Announcements

• HW2 due Thursday

• Office hours on Thursday: 4:15pm-5:45pm

• Exam
  – Mean 53.04 (76%)
  – Median 56.50 (81%)
Plan for the next few lectures

Neural network basics
- Architecture
- Biological inspiration
- Loss functions
- Training with gradient descent and backpropagation

Practical matters
- Overfitting prevention
- Transfer learning
- Software packages

Convolutional neural networks (CNNs)
- Special operations for processing images

Recurrent neural networks (RNNs)
- Special operations for processing sequences (e.g. language)
Neural network definition

- Activations:
  \[ a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \]

- Nonlinear activation function \( h \) (e.g. sigmoid, tanh, RELU):
  \[ z_j = h(a_j) \]
  \[ \tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \]
Neural network definition

- Layer 2

\[ a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \]

- Layer 3 (final)

\[ a_k = \]

- Outputs

(binary) \[ y_k = \sigma(a_k) = \frac{1}{1 + \exp(-a_k)} \]

(multiclass) \[ y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \]

- Finally:

(binary) \[ y_k(x, w) = \sigma \left( \sum_{j=1}^{M} w_{kj}^{(2)} h \left( \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \]
Activation functions

**Sigmoid**

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

**tanh** \( \tanh(x) \)

**ReLU** \( \max(0, x) \)

**Leaky ReLU**

\[ \max(0.1x, x) \]

**Maxout**

\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**

\[ f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \]
A multi-layer neural network

- **Nonlinear** classifier
- Can approximate any continuous function to arbitrary accuracy given sufficiently many hidden units
Inspiration: Neuron cells

- Neurons
  - accept information from multiple inputs
  - transmit information to other neurons
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node
- If output of function over threshold, neuron “fires”
Multilayer networks

- Cascade neurons together
- Output from one layer is the input to the next
- Each layer has its own sets of weights
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

• Predictions are fed forward through the network to classify
Feed-forward networks

• Predictions are fed forward through the network to classify
Deep neural networks

- Lots of hidden layers
- Depth = power (usually)
How do we train them?

• There is no analytical solution for the weights
• We will iteratively find such a set of weights that allow the outputs to match the desired outputs
• We want to minimize a loss function (a function of the weights in the network)
• For now let’s simplify and assume there’s a single layer of weights in the network
Softmax loss

scores = unnormalized log probabilities of the classes

\[ P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \]

where \( s = f(x_i; W) \)

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

\[ L_i = - \log P(Y = y_i | X = x_i) \]

cat 3.2

car 5.1

dog -1.7
Softmax loss

\[ L_i = -\log \left( \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right) \]

unnormalized log probabilities

<table>
<thead>
<tr>
<th>cat</th>
<th>3.2</th>
<th>24.5</th>
<th>0.13</th>
<th>( L_i = -\log(0.13) = 0.89 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>5.1</td>
<td>164.0</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>0.18</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

unnormalized probabilities

Adapted from Andrej Karpathy
Regularization

- L1, L2 regularization (*weight decay*)
- Dropout
  - Randomly turn off some neurons
  - Allows individual neurons to independently be responsible for performance

Dropout: A simple way to prevent neural networks from overfitting [Srivastava JMLR 2014]

Adapted from Jia-bin Huang
Gradient descent

- We’ll update the weights
- Move in direction opposite to gradient:

\[ w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}) \]
Mini-batch gradient descent

• Rather than compute the gradient from the loss for all training examples, could only use some of the data for each gradient update

• We cycle through all the training examples multiple times; each time we’ve cycled through all of them once is called an ‘epoch’

• Allows faster training (e.g. on GPUs), parallelization

Figure from Andrej Karpathy
Gradient descent in multi-layer nets

• **How to update the weights at all layers?**
• **Answer:** backpropagation of error from higher layers to lower layers

Figure from Andrej Karpathy
How to compute gradient?

- In a neural network:

\[ a_j = \sum_i w_{ji} z_i \quad z_j = h(a_j) \]

- Gradient is:

\[ \frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i \]

- Denote the “errors” as:

\[ \delta_j = \frac{\partial E_n}{\partial a_j} \]

- Also:

\[ \frac{\partial a_j}{\partial w_{ji}} = z_i \]
Backpropagation: Error

- For output units (e.g. identity output, least squares loss):
  \[ \delta_k = y_k - t_k \]

- For hidden units:
  \[ \delta_j = \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \]

- Backprop formula:
  \[ \delta_j = h'(a_j) \sum_k w_{kj} \delta_k \]
Example (identity output function)

- Two layer network w/ tanh at hidden layer:

\[ h(a) \equiv \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \]

- Derivative:

\[ h'(a) = 1 - h(a)^2 \]

- Minimize:

\[ E_n = \frac{1}{2} \sum_{k=1}^{K} (y_k - t_k)^2 \]

- Forward propagation:

\[
\begin{align*}
    a_j &= \sum_{i=0}^{D} w_{ji}^{(1)} x_i \\
    z_j &= \tanh(a_j) \\
    y_k &= \sum_{j=0}^{M} w_{kj}^{(2)} z_j
\end{align*}
\]
Example (identity output function)

- Errors at output:
  \[ \delta_k = y_k - t_k \]

- Errors at hidden units:
  \[ \delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{k,j} \delta_k \]

- Derivatives wrt weights:
  \[ \frac{\partial E_n}{\partial w^{(1)}_{ji}} = \delta_j x_i, \quad \frac{\partial E_n}{\partial w^{(2)}_{kj}} = \delta_k z_j \]
Backpropagation: Graphic example

First calculate error of output units and use this to change the top layer of weights.

\[ \delta_k = y_k - t_k \]

Update weights into \( j \)

\[ \frac{\partial E}{\partial w_{kj}^{(2)}} = \delta_k z_j \]

\[ w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}) \]

Adapted from Ray Mooney, equations from Chris Bishop
Next calculate error for hidden units based on errors on the output units it feeds into.

\[
\delta_k = y_k - t_k
\]

\[
\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{k,j} \delta_k
\]
Finally update bottom layer of weights based on errors calculated for hidden units.

\[ \delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k \]

Update weights into \( i \)

\[ \frac{\partial E}{\partial w_{ji}^{(1)}} = \delta_j x_i \]

\[ w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}) \]

Adapted from Ray Mooney, equations from Chris Bishop
Comments on training algorithm

• Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
• However, in practice, does converge to low error for many large networks on real data.
• Thousands of epochs (epoch = network sees all training data once) may be required, hours or days to train.
• To avoid local-minima problems, run several trials starting with different random weights (random restarts), and take results of trial with lowest training set error.
• May be hard to set learning rate and to select number of hidden units and layers.
• Neural networks had fallen out of fashion in 90s, early 2000s; back with a new name and significantly improved performance (deep networks trained with dropout and lots of data).

Ray Mooney, Carlos Guestrin, Dhruv Batra
Over-training prevention

- Running too many epochs can result in over-fitting.

- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.

Adapted from Ray Mooney
Determining best number of hidden units

- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.

- Use internal cross-validation to empirically determine an optimal number of hidden units.

\[ 	ext{error} \]

\[ \# \text{hidden units} \]

on training data

on test data

Ray Mooney
Effect of number of neurons

3 hidden neurons

6 hidden neurons

20 hidden neurons

more neurons = more capacity
Effect of regularization

Do not use size of neural network as a regularizer. Use stronger regularization instead:

\[
\lambda = 0.001 \quad \lambda = 0.01 \quad \lambda = 0.1
\]

(you can play with this demo over at ConvNetJS: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)
Hidden unit interpretation

- Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.
- On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc.
- However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.
“You need a lot of data if you want to train/use deep nets”

Transfer Learning

Adapted from Andrej Karpathy
Transfer learning: Motivation

• The more weights you need to learn, the more data you need

• That’s why with a deeper network, you need more data for training than for a shallower network

• But: If you have sparse data, you can just train the last few layers of a deep net

Set these to the already learned weights from another network  
Learn these on your own task
Transfer learning

Source: e.g. classification of animals

1. Train on source (large dataset)

Target: e.g. classification of cars

2. Small dataset:

3. Medium dataset: finetuning

more data = retrain more of the network (or all of it)

Freeze these

Train this

Another option: use network as feature extractor, train SVM/LR on extracted features for target task

Adapted from Andrej Karpathy
Pre-training on ImageNet

- Have a source domain and target domain
- Train a network to classify ImageNet classes
  - Coarse classes and ones with fine distinctions (dog breeds)
- Remove last layers and train layers to replace them, that predict target classes

Oquab et al., “Learning and Transferring Mid-Level Image Representations…”, CVPR 2014
Transfer learning with CNNs is pervasive…

Object Detection
*Ren et al., “Faster R-CNN”, NIPS 2015*

Image Captioning

Adapted from Andrej Karpathy
Another soln for sparse data: Augmentation

Create *virtual* training samples

- Horizontal flip
- Random crop
- Color casting
- Geometric distortion

Deep Image [Wu et al. 2015]
Packages

Caffe and Caffe Model Zoo

Torch

Theano with Keras/Lasagne

MatConvNet

TensorFlow
Learning Resources

http://deeplearning.net/
http://cs231n.stanford.edu (CNNs, vision)
http://cs224d.stanford.edu/ (RNNs, language)
Summary

• Feed-forward network architecture

• Training deep neural nets
  • We need an objective function that measures and guides us towards good performance
  • We need a way to minimize the loss function: (stochastic, mini-batch) gradient descent
  • We need backpropagation to propagate error towards all layers and change weights at those layers

• Practices for preventing overfitting, training with little data