Goals for the lecture

you should understand the following concepts

- weight initialization
- momentum
- second-order methods
- regularization
- early stopping
- dropout
- input encodings for neural networks
- output encodings
- the role of hidden units
- autoencoders
- restricted Boltzmann machines & deep belief networks
- convolutional neural nets
- recurrent neural nets
Initializing weights

- For sigmoid/tanh units, weights should be initialized to
  - small values so that the 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

\[
\eta \text{ too small (error goes down a little)}
\]

\[
\eta \text{ too large (error goes up)}
\]
Learning rate and momentum

- sometimes a momentum term is added

\[
\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1)
\]

- 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

Second-order methods

- recall: a Taylor series is a representation of a function as an infinite sum of terms based on the function's derivatives at a single point
- key idea of second-order methods: better optimization by taking 2nd derivatives (curvature) of objective function into account

Taylor series approximation for \( e^x \) (from Wikipedia)

gradient descent

\[
\Delta w = -\eta \nabla E(w)
\]

general second order idea

\[
\Delta w = -\eta \nabla E(w)(\nabla^2 E(w))^{-1}
\]
Regularization

- Regularization refers to an approach for avoiding overfitting by biasing the learning process away from completely fitting the training data.

- E.g. decision-tree pruning is a regularization method.

- Some regularization methods for neural networks:
  - early stopping
  - dropout
  - L1 or L2 penalty terms (we’ll discuss these later in the semester)
  - expanding training data by creating new instances that are slightly perturbed versions of existing ones (e.g. rotating images)

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.

![Plot of training and validation error over training iterations. The plot shows a downward trend for both error types. The validation error starts to rise before the training error, indicating overfitting. The stopping criteria is marked by an arrow indicating the point where the validation error begins to rise and the training error is still decreasing. The plot is labeled as follows: Training error and Validation error. The x-axis represents training iterations, and the y-axis represents error. The graph shows that the training and validation error decrease gradually over time. When the validation error starts to increase, the training is stopped. The figure is titled as 'Training vs Validation Error' with an annotation 'Stop training here.' ]
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
Input (feature) encoding for neural networks

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

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

ordinal features can be represented using a thermometer encoding

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

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 activation functions

binary classification tasks usually use one sigmoid output unit

\[
o_{i} = \frac{e^{\text{net}_{i}}}{\sum_{j \in \text{outputs}} e^{\text{net}_{j}}}
\]

\[\text{k-ary classification tasks usually use k sigmoid or softmax output units}\]
Output encoding for neural networks (continued)

Multi-label classification tasks (i.e. multiple binary labels are predicted for each instance) usually use multiple sigmoid output units

Most appropriate objective function for each output encoding

**Regression:** squared error

\[
E(w) = \frac{1}{2} \sum_{d \in D} (Y^{(d)} - a^{(d)})^2
\]

**Binary classification, multi-label classification:** cross entropy

\[
E(w) = \sum_{d \in D} -y^{(d)} \ln(a^{(d)}) - (1 - y^{(d)}) \ln(1 - a^{(d)})
\]

**K-ary classification:** multiclass cross entropy

\[
E(w) = - \sum_{d \in D} \sum_{i=1}^{\# \text{classes}} y_i^{(d)} \ln(a_i^{(d)})
\]
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:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>

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 Values</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](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
  - 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, special architectures 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 aproach 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 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

train autoencoder to represent $x$

discard $W^{(1)}$

train autoencoder to represent $h_1$

discard $W^{(2)}$

train weights $W^{(3)}$ for supervised task

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

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

where

\[
Z = \sum_{x,h} e^{-\sum_i c_i x_i + \sum_j h_j + \sum_{ij} x_i h_j w_{ij}}
\]
Training an RBM

- Adjust weights to maximize likelihood of observed data

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

\[ = \eta \left( E_{data} \left[ x_i h_j \right] - E_{model} \left[ x_i h_j \right] \right) \]

A fast approximate RBM training method

iteratively
- select \( x \)
- sample \( h \) from \( P(h \mid x) \) using

\[ P(H_i = 1 \mid x) = \frac{1}{1 + e^{-\sum_j w_{ij} x_j}} \]

- "reconstruct" \( x \) using

\[ P(X_i = 1 \mid h) = \frac{1}{1 + e^{-\sum_j w_{ij} h_j}} \]

- update weights after every 10-100 instances

\[ \Delta w_{ji} = \eta \left( E_{data} \left[ x_i h_j \right] - E_{reconstruct} \left[ x_i h_j \right] \right) \]
Training a deep belief network (DBN)

- train $W^{(1)}$ as RBM using $x$ as training data
- fix $W^{(1)}$
- sample $h^{(1)}$ from $P(h^{(1)} | x)$
- 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
  - specialized architectures
Rectified linear units

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

$f(x) = \max(0, x)$

Convolutional neural nets

- well suited to tasks in which the input has spatial structure, such as images or sequences
- based on four key ideas
  - local receptive fields
  - weight sharing
  - pooling
  - multiple layers of hidden units
Convolutional neural nets

• suppose we have a task in which are instances are 28 × 28 pixel images
• we can represent each using 28 × 28 input units

Convolutional neural nets

• we can connect hidden units so that each has a local receptive field (e.g. a 5 × 5 patch of the image).
Convolutional neural nets

- we can have a set of these units that differ in their local receptive field
- all of the units share the same set of weights
- so the units detect same “feature” in the image, but at different locations

Convolutional neural nets

- a set of units that detect the same “feature” is called a feature map
- typically we’ll have multiple feature maps in each layer
Convolutional neural nets

- feature-map layers are typically alternated with pooling layers
- each unit in a pooling layer outputs a max, or similar function, of a subset of the units in the previous layer

\[ f(x) = \max(x_1 \ldots x_i \ldots) \]

\[ f(x) = \log \sum_i e^{x_i} \]

\[ f(x) = \sqrt{\sum_i x_i^2} \]
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

Comments on neural networks

- Deep networks have had much recent success due to a combination of tricks and factors
  - Rectified linear units to handle the diminishing gradients problem
  - Dropout to avoid overfitting
  - Sparsely connected architectures (e.g. convolutional networks) to incorporate task-specific bias
  - Very large data sets and hardware to enable training with them

- Stochastic gradient descent often works well for very large data sets even with simple models (i.e. no hidden units)
  - One pass (or a few passes) through the data set may be sufficient to learn a good model
Comments on neural networks

- gradient descent/backpropagation generalizes to
  - arbitrary numbers of output and hidden units
  - arbitrary layers of hidden units
  - arbitrary connection patterns
  - multiple activation functions
  - multiple objective functions