Assignment 2 - Stanford CS 224N Winter 2018

Rohit Apte

2018-01-30

1 Tensorflow Softmax

a see code

b see code

c Carefully study the Model class in model.py. Briey 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 conguration variables are stored in the Cong class. You will need to use these conguration variables in the code.

Tensorflow allows us to set up the operations and computational graph and then run it at a later time. Placeholder variables are variables whose value will be provided later, but can be used to build out the computational graph.
Feed dictionaries create a dictionary of variables (and values) we pass to the calculation engine to compute operations we defined on the graph.

d see code

e Fill in the implementation for add training op in q1 classifier.py. Explain in a few sentences what happens when the models train op is called (what gets computed during forward propagation, what gets computed during backpropagation, and what will have changed after the op has been run?). Verify that your model is able to fit to synthetic data by running python q1 classifier.py and making sure that the tests pass.
Hint: Make sure to use the learning rate specied in Config.

When train op is called it first does the forward propagation where it computes pred (defined in addPredictionOp)
Tensorflow automatically does the back propagation calculation to calculate the gradients (we dont need to specify the gradients)
The weights and biases will get adjusted (changed) after the operation has run.

2 Neural Transition-Based Dependency Parsing

a 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.

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

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

Each word needs to be put on the stack and then operated on to remove from the stack. Therefore for each word there are 2 operations required to parse it. Therefore for n words there will be $2n$ operations.

c see code

d see code

e see code

f We will regularize our network by applying Dropout. During training this randomly sets units in the hidden layer h to zero with probability p_{drop} and then multiplies h by a constant γ (dropping different units each mini-batch). We can write this as
$h_{drop} = \gamma d \circ h$
where $d \in 0, 1^{D_h}$ (D_h is the size of h) is a mask vector where each entry is 0 with probability p_{drop} and 1 with probability $(1p_{drop})$. γ is chosen such that the value of h_{drop} in expectation equals h:
$E_{p_{drop}}[h_{drop}]_i = h_i$
for all $0 < i < D_h$. What must γ equal in terms of p_{drop}? Briey justify your answer.

We know that $E(X) = \sum Xp(X)$
Also $[h_{drop}]_i = \gamma d_i h_i$
Therefore $E_{p_{drop}}[h_{drop}]_i = 0p_{drop} + \gamma h_i(1 - p_{drop}) = \gamma h_i(1 - p_{drop})$
Since expectation equals h we have
$E_{p_{drop}}[h_{drop}]_i = \gamma h_i(1 - p_{drop}) = h_i$
$\therefore \gamma h_i(1 - p_{drop}) = h_i$
$\implies \gamma = \frac{1}{1-p_{drop}}$

g **We will train our model using the Adam optimizer. Recall that standard SGD uses the update rule**
$\theta \leftarrow \theta - \alpha \nabla_\theta J_{minibatch}(\theta)$
where θ is a vector containing all of the model parameters, J is the loss function, $\nabla_\theta J_{minibatch}(\theta)$ is 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.

g.1 **First, Adam uses a trick called momentum by keeping track of m, a rolling average of the gradients:**
$m \leftarrow \beta_1 m + (1\beta_1)\nabla_\theta J_{minibatch}(\theta)$
$\theta \leftarrow \theta - \alpha m$
where β_1 is a hyperparameter between 0 and 1 (often set to 0.9). Briey explain (you dont need to prove mathematically, just give an intuition) how using m stops the updates from varying as much. Why might this help with learning?

SGD doesnt smoothly find the minimum. It takes a jagged path. See image below courtesey of deeplearning4j

Using a rolling average smooths the gradient descent.

g.2 **Adam also uses adaptive learning rates by keeping track of v, a rolling average of the magnitudes of the gradients:**
$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla_\theta J_{minibatch}(\theta)$
$v \leftarrow \beta_2 v + (1 - \beta_2)(\nabla_\theta J_{minibatch}(\theta) \circ \nabla_\theta J_{minibatch}(\theta))$
$\theta \leftarrow \theta - \alpha \circ m/\sqrt{v}$
where $\circ$ and / denote elementwise multiplication and division (so $z \circ z$ is elementwise squaring) and β_2 is a hyperparameter between 0 and 1 (often set to 0.99). Since Adam divides the update by $\sqrt{v}$, which of the model parameters will get larger updates? Why might this help with learning?

Since we are dividing by $\sqrt{v}$, a smaller *sqrtv* will have a larger impact on the updates. v will be smaller if $J_{minibatch}(\theta))$ is small so the parameters with the smallest gradients will have the largest updates. Small gradients means we are in a flat section (a local minimum or plateau). So it will push these points off that section. This can help improve learning.

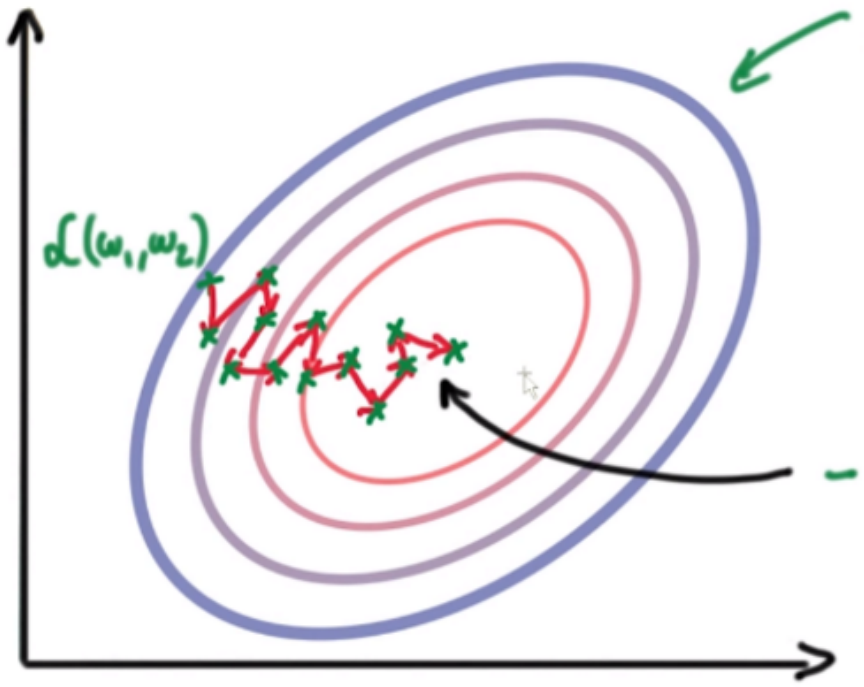

Figure 1: SGD

3 Recurrent Neural Networks: Language Modeling

a Conventionally, when reporting performance of a language model, we evaluate on perplexity, which is defined as:
$PP^{(t)}(y^{(t)}, \hat{y}^{(t)}) = \frac{1}{\bar{P}(x_{pred}^{(t+1)} = x^{(t+1)} | x^{(t)}, ..., x^{(1)})} = \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}}$
i.e. the inverse probability of the correct word, according to the model distribution $\bar{P}$.

a.1 Show how you can derive perplexity from the cross-entropy loss (Hint: remember that y(t) is one-hot!). This should be a very short problem - not too perplexing!

$CE(y^{(t)}, \hat{y}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} log \hat{y}_j^{(t)}$
But y is one hot vector so only one element is 1. Lets assume the i^{th} element is 1. Then
$CE(y^{(t)}, \hat{y}^{(t)}) = -1.log(\hat{y}_i^{(t)}) = log \frac{1}{\hat{y}_i^{(t)}}$
$PP^{(t)}(y^{(t)}, \hat{y}^{(t)}) = \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} . \hat{y}_j^{(t)}}$
Again y is one hot so assume element i is 1.
$PP^{(t)}(y^{(t)}, \hat{y}^{(t)}) = \frac{1}{1.\hat{y}_i^{(t)}} = \frac{1}{\hat{y}_i^{(t)}}$
$\therefore CE(y^{(t)}, \hat{y}^{(t)}) = log(PP^{(t)}(y^{(t)}, \hat{y}^{(t)}))$

a.2 Now use this relationship between perplexity and cross-entropy to show that minimizing the geometric mean perplexity, $\prod_{t=1}^{T} PP(y^{(t)}, \hat{y}^{(t)})^{1/T}$, is equivalent to minimizing the arithmetic mean cross-entropy loss, $\frac{1}{T} \sum_{t=1}^{T} CE(y^{(t)}, \hat{y}^{(t)})$, across the training set. (Hint: for any positive function f, minimizing $log(f)$ is equivalent to minimizing f itself)

Since $\hat{y}$ are probabilities and y are one hot vectors, both CE and PP are positive functions. Therefore minimizing PP is equivalent to minimizing $log(PP)$ $= CE$.
Therefore minimizing the geometric mean perplexity is equivalent to the arithmetic mean cross-entropy loss.

a.3 Suppose you have a vocabulary of $|V|$ words and your "language model" works by picking the next word uniformly at random from the vocabulary (mathematically, $\overline{P}(x^{(t+1)} = w_j|x^{(t)}, \ldots, x^{(1)})) = 1/|V|$ for every word w_j in the vocabulary). What would the perplexity $PP(y^{(t)}, \hat{y}^{(t)})$ for a single word to be? Compute the corresponding cross-entropy loss when $|V| = 10000$.

We know that the Expected value $= \sum xp(x)$
Since we are picking the next word uniformly at random, $E[\hat{y}^{(t)}] = \frac{1}{|V|}$
$CE = log(PP)$ therefore expected value of Cross Entropy if $|V| = 10000 = log(10000) = 9.2103$

b Compute the gradients of the loss J (the cross-entropy loss) with respect to the following model parameters at a single point in time t (to save a bit of time, you dont have to compute the gradients with the respect to the biases b_1 and b_2):

$\frac{\partial J^{(t)}}{\partial U}$ $\frac{\partial J^{(t)}}{\partial e^{(t)}}$ $\frac{\partial J^{(t)}}{\partial W_e^{(t)}}$ $\frac{\partial J^{(t)}}{\partial W_h^{(t)}}$

Additionally, compute the derivative with respect to the previous hidden layer value:

$\frac{\partial J^{(t)}}{\partial h^{(t-1)}}$

$x^{(t)} \in \mathbb{R}^{|V|\times 1}$, $E \in \mathbb{R}^{d\times|V|}$, $W_h \in \mathbb{R}^{D_h\times D_h}$, $W_e \in \mathbb{R}^{D_h\times d}$, $b_1 \in \mathbb{R}^{D_h\times 1}$, $U \in \mathbb{R}^{|V|\times D_h}$, $b_2 \in \mathbb{R}^{|V|\times 1}$, $h^{(0)} \in \mathbb{R}^{D_h\times 1}$

$e^{(t)} = Ex^{(t)} \in \mathbb{R}^{d\times 1}$
let $z^{(t)} = W_h h^{(t-1)} + W_e e^{(t)} + b_1 \in \mathbb{R}^{D_h\times 1}$
$h^{(t)} = \sigma(z^{(t)}) \in \mathbb{R}^{D_h\times 1}$
let $\theta^{(t)} = Uh^{(t)} + b_2 \in \mathbb{R}^{|V|\times 1}$
$\hat{y}^{(t)} = softmax(\theta^{(t)}) \in \mathbb{R}^{|V|\times 1}$

Then $\delta_1^{(t)} = \frac{\partial J}{\partial \theta^{(t)}} = (\hat{y} - y)$
$\delta_2^{(t)} = \frac{\partial J}{\partial z^{(t)}} = \frac{\partial J}{\partial \theta^{(t)}}\frac{\partial \theta^{(t)}}{\partial h^{(t)}}\frac{\partial h^{(t)}}{\partial z^{(t)}} = ((\hat{y} - y)^T U)^T \odot (1 - h^{(t)}) = U^T(\hat{y} - y) \odot h^{(t)} \odot (1 - h^{(t)})$
$\frac{\partial J^{(t)}}{\partial U} = (\hat{y} - y){h^{(t)}}^T$
$\frac{\partial J^{(t)}}{\partial e^{(t)}} = \frac{\partial J^{(t)}}{\partial z^{(t)}}\frac{\partial z^{(t)}}{\partial e^{(t)}} = W_e^T \delta_2^{(t)}$
$\frac{\partial J^{(t)}}{\partial W_e} = \frac{\partial J^{(t)}}{\partial z^{(t)}}\frac{\partial z^{(t)}}{\partial W_e} = \delta_2^{(t)} {e^{(t)}}^T$
$\frac{\partial J^{(t)}}{\partial W_h} = \frac{\partial J^{(t)}}{\partial z^{(t)}}\frac{\partial z^{(t)}}{\partial W_h} = \delta_2^{(t)} {h^{(t-1)}}^T$
$\frac{\partial J^{(t)}}{\partial h^{(t-1)}} = \frac{\partial J^{(t)}}{\partial z^{(t)}}\frac{\partial z^{(t)}}{\partial h^{(t-1)}} = W_h \delta_2^{(t)}$

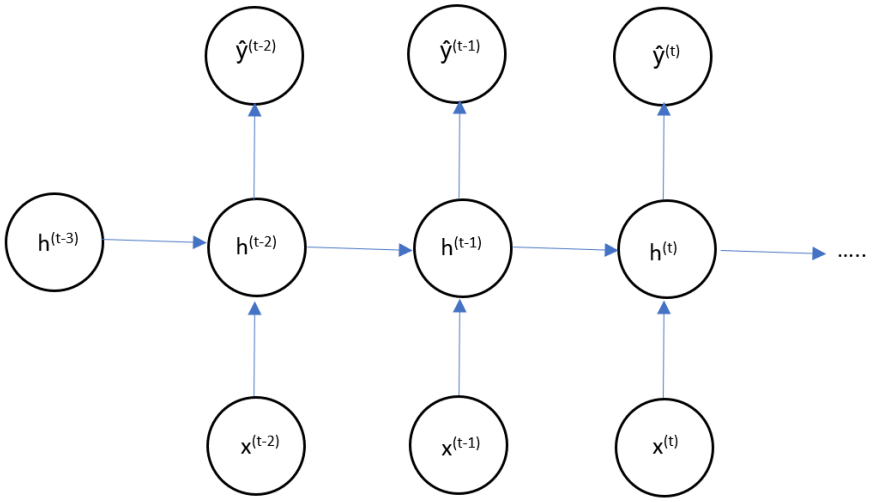

Figure 2: UnrolledRNN

c Below is a sketch of the network at a single timestep: Draw the "unrolled" network for 3 timesteps (your picture should show $h^{(t3)}$ to $h^{(t)}$. Next, compute the following backpropagation-through-time gradients:
$\frac{\partial J^{(t)}}{\partial e^{(t-1)}}$ $\frac{\partial J^{(t)}}{\partial W_e}$ $\frac{\partial J^{(t)}}{\partial W_h}$

See Figure unrolledRNN
$\gamma^{(t-1)} = \frac{\partial J^{(t)}}{\partial h^{(t-1)}}$
Let $\sigma'(z^{(t-1)}) = \frac{\partial h^{(t-1)}}{\partial z^{(t-1)}} = diag(h^{(t-1)} \odot (1 - h^{(t-1)}))$
Then
$\frac{\partial J^{(t)}}{\partial e^{(t-1)}} = W_e^T \sigma'(z^{(t-1)}) \gamma^{(t-1)}$
$\frac{\partial J^{(t)}}{\partial W_e} = \sigma'(z^{(t-1)}) \gamma^{(t-1)} e^{(t-1)^T}$
$\frac{\partial J^{(t)}}{\partial W_h} = \sigma'(z^{(t-1)}) \gamma^{(t-1)} h^{(t-2)^T}$

d **Given $h^{(t-1)}$, how many operations are required to perform backpropagation for a single timestep (i.e., compute the terms you found in part (b)). Express your answer in big-O notation in terms of the dimensions d, D_h and $|V|$ (Equation 1). Dont worry about the gradients you didnt compute (with respect to b_1 and b_2), they actually dont change the answer!**
(Hint: You only have to worry about matrix multiplications, the other operations do not change the big-O runtime. Multiplying a vector by an n by m matrix takes O(nm) operations.)

For 1 single timestep we have $O(|V|D_h + D_h d + D_h^2)$

e **Now suppose you have a sequence of T words. How many operations are required to compute the gradient of the loss with respect to the model parameters across the entire sequence ($\sum_{t=1}^{T} J^{(t)}(\theta)$)? Assume we backpropagate through time all the way to t = 0 for each word.**
Hint: Look at the computational graph you drew in part (c). Will we have to pass an error signal (upstream gradient) into h(0) T times (once for the loss from every word), or can we be more ecient that that? Therefore, how many times will we have to do the single timestep (your answer to (d)) computations?

For T words we would have to process $O(D_h d D_h^2)$ T times so the complexity is $O(T(D_h d D_h^2))$. SInce we are backpropogating all the way back to $t = 0$ its $O(T(|V|D_h + T(D_h d + D_h^2)) = O(T|V|D_h + T^2(D_h d + D_h^2))$

f **Which term in your big-O expressions is likely the largest? Which layer in the RNN is that term from?**

$|V|D_h$ would most likely be the largest calculation since the size of the vocabulary ($|V|$) would typically be very large. It comes from the $\theta^{(t)}$ layer just before the softmax calculation (so its used to calculate $\hat{y}$ which is the probability distribution of the next word given the words until now.