CSC2515 Winter 2015
Introduction to Machine Learning

Lecture 4: Neural Networks

All lecture slides will be available as .pdf on the course website:
http://www.cs.toronto.edu/~urtasun/courses/CSC2515/
CSC2515_Winter15.html
Limitations of linear classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features $x_i$

- Many decisions involve non-linear functions of the input

- Canonical example: do 2 input elements have the same value?
  
  Same: $(1,1) \rightarrow 1; (0,0) \rightarrow 1$
  
  Diff: $(1,0) \rightarrow 0; (0,1) \rightarrow 0$

The positive and negative cases cannot be separated by a plane
How to construct nonlinear classifiers?

- Would like to construct non-linear discriminative classifiers that utilize functions of input variables

- Two approaches:
  - Add infinite number of extra functions
    - Need to address over-fitting
  - Add finite but large number of extra functions
    - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then the optimization still involves linear combinations of (fixed functions of) the inputs
    - Or we can make these functions depend on additional parameters → need an efficient method of training extra parameters
Neural networks

- Many machine learning methods inspired by biology, brains
- Our brains contain $\sim 10^{11}$ neurons, each of which communicates to $\sim 10^4$ other neurons
- Multi-layer perceptron, or neural network, is a popular supervised learning approach
- Defines extra functions of the inputs (hidden features), computed by neurons
- Artificial neurons called units
- Network output is linear combination of hidden units
Neural network architecture

- Network with one layer of four hidden units:

- Each unit computes value based on linear combination of values of units that point into it
- Can add more layers of hidden units: deeper hidden unit response depends on earlier hiddens
What does the network compute?

- Output of network can be written as follows, with k indexing the two output units:

\[
o_k(x) = g(w_{k0} + \sum_{j=1}^{J} h_j(x)w_{kj})
\]

\[
h_j(x) = f(w_{j0} + \sum_{i=1}^{D} x_i v_{ji})
\]

- Network with non-linear activation function \(f()\) is a universal approximator (esp. with increasing \(J\))

- Standard choice of activation function: sigmoid/logistic, or tanh, or rectified linear (relu)

\[
tanh(z) = (\exp(z) - \exp(-z))/(\exp(z) + \exp(-z))
\]

\[
relu(z) = \max(0,z)
\]
Example application

- Consider trying to classify image of handwritten digit: 32x32 pixels
- Single output units – it is a 4 (one vs. all)?
- Use the sigmoid output function:

\[
O_k = \frac{1}{1 + \exp(-z_k)}
\]

\[
z_k = (w_{k0} + \sum_{j=1}^{J} h_j(x)v_{kj})
\]

- Can train the network, that is, adjust all the parameters \(w\), to optimize the training objective, but this is a complicated function of the parameters
Training multi-layer networks: back-propagation

**Back-propagation**: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network.

Loop until convergence:

- For each example $n$
  1) Input $x^{(n)}$, propagate activity forward ($x^{(n)} \rightarrow h^{(n)} \rightarrow o^{(n)}$)
  2) Propagate gradients backward
  3) Update each weight (via gradient descent)

Given any error function $E$, activation functions $g()$ and $f()$, just need to derive gradients
Key idea behind backpropagation

We don’t have targets for a hidden unit, but we can compute how fast the error changes as we change its activity

- Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities.
- Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.
- We can compute error derivatives for all the hidden units efficiently.
- Once we have the error derivatives for the hidden activities, it’s easy to get the error derivatives for the weights going into a hidden unit.
Non-linear neurons with smooth derivatives

- For backpropagation, we need neurons (units) that have well-behaved derivatives.
  - Typically they use the logistic function
  - The output is a smooth function of the inputs and the weights.

\[
x_j = b_j + \sum_i y_i w_{ij}
\]

\[
y_j = \frac{1}{1 + e^{-x_j}}
\]

\[
\frac{\partial x_j}{\partial w_{ij}} = y_i \quad \frac{\partial x_j}{\partial y_i} = w_{ij}
\]

\[
\frac{dy_j}{dx_j} = y_j (1 - y_j)
\]

It's odd to express it in terms of \(y\).
Back-propagation: sketch on one training case

1. Convert discrepancy between each output and its target value into an error derivative.

2. Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at $k$ to each unit $j$ according to its influence on $k$ (depends on $w_{kj}$)]

3. Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

$$E = \frac{1}{2} \sum_k (o_k - t_k)^2$$

$$\frac{\partial E}{\partial o_k} = o_k - t_k$$
The derivatives

\[
\frac{\partial E}{\partial x_j} = \frac{dy_j}{dx_j} \frac{\partial E}{\partial y_j} = y_j (1 - y_j) \frac{\partial E}{\partial y_j}
\]

\[
\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j} = y_i \frac{\partial E}{\partial x_j}
\]

\[
\frac{\partial E}{\partial y_i} = \sum_j \frac{dx_j}{dy_i} \frac{\partial E}{\partial x_j} = \sum_j w_{ij} \frac{\partial E}{\partial x_j}
\]
Ways to use weight derivatives

- How often to update
  - after each training case?
  - after a full sweep through the training data?
  - after a “mini-batch” of training cases?

\[ w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_{k}^{(n)} - t_{k}^{(n)})o_{k}^{(n)}(1 - o_{k}^{(n)})x_{i}^{(n)} \]

- How much to update
  - Use a fixed learning rate?
  - Adapt the learning rate?
  - Add momentum?

\[ w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} + \beta \Delta w_{ki}(t - 1) \]
Choosing activation and cost functions

When using a neural network as a function approximator (regressor) sigmoid activation and MSE work well.

For classification, if it is a binary (2-class) problem, then cross-entropy error function often does better (as we saw with logistic regression).

\[
E = - \sum_{n} t^{(n)} \log o^{(n)} + (1 - t^{(n)}) \log(1 - o^{(n)})
\]

\[
o^{(n)} = \frac{1}{1 + \exp(-z^{(n)})}
\]

\[
\frac{\partial E}{\partial o} = o - t
\]

\[
\frac{\partial o}{\partial z} = o(1 - o)
\]

\[
\frac{\partial E}{\partial z} = \frac{\partial E}{\partial o} \frac{\partial o}{\partial z} = (o - t)o(1 - o)
\]
Some Success Stories

• Back-propagation has been used for a large number of practical applications.
  – Recognizing hand-written characters
  – Recognize speech
  – Predicting the next word in a sentence from the previous words
  – Autonomous vehicle control
  – Recognizing objects in images
A basic problem in speech recognition

- We cannot identify phonemes perfectly in noisy speech
  - The acoustic input is often ambiguous: there are several different words that fit the acoustic signal equally well.

- People use their understanding of the meaning of the utterance to hear the right word.
  - We do this unconsciously
  - We are very good at it

- This means speech recognizers have to know which words are likely to come next and which are not.
  - Can this be done without full understanding?
The standard “trigram” method

- Take a huge amount of text and count the frequencies of all triples of words. Then use these frequencies to make bets on the next word in a b?

\[
p(w_3 = c \mid w_2 = b, w_1 = a) = \frac{\text{count}(abc)}{\text{count}(abd)}
\]

\[
p(w_3 = d \mid w_2 = b, w_1 = a) = \frac{\text{count}(abd)}{\text{count}(abd)}
\]

- Until very recently this was state-of-the-art.
  - We cannot use a bigger context because there are too many quadgrams
  - We have to “back-off” to bigrams when the count for a trigram is zero.

  - The probability is not zero just because we didn’t see one.
Why the trigram model is limited

• Suppose we have seen the sentence
  “the cat got squashed in the garden on friday”
• This should help us predict words in the sentence
  “the dog got flattened in the yard on monday”
• A trigram model does not understand the similarities between
  – cat/dog  squashed/flattened  garden/yard  friday/monday
• To overcome this limitation, we need to use the features of
  previous words to predict features of the next word
  – Using a feature representation and a learned model of how
    past features predict future ones, we can use many more
    words from the past history.
Softmax

Handling multiple classes: the output units use a non-local non-linearity:

The cost function is the negative log prob of the right answer

\[ y_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \]

\[ \frac{\partial y_i}{\partial x_i} = y_i (1 - y_i) \]

\[ C = - \sum_j d_j \log y_j \]

\[ \frac{\partial C}{\partial x_i} = \sum_j \frac{\partial C}{\partial y_j} \frac{\partial y_j}{\partial x_i} = y_i - d_i \]
A neural net for predicting the next word

Softmax units (one per possible word)

Units that learn to predict the output word from features of the input words

Learned distributed encoding of word $t-2$

Learned distributed encoding of word $t-1$

Index of word at $t-2$ ← inputs → Index of word at $t-1$

Table look-up

Skip-layer connections

output
An alternative architecture

A single output unit that gives a score for the candidate word in this context.

Use the scores from all candidate words in a softmax to get error derivatives that try to raise the score of the correct candidate and lower the score of its high-scoring rivals.

Units that discover good or bad combinations of features

Learned distributed encoding of word t-2

Learned distributed encoding of word t-1

Learned distributed encoding of candidate

Index of word at t-2

Index of word at t-1

Index of candidate

Try all candidate words one at a time.
Applying backpropagation to shape recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it

- Some reasons why it is difficult:
  - Segmentation: Real scenes are cluttered
  - Invariances: We are very good at ignoring all sorts of variations that do not affect shape
  - Deformations: Natural shape classes allow variations (faces, letters, chairs)
  - A huge amount of computation is required
The invariance problem

• Our perceptual systems are very good at dealing with invariances
  – translation, rotation, scaling
  – deformation, contrast, lighting, rate

• We are so good at this that it’s hard to appreciate how difficult it is
  – Its one of the main difficulties in making computers perceive
  – We still don’t have generally accepted solutions
The replicated feature approach

- Adopt approach apparently used in monkey visual systems
- Use many different copies of the same feature detector.
  - Copies have slightly different positions.
  - Could also replicate across scale and orientation.
    - Tricky and expensive
  - Replication reduces number of free parameters to be learned.
- Use several different feature types, each with its own replicated pool of detectors.
  - Allows each patch of image to be represented in several ways.

The red connections all have the same weight.
Backpropagation with weight constraints

- It is easy to modify the backpropagation algorithm to incorporate linear constraints between the weights.

- We compute the gradients as usual, and then modify the gradients so that they satisfy the constraints.
  - So if the weights started off satisfying the constraints, they will continue to satisfy them.

\[ \text{To constrain: } w_1 = w_2 \]
\[ \text{we need: } \Delta w_1 = \Delta w_2 \]

\[ \text{compute: } \frac{\partial E}{\partial w_1} \quad \text{and} \quad \frac{\partial E}{\partial w_2} \]

\[ \text{use } \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2} \quad \text{for } w_1 \text{ and } w_2 \]
Yann LeCun and others developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
- Many hidden layers
- Many pools of replicated units in each layer.
- Averaging the outputs of nearby replicated units.
- A wide net that can cope with several characters at once even if they overlap.

Demos of LENET at http://yann.lecun.com
Recognizing Digits

Hand-written digit recognition network
- 7291 training examples, 2007 test examples
- Both contain ambiguous and misclassified examples
- Input pre-processed (segmented, normalized)
  - 16x16 gray level [-1,1], 10 outputs
LeNet: Summary

Main ideas:
- Local $\rightarrow$ global processing
- Retain coarse posn info

Main technique: weight sharing – units arranged in feature maps

Connections: 1256 units, 64,660 cxns, 9760 free parameters

Results: 0.14% (train), 5.0% (test)

vs. 3-layer net w/ 40 hidden units:
1.6% (train), 8.1% (test)
The 82 errors made by LeNet5

Notice that most of the errors are cases that people find quite easy.

The human error rate is probably 20 to 30 errors.
A brute force approach

- LeNet uses knowledge about the invariances to design:
  - the network architecture
  - or the weight constraints
  - or the types of feature

- But it's much simpler to incorporate knowledge of invariances by just creating extra training data:
  - for each training image, produce new training data by applying all of the transformations we want to be insensitive to
  - Then train a large, dumb net on a fast computer.
  - This works surprisingly well
Fabricating training data

Good generalization requires lots of training data, including examples from all relevant input regions. Improve solution if good data can be constructed. Example: ALVINN.
ALVINN: simulating training examples

On-the-fly training: current video camera image as input, current steering direction as target
But: over-train on same inputs; no experience going off-road
Method: generate new examples by shifting images

Replace 10 low-error & 5 random training examples with 15 new
Key: relation between input and output known!
Making backpropagation work for recognizing digits

- Using the standard viewing transformations, and local deformation fields to get lots of data.

- Use many, globally connected hidden layers and learn for a very long time
  - This requires a GPU board or a large cluster

- Use the appropriate error measure for multi-class categorization
  - Cross-entropy, with softmax activation

- This approach can get 35 errors on MNIST!
Fig. 6. Training and test errors of LeNet-5 achieved using training sets of various sizes. This graph suggests that a larger training set could improve the performance of LeNet-5. The hollow square show the test error when more training patterns are artificially generated using random distortions. The test patterns are not distorted.
Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains noise
  - The target values may be unreliable.
  - There is **sampling error**. There will be accidental regularities just because of the particular training cases that were chosen

- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity.
  - If the model is very flexible it can model the sampling error really well. **This is a disaster.**
Preventing overfitting

• Use a model that has the right capacity:
  – enough to model the true regularities
  – not enough to also model the spurious regularities (assuming they are weaker)

• Standard ways to limit the capacity of a neural net:
  – Limit the number of hidden units.
  – Limit the size of the weights.
  – Stop the learning before it has time to overfit.
Limiting the size of the weights

Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.

- Keeps weights small unless they have big error derivatives.

\[
C = E + \frac{\lambda}{2} \sum_i w_i^2
\]

\[
\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i
\]

\[
\text{when } \frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda} \frac{\partial E}{\partial w_i}
\]
The effect of weight-decay

• It prevents the network from using weights that it does not need
  – This can often improve generalization a lot.
  – It helps to stop it from fitting the sampling error.
  – It makes a smoother model in which the output changes more slowly as the input changes.

• But, if the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one.→ other form of weight decay?

\[ \begin{align*}
  & \quad \text{w/2} \\
  \quad \text{w/2} & \quad \text{w} \\
  \quad \text{0} & \quad \text{other form of weight decay?}
\end{align*} \]
Deciding how much to restrict the capacity

• How do we decide which limit to use and how strong to make the limit?
  – If we use the test data we get an unfair prediction of the error rate we would get on new test data.
  – Suppose we compared a set of models that gave random results, the best one on a particular dataset would do better than chance. But it won’t do better than chance on another test set.

• So use a separate validation set to do model selection.
Using a validation set

• Divide the total dataset into three subsets:
  – Training data is used for learning the parameters of the model.
  – Validation data is not used for learning but is used for deciding what type of model and what amount of regularization works best.
  – Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.

• We could then re-divide the total dataset to get another unbiased estimate of the true error rate.
Preventing overfitting by early stopping

• If we have lots of data and a big model, it's very expensive to keep re-training it with different amounts of weight decay.

• It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse.

• The capacity of the model is limited because the weights have not had time to grow big.
Why early stopping works

• When the weights are very small, every hidden unit is in its linear range.
  – So a net with a large layer of hidden units is linear.
  – It has no more capacity than a linear net in which the inputs are directly connected to the outputs!

• As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.