CNTK_202_Language_Understanding

June 8, 2018

1 CNTK 202: Language Understanding with Recurrent Networks

This tutorial shows how to implement a recurrent network to process text, for the Air Travel Information Services (ATIS) task of slot tagging (tag individual words to their respective classes, where the classes are provided as labels in the training data set). We will start with a straight-forward (linear) embedding of the words followed by a recurrent LSTM. This will then be extended to include neighboring words and run bidirectionally. Lastly, we will turn this system into an intent classifier.

The techniques you will practice are: * model description by composing layer blocks, a convenient way to compose networks/models without requiring the need to write formulas, * creating your own layer block * variables with different sequence lengths in the same network * training the network

We assume that you are familiar with basics of deep learning, and these specific concepts: * recurrent networks (Wikipedia page) * text embedding (Wikipedia page)

1.0.1 Prerequisites

We assume that you have already installed CNTK. This tutorial requires CNTK V2. We strongly recommend to run this tutorial on a machine with a capable CUDA-compatible GPU. Deep learning without GPUs is not fun.

Downloading the data In this tutorial we are going to use a (lightly preprocessed) version of the ATIS dataset. You can download the data automatically by running the cells below or by executing the manual instructions.

Fallback manual instructions Download the ATIS training and test files and put them at the same folder as this notebook. If you want to see how the model is predicting on new sentences you will also need the vocabulary files for queries and slots

```
In [1]: from __future__ import print_function # Use a function definition from future version (s
        import requests
        import os

        def download(url, filename):
            """ utility function to download a file """
            response = requests.get(url, stream=True)
            with open(filename, "wb") as handle:
```

```
                for data in response.iter_content():
                    handle.write(data)

        locations = ['Tutorials/SLUHandsOn', 'Examples/LanguageUnderstanding/ATIS/BrainScript']

        data = {
          'train': { 'file': 'atis.train.ctf', 'location': 0 },
          'test': { 'file': 'atis.test.ctf', 'location': 0 },
          'query': { 'file': 'query.wl', 'location': 1 },
          'slots': { 'file': 'slots.wl', 'location': 1 }
        }

        for item in data.values():
            location = locations[item['location']]
            path = os.path.join('..', location, item['file'])
            if os.path.exists(path):
                print("Reusing locally cached:", item['file'])
                # Update path
                item['file'] = path
            elif os.path.exists(item['file']):
                print("Reusing locally cached:", item['file'])
            else:
                print("Starting download:", item['file'])
                url = "https://github.com/Microsoft/CNTK/blob/v2.0/%s/%s?raw=true"%(location, it
                download(url, item['file'])
                print("Download completed")
```

```
Starting download: query.wl
Download completed
Starting download: slots.wl
Download completed
Starting download: atis.train.ctf
Download completed
Starting download: atis.test.ctf
Download completed
```

Importing CNTK and other useful libraries CNTK's Python module contains several submodules like `io`, `learner`, and `layers`. We also use NumPy in some cases since the results returned by CNTK work like NumPy arrays.

```
In [2]: import math
        import numpy as np

        import cntk as C
```

In the block below, we check if we are running this notebook in the CNTK internal test machines by looking for environment variables defined there. We then select the right target device

(GPU vs CPU) to test this notebook. In other cases, we use CNTK's default policy to use the best available device (GPU, if available, else CPU).

```
In [3]: # Select the right target device when this notebook is being tested:
        if 'TEST_DEVICE' in os.environ:
            if os.environ['TEST_DEVICE'] == 'cpu':
                C.device.try_set_default_device(C.device.cpu())
            else:
                C.device.try_set_default_device(C.device.gpu(0))
```

```
In [4]: # Test for CNTK version
        if not C.__version__ == "2.0":
            raise Exception("this notebook was designed to work with 2.0. Current Version: " + C
```

1.1 Task and Model Structure

The task we want to approach in this tutorial is slot tagging. We use the ATIS corpus. ATIS contains human-computer queries from the domain of Air Travel Information Services, and our task will be to annotate (tag) each word of a query whether it belongs to a specific item of information (slot), and which one.

The data in your working folder has already been converted into the "CNTK Text Format." Let us look at an example from the test-set file `atis.test.ctf`:

```
19  |S0 178:1 |# BOS      |S1 14:1 |# flight  |S2 128:1 |# O
19  |S0 770:1 |# show                         |S2 128:1 |# O
19  |S0 429:1 |# flights                      |S2 128:1 |# O
19  |S0 444:1 |# from                         |S2 128:1 |# O
19  |S0 272:1 |# burbank                      |S2 48:1  |# B-fromloc.city_name
19  |S0 851:1 |# to                           |S2 128:1 |# O
19  |S0 789:1 |# st.                          |S2 78:1  |# B-toloc.city_name
19  |S0 564:1 |# louis                        |S2 125:1 |# I-toloc.city_name
19  |S0 654:1 |# on                           |S2 128:1 |# O
19  |S0 601:1 |# monday                       |S2 26:1  |# B-depart_date.day_name
19  |S0 179:1 |# EOS                          |S2 128:1 |# O
```

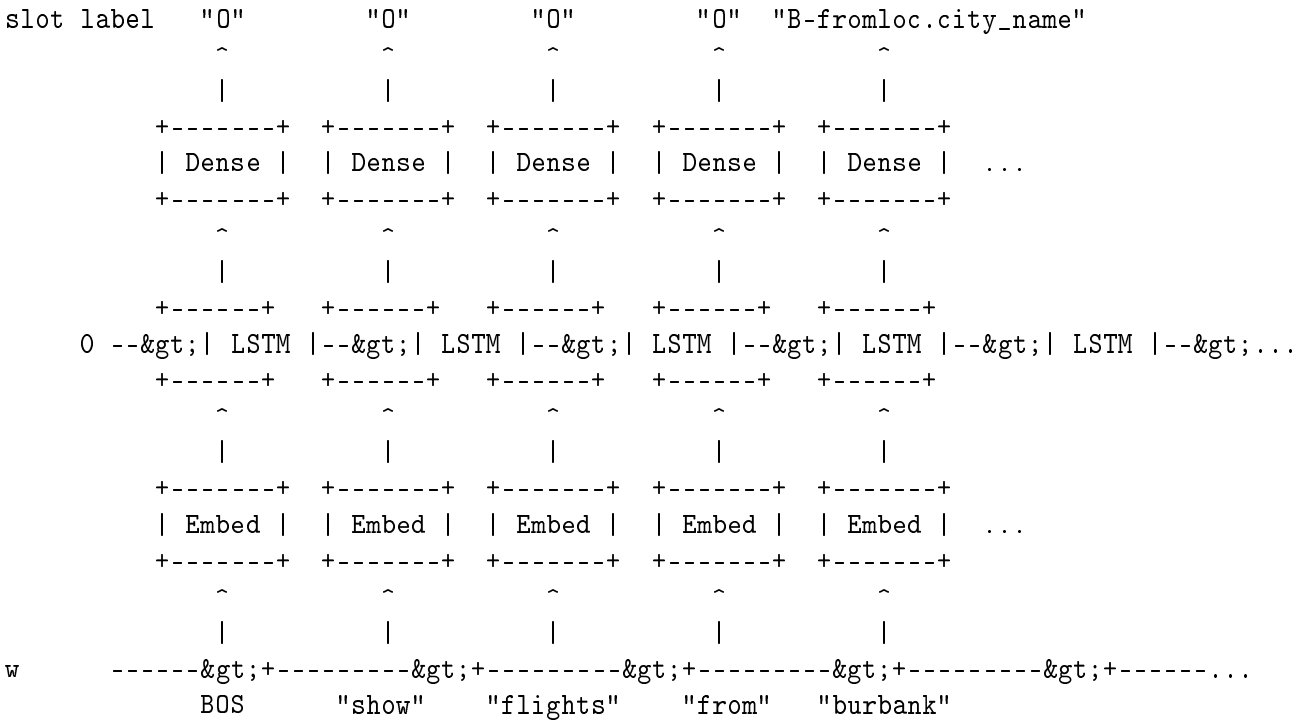
This file has 7 columns:

- a sequence id (19). There are 11 entries with this sequence id. This means that sequence 19 consists of 11 tokens;
- column `S0`, which contains numeric word indices; the input data is encoded in one-hot vectors. There are 943 words in the vocabulary, so each word is a 943 element vector of all 0 with a 1 at a vector index chosen to represent that word. For example the word "from" is represented with a 1 at index 444 and zero everywhere else in the vector. The word "monday" is represented with a 1 at index 601 and zero everywhere else in the vector.
- a comment column denoted by `#`, to allow a human reader to know what the numeric word index stands for; Comment columns are ignored by the system. `BOS` and `EOS` are special words to denote beginning and end of sentence, respectively;
- column `S1` is an intent label, which we will only use in the last part of the tutorial;
- another comment column that shows the human-readable label of the numeric intent index;

- column S2 is the slot label, represented as a numeric index; and
- another comment column that shows the human-readable label of the numeric label index.

The task of the neural network is to look at the query (column S0) and predict the slot label (column S2). As you can see, each word in the input gets assigned either an empty label O or a slot label that begins with B- for the first word, and with I- for any additional consecutive word that belongs to the same slot.

The model we will use is a recurrent model consisting of an embedding layer, a recurrent LSTM cell, and a dense layer to compute the posterior probabilities:



Or, as a CNTK network description. Please have a quick look and match it with the description above: (descriptions of these functions can be found at: the layers reference)

```
In [5]: # number of words in vocab, slot labels, and intent labels
        vocab_size = 943 ; num_labels = 129 ; num_intents = 26

        # model dimensions
        input_dim  = vocab_size
        label_dim  = num_labels
        emb_dim    = 150
        hidden_dim = 300

        # Create the containers for input feature (x) and the label (y)
        x = C.sequence.input_variable(vocab_size)
        y = C.sequence.input_variable(num_labels)

        def create_model():
            with C.layers.default_options(initial_state=0.1):
```

```
        return C.layers.Sequential([
            C.layers.Embedding(emb_dim, name='embed'),
            C.layers.Recurrence(C.layers.LSTM(hidden_dim), go_backwards=False),
            C.layers.Dense(num_labels, name='classify')
        ])
```

Now we are ready to create a model and inspect it.

The model attributes are fully accessible from Python. The first layer named `embed` is an Embedding layer. Here we use the CNTK default, which is linear embedding. It is a simple matrix with dimension (input word encoding x output projected dimension). You can access its parameter `E` (where the embeddings are stored) like any other attribute of a Python object. Its shape contains a `-1` which indicates that this parameter (with input dimension) is not fully specified yet, while the output dimension is set to `emb_dim` (= 150 in this tutorial).

Additionally we also inspect the value of the bias vector in the `Dense` layer named `classify`. The `Dense` layer is a fundamental compositional unit of a Multi-Layer Perceptron (as introduced in CNTK 103C tutorial). The `Dense` layer has both `weight` and `bias` parameters, one each per `Dense` layer. Bias terms are by default initialized to 0 (but there is a way to change that if you need). As you create the model, one should name the layer component and then access the parameters as shown here.

Suggested task: What should be the expected dimension of the `weight` matrix from the layer named `classify`? Try printing the weight matrix of the `classify` layer? Does it match with your expected size?

```
In [6]: # peek
        z = create_model()
        print(z.embed.E.shape)
        print(z.classify.b.value)
```

```
(-1, 150)
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.]
```

In our case we have input as one-hot encoded vector of length 943 and the output dimension `emb_dim` is set to 150. In the code below we pass the input variable `x` to our model `z`. This binds the model with input data of known shape. In this case, the input shape will be the size of the input vocabulary. With this modification, the parameter returned by the embed layer is completely specified (943, 150). **Note**: You can initialize the Embedding matrix with pre-computed vectors using Word2Vec or GloVe.

```
In [7]: # Pass an input and check the dimension
        z = create_model()
        print(z(x).embed.E.shape)
```

```
(943, 150)
```

1.2 CNTK Configuration

To train and test a model in CNTK, we need to create a model and specify how to read data and perform training and testing.

In order to train we need to specify:

- how to read the data
- the model function, its inputs, and outputs
- hyper-parameters for the learner such as the learning rate

1.2.1 A Brief Look at Data and Data Reading

We already looked at the data. But how do you generate this format? For reading text, this tutorial uses the `CNTKTextFormatReader`. It expects the input data to be in a specific format, as described here.

For this tutorial, we created the corpora by two steps: * convert the raw data into a plain text file that contains of TAB-separated columns of space-separated text. For example:

```
BOS show flights from burbank to st. louis on monday EOS (TAB) flight
(TAB) O O O O B-fromloc.city_name O B-toloc.city_name I-toloc.city_name O
B-depart_date.day_name O
```

This is meant to be compatible with the output of the `paste` command. * convert it to CNTK Text Format (CTF) with the following command:

```
python [CNTK root]/Scripts/txt2ctf.py --map query.wl intent.wl slots.wl
--annotated True --input atis.test.txt --output atis.test.ctf
```

where the three `.wl` files give the vocabulary as plain text files, one word per line.

In these CTF files, our columns are labeled `S0`, `S1`, and `S2`. These are connected to the actual network inputs by the corresponding lines in the reader definition:

```
In [8]: def create_reader(path, is_training):
            return C.io.MinibatchSource(C.io.CTFDeserializer(path, C.io.StreamDefs(
                 query         = C.io.StreamDef(field='S0', shape=vocab_size,  is_sparse=True),
                 intent_unused = C.io.StreamDef(field='S1', shape=num_intents, is_sparse=True),
                 slot_labels   = C.io.StreamDef(field='S2', shape=num_labels,  is_sparse=True)
             )), randomize=is_training, max_sweeps = C.io.INFINITELY_REPEAT if is_training else
```

```
In [9]: # peek
        reader = create_reader(data['train']['file'], is_training=True)
        reader.streams.keys()
```

```
Out[9]: dict_keys(['intent_unused', 'query', 'slot_labels'])
```

1.2.2 Trainer

We also must define the training criterion (loss function), and also an error metric to track. In most tutorials, we know the input dimensions and the corresponding labels. We directly create the loss

and the error functions. In this tutorial we will do the same. However, we take a brief detour and learn about placeholders. This concept would be useful for Task 3.

Learning note: Introduction to `placeholder`: Remember that the code we have been writing is not actually executing any heavy computation it is just specifying the function we want to compute on data during training/testing. And in the same way that it is convenient to have names for arguments when you write a regular function in a programming language, it is convenient to have placeholders that refer to arguments (or local computations that need to be reused). Eventually, some other code will replace these placeholders with other known quantities in the same way that in a programming language the function will be called with concrete values bound to its arguments.

Specifically, the input variables you have created above `x = C.sequence.input_variable(vocab_size)` holds data pre-defined by `vocab_size`. In the case where such instantiations are challenging or not possible, using `placeholder` is a logical choice. Having the `placeholder` only allows you to defer the specification of the argument at a later time when you may have the data.

Here is an example below that illustrates the use of `placeholder`.

```
In [10]: def create_criterion_function(model):
             labels = C.placeholder(name='labels')
             ce   = C.cross_entropy_with_softmax(model, labels)
             errs = C.classification_error      (model, labels)
             return C.combine ([ce, errs]) # (features, labels) -> (loss, metric)

         criterion = create_criterion_function(create_model())
         criterion.replace_placeholders({criterion.placeholders[0]: C.sequence.input_variable(nu

Out[10]: Composite(Combine): Input('Input2300', [#, *], [129]), Placeholder('labels', [???], [??
```

While the cell above works well when one has input parameters defined at network creation, it compromises readability. Hence we prefer creating functions as shown below

```
In [11]: def create_criterion_function_preferred(model, labels):
             ce   = C.cross_entropy_with_softmax(model, labels)
             errs = C.classification_error      (model, labels)
             return ce, errs # (model, labels) -> (loss, error metric)

In [12]: def train(reader, model_func, max_epochs=10):

             # Instantiate the model function; x is the input (feature) variable
             model = model_func(x)

             # Instantiate the loss and error function
             loss, label_error = create_criterion_function_preferred(model, y)

             # training config
             epoch_size = 18000        # 18000 samples is half the dataset size
             minibatch_size = 70

             # LR schedule over epochs
```

```
# In CNTK, an epoch is how often we get out of the minibatch loop to
# do other stuff (e.g. checkpointing, adjust learning rate, etc.)
# (we don't run this many epochs, but if we did, these are good values)
lr_per_sample = [0.003]*4+[0.0015]*24+[0.0003]
lr_per_minibatch = [lr * minibatch_size for lr in lr_per_sample]
lr_schedule = C.learning_rate_schedule(lr_per_minibatch, C.UnitType.minibatch, epoc

# Momentum schedule
momentum_as_time_constant = C.momentum_as_time_constant_schedule(700)

# We use a the Adam optimizer which is known to work well on this dataset
# Feel free to try other optimizers from
# https://www.cntk.ai/pythondocs/cntk.learner.html#module-cntk.learner
learner = C.adam(parameters=model.parameters,
                 lr=lr_schedule,
                 momentum=momentum_as_time_constant,
                 gradient_clipping_threshold_per_sample=15,
                 gradient_clipping_with_truncation=True)

# Setup the progress updater
progress_printer = C.logging.ProgressPrinter(tag='Training', num_epochs=max_epochs)

# Uncomment below for more detailed logging
#progress_printer = ProgressPrinter(freq=100, first=10, tag='Training', num_epochs=

# Instantiate the trainer
trainer = C.Trainer(model, (loss, label_error), learner, progress_printer)

# process minibatches and perform model training
C.logging.log_number_of_parameters(model)

t = 0
for epoch in range(max_epochs):         # loop over epochs
    epoch_end = (epoch+1) * epoch_size
    while t < epoch_end:                # loop over minibatches on the epoch
        data = reader.next_minibatch(minibatch_size, input_map={  # fetch minibatch
            x: reader.streams.query,
            y: reader.streams.slot_labels
        })
        trainer.train_minibatch(data)               # update model with it
        t += data[y].num_samples                    # samples so far
    trainer.summarize_training_progress()
```

1.2.3 Running it

You can find the complete recipe below.

```
In [13]: def do_train():
             global z
             z = create_model()
             reader = create_reader(data['train']['file'], is_training=True)
             train(reader, z)
         do_train()
```

```
Training 721479 parameters in 6 parameter tensors.
Learning rate per minibatch: 0.21
Finished Epoch[1 of 10]: [Training] loss = 0.822525 * 18010, metric = 16.20% * 18010 70.657s (25
Finished Epoch[2 of 10]: [Training] loss = 0.237265 * 18051, metric = 5.25% * 18051 80.600s (224
Finished Epoch[3 of 10]: [Training] loss = 0.155346 * 17941, metric = 3.55% * 17941 74.494s (240
Finished Epoch[4 of 10]: [Training] loss = 0.114119 * 18059, metric = 2.72% * 18059 80.340s (224
Learning rate per minibatch: 0.105
Finished Epoch[5 of 10]: [Training] loss = 0.075465 * 17957, metric = 1.83% * 17957 76.424s (235
Finished Epoch[6 of 10]: [Training] loss = 0.064977 * 18021, metric = 1.56% * 18021 70.294s (256
Finished Epoch[7 of 10]: [Training] loss = 0.059071 * 17980, metric = 1.45% * 17980 74.459s (241
Finished Epoch[8 of 10]: [Training] loss = 0.052832 * 18025, metric = 1.43% * 18025 53.009s (340
Finished Epoch[9 of 10]: [Training] loss = 0.034388 * 17956, metric = 0.95% * 17956 54.130s (331
Finished Epoch[10 of 10]: [Training] loss = 0.033605 * 18039, metric = 0.86% * 18039 51.278s (35
```

This shows how learning proceeds over epochs (passes through the data). For example, after four epochs, the loss, which is the cross-entropy criterion, has reached 0.11 as measured on the ~18000 samples of this epoch, and that the error rate is 2.6% on those same 18000 training samples.

The epoch size is the number of samples--counted as *word tokens*, not sentences--to process between model checkpoints.

Once the training has completed (a little less than 2 minutes on a Titan-X or a Surface Book), you will see an output like this

```
Finished Epoch [10]: [Training] loss = 0.033263 * 18039, metric = 0.9% * 18039
```

which is the loss (cross entropy) and the metric (classification error) averaged over the final epoch.

On a CPU-only machine, it can be 4 or more times slower. You can try setting

```
emb_dim    = 50
hidden_dim = 100
```

to reduce the time it takes to run on a CPU, but the model will not fit as well as when the hidden and embedding dimension are larger.

1.2.4 Evaluating the model

Like the train() function, we also define a function to measure accuracy on a test set by computing the error over multiple minibatches of test data. For evaluating on a small sample read from a file, you can set a minibatch size reflecting the sample size and run the test_minibatch on that instance of data. To see how to evaluate a single sequence, we provide an instance later in the tutorial.

```
In [14]: def evaluate(reader, model_func):

             # Instantiate the model function; x is the input (feature) variable
             model = model_func(x)

             # Create the loss and error functions
             loss, label_error = create_criterion_function_preferred(model, y)

             # process minibatches and perform evaluation
             progress_printer = C.logging.ProgressPrinter(tag='Evaluation', num_epochs=0)

             while True:
                 minibatch_size = 500
                 data = reader.next_minibatch(minibatch_size, input_map={  # fetch minibatch
                     x: reader.streams.query,
                     y: reader.streams.slot_labels
                 })
                 if not data:                                 # until we hit the end
                     break

                 evaluator = C.eval.Evaluator(loss, progress_printer)
                 evaluator.test_minibatch(data)

             evaluator.summarize_test_progress()
```

Now we can measure the model accuracy by going through all the examples in the test set and using the `test_minibatch` method of the trainer created inside the evaluate function defined above. At the moment (when this tutorial was written) the Trainer constructor requires a learner (even if it is only used to perform `test_minibatch`) so we have to specify a dummy learner. In the future it will be allowed to construct a Trainer without specifying a learner as long as the trainer only calls `test_minibatch`

```
In [15]: def do_test():
             reader = create_reader(data['test']['file'], is_training=False)
             evaluate(reader, z)
         do_test()
         z.classify.b.value
```

```
Finished Evaluation [1]: Minibatch[1-23]: metric = 0.34% * 10984;
```

```
Out[15]: array([ -1.92054473e-02,  -9.52135623e-02,   1.08148195e-02,
                  3.52970250e-02,  -2.06884108e-02,   2.02884115e-02,
                 -1.74519829e-02,  -1.06526069e-01,   5.82500082e-03,
                 -2.45601051e-02,   5.50178625e-02,   5.40859737e-02,
                 -7.46583790e-02,   6.81229532e-02,  -8.45809132e-02,
                 -1.09936722e-01,  -8.47302303e-02,  -6.53339624e-02,
                 -9.58734564e-03,  -9.30738449e-02,  -2.16507241e-02,
                 -9.10663977e-02,   4.83943634e-02,   2.84409784e-02,
```

```
       -1.02759211e-03,  -9.49962996e-04,   2.27191672e-02,
       -1.18115388e-01,   1.36329997e-02,  -5.19461595e-02,
       -4.95415181e-02,   6.05669171e-02,  -1.95788629e-02,
        9.75460280e-03,   6.31383657e-02,   4.28247340e-02,
       -4.01832834e-02,  -4.06631008e-02,   6.23724423e-03,
        2.49000196e-03,  -8.29084665e-02,  -5.62636964e-02,
        2.72212885e-02,  -4.69312593e-02,  -4.33527865e-02,
       -5.77107519e-02,   4.47729304e-02,   4.94898669e-02,
        4.10377346e-02,  -4.54721190e-02,  -3.83926220e-02,
       -3.44851427e-02,   4.62128855e-02,  -5.76279722e-02,
        4.11057770e-02,   4.13490571e-02,  -4.38151136e-02,
       -3.95155139e-02,  -4.04885560e-02,  -3.56947817e-02,
        4.32800800e-02,  -2.65491866e-02,   2.15679596e-04,
       -2.90235714e-03,   1.05655435e-02,   2.27081031e-02,
       -1.24581560e-01,  -4.21190225e-02,   1.66358277e-02,
       -9.51309800e-02,  -1.85963232e-02,  -2.49189567e-02,
       -5.94329182e-03,   7.57769728e-03,  -5.39907701e-02,
        2.52893306e-02,   4.42031547e-02,   3.49311829e-02,
        2.57153288e-02,   7.01648975e-03,  -3.02192140e-02,
       -2.81920861e-02,  -3.58696766e-02,  -1.40104100e-01,
       -7.58493468e-02,  -2.42353417e-02,   6.56584976e-03,
        3.69094312e-02,   1.06784925e-02,  -1.01001427e-01,
       -1.98039617e-02,  -8.59586895e-02,  -1.10523738e-01,
       -1.61005080e-01,  -5.55543695e-03,  -1.28297240e-01,
        6.98479190e-02,  -2.51891222e-02,  -1.90122817e-02,
       -5.81395701e-02,  -2.03327909e-02,  -5.15457131e-02,
       -7.28933737e-02,  -9.92533341e-02,  -1.30062079e-04,
        3.32506709e-02,  -8.82507488e-02,  -4.63882200e-02,
       -9.63402539e-02,  -3.74367125e-02,  -8.24951902e-02,
        1.31833255e-02,  -8.93121399e-03,   3.16066332e-02,
        2.25433540e-02,  -9.88071412e-02,   3.78055982e-02,
        1.75067689e-02,   1.69068221e-02,  -1.69945940e-01,
       -9.24684554e-02,  -5.47090098e-02,  -4.84931618e-02,
        3.60509492e-02,  -1.83330197e-02,  -3.83731760e-02,
       -6.26878766e-03,  -7.09694251e-02,   6.12073764e-02], dtype=float32)
```

The following block of code illustrates how to evaluate a single sequence. Additionally we show how one can pass in the information using NumPy arrays.

```
In [16]: # load dictionaries
         query_wl = [line.rstrip('\n') for line in open(data['query']['file'])]
         slots_wl = [line.rstrip('\n') for line in open(data['slots']['file'])]
         query_dict = {query_wl[i]:i for i in range(len(query_wl))}
         slots_dict = {slots_wl[i]:i for i in range(len(slots_wl))}

         # let's run a sequence through
         seq = 'BOS flights from new york to seattle EOS'
         w = [query_dict[w] for w in seq.split()] # convert to word indices
```

```
        print(w)
        onehot = np.zeros([len(w),len(query_dict)], np.float32)
        for t in range(len(w)):
            onehot[t,w[t]] = 1

        #x = C.sequence.input_variable(vocab_size)
        pred = z(x).eval({x:[onehot]})[0]
        print(pred.shape)
        best = np.argmax(pred,axis=1)
        print(best)
        list(zip(seq.split(),[slots_wl[s] for s in best]))
```

```
[178, 429, 444, 619, 937, 851, 752, 179]
(8, 129)
[128 128 128  48 110 128  78 128]
```

```
Out[16]: [('BOS', 'O'),
          ('flights', 'O'),
          ('from', 'O'),
          ('new', 'B-fromloc.city_name'),
          ('york', 'I-fromloc.city_name'),
          ('to', 'O'),
          ('seattle', 'B-toloc.city_name'),
          ('EOS', 'O')]
```

1.3 Modifying the Model

In the following, you will be given tasks to practice modifying CNTK configurations. The solutions are given at the end of this document... but please try without!

1.3.1 A Word About `Sequential()`

Before jumping to the tasks, let's have a look again at the model we just ran. The model is described in what we call *function-composition style*.

```
Sequential([
    Embedding(emb_dim),
    Recurrence(LSTM(hidden_dim), go_backwards=False),
    Dense(num_labels)
])
```

You may be familiar with the "sequential" notation from other neural-network toolkits. If not, `Sequential()` is a powerful operation that, in a nutshell, allows to compactly express a very common situation in neural networks where an input is processed by propagating it through a progression of layers. `Sequential()` takes an list of functions as its argument, and returns a *new* function that invokes these functions in order, each time passing the output of one to the next. For example,

```
FGH = Sequential ([F,G,H])
y = FGH (x)
```

means the same as

```
y = H(G(F(x)))
```

This is known as "function composition", and is especially convenient for expressing neural networks, which often have this form:

```
     +-------+   +-------+   +-------+
x --&gt;|   F   |--&gt;|   G   |--&gt;|   H   |--&gt; y
     +-------+   +-------+   +-------+
```

Coming back to our model at hand, the `Sequential` expression simply says that our model has this form:

```
     +-----------+   +----------------+   +------------+
x --&gt;| Embedding |--&gt;| Recurrent LSTM |--&gt;| DenseLayer |--&gt; y
     +-----------+   +----------------+   +------------+
```

1.3.2 Task 1: Add Batch Normalization

We now want to add new layers to the model, specifically batch normalization.

Batch normalization is a popular technique for speeding up convergence. It is often used for image-processing setups, for example our other hands-on lab on image recognition. But could it work for recurrent models, too?

> Note: training with Batch Normalization is currently only supported on GPU.

So your task will be to insert batch-normalization layers before and after the recurrent LSTM layer. If you have completed the hands-on labs on image processing, you may remember that the batch-normalization layer has this form:

```
BatchNormalization()
```

So please go ahead and modify the configuration and see what happens.

If everything went right, you will notice improved convergence speed (`loss` and `metric`) compared to the previous configuration.

```
In [17]: # Your task: Add batch normalization
         def create_model():
             with C.layers.default_options(initial_state=0.1):
                 return C.layers.Sequential([
                     C.layers.Embedding(emb_dim),
                     C.layers.Recurrence(C.layers.LSTM(hidden_dim), go_backwards=False),
                     C.layers.Dense(num_labels)
                 ])

         # Enable these when done:
         z = create_model()
         #do_train()
         #do_test()
```

1.3.3 Task 2: Add a Lookahead

Our recurrent model suffers from a structural deficit: Since the recurrence runs from left to right, the decision for a slot label has no information about upcoming words. The model is a bit lopsided. Your task will be to modify the model such that the input to the recurrence consists not only of the current word, but also of the next one (lookahead).

Your solution should be in function-composition style. Hence, you will need to write a Python function that does the following:

- takes no input arguments
- creates a placeholder (sequence) variable
- computes the "next value" in this sequence using the `sequence.future_value()` operation and
- concatenates the current and the next value into a vector of twice the embedding dimension using `splice()`

and then insert this function into `Sequential()`'s list right after the embedding layer.

```
In [18]: # Your task: Add lookahead
         def create_model():
             with C.layers.default_options(initial_state=0.1):
                 return C.layers.Sequential([
                     C.layers.Embedding(emb_dim),
                     C.layers.Recurrence(C.layers.LSTM(hidden_dim), go_backwards=False),
                     C.layers.Dense(num_labels)
                 ])

         # Enable these when done:
         z = create_model()
         #do_train()
         #do_test()
```

1.3.4 Task 3: Bidirectional Recurrent Model

Aha, knowledge of future words help. So instead of a one-word lookahead, why not look ahead until all the way to the end of the sentence, through a backward recurrence? Let us create a bidirectional model!

Your task is to implement a new layer that performs both a forward and a backward recursion over the data, and concatenates the output vectors.

Note, however, that this differs from the previous task in that the bidirectional layer contains learnable model parameters. In function-composition style, the pattern to implement a layer with model parameters is to write a *factory function* that creates a *function object*.

A function object, also known as *functor*, is an object that is both a function and an object. Which means nothing else that it contains data yet still can be invoked as if it was a function.

For example, `Dense(outDim)` is a factory function that returns a function object that contains a weight matrix `W`, a bias `b`, and another function to compute `input @ W + b`. (This is using Python 3.5 notation for matrix multiplication. In Numpy syntax it is `input.dot(W) + b`). E.g. saying `Dense(1024)` will create this function object, which can then be used like any other function, also immediately: `Dense(1024)(x)`.

Let's look at an example for further clarity: Let us implement a new layer that combines a linear layer with a subsequent batch normalization. To allow function composition, the layer needs to be realized as a factory function, which could look like this:

```
def DenseLayerWithBN(dim):
    F = Dense(dim)
    G = BatchNormalization()
    x = placeholder()
    apply_x = G(F(x))
    return apply_x
```

Invoking this factory function will create `F`, `G`, `x`, and `apply_x`. In this example, `F` and `G` are function objects themselves, and `apply_x` is the function to be applied to the data. Thus, e.g. calling `DenseLayerWithBN(1024)` will create an object containing a linear-layer function object called `F`, a batch-normalization function object `G`, and `apply_x` which is the function that implements the actual operation of this layer using `F` and `G`. It will then return `apply_x`. To the outside, `apply_x` looks and behaves like a function. Under the hood, however, `apply_x` retains access to its specific instances of `F` and `G`.

Now back to our task at hand. You will now need to create a factory function, very much like the example above. You shall create a factory function that creates two recurrent layer instances (one forward, one backward), and then defines an `apply_x` function which applies both layer instances to the same `x` and concatenate the two results.

Alright, give it a try! To know how to realize a backward recursion in CNTK, please take a hint from how the forward recursion is done. Please also do the following: * remove the one-word lookahead you added in the previous task, which we aim to replace; and * make sure each LSTM is using `hidden_dim//2` outputs to keep the total number of model parameters limited.

```
In [19]: # Your task: Add bidirectional recurrence
         def create_model():
             with C.layers.default_options(initial_state=0.1):
                 return C.layers.Sequential([
                     C.layers.Embedding(emb_dim),
                     C.layers.Recurrence(C.layers.LSTM(hidden_dim), go_backwards=False),
                     C.layers.Dense(num_labels)
                 ])

         # Enable these when done:
         #do_train()
         #do_test()
```

Works like a charm! This model achieves 0.32%, better than the lookahead model above. The bidirectional model has 40% less parameters than the lookahead one. However, if you go back and look closely you may find that the lookahead one trained about 30% faster. This is because the lookahead model has both less horizontal dependencies (one instead of two recurrences) and larger matrix products, and can thus achieve higher parallelism.

1.3.5 Solution 1: Adding Batch Normalization

```
In [20]: def create_model():
             with C.layers.default_options(initial_state=0.1):
```

```
            return C.layers.Sequential([
                C.layers.Embedding(emb_dim),
                #C.layers.BatchNormalization(),
                C.layers.Recurrence(C.layers.LSTM(hidden_dim), go_backwards=False),
                #C.layers.BatchNormalization(),
                C.layers.Dense(num_labels)
            ])

        do_train()
        do_test()
```

```
Training 721479 parameters in 6 parameter tensors.
Learning rate per minibatch: 0.21
Finished Epoch[1 of 10]: [Training] loss = 0.818963 * 18010, metric = 16.14% * 18010 2.665s (675
Finished Epoch[2 of 10]: [Training] loss = 0.235080 * 18051, metric = 5.26% * 18051 3.092s (5838
Finished Epoch[3 of 10]: [Training] loss = 0.154144 * 17941, metric = 3.51% * 17941 3.113s (5763
Finished Epoch[4 of 10]: [Training] loss = 0.108188 * 18059, metric = 2.60% * 18059 2.806s (6435
Learning rate per minibatch: 0.105
Finished Epoch[5 of 10]: [Training] loss = 0.070542 * 17957, metric = 1.62% * 17957 3.564s (5038
Finished Epoch[6 of 10]: [Training] loss = 0.063522 * 18021, metric = 1.59% * 18021 2.893s (6229
Finished Epoch[7 of 10]: [Training] loss = 0.054584 * 17980, metric = 1.36% * 17980 2.948s (6099
Finished Epoch[8 of 10]: [Training] loss = 0.048787 * 18025, metric = 1.23% * 18025 2.759s (6533
Finished Epoch[9 of 10]: [Training] loss = 0.031695 * 17956, metric = 0.94% * 17956 3.758s (4778
Finished Epoch[10 of 10]: [Training] loss = 0.031626 * 18039, metric = 0.83% * 18039 2.585s (697
Finished Evaluation [1]: Minibatch[1-23]: metric = 0.46% * 10984;
```

1.3.6 Solution 2: Add a Lookahead

```
In [21]: def OneWordLookahead():
             x = C.placeholder()
             apply_x = C.splice(x, C.sequence.future_value(x))
             return apply_x

         def create_model():
             with C.layers.default_options(initial_state=0.1):
                 return C.layers.Sequential([
                     C.layers.Embedding(emb_dim),
                     OneWordLookahead(),
                     C.layers.Recurrence(C.layers.LSTM(hidden_dim), go_backwards=False),
                     C.layers.Dense(num_labels)
                 ])

         do_train()
         do_test()
```

```
Training 901479 parameters in 6 parameter tensors.
Learning rate per minibatch: 0.21
```

```
Finished Epoch[1 of 10]: [Training] loss = 0.617353 * 18010, metric = 12.35% * 18010 2.766s (651
Finished Epoch[2 of 10]: [Training] loss = 0.170461 * 18051, metric = 3.90% * 18051 3.671s (4917
Finished Epoch[3 of 10]: [Training] loss = 0.108302 * 17941, metric = 2.34% * 17941 2.704s (6635
Finished Epoch[4 of 10]: [Training] loss = 0.073717 * 18059, metric = 1.74% * 18059 3.357s (5379
Learning rate per minibatch: 0.105
Finished Epoch[5 of 10]: [Training] loss = 0.042322 * 17957, metric = 1.05% * 17957 4.121s (4357
Finished Epoch[6 of 10]: [Training] loss = 0.037643 * 18021, metric = 0.87% * 18021 3.492s (5160
Finished Epoch[7 of 10]: [Training] loss = 0.036649 * 17980, metric = 0.88% * 17980 2.881s (6240
Finished Epoch[8 of 10]: [Training] loss = 0.025395 * 18025, metric = 0.66% * 18025 3.216s (5604
Finished Epoch[9 of 10]: [Training] loss = 0.014108 * 17956, metric = 0.36% * 17956 2.952s (6082
Finished Epoch[10 of 10]: [Training] loss = 0.013558 * 18039, metric = 0.34% * 18039 3.062s (589
Finished Evaluation [1]: Minibatch[1-23]: metric = 0.40% * 10984;
```

1.3.7 Solution 3: Bidirectional Recurrent Model

```
In [22]: def BiRecurrence(fwd, bwd):
             F = C.layers.Recurrence(fwd)
             G = C.layers.Recurrence(bwd, go_backwards=True)
             x = C.placeholder()
             apply_x = C.splice(F(x), G(x))
             return apply_x

         def create_model():
             with C.layers.default_options(initial_state=0.1):
                 return C.layers.Sequential([
                     C.layers.Embedding(emb_dim),
                     BiRecurrence(C.layers.LSTM(hidden_dim//2),
                                  C.layers.LSTM(hidden_dim//2)),
                     C.layers.Dense(num_labels)
                 ])

         do_train()
         do_test()
```

```
Training 541479 parameters in 9 parameter tensors.
Learning rate per minibatch: 0.21
Finished Epoch[1 of 10]: [Training] loss = 0.714729 * 18010, metric = 13.61% * 18010 4.189s (429
Finished Epoch[2 of 10]: [Training] loss = 0.169666 * 18051, metric = 3.86% * 18051 3.622s (4983
Finished Epoch[3 of 10]: [Training] loss = 0.101591 * 17941, metric = 2.30% * 17941 4.458s (4024
Finished Epoch[4 of 10]: [Training] loss = 0.070635 * 18059, metric = 1.68% * 18059 3.922s (4604
Learning rate per minibatch: 0.105
Finished Epoch[5 of 10]: [Training] loss = 0.041811 * 17957, metric = 0.95% * 17957 3.664s (4900
Finished Epoch[6 of 10]: [Training] loss = 0.037363 * 18021, metric = 0.91% * 18021 3.798s (4744
Finished Epoch[7 of 10]: [Training] loss = 0.038099 * 17980, metric = 0.91% * 17980 3.616s (4972
Finished Epoch[8 of 10]: [Training] loss = 0.026900 * 18025, metric = 0.72% * 18025 3.879s (4646
Finished Epoch[9 of 10]: [Training] loss = 0.017272 * 17956, metric = 0.45% * 17956 3.922s (4578
Finished Epoch[10 of 10]: [Training] loss = 0.016888 * 18039, metric = 0.49% * 18039 4.337s (415
```

```
Finished Evaluation [1]: Minibatch[1-23]: metric = 0.24% * 10984;
```