CS 224N: Assignment 2

RYAN MCMAHON FRIDAY 10TH FEBRUARY, 2017

Contents

Problem 1: Tensorflow Softmax (25 pts)

In this question, we will implement a linear classifier with loss function

$$J(\boldsymbol{W}) = CE(\boldsymbol{y}, softmax(\boldsymbol{x}\boldsymbol{W})) \tag{1.1}$$

Where $\boldsymbol{x}$ is a row vector of features and $\boldsymbol{W}$ is the weight matrix for the model. We will use TensorFlow's automatic differentiation capability to fit this model to provided data.

1.1 (a) Softmax in Tensorflow (5 pts)

Implement the softmax function using TensorFlow in `q1_softmax.py`. *Remember that*

$$softmax(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}} \tag{1.2}$$

Note that you may not use `tf.nn.softmax` *or related built-in functions. You can run basic (nonexhaustive tests) by running* `python q1_softmax.py`.

Answer:

See code: `~/code/q1_softmax.py`.

1.2 (b) Cross-Entropy Loss in Tensorflow (5 pts)

Implement the cross-entropy loss using TensorFlow in `q1_softmax.py`. *Remember that*

$$CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{N_c} y_i log(\hat{y}_i) \tag{1.3}$$

*where $\boldsymbol{y} \in \mathbb{R}^{N_c}$ is a one-hot label vector and N_c is the number of classes. This loss is summed over all examples (rows) of a minibatch. Note that you may **not** use TensorFlows built-in cross-entropy functions for this question. You can run basic (non-exhaustive tests) by running* `python q1_softmax.py`.

Answer:

See code: `~/code/q1_softmax.py`.

1.3 (c) Placeholders and Feed Dictionaries (5 pts)

Carefully study the Model class in `model.py`. *Briefly explain the purpose of placeholder variables and feed dictionaries in TensorFlow computations. Fill in the implementations for* `add_placeholders` *and* `create_feed_dict` *in* `q1_classifier.py`.

Hint: *Note that configuration variables are stored in the* `Config` *class. You will need to use these configurations variables in the code.*

Answer:

The purpose of placeholder variables and feed dictionaries is, in short, to improve computational efficiency. They allow us to specify the graph structure without supplying the actual data. By doing this, our model can be flexible to different training approaches (e.g., different mini-batch sizes). Specifically, the placeholders stand in for whatever data/values we want to use and the feed dictionaries tell the placeholders what data/values to use.

Also, see code: `~/code/q1_classifier.py`.

1.4 (d) Add Softmax and Add CE Loss (5 pts)

Implement the transformation for a softmax classifier in the function `add_prediction_op` *in* `q1_classifier.py`. *Add cross-entropy loss in the function* `add_loss_op` *in the same file. Use the implementations from the earlier parts of the problem,* ***not*** *TensorFlow built-ins.*

Answer:

See code: `~/code/q1_classifier.py`.

1.5 (e) Add Training Optimizer – Gradient Descent (5 pts)

Fill in the implementation for `add_training_op` *in* `q1_classifier.py`. *Explain how TensorFlows automatic differentiation removes the need for us to define gradients explicitly. Verify that your model is able to fit to synthetic data by running* `q1_classifier.py` *and making sure that the tests pass.*

Hint: *Make sure to use the learning rate specified in* `Config`.

Answer:

TensorFlow's automatic differentiation works by representing a given model as a symbolic Directed Acyclic Graph (DAG). Using this DAG, we can break down an arbitrarily complex function into basic mathematical operations (e.g., $+$, $-$, $\times$, $\div$, log, $\dots$ etc.). Once this is done, TensorFlow is able to numerically calculate the necessary partial derivatives for the gradients using the chain-rule, which alleviates the need for us to define them ourselves.

Also, see code: $\sim$`/code/q1_classifier.py`.

Problem 2: Neural Transition-Based Dependency Parsing (50 pts)

In this section, youll be implementing a neural-network based dependency parser. A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between "head" words and words which modify those heads. Your implementation will be a ***transition-based*** *parser, which incrementally builds up a parse one step at a time. At every step it maintains a partial parse, which is represented as follows:*

- *A* ***stack*** *of words that are currently being processed.*
- *A* ***buffer*** *of words yet to be processed.*
- *A list of* ***dependencies*** *predicted by the parser.*

Initially, the stack only contains ROOT, the dependencies lists is empty, and the buffer contains all words of the sentence in order. At each step, the parse applies a ***transition*** *to the partial parse until its buffer is empty and the stack is of size 1. The following transitions can be applied:*

- `SHIFT`: *removes the first word from the buffer and pushes it onto the stack.*
- `LEFT-ARC`: *marks the second (second most recently added) item on the stack as a dependent of the first item and removes the second item from the stack.*
- `RIGHT-ARC`: *marks the first (most recently added) item on the stack as a dependent of the second item and removes the first item from the stack*

Your parser will decide among transitions at each state using a neural network classifier. First, you will implement the partial parse representation and transition functions.

2.1 (a) Parsing Steps by Hand (6 pts)

*Go through the sequence of transitions needed for parsing the sentence "**I parsed this sentence correctly**". The dependency tree for the sentence is shown below. At each step, give the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any). The first three steps are provided below as an example.*

NOTE: See assignment for table and tree.

Answer:

stack	buffer	new dependency	transition
$[ROOT]$	$[I, parsed, this, sentence, correctly]$		Initial Configuration
$[ROOT, I]$	$[parsed, this, sentence, correctly]$		SHIFT
$[ROOT, I, parsed]$	$[this, sentence, correctly]$		SHIFT
$[ROOT, parsed]$	$[this, sentence, correctly]$	parsed → I	LEFT-ARC
$[ROOT, parsed, this]$	$[sentence, correctly]$		SHIFT
$[ROOT, parsed, this, sentence]$	$[correctly]$		SHIFT
$[ROOT, parsed, sentence]$	$[correctly]$	sentence → this	LEFT-ARC
$[ROOT, parsed]$	$[correctly]$	parsed → sentence	RIGHT-ARC
$[ROOT, parsed, correctly]$	[]		SHIFT
$[ROOT, parsed]$	[]	parsed → correctly	RIGHT-ARC
$[ROOT]$	[]	ROOT → parsed	RIGHT-ARC

2.2 (b) Number of Steps to Parse (2 pts)

A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain why.

Answer:

Assuming we have a grammatically well-formed sentence and need a first step from ROOT:

$$n + 2 \leqslant Steps \leqslant n^3$$

The lower bound exists because we need: 1) an arc from ROOT to the sentence head, and 2) an arc between a subject and a verb. The n is derived from all of the necessary shifts. Our upper bound comes from Eisner (1996) – whose algorithm allows sub-sequences to be parsed independently of one another.

2.3 (c) Partial Parse Coding (6 pts)

Implement the `__init__` *and* `parse_step` *functions in the* `PartialParse` *class in* `q2_parser_transitions.py`. *This implements the transition mechanics your parser will use. You can run basic (not-exhaustive) tests by running* `python q2_parser_transitions.py`.

Answer:

See code: ∼`/code/q2_parser_transitions.py`.

2.4 (d) Minibatch Parsing (6 pts)

Our network will predict which transition should be applied next to a partial parse. We could use it to parse a single sentence by applying predicted transitions until the parse is complete. However, neural networks run much more efficiently when making predictions about batches of data at a time (i.e., predicting the next transition for many different partial parses simultaneously). We can parse sentences in minibatches with the following algorithm:

Algorithm 1: Minibatch Dependency Parsing

INPUT: *sentences*, a list of sentences to be parsed and *model*, our model that makes parse decisions

Initialize *partial_parses* as a list of partial parses, one for each sentence in *sentences* ;
Initialize *unfinished_parses* as a shallow copy of *partial_parses* ;
while *unfinished_parses* $\neq \emptyset$ **do**
    Take the first *batch_size* parses in *unfinished_parses* as a minibatch ;
    Use the *model* to predict the next transition for each partial parse in the minibatch ;
    Perform a parse step on each partial parse in the minibatch with its predicted transition ;
    Remove the completed parses from *unfinished_parses* ;
end

Return: The *dependencies* for each (now completed) parse in *partial_parses*

Implement this algorithm in the `minibatch_parse` *function in* `q2_parser_transitions.py`. *You can run basic (not-exhaustive) tests by running* `python q2_parser_transitions.py`.

Note: You will need `minibatch_parse` *to be correctly implemented to evaluate the model you will build in part (h). However, you do not need it to train the model, so you should be able to complete most of part (h) even if* `minibatch_parse` *is not implemented yet.*

Answer:

See code: ∼`/code/q2_parser_transitions.py`.