Lecture 5: Project Information + Neural Networks & Backprop

Richard Socher
richard@metamind.io
Overview Today:

- Organizational Stuff
- Project Tips
- From one-layer to multi layer neural network!
- Max-Margin loss and backprop!
  (This is the hardest lecture of the quarter)
Announcement:

- 1% extra credit for Piazza participation!
- Hint for PSet1: Understand math and dimensionality, then add print statements, e.g.

```python
def softmaxCostAndGradient(predicted, target, outputVectors, dataset):
    """ Softmax cost function for word2vec models """

    ### YOUR CODE HERE

    print "v_hat", predicted.shape
    print "expected", target
    print "U", outputVectors.shape

    assert False
```

- Student survey sent out last night, please give us feedback to improve the class :)
Class Project

- Most important (40%) and lasting result of the class
- PSet 3 a little easier to have more time
- Start early and clearly define your task and dataset

- Project types:
  1. Apply existing neural network model to a new task
  2. Implement a complex neural architecture
  3. Come up with a new neural network model
  4. Theory
Class Project: Apply Existing NNets to Tasks

1. Define Task:
   - Example: **Summarization**

2. Define Dataset
   1. Search for academic datasets
      - They already have baselines
      - E.g.: Document Understanding Conference (DUC)
   2. Define your own (harder, need more new baselines)
      - If you’re a graduate student: connect to your research
      - Summarization, Wikipedia: Intro paragraph and rest of large article
      - Be creative: Twitter, Blogs, News
Class Project: Apply Existing NNets to Tasks

3. Define your metric
   • Search online for well established metrics on this task
   • Summarization: Rouge (Recall-Oriented Understudy for Gisting Evaluation) which defines n-gram overlap to human summaries

4. Split your dataset!
   • Train/Dev/Test
   • Academic dataset often come pre-split
   • Don’t look at the test split until ~1 week before deadline!
Class Project: Apply Existing NNets to Tasks

5. Establish a baseline
   - Implement the simplest model (often logistic regression on unigrams and bigrams) first
   - Compute metrics on train AND dev
   - Analyze errors
   - If metrics are amazing and no errors: done, problem was too easy, restart :)

6. Implement existing neural net model
   - Compute metric on train and dev
   - Analyze output and errors
   - Minimum bar for this class
Class Project: Apply Existing NNets to Tasks

7. Always be close to your data!
   - Visualize the dataset
   - Collect summary statistics
   - Look at errors
   - Analyze how different hyperparameters affect performance

8. Try out different model variants
   - Soon you will have more options
     - Word vector averaging model (neural bag of words)
     - Fixed window neural model
     - Recurrent neural network
     - Recursive neural network
     - Convolutional neural network
Class Project: A New Model -- Advanced Option

- Do all other steps first (Start early!)
- Gain intuition of why existing models are flawed

- Talk to other researchers, come to my office hours a lot
- Implement new models and iterate quickly over ideas
- Set up efficient experimental framework
- Build simpler new models first
- Example Summarization:
  - Average word vectors per paragraph, then greedy search
  - Implement language model or autoencoder (introduced later)
  - Stretch goal for potential paper: Generate summary!
Project Ideas

- **Summarization**

- **NER, like PSet 2 but with larger data**


- **Image to text mapping or generation**, Grounded Compositional Semantics for Finding and Describing Images with Sentences, Richard Socher, Andrej Karpathy, Quoc V. Le, Christopher D. Manning, Andrew Y. Ng. (TACL 2014) or
  Deep Visual-Semantic Alignments for Generating Image Descriptions, Andrej Karpathy, Li Fei-Fei

- **Entity level sentiment**

Default project: sentiment classification

- Sentiment on movie reviews: [http://nlp.stanford.edu/sentiment/](http://nlp.stanford.edu/sentiment/)
- Lots of deep learning baselines and methods have been tried
A more powerful window classifier

- Revisiting

- \( X_{\text{window}} = [x_{\text{museums}} \ x_{\text{in}} \ x_{\text{Paris}} \ x_{\text{are}} \ x_{\text{amazing}}] \)

- Assume we want to classify whether the center word is a location or not
A Single Layer Neural Network

- A single layer is a combination of a linear layer and a nonlinearity:
  \[ z = Wx + b \]
  \[ a = f(z) \]

- The neural activations \( a \) can then be used to compute some function

- For instance, an unnormalized score or a softmax probability we care about:
  \[ \text{score}(x) = U^T a \in \mathbb{R} \]
Summary: Feed-forward Computation

Computing a window’s score with a 3-layer neural net: \( s = \text{score}(\text{museums in Paris are amazing}) \)

\[
s = U^T f(Wx + b) \quad x \in \mathbb{R}^{20 \times 1}, \quad W \in \mathbb{R}^{8 \times 20}, \quad U \in \mathbb{R}^{8 \times 1}
\]

\[
s = U^T a \\
\]
\[
a = f(z) \\
\]
\[
z = Wx + b \\
\]

\( X_{\text{window}} = [x_{\text{museums}}, x_{\text{in}}, x_{\text{Paris}}, x_{\text{are}}, x_{\text{amazing}}] \)
Main intuition for extra layer

The layer learns non-linear interactions between the input word vectors.

Example: only if “museums” is first vector should it matter that “in” is in the second position

\[ x_{\text{window}} = [x_{\text{museums}} \ x_{\text{in}} \ x_{\text{Paris}} \ x_{\text{are}} \ x_{\text{amazing}}] \]
Summary: Feed-forward Computation

- $s = \text{score}(\text{museums in Paris are amazing})$
- $s_c = \text{score}(\text{Not all museums in Paris})$

- Idea for training objective: make score of true window larger and corrupt window’s score lower (until they’re good enough): minimize

\[ J = \max(0, 1 - s + s_c) \]

- This is continuous, can perform SGD
Max-margin Objective function

- Objective for a single window:

\[ J = \max(0, 1 - s + s_c) \]

- Each window with a location at its center should have a score +1 higher than any window without a location at its center

- For full objective function: Sum over all training windows
Training with Backpropagation

\[
J = \max(0, 1 - s + s_c)
\]

\[
s = U^T f(Wx + b)
\]

\[
s_c = U^T f(Wx_c + b)
\]

Assuming cost \( J \) is > 0,
compute the derivatives of \( s \) and \( s_c \) wrt all the involved variables: \( U, W, b, x \)

\[
\frac{\partial s}{\partial U} = \frac{\partial}{\partial U} U^T a \quad \frac{\partial s}{\partial U} = a
\]
Training with Backpropagation

- Let’s consider the derivative of a single weight $W_{ij}$

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

- This only appears inside $a_i$

- For example: $W_{23}$ is only used to compute $a_2$
Training with Backpropagation

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

Derivative of weight $W_{ij}$:

$$\frac{\partial}{\partial W_{ij}} U^T a \rightarrow \frac{\partial}{\partial W_{ij}} U_i a_i$$

$$U_i \frac{\partial}{\partial W_{ij}} a_i = \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial W_{ij}}$$

$$= U_i \frac{\partial f(z_i)}{\partial z_i} \cdot \frac{\partial z_i}{\partial W_{ij}}$$

$$= U_i f'(z_i) \cdot \frac{\partial z_i}{\partial W_{ij}}$$

$$= U_i f'(z_i) \frac{\partial (W_i \cdot x + b_i)}{\partial W_{ij}}$$

$$z_i = W_i \cdot x + b_i = \sum_{j=1}^{3} W_{ij} \cdot x_j + b_i$$

$$a_i = f(z_i)$$

![Diagram of a neural network with labels $x_1$, $x_2$, $x_3$, $+1$, $s$, $a_1$, $a_2$, $U_2$, and $W_{23}$ connections.]}
Training with Backpropagation

Derivative of single weight $W_{ij}$:

$$U_i \frac{\partial}{\partial W_{ij}} a_i = U_i f'(z_i) \frac{\partial W_i x + b_i}{\partial W_{ij}}$$

$$= U_i f'(z_i) \frac{\partial}{\partial W_{ij}} \sum_k W_{ik} x_k$$

$$= U_i f'(z_i) x_j$$

$$= \delta_i x_j$$

where $f'(z) = f(z)(1 - f(z))$ for logistic $f$
Training with Backpropagation

- From single weight $W_{ij}$ to full $W$:

$$\frac{\partial s}{\partial W_{ij}} = U_i f'(z_i) x_j = \delta_i x_j$$

$$z_i = W_i.x + b_i = \sum_{j=1}^{3} W_{ij}x_j + b_i$$

$$a_i = f(z_i)$$

- We want all combinations of $i = 1, 2$ and $j = 1, 2, 3 \rightarrow \oplus$

- Solution: Outer product: $\frac{\partial J}{\partial W} = \delta x^T$

where $\delta \in \mathbb{R}^{2 \times 1}$ is the “responsibility” or error message coming from each activation $a$
• For biases $b$, we get:

\[ z_i = W_i \cdot x + b_i = \sum_{j=1}^{3} W_{ij} x_j + b_i \]

\[ a_i = f(z_i) \]

\[
U_i \frac{\partial}{\partial b_i} a_i \\
= U_i f'(z_i) \frac{\partial W_i \cdot x + b_i}{\partial b_i} \\
= \delta_i
\]
Training with Backpropagation

That’s almost backpropagation

It’s simply taking derivatives and using the chain rule!

Remaining trick: we can re-use derivatives computed for higher layers in computing derivatives for lower layers!

Example: last derivatives of model, the word vectors in $x$
Training with Backpropagation

• Take derivative of score with respect to single element of word vector

\[
\frac{\partial s}{\partial x_j} = \sum_{i=1}^{2} \frac{\partial s}{\partial a_i} \frac{\partial a_i}{\partial x_j}
\]

= \sum_{i=1}^{2} \frac{\partial U^T a}{\partial a_i} \frac{\partial a_i}{\partial x_j}

= \sum_{i=1}^{2} U_i \frac{\partial f(W_i x + b)}{\partial x_j}

= \sum_{i=1}^{2} U_i f'(W_i x + b) \frac{\partial W_i x}{\partial x_j}

= \sum_{i=1}^{2} \delta_i W_{ij}

= W_j^T \delta

Re-used part of previous derivative
Training with Backpropagation

• With \( \frac{\partial s}{\partial x_j} = W^T \delta \), what is the full gradient? →

\[
\frac{\partial s}{\partial x} = W^T \delta
\]

• Observations: The error message \( \pm \) that arrives at a hidden layer has the same dimensionality as that hidden layer
Putting all gradients together:

• Remember: Full objective function for each window was:

\[
J = \max(0, 1 - s + s_c)
\]

\[
s = U^T f(Wx + b)
\]

\[
s_c = U^T f(Wx_c + b)
\]

• For example: gradient for U:

\[
\frac{\partial s}{\partial U} = 1\{1 - s + s_c > 0\} \left(-f(Wx + b) + f(Wx_c + b)\right)
\]

\[
\frac{\partial s}{\partial U} = 1\{1 - s + s_c > 0\} \left(-a + a_c\right)
\]
Two layer neural nets and full backprop

- Let’s look at a 2 layer neural network
- Same window definition for $x$
- Same scoring function
- 2 hidden layers (carefully not superscripts now!)

\[
\begin{align*}
x &= z^{(1)} = a^{(1)} \\
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
a^{(3)} &= f(z^{(3)}) \\
s &= U^T a^{(3)}
\end{align*}
\]
Two layer neural nets and full backprop

- Fully written out as one function:

\[
\begin{align*}
    s &= U^T f \left( W^{(2)} f \left( W^{(1)} x + b^{(1)} \right) + b^{(2)} \right) \\
    &= U^T f \left( W^{(2)} a^{(2)} + b^{(2)} \right) \\
    &= U^T a^{(3)}
\end{align*}
\]

- Same derivation as before for \( W^{(2)} \) (now sitting on \( a^{(1)} \))

\[
\begin{align*}
    \frac{\partial s}{\partial W_{ij}} &= U_i f'(z_i) x_j \\
    &= \delta_i x_j \\
    \frac{\partial s}{\partial W^{(2)}_{ij}} &= U_i f' \left( z_i^{(3)} \right) a_j^{(1)} \\
    &= \delta_i^{(3)} a_j^{(2)}
\end{align*}
\]
Two layer neural nets and full backprop

- Same derivation as before for top $W^{(2)}$:

  \[
  \frac{\partial s}{\partial W_{ij}^{(2)}} = U_i f'(z_i^{(3)}) a_j^{(1)}
  = \delta_i^{(3)} a_j^{(2)}
  \]

- In matrix notation:

  \[
  \frac{\partial s}{\partial W^{(2)}} = \delta^{(3)} a^{(2)^T}
  \]

  where \( \delta^{(3)} = U \circ f'(z^{(3)}) \) and \( \circ \) is the element-wise product also called Hadamard product

- Last missing piece for understanding general backprop:

  \[
  \frac{\partial s}{\partial W^{(1)}}
  \]
Two layer neural nets and full backprop

• Last missing piece: \[ \frac{\partial s}{\partial W^{(1)}} \]

• What’s the bottom layer’s error message \( \pm^{(2)} \)?

• Similar derivation to single layer model

\[ \begin{align*}
    x &= z^{(1)} = a^{(1)} \\
    z^{(2)} &= W^{(1)} x + b^{(1)} \\
    a^{(2)} &= f \left( z^{(2)} \right) \\
    z^{(3)} &= W^{(2)} a^{(2)} + b^{(2)} \\
    a^{(3)} &= f \left( z^{(3)} \right) \\
    s &= U^T a^{(3)}
\end{align*} \]

• Main difference, we already have \( W^{(2)^T} \delta^{(3)} \) and need to apply the chain rule again on \( f' \left( z^{(2)} \right) \)
Two layer neural nets and full backprop

- Chain rule for: \( s = U^T f \left( W^{(2)} f \left( W^{(1)} x + b^{(1)} \right) + b^{(2)} \right) \)

- Get intuition by deriving \( \frac{\partial s}{\partial W^{(1)}} \) as if it was a scalar

- Intuitively, we have to sum over all the nodes coming into layer

- Putting it all together: \( \delta^{(2)} = \left( W^{(2)^T} \delta^{(3)} \right) \circ f' \left( z^{(2)} \right) \)
Two layer neural nets and full backprop

- Last missing piece:
  \[
  \frac{\partial s}{\partial W^{(1)}} = \delta^{(2)} x^T
  \]

- In general for any matrix \(W^{(l)}\) at internal layer \(l\) and any error with regularization \(E_R\) all backprop in standard multilayer neural networks boils down to 2 equations:

\[
\begin{align*}
  x &= z^{(1)} = a^{(1)} \\
  z^{(2)} &= W^{(1)} x + b^{(1)} \\
  a^{(2)} &= f \left( z^{(2)} \right) \\
  z^{(3)} &= W^{(2)} a^{(2)} + b^{(2)} \\
  a^{(3)} &= f \left( z^{(3)} \right) \\
  s &= U^T a^{(3)}
\end{align*}
\]

\[
\frac{\partial}{\partial W^{(l)}} E_R = \delta^{(l+1)} (a^{(l)})^T + \lambda W^{(l)}
\]

- Top and bottom layers have simpler ±
Visualization of intuition

- Let’s say we want $\frac{\partial s}{\partial W^{(1)}} = \delta^{(2)} a^{(1)^T}$ with previous layer and $f = \frac{3}{4}$

Gradient w.r.t $W^{(2)} = \delta^{(3)} a^{(2)^T}$
Our first example: Backpropagation using error vectors

\[ z^{(1)} \rightarrow a^{(1)} \rightarrow W^{(1)} \rightarrow z^{(2)} \rightarrow \sigma \rightarrow a^{(2)} \rightarrow W^{(2)} \rightarrow z^{(3)} \rightarrow 1 \rightarrow s \]

\[ \delta^{(3)} = W^{(2)T} \delta^{(3)} \]

--Reusing the \( \delta^{(3)} \) for downstream updates.
--Moving error vector across affine transformation simply requires multiplication with the transpose of forward matrix
--Notice that the dimensions will line up perfectly too!
Visualization of intuition

\[ \sigma'(z^{(2)}) \otimes W^{(2)T} \delta^{(3)} = \delta^{(2)} \]

--Moving error vector across point-wise non-linearity requires point-wise multiplication with local gradient of the non-linearity.
Visualization of intuition

\[ z^{(1)} \rightarrow a^{(1)} \rightarrow W^{(1)} \rightarrow z^{(2)} \rightarrow \sigma \rightarrow a^{(2)} \rightarrow W^{(2)} \rightarrow z^{(3)} \rightarrow 1 \rightarrow s \]

\[ W^{(1)T} \delta^{(2)} \]

Gradient w.r.t. \( W^{(1)} = \delta^{(2)} a^{(1)T} \)
Backpropagation (Another explanation)

• Compute gradient of example-wise loss wrt parameters

• Simply applying the derivative chain rule wisely

\[ z = f(y) \quad y = g(x) \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \]

• If computing the loss(example, parameters) is \( O(n) \) computation, then so is computing the gradient
Simple Chain Rule

\[ \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \]
Multiple Paths Chain Rule

\[
\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x}
\]
Multiple Paths Chain Rule - General

\[
\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}
\]

Diagram: A network of nodes labeled with \(x, y_1, y_2, \ldots, y_n, z\) connected by arrows indicating the flow of influence. The node labeled \(z\) is connected to the nodes labeled \(y_1, y_2, \ldots, y_n\), and \(x\) is connected to \(y_1\) and \(y_2\), with a dotted line indicating an unspecified relationship to \(y_2\).
Chain Rule in Flow Graph

Flow graph: any directed acyclic graph
node = computation result
arc = computation dependency

\[ \{y_1, y_2, \ldots, y_n\} = \text{successors of } x \]

\[
\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}
\]
Back-Prop in Multi-Layer Net

\[ NLL = - \log P(Y = y|x) \]

\[ P(Y = .|x) = \text{softmax}(Wh) \]

\[ h = \text{sigmoid}(Vx) \]
Back-Prop in General Flow Graph

1. Fprop: visit nodes in topo-sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors

\[ \{y_1, y_2, \ldots, y_n\} = \text{successors of } x \]

\[ \frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x} \]
Automatic Differentiation

- The gradient computation can be **automatically inferred** from the symbolic expression of the fprop.
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.
- Easy and fast prototyping
Summary

• Congrats!

• You survived the hardest part of this class.

• Everything else from now on is just more matrix multiplications and backprop :)

• Next up:
  • Recurrent Neural Networks