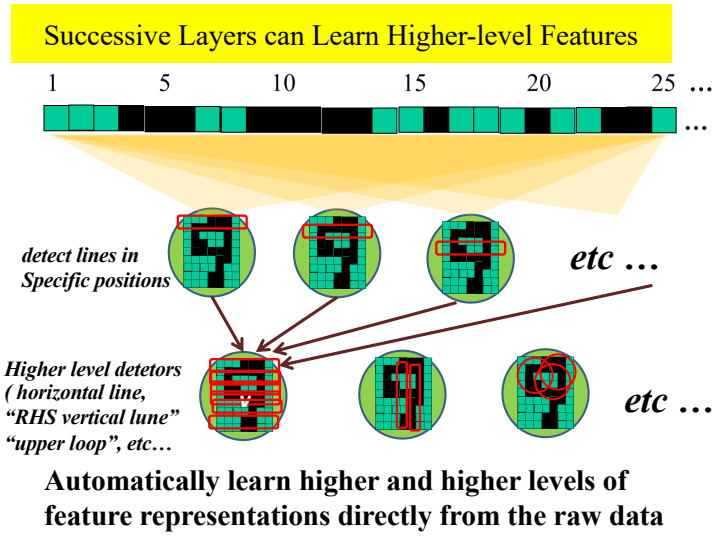
171

Hierarchical, Representation Learning



- Natural progression from low-level to high-level features as seen in natural complexity
- **Automatically learns features from the raw data**
- Easier to monitor what is being learned and to guide the learner to better subspaces

172



174

Difficulties Training Deep Networks

- Bottom layers do not get trained easily
  - **Error attenuates as it propagates to earlier layers**
  - Leads to very slow training
- For small data sets, not enough labeled data available for training
  - Use a pre-processing step to initialize weights
  - Then use traditional back-prop to train the full network

181

Convolutional Neural Networks

also called **CNNs** and **ConvNets**

231

CNNs

- Many applications process **array data**
  - 1D: time-series, acoustic, text data
  - 2D: images and audio spectrograms
  - 3D: video and volumetric data
- Properties of these types of data
  - Local, shift-invariant processing (i.e., doesn't depend on position in the input array)
  - Hierarchy of feature descriptions
  - Noisy data

232

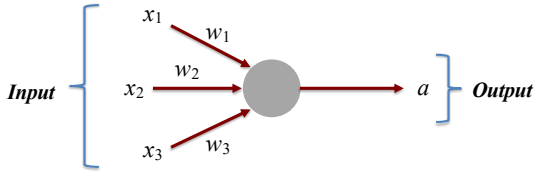
CNNs are an extension of traditional multi-layer, feed-forward networks that incorporate **4 key ideas**:

- Use of **many layers**
  - Learn a hierarchy of features
- **Local "receptive fields"/filters and local connections**
  - Layers are *not* completely connected
  - Want translation-invariant and distortion-invariant local features
- **Shared weights**
- **Pooling**

233

Computation by a Unit

Takes several inputs and computes a single scalar output value



$$a = g(w_1x_1 + w_2x_2 + ... + w_nx_n) = g(\sum_{i=1}^{n} w_ix_i) = g(in)$$

First step is to compute *in*, the linear combination of the inputs

234

Image Filtering

Filter

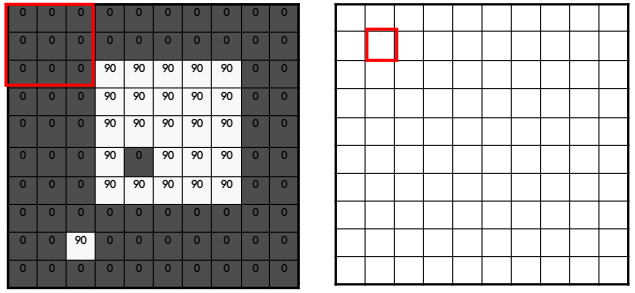
$g[\cdot,\cdot]$ $\frac{1}{9}$ [1 1 1; 1 1 1; 1 1 1]

Input Image

$f[.,.]$

Output Image

$h[.,.]$



$$h[m,n] = \sum_{k,l} g[k,l]\, f[m+k,n+l]$$

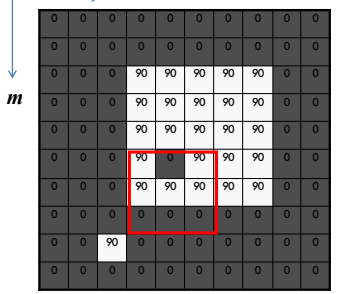
Credit: S. Seitz

235

Image Filtering

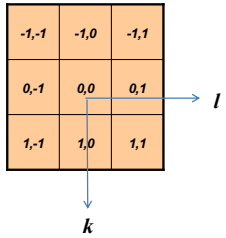
$g[\cdot,\cdot]$

$f[.,.]$



$\frac{1}{9}$

-1,-1	-1,0	-1,1
0,-1	0,0	0,1
1,-1	1,0	1,1



$$h[m,n] = \sum_{k,l=-1}^{+1} g[k,l] f[m+k,n+l]$$

$h[7,5] = g[-1,-1]f[6,4] + g[-1,0]f[6,5] +$
$g[-1,1]f[6,6] + g[0,-1]f[7,4] +$
$g[0,0]f[7,5] + ...$

Credit: S. Seitz

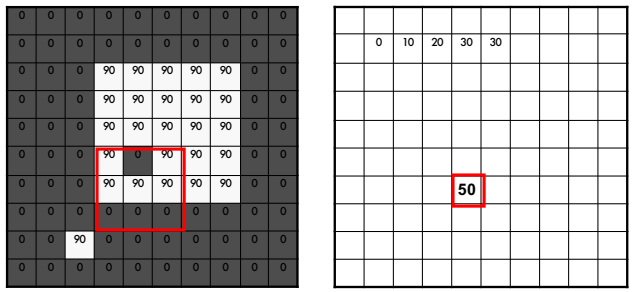
236

Image Filtering

$g[\cdot,\cdot]$ $\frac{1}{9}$ [1 1 1; 1 1 1; 1 1 1]

$f[.,.]$

$h[.,.]$



$$h[m,n] = \sum_{k,l} g[k,l]\, f[m+k,n+l]$$

Credit: S. Seitz

237

Example: Box Filter

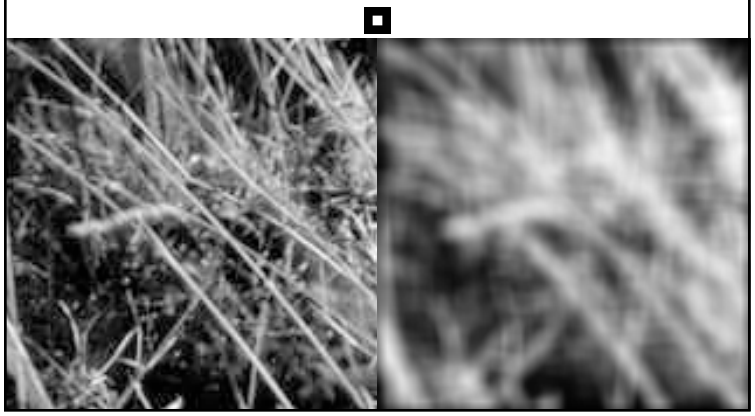
$g[\cdot,\cdot]$

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} = \begin{bmatrix}1/9 & 1/9 & 1/9\\ 1/9 & 1/9 & 1/9\\ 1/9 & 1/9 & 1/9\end{bmatrix}$$

Credit: David Lowe

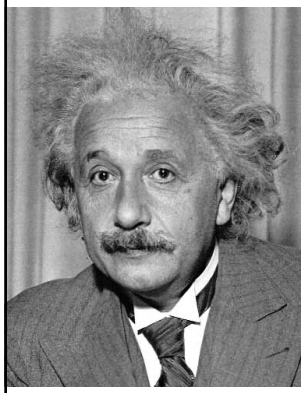
238

Image Smoothing with Box Filter



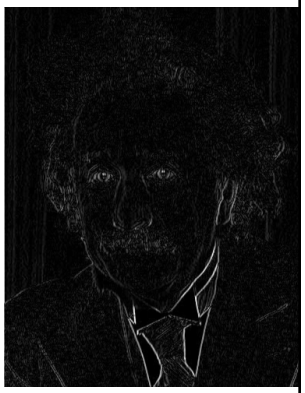
239

Vertical Edge Detection Filter



1	0	-1
2	0	-2
1	0	-1

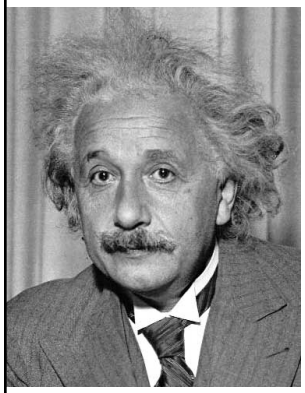
Sobel filter



Vertical edges (absolute value)

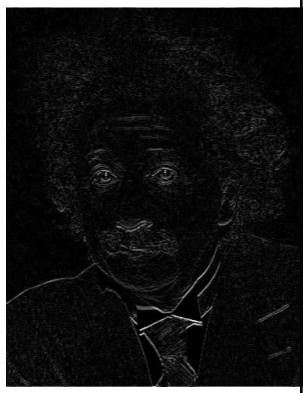
240

Horizontal Edge Detection Filter



1	2	1
0	0	0
-1	-2	-1

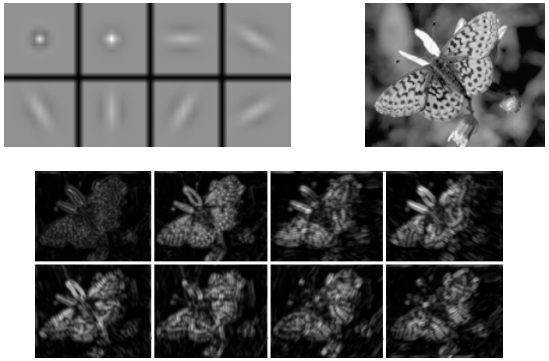
Sobel filter



Horizontal edges (absolute value)

241

Other Filters



243

Why are they called "Convolutional" NNs?

The image filtering operation defined as

$$h[m,n] = \sum_{k,l} g[k,l]\, f[m+k, n+l]$$

is very similar to the **Convolution** operation defined as

$$h[m,n] = \sum_{k,l} g[k,l]\, f[m-k, n-l]$$

- **In CNN's, *f* corresponds to the inputs from the layer below and *g* corresponds to the weights**
- So, CNN's will learn a set of filters!

Credit: K. Grauman

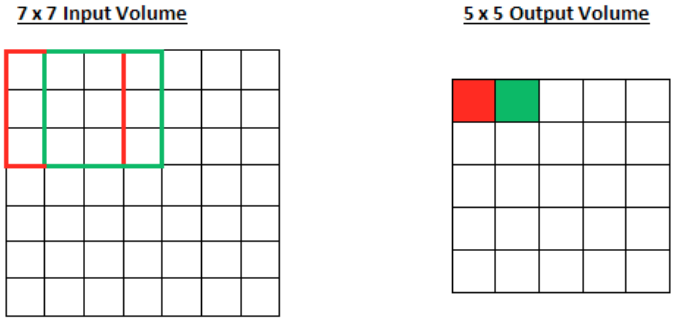
244

Convolution Layers

- Learn "filters" (i.e., weights) that process only **small regions** of the layer below it and compute "features" at many spatial positions
- Example: 32 x 32 x 3 input RGB image, and receptive field (filter size): 5 x 5
  - Each unit in the "Conv layer" will have weights connected to a 5 x 5 x 3 region in the input layer, with 5*5*3 = 75 weights
  - Can have multiple units associated with a given receptive field in order to compute multiple features at each position
  - "Stride" parameter defines shift amount

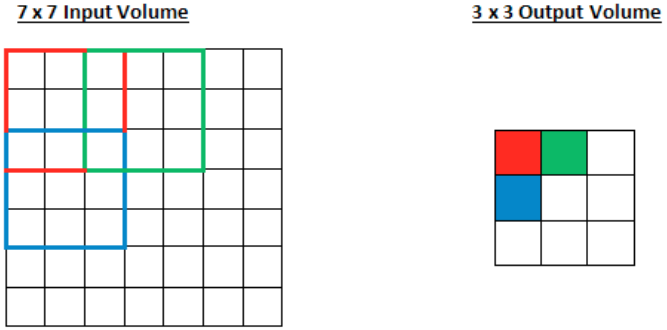
245

3 x 3 Filter, Stride = 1



246

3 x 3 Filter, Stride = 2

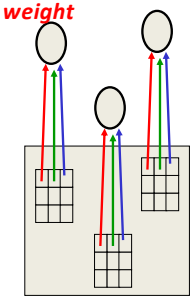


247

The Replicated Feature Approach: Local Connections and Shared Weights

- Use many different copies of the *same feature detector* at different positions
  - Could also replicate across scale and orientation (but tricky and expensive)
  - Replication greatly reduces the number of weights to be learned
- Use several different feature types, each with its own map of replicated detectors
  - Allows each patch of image to be represented in several ways

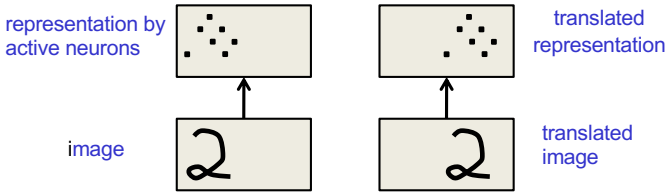
The red connections all have the same weight



248

What does Replicating the Feature Detectors Achieve?

- Equivariant activities

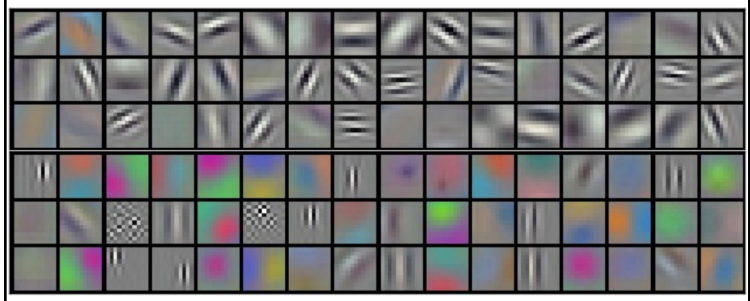


- Invariant knowledge: If a feature is useful in some locations during training, detectors for that feature will be available in ***all*** locations during testing

249

Learned Features

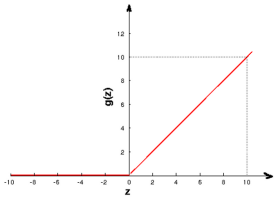
96 learned low-level (1st layer) filters:



250

ReLU Layers

- Each Convolution layer often followed by an "**ReLU layer**," which applies the ReLU function to the output of each "**Convolution layer**"
- No weights to be learned
- Layer size equal to size of the layer below it



The rectified linear activation function

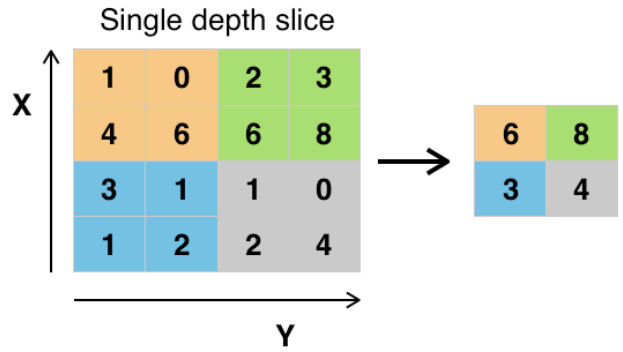
251

Pooling Layers

- Each ReLU layer usually followed by a "**Pooling layer**"
- Get a small amount of translational invariance and robustness to noise by combining *neighboring values* to give a *single output* at the next layer
- **Max Pooling**: Combine by taking the *maximum*
- **Average Pooling**: Combine by taking the *average*
- **Sub-Sampling**: Reduces the size of the next layer according to the size of the pooling filter
  - For example, a 2 x 2 pooling filter reduces the size by ½ in each dimension

252

Max Pooling Example



2 x 2 filter with a stride of 2

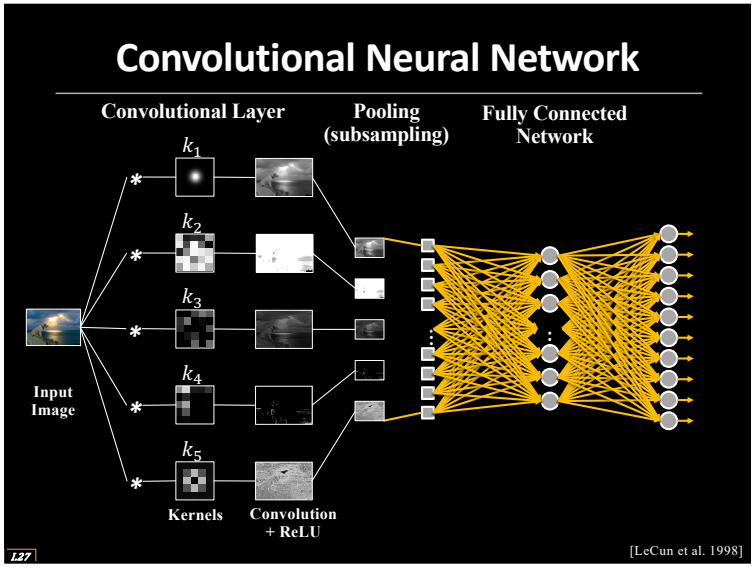
Adit Deshpande

253

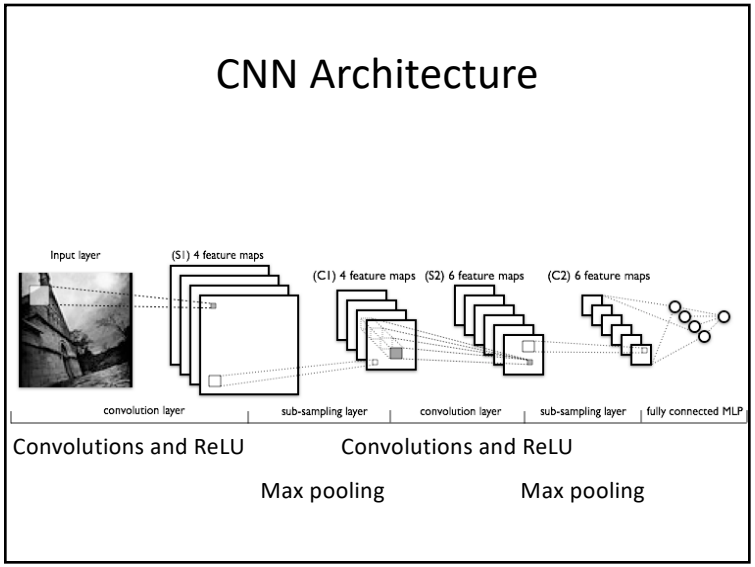
Top Layers Fully Connected

- Top few layers are usually fully-connected, feed-forward layers
- Hidden layers and Output layer

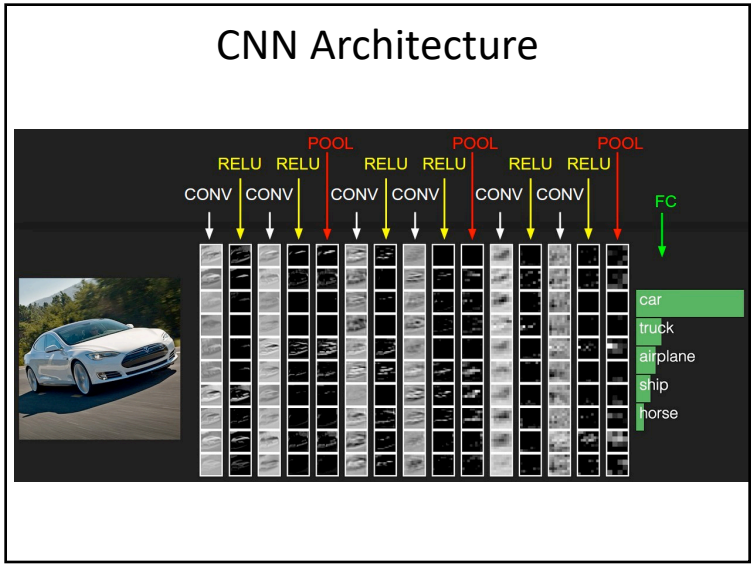
254



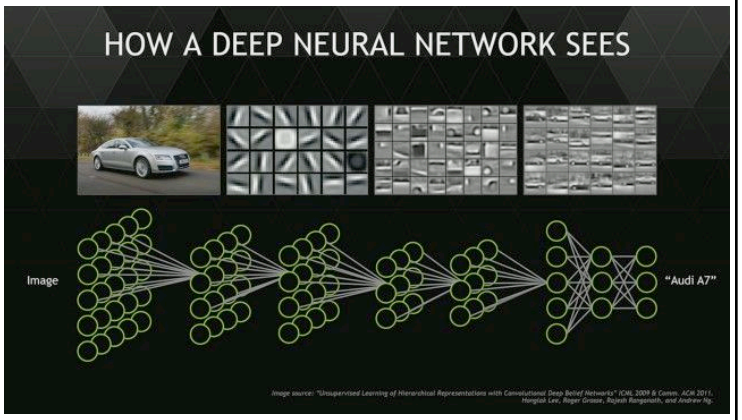
257



258



259



260

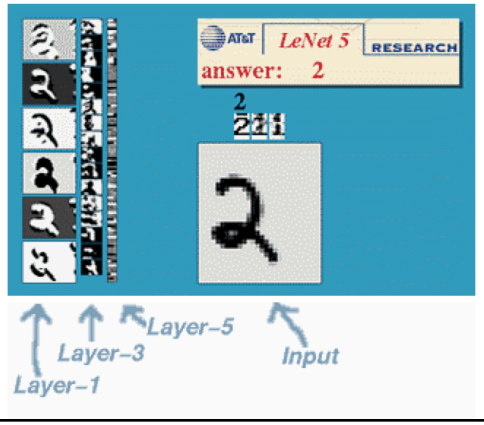
CNNs are an extension of traditional multi-layer, feed-forward networks that incorporate **4 key ideas**:

- Use of **many layers**
  - Learn a hierarchy of features
- **Local "receptive fields"/filters and local connections**
  - Layers are *not* completely connected
  - Want translation-invariant and distortion-invariant local features
- **Shared weights**
- **Pooling**

262

LeCunn's Digit Recognizer: LeNet

http://yann.lecun.com/exdb/lenet/index.html



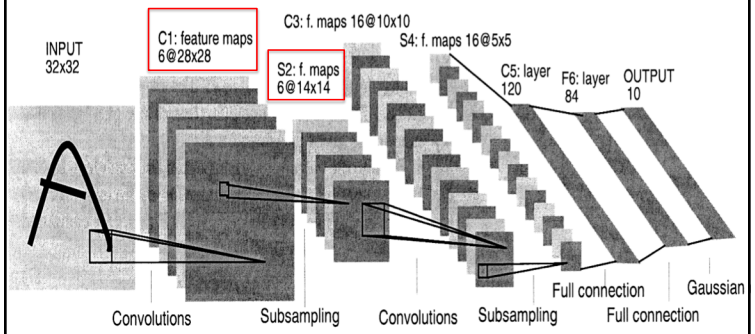
263

LeNet

- Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using backpropagation in a feed-forward net with:
  - Many hidden layers
  - Many maps of replicated units in each layer
  - Pooling of the outputs of nearby replicated units
  - A wide net that can cope with several characters at once even if they overlap
- This net was used for reading ~10% of the checks in North America

264

The Architecture of LeNet5



265

From Handwritten Digits to 3D Objects

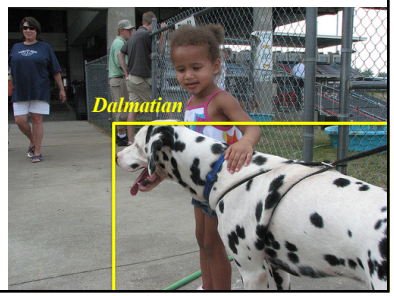
- Recognizing real objects in color photographs downloaded from the web is much more complicated than recognizing handwritten digits:
  - Hundred times as many classes (1,000 vs 10)
  - Hundred times as many pixels (256 x 256 color vs 28 x 28 grayscale images)
  - Cluttered scenes
  - Multiple objects in each image
- Will the same type of CNN work?

267

IMAGENET Large Scale Visual Recognition Challenge (ILSVRC 2010)

Training set: 1.2 million 256 x 256 RGB images

- Classification task:
  –Get the "correct" class in your top 5 predictions. There are **1000 classes**
- Localization task:
  –For each prediction, put a box around the object. Your box must have at least 50% overlap with the correct box



268

ImageNet

- Over 15M labeled high-resolution images
- About 22K categories
- Collected from the web and manually labeled by Amazon Mechanical Turk

269

Variety of Object Classes in ILSVRC

270

AlexNet

A. Krizhevsky, I. Sutskever, G. Hinton, *NIPS* 2012

- 7 hidden "weight" layers:
- `data` → `conv1` → `pool1` → `conv2` → `pool2` → `conv3` → `conv4` → `conv5` → `pool3` → `full` → `full` → `output`
- 650K units, 60M weights, 630M connections
- ReLUs, max pooling, dropout trick
- Top feature layer: 4096 units
- Output layer: 1000 units
- Trained with SGD on two GPUs for a week
- Classification: "Top-1" error rate: 37.5%
  "Top-5" error rate: 17%
- Localization: Regression on (x,y,w,h)

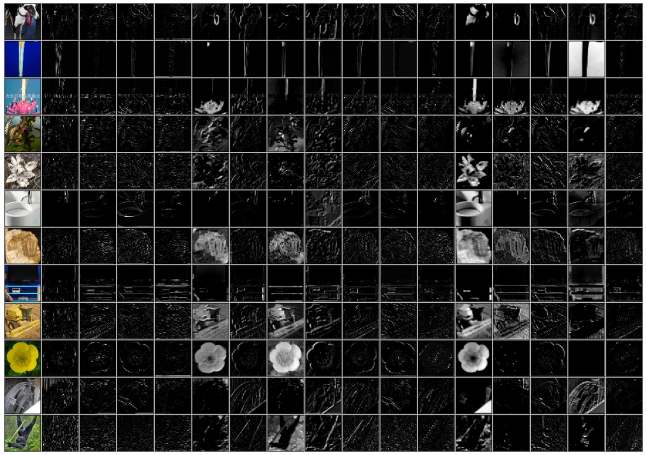
271

Dropout Observations by G. Hinton

- If your deep neural net is significantly overfitting, dropout will usually reduce the number of errors by a lot
  - Any net that uses "early stopping" can do better by using dropout (at the cost of taking quite a lot longer to train)
- If your deep neural net is not overfitting, you should be using a bigger one!

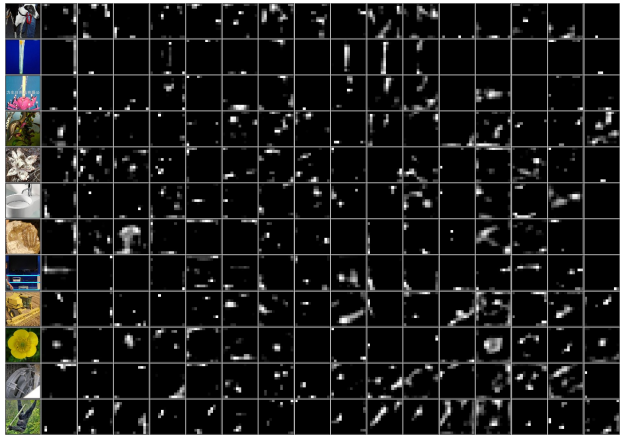
272

Learned Features at 1st Layer



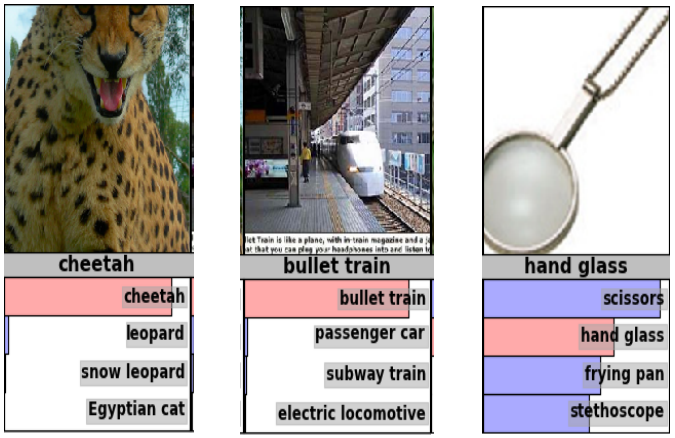
273

Learned Features at 5th Layer



274

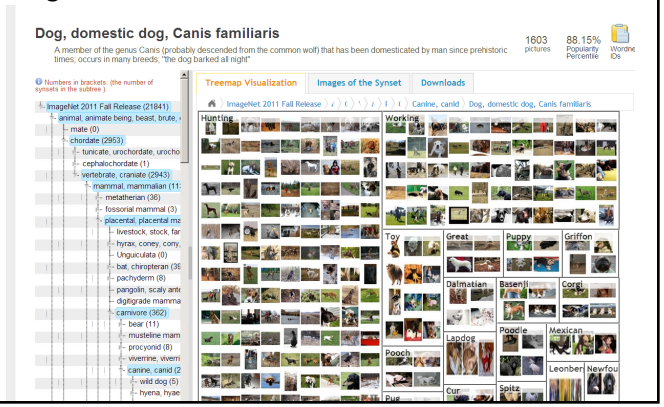
Examples from the Test Set (with the network's predictions)



276

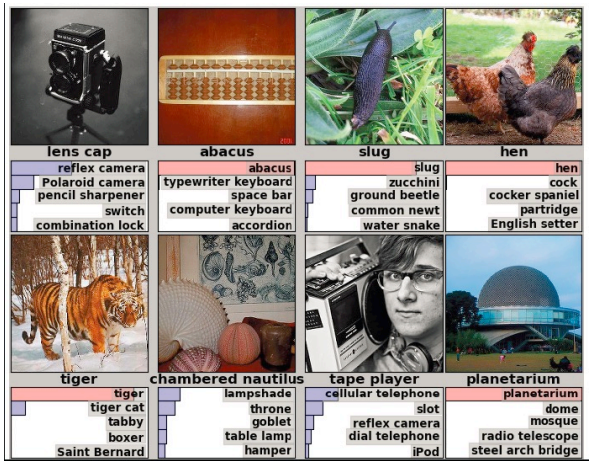
ILSVRC 2012

Training set: 15 million labeled images from ImageNet, 20K categories



277

Example Classification Results

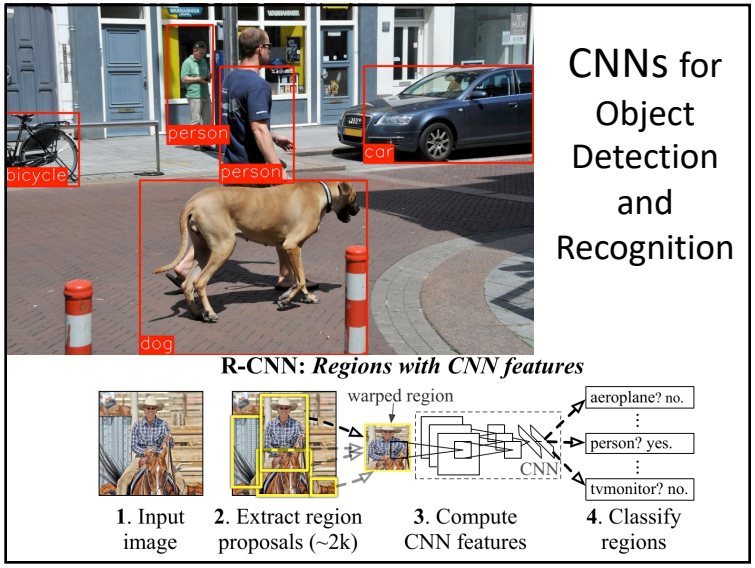


278

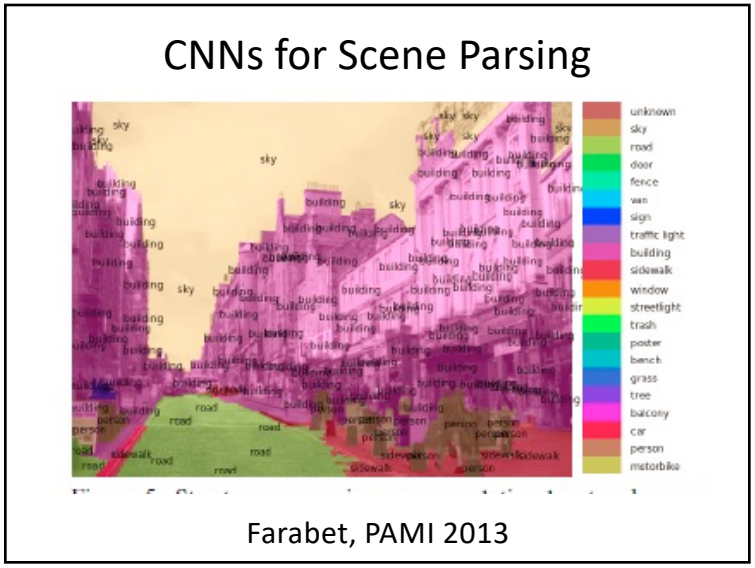
Error Rates on the ILSVRC 2012 Competition

	classification	*classification & localization*
• University of Toronto (CNN)	16.4%	34.1%
• University of Tokyo	26.1%	53.6%
• Oxford University	26.9%	50.0%
• INRIA + Xerox	27.0%	
• University of Amsterdam	29.5%	

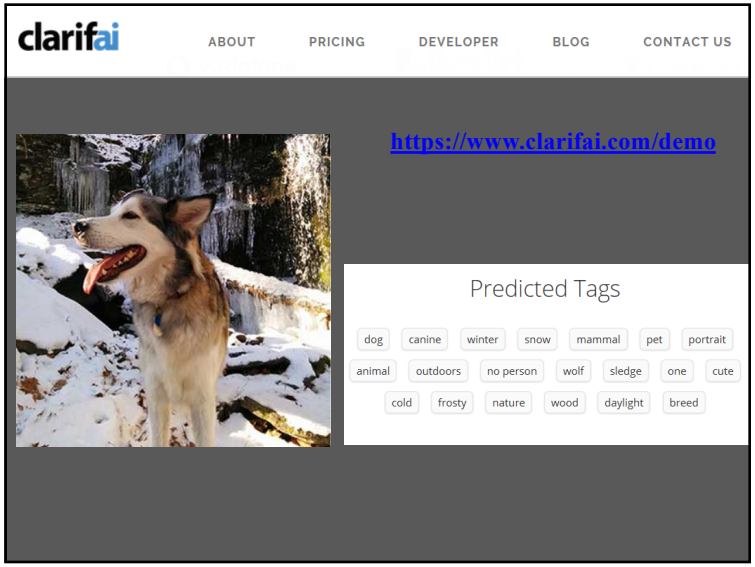
280



286



288



290

294

Deep Learning Demos and Startups

- Clarifai
  - http://www.clarifai.com
- Dextro
  - www.dextro.co
- Microsoft's CaptionBot
  - https://www.captionbot.ai
- Yann LeCun's Digit Recognizer
  - http://yann.lecun.com/exdb/lenet/
- Toronto Deep Learning Demos
  - http://deeplearning.cs.toronto.edu/

296

Number Recognition from Google's Street View Images

297

Number Recognition from Google's Street View Images

- Training set: 200,000 images of (cropped) house numbers (up to 5 digits long)
- CNN with 11 hidden layers
  - 54 x 54 input image layer
  - 8 convolutional layers
    - 48 x 54 x 54 = 140,000 units at first hidden layer
    - 5 x 5 convolution kernels
  - 1 locally-connected hidden layer
  - 2 densely-connected hidden layers
    - 3,072 units in each

298

- Training time: 6 days
- Accuracy: Transcribed ~100 million numbers with ~98% accuracy (meaning *all* numbers in a sequence were recognized correctly) and 95% coverage

299

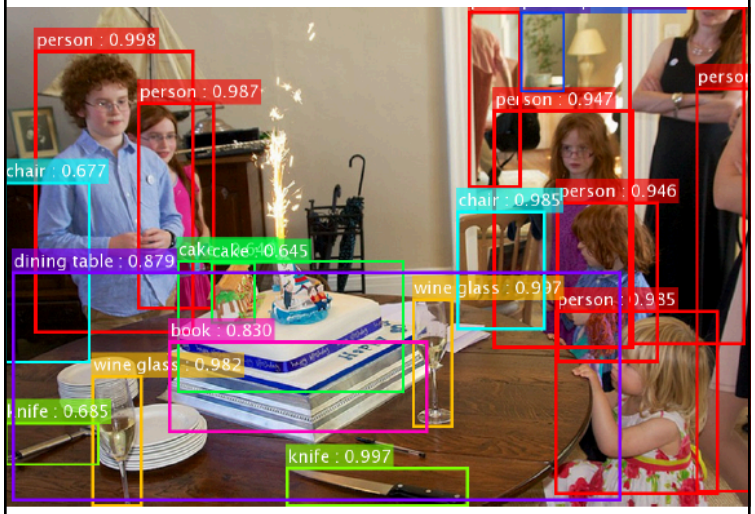
ResNets @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: "*Ultra-deep*" 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

Also have tried using a 1000-layer network!

Deep residual learning for image recognition, K. He *et al.*, *Proc. CVPR*, 2016 (Best paper award)

301



302

Results on Real Video (trained on 80 categories)

303

CNNs for Video Classification

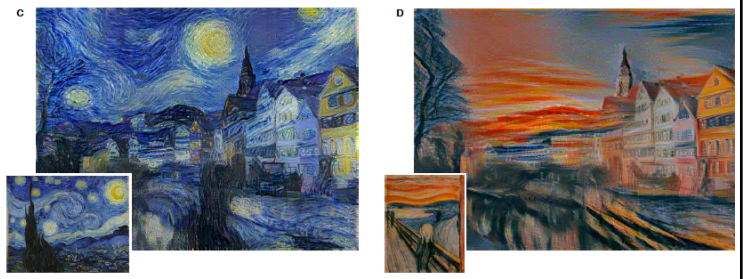
[Karpathy 2014]

304

Image Style Transfer using CNN

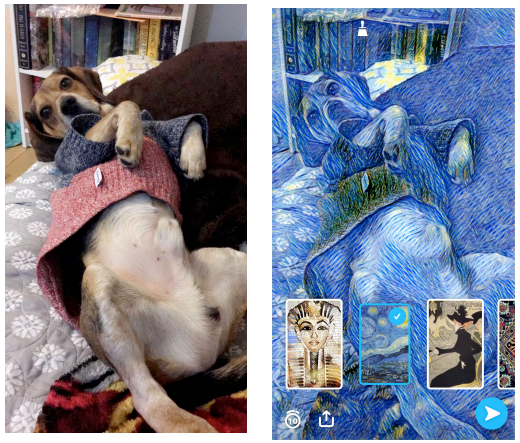
"Image style transfer using convolutional neural networks," L. Gatys, *CVPR* 2015

305



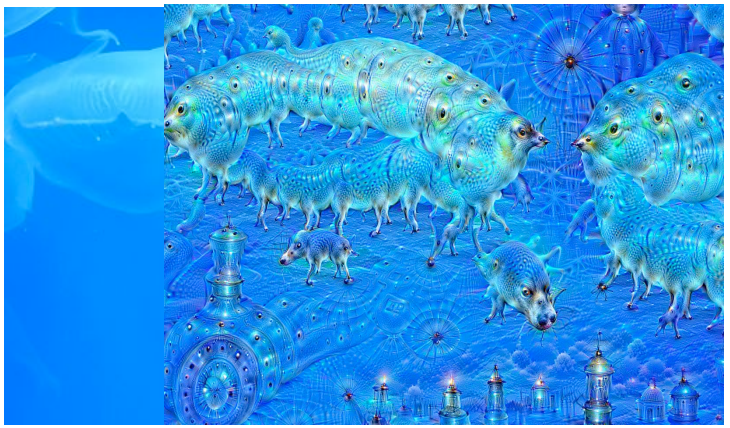
306

Snapchat's Paintbrush



307

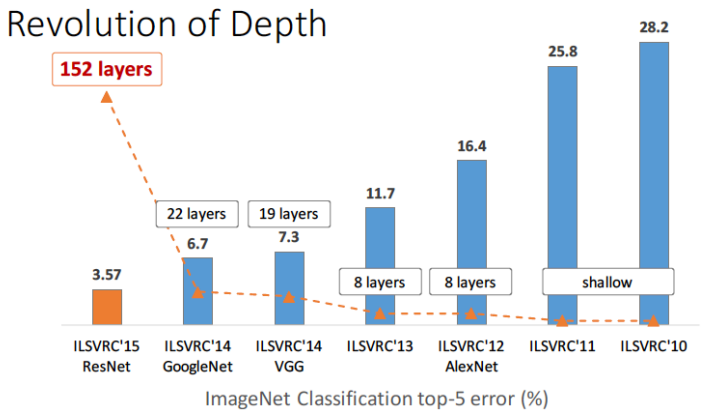
Google's DeepDream



Input image Output image

308

Current Trend: *Very Deep* Networks (using Deep Residual Learning)



309

Deep Learning Software

- TensorFlow (Google)
- Caffe (UC Berkeley)
- Torch
- Deeplearning4j
- OpenNN
- Microsoft Cognitive Toolkit
- ConvNetJS
- etc.

311