Recurrent Networks in Lasagne

Lasagne incudes classes for a "vanilla" (densely connected) recurrent layer, a recurrent layer with arbitrary input-to-hidden and hidden-to-hidden connections, an LSTM layer, and a GRU layer. The usage is a little different due to the fact that the network expects batches time sequences as input; here we will demonstrate usage on a common longterm memory benchmark.

```
In [1]: import theano
        import theano.tensor as T
        import lasagne
        import numpy as np
        import sklearn.datasets
              _future__ import print_function
        from
        import os
        import matplotlib.pyplot as plt
        %matplotlib inline
        import IPython.display
        IPython.display.Image("http://static-vegetariantimes.s3.amazonaws.com/wp-content/uploads/2009/03/10851medium.jpg")
```

Out[1]:



The "add" task In "Long short-term memory", the paper which proposed the LSTM unit, a handful of tasks are proposed which are meant to test a model's ability to handle long-term

dependencies. Here, we will use the "add" task, in which a two-dimensional time series is presented to the model where one dimension is sampled uniformly from [0, 1] and the other is all zeros except at two samples where it is 1. The goal for each time series is to output the sum of the values in the first dimension at the indices where the second dimension is 1. For example, the target for the following $\mid \ 0.5 \ \mid \ 0.7 \ \mid \ 0.3 \ \mid \ 0.1 \ \mid \ 0.2 \ \mid \ \dots \ \mid \ 0.5 \ \mid \ 0.9 \ \mid \ \dots \ \mid \ 0.8 \ \mid \ 0.2 \ \mid$

```
would be 0.3 + .9 = 1.2. The recurrent.py example includes a function gen_data which generates data for this task; we'll use this function here.
```

| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

```
In [21]: from recurrent import gen_data
```

Because recurrent layers are used in tasks where the sequential nature of the input data matters, an additional dimension is needed beyond the (batch_size, n_features_1, n_features_2, ...) shape expected by convolutional and dense layers. We could, of course, use the first dimension as the "number of sequence

Recurrent layer shape conventions

So, the convention used in lasagne is that sequential data is presented in the shape (batch_size, n_time_steps, n_features_1, n_features_2, ...). Masks Because not all sequences in each minibatch will always have the same length, all recurrent layers in lasagne accept a separate mask input which has shape

steps" dimension, but that would mean that we could not present minibatches of sequences to the network, which would slow down training and probably hurt convergence.

$(batch_size, n_time_steps), which is populated such that mask[i, j] = 1 when j <= (length of sequence i) and mask[i, j] = 0 when j > (length of sequence i) and when j > (length of sequence i) and when j > (length of sequence i) and when j > (length of sequenc$

of sequence i). When no mask is provided, it is assumed that all sequences in the minibatch are of length n_{time_steps} . Variable number of time steps

Finally, as is true of the first (batch_size) dimension, the n_time_steps dimension can be set to None, which means that it can vary from batch to batch. This means that

the network can take in minibatches which have an arbitrary number of sequences which are of arbitrary length - very convenient! In [22]: # By setting the first and second dimensions to None, we allow

```
# arbitrary minibatch sizes with arbitrary sequence lengths.
# The number of feature dimensions is 2, as described above.
l_in = lasagne.layers.InputLayer(shape=(None, None, 2))
# This input will be used to provide the network with masks.
# Masks are expected to be matrices of shape (n_batch, n_time_steps);
# both of these dimensions are variable for us so we will use
# an input shape of (None, None)
l_mask = lasagne.layers.InputLayer(shape=(None, None))
```

The interface for RecurrentLayer, GRULayer, and LSTMLayer are all pretty similar. Here, we will be using the LSTMLayer, which has the most construction arguments;

Recurrent layers

usage of the other layers will generally be a little simpler! Parameter initialization

In order to cut down on the number of constructor arguments and make changing initialization schemes more convenient, the GRULayer and LSTMLayer utilize the Gate

class, which is essentially just a container for parameter initializers. The LSTMLayer initializer accepts four Gate instances - one for the input gate, one for the forget gate, one for the cell, and one for the output gate. Here, we will demonstrate how to change the weight initialization for all gates to use orthogonal initialization, which (anecdotally) can facilitate learning in RNNs. Note that the cell "gate" doesn't use a weight matrix for a cell connection (that wouldn't make sense), and it uses a different nonlinearity. In [23]: # All gates have initializers for the input-to-gate and hidden state-to-gate

weight matrices, the cell-to-gate weight vector, the bias vector, and the nonlinearity.

```
# The convention is that gates use the standard sigmoid nonlinearity,
# which is the default for the Gate class.
gate_parameters = lasagne.layers.recurrent.Gate(
                  W_in=lasagne.init.Orthogonal(), W_hid=lasagne.init.Orthogonal(),
                 b=lasagne.init.Constant(0.))
cell_parameters = lasagne.layers.recurrent.Gate(
                \label{eq:w_index} \begin{tabular}{ll} $W\_in=lasagne.init.Orthogonal(), & M\_hid=lasagne.init.Orthogonal(), & M\_hid=lasa
                  # Setting W_cell to None denotes that no cell connection will be used.
                W cell=None, b=lasagne.init.Constant(0.),
                  # By convention, the cell nonlinearity is tanh in an LSTM.
                 nonlinearity=lasagne.nonlinearities.tanh)
```

Cell and hidden state initialization

LSTMLayer options

The layer's state at the beginning of the sequence can be set by passing initializers to the cell_init and hid_init arguments; by default, they are initialized to zeros. Somtimes these initial values are learned as parameters of the network, sometimes they aren't; you can decide whether you want to include them as parameters to be

Apart from taking in the layer's input connection, the number of units, and Gate instances for parameter initialization, the LSTMLayer has a variety of other constructor

arguments which determine the behavior of the layer. As mentioned above, many of these arguments are shared by the other recurrent layers.

learned by setting the $learn_init$ argument to True or False.

"Peephole" connections A common augmentation to the originally propsed LSTM architecture is to include connections from the cell state to the gates. Some people use this, some people don't, you

Truncating the number of backpropagation steps

The de facto method for training recurrent networks is backpropagation through time, which simply unrolls the network across timesteps and treats it as a network which is repeated for each time step. This can result in an incredibly "deep" and computationally expensive network, so it's common practice to truncate the number of unrolled sequence steps. This can be controlled with the gradient_steps argument; when it's -1 (the default), this means "don't truncate".

Clipping gradients

In [24]: # Our LSTM will have 10 hidden/cell units

l_in, N_HIDDEN,

mask_input=l_mask,

Combining recurrent and feed-forward layers

1_lstm = lasagne.layers.recurrent.LSTMLayer(

In [25]: # The "backwards" layer is the same as the first,

except that the backwards argument is set to True. 1_lstm_back = lasagne.layers.recurrent.LSTMLayer(

included with lasagne) which is a bit more efficient. In this tutorial, we'll do it with the ReshapeLayer for illustration.

In [26]: # First, retrieve symbolic variables for the input shape

Now, we can apply feed-forward layers as usual.

l dense = lasagne.layers.DenseLayer(

so we simply slice it out.

predicted_values = network output[:, -1] # Our cost will be mean-squared error

Epoch 1 validation cost = 0.168619907953 Epoch 2 validation cost = 0.155535056081 Epoch 3 validation cost = 0.0738346717391 Epoch 4 validation cost = 0.0452366159954 Epoch 5 validation cost = 0.0324324088071 Epoch 6 validation cost = 0.030376732186 Epoch 7 validation cost = 0.0178713113081 Epoch 8 validation cost = 0.0179671934104 Epoch 9 validation cost = 0.0144859958564 Epoch 10 validation cost = 0.0178673797724

Retrieve all parameters from the network

cost = T.mean((predicted_values - target_values)**2)

Objectives, updates, and training

n_batch, n_time_steps, n_features = 1_in.input_var.shape # Now, squash the n_batch and n_time_steps dimensions

l_reshape = lasagne.layers.ReshapeLayer(l_sum, (-1, N_HIDDEN))

We want the network to predict a single value, the sum, so we'll use a single unit.

We need to specify a separate input for masks

Here, we supply the gate parameters for each gate ingate=gate_parameters, forgetgate=gate_parameters,

can decide which you want by setting the peepholes argument to True or False.

A common method for mitigating the exponentially growing gradients commonly found when "unrolling" recurrent networks through time and backpropagating is to simply preventing them from being larger than a pre-set value. In recurrent layers, this can be achieved by passing in a float (rather than False to grad clipping.

theano's underlying method for recursion, scan, isn't terribly efficient. Some speedup can be achieved (with some additional memory usage) by explicitly using a for loop in Python instead of scan. lasagne has a utility function for this, utils.unroll_scan, which can be swapped in for theano.scan by setting unroll_scan=True. Unfortunately, this currently requires that the n_time_steps dimension be known beforehand, so we will use the default of unroll_scan=False.

Precomputing input dot products

Unrolling recursion

Some of the dot products computed in recurrent layers are non-recursive, which means they can be computed ahead of time in one big dot product. Since one big dot product is more efficient than lots of little dot products, lasagne does it by default. However, it imposes an additional memory requirement, so if you're running out of memory, set precompute_input to False.

```
cell=cell_parameters, outgate=gate_parameters,
# We'll learn the initialization and use gradient clipping
                   learn_init=True, grad_clipping=100.)
Bidirectional recurrence
It's often helpful to process sequences simultaneously forwards and backwards. In lasagne this is achieved by using two recurrent layers in parallel, one of which has the
backwards argument set to True. When backwards=True, the sequence is processed in reverse, then the layer's output is reversed so that recurrent layer outputs always
go from the start of the sequence to the finish. The output of the forward and backward layers should then be combined using a MergeLayer of some kind - e.g.
concatenating their output or summing it.
```

l_in, N_HIDDEN, ingate=gate_parameters, mask_input=l_mask, forgetgate=gate_parameters,

cell=cell_parameters, outgate=gate_parameters learn_init=True, grad_clipping=100., backwards=True) # We'll combine the forward and backward layer output by summing. # Merge layers take in lists of layers to merge as input. l_sum = lasagne.layers.ElemwiseSumLayer([l_lstm, l_lstm_back])

As mentioned above, recurrent layers and feed-forward layers expect different input shapes. The output of 1_merge will be of shape (n_batch, n_time_steps, 2). If we fed this into a non-recurrent layer, it would think that the n_time_steps dimension was a "feature" dimension, when in fact it's a "sample" dimension. That is, each index in the second dimension should be treated as a different sample, and a non-recurrent lasagne layer would instead treat them as different feature values, which would be incorrect. Fortunately, the ReshapeLayer makes combining these conventions very convenient - we just combine the first and second dimension so that there are essentially n_batch*n_time_steps individual samples before using any non-recurrent layers, then (optionally) reshape the output back to the original shape. Now, as mentioned above, we are using None in the n batch and n time steps dimensions to allow for minibatches with an arbitrary number of sequences with an arbitrary number of timesteps. We can't tell the ReshapeLayer exactly what to reshape to, but we can instead just tell it to squash all of the dimensions up to the last by using -1 for the first dimension. In order to reshape back to the original shape after using a non-recurrent layer, we can just retrieve the input layer's symbolic shape and use those symbolic variables in another ReshapeLayer, so that the correct values will be filled in at compile time.

Note that because we will only be using the output of the network at the end of the sequence, this could also be done using a SliceLayer (as in the recurrent py example

l reshape, num units=1, nonlinearity=lasagne.nonlinearities.tanh) # Now, the shape will be n_batch*n_timesteps, 1. We can then reshape to # n_batch, n_timesteps to get a single value for each timstep from each sequence l_out = lasagne.layers.ReshapeLayer(l_dense, (n_batch, n_time_steps))

```
The rest of the code for training the recurrent network is guite similar to the feed-forward network with two key differences; First, as mentioned above, a separate theano
tensor variable mask is required to tell the network the length of each sequence in each minibatch, and second, the effectiveness of the network only really depends on the
final time step output by the network, so we will simply slice away that value when computing a loss function.
 In [27]: # Symbolic variable for the target network output.
            # It will be of shape n_batch, because there's only 1 target value per sequence.
            target_values = T.vector('target_output')
            # This matrix will tell the network the length of each sequences.
            # The actual values will be supplied by the gen_data function.
            mask = T.matrix('mask')
            # lasagne.layers.get_output produces an expression for the output of the net
            network_output = lasagne.layers.get_output(l_out)
            # The value we care about is the final value produced for each sequence
```

```
all_params = lasagne.layers.get_all_params(l_out)
# Compute adam updates for training
updates = lasagne.updates.adam(cost, all_params)
# Theano functions for training and computing cost
train = theano.function(
   [l_in.input_var, target_values, l_mask.input_var],
    cost, updates=updates)
compute_cost = theano.function(
    [l in.input var, target values, l mask.input var], cost)
# We'll use this "validation set" to periodically check progress
X_val, y_val, mask_val = gen_data()
\# We'll train the network with 10 epochs of 100 minibatches each
NUM EPOCHS = 10
EPOCH SIZE = 100
for epoch in range(NUM EPOCHS):
    for _ in range(EPOCH_SIZE):
        X, y, m = gen_data()
        train(X, y, m)
    cost_val = compute_cost(X_val, y_val, mask_val)
    print("Epoch {} validation cost = {}".format(epoch + 1, cost_val))
/usr/local/lib/python2.7/site-packages/Theano-0.7.0-py2.7.egg/theano/scan_module/scan.py:1019: Warning: In the strict m
ode, all neccessary shared variables must be passed as a part of non_sequences
  'must be passed as a part of non_sequences', Warning)
/usr/local/lib/python2.7/site-packages/Theano-0.7.0-py2.7.egg/theano/scan_module/scan_perform_ext.py:135: RuntimeWarnin
g: numpy.ndarray size changed, may indicate binary incompatibility
 from scan_perform.scan_perform import *
```