Neural Networks and Computation Graphs

Based on slides and material from Geoffrey Hinton, Richard Socher, Yoav Goldberg, Chris Dyer, Graham Neubig and others.

This lecture

- What is a neural network?
- Computation Graphs
- Algorithms over computation graphs
  - The forward pass
  - The backward pass

Where are we?

- What is a neural network?
- Computation Graphs
- Algorithms over computation graphs
  - The forward pass
  - The backward pass

Three computational questions

1. Forward propagation
   - Given inputs to the graph, compute the value of the function expressed by the graph
   - Something to think about: Given a node, can we say which nodes are inputs? Which nodes are outputs?

2. Backpropagation
   - After computing the function value for an input, compute the gradient of the function at that input
   - Or equivalently: *How does the output change if I make a small change to the input?*

3. Constructing graphs
   - Need an easy-to-use framework to construct graphs
   - The size of the graph may be input dependent
     - A templating language that creates graphs on the fly
   - Tensorflow, PyTorch are the most popular frameworks today

Constructing computation graphs

Three computational questions

1. Forward propagation
   - Given inputs to the graph, compute the value of the function expressed by the graph
   - Something to think about: Given a node, can we say which nodes are inputs? Which nodes are outputs?

2. Backpropagation
   - After computing the function value for an input, compute the gradient of the function at that input
   - Or equivalently: *How does the output change if I make a small change to the input?*

3. Constructing graphs
   - Need an easy-to-use framework to construct graphs
   - The size of the graph may be input dependent
     - A templating language that creates graphs on the fly
   - Tensorflow, PyTorch are the most popular frameworks today

Two methods for constructing graphs

We may require different sized computation graphs for different inputs

- Eg: different sentences have different lengths. We may have a neural network whose size depends on sentence length.
- How could we statically declare a computation graph of a fixed size?

- One option: Assume a size that is big enough and for smaller examples, pad it with dummy values

- Another option: Dynamically create a computation graph on the fly when we need to.

Two methods for constructing graphs

- Static declaration
  - Phase 1: Define an architecture
    - Maybe using standard control flow operations like loops, conditionals, etc to simplify repeated code
  - Phase 2: Run a bunch of data through the graph to train and make predictions

- Dynamic declaration
  - Graph is constructed implicitly (perhaps via operator overloading) at the same time as the forward propagation

Static declaration

- Pros
  - Offline optimization/scheduling of graphs is powerful
  - Limits on operations mean better hardware support
- Cons
  - Structured data (even simple stuff like sequences), even variable-sized data, is ugly
  - You effectively learn a new programming language (“the Graph Language”) and you write programs in that language to process data.
- Examples: PyTorch, TensorFlow

Dynamic declaration

- Pros
  - The library is less invasive, no need to learn a new syntax
  - Forward computation is written in your favorite programming language with all its features, using your favorite algorithms
  - Interleave construction and evaluation of the graph
- Cons
  - We can't do offline graph optimization because there is little time
  - If the graph is static, the effort can be wasted
- Examples: Chainer, most automatic differentiation libraries, DyNet

Summary: Computation graphs

An abstraction that allows us to write any differentiable (or sub-differentiable) functions as a directed acyclic graph

- Building blocks for modern neural networks
- This will allow us to think about differentiable programs

Two algorithms:

- Forward propagation: process nodes in topological order to compute function value
- Backpropagation: process nodes in reverse topological order to compute derivative

Two methods for constructing graphs: Static vs dynamic

Why computation graphs?

Neural networks are differentiable functions

Neural networks can be written as computation graphs

- We have already seen an example of a two layer neural network

The abstraction allows us to work with named modules

- We saw `nn.Linear`
- Standard libraries define both the low level operators (addition, tanh, softmax, etc) and entire neural network architectures (e.g. Transformer) as computation graphs
- Allow for plug-and-play

Loss functions are differentiable functions

Loss functions are also computation graphs

The standard recipe:

1. Define a neural network architecture (e.g. a module in PyTorch)
2. Initialize the model randomly
3. Load a dataset and iterate over it in batches
4. For each batch,
    1. define the loss (internally a computation graph)
    2. Use the standard algorithms for prediction (forward pass) and taking gradients of the loss (backward pass)
    3. Use standard optimizers (e.g. Adam) to optimize the loss

Worked example: Multiclass logistic regression

General setting

- Inputs: $\mathbf{x}$ (represented in some simple feature space as d-dimensional vectors)
- Output: $\mathrm{y} \in \{1, 2, \cdots, K\}$

The model is defined by

- a $K \times d$ matrix W (i.e. one as d-dimensional vector per label)
- a $K \times 1$ vector b (one bias term per label)

Forward pass: $\mathrm{W}\mathbf{x} + \mathrm{b}$

- This produces a K dimensional vector of scores (one per label)
- Also called the *logits*

Prediction: Pick the label that has the highest score

The underlying probabilistic model

The conditional probability of the label is defined as the *softmax* of the scores:

$$P(y \mid \mathbf{x}) = \frac{\exp\big(\text{score}(y, \mathbf{x})\big)}{\sum_{i=1}^{K} \exp(\text{score}(i, \mathbf{x}))}$$

$$\text{score}(y, x) = (\mathrm{W}\mathbf{x} + b)_y$$

The y^{th} element of the logits vector

The underlying probabilistic model

The conditional probability of the label is defined as the *softmax* of the scores:

$$P(y \mid \mathbf{x}) = \frac{\exp\big(\text{score}(y, \mathbf{x})\big)}{\sum_{i=1}^{K} \exp(\text{score}(i, \mathbf{x}))}$$

Interpretation: Score each label, and then convert to a well-formed probability distribution by exponentiating + normalizing

$$\text{score}(y, x) = (\mathbf{W}\mathbf{x} + b)_y$$

The y^{th} element of the logits vector

The underlying probabilistic model

The conditional probability of the label is defined as the *softmax* of the scores:

$$P(y \mid \mathbf{x}) = \frac{\exp(\text{score}(y, \mathbf{x}))}{\sum_{i=1}^{K} \exp(\text{score}(i, \mathbf{x}))}$$

$$\text{score}(y, x) = (W\mathbf{x} + b)_y$$

The y^{th} element of the logits vector

Interpretation: Score each label, and then convert to a well-formed probability distribution by exponentiating + normalizing

The underlying probabilistic model

The conditional probability of the label is defined as the *softmax* of the scores:

$$P(y \mid \mathbf{x}) = \frac{\exp\big(\text{score}(y, \mathbf{x})\big)}{\sum_{i=1}^{K} \exp(\text{score}(i, \mathbf{x}))}$$

$$\text{score}(y, x) = (\mathrm{W}\mathbf{x} + b)_y$$

The y^{th} element of the logits vector

Interpretation: Score each label, and then convert to a well-formed probability distribution by exponentiating + normalizing

The underlying probabilistic model

The conditional probability of the label is defined as the *softmax* of the scores:

$$P(y \mid \mathbf{x}) = \frac{\exp\big(\text{score}(y, \mathbf{x})\big)}{\sum_{i=1}^{K} \exp(\text{score}(i, \mathbf{x}))}$$

$$\text{score}(y, x) = (W\mathbf{x} + b)_y$$

The y^{th} element of the logits vector

Interpretation: Score each label, and then convert to a well-formed probability distribution by exponentiating + normalizing

This expression uses the softmax function:

$$\text{softmax}(z_1, z_2, \cdots) = \left(\frac{\exp z_1}{\sum_j \exp z_j}, \frac{\exp z_2}{\sum_j \exp z_j}, \cdots\right)$$

The underlying probabilistic model

The conditional probability of the label is defined as the *softmax* of the scores:

$$P(y \mid \mathbf{x}) = \frac{\exp\big(\text{score}(y, \mathbf{x})\big)}{\sum_{i=1}^{K} \exp(\text{score}(i, \mathbf{x}))}$$

Interpretation: Score each label, and then convert to a well-formed probability distribution by exponentiating + normalizing

$$\text{score}(y, x) = (\mathbf{W}\mathbf{x} + b)_y$$

The y^{th} element of the logits vector

The underlying probabilistic model

The conditional probability of the label is defined as the *softmax* of the scores:

$$P(y \mid \mathbf{x}) = \frac{\exp\big(\text{score}(y, \mathbf{x})\big)}{\sum_{i=1}^{K} \exp(\text{score}(i, \mathbf{x}))}$$

Interpretation: Score each label, and then convert to a well-formed probability distribution by exponentiating + normalizing

$$\text{score}(y, x) = (\mathbf{W}\mathbf{x} + b)_y$$

The y^{th} element of the logits vector

If the number of labels is two, this is identical to the probabilistic model for logistic regression.

Exercise: Prove it

Cross-entropy loss

Generalizes the logistic loss for binary classification

Given a labeled example $(\mathbf{x}, y)$ and a model defined probabilistically via W, b the loss is defined as

$$L_{CE}(\mathbf{x}, y, \mathrm{W}, \mathrm{b}) = -\log P(y \mid x, \mathrm{W}, \mathrm{b})$$

Often written differently in documentation. Exercise: Prove that they are the same

Both the model and the loss are computation graphs, so we can use standard machinery

This is a standard building block when we design models. Any time you need your model to make choice between K items, think softmax and cross-entropy loss