Assignment2

1 Dynet Softmax

(a)

```
def softmax(x):
    ### YOUR CODE HERE
    fz = dy.exp(dy.colwise_add(x, -dy.max_dim(x, d=1)))
    fm = dy.sum_cols(fz)
    out = dy.cdiv(fz, fm)
    ### END YOUR CODE
```

(b)

```
def cross_entropy_loss(y, yhat):
    ### YOUR CODE HERE
    out = dy.sum_elems(-dy.cmult(y, dy.log(yhat)))
    ### END YOUR CODE
```

(c) (d) (e)

```
class SoftmaxModel(object):

    def __init__(self, config, m):
        self.config = config
        self.pW = m.add_parameters((config.n_features, config.n_classes))
        self.pb = m.add_parameters((config.n_classes,))

    def create_network_return_loss(self, input, label):
        self.input = dy.inputTensor(input)
        self.label = dy.inputTensor(label)
        W = dy.parameter(self.pW)
        b = dy.parameter(self.pb)
        y = softmax(self.input * W + self.label)
        return cross_entropy_loss(self.label, y)

def test_softmax_model():
    """Train softmax model for a number of steps."""
    config = Config()

    # Generate random data to train the model on
    np.random.seed(1234)
    inputs = np.random.rand(config.n_samples, config.n_features)
    labels = np.zeros((config.n_samples, config.n_classes), dtype=np.int32)
```

```
    labels[:, 1] = 1
    #for i in xrange(config.n_samples):
    #    labels[i, i%config.n_classes] = 1

    mini_batches = [
        [inputs[k:k+config.batch_size], labels[k:k+config.batch_size]]
        for k in xrange(0, config.n_samples, config.batch_size)]

    m = dy.ParameterCollection()
    trainer = dy.SimpleSGDTrainer(m)
    trainer.learning_rate = config.lr
    net = SoftmaxModel(config, m)
    for epoch in range(config.n_epochs):
        start_time = time.time()
        for mini_batch in mini_batches:
            dy.renew_cg()
            losses = []
            for ix in xrange(config.batch_size):
                l = net.create_network_return_loss(
                    np.array(mini_batch[0][ix]).reshape(1,
config.n_features),
                    np.array(mini_batch[1][ix]).reshape(1,
config.n_classes))
                losses.append(l)
            loss = dy.esum(losses) / config.batch_size
            loss.forward()
            loss.backward()
            trainer.update()
        duration = time.time() - start_time
        print 'Epoch {:}: loss = {:.2f} ({:.3f} sec)'.format(epoch,
loss.value(), duration)

    print loss.value()
    assert loss.value() < .5
    print "Basic (non-exhaustive) classifier tests pass"
```

2 Neural Transition-Based Dependency Parsing

(a)

stack	buffer	dependency	transition
[ROOT]	[I, parsed, this, sentence, correctly]		*Initial*
[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)

2n.

n SHIFT: move all n words from buffer to stack

n ARC(LEFT/RIGHT): pop all n words on stack

(c)

```python
def __init__(self, sentence):
    ### YOUR CODE HERE
    self.stack = ["ROOT"]
    self.buffer = sentence[:]
    self.dependencies = []
    ### END YOUR CODE

def parse_step(self, transition):
    ### YOUR CODE HERE
    if transition == "S":
        if self.buffer:
            self.stack.append(self.buffer[0])
            self.buffer.pop(0)
    elif transition == "LA":
        if len(self.stack) >= 2:
            self.dependencies.append((self.stack[-1], self.stack[-2]))
            self.stack.pop(-2)
    else:
        if len(self.stack) >= 2:
            self.dependencies.append((self.stack[-2], self.stack[-1]))
            self.stack.pop(-1)
    ### END YOUR CODE
```

(d)

```python
# Not use batch
def minibatch_parse(sentences, model, batch_size):
    ### YOUR CODE HERE
    dependencies = []
    for sentence in sentences:
        pp = PartialParse(sentence)
        for i in xrange(2*len(sentence)):
            action = model.predict([pp])
            pp.parse(action)
        dependencies.append(pp.dependencies)
    ### END YOUR CODE
```

(e)

```python
# Dynet aotu use xavier(Glorot) initializer
def _xavier_initializer(shape, **kwargs):
    ### YOUR CODE HERE
    m = dy.ParameterCollection()
    out = m.add_parameters(shape).as_array()
    ### END YOUR CODE
```

(f)

$$
\begin{aligned}
\mathbb{E}_{p_{drop}}[\boldsymbol{h}_{drop}]_i &= \mathbb{E}_{p_{drop}}[\gamma \boldsymbol{d}_i \boldsymbol{h}_i] \\
&= p_{drop} \cdot 0\gamma \boldsymbol{h}_i + (1 - p_{drop}) \cdot 1\gamma \boldsymbol{h}_i \\
&= (1 - p_{drop}) \cdot \gamma \boldsymbol{h}_i \\
&= \boldsymbol{h}_i \\
\therefore \gamma &= \frac{1}{1 - p_{drop}}
\end{aligned}
$$

(g)

(i)

增加动量可以在当前梯度下降时考虑之前梯度下降的趋势，这样相当于考虑了整体，有效避免了当前步出错带来的巨大影响，从而使梯度下降更准确。

(ii)

因为一次梯度下降中，不同参数的梯度大小不一，使用相同的学习率对参数不公平，梯度小的参数相比其他参数更新缓慢。Adam将梯度除以$\sqrt{\boldsymbol{v}}$ 使得梯度小的参数学习率更高，更新较大，有利于整体的梯度下降。

(h)

```
class ParserModel(object):

    def __init__(self, config, pretrained_embeddings, parser):
        self.config = config
        print len(pretrained_embeddings)
        self.m = dy.ParameterCollection()
        self.Initializer = dy.ConstInitializer(0.0)
        self.pW =
self.m.add_parameters((self.config.n_features*self.config.embed_size,
self.config.hidden_size))
        self.pB1 = self.m.add_parameters((1, self.config.hidden_size),
init=self.Initializer)
        self.pU = self.m.add_parameters((self.config.hidden_size,
self.config.n_classes))
        self.pB2 = self.m.add_parameters((1, self.config.n_classes),
init=self.Initializer)

        self.word_lookup =
self.m.lookup_parameters_from_numpy(pretrained_embeddings)
        self.pos_lookup = self.m.add_lookup_parameters((self.config.n_pos,
self.config.embed_size))

        self.trainer = dy.AdamTrainer(self.m)

    def create_network_return_pred(self, input, drop=False):
        n = self.config.n_features / 2
```

```
        p_embd = [dy.lookup(self.pos_lookup, input[0][i+n]) for i in
xrange(n)]
        w_embd = [dy.lookup(self.word_lookup, input[0][i]) for i in
xrange(n)]
        embd = w_embd + p_embd
        x = dy.reshape(dy.concatenate(embd), (1,
self.config.n_features*self.config.embed_size))
        W = dy.parameter(self.pW)
        b1 = dy.parameter(self.pB1)
        h = dy.rectify(x*W+b1)
        h_drop = dy.dropout(h, self.config.dropout) if drop else h

        U = dy.parameter(self.pU)
        b2 = dy.parameter(self.pB2)

        pred = h_drop*U + b2
        pred = dy.softmax(dy.reshape(pred, (3,)))
        return pred

    def create_network_return_loss(self, pred, label):
        if label[0]:
            expected_output = 0
        elif label[1]:
            expected_output = 1
        else:
            expected_output = 2
        loss = -dy.log(dy.pick(pred, expected_output))
        return loss

    def predict_on_batch(self, x_batch):
        out = []
        for x in x_batch:
            pred = self.create_network_return_pred(np.array(x).reshape(1,
self.config.n_features), drop=False)
            out.append(pred.npvalue())
        return out


def run_epoch(model, config, parser, train_examples, dev_set):
    prog = Progbar(target=1 + len(train_examples) / config.batch_size)
    flag = False

    for i, (train_x, train_y) in enumerate(minibatches(train_examples,
config.batch_size)):
        dy.renew_cg()
        losses = []
        for x, y in zip(train_x, train_y):
            pred = model.create_network_return_pred(np.array(x).reshape(1,
config.n_features), drop=True)
```

```
            loss = model.create_network_return_loss(pred, y)
            losses.append(loss)
        loss = dy.esum(losses) / config.batch_size
        loss.forward()
        loss.backward()
        model.trainer.update()

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

def train_network(config, saver, parser, embeddings, train_examples,
dev_set, test_set):
    best_dev_UAS = 0
    model = ParserModel(config, embeddings, parser)
    parser.model = model
    for epoch in range(config.n_epochs):
        print "Epoch {:} out of {:}".format(epoch + 1, config.n_epochs)
        dev_UAS = run_epoch(model, config, parser, train_examples, dev_set)
        if dev_UAS > best_dev_UAS:
            best_dev_UAS = dev_UAS
            if not saver:
                print "New best dev UAS! Saving model in
./data/weights/parser.weights"
                dy.save('./data/weights/parser.weights', [model.pW,
model.pB1, model.pU, model.pB2])


    if saver:
        print 80 * "="
        print "TESTING"
        print 80 * "="
        print "Restoring the best model weights found on the dev set"
        model.pW, model.pB1, model.pU, model.pB2 =
dy.load('./data/weights/parser.weights', model.m)
        print "Final evaluation on test set",
        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:
            cPickle.dump(dependencies, f, -1)
        print "Done!"
```

3、Recurrent Neural Networks: Language Modeling

(a)

$$
\begin{aligned}
PP^{(t)}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) &= \frac{1}{\overline{P}(\boldsymbol{x}_{pred}^{t+1} = \boldsymbol{x}^{t+1} | \boldsymbol{x}^{(t)}, \cdots, \boldsymbol{x}^{(1)})} \\
&= \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \\
&= \frac{1}{\hat{y}_i^{(t)}} = \frac{1}{\frac{1}{|V|}} = |V| \\
CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) &= \log PP^{(t)}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) \\
&= \log(|V|) = \log 10000
\end{aligned}
$$

(b)

$$
\begin{aligned}
\boldsymbol{e}^{(t)} &= \boldsymbol{x}^{(t)} \boldsymbol{L} \\
\boldsymbol{z}^{(t)} &= \boldsymbol{h}^{(t-1)} \boldsymbol{H} + \boldsymbol{e}^{(t)} \boldsymbol{I} + \boldsymbol{b}_1 \\
\boldsymbol{h}^{(t)} &= \sigma(\boldsymbol{z}^{(t)}) \\
\boldsymbol{a}^{(t)} &= \boldsymbol{h}^{(t)} \boldsymbol{U} + \boldsymbol{b}_2 \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{a}^{(t)})
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial J^{(t)}}{\partial \boldsymbol{a}^{(t)}} &= \hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)} \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_2} &= \frac{\partial J^{(t)}}{\partial \boldsymbol{a}^{(t)}} \cdot \frac{\partial \boldsymbol{a}^{(t)}}{\partial \boldsymbol{b}_2} = \hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)} \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{z}^{(t)}} &= \frac{\partial J^{(t)}}{\partial \boldsymbol{a}^{(t)}} \cdot \frac{\partial \boldsymbol{a}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \cdot \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{z}^{(t)}} = (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \cdot \boldsymbol{U}^\top \cdot \boldsymbol{z}^{(t)} \cdot (1 - \boldsymbol{z}^{(t)}) \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{I}} &= \frac{\partial J^{(t)}}{\partial \boldsymbol{z}^{(t)}} \cdot \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{I}} = (\boldsymbol{e}^{(t)})^\top \cdot (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \cdot \boldsymbol{U}^\top \cdot \boldsymbol{z}^{(t)} \cdot (1 - \boldsymbol{z}^{(t)}) \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{H}} &= \frac{\partial J^{(t)}}{\partial \boldsymbol{z}^{(t)}} \cdot \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{H}} = (\boldsymbol{h}^{(t-1)})^\top \cdot (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \cdot \boldsymbol{U}^\top \cdot \boldsymbol{z}^{(t)} \cdot (1 - \boldsymbol{z}^{(t)}) \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} &= \frac{\partial J^{(t)}}{\partial \boldsymbol{z}^{(t)}} \cdot \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \cdot \boldsymbol{U}^\top \cdot \boldsymbol{z}^{(t)} \cdot (1 - \boldsymbol{z}^{(t)}) \cdot \boldsymbol{H}^\top \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{L}_{x^{(t)}}} &= \frac{\partial J^{(t)}}{\partial \boldsymbol{z}^{(t)}} \cdot \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{e}^{(t)}} \cdot \frac{\partial \boldsymbol{e}^{(t)}}{\partial \boldsymbol{L}_{x^{(t)}}} = (x^{(t)})^\top \cdot (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \cdot \boldsymbol{U}^\top \cdot \boldsymbol{z}^{(t)} \cdot (1 - \boldsymbol{z}^{(t)}) \cdot \boldsymbol{I}^\top
\end{aligned}
$$

(c)

略，类似`(b)`中的求导。

(d)

forward: $O(|V|D_h + dD_h + D_h^2)$

backpropagation: $O(|V|D_h + dD_h + D_h^2)$

For τ steps: $O\big(\tau(|V|D_h + dD_h + D_h^2)\big)$