An introduction to Deep Learning

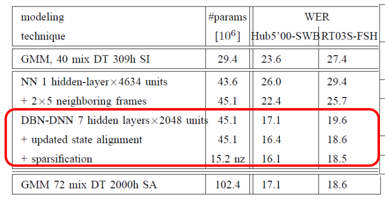
Collection of relevant presentations from others

1

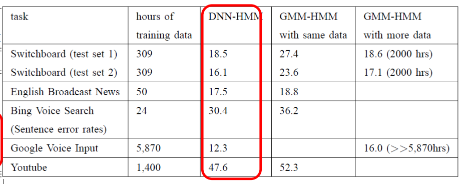
DL is providing breakthrough results in speech recognition and image classification …

From this Hinton et al 2012 paper:

http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/38131.pdf



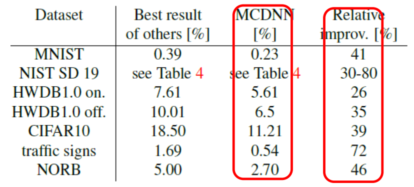
modeling technique	#params [10^6]	WER Hub5'00-SWB	WER RT03S-FSH
GMM, 40 mix DT 309h SI	29.4	23.6	27.4
NN 1 hidden-layer×4634 units	43.6	26.0	29.4
+ 2×5 neighboring frames	45.1	22.4	25.7
DBN-DNN 7 hidden layers×2048 units	45.1	17.1	19.6
+ updated state alignment	45.1	16.4	18.6
+ sparsification	15.2 nz	16.1	18.5
GMM 72 mix DT 2000h SA	102.4	17.1	18.6



task	hours of training data	DNN-HMM	GMM-HMM with same data	GMM-HMM with more data
Switchboard (test set 1)	309	18.5	27.4	18.6 (2000 hrs)
Switchboard (test set 2)	309	16.1	23.6	17.1 (2000 hrs)
English Broadcast News	50	17.5	18.8	
Bing Voice Search (Sentence error rates)	24	30.4	36.2	
Google Voice Input	5,870	12.3		16.0 (>>5,870hrs)
Youtube	1,400	47.6	52.3	

go here: http://yann.lecun.com/exdb/mnist/

From here:
http://people.idsia.ch/~juergen/cvpr2012.pdf



Dataset	Best result of others [%]	MCDNN [%]	Relative improv. [%]
MNIST	0.39	0.23	41
NIST SD 19	see Table 4	see Table 4	30-80
HWDB1.0 on.	7.61	5.61	26
HWDB1.0 off.	10.01	6.5	35
CIFAR10	18.50	11.21	39
traffic signs	1.69	0.54	72
NORB	5.00	2.70	46

2

So, 1. **what exactly is deep learning** ?

And, 2. **why is it generally better** than other methods on image, speech and certain other types of data?

The short answers

1. **'Deep Learning'** **means** using a **neural network** with **several layers of nodes** between input and output

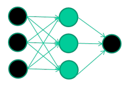
2. **the series of layers between input & output do feature identification and processing in a series of stages, just as our brains seem to.**

3

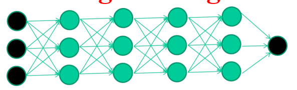
hmmm… OK, but:

3. **multilayer neural networks have been around for 25 years. What's actually new?**

we have always had good algorithms for learning the weights in networks with 1 hidden layer



but these algorithms are not good at learning the weights for networks with more hidden layers



what's new is: **algorithms for training many-later networks**

4

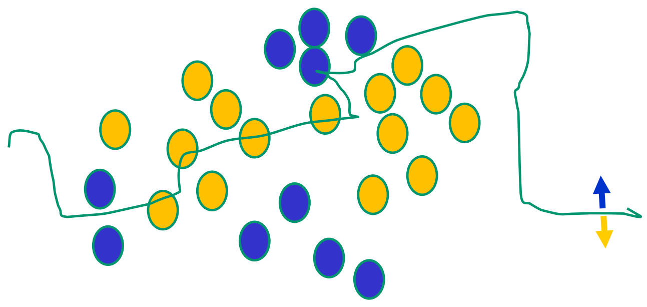
longer answers

1. reminder/quick-explanation of how neural network weights are learned;
2. the idea of **unsupervised feature learning** (why 'intermediate features' are important for difficult classification tasks, and how NNs seem to naturally learn them)
3. The 'breakthrough' – the simple trick for training Deep neural networks

5

The decision boundary perspective…

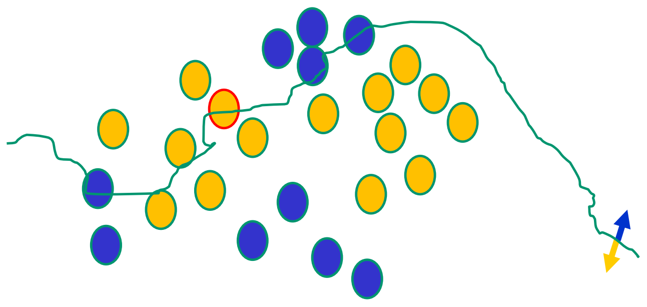
Initial random weights



6

The decision boundary perspective…

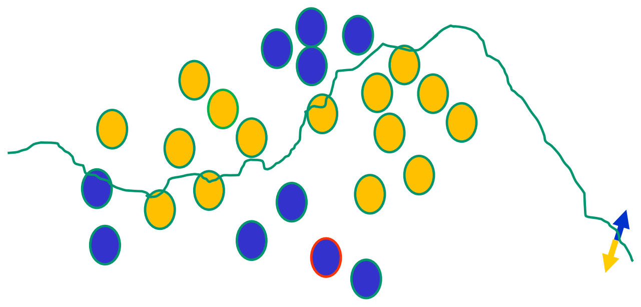
Present a training instance / adjust the weights



7

The decision boundary perspective…

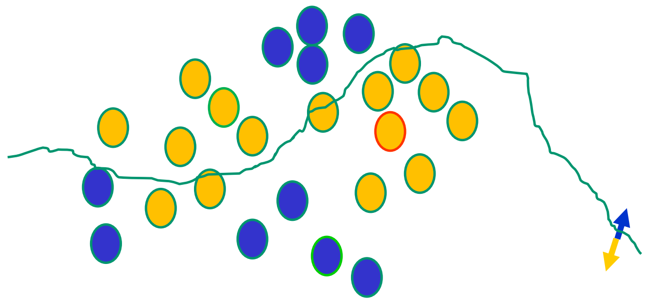
Present a training instance / adjust the weights



8

The decision boundary perspective…

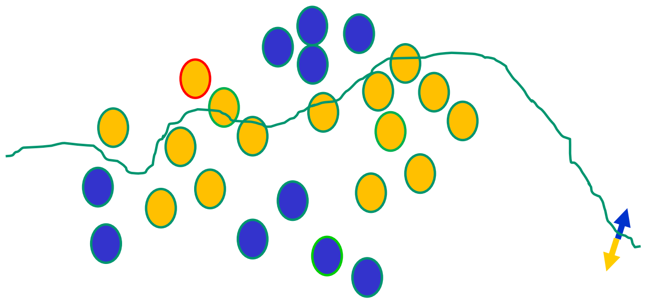
Present a training instance / adjust the weights



9

The decision boundary perspective…

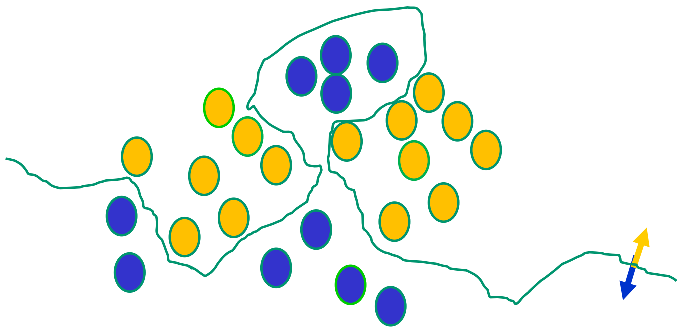
Present a training instance / adjust the weights



10

The decision boundary perspective…

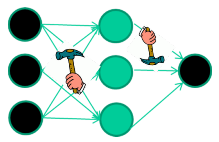
Eventually ….



11

Observation

- weight-learning algorithms for NNs are dumb
- they work by making thousands and thousands of tiny adjustments, each making the network do better at the most recent pattern, but perhaps a little worse on many others
- but, by dumb luck, eventually this tends to be good enough to learn effective classifiers for many real applications



12

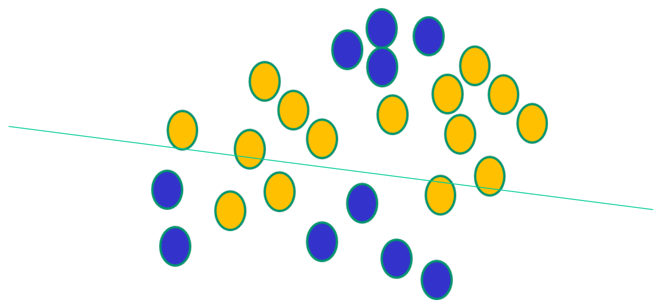
Some other points

Detail of a standard NN weight learning algorithm – **later**

If *f*(*x*) is non-linear, a network with 1 hidden layer can, in theory, learn perfectly any classification problem. A set of weights exists that can produce the targets from the inputs. The problem is finding them.

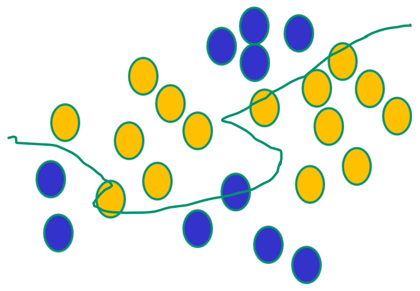
Some other ‘by the way’ points

If *f*(*x*) is linear, the NN can **only** draw straight decision boundaries (even if there are many layers of units)



Some other ‘by the way’ points

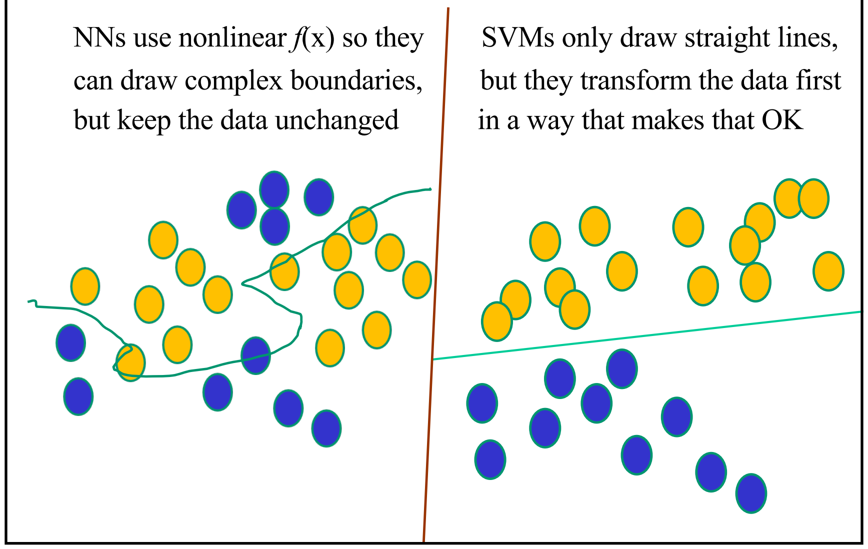
NNs use nonlinear *f*(x) so they can draw complex boundaries, but keep the data unchanged



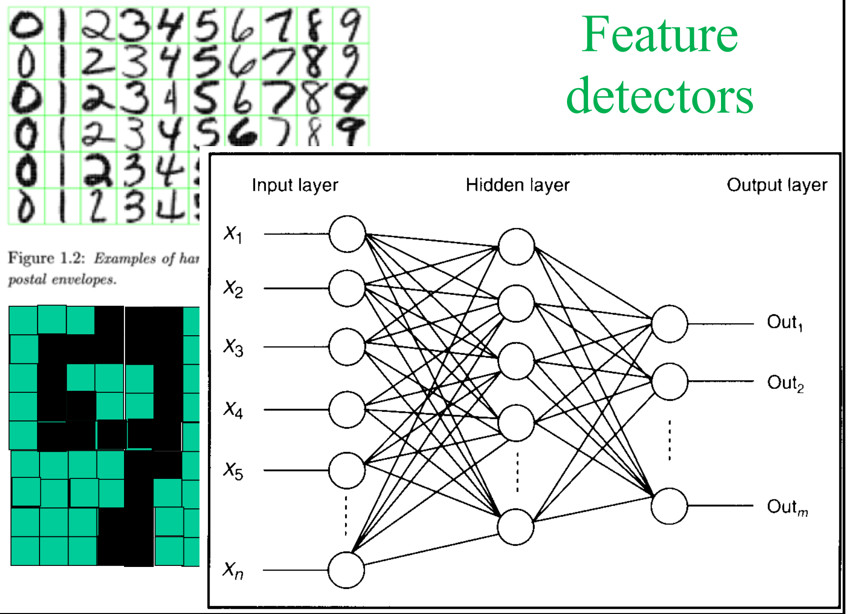
Some other ‘by the way’ points

NNs use nonlinear *f*(x) so they can draw complex boundaries, but keep the data unchanged

SVMs only draw straight lines, but they transform the data first in a way that makes that OK

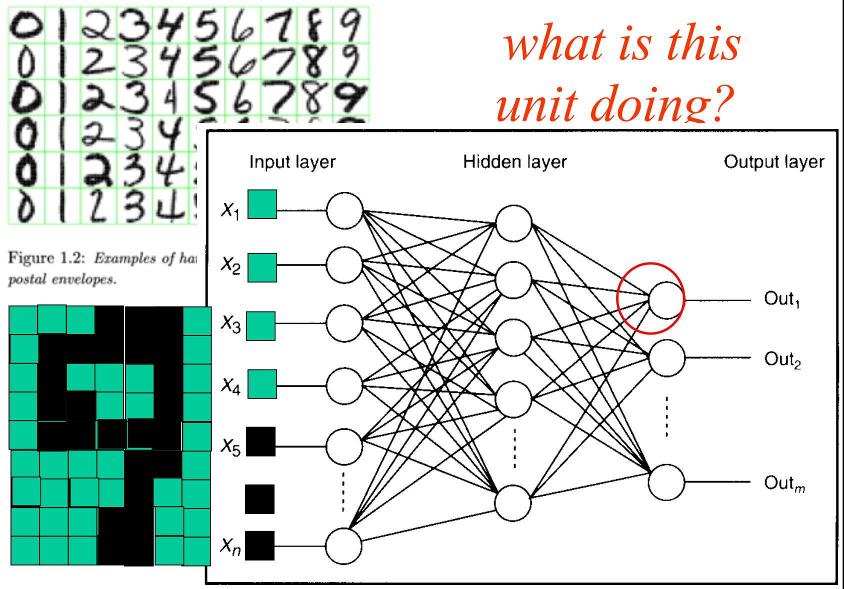


Feature detectors



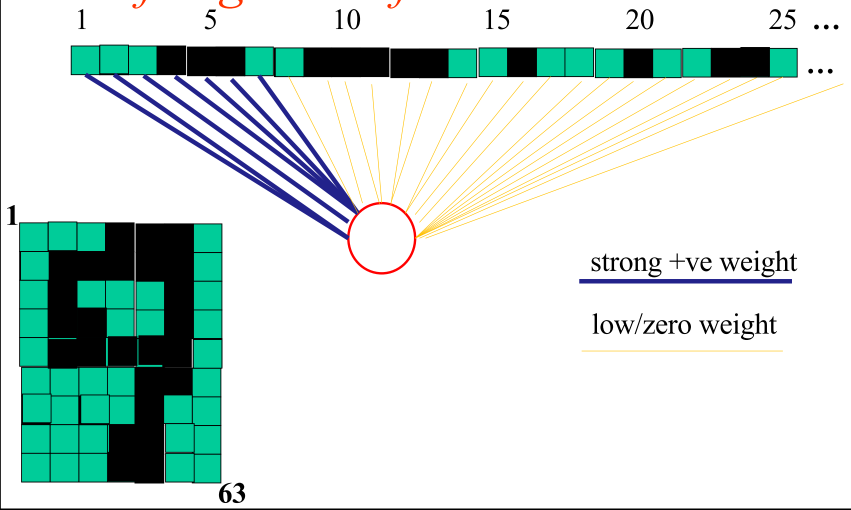
17

what is this unit doing?



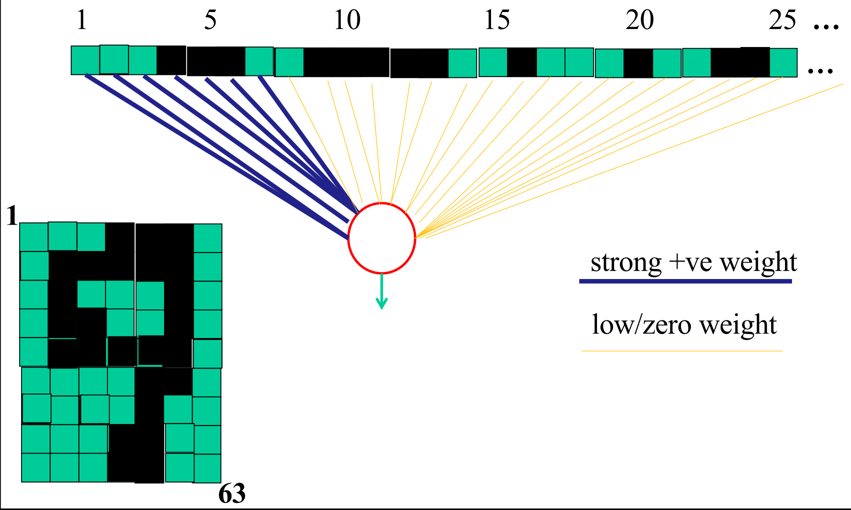
18

Hidden layer units become *self-organised feature detectors*



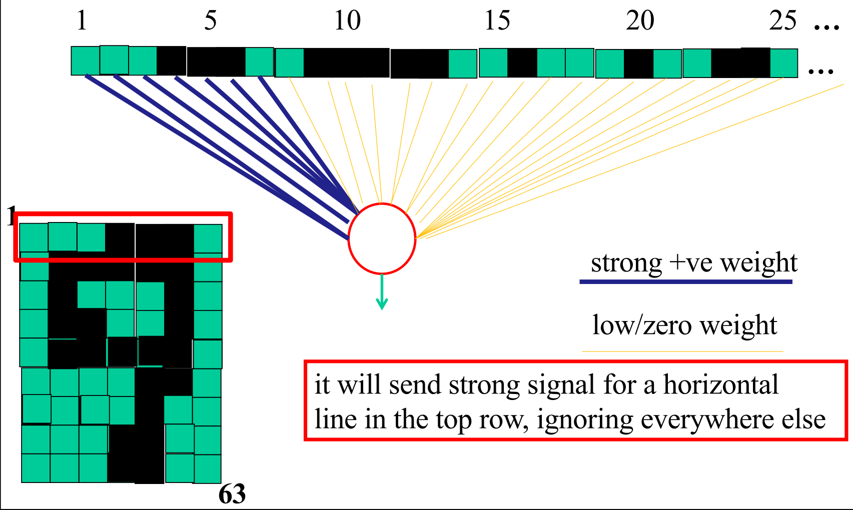
19

What does this unit detect?



20

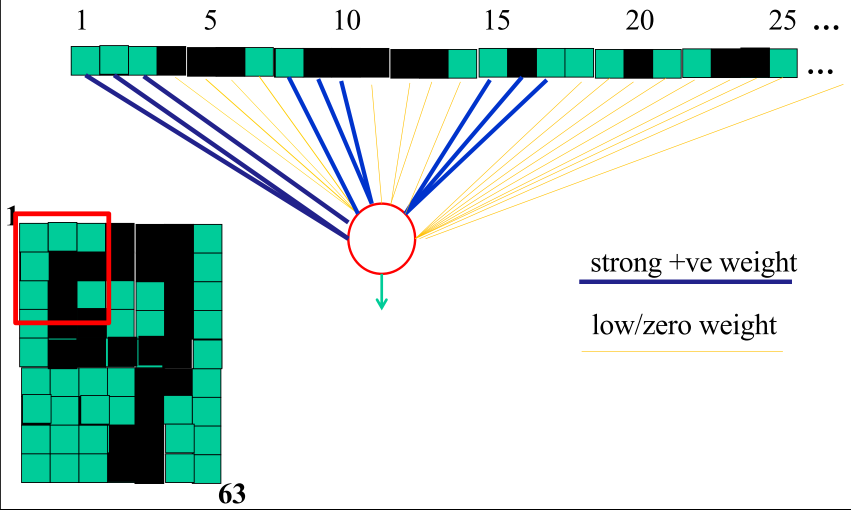
What does this unit detect?



it will send strong signal for a horizontal line in the top row, ignoring everywhere else

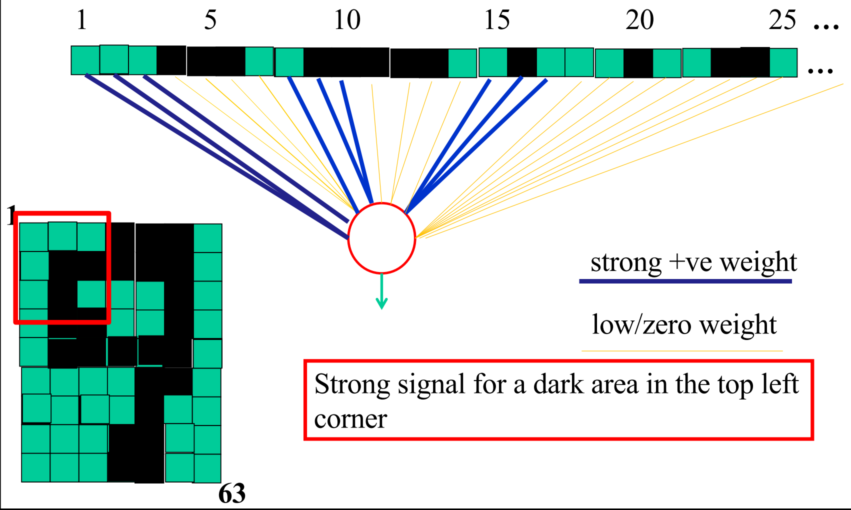
21

What does this unit detect?



22

What does this unit detect?



Strong signal for a dark area in the top left corner

23

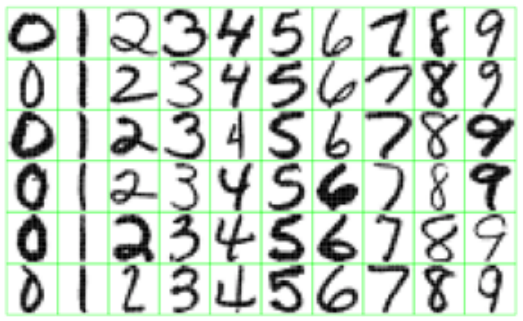


Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

What features might you expect a good NN to learn, when trained with data like this?

24

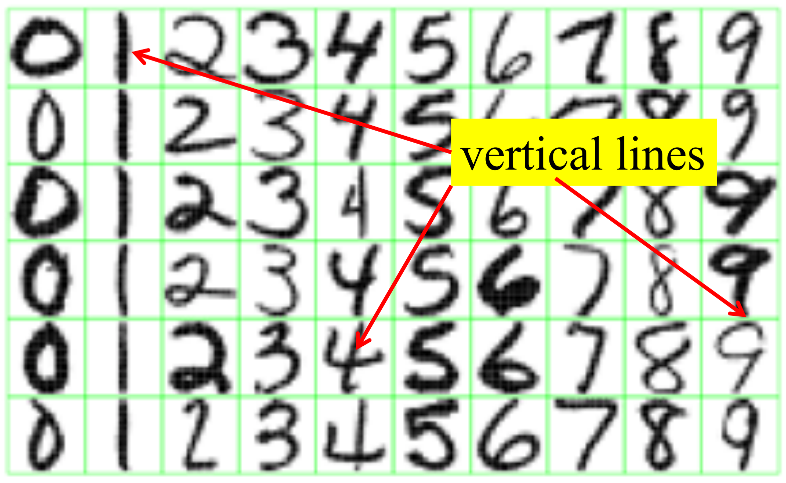


Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

25

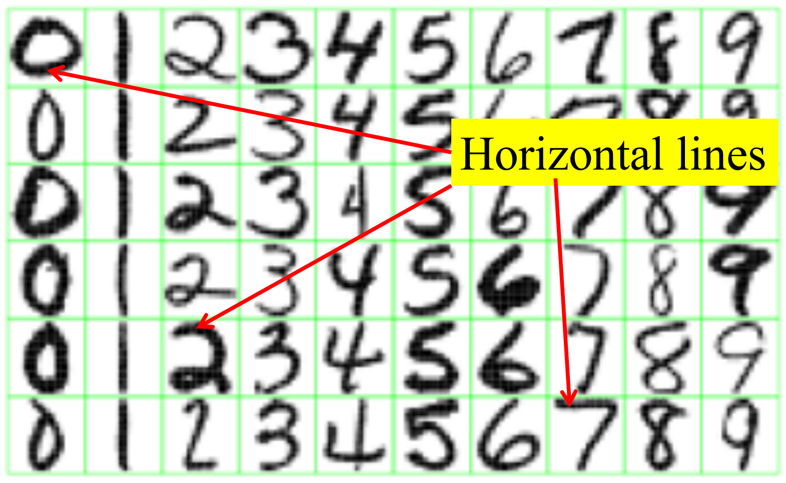


Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

26

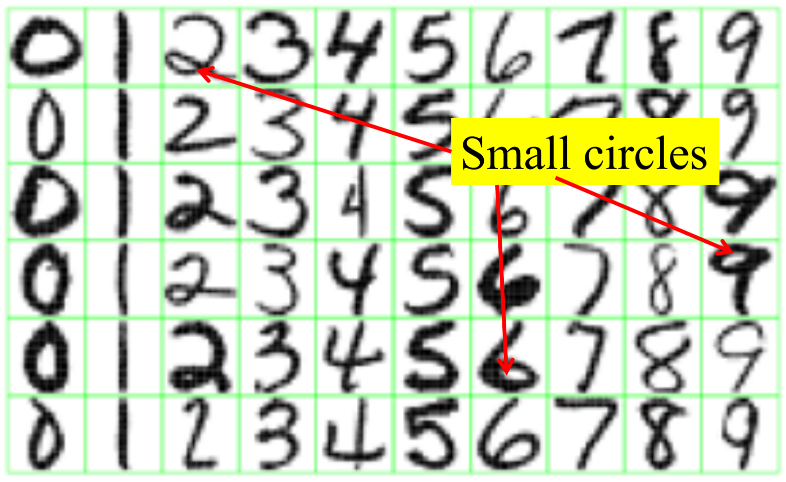
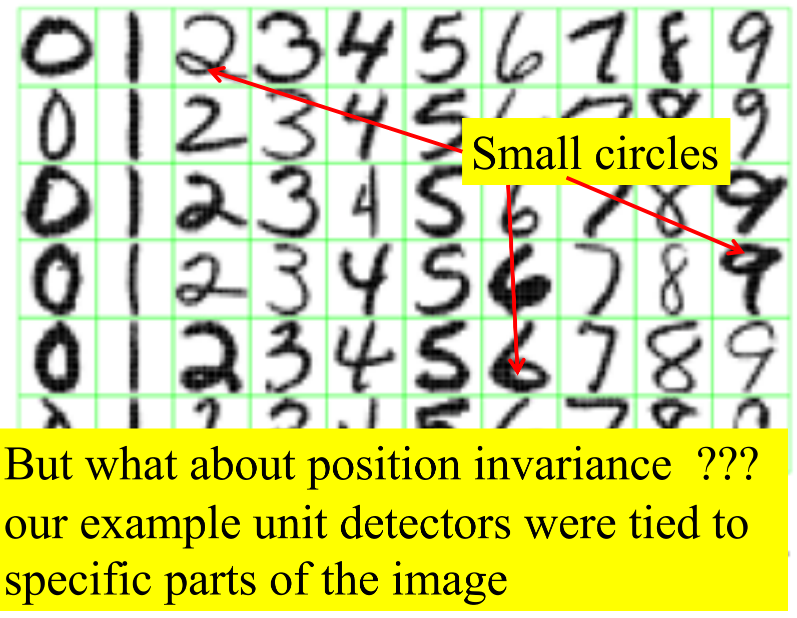


Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

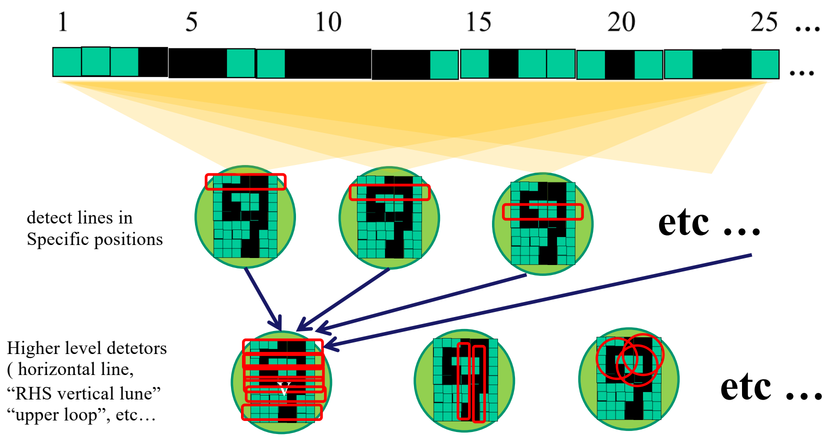
27



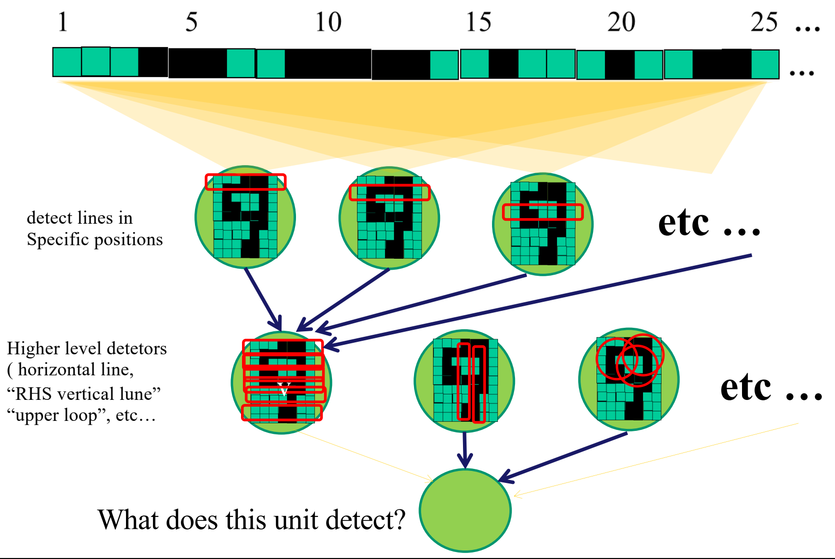
But what about position invariance ???
our example unit detectors were tied to
specific parts of the image

28

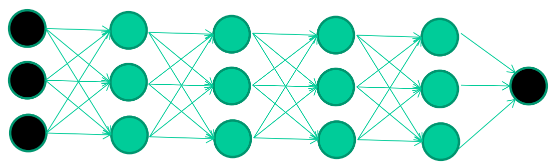
successive layers can learn higher-level features …



successive layers can learn higher-level features …

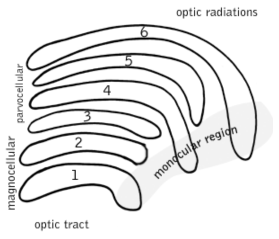


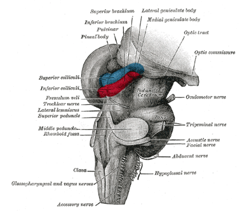
So: *multiple layers make sense*

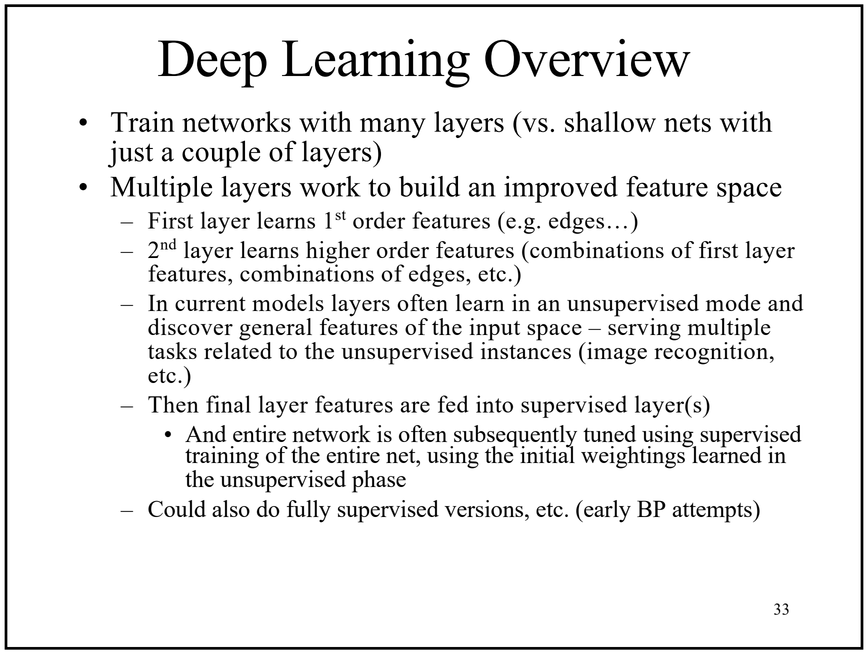


So: *multiple layers make sense*

Your brain works that way



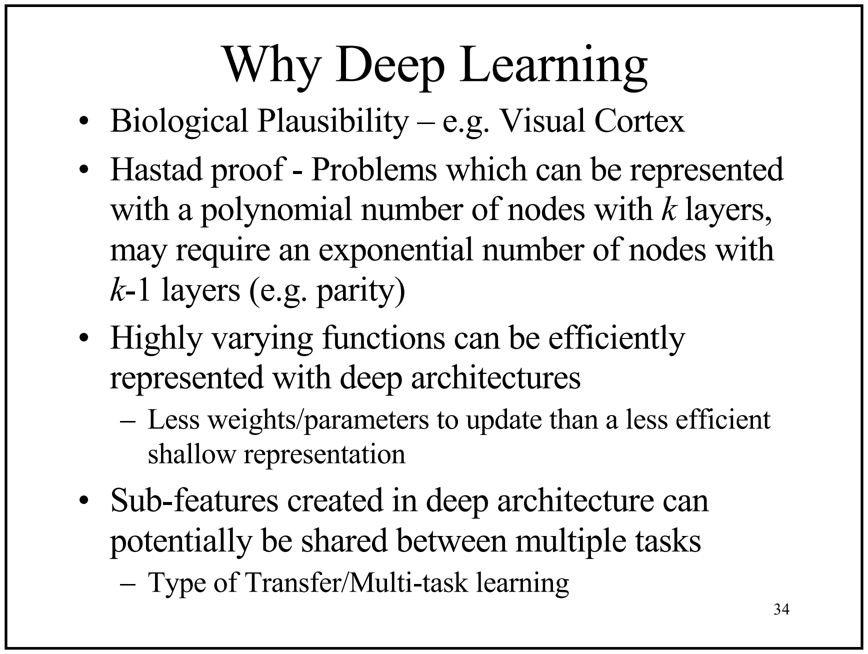




Deep Learning Overview

- Train networks with many layers (vs. shallow nets with just a couple of layers)
- Multiple layers work to build an improved feature space
  - First layer learns 1st order features (e.g. edges…)
  - 2nd layer learns higher order features (combinations of first layer features, combinations of edges, etc.)
  - In current models layers often learn in an unsupervised mode and discover general features of the input space – serving multiple tasks related to the unsupervised instances (image recognition, etc.)
  - Then final layer features are fed into supervised layer(s)
    - And entire network is often subsequently tuned using supervised training of the entire net, using the initial weightings learned in the unsupervised phase
  - Could also do fully supervised versions, etc. (early BP attempts)

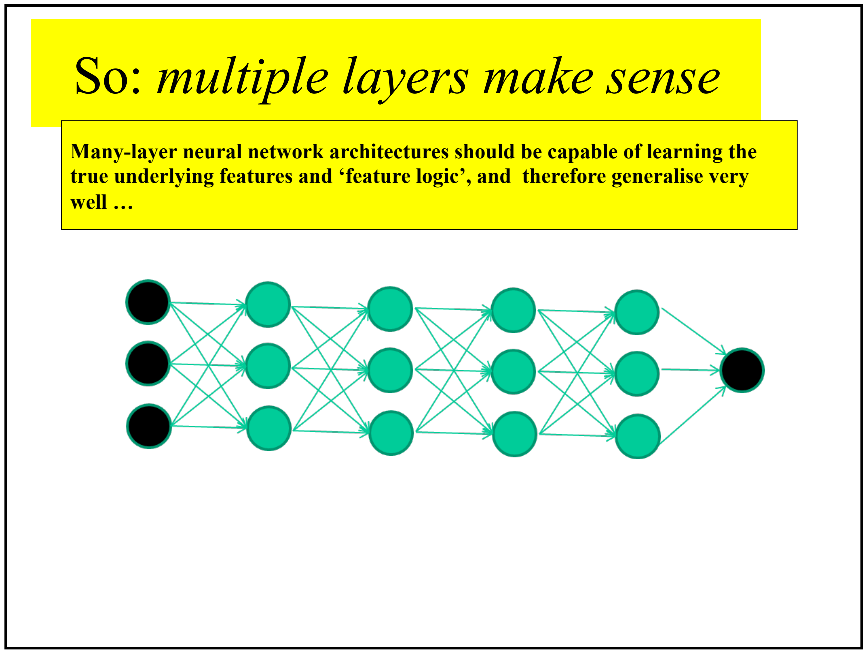
33



Why Deep Learning

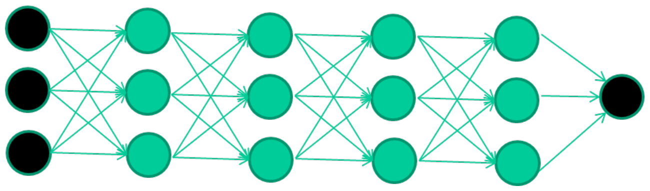
- Biological Plausibility – e.g. Visual Cortex
- Hastad proof - Problems which can be represented with a polynomial number of nodes with *k* layers, may require an exponential number of nodes with *k*-1 layers (e.g. parity)
- Highly varying functions can be efficiently represented with deep architectures
  - Less weights/parameters to update than a less efficient shallow representation
- Sub-features created in deep architecture can potentially be shared between multiple tasks
  - Type of Transfer/Multi-task learning

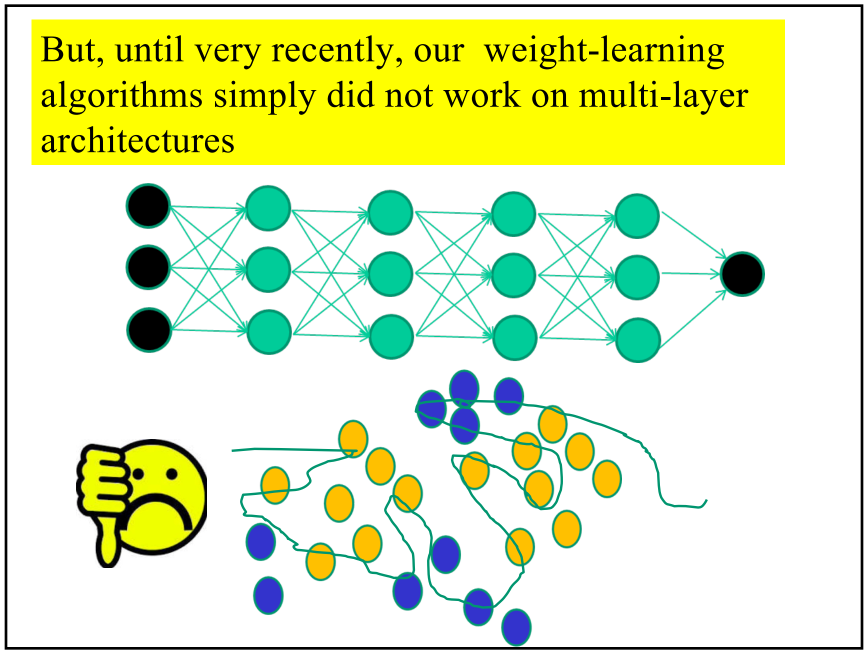
34



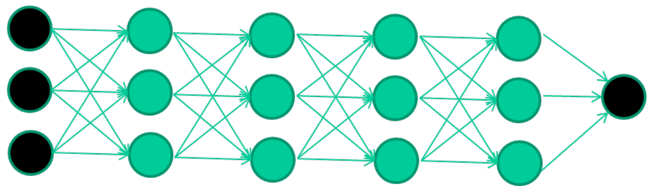
So: *multiple layers make sense*

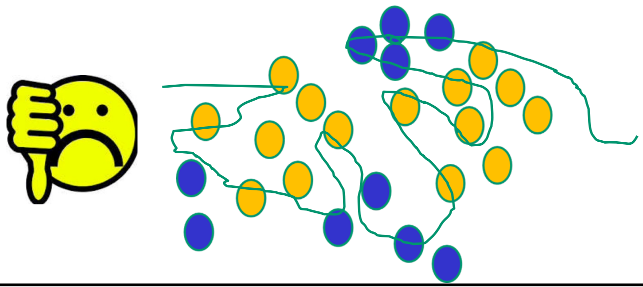
Many-layer neural network architectures should be capable of learning the true underlying features and 'feature logic', and therefore generalise very well …





But, until very recently, our weight-learning algorithms simply did not work on multi-layer architectures

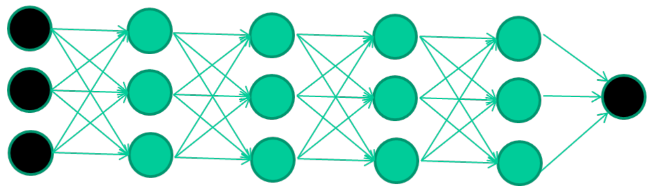




Along came deep learning …

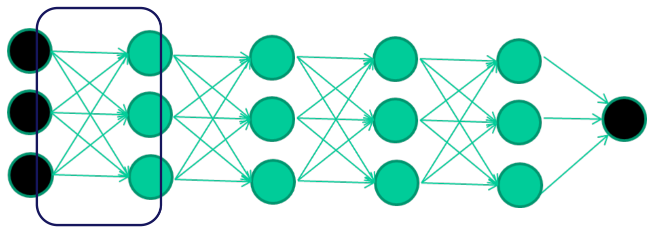
37

The new way to train multi-layer NNs…



38

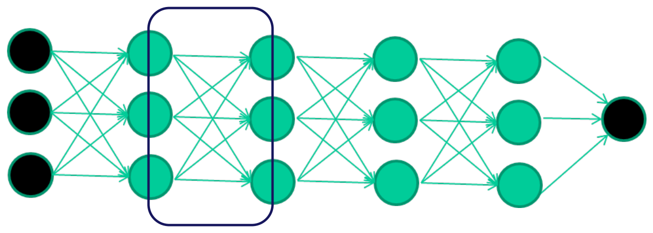
The new way to train multi-layer NNs…



Train **this** layer first

39

The new way to train multi-layer NNs…

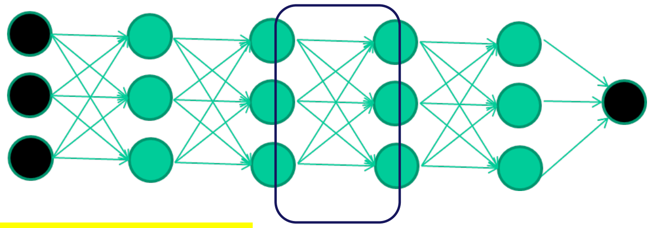


Train **this** layer first

then **this** layer

40

The new way to train multi-layer NNs…



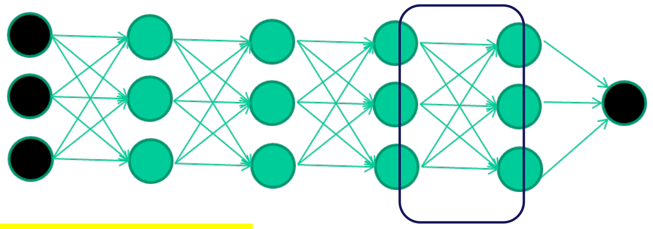
Train **this** layer first

then **this** layer

then **this** layer

41

The new way to train multi-layer NNs…



Train **this** layer first

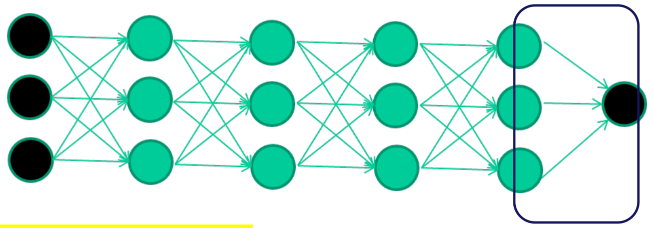
then **this** layer

then **this** layer

then **this** layer

42

The new way to train multi-layer NNs…



Train **this** layer first

then **this** layer

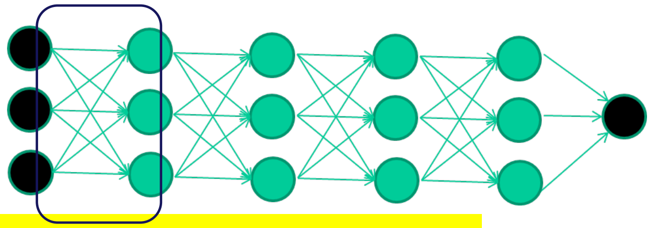
then **this** layer

then **this** layer

finally **this** layer

43

The new way to train multi-layer NNs…

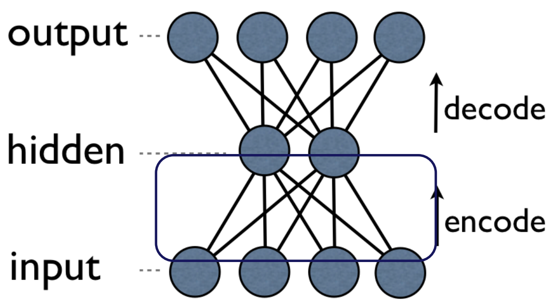


EACH of the (non-output) layers is trained to be an ***auto-encoder***

Basically, it is forced to learn good features that describe what comes from the previous layer

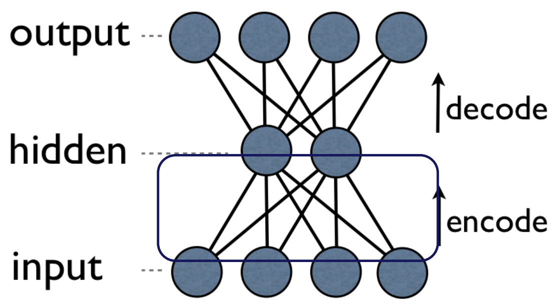
44

an auto-encoder is trained, with an absolutely standard weight-adjustment algorithm to reproduce the input



45

an auto-encoder is trained, with an absolutely standard weight-adjustment algorithm to reproduce the input

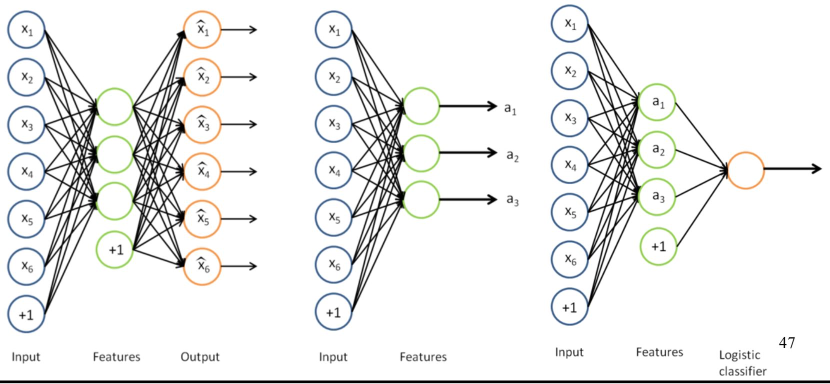


By making this happen with (many) fewer units than the inputs, this forces the 'hidden layer' units to become good feature detectors

46

Auto-Encoders

- A type of unsupervised learning which tries to discover generic features of the data
  - Learn identity function by learning important sub-features (not by just passing through data)
  - Compression, etc.
  - Can use just new features in the new training set or concatenate both

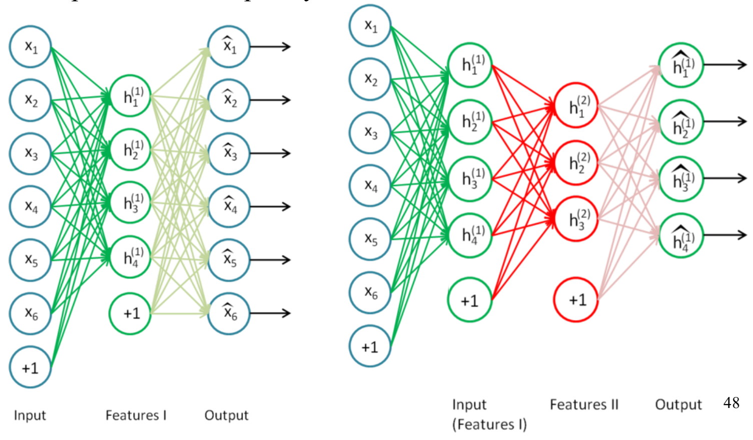


47

47

Stacked Auto-Encoders

- Bengio (2007) – After Deep Belief Networks (2006)
- Stack many (sparse) auto-encoders in succession and train them using greedy layer-wise training
- Drop the decode output layer each time

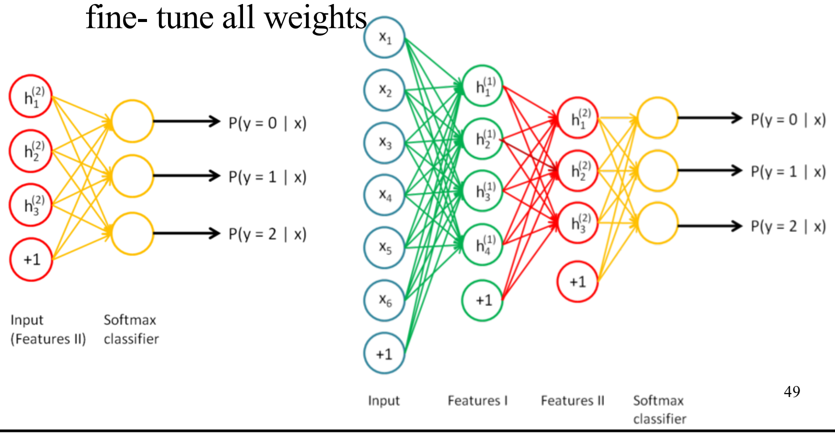


48

48

Stacked Auto-Encoders

- Do supervised training on the last layer using final features
- Then do supervised training on the entire network to fine- tune all weights



49

49

Sparse Encoders

- Auto encoders will often do a dimensionality reduction
  - PCA-like or non-linear dimensionality reduction
- This leads to a "dense" representation which is nice in terms of parsimony
  - All features typically have non-zero values for any input and the combination of values contains the compressed information
- However, this distributed and entangled representation can often make it more difficult for successive layers to pick out the salient features
- A *sparse* representation uses more features where at any given time many/most of the features will have a 0 value
  - Thus there is an implicit compression each time but with varying nodes
  - This leads to more localist variable length encodings where a particular node (or small group of nodes) with value 1 signifies the presence of a feature (small set of bases)
  - A type of simplicity bottleneck (regularizer)
  - This is easier for subsequent layers to use for learning

50

50

How do we implement a sparse Auto-Encoder?

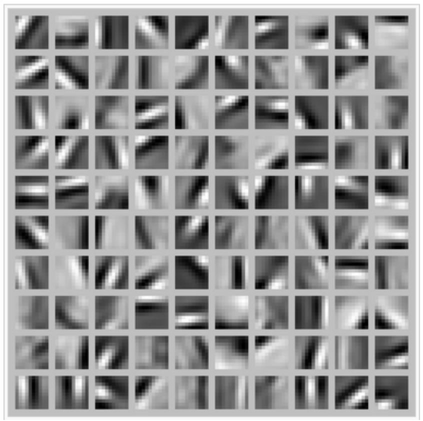
- Use more hidden nodes in the encoder
- Use regularization techniques which encourage sparseness (e.g. a significant portion of nodes have 0 output for any given input)
  - Penalty in the learning function for non-zero nodes
  - Weight decay
  - etc.
- De-noising Auto-Encoder
  - Stochastically corrupt training instance each time, but still train auto-encoder to decode the uncorrupted instance, forcing it to learn conditional dependencies within the instance
  - Better empirical results, handles missing values well

51

51

Sparse Representation

- For bases below, which is easier to see intuition for current pattern - if a few of these are on and the rest 0, or if all have some non-zero value?
- Easier to learn if sparse



52

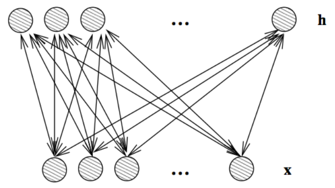
52

Stacked Auto-Encoders

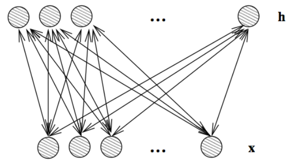
- Concatenation approach (i.e. using both hidden features and original features in final (or other) layers) can be better if not doing fine tuning. If fine tuning, the pure replacement approach can work well.
- Always fine tune if there is a sufficient amount of labeled data
- For real valued inputs, MLP training is like regression and thus could use linear output node activations, still sigmoid at hidden
- Stacked Auto-Encoders empirically not quite as accurate as DBNs (Deep Belief Networks)
  - (with De-noising auto-encoders, stacked auto-encoders competitive with DBNs)
  - Not generative like DBNs, though recent work with de-noising auto-encoders may allow generative capacity

Deep Belief Networks (DBN)

- Geoff Hinton (2006)
- Uses Greedy layer-wise training but each layer is an RBM (Restricted Boltzmann Machine)
- RBM is a constrained Boltzmann machine with
  - No lateral connections between hidden (**h**) and visible (**x**) nodes
  - Symmetric weights
  - Does not use annealing/temperature, but that is all right since each RBM not seeking a global minima, but rather an incremental transformation of the feature space
  - Typically uses probabilistic logistic node, but other activations possible



RBM Sampling and Training



- Initial state typically set to a training example **x** (can be real valued)
- Sampling is an iterative back and forth process
  - $P(h_i = 1|\mathbf{x}) = \text{sigmoid}(W_i\mathbf{x} + c_i) = 1/(1+e^{-\text{net}(h_i)})$ // c_i is hidden node bias
  - $P(x_i = 1|\mathbf{h}) = \text{sigmoid}(W'_i\mathbf{h} + b_i) = 1/(1+e^{-\text{net}(x_i)})$ // b_i is visible node bias
- Contrastive Divergence (CD-*k*): How much contrast (in the statistical distribution) is there in the divergence from the original training example to the relaxed version after *k* relaxation steps
- Then update weights to decrease the divergence as in Boltzmann
- Typically just do CD-1 (Good empirical results)
  - Since small learning rate, doing many of these is similar to doing fewer versions of CD-*k* with *k* > 1
  - Note CD-1 just needs to get the gradient direction right, which it usually does, and then change weights in that direction according to the learning rate

`RBMupdate`$(\mathbf{x}_1, \epsilon, W, \mathbf{b}, \mathbf{c})$

This is the RBM update procedure for binomial units. It can easily adapted to other types of units.

$\mathbf{x}_1$ is a sample from the training distribution for the RBM

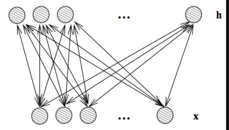
ϵ is a learning rate for the stochastic gradient descent in Contrastive Divergence

W is the RBM weight matrix, of dimension (number of hidden units, number of inputs)

$\mathbf{b}$ is the RBM offset vector for input units

$\mathbf{c}$ is the RBM offset vector for hidden units

Notation: $Q(\mathbf{h}_{2\cdot} = 1|\mathbf{x}_2)$ is the vector with elements $Q(\mathbf{h}_{2i} = 1|\mathbf{x}_2)$



```
for all hidden units $i$ do
    • compute $Q(\mathbf{h}_{1i} = 1|\mathbf{x}_1)$ (for binomial units, sigm$(\mathbf{c}_i + \sum_j W_{ij}\mathbf{x}_{1j})$)
    • sample $\mathbf{h}_{1i} \in \{0, 1\}$ from $Q(\mathbf{h}_{1i}|\mathbf{x}_1)$
end for
for all visible units $j$ do
    • compute $P(\mathbf{x}_{2j} = 1|\mathbf{h}_1)$ (for binomial units, sigm$(\mathbf{b}_j + \sum_i W_{ij}\mathbf{h}_{1i})$)
    • sample $\mathbf{x}_{2j} \in \{0, 1\}$ from $P(\mathbf{x}_{2j} = 1|\mathbf{h}_1)$
end for
for all hidden units $i$ do
    • compute $Q(\mathbf{h}_{2i} = 1|\mathbf{x}_2)$ (for binomial units, sigm$(\mathbf{c}_i + \sum_j W_{ij}\mathbf{x}_{2j})$)
end for
• $W \leftarrow W + \epsilon(\mathbf{h}_1\mathbf{x}'_1 - Q(\mathbf{h}_{2\cdot} = 1|\mathbf{x}_2)\mathbf{x}'_2)$
• $\mathbf{b} \leftarrow \mathbf{b} + \epsilon(\mathbf{x}_1 - \mathbf{x}_2)$
• $\mathbf{c} \leftarrow \mathbf{c} + \epsilon(\mathbf{h}_1 - Q(\mathbf{h}_{2\cdot} = 1|\mathbf{x}_2))$
```

$$\Delta w_{ij} = \varepsilon(h_{1,j} \cdot x_{1,i} - Q(h_{k+1,j} = 1 | \mathbf{x}_{k+1})x_{k+1,i})$$

RBM Update Notes and Variations

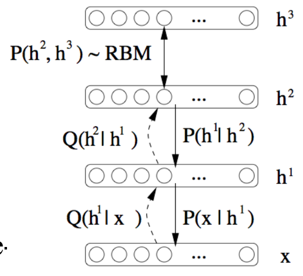
- Binomial unit means the standard MLP sigmoid unit
- *Q* and *P* are probability distribution vectors for hidden (**h**) and visible/input (**x**) vectors respectively
- During relaxation/weight update can alternatively do updates based on the real valued probabilities (sigmoid(*net*)) rather than the 1/0 sampled logistic states
  - Always use actual/binary values from initial x -> h
    - Doing this makes the hidden nodes a sparser bottleneck and is a regularizer helping to avoid overfit
  - Could use probabilities on the h -> x and/or final x -> h
    - in CD-*k* the final update of the hidden nodes usually use the probability value to decrease the final arbitrary sampling variation (sampling noise)
- Lateral restrictions of RBM allow this fast sampling

RBM Update Variations and Notes

- Initial weights, small random, 0 mean, sd ~ .01
  - Don't want hidden node probabilities early on to be close to 0 or 1, else slows learning, since less early randomness/mixing? Note that this is a bit like annealing/temperature in Boltzmann
- Set initial **x** bias values as a function of how often node is on in the training data, and **h** biases to 0 or negative to encourage sparsity
- Better speed when using momentum (~.5)
- Weight decay good for smoothing and also encouraging more mixing (hidden nodes more stochastic when they do not have large net magnitudes)
  - Also a reason to increase *k* over time in CD-*k* as mixing decreases as weight magnitudes increase

Deep Belief Network Training

- Same greedy layer-wise approach
- First train lowest RBM ($h^0 – h^1$) using RBM update algorithm (note h^0 is x)
- Freeze weights and train subsequent RBM layers
- Then connect final outputs to a supervised model and train that model
- Finally, unfreeze all weights, and train full DBN with supervised model to fine-tune weights

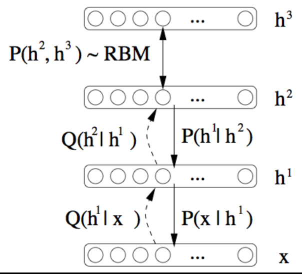


During execution typically iterate multiple times at the top RBM layer

Can use DBN as a Generative model to create sample x vectors

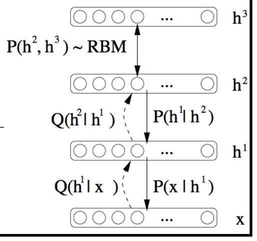
1. Initialize top layer to an arbitrary vector
   - Gibbs sample (relaxation) between the top two layers *m* times
   - If we initialize top layer with values obtained from a training example, then need less Gibbs samples
2. Pass the vector down through the network, sampling with the calculated probabilities at each layer
3. Last sample at bottom is the generated x value (can be real valued if we use the probability vector rather than sample)

Alternatively, can start with an x at the bottom, relax to a top value, then start from that vector when generating a new x, which is the dotted lines version. More like standard Boltzmann machine processing.



```
TrainUnsupervisedDBN($\widehat{P}, \epsilon, \ell, W, \mathbf{b}, \mathbf{c}$, mean_field_computation)
Train a DBN in a purely unsupervised way, with the greedy layer-wise procedure in which each added layer
is trained as an RBM (e.g. by Contrastive Divergence).
$\widehat{P}$ is the input training distribution for the network
$\epsilon$ is a learning rate for the RBM training
$\ell$ is the number of layers to train
$W^k$ is the weight matrix for level $k$, for $k$ from 1 to $\ell$
$\mathbf{b}^k$ is the visible units offset vector for RBM at level $k$, for $k$ from 1 to $\ell$
$\mathbf{c}^k$ is the hidden units offset vector for RBM at level $k$, for $k$ from 1 to $\ell$
mean_field_computation is a Boolean that is true iff training data at each additional level is obtained
by a mean-field approximation instead of stochastic sampling

  for $k = 1$ to $\ell$ do
    • initialize $W^k = 0, \mathbf{b}^k = 0, \mathbf{c}^k = 0$
    while not stopping criterion do
      • sample $\mathbf{h}^0 = \mathbf{x}$ from $\widehat{P}$
      for $i = 1$ to $k - 1$ do
        if mean_field_computation then
          • assign $\mathbf{h}^i_j$ to $Q(\mathbf{h}^i_j = 1|\mathbf{h}^{i-1})$, for all elements $j$ of $\mathbf{h}^i$
        else
          • sample $\mathbf{h}^i_j$ from $Q(\mathbf{h}^i_j|\mathbf{h}^{i-1})$, for all elements $j$ of $\mathbf{h}^i$
        end if
      end for
      • RBMupdate($\mathbf{h}^{k-1}, \epsilon, W^k, \mathbf{b}^k, \mathbf{c}^k$) {thus providing $Q(\mathbf{h}^k|\mathbf{h}^{k-1})$ for future use}
    end while
  end for
```

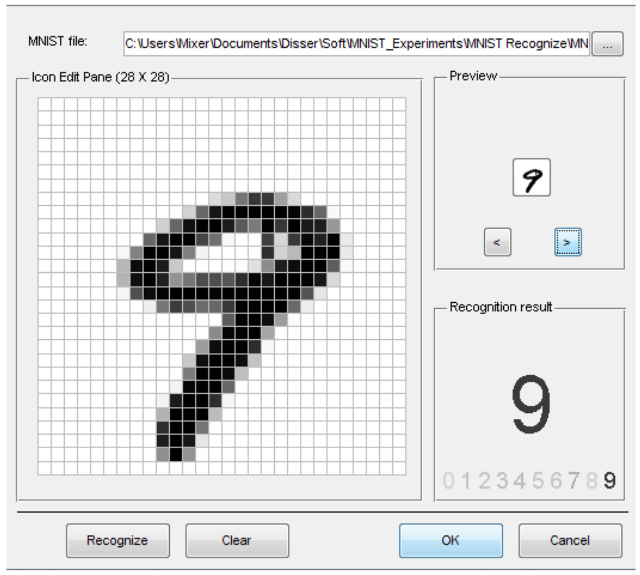


DBN Execution

- After all layers have learned then the output of the last layer can be input to a supervised learning model
- Note that at this point we could potentially throw away all the downward weights in the network as they will not actually be used during the normal feedforward execution process (as we did with the Stacked Auto Encoder)
  - Note that except for the downward bias weights **b** they are the same symmetric weights anyways
  - If we are relaxing *M* times in the top layer then we would still need the downward weights for that layer
  - Also if we are generating **x** values we would need all of them
- The final weight tuning is usually done with backpropagation, which only updates the feedforward weights, ignoring any downward weights

DBN Learning Notes

- RBM stopping criteria still in issue. One common approach is reconstruction error which is the probability that the final x after CD-k is the initial x. (i.e. $P(\square x|E[h|x\square])$. The other most common approach is AIC (Annealed Importance Sampling). Both have been shown to be problematic.
- Each layer updates weights so as to make training sample patterns more likely (lower energy) in the free state (and non-training sample patterns less likely).
- This unsupervised approach learns broad features (in the hidden/subsequent layers of RBMs) which can aid in the process of making the types of patterns found in the training set more likely. This discovers features which can be associated across training patterns, and thus potentially helpful for other goals with the training set (classification, compression, etc.)
- Note still pairwise weights in RBMs, but because we can pick the number of hidden units and layers, we can represent any arbitrary distribution


MNIST file: C:\Users\Mixer\Documents\Disser\Soft\MNIST_Experiments\MNIST Recognize\MN

DBN Project Notes

- To be consistent just use 28×28 (764) data set of gray scale values (0-255)
  - Normalize to 0-1
  - Could try better preprocessing if want and helps in published accuracies, but start/stay with this
  - Small random initial weights
- Parameters
  - Hinton Paper, others – do a little searching and e-mail me a reference for extra credit points
  - http://yann.lecun.com/exdb/mnist/ for sample approaches
- Straight 200 hidden node MLP does quite good ~98%
  - Rough Hyperparameters - LR: ~.05-.1, Momentum ~.5
- Best class deep net results: ~98.5% - which is competitive
  - About half students never beat MLP baseline
  - Can you beat the 98.5%?

Deep Learning Project Past Experience

- Structure: ~3 hidden layers, ~500ish nodes/layer, more nodes/layers can be better but training is longer
- Training time:
  - DBN: ~10 epochs with the 60K set, small LR ~.005 often good
  - Can go longer, does not seem to overfit with the large data set
  - SAE: Can saturate/overfit, ~3 epochs good, but will be a function of your de-noising approach, which is essential for sparsity, use small LR ~.005, long training – up to 50 hours, got 98.55
    - Larger learning rates often lead low accuracy for both DBN and SAE
- Sampling vs using real probability value in DBN
  - Best results found when using real values vs. sampling
  - Some found sampling on the back-step of learning helps
  - When using sampling, probably requires longer training, but could actually lead to bigger improvements in the long run
  - Typical forward flow real during execution, but could do some sampling on the *m* iterations at the top layer. Some success with back-step at the top layer iteration (most don't do this at all)
  - We need to try/discover better variations

Deep Learning Project Past Experience

- Note: If we held out 50K of the dataset as unsupervised, then deep nets would more readily show noticeable improvement over BP
- A final full network fine tune with BP always helps
  - But can take 20+ hours
- Key take away – Most actual time spent training with different parameters. Thus, start early, and then you will have time to try multiple long runs to see which variations work. This does not take that much personal time, as you simply start it with some different parameters and go away for a day. If you wait until the last few days, there is no time to do these experiments.

DBN Notes

- Can use lateral connections in RBM (no longer RBM) but sampling becomes more difficult
  - ala standard Boltzmann requiring longer sampling chains.
  - Lateral connections can capture pairwise dependencies allowing the hidden nodes to focus on higher order issues. Can get better results.
- Deep Boltzmann machine – Allow continual relaxation across the full network
  - Receive input from above and below rather than sequence through RBM layers
  - Typically for successful training, first initialize weights using the standard greedy DBN training with RBM layers
  - Requires longer sampling chains ala Boltzmann
- Conditional and Temporal RBMs – allow node probabilities to be conditioned by some other inputs – context, recurrence (time series changes in input and internal state), etc.

Discrimination with Deep Networks

- Discrimination approaches with DBNs (Deep Belief Net)
  - Use outputs of DBNs as inputs to supervised model (i.e. just an unsupervised preprocessor for feature extraction)
    - Basic approach we have been discussing
  - Train a DBN for each class. For each clamp the unknown x and iterate *m* times. The DBN that ends with the lowest normalized free energy (softmax variation) is the winner.
  - Train just one DBN for all classes, but with an additional visible unit for each class. For each output class:
    - Clamp the unknown x, relax, and then see which final state has lowest free energy – no need to normalize since all energies come from the same network.
- See http://deeplearning.net/demos/

Demo

- Variational Autoencoder Demos:
  - **Durk Kingma's MNIST demo**
  - **Vincent Dumolin's TFD demo**
- Stanford's Sentiment Analysis Demo using Recursive Neural Networks:
  - **Recursive Neural networks Sentiment Analysis**
- Here is a link for demo of the above and others
  - http://deeplearning.net/demos/

Summary

- Much recent excitement, still much to be discovered
- "Google-Brain"
- Sum of Products Nets
- Biological Plausibility
- Potential for significant improvements
- Good in structured/Markovian spaces
  - Important research question: To what extent can we use Deep Learning in more arbitrary feature spaces?
  - Recent deep training of MLPs with BP shows potential in this area