CSC 2515: Lecture 05: Neural Networks

Richard Zemel & Raquel Urtasun

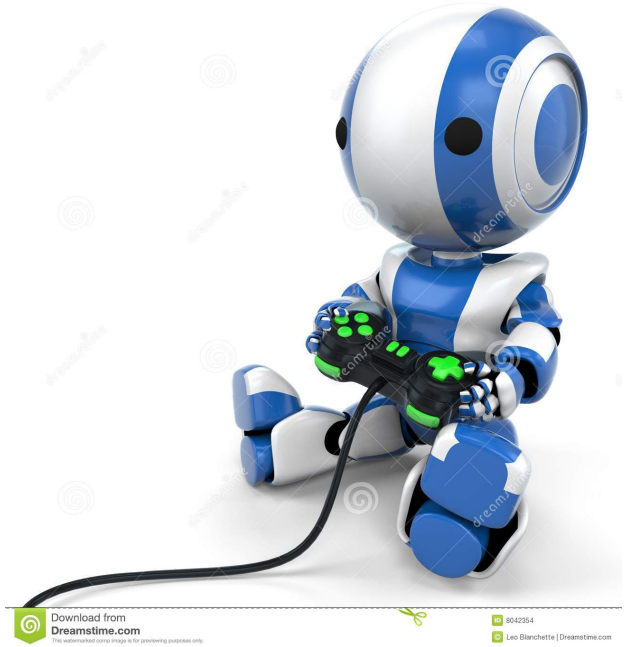
University of Toronto

Oct 15, 2015

Today (Part 1)

- Forward propagation
- Backward propagation
- Deep learning

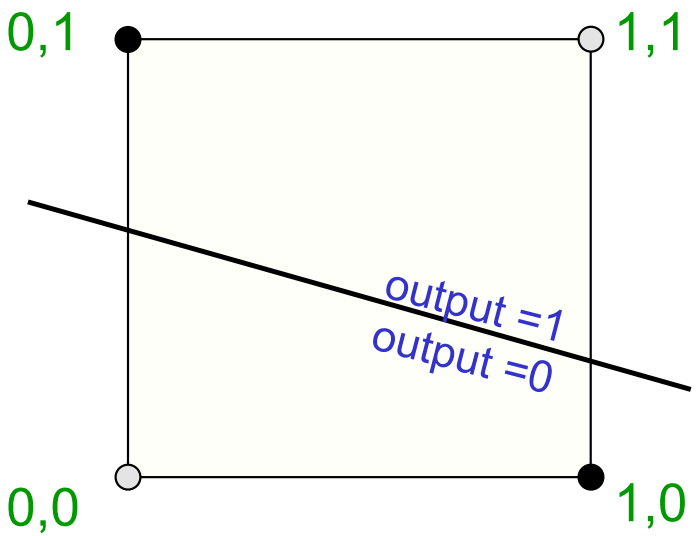
Motivating Examples



Are you excited about deep learning?

Limitations of linear classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features x_i
- Many decisions involve non-linear functions of the input
- Canonical example: do 2 input elements have the same value?



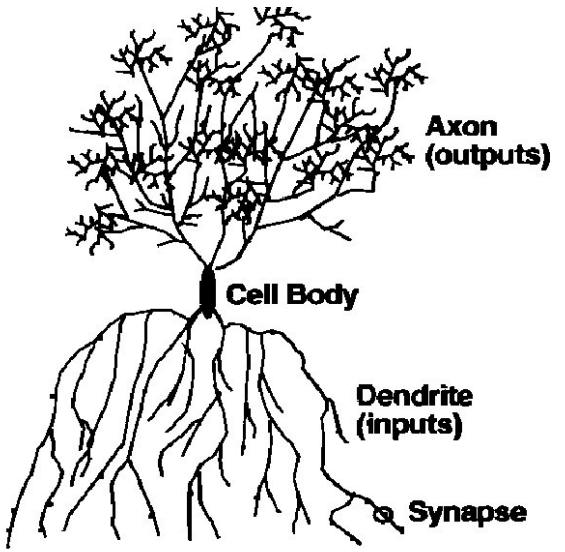
- The positive and negative cases cannot be separated by a plane
- What can we do?

How to construct nonlinear classifiers?

- Would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Add large number of extra functions
  - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs
  - Or we can make these functions depend on additional parameters $\rightarrow$ need an efficient method of training extra parameters

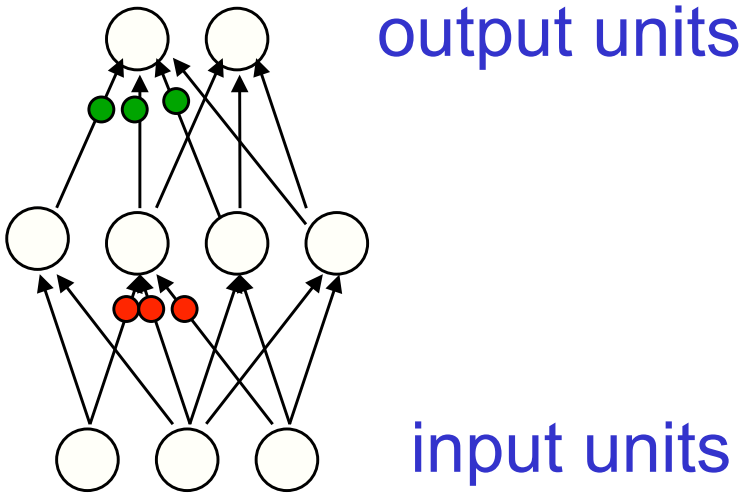
Neural Networks

- Many machine learning methods inspired by biology, brains
- Our brains contain $\sim 10^{11}$ neurons, each of which communicates to $\sim 10^4$ other neurons
- Multi-layer perceptron, or neural network, is popular supervised approach
- Defines extra functions of the inputs (hidden features), computed by neurons
- Artificial neurons called units
- Network output is linear combination of hidden units



Neural network architecture

- Network with one layer of four hidden units:

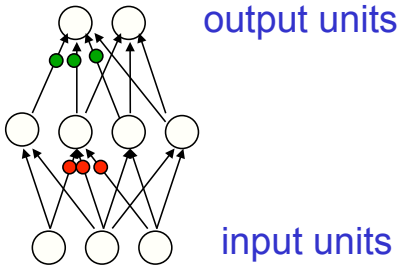


- Each unit computes its value based on linear combination of values of units that point into it
- Can add more layers of hidden units: deeper hidden unit response depends on earlier hiddens

Neural Networks

- We only need to know two algorithms
  - Forward pass: performs inference
  - Backward pass: performs learning

What does the network compute?



- Output of network can be written as (with k indexing the two output units):

$$h_j(\mathbf{x}) = f(w_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

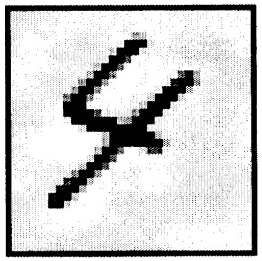
$$o_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})$$

- Network with non-linear activation function $f()$ is a universal approximator (esp. with increasing J)
- Standard f: sigmoid/logistic, or tanh, or rectified linear (relu)

$$tanh(z) = \frac{\exp(z) - exp(-z)}{exp(z) + exp(-z)} \qquad relu(z) = \max(0, z)$$

Example application

- Consider trying to classify image of handwritten digit: 32x32 pixels



- Single output unit – is it a 4 (one vs. all)?
- Use the sigmoid output function:

$$o_k(\mathbf{x}) = \frac{1}{1+\exp(-z_k)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

- What do I recover if $h_j(\mathbf{x}) = x_j$?
- How can we train the network, that is, adjust all the parameters $\mathbf{w}$?
- If we have trained the network, how can we do inference?

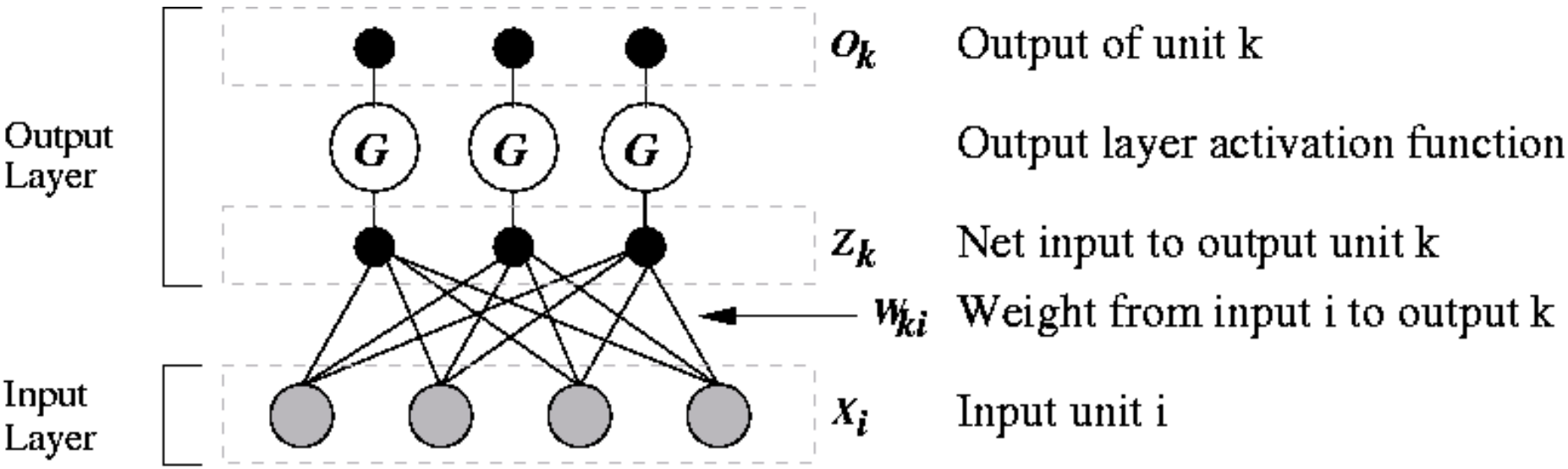
Training multi-layer networks: back-propagation

- Use gradient descent to learn the weights
- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network
- Loop until convergence:
  - for each example n
    1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$)
    2. Propagate gradients backward
    3. Update each weight (via gradient descent)
- Given any error function E, activation functions $g()$ and $f()$, just need to derive gradients

Key idea behind backpropagation

- We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit
- This is just the chain rule!

Computing gradient: single layer network



- Error gradients for single layer network:

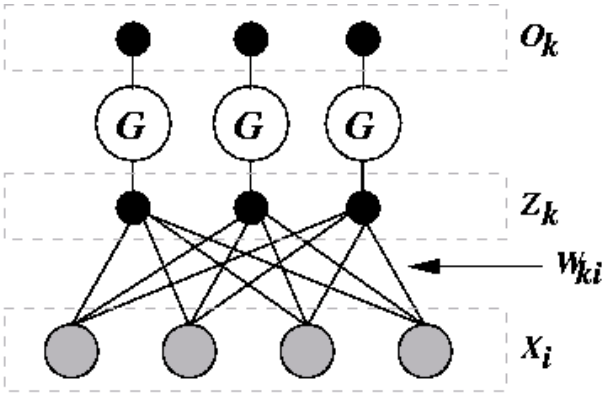
$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

- Error gradient is computable for any continuous activation function $g()$, and any continuous error function

Gradient descent for single layer network

- Assuming the error function is mean-squared error (MSE), on a single training example n, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)}$$



Using logistic activations

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1 + \exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1 - o_k^{(n)}) \end{aligned}$$

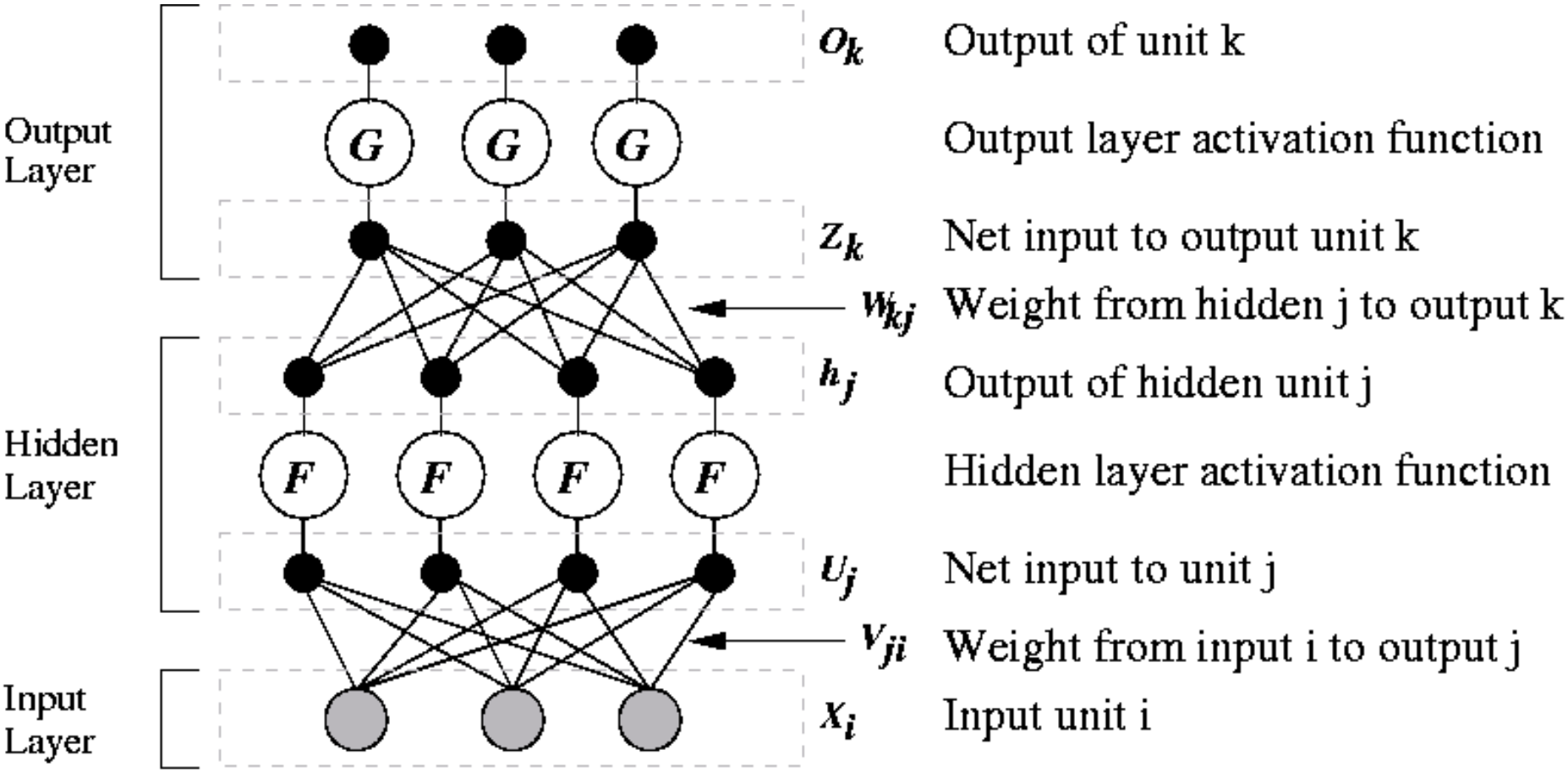
- The error gradient is then:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{ki}} = \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

- The gradient descent update rule is given by:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

Multi-layer neural network

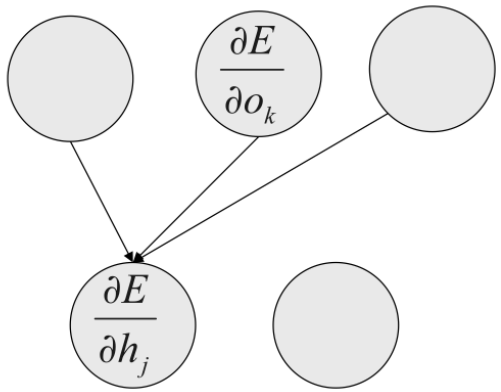


Back-propagation: sketch on one training case

- Convert discrepancy between each output and its target value into an error derivative

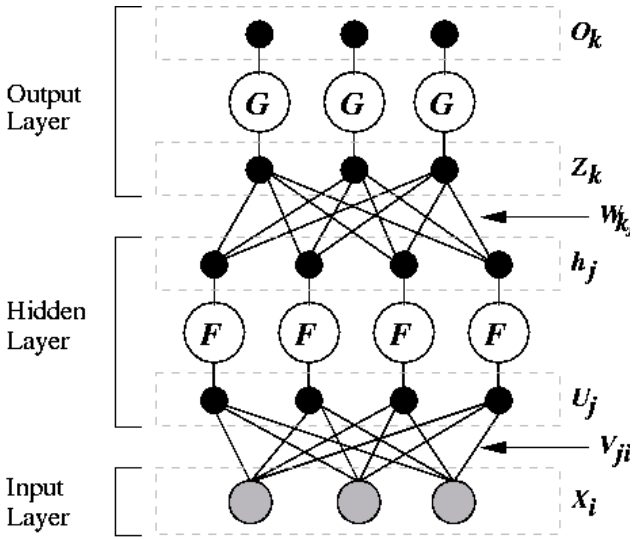
$$E = \frac{1}{2}\sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

- Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at k to each unit j according to its influence on k (depends on w_{kj})]



- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

Gradient descent for multi-layer network



- The output weight gradients for a multi-layer network are the same as for a single layer network

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

- Hidden weight gradients are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{(n)} w_{kj}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \left(\sum_k \delta_k^{(n)} w_{kj} \right) f'(u_j^{(n)}) x_i^{(n)} = \sum_{n=1}^{N} \bar{\delta}_j^{(n)} x_i^n$$

Choosing activation and cost functions

- When using a neural network as a function approximator (regressor) sigmoid activation and MSE as loss function work well
- For classification, if it is a binary (2-class) problem, then cross-entropy error function often does better (as we saw with logistic regression)

$$E = -\sum_{n=1}^{N} t^{(n)} \log o^{(n)} + (1 - t^{(n)}) \log(1 - o^{(n)})$$

$$o^{(n)} = (1 + \exp(-z^{(n)})^{-1}$$

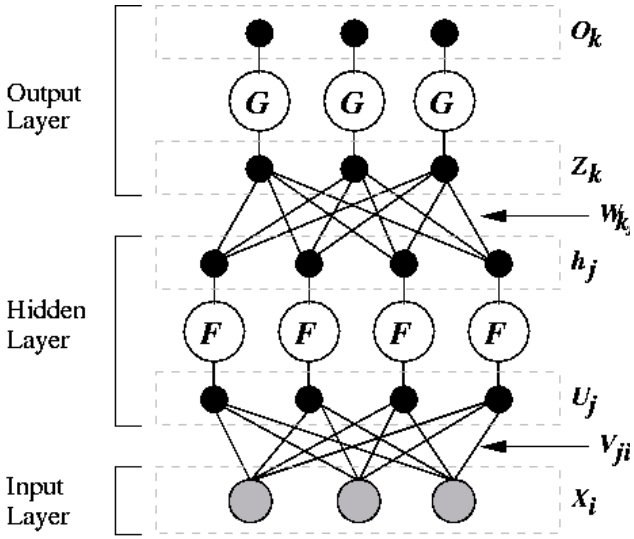
- We can then compute via the chain rule

$$\frac{\partial E}{\partial o} = (o - t)/(o(1 - o))$$

$$\frac{\partial o}{\partial z} = o(1 - o)$$

$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z} = (o - t)$$

Multi-class classification



- For multi-class classification problems, use the softmax activation

$$E = -\sum_n \sum_k t_k^{(n)} \log o_k^{(n)}$$

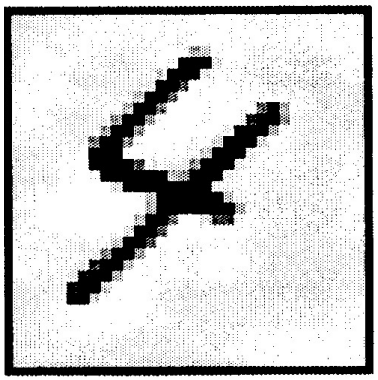
$$o_k^{(n)} = \frac{\exp(z_k^{(n)})}{\sum_j \exp(z_j^{(n)})}$$

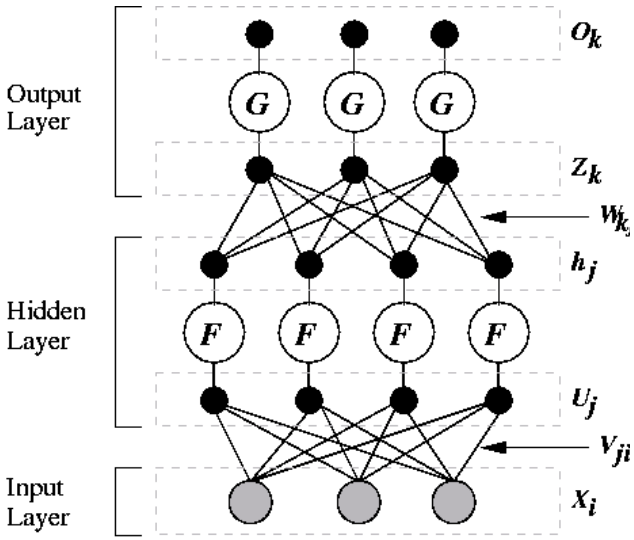
- And the derivatives become

$$\frac{\partial o_k}{\partial z_k} = o_k(1 - o_k)$$

$$\frac{\partial E}{\partial z_k} = \sum_j \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial z_k} = (o_k - t_k) o_k (1 - o_k)$$

Example Application





- Now trying to classify image of handwritten digit: 32x32 pixels
- 10 output units, 1 per digit
- Use the softmax function:

$$o_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

- What is J ?

Ways to use weight derivatives

- How often to update
  - after a full sweep through the training data

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

  - after each training case
  - after a mini-batch of training cases
- How much to update
  - Use a fixed learning rate
  - Adapt the learning rate
  - Add momentum

$$\begin{aligned} w_{ki} &\leftarrow w_{ki} - v \\ v &\leftarrow \gamma v + \eta \frac{\partial E}{\partial w_{ki}} \end{aligned}$$

Deep Neural Networks

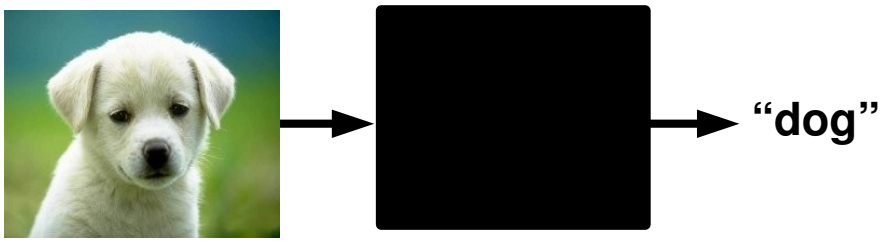
- We only need to know two algorithms
  - Forward pass: performs inference
  - Backward pass: performs learning

- Neural nets are now called deep learning. Why?

Why "Deep"?

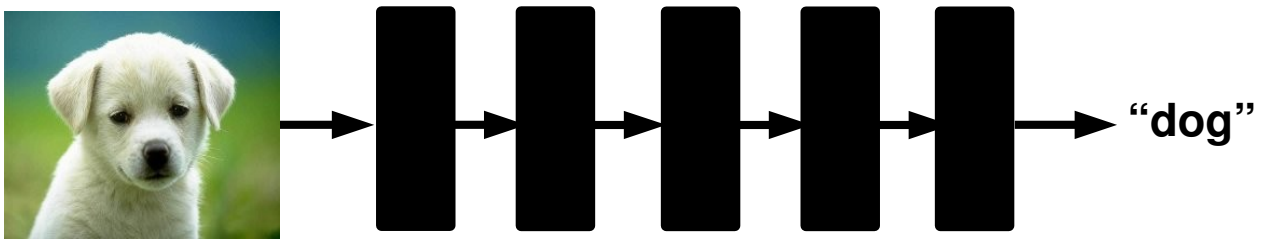
Supervised Learning: Examples

Classification



Supervised Deep Learning

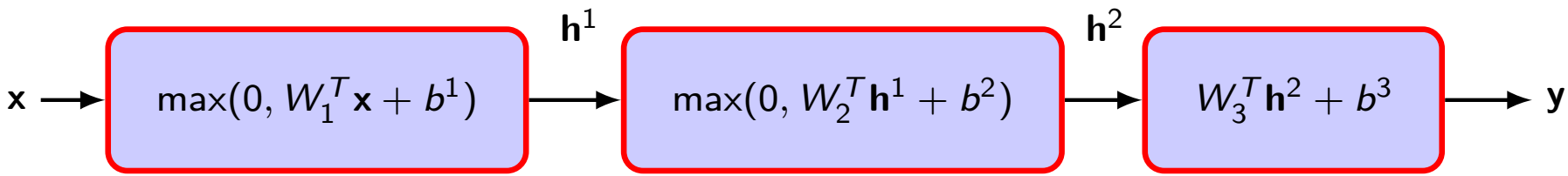
Classification



[Picture from M. Ranzato]

Neural Networks

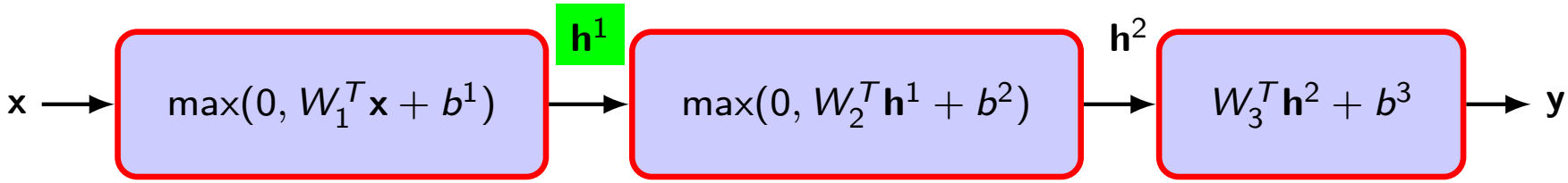
- Deep learning uses a composite of simple functions (e.g., ReLU, sigmoid, tanh, max) to create complex non-linear functions
- Note: a composite of linear functions is linear!
- Example: 2 layer NNet (now matrix and vector form!)



  - $\mathbf{x}$ is the input
  - $\mathbf{y}$ is the output (what we want to predict)
  - $\mathbf{h}^i$ is the i-th hidden layer
  - W^i are the parameters of the i-th layer

Evaluating the Function

- Assume we have learned the weights and we want to do inference
- Forward Propagation: compute the output given the input

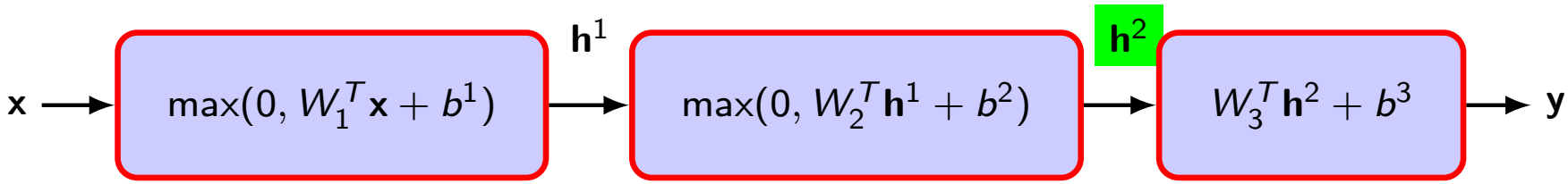


- Fully connected layer: Each hidden unit takes as input all the units from the previous layer
- Here the non-linearity is called a ReLU (rectified linear unit), with $\mathbf{x} \in \Re^D$, $b^i \in \Re^{N_i}$ the biases and $W^i \in \Re^{N_i \times N_{i-1}}$ the weights
- Do it in a compositional way,

$$\mathbf{h}^1 = \max(0, W^1 \mathbf{x} + b^1)$$

Evaluating the Function

- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input

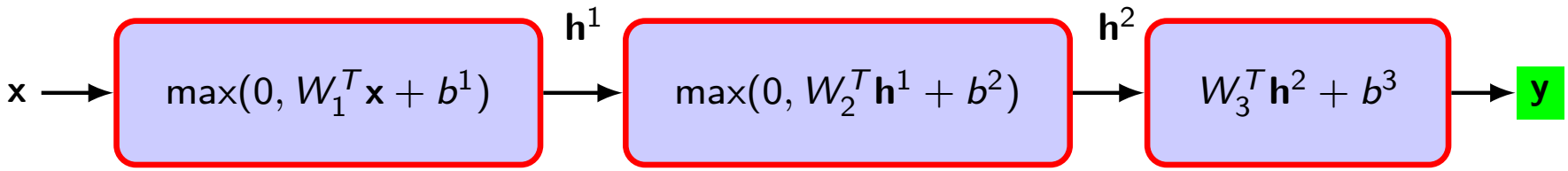


- Fully connected layer: Each hidden unit takes as input all the units from the previous layer
- The non-linearity is called a ReLU (rectified linear unit), with $\mathbf{x} \in \Re^D$, $b^i \in \Re^{N_i}$ the biases and $W^i \in \Re^{N_i \times N_{i-1}}$ the weights
- Do it in a compositional way

$$\begin{aligned} \mathbf{h}^1 &= \max(0, W^1 \mathbf{x} + b^1) \\ \mathbf{h}^2 &= \max(0, W^2 \mathbf{h}^1 + b^2) \end{aligned}$$

Evaluating the Function

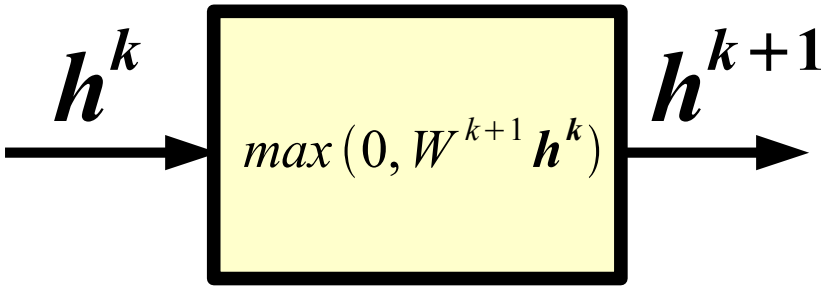
- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input

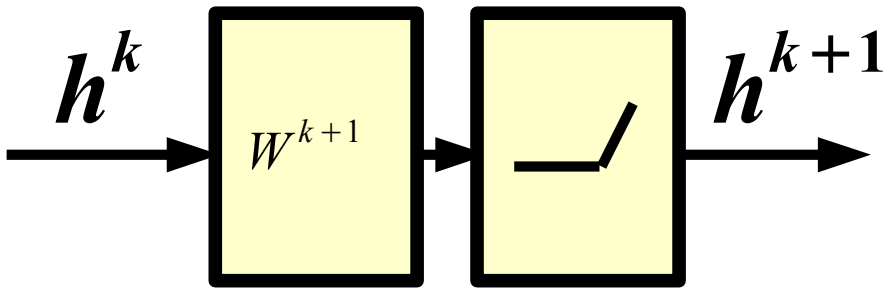


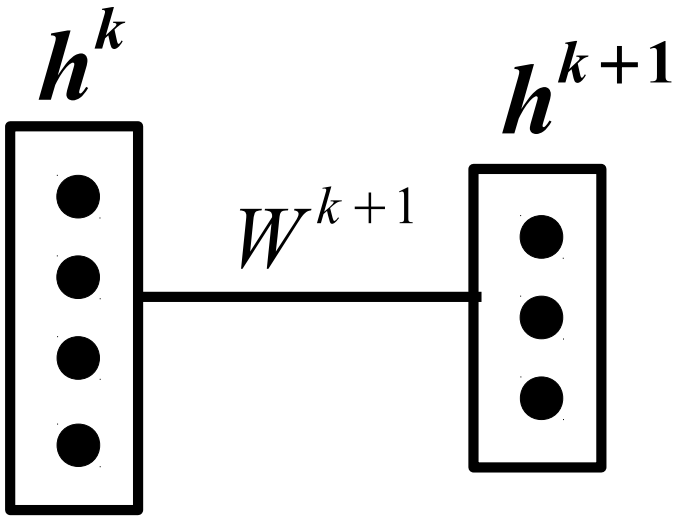
- Fully connected layer: Each hidden unit takes as input all the units from the previous layer
- The non-linearity is called a ReLU (rectified linear unit), with $\mathbf{x} \in \Re^D$, $b^i \in \Re^{N_i}$ the biases and $W^i \in \Re^{N_i \times N_{i-1}}$ the weights
- Do it in a compositional way

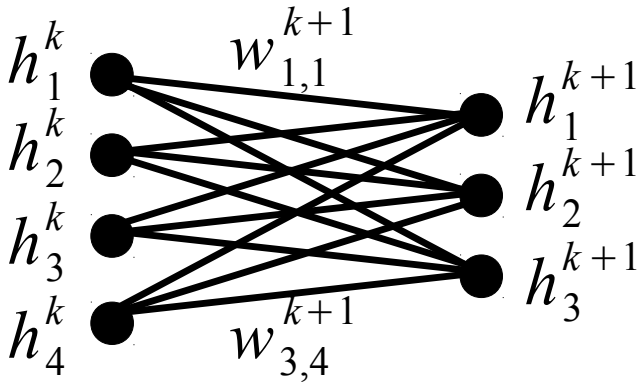
$$\begin{aligned} \mathbf{h}^1 &= \max(0, W^1\mathbf{x} + b^1) \\ \mathbf{h}^2 &= \max(0, W^2\mathbf{h}^1 + b^2) \\ \mathbf{y} &= \max(0, W^3\mathbf{h}^2 + b^3) \end{aligned}$$

Alternative Graphical Representation



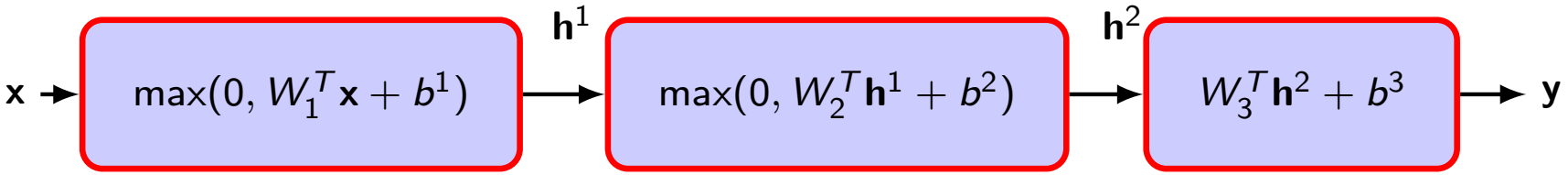






Ranzato

Learning



- We want to estimate the parameters, biases and hyper-parameters (e.g., number of layers, number of units) such that we do good predictions
- Collect a training set of input-output pairs $\{\mathbf{x}, t\}$
- Encode the output with 1-K (one-hot) encoding $t = [0, \cdots, 1, \cdots, 0]$
- Define a loss per training example and minimize the empirical risk

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)})$$

with N number of examples and $\mathbf{w}$ contains all parameters

Loss Functions

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)})$$

- Probability of class k given input (softmax):

$$p(c_k = 1|\mathbf{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)}$$

- Cross entropy is the most common loss function for classification

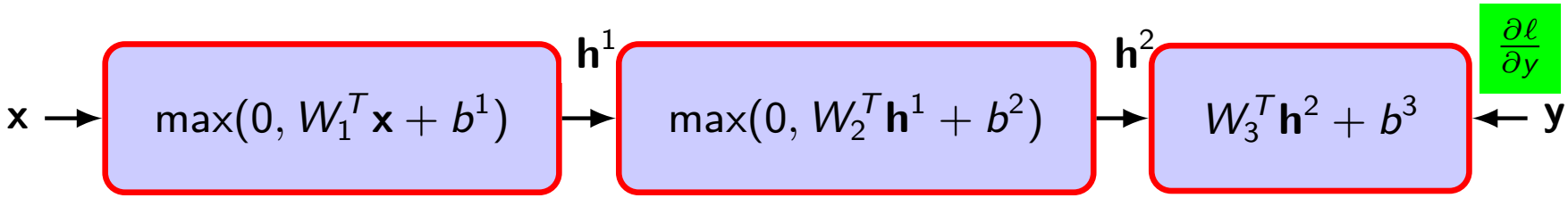
$$\ell(\mathbf{x}, t, \mathbf{w}) = -\sum_i t^{(i)} \log p(c_i|\mathbf{x})$$

- Use gradient descent to train the network

$$\min_{\mathbf{w}} \frac{1}{N} \sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)})$$

Backpropagation

- Efficient computation of the gradients by applying the chain rule



$$p(c_k = 1|\mathbf{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)}$$

$$\ell(\mathbf{x}, t, \mathbf{w}) = -\sum_i t^{(i)} \log p(c_i|\mathbf{x})$$

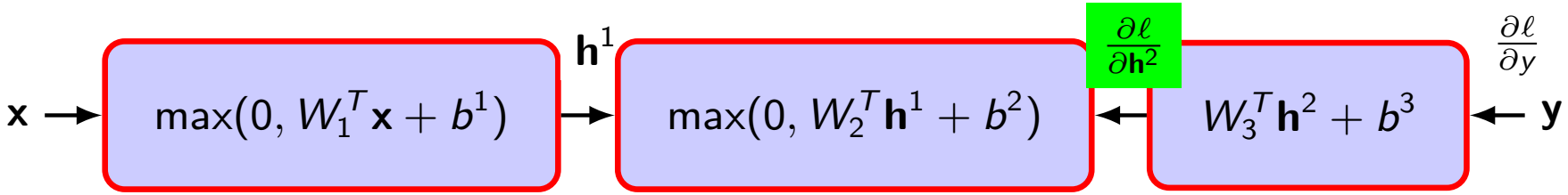
- Compute the derivative of loss w.r.t. the output

$$\frac{\partial \ell}{\partial y} = p(c|\mathbf{x}) - t$$

- Note that the forward pass is necessary to compute $\frac{\partial \ell}{\partial y}$

Backpropagation

- Efficient computation of the gradients by applying the chain rule



- We have computed the derivative of loss w.r.t the output

$$\frac{\partial \ell}{\partial y} = p(c|\mathbf{x}) - t$$

- Given $\frac{\partial \ell}{\partial y}$ if we can compute the Jacobian of each module

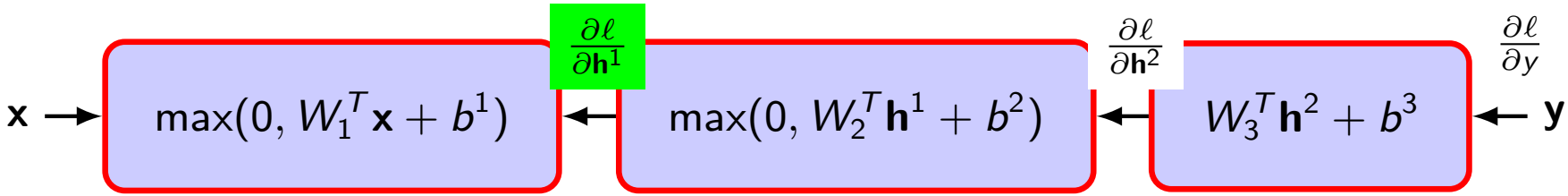
$$\frac{\partial \ell}{\partial W^3} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial W^3} = (p(c|\mathbf{x}) - t)(\mathbf{h}^2)^T$$

$$\frac{\partial \ell}{\partial \mathbf{h}^2} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial \mathbf{h}^2} = (W^3)^T(p(c|\mathbf{x}) - t)$$

- Need to compute gradient w.r.t. inputs and parameters in each layer

Backpropagation

- Efficient computation of the gradients by applying the chain rule



$$\frac{\partial \ell}{\partial \mathbf{h}^2} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial \mathbf{h}^2} = (W^3)^T (p(c|\mathbf{x}) - t)$$

- Given $\frac{\partial \ell}{\partial \mathbf{h}^2}$ if we can compute the Jacobian of each module

$$\frac{\partial \ell}{\partial W^2} = \frac{\partial \ell}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial W^2}$$

$$\frac{\partial \ell}{\partial \mathbf{h}^1} = \frac{\partial \ell}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial \mathbf{h}^1}$$

Toy Code (Matlab): Neural Net Trainer

```
% F-PROP
for i = 1 : nr_layers - 1
  [h{i}  jac{i}]  =  nonlinearity(W{i} * h{i-1} +  b{i});
end
h{nr_layers-1}  =  W{nr_layers-1} * h{nr_layers-2}  +  b{nr_layers-1};
prediction  =  softmax(h{l-1});


% CROSS ENTROPY LOSS
loss  =  -  sum(sum(log(prediction)  .*  target)) / batch_size;


% B-PROP
dh{l-1}  =  prediction  -  target;
for i = nr_layers - 1 : -1 : 1
  Wgrad{i}  =  dh{i} * h{i-1}';
  bgrad{i}  =  sum(dh{i}, 2);
  dh{i-1}  =  (W{i}' * dh{i})  .*  jac{i-1};
end


% UPDATE
for i = 1 : nr_layers - 1
  W{i}  =  W{i}  -  (lr / batch_size)  *  Wgrad{i};
  b{i}  =  b{i}  -  (lr / batch_size)  *  bgrad{i};
end
```


Ranzato

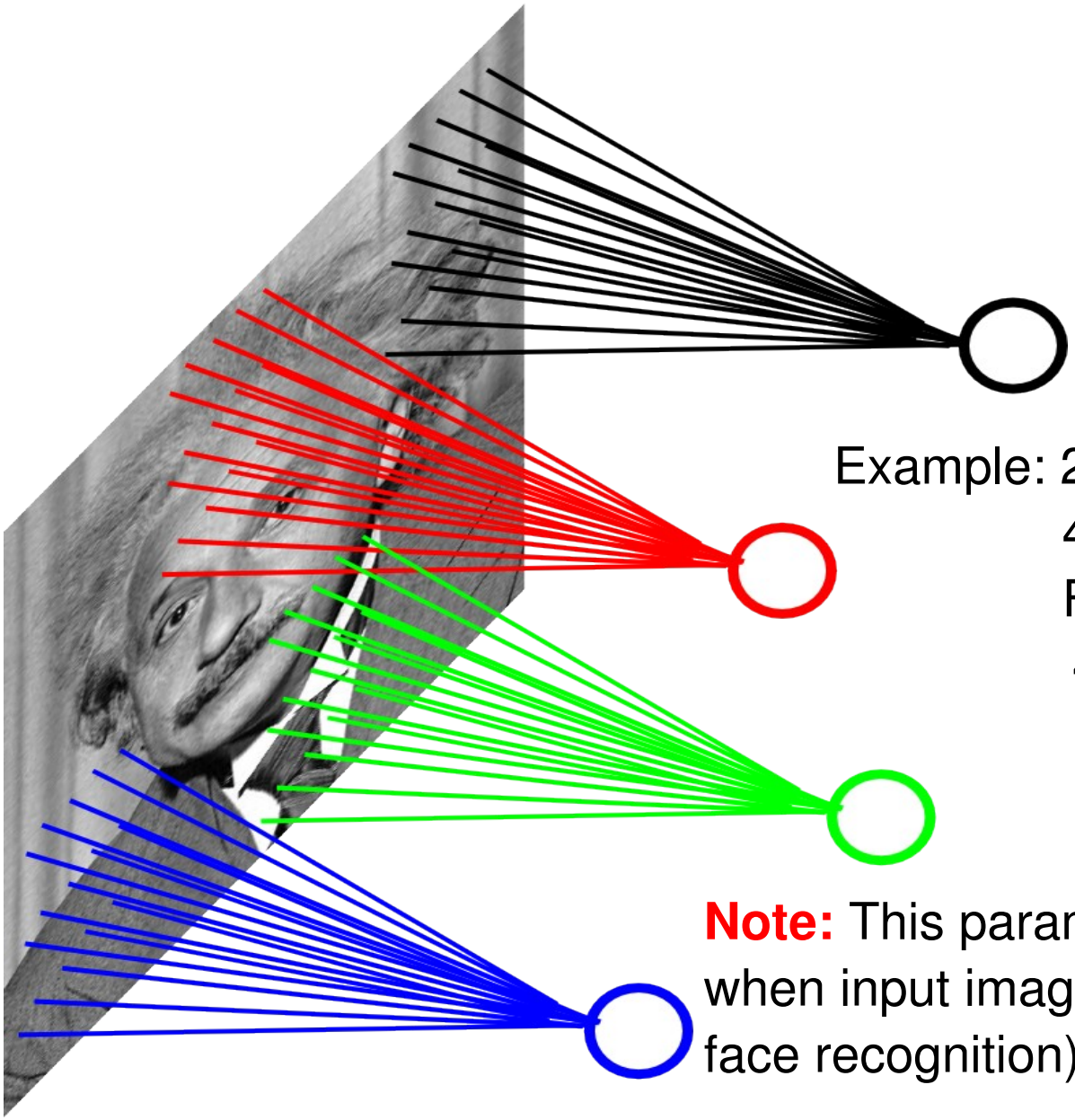
Neural Nets for Object Recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Some reasons why it is difficult:
  - Segmentation: Real scenes are cluttered
  - Invariances: We are very good at ignoring all sorts of variations that do not affect shape
  - Deformations: Natural shape classes allow variations (faces, letters, chairs)
  - A huge amount of computation is required

How to deal with large Input Spaces

- Images can have millions of pixels, i.e., $\mathbf{x}$ is very high dimensional
- Prohibitive to have fully-connected layer
- We can use a locally connected layer
- This is good when the input is registered

Locally Connected Layer



Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

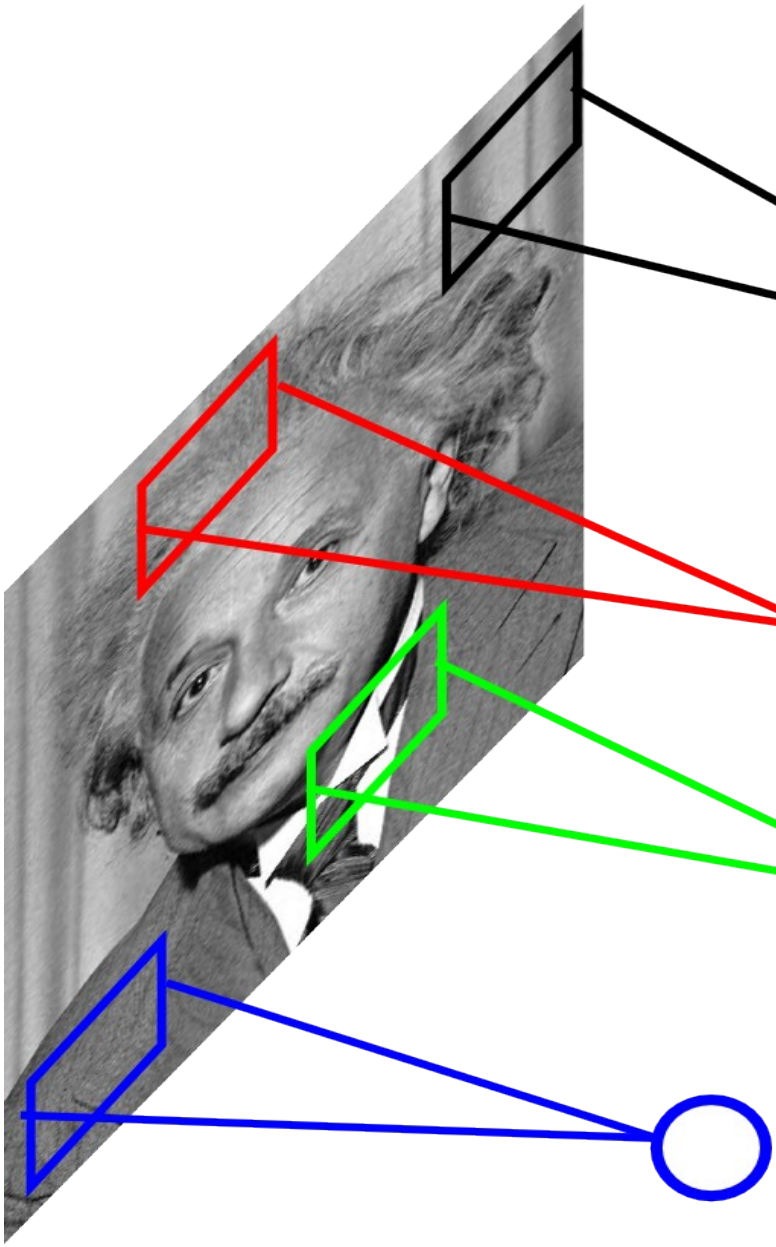
34

Ranzato

The invariance problem

- Our perceptual systems are very good at dealing with invariances
  - translation, rotation, scaling
  - deformation, contrast, lighting, rate
- We are so good at this that its hard to appreciate how difficult it is
  - Its one of the main difficulties in making computers perceive
  - We still don't have generally accepted solutions

Locally Connected Layer



STATIONARITY? Statistics is similar at different locations

Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

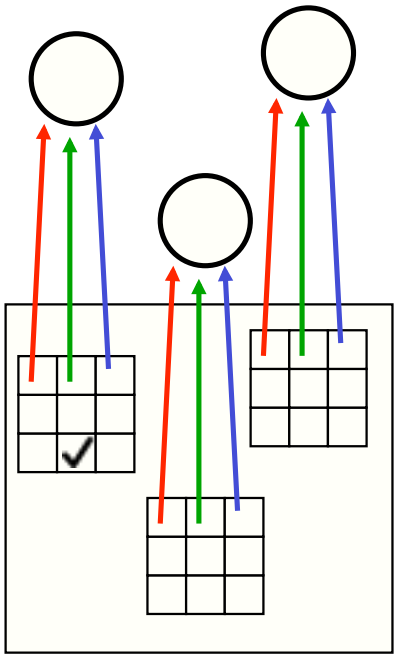
Note: This parameterization is good when input image is registered (e.g., face recognition).

35

Ranzato

The replicated feature approach

The red connections all have the same weight.

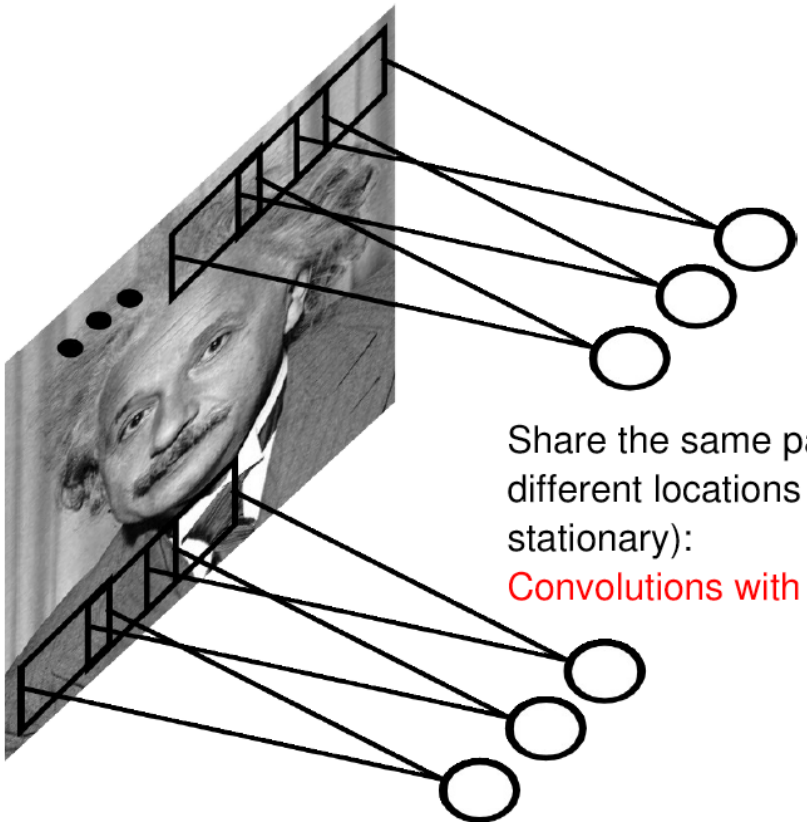


5

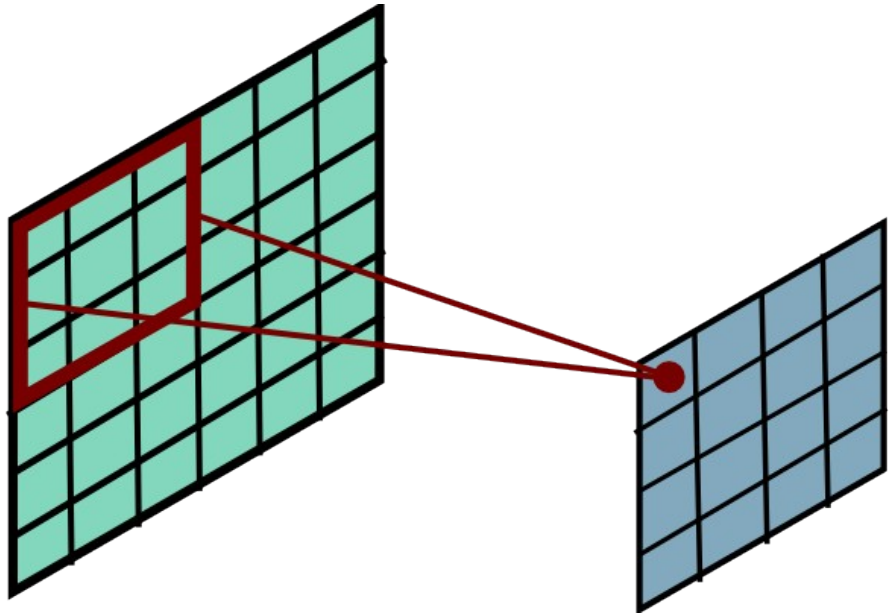
- Adopt approach apparently used in monkey visual systems
- Use many different copies of the same feature detector.
  - Copies have slightly different positions.
  - Could also replicate across scale and orientation.
    - Tricky and expensive
  - Replication reduces number of free parameters to be learned.
- Use several different feature types, each with its own replicated pool of detectors.
  - Allows each patch of image to be represented in several ways.

Convolutional Neural Net

- Idea: statistics are similar at different locations (Lecun 1998)
- Connect each hidden unit to a small input patch and share the weight across space
- This is called a convolution layer and the network is a convolutional network



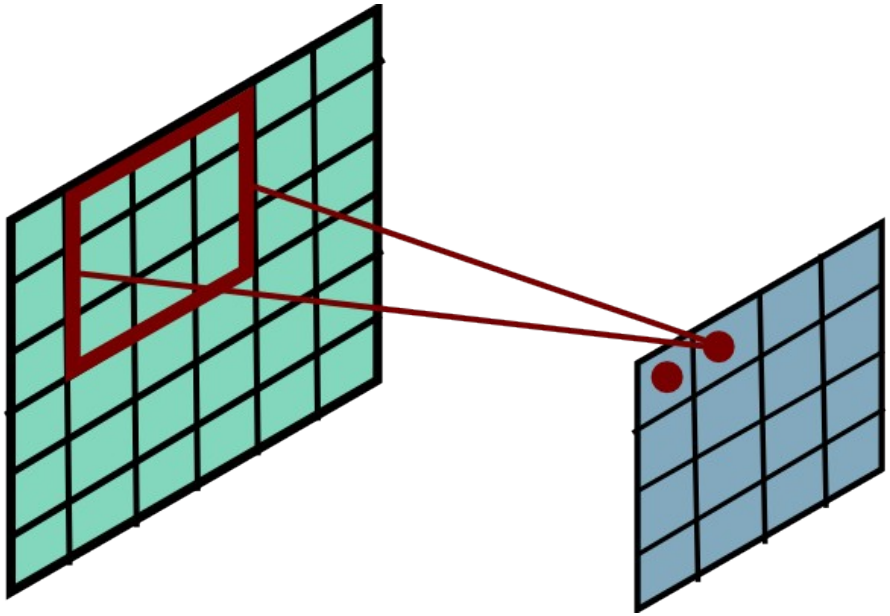
Convolutional Layer



Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

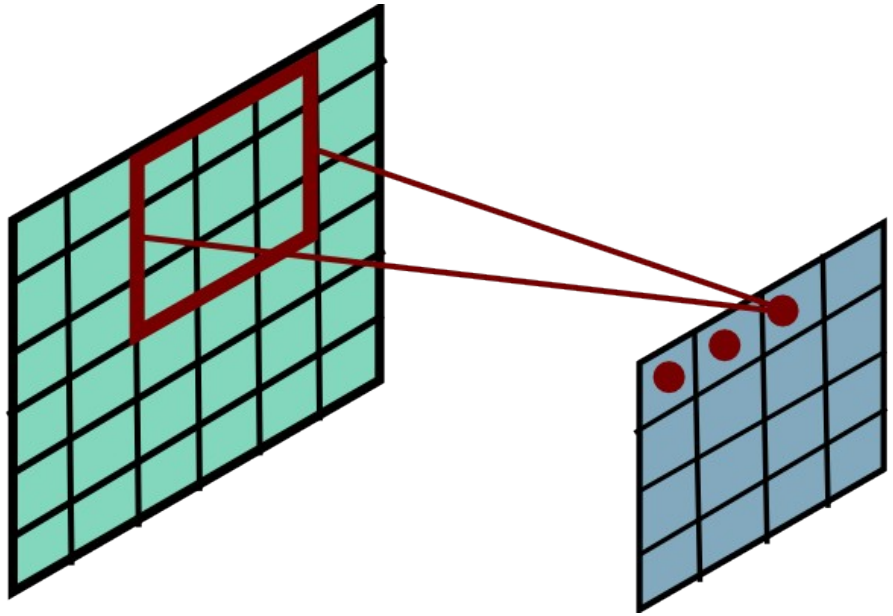
Convolutional Layer



Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

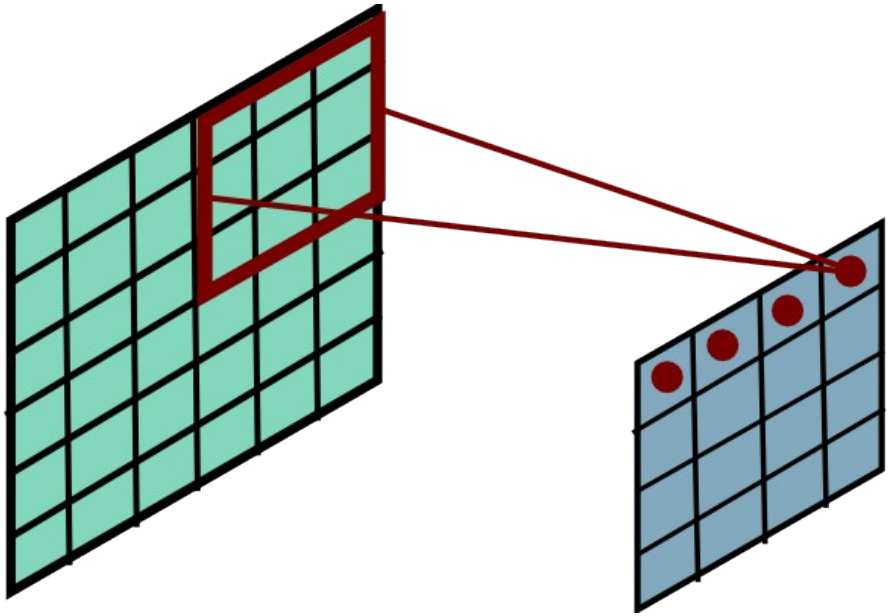
Convolutional Layer



Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

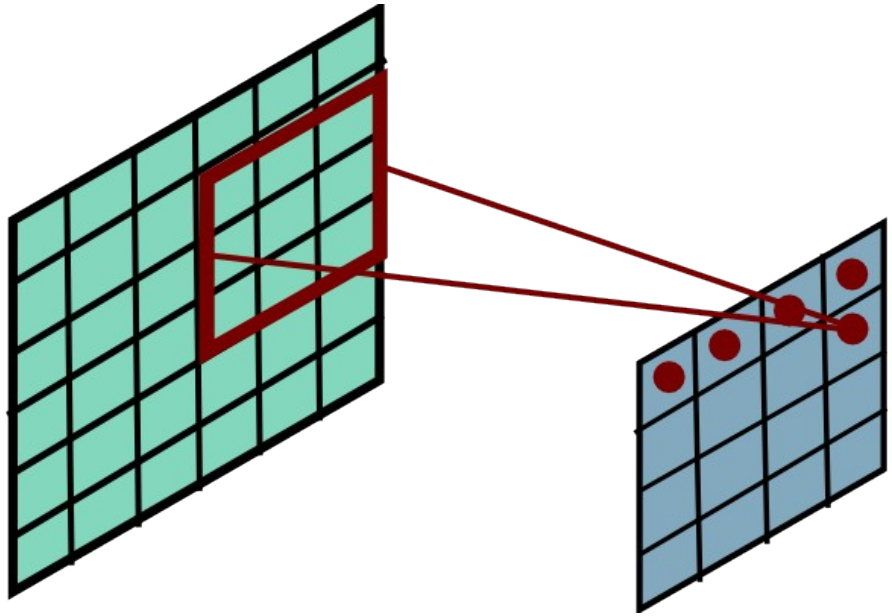
Convolutional Layer



Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

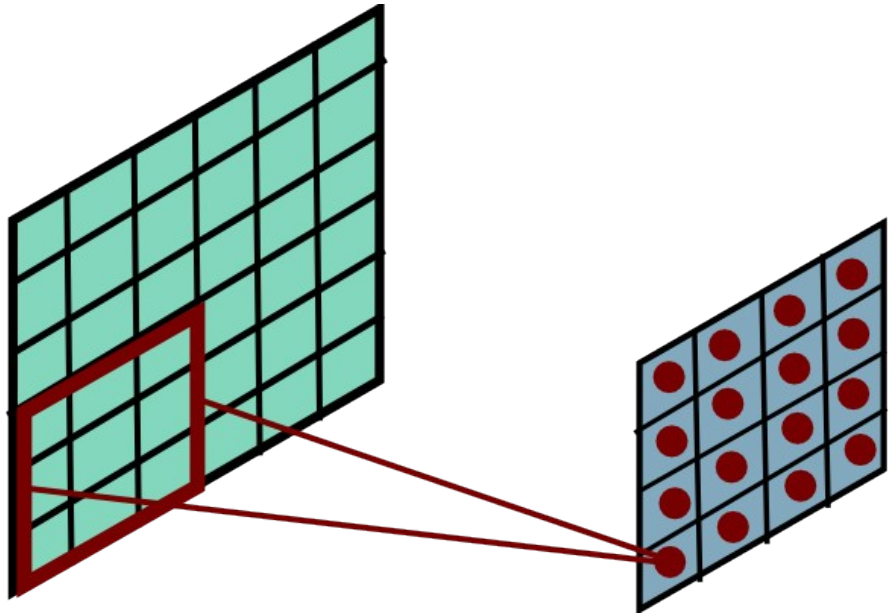
Convolutional Layer



Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Convolutional Layer



Ranzato

$$h_j^n = \max(0, \sum_{k=1}^{K} h_k^{n-1} * w_{jk}^n)$$

Backpropagation with weight constraints

- It is easy to modify the backpropagation algorithm to incorporate linear constraints between the weights

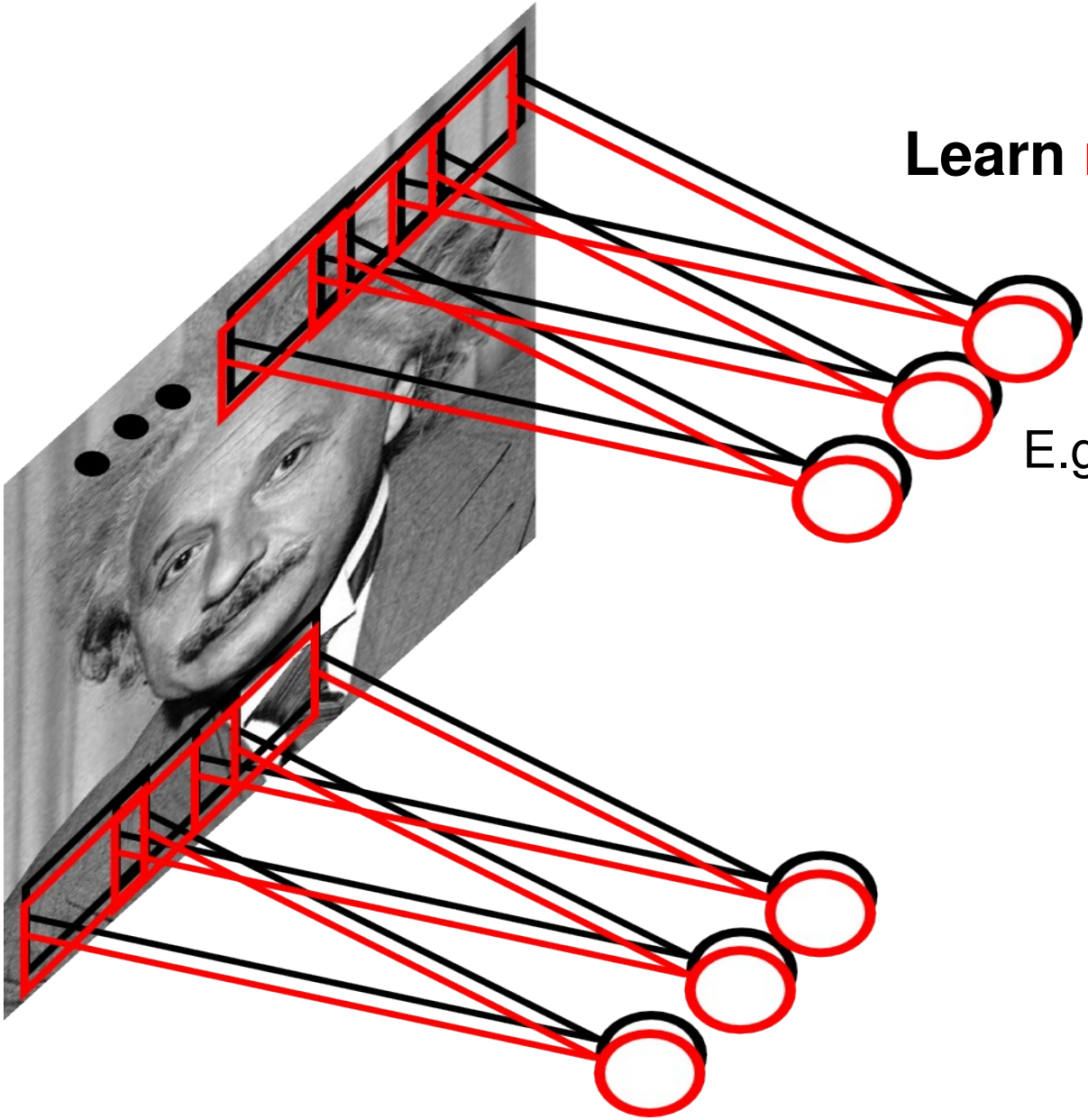
$$\begin{aligned} \text{To constrain:} \quad w_1 &= w_2 \\ \text{we need:} \quad \Delta w_1 &= \Delta w_2 \end{aligned}$$

- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.

$$\begin{aligned} \text{compute:} \quad & \frac{\partial E}{\partial w_1} \text{ and } \frac{\partial E}{\partial w_1} \\ \text{use:} \quad & \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2} \text{ for } w_1 \text{ and } w_2 \end{aligned}$$

- So if the weights started off satisfying the constraints, they will continue to satisfy them.

Convolutional Layer

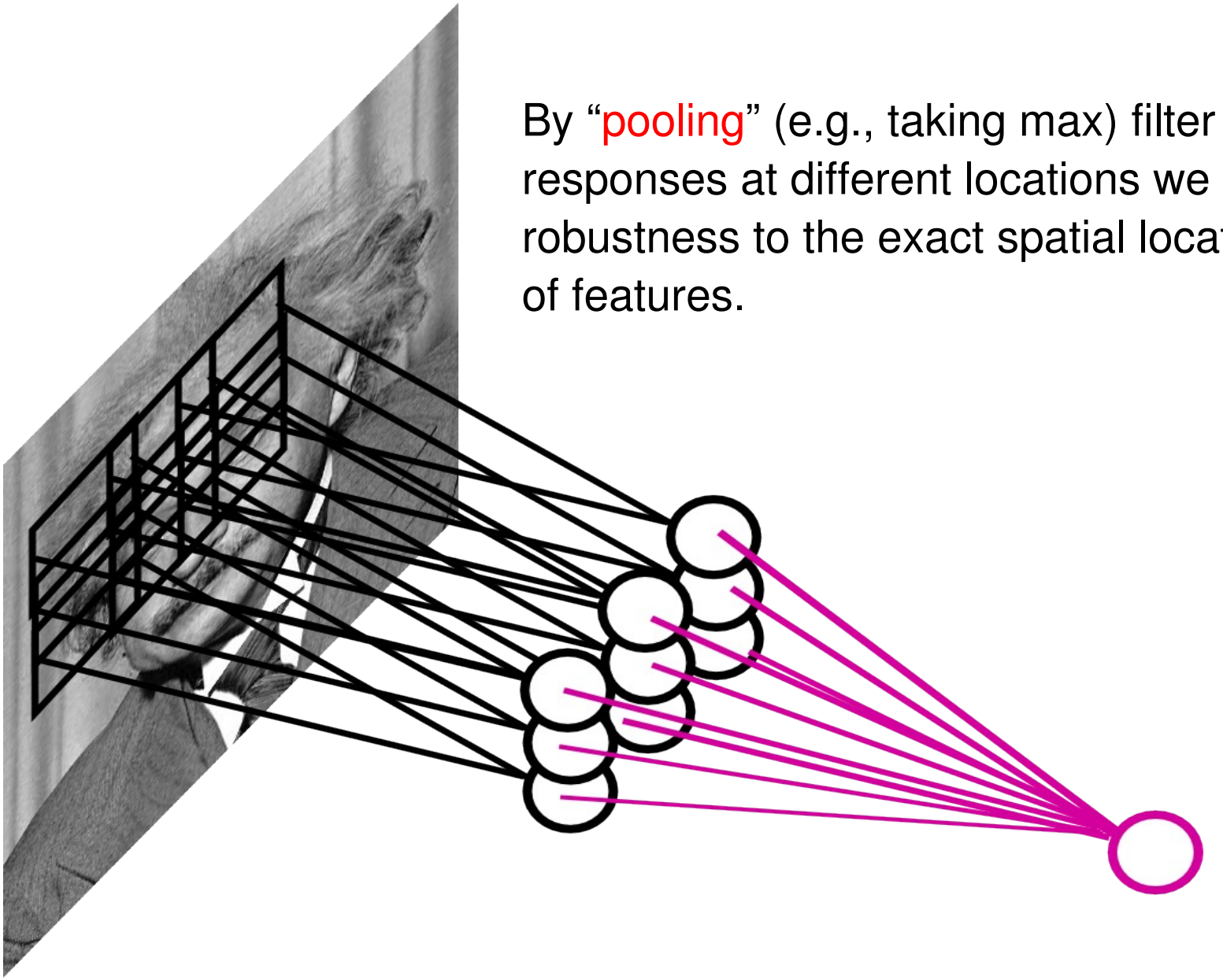


Learn multiple filters.

E.g.: 200x200 image
100 Filters
Filter size: 10x10
10K parameters

Ranzato

Pooling Layer



By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

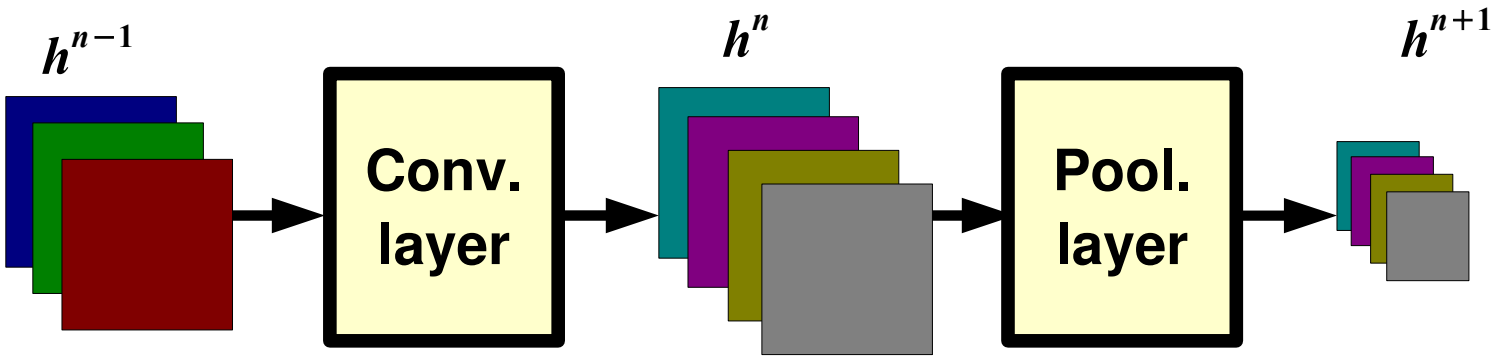
61

Ranzato

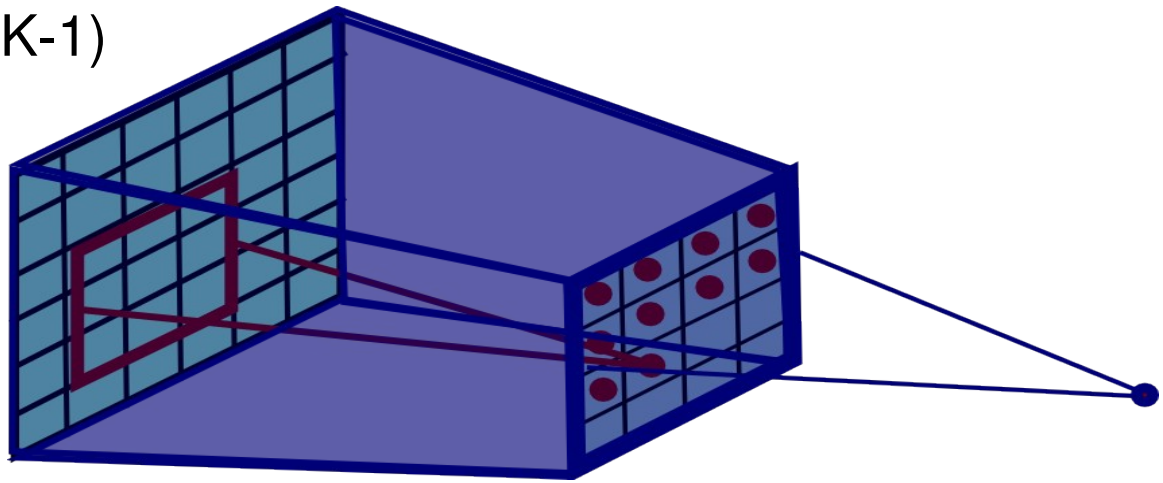
Pooling Options

- Max Pooling: return the maximal argument
- Average Pooling: return the average of the arguments
- Other types of pooling exist.

Pooling Layer: Receptive Field Size



If convolutional filters have size KxK and stride 1, and pooling layer has pools of size PxP, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: (P+K-1)x(P+K-1)



67

Ranzato

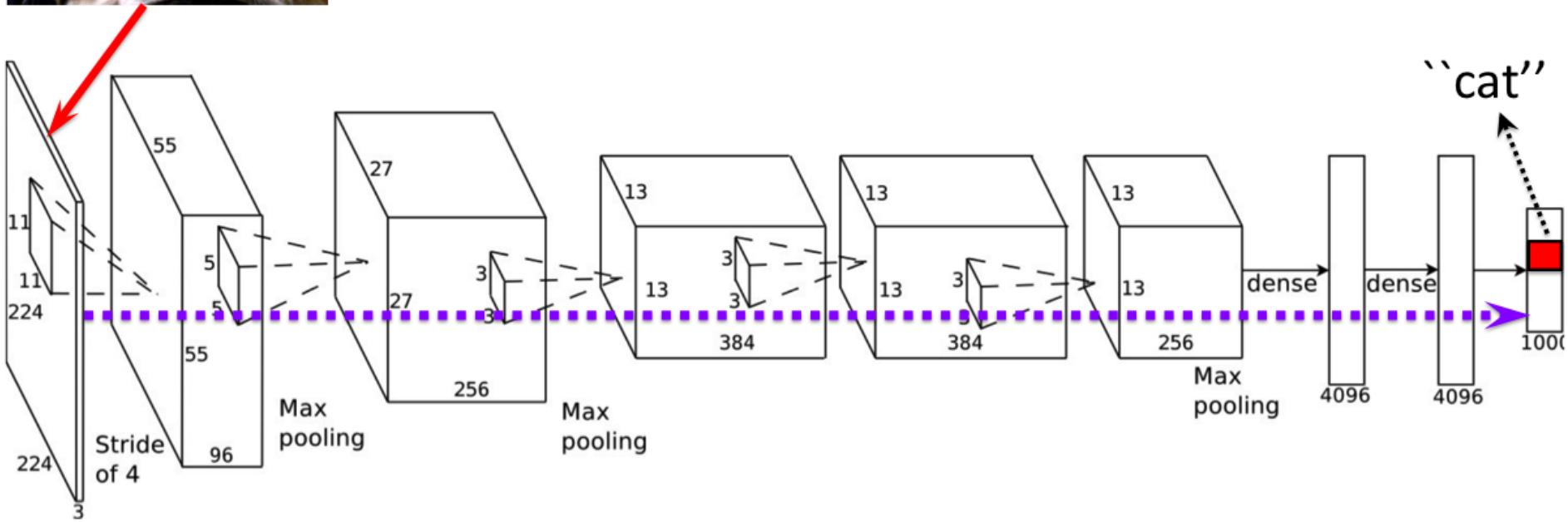
Now let's make this very deep to get a real state-of-the-art object recognition system

Convolutional Neural Networks (CNN)

- Remember from your image processing / computer vision course about filtering?
- If our filter was $[-1, 1]$, we got a vertical edge detector
- Now imagine we want to have many filters (e.g., vertical, horizontal, corners, one for dots). We will use a filterbank.
- So applying a filterbank to an image yields a cube-like output, a 3D matrix in which each slice is an output of convolution with one filter.
- Do some additional tricks. A popular one is called max pooling: to get invariance to small shifts in position.
- Now add another "layer" of filters. For each filter again do convolution, but this time with the output cube of the previous layer.
- Keep adding a few layers. Any idea what's the purpose of more layers? Why can't we just have a full bunch of filters in one layer?

Classification

- Once trained we feed in an image or a crop, run through the network, and read out the class with the highest probability in the last (classif) layer.

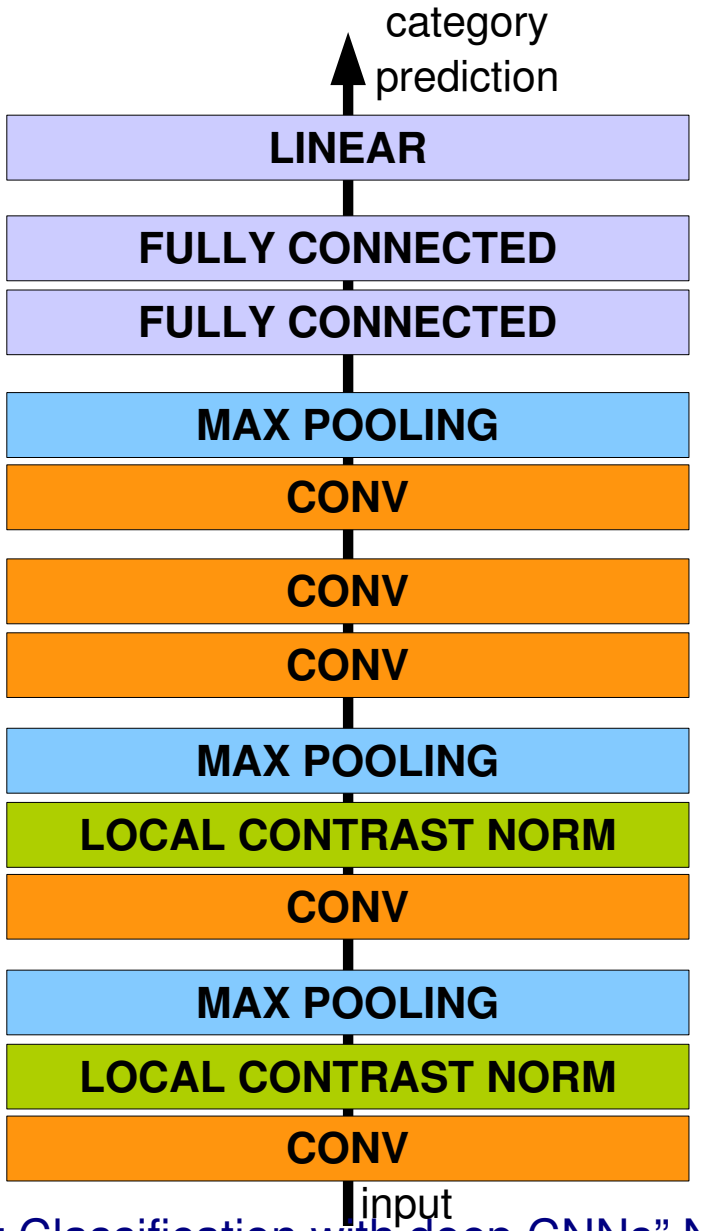


[Slide Credit: Sanja Fidler]

Classification Performance

- Imagenet, main challenge for object classification: `http://image-net.org/`
- 1000 classes, 1.2M training images, 150K for test

Architecture for Classification

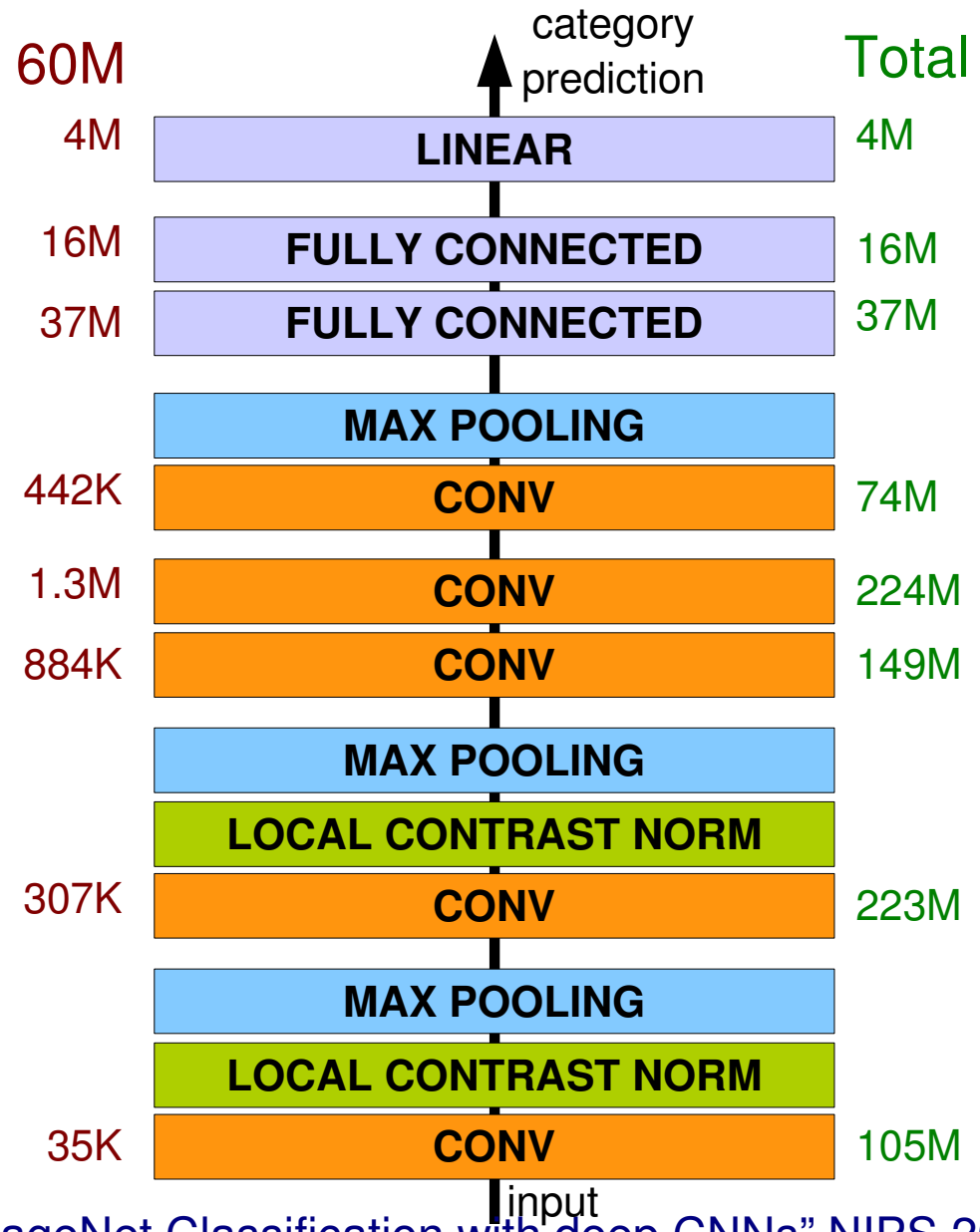


Krizhevsky et al. "ImageNet Classification with deep CNNs" NIPS 2012

Architecture for Classification

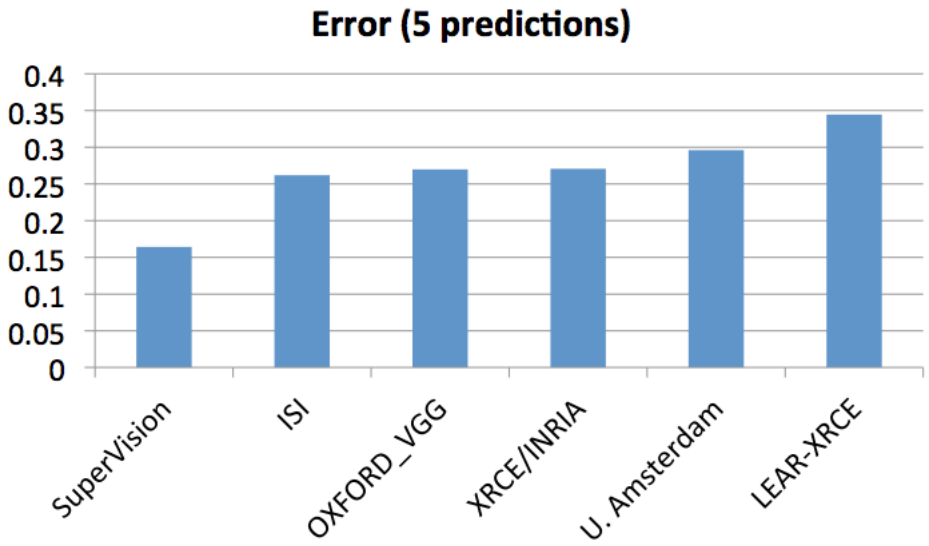
Total nr. params: 60M

Total nr. flops: 832M

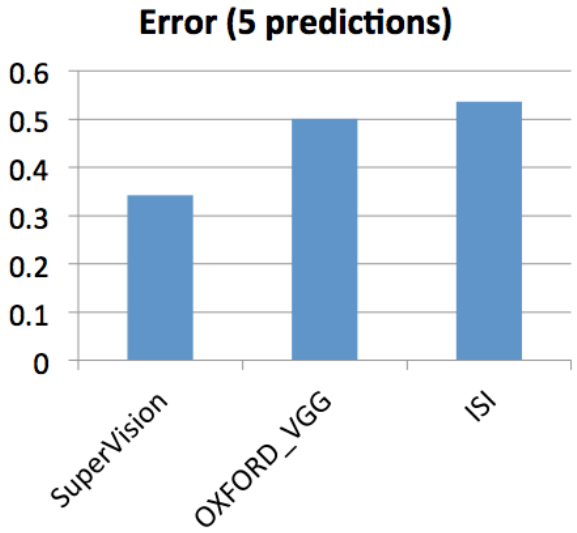


Krizhevsky et al. "ImageNet Classification with deep CNNs" NIPS 2012

The 2012 Computer Vision Crisis



(Classification)



(Detection)

So Neural Networks are Great

- So networks turn out to be great.
- Everything is deep, even if it's shallow!
- Companies leading the competitions as they have more computational power
- But to train the networks you need quite a bit of computational power (e.g., GPU farm). So what do you do?
- Buy even more. And train more layers. 16 instead of 7 before. 144 million parameters.

Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains noise
  - The target values may be unreliable.
  - There is sampling error. There will be accidental regularities just because of the particular training cases that were chosen
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity.
  - If the model is very flexible it can model the sampling error really well. This is a disaster.

Preventing overfitting

- Use a model that has the right capacity:
  - enough to model the true regularities
  - not enough to also model the spurious regularities (assuming they are weaker)
- Standard ways to limit the capacity of a neural net:
  - Limit the number of hidden units.
  - Limit the size of the weights.
  - Stop the learning before it has time to overfit.

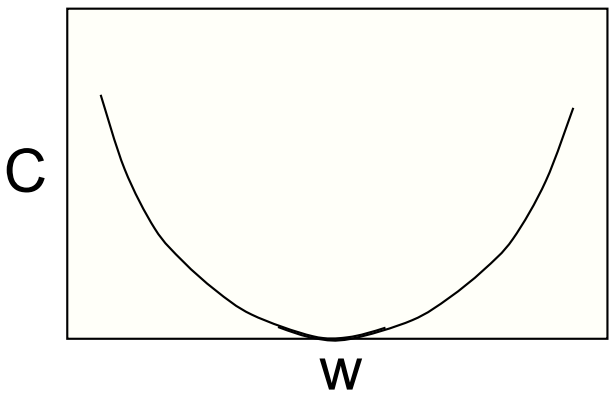
Limiting the size of the weights

- Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.

$$C = \ell + \frac{\lambda}{2} \sum_i w_i^2$$

- Keeps weights small unless they have big error derivatives.

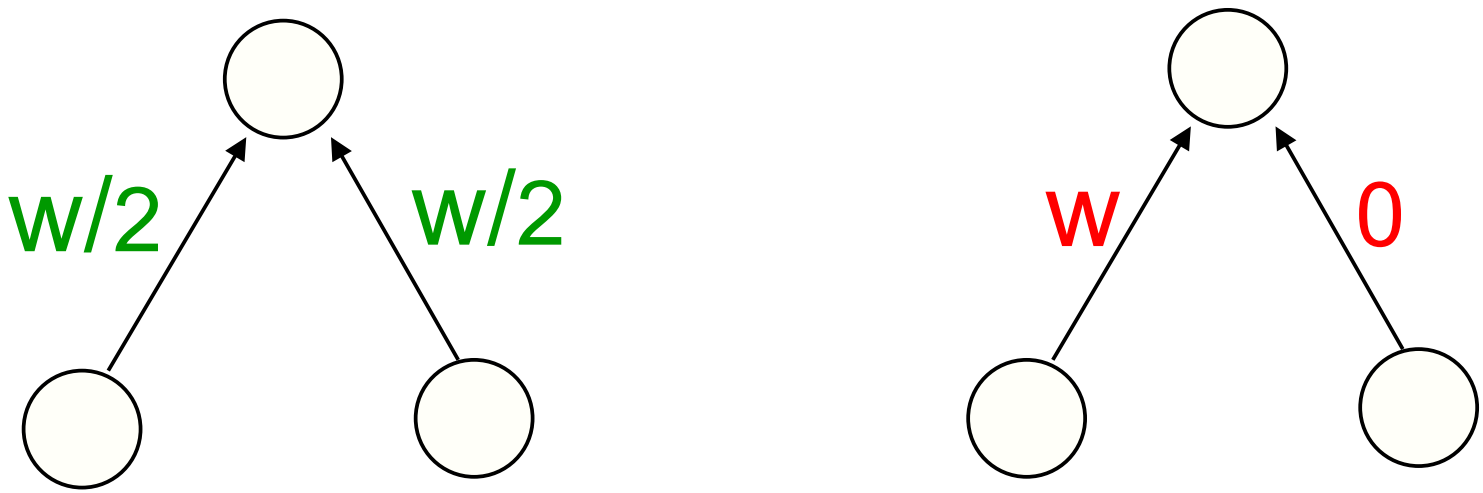
$$\frac{\partial C}{\partial w_i} = \frac{\partial \ell}{\partial w_i} + \lambda w_i$$



$$\text{when } \frac{\partial C}{\partial w_i} = 0, \quad w_i = \frac{1}{\lambda} \frac{\partial \ell}{\partial w_i}$$

The effect of weight-decay

- It prevents the network from using weights that it does not need
  - This can often improve generalization a lot.
  - It helps to stop it from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.
- But, if the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one $\rightarrow$ other form of weight decay?



Deciding how much to restrict the capacity

- How do we decide which limit to use and how strong to make the limit?
  - If we use the test data we get an unfair prediction of the error rate we would get on new test data.
  - Suppose we compared a set of models that gave random results, the best one on a particular dataset would do better than chance. But it won't do better than chance on another test set.
- So use a separate validation set to do model selection.

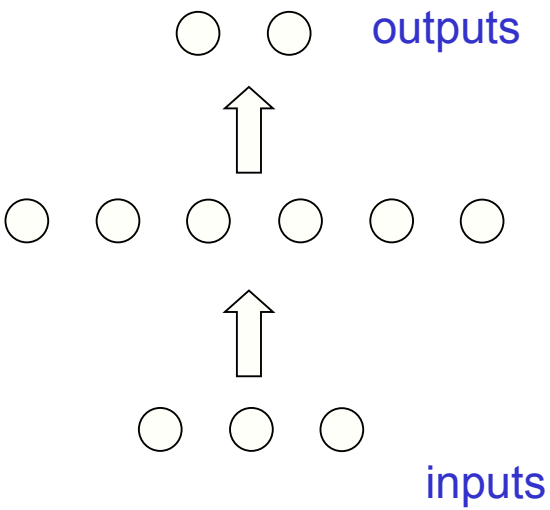
Using a validation set

- Divide the total dataset into three subsets:
  - Training data is used for learning the parameters of the model.
  - Validation data is not used of learning but is used for deciding what type of model and what amount of regularization works best
  - Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data
- We could then re-divide the total dataset to get another unbiased estimate of the true error rate.

Preventing overfitting by early stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse
- The capacity of the model is limited because the weights have not had time to grow big.

Why early stopping works



- When the weights are very small, every hidden unit is in its linear range.
  - So a net with a large layer of hidden units is linear.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.