Week 4: Language Modeling II

Note: We recommend working through the Week 1 TensorFlow tutorial (../week1/TensorFlow%20Tutorial.ipynb) before starting this notebook.

Note on training time

The NPLM can take a while to train on a slower machine - we clocked it at 10-20 min on a 2-core Cloud Compute instance.

If you're using a cloud compute instance, you can add more CPUs without having to re-do setup. With your instance turned off, go to https://console.cloud.google.com/compute/instances (https://console.cloud.google.com/compute/instances), click your instance, and go to "Edit". Under machine type, select "Custom" and pick 4-8 CPUs and 2 GB of RAM. Make sure you shut down when you're done, and use the Edit menu again to scale back the size to something less expensive.

In [1]:
```
#!pip install --upgrade pandas
#!pip install --upgrade pip
import os, sys, re, json, time
import itertools
import collections
from IPython.display import display

# NLTK for NLP utils and corpora
import nltk

# NumPy and TensorFlow
import numpy as np
import tensorflow as tf

# Pandas because pandas are awesome, and for pretty-printing
import pandas as pd
# Set pandas floating point display
pd.set_option('float_format', lambda f: "{0:.04f}".format(f))

# Helper libraries for this notebook
import utils
reload(utils)
import vocabulary
reload(vocabulary)
```

Out[1]:
```
<module 'vocabulary' from 'vocabulary.pyc'>
```

For this week's notebook, we'll implement the Neural Probabilistic Language Model (Bengio et al. 2003) (http://machinelearning.wustl.edu/mlpapers/paper_files/BengioDVJ03.pdf). This model is a straightforward extension of n-gram language modeling: it uses a fixed context window, but uses a neural network to predict the next word.

Recall that our n-gram mode of order $k+1$ was:

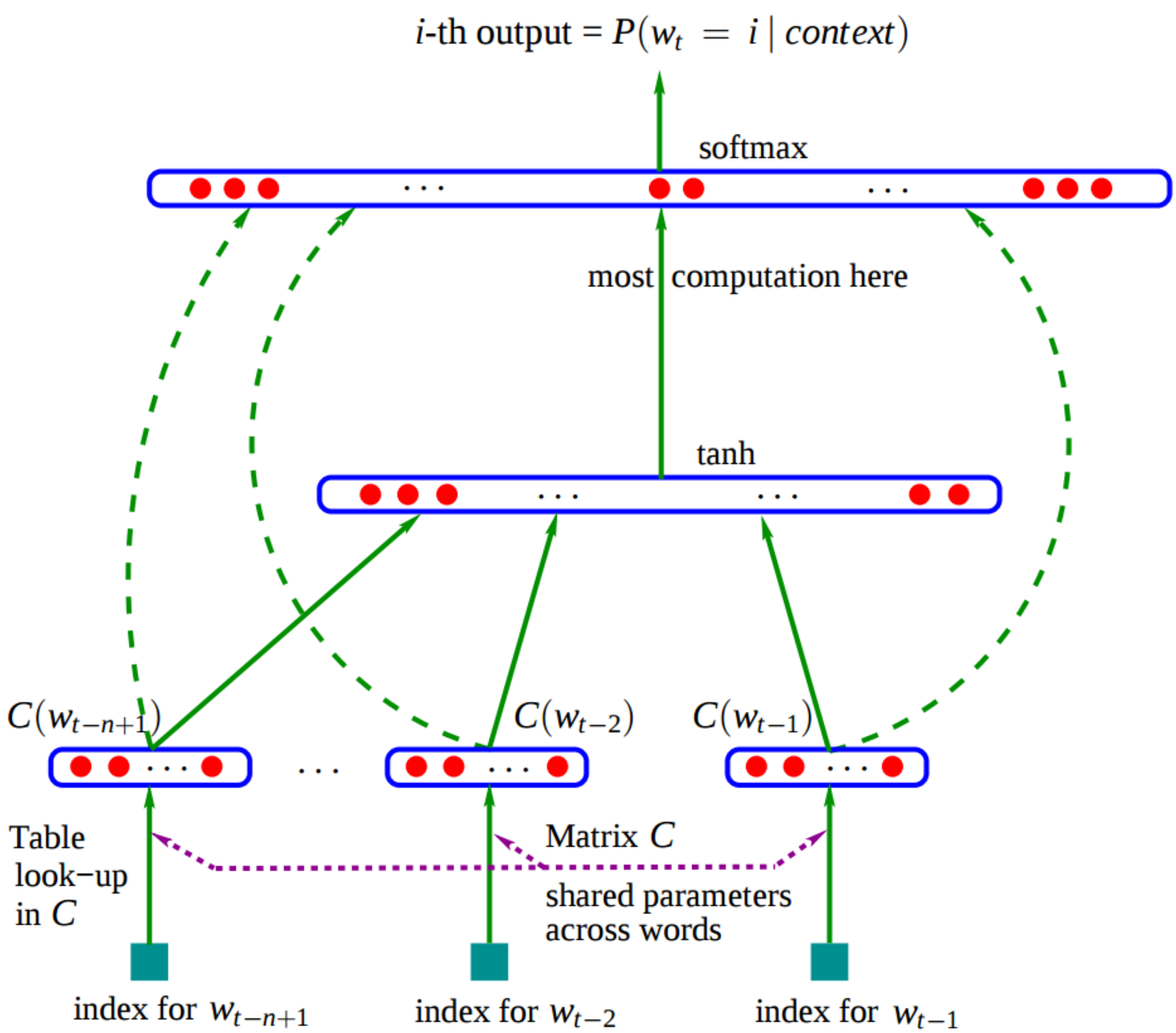
$$P(w_i|w_{i-1}, w_{i-2}, \ldots, w_0) \approx P(w_i|w_{i-1}, \ldots, w_{i-k})$$

Where we estimated the probabilities by smoothed maximum likelihood.

For the NPLM, we'll replace that estimate with a neural network predictor that directly learns a mapping from contexts $(w_{i-1}, \ldots, w_{i-k})$ to a distribution over words w_i:

$$P(w_i|w_{i-1}, \ldots, w_{i-k}) = f(w_i, (w_{i-1}, \ldots, w_{i-k}))$$

Here's what that network will look like:



Broadly, there are three parts:

1. **Embedding layer**: map words into vector space
2. **Hidden layer**: compress and apply nonlinearity
3. **Output layer**: predict next word using softmax

With modern computers and a couple tricks, we should be able to get a decent model to run in just a few minutes - a far cry from the three weeks it took in 2003!

Constructing our Model

To implement the NPLM in TensorFlow, we need to define a Tensor for each model component. As in the tutorial, we'll use variable names that end in an underscore for Tensor objects. We'll also construct the model so it can accept batch inputs, as this will greatly speed up training:

Hyperparameters:

- `V` : vocabulary size
- `M` : embedding size
- `N` : context window size
- `H` : hidden units

Inputs:

- `ids_` : (batch_size, N), integer indices for context words
- `y_` : (batch_size,), integer indices for target word

Model parameters:

- `C_` : (V,M), input-side word embeddings
- `W1_` : (NxM, H)
- `b1_` : (H,)
- `W2_` : (H, V)
- `W3_` : (NxM, V), matrix for skip-layer connection
- `b3_` : (V,)

Intermediate states:

- `x_` : (batch_size, NxM), concatenated embeddings
- `h_` : (batch_size, H), hidden state $= \tanh(xW_1 + b_1)$
- `logit_` : (batch_size, V), $= hW_2 + xW_3 + b_3$

In [2]:
```
tf.reset_default_graph()
tf.set_random_seed(42)

##
# Hyperparameters
V = 1000
M = 30
N = 3
H = 50

# Inputs
# Using "None" in place of batch size allows
# it to be dynamically computed later.
with tf.name_scope("Inputs"):
    ids_ = tf.placeholder(tf.int32, shape=[None, N], name="ids")
    y_ = tf.placeholder(tf.int32, shape=[None], name="y")

with tf.name_scope("Embedding_Layer"):
    C_ = tf.Variable(tf.random_uniform([V, M], -1.0, 1.0), name="C")
    # embedding_lookup gives shape (batch_size, N, M)
    x_ = tf.reshape(tf.nn.embedding_lookup(C_, ids_),
                    [-1, N*M], name="x")

with tf.name_scope("Hidden_Layer"):
    W1_ = tf.Variable(tf.random_normal([N*M,H]), name="W1")
    b1_ = tf.Variable(tf.zeros([H,], dtype=tf.float32), name="b1")
    # We could write tf.matmul(x_, W1_) + b1_,
    # but tf.add lets us give it a name.
    h_ = tf.tanh(tf.matmul(x_, W1_) + b1_, name="h")

with tf.name_scope("Output_Layer"):
    W2_ = tf.Variable(tf.random_normal([H,V]), name="W2")
    W3_ = tf.Variable(tf.random_normal([N*M,V]), name="W3")
    b3_ = tf.Variable(tf.zeros([V,], dtype=tf.float32), name="b3")
    # Concat [h x] and [W2 W3]
    hx_ = tf.concat(1, [h_, x_], name="hx")
    W23_ = tf.concat(0, [W2_, W3_], name="W23")
    logits_ = tf.add(tf.matmul(hx_, W23_), b3_, name="logits")
```

We'll add in our usual cross-entropy loss. Recall from async that this is *very* slow for a large vocabulary, and even for a small vocabulary it represents the bulk of the computation time. To speed up training we'll use a sampled softmax loss, as in Jozefowicz et al. 2016 (https://arxiv.org/abs/1602.02410):

In [3]:
```
with tf.name_scope("Cost_Function"):
    # Sampled softmax loss, for training
    per_example_train_loss_ = tf.nn.sampled_softmax_loss(tf.transpose(W23_), b
3_, hx_,
                                                         labels=tf.expand_dims(y_, 1),
                                                         num_sampled=100, num_classes=V,
                                                         name="per_example_sampled_softmax
_loss")
    train_loss_ = tf.reduce_sum(per_example_train_loss_, name="sampled_softmax
_loss")

    # Full softmax loss, for scoring
    per_example_loss_ =
tf.nn.sparse_softmax_cross_entropy_with_logits(logits_, y_, name="per_example_
loss")
    loss_ = tf.reduce_sum(per_example_loss_, name="loss")
```

And add training ops. We'll use AdaGrad instead of vanilla SGD, as this tends to converge faster:

In [4]:
```
with tf.name_scope("Training"):
    alpha_ = tf.placeholder(tf.float32, name="learning_rate")
    optimizer_ = tf.train.AdagradOptimizer(alpha_)
    # train_step_ = optimizer_.minimize(loss_)
    train_step_ = optimizer_.minimize(train_loss_)

# Initializer step
init_ = tf.initialize_all_variables()
```

Finally, we'll add a few ops to do prediction:

- `pred_proba_` : (batch*size, V), $ = P(w_i | w*{i-1}, ...)$forallwords$i$
- `pred_max` : (batch_size,), id of most likely next word
- `pred_random` : (batch_size,), id of a randomly-sampled next word

In [5]:
```
with tf.name_scope("Prediction"):
    pred_proba_ = tf.nn.softmax(logits_, name="pred_proba")
    pred_max_ = tf.argmax(logits_, 1, name="pred_max")
    pred_random_ = tf.multinomial(logits_, 1, name="pred_random")
```

We can use TensorBoard to view this graph, even before we run the model:

In [6]:
```
summary_writer = tf.train.SummaryWriter("tf_summaries",
                                        tf.get_default_graph())
```

In [11]:
```
!tensorboard --logdir="tf_summaries" --port 6006
```

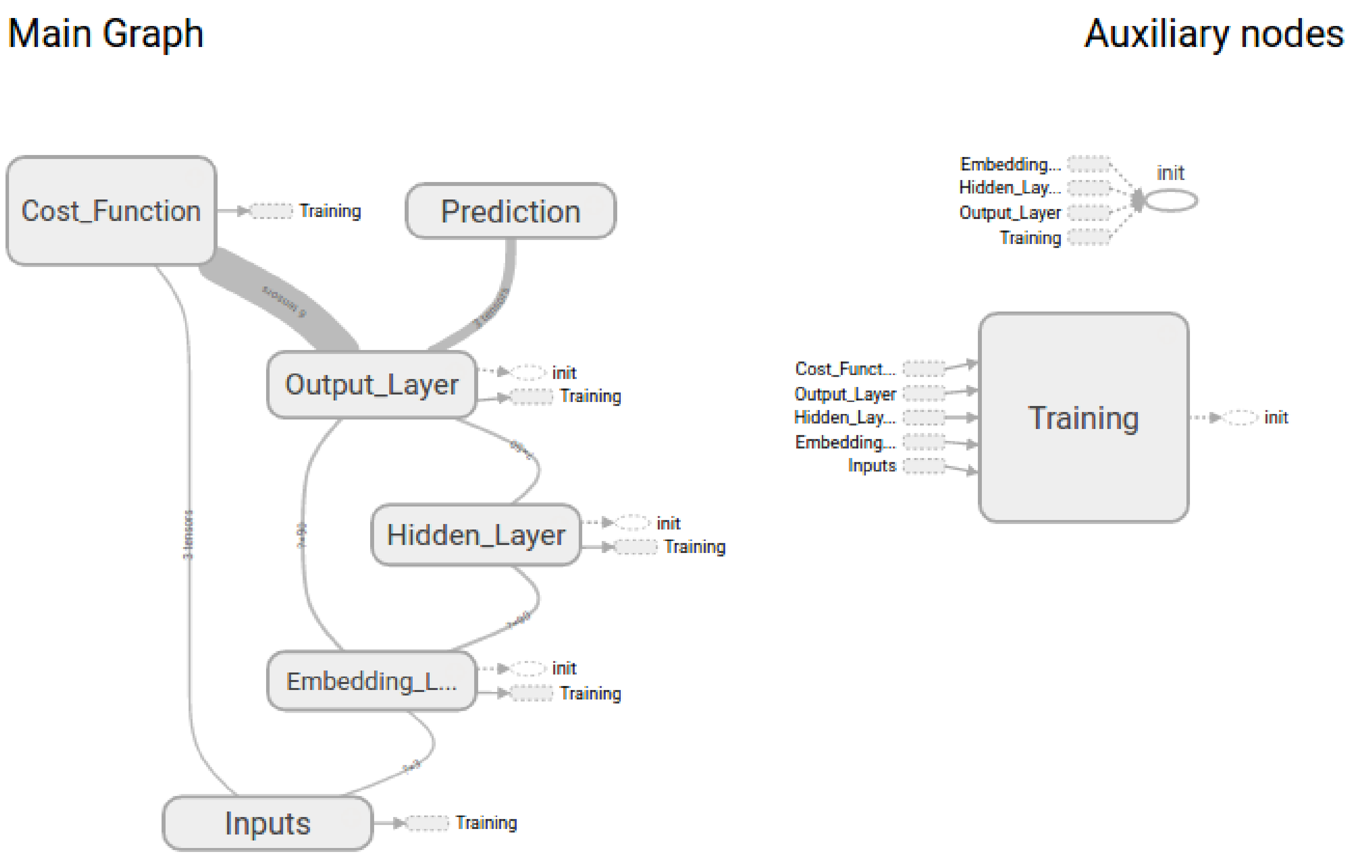
```
ERROR:tensorflow:Tried to connect to port 6006, but address is in use.
Tried to connect to port 6006, but address is in use.
```

In a separate terminal, run:

```
tensorboard --logdir="~/w266/week4/tf_summaries" --port 6006
```

and go to [http://localhost:6006/#graphs (http://localhost:6006/#graphs)](http://localhost:6006/#graphs)

It should look something like this:



Loading the Corpus

As in the original paper, we'll train on the Brown corpus.

In [7]:
```
corpus = nltk.corpus.brown

token_feed = (utils.canonicalize_word(w) for w in corpus.words())
vocab = vocabulary.Vocabulary(token_feed, size=V)
```

In [8]:
```
# Train-test split
sentences = list(corpus.sents())
print "Loaded %d sentences (%g tokens)" % (len(sentences), sum(map(len, senten
ces)))

train_frac = 0.8
split_idx = int(train_frac * len(sentences))
train_sentences = sentences[:split_idx]
dev_sentences = sentences[split_idx:]

print "Sample: "
print " ".join(train_sentences[0])
```

```
Loaded 57340 sentences (1.16119e+06 tokens)
Sample:
The Fulton County Grand Jury said Friday an investigation of Atlanta's recent
 primary election produced `` no evidence '' that any irregularities took pla
ce .
```

As with the n-gram models in week 2, we'll canonicalize the words, and truncate our vocabulary to a fixed size. We'll also add a sentence boundary marker `<s>` in between sentences.

We'll also need to represent each word as a numerical id. The `Vocabulary` class already makes this mapping for us, so we can just use the utility functions to get our id list.

The function below will handle all of this for us:

In [9]:
```
def preprocess_sentences(sentences):
    # Add sentence boundaries, canonicalize, and handle unknowns
    words = ["<s>"]*N + utils.flatten(s + ["<s>"] for s in sentences)
    words = [utils.canonicalize_word(w, wordset=vocab.word_to_id)
             for w in words]
    return np.array(vocab.words_to_ids(words))

train_ids = preprocess_sentences(train_sentences)
dev_ids = preprocess_sentences(dev_sentences)

print "Sample words: "
print " ".join(vocab.ids_to_words(train_ids[:50]))
print ""
print "Sample words, as ids: "
print " ".join(map(str, train_ids[:50]))
```

```
Sample words:
<s> <s> <s> the <unk> county <unk> <unk> said <unk> an <unk> of <unk> recent
 <unk> <unk> <unk> `` no evidence '' that any <unk> took place . <s> the <unk
> further said in <unk> <unk> that the city <unk> committee , which had <unk>
 charge of the <unk> ,

Sample words, as ids:
0 0 0 3 2 655 2 2 65 2 37 2 6 2 553 2 2 2 16 61 477 17 11 90 2 217 177 5 0 3
 2 443 65 10 2 2 11 3 242 2 603 4 39 29 2 866 6 3 2 4
```

Our model is designed to accept batches of data, so we need to do a little re-formatting. We want our input batches to look like the following, where the first N columns are the inputs and the last is the target word:

In [10]:
```
cols = ["w_{i-%d}" % d for d in range(N,0,-1)] + ["target: w_i"]
M = np.array([[0,3,5613,655], [3,5613,655,2288], [5613,655,2288,1640]])
utils.pretty_print_matrix(M, cols=cols, dtype=int)
```

	w_{i-3}	w_{i-2}	w_{i-1}	target: w_i
0	0	3	5613	655
1	3	5613	655	2288
2	5613	655	2288	1640

We'll format our entire corpus like this, and then we can just sample blocks from it to get our training minibatches:

In [11]:
```
def build_windows(ids, shuffle=True):
    windows = np.zeros((len(ids)-N, N+1), dtype=int)
    for i in xrange(N+1):
        # First column: first word, etc.
        windows[:,i] = ids[i:len(ids)-(N-i)]
    if shuffle:
        # Shuffle rows
        np.random.shuffle(windows)
    return windows

train_windows = build_windows(train_ids)
dev_windows = build_windows(dev_ids)

# Check that we got what we want
# Just look at the first few IDs for this sample
utils.pretty_print_matrix(build_windows(train_ids[:(N+5)], shuffle=False),
                          cols=cols, dtype=int)
```

	w_{i-3}	w_{i-2}	w_{i-1}	target: w_i
0	0	0	0	3
1	0	0	3	2
2	0	3	2	655
3	3	2	655	2
4	2	655	2	2

Training time!

With our data in array form, we can train our model much like any machine learning model. The code below should look familiar - it's very similar to what we defined for the logistic regression demo. We'll factor out a few operations into helpers, so that the basic structure is clearer:

In [12]:
```
##
# Helper functions for training, to reduce boilerplate code

def train_batch(session, batch, alpha):
    feed_dict = {ids_:batch[:,:-1],
                 y_:batch[:,-1],
                 alpha_:alpha}
    c, _ = session.run([train_loss_, train_step_],
                       feed_dict=feed_dict)
    return c

def score_batch(session, batch):
    feed_dict = {ids_:batch[:,:-1],
                 y_:batch[:,-1]}
    return session.run(loss_, feed_dict=feed_dict)

def batch_generator(data, batch_size):
    """Generate minibatches from data."""
    for i in xrange(0, len(data), batch_size):
        yield data[i:i+batch_size]
```

Training a single epoch should take around 6-7 minutes on a 2-core Cloud Compute instance, or around 30 seconds on a GTX 980 GPU. You should get good results after just 2-3 epochs.

In [ ]:
```
# One epoch = one pass through the training data
num_epochs = 3
batch_size = 100
alpha = 0.1  # learning rate
print_every = 1000

np.random.seed(42)

session = tf.Session()
session.run(init_)

t0 = time.time()
for epoch in xrange(1,num_epochs+1):
    t0_epoch = time.time()
    epoch_cost = 0.0
    print ""
    for i, batch in enumerate(batch_generator(train_windows, batch_size)):
        if (i % print_every == 0):
            print "[epoch %d] seen %d minibatches" % (epoch, i)

        epoch_cost += train_batch(session, batch, alpha)

    avg_cost = epoch_cost / len(train_windows)
    print "[epoch %d] Completed %d minibatches in %s" % (epoch, i, utils.prett
y_timedelta(since=t0_epoch))
    print "[epoch %d] Average cost: %.03f" % (epoch, avg_cost,)
```

```
[epoch 1] seen 0 minibatches
[epoch 1] seen 1000 minibatches
[epoch 1] seen 2000 minibatches
[epoch 1] seen 3000 minibatches
[epoch 1] seen 4000 minibatches
[epoch 1] seen 5000 minibatches
[epoch 1] seen 6000 minibatches
```

Scoring

We'll score our model the same as the n-gram model, by computing perplexity over the dev set. Recall that perplexity is just the exponentiated average cross-entropy loss:

$$\text{Perplexity} = \left(\prod_i \frac{1}{Q(x_i)}\right)^{1/N} = \left(\prod_i 2^{-\log_2 Q(x_i)}\right)^{1/N} = 2^{\left(\frac{1}{N}\sum_i -\log_2 Q(x_i)\right)} = 2^{\tilde{CE}(P,Q)}$$

In [15]:
```
def score_dataset(data):
    total_cost = 0.0
    for batch in batch_generator(data, 1000):
        total_cost += score_batch(session, batch)

    avg_cost = total_cost / len(data)
    return avg_cost
```

In [16]:
```
print "Train set perplexity: %.03f" % 2**score_dataset(train_windows)
print "Dev set perplexity: %.03f" % 2**score_dataset(dev_windows)
```

```
Train set perplexity: 43.476
Dev set perplexity: 50.113
```

Looks pretty good! Note that these numbers aren't directly comparable to the literature, since we made the task easier by lowercasing everything, canonicalizing digits, and treating a fairly large number of words as an `<unk>` token.

We can remove some of this handicap by looking at our perplexity on non-`<unk>` target words:

In [17]:
```
filtered_dev_windows = dev_windows[dev_windows[:,-1] != vocab.UNK_ID]
print "Filtered dev set perplexity: %.03f" % 2**score_dataset(filtered_dev_win
dows)
```

```
Filtered dev set perplexity: 57.350
```

Sampling

We can sample sentences from the model much as we did with n-gram models. We'll use the `pred_random_` op that we defined before:

In [20]:
```
def predict_next(session, seq):
    feed_dict={ids_:np.array([seq[-N:]])}
    next_id = session.run(pred_random_, feed_dict=feed_dict)
    return next_id[0][0]

def score_seq(session, seq):
    # Some gymnastics to generate windows for scoring
    windows = [seq[i:i+N+1] for i in range(len(seq)-(N+1))]
    return score_batch(session, np.array(windows))

max_length = 30
num_sentences = 5

for _ in range(num_sentences):
    seq = [vocab.word_to_id["<s>"]]*N  # init N+1-gram model
    for i in range(max_length):
        seq.append(predict_next(session, seq))
        if seq[-1] == vocab.word_to_id["<s>"]: break
    print " ".join(vocab.ids_to_words(seq))
    score = score_seq(session, seq)
    print "[%d tokens; log P(seq): %.02f, per-token: %.02f]" % (len(seq), scor
e,
                                                                score/(len(seq
2))
    print ""
```

```
<s> <s> <s> was the best looking his art gives <unk> time . <s>
[14 tokens; log P(seq): 55.03, per-token: 4.59]

<s> <s> <s> they might have friends of up '' . <s>
[12 tokens; log P(seq): 34.53, per-token: 3.45]

<s> <s> <s> street , . <s>
[7 tokens; log P(seq): 14.06, per-token: 2.81]

<s> <s> <s> he had done . <s>
[8 tokens; log P(seq): 13.29, per-token: 2.22]

<s> <s> <s> he were interesting ' m. against the substrate with elected in ne
w correlation . <s>
[18 tokens; log P(seq): 65.96, per-token: 4.12]
```

In [ ]:

In [ ]: