1 Assignment #2

In [40]:

```
import tensorflow as tf
import numpy as np
from utils.general_utils import test_all_close
sess.close()
sess = tf.InteractiveSession()
```

```
C:\Users\janr\AppData\Local\Continuum\anaconda3\lib\site-packages\tensorflow
\python\client\session.py:1714: UserWarning: An interactive session is alrea
dy active. This can cause out-of-memory errors in some cases. You must expli
citly call `InteractiveSession.close()` to release resources held by the oth
er session(s).
  warnings.warn('An interactive session is already active. This can '
```

1.1 Tensorflow Softmax

1.1.1 (a) q1_softmax.py - softmax(x)

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

```
def softmax(x):
    exp = tf.exp(x - tf.reduce_max(x, axis=1, keepdims=True))
    return exp / tf.reduce_sum(exp, axis=1, keepdims=True)
```

1.1.2 (b) q1_softmax.py - cross-entropy

$$CE(y, \hat{y}) = -\sum_{i=1}^{N_c} y_i \log(\hat{y}_i) \quad \Big| \quad N_c \text{ .. number of classes}$$

```
def cross_entropy_loss(y, yhat):
    return - tf.reduce_sum(tf.to_float(y) * tf.log(yhat))
```

1.1.3 (c) model.py and q1_classifier.py

In [ ]:

```
class Config(object):
    """Holds model hyperparams and data information.

    The config class is used to store various hyperparameters and dataset
    information parameters. Model objects are passed a Config() object at
    instantiation.
    """
    n_samples = 1024
    n_features = 100
    n_classes = 5
    batch_size = 64
    n_epochs = 50
    lr = 1e-4
```

1.1.3.1 Briefly explain the purpose of placeholder variables and feed dictionaries in TensorFlow computations.

Placeholders are useful, because then the data is not stored in the graph. The data provided for this is only passed during the graph execution in a session via feed_dict.

1.1.3.2 q1_classifier.py - add_placeholders, create_feed_dict

In [ ]:

```
class SoftmaxModel(Model):
    """Implements a Softmax classifier with cross-entropy loss."""
    def add_placeholders(self):
        """Generates placeholder variables to represent the input tensors.
        These placeholders are used as inputs by the rest of the model building
        and will be fed data during training.
        Adds following nodes to the computational graph
        input_placeholder: Input placeholder tensor of shape
                                              (batch_size, n_features), type tf.float32
        labels_placeholder: Labels placeholder tensor of shape
                                              (batch_size, n_classes), type tf.int32
        Add these placeholders to self as the instance variables
            self.input_placeholder
            self.labels_placeholder
        """
        self.input_placeholder = tf.placeholder(tf.float32, shape=(self.config.batch_size
        self.labels_placeholder = tf.placeholder(tf.int32, shape=(self.config.batch_size,

    def create_feed_dict(self, inputs_batch, labels_batch=None):
        """Creates the feed_dict for training the given step.
        A feed_dict takes the form of:
        feed_dict = {
                <placeholder>: <tensor of values to be passed for placeholder>,
                ....
        }
        If label_batch is None, then no labels are added to feed_dict.
        Hint: The keys for the feed_dict should be the placeholder
                tensors created in add_placeholders.
        Args:
            inputs_batch: A batch of input data.
            labels_batch: A batch of label data.
        Returns:
            feed_dict: The feed dictionary mapping from placeholders to values.
        """
        feed_dict = {
            self.input_placeholder: inputs_batch,
            self.labels_placeholder: labels_batch,
        }
        return feed_dict
```

1.1.4 (d) q1_classifier.py - add_prediction_op, add_loss_op

In [ ]:

```
def add_prediction_op(self):
    """Adds the core transformation for this model which transforms a batch of input
    data into a batch of predictions. In this case, the transformation is a linear layer
    softmax transformation:

    y = softmax(Wx + b)

    Hint: Make sure to create tf.Variables as needed.
    Hint: For this simple use-case, it's sufficient to initialize both weights W
                and biases b with zeros.

    Args:
        input_data: A tensor of shape (batch_size, n_features).
    Returns:
        pred: A tensor of shape (batch_size, n_classes)
    """
    ### YOUR CODE HERE
    W = tf.Variable(tf.zeros([self.config.n_features,self.config.n_classes]))
    b = tf.Variable(tf.zeros([self.config.batch_size,self.config.n_classes]))
    pred = softmax(tf.matmul(self.input_placeholder,W) + b)
    ### END YOUR CODE
    return pred

def add_loss_op(self, pred):
    """Adds cross_entropy_loss ops to the computational graph.

    Hint: Use the cross_entropy_loss function we defined. This should be a very
                short function.
    Args:
        pred: A tensor of shape (batch_size, n_classes)
    Returns:
        loss: A 0-d tensor (scalar)
    """
    ### YOUR CODE HERE
    loss = cross_entropy_loss(self.labels_placeholder,pred)
    ### END YOUR CODE
    return loss
```

1.1.5 (e) q1_classifier.py - add_training_op

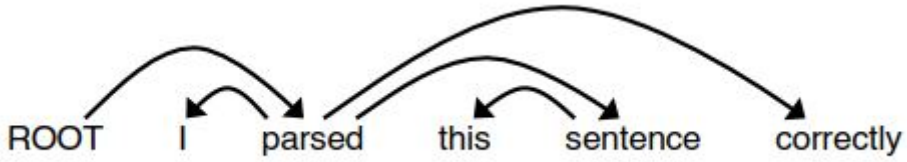
In [ ]:

```
def add_training_op(self, loss):
    """Sets up the training Ops.
    Creates an optimizer and applies the gradients to all trainable variables.
    The Op returned by this function is what must be passed to the
    `sess.run()` call to cause the model to train. See
    https://www.tensorflow.org/versions/r0.7/api_docs/python/train.html#Optimizer
    for more information.
    Hint: Use tf.train.GradientDescentOptimizer to get an optimizer object.
                Calling optimizer.minimize() will return a train_op object.
    Args:
       loss: Loss tensor, from cross_entropy_loss.
    Returns:
       train_op: The Op for training.
    """
    optimizer = tf.train.GradientDescentOptimizer(self.config.lr)
    train_op = optimizer.minimize(loss)
    return train_op
```

1.1.5.1 Explain how TensorFlow's automatic differentiation removes the need for us to define gradients explicitly.

TensorFlow's automatic differentiation already calculates the gradients by using the chain rule along all the dependent variables needed for the gradient. This is particularly efficient because the required values are already calculated by the function during the forward pass.

1.2 Neural Transition-Based



stackbuffer	*newdependency*	*transition*
$[ROOT][I, parsed, this, sentence, correctly]$		*InitialConfigurc*
$[ROOT, I][parsed, this, sentence, correctly]$		SHIFT
$[ROOT, I, parsed][this, sentence, correctly]$		SHIFT
$[ROOT, parsed][this, sentence, correctly]$	$parsed \rightarrow I$	LEFT-ARC
$[ROOT, parsed, this][sentence, correctly]$		SHIFT
$[ROOT, parsed, this, sentence][correctly]$		SHIFT
$[ROOT, parsed, sentence][correctly]$	$sentence \rightarrow this$	LEFT-ARC
$[ROOT, parsed][correctly]$	$parsed \rightarrow sentence$	RIGHT-ARC
$[ROOT, parsed, correctly][]$		SHIFT
$[ROOT, parsed][]$	$parsed \rightarrow correctly$	RIGHT-ARC
$[ROOT][]$	$ROOT \rightarrow parsed$	RIGHT-ARC

1.2.1 (b) A sentence containing n words will be parsed in how many steps

(in terms of n)? Briefly explain why

2n steps. Because you need n shift steps until you have all the words out of the buffer and in the stack and you need another n arc of steps to empty the stack.

1.2.2 (c) q2_parser_transitions.py - init, parse_step

In [98]:

```
class PartialParse(object):
    def __init__(self, sentence):
        """Initializes this partial parse.

        Your code should initialize the following fields:
            self.stack: The current stack represented as a list with the top of the stack
                        last element of the list.
            self.buffer: The current buffer represented as a list with the first item on
                         buffer as the first item of the list
            self.dependencies: The list of dependencies produced so far. Represented as a
                    tuples where each tuple is of the form (head, dependent).
                    Order for this list doesn't matter.

        The root token should be represented with the string "ROOT"

        Args:
            sentence: The sentence to be parsed as a list of words.
                      Your code should not modify the sentence.
        """
        # The sentence being parsed is kept for bookkeeping purposes. Do not use it in yo
        self.sentence = sentence

        ### YOUR CODE HERE
        self.stack = ['ROOT']
        self.buffer = sentence
        self.dependencies = []
        ### END YOUR CODE

    def parse_step(self, transition):
        """Performs a single parse step by applying the given transition to this partial

        Args:
            transition: A string that equals "S", "LA", or "RA" representing the shift, l
                        and right-arc transitions.
        """
        ### YOUR CODE HERE
        if transition == "S":
            self.stack.append(self.buffer[0])
            del self.buffer[0]
        elif transition == "LA":
            self.dependencies.append((self.stack[-1],self.stack[-2]))
            del self.stack[-2]
        elif transition == "RA":
            self.dependencies.append((self.stack[-2],self.stack[-1]))
            del self.stack[-1]
        ### END YOUR CODE

    def parse(self, transitions):
        """Applies the provided transitions to this PartialParse

        Args:
            transitions: The list of transitions in the order they should be applied
        Returns:
            dependencies: The list of dependencies produced when parsing the sentence. Re
                          as a list of tuples where each tuple is of the form (head, depe
        """
        for transition in transitions:
            self.parse_step(transition)
        return self.dependencies
```

```
def minibatch_parse(sentences, model, batch_size):
    """Parses a list of sentences in minibatches using a model.

    Args:
        sentences: A list of sentences to be parsed (each sentence is a list of words)
        model: The model that makes parsing decisions. It is assumed to have a function
               model.predict(partial_parses) that takes in a list of PartialParses as inp
               returns a list of transitions predicted for each parse. That is, after cal
                   transitions = model.predict(partial_parses)
               transitions[i] will be the next transition to apply to partial_parses[i].
        batch_size: The number of PartialParses to include in each minibatch
    Returns:
        dependencies: A list where each element is the dependencies list for a parsed sen
                      Ordering should be the same as in sentences (i.e., dependencies[i]
                      contain the parse for sentences[i]).
    """

    ### YOUR CODE HERE
    ### END YOUR CODE

    return dependencies


def test_step(name, transition, stack, buf, deps,
              ex_stack, ex_buf, ex_deps):
    """Tests that a single parse step returns the expected output"""
    pp = PartialParse([])
    pp.stack, pp.buffer, pp.dependencies = stack, buf, deps

    pp.parse_step(transition)
    stack, buf, deps = (tuple(pp.stack), tuple(pp.buffer), tuple(sorted(pp.dependencies))
    assert stack == ex_stack, \
        "{:} test resulted in stack {:}, expected {:}".format(name, stack, ex_stack)
    assert buf == ex_buf, \
        "{:} test resulted in buffer {:}, expected {:}".format(name, buf, ex_buf)
    assert deps == ex_deps, \
        "{:} test resulted in dependency list {:}, expected {:}".format(name, deps, ex_de
    print("{:} test passed!".format(name))


def test_parse_step():
    """Simple tests for the PartialParse.parse_step function
    Warning: these are not exhaustive
    """
    test_step("SHIFT", "S", ["ROOT", "the"], ["cat", "sat"], [],
              ("ROOT", "the", "cat"), ("sat",), ())
    test_step("LEFT-ARC", "LA", ["ROOT", "the", "cat"], ["sat"], [],
              ("ROOT", "cat",), ("sat",), (("cat", "the"),))
    test_step("RIGHT-ARC", "RA", ["ROOT", "run", "fast"], [], [],
              ("ROOT", "run",), (), (("run", "fast"),))

if __name__ == '__main__':
    test_parse_step()
```

```
SHIFT test passed!
LEFT-ARC test passed!
RIGHT-ARC test passed!
```

1.2.3 (d) Implement the Alg in minibatch_parse function

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` is not empty **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 while

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

In [111]:

```
class PartialParse(object):
    def __init__(self, sentence):
        """Initializes this partial parse.

        Your code should initialize the following fields:
            self.stack: The current stack represented as a list with the top of the stack
                        last element of the list.
            self.buffer: The current buffer represented as a list with the first item on
                         buffer as the first item of the list
            self.dependencies: The list of dependencies produced so far. Represented as a
                    tuples where each tuple is of the form (head, dependent).
                    Order for this list doesn't matter.

        The root token should be represented with the string "ROOT"

        Args:
            sentence: The sentence to be parsed as a list of words.
                      Your code should not modify the sentence.
        """
        # The sentence being parsed is kept for bookkeeping purposes. Do not use it in yo
        self.sentence = sentence

        ### YOUR CODE HERE
        self.stack = ['ROOT']
        self.buffer = sentence[:]
        self.dependencies = []
        ### END YOUR CODE

    def parse_step(self, transition):
        """Performs a single parse step by applying the given transition to this partial

        Args:
            transition: A string that equals "S", "LA", or "RA" representing the shift, l
                        and right-arc transitions.
        """
        ### YOUR CODE HERE
        if transition == "S":
            self.stack.append(self.buffer[0])
            del self.buffer[0]
        elif transition == "LA":
            self.dependencies.append((self.stack[-1],self.stack[-2]))
            del self.stack[-2]
        elif transition == "RA":
            self.dependencies.append((self.stack[-2],self.stack[-1]))
            del self.stack[-1]
        ### END YOUR CODE
    def parse(self, transitions):
        """Applies the provided transitions to this PartialParse

        Args:
            transitions: The list of transitions in the order they should be applied
        Returns:
            dependencies: The list of dependencies produced when parsing the sentence. Re
                          as a list of tuples where each tuple is of the form (head, depe
        """
        for transition in transitions:
            self.parse_step(transition)
        return self.dependencies
```

```
def minibatch_parse(sentences, model, batch_size):
    """Parses a list of sentences in minibatches using a model.

    Args:
        sentences: A list of sentences to be parsed (each sentence is a list of words)
        model: The model that makes parsing decisions. It is assumed to have a function
               model.predict(partial_parses) that takes in a list of PartialParses as inp
               returns a list of transitions predicted for each parse. That is, after cal
                   transitions = model.predict(partial_parses)
               transitions[i] will be the next transition to apply to partial_parses[i].
        batch_size: The number of PartialParses to include in each minibatch
    Returns:
        dependencies: A list where each element is the dependencies list for a parsed sen
                      Ordering should be the same as in sentences (i.e., dependencies[i]
                      contain the parse for sentences[i]).
    """

    ### YOUR CODE HERE
    partial_parses = [PartialParse(s) for s in sentences]
    unfinished_parses = partial_parses.copy()
    while len(unfinished_parses) > 0:
        minibatch = unfinished_parses[:batch_size]
        transitions = model.predict(minibatch)
        for i, parse in enumerate(minibatch):
            parse.parse_step(transitions[i])
            if len(parse.stack) <= 1 or len(parse.stack) <= 0:
                unfinished_parses.remove(parse)

    dependencies = [parse.dependencies for parse in partial_parses]


    ### END YOUR CODE

    return dependencies


def test_step(name, transition, stack, buf, deps,
              ex_stack, ex_buf, ex_deps):
    """Tests that a single parse step returns the expected output"""
    pp = PartialParse([])
    pp.stack, pp.buffer, pp.dependencies = stack, buf, deps

    pp.parse_step(transition)
    stack, buf, deps = (tuple(pp.stack), tuple(pp.buffer), tuple(sorted(pp.dependencies))
    assert stack == ex_stack, \
        "{:} test resulted in stack {:}, expected {:}".format(name, stack, ex_stack)
    assert buf == ex_buf, \
        "{:} test resulted in buffer {:}, expected {:}".format(name, buf, ex_buf)
    assert deps == ex_deps, \
        "{:} test resulted in dependency list {:}, expected {:}".format(name, deps, ex_de
    print("{:} test passed!".format(name))


def test_parse_step():
    """Simple tests for the PartialParse.parse_step function
    Warning: these are not exhaustive
    """
    test_step("SHIFT", "S", ["ROOT", "the"], ["cat", "sat"], [],
              ("ROOT", "the", "cat"), ("sat",), ())
    test_step("LEFT-ARC", "LA", ["ROOT", "the", "cat"], ["sat"], [],
```

```
                ("ROOT", "cat",), ("sat",), (("cat", "the"),))
    test_step("RIGHT-ARC", "RA", ["ROOT", "run", "fast"], [], [],
                ("ROOT", "run",), (), (("run", "fast"),))


def test_parse():
    """Simple tests for the PartialParse.parse function
    Warning: these are not exhaustive
    """
    sentence = ["parse", "this", "sentence"]
    dependencies = PartialParse(sentence).parse(["S", "S", "S", "LA", "RA", "RA"])
    dependencies = tuple(sorted(dependencies))
    expected = (('ROOT', 'parse'), ('parse', 'sentence'), ('sentence', 'this'))
    assert dependencies == expected,  \
        "parse test resulted in dependencies {:}, expected {:}".format(dependencies, expe
    assert tuple(sentence) == ("parse", "this", "sentence"), \
        "parse test failed: the input sentence should not be modified"
    print("parse test passed!")


class DummyModel:
    """Dummy model for testing the minibatch_parse function
    First shifts everything onto the stack and then does exclusively right arcs if the fi
    the sentence is "right", "left" if otherwise.
    """
    def predict(self, partial_parses):
        return [("RA" if pp.stack[1] is "right" else "LA") if len(pp.buffer) == 0 else "S
                for pp in partial_parses]


def test_dependencies(name, deps, ex_deps):
    """Tests the provided dependencies match the expected dependencies"""
    deps = tuple(sorted(deps))
    assert deps == ex_deps, \
        "{:} test resulted in dependency list {:}, expected {:}".format(name, deps, ex_de


def test_minibatch_parse():
    """Simple tests for the minibatch_parse function
    Warning: these are not exhaustive
    """
    sentences = [["right", "arcs", "only"],
                 ["right", "arcs", "only", "again"],
                 ["left", "arcs", "only"],
                 ["left", "arcs", "only", "again"]]
    deps = minibatch_parse(sentences, DummyModel(), 2)
    test_dependencies("minibatch_parse", deps[0],
                      (('ROOT', 'right'), ('arcs', 'only'), ('right', 'arcs')))
    test_dependencies("minibatch_parse", deps[1],
                      (('ROOT', 'right'), ('arcs', 'only'), ('only', 'again'), ('right',
    test_dependencies("minibatch_parse", deps[2],
                      (('only', 'ROOT'), ('only', 'arcs'), ('only', 'left')))
    test_dependencies("minibatch_parse", deps[3],
                      (('again', 'ROOT'), ('again', 'arcs'), ('again', 'left'), ('again',
    print("minibatch_parse test passed!")

if __name__ == '__main__':
    test_parse_step()
    test_parse()
    test_minibatch_parse()
```

```
SHIFT test passed!
```

```
SHIFT test passed!
LEFT-ARC test passed!
RIGHT-ARC test passed!
parse test passed!
minibatch_parse test passed!
```

1.2.4 (e) q2_initialization.py - xavier_weight_init

In [115]:

```
import numpy as np
import tensorflow as tf


def xavier_weight_init():
    """Returns function that creates random tensor.

    The specified function will take in a shape (tuple or 1-d array) and
    returns a random tensor of the specified shape drawn from the
    Xavier initialization distribution.

    Hint: You might find tf.random_uniform useful.
    """
    def _xavier_initializer(shape, **kwargs):
        """Defines an initializer for the Xavier distribution.
        Specifically, the output should be sampled uniformly from [-epsilon, epsilon] whe
            epsilon = sqrt(6) / <sum of the sizes of shape's dimensions>
        e.g., if shape = (2, 3), epsilon = sqrt(6 / (2 + 3))

        This function will be used as a variable initializer.

        Args:
            shape: Tuple or 1-d array that species the dimensions of the requested tensor
        Returns:
            out: tf.Tensor of specified shape sampled from the Xavier distribution.
        """
        ### YOUR CODE HERE
        epsilon = np.sqrt(6) / np.sum(shape)
        out = tf.random_uniform(shape, minval=-epsilon, maxval=epsilon)
        ### END YOUR CODE
        return out
    # Returns defined initializer function.
    return _xavier_initializer


def test_initialization_basic():
    """Some simple tests for the initialization.
    """
    print("Running basic tests...")
    xavier_initializer = xavier_weight_init()
    shape = (1,)
    xavier_mat = xavier_initializer(shape)
    assert xavier_mat.get_shape() == shape

    shape = (1, 2, 3)
    xavier_mat = xavier_initializer(shape)
    assert xavier_mat.get_shape() == shape
    print("Basic (non-exhaustive) Xavier initialization tests pass")


if __name__ == "__main__":
    test_initialization_basic()
```

```
Running basic tests...
Basic (non-exhaustive) Xavier initialization tests pass
```

1.2.5 (f) dropout: What must gamma equal in terms of p_{drop}

$$\begin{aligned}\mathbb{E}_{p_{drop}}[h_{drop}]_i &= \mathbb{E}_{p_{drop}}[\gamma d_i \circ h_i] \quad \Big| \quad d_i \text{..only 0 for prob } p_{drop}\text{, else 1 for prob } (1-p_{drop}) \\ &= \underbrace{p_{drop} \cdot 0 + (1-p_{drop}) \cdot 1 \cdot (\gamma h_i)}_{\text{should be 1}} \quad \Big| \quad \gamma = \frac{1}{1-p_{drop}} \\ &= h_i\end{aligned}$$

One has to reduce the activations by gamma in relation to p, because the neurons still active after dropout only make up a part of the overall model.

1.2.6 (g) Adam optimizer

(i) Adam and momentum: How using m stops the updates from varying as much. Why might this help with learning?

---

m stops the updates because 90% of the previous updates are taken into account and only 10% of the new gradient is used. So a sudden larger gradient would barely affect the momentum achieved so far. Due to the fact that the new gradient after a few time steps has little influence on the learning behavior, ie the change of theta, one approaches the once targeted minimum much faster.

Dadurch, dass der neue Gradient nun nach einigen Zeitschritten wenig Einfluss auf das Lernverhalten nimmt, das heißt das Ändern von theta, nähert man sich so dem einmal anvisierten Minimum wesentlich schneller.

(ii) Since Adam divides the update by $\sqrt{v}$, which of the model parameters will get larger updates? Why might this help with learning?

It helps to scale the gradient. In steep dimensions, this means smaller changes and larger changes in flat dimensions. So there is a balancing effect, resulting in a faster learning process.

Es hilft, da dadurch der Gradient skaliert wird. In steilen Dimensionen bedeutet dies kleinere Änderungen und in flachen Dimensionen größere Änderungen. Es findet also ein ausgleichender Effekt statt, was einen schnelleren Lernprozess zur Folge hat.

1.2.7 (h)

In [1]:

```
import os
import time
import tensorflow as tf
import pickle

from model import Model
#from q2_initialization import xavier_weight_init
from utils.general_utils import Progbar
from utils.parser_utils import minibatches, load_and_preprocess_data


class Config(object):
    """Holds model hyperparams and data information.
    The config class is used to store various hyperparameters and dataset
    information parameters. Model objects are passed a Config() object at
    instantiation.
    """
    n_features = 36
    n_classes = 3
    dropout = 0.5
    embed_size = 50
    hidden_size = 200
    batch_size = 2048
    n_epochs = 10
    lr = 0.001


class ParserModel(Model):
    """
    Implements a feedforward neural network with an embedding layer and single hidden lay
    This network will predict which transition should be applied to a given partial parse
    configuration.
    """

    def add_placeholders(self):
        """Generates placeholder variables to represent the input tensors
        These placeholders are used as inputs by the rest of the model building and will
        data during training.  Note that when "None" is in a placeholder's shape, it's fl
        (so we can use different batch sizes without rebuilding the model).
        Adds following nodes to the computational graph
        input_placeholder: Input placeholder tensor of  shape (None, n_features), type tf
        labels_placeholder: Labels placeholder tensor of shape (None, n_classes), type tf
        dropout_placeholder: Dropout value placeholder (scalar), type tf.float32
        Add these placeholders to self as the instance variables
            self.input_placeholder
            self.labels_placeholder
            self.dropout_placeholder
        (Don't change the variable names)
        """
        ### YOUR CODE HERE
        n_features = self.config.n_features
        n_classes = self.config.n_classes

        self.input_placeholder = tf.placeholder(tf.int32,
                                                shape=(None, n_features))
        self.labels_placeholder = tf.placeholder(tf.float32,
                                                 (None, n_classes))
        self.dropout_placeholder = tf.placeholder(tf.float32)
        ### END YOUR CODE
```

```
    def create_feed_dict(self, inputs_batch, labels_batch=None, dropout=1):
        """Creates the feed_dict for the dependency parser.
        A feed_dict takes the form of:
        feed_dict = {
                <placeholder>: <tensor of values to be passed for placeholder>,
                ....
        }
        Hint: The keys for the feed_dict should be a subset of the placeholder
                    tensors created in add_placeholders.
        Hint: When an argument is None, don't add it to the feed_dict.
        Args:
            inputs_batch: A batch of input data.
            labels_batch: A batch of label data.
            dropout: The dropout rate.
        Returns:
            feed_dict: The feed dictionary mapping from placeholders to values.
        """
        ### YOUR CODE HERE
        feed_dict = {
            self.input_placeholder: inputs_batch,
            self.dropout_placeholder: dropout
        }

        if labels_batch is not None:
            feed_dict[self.labels_placeholder] = labels_batch

        ### END YOUR CODE
        return feed_dict

    def add_embedding(self):
        """Adds an embedding layer that maps from input tokens (integers) to vectors and
        concatenates those vectors:
            - Creates an embedding tensor and initializes it with self.pretrained_embeddi
            - Uses the input_placeholder to index into the embeddings tensor, resulting i
              tensor of shape (None, n_features, embedding_size).
            - Concatenates the embeddings by reshaping the embeddings tensor to shape
              (None, n_features * embedding_size).
        Hint: You might find tf.nn.embedding_lookup useful.
        Hint: You can use tf.reshape to concatenate the vectors. See following link to un
            what -1 in a shape means.
            https://www.tensorflow.org/api_docs/python/array_ops/shapes_and_shaping#resha
        Returns:
            embeddings: tf.Tensor of shape (None, n_features*embed_size)
        """
        ### YOUR CODE HERE
        n_features = self.config.n_features
        embedding_size = self.config.embed_size

        vocabulary = tf.Variable(self.pretrained_embeddings)
        embeddings = tf.nn.embedding_lookup(vocabulary, self.input_placeholder)
        embeddings = tf.reshape(embeddings, (-1, n_features * embedding_size))
        ### END YOUR CODE

        return embeddings

    def add_prediction_op(self):
        """Adds the 1-hidden-layer NN:
            h = Relu(xW + b1)
            h_drop = Dropout(h, dropout_rate)
            pred = h_dropU + b2
```

```
        Note that we are not applying a softmax to pred. The softmax will instead be done
        the add_loss_op function, which improves efficiency because we can use
        tf.nn.softmax_cross_entropy_with_logits
        Use the initializer from q2_initialization.py to initialize W and U (you can init
        and b2 with zeros)
        Hint: Here are the dimensions of the various variables you will need to create
                    W:  (n_features*embed_size, hidden_size)
                    b1: (hidden_size,)
                    U:  (hidden_size, n_classes)
                    b2: (n_classes)
        Hint: Note that tf.nn.dropout takes the keep probability (1 - p_drop) as an argum
            The keep probability should be set to the value of self.dropout_placeholder
        Returns:
            pred: tf.Tensor of shape (batch_size, n_classes)
        """

        x = self.add_embedding()
        ### YOUR CODE HERE
        xavier_init = xavier_weight_init()

        n_features = self.config.n_features
        n_classes = self.config.n_classes
        embed_size = self.config.embed_size
        hidden_size = self.config.hidden_size

        W = tf.Variable(
            xavier_init((n_features * embed_size, hidden_size)))
        b1 = tf.Variable(xavier_init((1, hidden_size)))
        U = tf.Variable(xavier_init((hidden_size, n_classes)))
        b2 = tf.Variable(xavier_init((1, n_classes)))

        z = tf.add(tf.matmul(x, W), b1)
        h = tf.nn.relu(z)
        h_drop = tf.nn.dropout(h, self.dropout_placeholder)
        pred = tf.add(tf.matmul(h_drop, U), b2)
        ### END YOUR CODE
        return pred

    def add_loss_op(self, pred):
        """Adds Ops for the loss function to the computational graph.
        In this case we are using cross entropy loss.
        The loss should be averaged over all examples in the current minibatch.
        Hint: You can use tf.nn.softmax_cross_entropy_with_logits to simplify your
                    implementation. You might find tf.reduce_mean useful.
        Args:
            pred: A tensor of shape (batch_size, n_classes) containing the output of the
                  network before the softmax layer.
        Returns:
            loss: A 0-d tensor (scalar)
        """
        ### YOUR CODE HERE
        probs = tf.nn.softmax_cross_entropy_with_logits(
            pred,
            self.labels_placeholder)
        loss = tf.reduce_mean(probs)
        ### END YOUR CODE
        return loss

    def add_training_op(self, loss):
        """Sets up the training Ops.
        Creates an optimizer and applies the gradients to all trainable variables.
```

```
        The Op returned by this function is what must be passed to the
        `sess.run()` call to cause the model to train. See
        https://www.tensorflow.org/versions/r0.7/api_docs/python/train.html#Optimizer
        for more information.
        Use tf.train.AdamOptimizer for this model.
        Calling optimizer.minimize() will return a train_op object.
        Args:
            loss: Loss tensor, from cross_entropy_loss.
        Returns:
            train_op: The Op for training.
        """
        ### YOUR CODE HERE
        optimizer = tf.train.AdamOptimizer(self.config.lr)
        train_op = opimizer.minimize(loss)
        ### END YOUR CODE
        return train_op

    def train_on_batch(self, sess, inputs_batch, labels_batch):
        feed = self.create_feed_dict(inputs_batch, labels_batch=labels_batch,
                                     dropout=self.config.dropout)
        _, loss = sess.run([self.train_op, self.loss], feed_dict=feed)
        return loss

    def run_epoch(self, sess, parser, train_examples, dev_set):
        prog = Progbar(target=1 + len(train_examples) / self.config.batch_size)
        for i, (train_x, train_y) in enumerate(minibatches(train_examples, self.config.ba
            loss = self.train_on_batch(sess, train_x, train_y)
            prog.update(i + 1, [("train loss", loss)])

        print("Evaluating on dev set", end=' ')
        dev_UAS, _ = parser.parse(dev_set)
        print("- dev UAS: {:.2f}".format(dev_UAS * 100.0))
        return dev_UAS

    def fit(self, sess, saver, parser, train_examples, dev_set):
        best_dev_UAS = 0
        for epoch in range(self.config.n_epochs):
            print("Epoch {:} out of {:}".format(epoch + 1, self.config.n_epochs))
            dev_UAS = self.run_epoch(sess, parser, train_examples, dev_set)
            if dev_UAS > best_dev_UAS:
                best_dev_UAS = dev_UAS
                if saver:
                    print("New best dev UAS! Saving model in ./data/weights/parser.weight
                    saver.save(sess, './data/weights/parser.weights')
            print()

    def __init__(self, config, pretrained_embeddings):
        self.pretrained_embeddings = pretrained_embeddings
        self.config = config
        self.build()


def main(debug=True):
    print(80 * "=")
    print("INITIALIZING")
    print(80 * "=")
    config = Config()
    parser, embeddings, train_examples, dev_set, test_set = load_and_preprocess_data(debu
    if not os.path.exists('./data/weights/'):
        os.makedirs('./data/weights/')
```

```
    with tf.Graph().as_default():
        print("Building model...", end=' ')
        start = time.time()
        model = ParserModel(config, embeddings)
        parser.model = model
        print("took {:.2f} seconds\n".format(time.time() - start))

        init = tf.global_variables_initializer()
        # If you are using an old version of TensorFlow, you may have to use
        # this initializer instead.
        # init = tf.initialize_all_variables()
        saver = None if debug else tf.train.Saver()

        with tf.Session() as session:
            parser.session = session
            session.run(init)

            print(80 * "=")
            print("TRAINING")
            print(80 * "=")
            model.fit(session, saver, parser, train_examples, dev_set)

            if not debug:
                print(80 * "=")
                print("TESTING")
                print(80 * "=")
                print("Restoring the best model weights found on the dev set")
                saver.restore(session, './data/weights/parser.weights')
                print("Final evaluation on test set", end=' ')
                UAS, dependencies = parser.parse(test_set)
                print("- test UAS: {:.2f}".format(UAS * 100.0))
                print("Writing predictions")
                with open('q2_test.predicted.pkl', 'w') as f:
                    pickle.dump(dependencies, f, -1)
                print("Done!")

if __name__ == '__main__':
    main()
```

```
C:\Users\janr\AppData\Local\Continuum\anaconda3\lib\site-packages\h5py\__ini
t__.py:36: FutureWarning: Conversion of the second argument of issubdtype fr
om `float` to `np.floating` is deprecated. In future, it will be treated as
`np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters

================================================================================
====
INITIALIZING
================================================================================
====
Loading data...

---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-1-8f93ca3f5579> in <module>()
    278
    279 if __name__ == '__main__':
--> 280     main()

<ipython-input-1-8f93ca3f5579> in main(debug)
    237     print(80 * "=")
    238     config = Config()
```

```
--> 239     parser, embeddings, train_examples, dev_set, test_set = load_and
_preprocess_data(debug)
    240     if not os.path.exists('./data/weights/'):
    241         os.makedirs('./data/weights/')

~\AppData\Local\Continuum\anaconda3\envs\cs224n\utils\parser_utils.py in loa
d_and_preprocess_data(reduced)
    344     start = time.time()
    345     train_set = read_conll(os.path.join(config.data_path, config.tra
in_file),
--> 346                            lowercase=config.lowercase)
    347     dev_set = read_conll(os.path.join(config.data_path, config.dev_f
ile),
    348                          lowercase=config.lowercase)

~\AppData\Local\Continuum\anaconda3\envs\cs224n\utils\parser_utils.py in rea
d_conll(in_file, lowercase, max_example)
    280 def read_conll(in_file, lowercase=False, max_example=None):
    281     examples = []
--> 282     with open(in_file) as f:
    283         word, pos, head, label = [], [], [], []
    284         for line in f.readlines():

FileNotFoundError: [Errno 2] No such file or directory: './data\\train.conl
l'
```