CS 224N: Assignment 2

RYAN MCMAHON TUESDAY 14TH 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, $\ldots$ 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: ∼`/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`.

2.5 (N/A) Description (for actual problem see Subproblem 2.6)

We are now going to train a neural network to predict, given the state of the stack, buffer, and dependencies, which transition should be applied next. First, the model extracts a feature vector representing the current state. We will be using the feature set presented in the original neural dependency parsing paper: "A Fast and Accurate Dependency Parser using Neural Networks".[1] *The function extracting these features has been implemented for you in* `parser_utils`*. This feature vector consists of a list of tokens (e.g., the last word in the stack, first word in the buffer, dependent of the second-to-last word in the stack if there is one, etc.). They can be represented as a list of integers*

$$[w_1, w_2, \ldots, w_m]$$

where m is the number of features and each $0 \leqslant w_i \leqslant |V|$ is the index of a token in the vocabulary ($|V|$ is the vocabulary size). First our network looks up an embedding for each word and concatenates them into a single input vector:

$$\boldsymbol{x} = [\boldsymbol{L}_{w_1}, \boldsymbol{L}_{w_2}, \ldots, \boldsymbol{L}_{w_m}] \in \mathbb{R}^{dm}$$

where $\boldsymbol{L} \in \mathbb{R}^{|V| \times d}$ is an embedding matrix with each row $\boldsymbol{L}_i$ as the vector for a particular word i. We then compute our prediction as:

$$\boldsymbol{h} = \text{ReLU}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}_1) \tag{2.1}$$

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{h}\boldsymbol{U} + \boldsymbol{b}_2) \tag{2.2}$$

(recall that ReLU(z) = max(z, 0)). We evaluate using cross-entropy loss:

$$J(\theta) = CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{N_c} y_i log\hat{y}_i \tag{2.3}$$

To compute the loss for the training set, we average this $J(\theta)$ across all training examples.

[1] Chen and Manning (2014)

2.6 (e) Xavier Initialization (4 pts)

In order to avoid neurons becoming too correlated and ending up in poor local minima, it is often helpful to randomly initialize parameters. One of the most frequent initializations used is called Xavier initialization.[2]

Given a matrix $\boldsymbol{A}$ of dimension $m \times n$, Xavier initialization selects values A_{ij} uniformly from $[-\epsilon, \epsilon]$, where

$$\epsilon = \frac{\sqrt{6}}{\sqrt{m+n}} \tag{2.4}$$

Implement the initialization in `xavier_weight_init` *in* `q2_initialization.py`. *You can run basic (nonexhaustive tests) by running* `python q2_initialization.py`. *This function will be used to initialize $\boldsymbol{W}$ and $\boldsymbol{U}$.*

Answer:

See code: `~/code/q2_initialization.py`.

2.7 (f) Dropout Regularization (2 pts)

We will regularize our network by applying Dropout.[3] *During training this randomly sets units in the hidden layer $\boldsymbol{h}$ to zero with probability p and then multiplies $\boldsymbol{h}$ by a constant γ (dropping different units each minibatch). We can write this as*

$$\boldsymbol{h}_{drop} = \lambda \boldsymbol{d} \circ \boldsymbol{h} \tag{2.5}$$

where $\boldsymbol{d} \in \{0,1\}^{D_h}$ (D_h is the size of $\boldsymbol{h}$) is a mask vector where each entry is 0 with probability p and 1 with probability $(1-p)$. γ is chosen such that the value of $\boldsymbol{h}_{drop}$ in expectation equals $\boldsymbol{h}$:

$$\mathbb{E}_{\text{,}}[\boldsymbol{h}_{drop}]_i = \boldsymbol{h}_i \tag{2.6}$$

for all $0 \leqslant i \leqslant D_h$. What must γ equal in terms of p? Briefly justify your answer.

Answer:

γ must be the inverse of p: e.g., $\gamma = \frac{1}{p}$. We have to multiply by the reciprocal of p to get back to our expectation since we are only seeing the unit p times. This is best illustrated with an example. Imagine that $x = [1,1,1,1,1,1]$ and $p = 0.5$. If we randomly set half of x to equal zero a bunch of times, $\bar{x}$ should equal 0.5, and the same will be true for any individual element in x. This is obviously not the case as we can see that the "true" mean of $x = 1$. We can get back to that "true" value by multiplying our estimates by the inverse of p: $p^{-1} = \frac{1}{p} = \frac{2}{1} = 2$.

[2] This is also referred to as Glorot initialization and was initially described in Glorot and Bengio (2010).
[3] Srivastava et al. (2014)

2.8 (g) Adam Optimizer (4 pts)

We will train our model using the Adam optimizer.[4] *Recall that standard SGD uses the update rule*

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta}) \tag{2.7}$$

where $\boldsymbol{\theta}$ is a vector containing all of the model parameters, J is the loss function, $\nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta})$ is the the gradient of the loss function with respect to the parameters on a minibatch of data, and α is the learning rate. Adam uses a more sophisticated update rule with two additional steps.[5]

(i) *First, Adam uses a trick called momentum by keeping track of $\boldsymbol{m}$, a rolling average of the gradients:*

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta}) \tag{2.8}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{m} \tag{2.9}$$

where β_1 s a hyperparameter between 0 and 1 (often set to 0.9). Briefly explain (you dont need to prove mathematically, just give an intuition) how using $\boldsymbol{m}$ stops the updates from varying as much. Why might this help with learning?

(ii) *Adam also uses adaptive learning rates by keeping track of $\boldsymbol{v}$, a rolling average of the magnitudes of the gradients:*

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta}) \tag{2.10}$$
$$\boldsymbol{v} \leftarrow \beta_2 \boldsymbol{v} + (1 - \beta_2)(\nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta}) \circ \nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta})) \tag{2.11}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{m} / \sqrt{\boldsymbol{v}} \tag{2.12}$$

where $\circ$ and $/$ denote elementwise multiplication and division (so $\boldsymbol{z} \circ \boldsymbol{z}$ is elementwise squaring) and β_2 is a hyperparameter between 0 and 1 (often set to 0.99). Since Adam divides the update by $\sqrt{\boldsymbol{v}}$, which of the model parameters will get larger updates? Why might this help with learning?

Answer:

(i): The rolling average, $\boldsymbol{m}$, reduces variance in our updates by effectively down-weighting the impact of the current minibatch's gradient. This way, any outlier set of observations should not dramatically affect our estimates, but will still have some impact. Imagine that we have gone through 10 minibatches, all of which have told us that some parameter's weight needs to change by a fair bit for us to reach some minima (i.e., a "steep gradient"). Then we get to the 11^{th} minibatch and it says we're going the wrong way with our estimates. Intuitively, it makes more sense that this is an outlier batch and that we are indeed traversing the gradient in the correct direction – based on our previous 10 observations. The $\boldsymbol{m}$ parameter captures this intuition.

(ii): Since Adam divides $\boldsymbol{m}$ by $\sqrt{\boldsymbol{v}}$, the parameters with the steepest gradients and the smallest variances will have the largest updates. This makes sense because we *know* that we need to update those values. If the variance of the gradients is large, we aren't sure where we need to go, so we don't want to move too far – for fear of going the wrong way.

[4] Kingma and Ma (2015)
[5] The actual Adam update uses a few additional tricks that are less important, but we wont worry about them for this problem.

2.9 (h) Neural Dependency Parser (20 pts)

In `q2_parser_model.py` *implement the neural network classifier governing the dependency parser by filling in the appropriate sections. We will train and evaluate our model on the Penn Treebank (annotated with Universal Dependencies). Run* `python q2_parser_model.py` *to train your model and compute predictions on the test data (make sure to turn off debug settings when doing final evaluation).*

Hints:

- *When debugging, pass the keyword argument* `debug=True` *to the* `main` *method (it is set to* `True` *by default). This will cause the code to run over a small subset of the data, so training the model won't take as long.*
- *This code should run within 1 hour on a CPU.*
- *When running with* `debug=False`*, you should be able to get a loss smaller than 0.07 on the train set (by the end of the last epoch) and an Unlabeled Attachment Score larger than 88 on the dev set (with the best-performing model out of all the epochs). For comparison, the model in the original neural dependency parsing paper gets 92.5. If you want, you can tweak the hyperparameters for your model (hidden layer size, hyperparameters for Adam, number of epochs, etc.) to improve the performance (but you are not required to do so).*

Deliverables:

- *Working implementation of the neural dependency parser in* `q2_parser_model.py`*. (Well look at, and possibly run this code for grading).*
- *Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set.*
- *List of predicted labels for the test set in the file* `q2_test.predicted`*.*

(i) ***Bonus*** *(1 point): Add an extension to your model (e.g., l2 regularization, an additional hidden layer) and report the change in UAS on the dev set. Briefly explain what your extension is and why it helps (or hurts!) the model. Some extensions may require tweaking the hyperparameters in* `Config` *to make them effective.*

Answer:

See code: `~/code/q2_parser_model.py`.

The best UAS I achieved on the development data set was 88.67%. This translated to an unlabeled attachment score of 89.07% on the test set.

Problem 3: Recurrent Neural Networks: Language Modeling (25 pts)

In this section, youll compute the gradients of a recurrent neural network (RNN) for language modeling.

Language modeling is a central task in NLP, and language models can be found at the heart of speech recognition, machine translation, and many other systems. Given a sequence of words (represented as one-hot row vectors) $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$, a language model predicts the next word $\boldsymbol{x}^{(t+1)}$ by modeling:

$$P(\boldsymbol{x}^{(t+1)} = \boldsymbol{v}_j | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)}) \tag{3.1}$$

where $\boldsymbol{v}_j$ is a word in the vocabulary.

Your job is to compute the gradients of a recurrent neural network language model, which uses feedback information in the hidden layer to model the "history" $\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)}$. Formally, the model is, for $t = 1, \ldots, n-1$:[1]

$$\boldsymbol{e}^{(t)} = \boldsymbol{x}^{(t)}\boldsymbol{L} \tag{3.2}$$

$$\boldsymbol{h}^{(t)} = \text{sigmoid}\left(\boldsymbol{h}^{(t-1)}\boldsymbol{H} + \boldsymbol{e}^{(t)}\boldsymbol{I} + \boldsymbol{b}_1\right) \tag{3.3}$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{h}^{(t)}\boldsymbol{U} + \boldsymbol{b}_2\right) \tag{3.4}$$

$$\bar{P}\left(\boldsymbol{x}^{(t+1)} = \boldsymbol{v}_j | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)}\right) = \hat{\boldsymbol{y}}_j^{(t)} \tag{3.5}$$

where $\boldsymbol{h}^{(0)} = \boldsymbol{h}_0 \in \mathbb{R}^{D_k}$ is some initialization vector for the hidden layer and $\boldsymbol{x}^{(t)}\boldsymbol{L}$ is the product of $\boldsymbol{L}$ with the one-hot row vector $\boldsymbol{x}^{(t)}$ representing the current word. The parameters are:

$$\boldsymbol{L} \in \mathbb{R}^{|V| \times d} \qquad \boldsymbol{H} \in \mathbb{R}^{D_h \times D_h} \tag{3.6}$$

$$\boldsymbol{I} \in \mathbb{R}^{d \times D_h} \qquad \boldsymbol{b}_1 \in \mathbb{R}^{D_h} \tag{3.7}$$

$$\boldsymbol{U} \in \mathbb{R}^{D_h \times |V|} \qquad \boldsymbol{b}_2 \in \mathbb{R}^{|V|} \tag{3.8}$$

where $\boldsymbol{L}$ is the embedding matrix, $\boldsymbol{I}$ is the input word representation matrix, $\boldsymbol{H}$ is the hidden transformation matrix, and $\boldsymbol{U}$ is the output word representation matrix. $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are biases. d is the embedding dimension, $|V|$ is the vocabulary size, and D_h is the hidden layer dimension.

The output vector $\hat{\boldsymbol{y}}^{(t)} \in \mathbb{R}^{|V|}$ is a probability distribution over the vocabulary. The model is trained by minimizing the (un-regularized) cross entropy loss:

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} log \hat{y}_j^{(t)} \tag{3.9}$$

where $\boldsymbol{y}^{(t)}$ is the one-hot vector corresponding to the target word (which here is equal to $\boldsymbol{x}^{(t+1)}$). We average the cross-entropy loss across all examples (i.e., words) in a sequence to get the loss for a single sequence.

[1] Model adapted from Mikolov et al. (2010).