Deep Learning

Ravi Kothari, Ph.D.
ravi.kothari@ashoka.edu.in

"What a tangled web we weave" - Macbeth

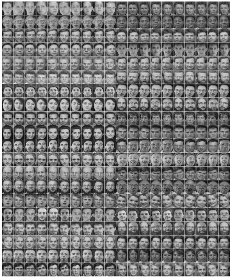
Learning from Examples

Learning from Examples

- As we have seen, in many situations, enough is not known to construct first-principles based models

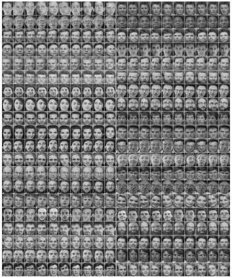
Learning from Examples

- As we have seen, in many situations, enough is not known to construct first-principles based models
- For example, designing a face recognition system using first-principles is very (very) hard



Learning from Examples

- As we have seen, in many situations, enough is not known to construct first-principles based models
- For example, designing a face recognition system using first-principles is very (very) hard



- *Empirical model construction* becomes very attractive

The Classical Approach

The Classical Approach

- Feature engineering

The Classical Approach

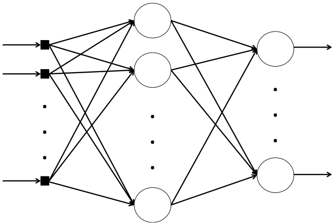
- Feature engineering
  - Based on our own (domain) knowledge, we define what *we* think would be a good set of features and construct a training data set in terms of those features (inputs) and desired output

The Classical Approach

- Feature engineering
  - Based on our own (domain) knowledge, we define what *we* think would be a good set of features and construct a training data set in terms of those features (inputs) and desired output
  - We then train an estimator (e.g. neural network) using the training data

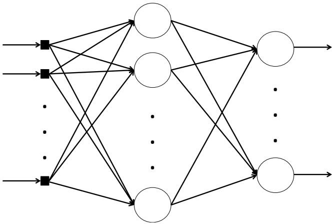
The Classical Approach

- Feature engineering
  - Based on our own (domain) knowledge, we define what *we* think would be a good set of features and construct a training data set in terms of those features (inputs) and desired output
  - We then train an estimator (e.g. neural network) using the training data
- Learning is largely a parameter (weight) adaptation task



The Classical Approach

- Feature engineering
  - Based on our own (domain) knowledge, we define what *we* think would be a good set of features and construct a training data set in terms of those features (inputs) and desired output
  - We then train an estimator (e.g. neural network) using the training data
- Learning is largely a parameter (weight) adaptation task
- Not (real) learning since the network is merely mimicking what we already know. What if we do not have the knowledge to know what are good features?



Representability

Representability

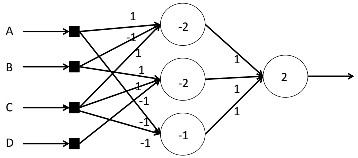
- Feed-forward networks with an arbitrary number of neurons in the hidden layer are dense in the space of continuous functions i.e. they are *universal approximators*

Representability

- Feed-forward networks with an arbitrary number of neurons in the hidden layer are dense in the space of continuous functions i.e. they are *universal approximators*
- They also have superior scaling characteristics (efficient representation). For example,

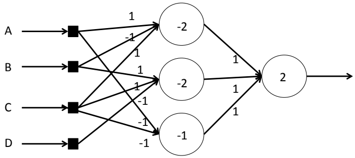
Representability

- Feed-forward networks with an arbitrary number of neurons in the hidden layer are dense in the space of continuous functions i.e. they are *universal approximators*
- They also have superior scaling characteristics (efficient representation). For example,



Representability

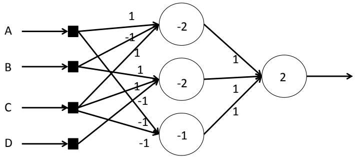
- Feed-forward networks with an arbitrary number of neurons in the hidden layer are dense in the space of continuous functions i.e. they are *universal approximators*
- They also have superior scaling characteristics (efficient representation). For example,



- Each hidden layer neuron realizes a DNF term i.e. $A\overline{B}C \cup BC\overline{D} \cup \overline{AC}$. Output layer realizes the sum of products

Representability

- Feed-forward networks with an arbitrary number of neurons in the hidden layer are dense in the space of continuous functions i.e. they are *universal approximators*
- They also have superior scaling characteristics (efficient representation). For example,



- Each hidden layer neuron realizes a DNF term i.e. $A\overline{B}C \cup BC\overline{D} \cup \overline{AC}$. Output layer realizes the sum of products
- $\#$ of neurons $=$ $\#$ of DNF terms $+ 1$. DNF is λ representable by FFNN's
- FFNN is not even π representable by DNF e.g. plurality

So, Why Deep Networks?

So, Why Deep Networks?

- Evolve features as part of the training process (avoid human based feature engineering)

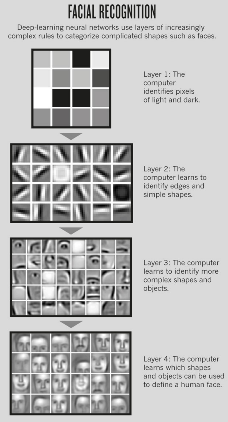
So, Why Deep Networks?

- Evolve features as part of the training process (avoid human based feature engineering)
- "Compositional" capability (may require fewer neurons) i.e. Feature reuse

So, Why Deep Networks?

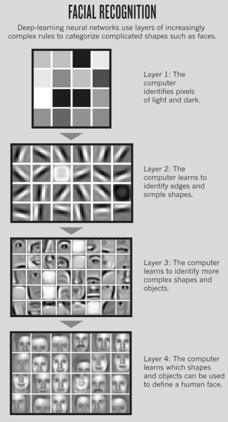
- Evolve features as part of the training process (avoid human based feature engineering)
- "Compositional" capability (may require fewer neurons) i.e. Feature reuse
- Availability of large data sets, parallel open source runtimes that leverage accelerators (e.g. GPUs) make it fesible

An Example



From N. Jones, "The Learning Machines", Nature 2014

An Example



From N. Jones, "The Learning Machines", Nature 2014

- Each successive layer learns more complex features e.g. edges in the early layer, then local shapes (composition of edges), then complex shapes (composition of local shapes)

Training Deep Networks

Training Deep Networks

- "Back propagating" is typically ineffective (vanishing gradient problem). See the use of Rectified Linear Units, normalization between layers and other "tricks" to get around this

Training Deep Networks

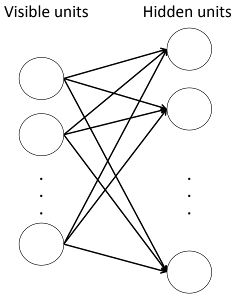
- "Back propagating" is typically ineffective (vanishing gradient problem). See the use of Rectified Linear Units, normalization between layers and other "tricks" to get around this
- Deeper networks typically are very likely to get trapped in a local minima

Training Deep Networks

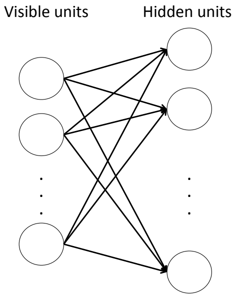
- "Back propagating" is typically ineffective (vanishing gradient problem). See the use of Rectified Linear Units, normalization between layers and other "tricks" to get around this
- Deeper networks typically are very likely to get trapped in a local minima
- Restricted Boltzman Machines based auto-encoders

Restricted Boltzman Machines (Optional)

Restricted Boltzman Machines (Optional)



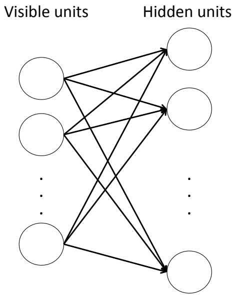
Restricted Boltzman Machines (Optional)



$$E(v,h) = -\sum_{i\in\text{visible}} a_i v_i - \sum_{j\in\text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

v_i, h_j are the binary states of visible unit i and hidden unit j, a_i and b_j are the biases and w_{ij} is the weight between them

Restricted Boltzman Machines (Optional)



$$E(v,h) = -\sum_{i \in \text{visible}} a_i v_i - \sum_{j \in \text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

v_i, h_j are the binary states of visible unit i and hidden unit j, a_i and b_j are the biases and w_{ij} is the weight between them

- Generative models

Training Restricted Boltzman Machines (1/2)

Training Restricted Boltzman Machines (1/2)

$$p(v, h) = \frac{1}{Z} e^{-E(v,h)}$$

Training Restricted Boltzman Machines (1/2)

$$p(v, h) = \frac{1}{Z} e^{-E(v,h)}$$

where, the partition function Z is given by,

$$Z = \sum_{v,h} e^{-E(v,h)}$$

Training Restricted Boltzman Machines (1/2)

$$p(v, h) = \frac{1}{Z} e^{-E(v,h)}$$

where, the partition function Z is given by,

$$Z = \sum_{v,h} e^{-E(v,h)}$$

The probability that the network assigns to a visible vector v is,

$$p(v) = \frac{1}{Z} \sum_{h} e^{-E(v,h)}$$

Training Restricted Boltzman Machines (2/2)

Training Restricted Boltzman Machines (2/2)

$$\frac{\partial \log p(v)}{\partial w_{ij}} = < v_i h_j >_{\text{data}} - < v_i h_j >_{\text{model}}$$

Training Restricted Boltzman Machines (2/2)

$$\frac{\partial \log p(v)}{\partial w_{ij}} = < v_i h_j >_{\mathrm{data}} - < v_i h_j >_{\mathrm{model}}$$

$$\Delta w_{ij} = \eta \left(\langle v_i h_j \rangle_{\mathrm{data}} - \langle v_i h_j \rangle_{\mathrm{model}} \right)$$

Training Restricted Boltzman Machines (2/2)

$$\frac{\partial \log p(v)}{\partial w_{ij}} =< v_i h_j >_{\mathrm{data}} - < v_i h_j >_{\mathrm{model}}$$

$$\Delta w_{ij} = \eta \left(\langle v_i h_j \rangle_{\mathrm{data}} - \langle v_i h_j \rangle_{\mathrm{model}} \right)$$

Contrastive divergence

Updating the States

Updating the States

$$p(h_j = 1) = \sigma(b_j + \sum_i v_i w_{ij})$$

Updating the States

$$p(h_j = 1) = \sigma(b_j + \sum_i v_i w_{ij})$$

$$p(v_i = 1) = \sigma(a_i + \sum_j h_j w_{ij})$$

Updating the States

Updating the States

$$s_i = \sum_j w_{ij} x_j$$

where, the sum runs over all units that unit i is connected to, w_{ij} is the weight between i and j and x_j is the state of the j^{th} unit (0 or 1).

Updating the States

$$s_i = \sum_j w_{ij} x_j$$

where, the sum runs over all units that unit i is connected to, w_{ij} is the weight between i and j and x_j is the state of the j^{th} unit (0 or 1).

$$p_i = \sigma(s_i)$$

Updating the States

$$s_i = \sum_j w_{ij} x_j$$

where, the sum runs over all units that unit i is connected to, w_{ij} is the weight between i and j and x_j is the state of the j^{th} unit (0 or 1).

$$p_i = \sigma(s_i)$$

So, we turn unit i on with probability p_i and off with probability $(1 - p_i)$. Obviously, units that are positively (negatively) connected tend to be in the same (different) state

Updating the Weights

Updating the Weights

- Apply a training pattern at the visible unit and update the hidden units i.e. $s_j = \sum_i W_{ij} v_i$

Updating the Weights

- Apply a training pattern at the visible unit and update the hidden units i.e. $s_j = \sum_i W_{ij} v_i$
- Set $h_j = 1$ with probability $\sigma(s_j)$ and 0 with probability $(1 - \sigma(s_j))$

Updating the Weights

- Apply a training pattern at the visible unit and update the hidden units i.e. $s_j = \sum_i W_{ij} v_i$
- Set $h_j = 1$ with probability $\sigma(s_j)$ and 0 with probability $(1 - \sigma(s_j))$
- For each edge e_{ij} compute $\mathrm{POS}(e_{ij})$ i.e. see if x_i and x_j are both on

Updating the Weights

- Apply a training pattern at the visible unit and update the hidden units i.e. $s_j = \sum_i W_{ij} v_i$
- Set $h_j = 1$ with probability $\sigma(s_j)$ and 0 with probability $(1 - \sigma(s_j))$
- For each edge e_{ij} compute $\mathrm{POS}(e_{ij})$ i.e. see if x_i and x_j are both on
- Reconstruct the visible units and compute $\mathrm{NEG}(e_{ij})$

Updating the Weights

- Apply a training pattern at the visible unit and update the hidden units i.e. $s_j = \sum_i W_{ij} v_i$
- Set $h_j = 1$ with probability $\sigma(s_j)$ and 0 with probability $(1 - \sigma(s_j))$
- For each edge e_{ij} compute $\mathrm{POS}(e_{ij})$ i.e. see if x_i and x_j are both on
- Reconstruct the visible units and compute $\mathrm{NEG}(e_{ij})$
- $w_{ij} = w_{ij} - \eta(\mathrm{POS}(e_{ij}) - \mathrm{NEG}(e_{ij}))$

An Example (1/2)

Users indicate their preference of 6 movies[1],

Alice: (Harry Potter = 1, Avatar = 1, LOTR 3 = 1, Gladiator = 0, Titanic = 0, Glitter = 0)
Bob: (Harry Potter = 1, Avatar = 0, LOTR 3 = 1, Gladiator = 0, Titanic = 0, Glitter = 0)
Carol: (Harry Potter = 1, Avatar = 1, LOTR 3 = 1, Gladiator = 0, Titanic = 0, Glitter = 0)
David: (Harry Potter = 0, Avatar = 0, LOTR 3 = 1, Gladiator = 1, Titanic = 1, Glitter = 0)
Eric: (Harry Potter = 0, Avatar = 0, LOTR 3 = 1, Gladiator = 1, Titanic = 1, Glitter = 0)
Fred: (Harry Potter = 0, Avatar = 0, LOTR 3 = 1, Gladiator = 1, Titanic = 1, Glitter = 0)

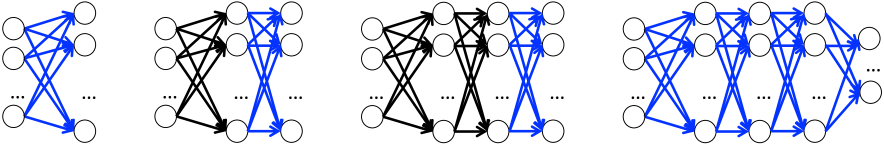
[1] Example from Edwin Chen

An Example (2/2)

	Bias Unit	Hidden 1	Hidden 2
Bias Unit	-0.08257658	-0.19041546	1.57007782
Harry Potter	-0.82602559	-7.08986885	4.96606654
Avatar	-1.84023877	-5.18354129	2.27197472
LOTR 3	3.92321075	2.51720193	4.11061383
Gladiator	0.10316995	6.74833901	-4.00505343
Titanic	-0.97646029	3.25474524	-5.59606865
Glitter	-4.44685751	-2.81563804	-2.91540988

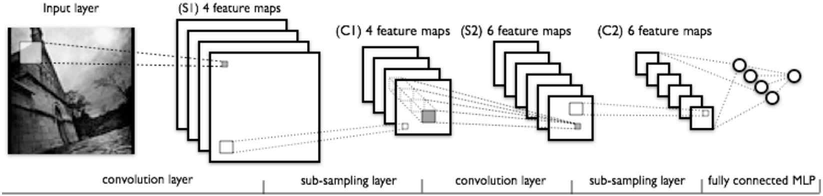
Hidden unit 1 seems to encode Oscar winners and Hidden unit 2 seems to encode for Sciecne Fantasy movies

Training Deep Networks

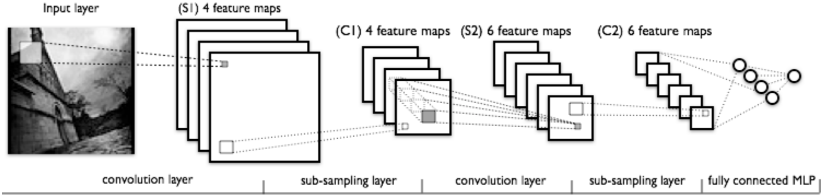


Convolution Networks

Convolution Networks



Convolution Networks



- Designed for inputs with a $2-D$ neighborhood e.g. images

Convolution Layer

Convolution Layer

Convolution Layer

- The convolution layer has multiple planes of neurons. Neurons in a plane "evolve" to detect something useful for the task at hand. A plane is also called a feature map. There are many feature maps i.e. many many features may be evolved

Convolution Layer

- The convolution layer has multiple planes of neurons. Neurons in a plane "evolve" to detect something useful for the task at hand. A plane is also called a feature map. There are many feature maps i.e. many many features may be evolved
- A neuron in a plane computes a "dot" product (weighted-sum) from the previous layer but only with a small portion (receptive field) of the previous layer e.g. in the first convolution layer, a node can compute the weighted sum of the inputs in a 3×3 or 5×5 path (receptive field or filter size)

Convolution Layer

- The convolution layer has multiple planes of neurons. Neurons in a plane "evolve" to detect something useful for the task at hand. A plane is also called a feature map. There are many feature maps i.e. many many features may be evolved
- A neuron in a plane computes a "dot" product (weighted-sum) from the previous layer but only with a small portion (receptive field) of the previous layer e.g. in the first convolution layer, a node can compute the weighted sum of the inputs in a 3×3 or 5×5 path (receptive field or filter size)
- Each neuron in a feature map does the exact same computation (though over different regions of he input) and has the same weights i.e. shared parameters

Convolution Layer

- The convolution layer has multiple planes of neurons. Neurons in a plane "evolve" to detect something useful for the task at hand. A plane is also called a feature map. There are many feature maps i.e. many many features may be evolved
- A neuron in a plane computes a "dot" product (weighted-sum) from the previous layer but only with a small portion (receptive field) of the previous layer e.g. in the first convolution layer, a node can compute the weighted sum of the inputs in a 3×3 or 5×5 path (receptive field or filter size)
- Each neuron in a feature map does the exact same computation (though over different regions of he input) and has the same weights i.e. shared parameters
- The output of each neuron is passed through an activation function like ReLU

Sub-Sampling (Pooling) Layer

Sub-Sampling (Pooling) Layer

- Sub-sampling layer is typically interleaved with the convolution layer

Sub-Sampling (Pooling) Layer

- Sub-sampling layer is typically interleaved with the convolution layer
- Sub-sampling replaces the value in a neighborhood by a representative value (such as the largest or the average). When the value is replaced by the maximum, it is also called Max-Pooling

Sub-Sampling (Pooling) Layer

- Sub-sampling layer is typically interleaved with the convolution layer
- Sub-sampling replaces the value in a neighborhood by a representative value (such as the largest or the average). When the value is replaced by the maximum, it is also called Max-Pooling
- Pooling reduces the spatial resolution. For example, a 3×3 pooling would reduce the spatial resolution by 9

Sub-Sampling (Pooling) Layer

- Sub-sampling layer is typically interleaved with the convolution layer
- Sub-sampling replaces the value in a neighborhood by a representative value (such as the largest or the average). When the value is replaced by the maximum, it is also called Max-Pooling
- Pooling reduces the spatial resolution. For example, a 3×3 pooling would reduce the spatial resolution by 9
- Note that the number of feature maps are still the same

Some Hyper-Parameters

Some Hyper-Parameters

- Number of layers, number of feature maps, size of the filters, activation function are all user chosen

Some Hyper-Parameters

- Number of layers, number of feature maps, size of the filters, activation function are all user chosen
- Drop-out percentage i.e. a specified % of neurons are "dropped out" of the network almost as if they never existed. Prevents overfitting

Some Hyper-Parameters

- Number of layers, number of feature maps, size of the filters, activation function are all user chosen
- Drop-out percentage i.e. a specified % of neurons are "dropped out" of the network almost as if they never existed. Prevents overfitting
- Stride allows the receptive field to overlap to different extents

Some Available Deep Learning Libraries

Name	Comments
Theano	Phython based
Tensorflow	From the Google Brain team
Caffe	From the Berkeley Vision and Learning Center
RNNLM	Mikolov's RNN based language models
deeplearning4j	Distributed NN library written in Java and Scala
Convnet	Multiple implementations e.g. OverFeat, Caffe etc.

We will use Keras which is capable of running on top of TensorFlow or Theano