Some of the slides in these lectures have been adapted/borrowed from materials developed by Tom Dietterich, Pedro Domingos, Tom Mitchell, David Page, and Jude Shavlik

Goals for the lecture

you should understand the following concepts

• backpropagation
• weight initialization
• momentum
• early stopping
• input encodings for neural networks
• output encodings
• recurrent neural networks
• the role of hidden units
• autoencoders
• restricted Boltzmann machines & deep belief networks
• rectified linear units
• dropout
Taking derivatives in neural nets

recall the chain rule from calculus

\[ y = f(u) \]
\[ u = g(x) \]

\[ \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x} \]

we'll make use of this as follows

\[ \frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial o} \frac{\partial o}{\partial \text{net}} \frac{\partial \text{net}}{\partial w_i} \]

\[ \text{net} = w_0 + \sum_{i=1}^{n} w_i x_i \]

Gradient descent: simple case

Consider a simple case of a network with one linear output unit and no hidden units:

\[ o^{(d)} = \text{net}^{(d)} = w_0 + \sum_{i=1}^{n} w_i x_{i}^{(d)} \]

let's learn \( w_i \)'s that minimize squared error

\[ E(w) = \frac{1}{2} \sum_{d \in D} \left( y^{(d)} - o^{(d)} \right)^2 \]

batch case

\[ \frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} \left( y^{(d)} - o^{(d)} \right)^2 \]

online case

\[ \frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \left( y^{(d)} - o^{(d)} \right)^2 \]
Stochastic gradient descent: simple case

let’s focus on the online case (stochastic gradient descent):

\[
\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial E^{(d)}}{\partial o^{(d)}} \frac{\partial o^{(d)}}{\partial \text{net}^{(d)}} \frac{\partial \text{net}^{(d)}}{\partial w_i}
\]

\[
\frac{\partial E^{(d)}}{\partial o^{(d)}} = -(y^{(d)} - o^{(d)})
\]

\[
\frac{\partial o^{(d)}}{\partial \text{net}^{(d)}} = 1
\]

(assuming linear output unit)

\[
\frac{\partial \text{net}^{(d)}}{\partial w_i} = x_i^{(d)}
\]

\[
\frac{\partial E^{(d)}}{\partial w_i} = -(y^{(d)} - o^{(d)})x_i^{(d)}
\]

Gradient descent with a sigmoid

Now let’s consider the case in which we have a sigmoid output unit and no hidden units:

\[\text{net}^{(d)} = w_0 + \sum_{i=1}^{n} w_i x_i^{(d)}\]

\[o^{(d)} = \frac{1}{1 + e^{-\text{net}^{(d)}}}\]

useful property:

\[
\frac{\partial o^{(d)}}{\partial \text{net}^{(d)}} = o^{(d)}(1 - o^{(d)})
\]
Stochastic GD with sigmoid output unit

\[
\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial E^{(d)}}{\partial o^{(d)}} \frac{\partial o^{(d)}}{\partial net^{(d)}} \frac{\partial net^{(d)}}{\partial w_i}
\]

\[
\frac{\partial E^{(d)}}{\partial o^{(d)}} = -(y^{(d)} - o^{(d)})
\]

\[
\frac{\partial o^{(d)}}{\partial net^{(d)}} = o^{(d)}(1-o^{(d)})
\]

\[
\frac{\partial net^{(d)}}{\partial w_i} = x_i^{(d)}
\]

\[
\frac{\partial E^{(d)}}{\partial w_i} = -(y^{(d)} - o^{(d)})o^{(d)}(1-o^{(d)})x_i^{(d)}
\]

Backpropagation

- now we’ve covered how to do gradient descent for single-layer networks with
  - linear output units
  - sigmoid output units

- how can we calculate \( \frac{\partial E}{\partial w_i} \) for every weight in a multilayer network?

  \( \Rightarrow \) backpropagate errors from the output units to the hidden units
Backpropagation notation

Let’s consider the online case, but drop the \((d)\) superscripts for simplicity.

We’ll use:

- Subscripts on \(y, o, \text{net}\) to indicate which unit they refer to.
- Subscripts to indicate the unit a weight emanates from and goes to.

\[
\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}
\]

\[
= -\eta \frac{\partial E}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}}
\]

\[
= \eta \delta_j o_i
\]

Where \(\delta_j = -\frac{\partial E}{\partial \text{net}_j}\)
Backpropagation

\[ \Delta w_{ji} = \eta \, \delta_j \, o_i \]

where \[ \delta_j = -\frac{\partial E}{\partial net_j} \]

\[ \delta_j = o_j (1 - o_j) (y_j - o_j) \quad \text{if } j \text{ is an output unit} \]

\[ \delta_j = o_j (1 - o_j) \sum_k \delta_k w_{kj} \quad \text{if } j \text{ is a hidden unit} \]

Backpropagation illustrated

1. calculate error of output units
   \[ \delta_j = o_j (1 - o_j) (y_j - o_j) \]

2. calculate error for hidden units
   \[ \delta_j = o_j (1 - o_j) \sum_k \delta_k w_{kj} \]
Backpropagation illustrated

3. determine updates for weights going to output units
   \[ \Delta w_{ji} = \eta \delta_j o_i \]

4. determine updates for weights to hidden units using hidden-unit errors
   \[ \Delta w_{ji} = \eta \delta_j o_i \]

Backpropagation

- particular derivatives depend on error and transfer functions
- here we show derivatives for squared error and sigmoid functions
- gradient descent and backprop generalize to other cases in which these functions are differentiable
Initializing weights

- Weights should be initialized to
  - small values so that the sigmoid activations are in the range where the derivative is large (learning will be quicker)
  - random values to ensure symmetry breaking (i.e. if all weights are the same, the hidden units will all represent the same thing)
  - typical initial weight range [-0.01, 0.01]

Setting the learning rate

convergence depends on having an appropriate learning rate
Learning rate and momentum

- sometimes a *momentum* term is added
  \[
  \Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1)
  \]
  - weight update on iteration \( t \)
  - momentum
  - weight update on iteration \( t-1 \)

- keeps weights moving in the same direction as the previous update
  - can help to avoid local minima
  - increases step size in flat regions, speeding convergence

Stopping criteria

- conventional gradient descent: train until local minimum reached

- empirically better approach: *early stopping*
  - use a validation set to monitor accuracy during training iterations
  - return the weights that result in minimum validation-set error

![Graph showing training and validation error over iterations, with a point indicating the stopping criterion.](image)
Input (feature) encoding for neural networks

nominal features are usually represented using a 1-of-\(k\) encoding

\[
A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad G = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}
\]

ordinal features can be represented using a thermometer encoding

\[
tiny = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad small = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad medium = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad large = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\]

real-valued features can be represented using individual input units (we may want to scale/normalize them first though)

\[\text{precipitation} = [0.68]\]

Output encoding for neural networks

regression tasks usually use output units with linear transfer functions

binary classification tasks usually use one sigmoid output unit

\[O_i = \frac{e^{net_i}}{\sum_{j \in \text{outputs}} e^{net_j}}\]

\(k\)-ary classification tasks usually use \(k\) sigmoid or softmax output units
Recurrent neural networks

Recurrent networks are sometimes used for tasks that involve making sequences of predictions

- Elman networks: recurrent connections go from hidden units to inputs
- Jordan networks: recurrent connections go from output units to inputs

The role of hidden units

- Hidden units transform the input space into a new space where perceptrons suffice
- They numerically represent "constructed" features
- Consider learning the target function using the network structure below:
The role of hidden units

- in this task, hidden units learn a compressed numerical coding of the inputs/outputs

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>.89 .04 .08</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>.01 .11 .88</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>.01 .97 .27</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>.99 .97 .71</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>.03 .05 .02</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>.22 .99 .99</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>.80 .01 .98</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>.60 .94 .01</td>
<td>00000001</td>
</tr>
</tbody>
</table>

Learning representations

- the feature representation provided is often the most significant factor in how well a learning system works

- an appealing aspect of multilayer neural networks is that they are able to change the feature representation

- can think of the nodes in the hidden layer as new features constructed from the original features in the input layer

[Figures from Lee et al., ICML 2009]
Backpropagation with multiple hidden layers

• in principle, backpropagation can be used to train arbitrarily deep networks (i.e. with multiple hidden layers)

• in practice, this doesn’t usually work well with sigmoid units
  • there are likely to be lots of local minima
  • diffusion of gradients leads to slow training in lower layers

• better ways of training deep networks (multiple hidden layers)
  1. pretraining: greedy layer-wise unsupervised learning
  2. backprop with rectified linear units and other tricks

Deep network approach 1: pretraining

1. Use unsupervised learning for greedy layer-wise training
   • allows abstractions to develop from one layer to the next
   • helps initialize network with good parameters
   • enables unlabeled data to be used for training!

2. Use supervised learning (gradient descent) to learn the last layer
   • … and often to refine the other layers
Pretraining approach 1a: Autoencoders

• one approach for pretraining: use autoencoders to learn hidden-unit representations
• in an autoencoder, the network is trained to reconstruct the inputs

![Autoencoder Diagram]

Autoencoder variants

• various approaches can be used to encourage the autoencoder to generalize

  • bottleneck: use fewer hidden units than inputs

  • sparsity: use a penalty function that encourages most hidden unit activations to be near 0 [Goodfellow et al. 2009]

  • denoising: train to predict true input from corrupted input [Vincent et al. 2008]

  • contractive: force encoder to have small derivatives [Rifai et al. 2011]
Stacking autoencoders

- autoencoders can be stacked to form highly nonlinear representations

Each $W^{(i)}$ here represents the matrix of weights between layers, and $h_i$ represents the $i^{th}$ layer of hidden units.

Fine tuning

- after completion, can run backpropagation on the entire network to fine-tune weights for the supervised task

- because this backpropagation starts with good weights, its credit assignment is better and the learned model is likely to be better than if we just ran backpropagation initially
Pretraining approach 1b: Restricted Boltzmann machines (RBMs)

- a type of undirected graphical model with a topology that connects hidden variables to “visible” variables, but with no edges between hidden or between visible variables

- the probability of a configuration of variable settings is given by

\[
P(x,h) = \frac{1}{Z} e^{\sum_i c_i x_i + \sum_j b_j h_j + \sum_{i,j} x_i h_j w_{ij}}
\]

where

\[
Z = \sum_{x,h} e^{\sum_i c_i x_i + \sum_j b_j h_j + \sum_{i,j} x_i h_j w_{ij}}
\]

Training an RBM

- Adjust weights to maximize likelihood of observed data

\[
\Delta w_{ji} = \eta \frac{\partial \log P(x)}{\partial w_{ji}}
\]

\[
= \eta \left( E_{\text{data}}[x_i h_j] - E_{\text{model}}[x_i h_j] \right)
\]
A fast approximate RBM training method

iteratively

- select \( x \)
- sample \( h \) from \( P(h \mid x) \) using
  \[
P(H_i = 1 \mid x) = \frac{1}{1 + e^{-c_i - \sum_j w_{ij} x_j}}
  \]
- “reconstruct” \( x \) using
  \[
P(X_i = 1 \mid h) = \frac{1}{1 + e^{-b_i - \sum_j w_{ij} h_j}}
  \]
- update weights after every 10-100 instances
  \[
  \Delta w_{ji} = \eta \left( E_{\text{data}} \left[ x_i h_j \right] - E_{\text{reconstruct}} \left[ x_i h_j \right] \right)
  \]

Training a deep belief network (DBN)

- train \( W^{(1)} \) as RBM using \( x \) as training data
- fix \( W^{(1)} \)
- sample \( h^{(1)} \) from \( P(h^{(1)} \mid x) \)
- train \( W^{(2)} \) as RBM using \( h^{(1)} \) as training data
- iterate to add more layers
- train last layer for supervised task
Deep network approach 2: direct supervised training

- direct supervised training of deep networks commonly uses a few techniques to avoid slow training and overfitting
  - rectified linear units (ReLUs) instead of sigmoids
  - dropout

Rectified linear units

- faster learning than sigmoids because gradients don’t vanish as \( x \) increases
- more efficient computation because exponential function is not used

\[ f(x) = \max(0, x) \]
Dropout

On each training iteration
– randomly “drop out” a subset of the units and their weights
– do forward and backprop on remaining network

Figures from Srivastava et al., Journal of Machine Learning Research 2014

Dropout

At test time
– use all units and weights in the network
– adjust weights according to the probability that the source unit was dropped out

Figures from Srivastava et al., Journal of Machine Learning Research 2014
Comments on neural networks

• stochastic gradient descent often works well for very large data sets
  • one pass (or a few passes) through the data set may be sufficient to learn a model
  • Google uses it a lot [Andrew Moore, former VP of Engineering]

• other continuous optimization methods (e.g. conjugate gradient) can be used for learning

• backpropagation generalizes to
  • arbitrary numbers of output and hidden units
  • arbitrary layers of hidden units
  • arbitrary connection patterns
  • other transfer (i.e. output) functions
  • other error measures