DS-GA 1008
Deep Learning, Spring 2019
Homework Assignment 3
Yves Greatti - yg390
Read-Only Link to tex file

1. Fundamentals

1.1. Dropout

(a) List the torch.nn module corresponding to 2D dropout. The 2D dropout technique is implemented by torch.nn.Dropout2d(p=0.5, inplace=False)

(b) Read on what dropout is and give a short explanation on what it does and why it is useful. Deep neural networks, with non-linear "hidden" layers, will model almost perfectly complex relationships between the input and the correct output and there will be many different settings of the weight vectors. Each of these instance of trained neural network, will do worst on the test data than on the training data, and in essence they will overfit. With unlimited resources, the best way to "regularize" a network is to average the settings of all these weight vectors. Dropout is a cheap albeit efficient method of regularizing. Dropout provides an approximation to model combination in evaluating a bagged ensemble of exponentially many neural networks. It helps the network to not give too much importance to a particular feature. It helps to learn robust features which are more useful with many different random subsets. It also helps to reduce interdependence amongst the neurons, and limits the network ability to memorize very specific conditions during training.
Dropout could apply to input and hidden units ,which mean they are temporarily removed from the network. In the simplest case, each unit is retained with a fixed probability p independent of other units, p is an hyper-parameter which can be chosen using a validation set or is fixed. Typically, 0.5 for a hidden unit seems the optimal value for a wide range of networks and tasks and for the input units, the value is closer to 1 like 0.8.
In practice, for each mini-batch a random binary mask is applied to all the input and hidden units of a layer, the mask is generated independently for each dropout layer. If a unit in a layer, is retained with a probability p during training, the outgoing weights of that unit are multiplied by p at test time: the output at test time is same as expected output at training time.
PyTorch torch.nn.Dropout implementation to keep the inference as fast pos-

sible scales the units using the reciprocal of the keep probability $\frac{1}{1-p}$ during training which yields the same result. 2D dropout in PyTorch performs the same function as the previous one, however it drops the entire 2D feature map instead of individual unit. In early convolution layers adjacent pixels within each feature maps are strongly correlated, the regular dropout will not regularize the activations and, instead spatial dropout 2D helps in promoting independence between feature maps and should be preferred instead. Because dropout is a regularization technique, it reduces the capacity of the network, which leads to an increase of its size. Also dropout makes the learning process harder and increases the number of iterations (epochs) during training. In very large datasets, regularization does not have a direct impact on the generalization error, in these cases, dropout could be less relevant. On very small training examples, dropout is less effective compared to **bayesian networks**. Dropout has inspired other approaches like **fast dropout**, or **drop out boosting** but none of them have outperformed its performances. Today Batch Norm has made dropout less relevant.

1.2. Batch Norm

(a) What does mini-batch refer to in the context of deep learning?
Mini-batch in the context of deep learning is related to the batch size of the data used by the gradient descent algorithm that splits the training data into small batches that are used to compute model error and update its parameters. It seeks to combine the effects of batch gradient descent and stochastic gradient descent. In batch gradient descent, the gradient is computed over the entire dataset. In stochastic gradient descent, the gradient is computed on a single instance of the dataset. On expectation, the gradient on a random sample will point to the same direction of the full dataset samples. SGD is more efficient and its stochastic property can allow the model to avoid local minima. SGD actually, helps generalization by finding "flat" minima on the training set, which are more likely to also be minima on the test set (see Theory of Deep Learning III: Generalization Properties of SGD). Mini-batch will have a size ranging from 2 to 32 (see Revisiting Small Batch Training for Deep Neural Networks). You seek to look for a size of a mini-batch which saturates the hardware. Mini-batch, compared to SGD, can still be parallelized.

(b) Read on what batch norm is and give a short explanation on what it does and why it is useful. Batch normalization reduces the *"Internal Covariate Shift"* which is defined in Batch Normalization: Accelerating Deep Net-

work Training by Reducing Internal Covariate Shift, as the change in the distribution of network activations due to the change in network parameters during training: inputs to each layer are affected by the parameters of all preceding layers. Before batch normalization, saturated regime of nonlinear activation resulting in vanishing gradient, were usually addressed by using ReLU, small learning rates or careful initialization of the weights. In multi-dimension, the objective function is locally quadratic and the surfaces of equal cost are ellipsoids. The optimal learning rate is the inverse of the second derivative in the direction which it is the largest. The second derivative is contained in the Hessian matrix. If the Hessian is diagonalizable, the optimal learning rate, is then, the inverse of the largest eigenvalue. When the contours of equi-energy are all equi-distant, the convergence to a local minima is the fastest, as the gradient at each iteration, is orthogonal to the directions of the eigenvectors. Normalizing all the input variables with mean zero and variance one makes the Hessian diagonalizable. This is what batch normalization achieves by making all the variables in the network of variance one and zero mean. Batch normalization helps to avoid these issues by subtracting the mean and dividing by the batch standard distribution, normalizing the scalar feature to have a Gaussian distribution $\mathcal{N}(0, 1)$. In implementation, this technique usually amounts to insert the BatchNorm layer immediately after the fully connected layer or convolutional layer, and before non-linearities. Batch normalization, by preventing the model from getting stuck in the saturated regime of nonlinearities, enables higher learning rates and allows to achieve faster training. By whitening the inputs of each layer, BatchNorm regularizes the model, removing or reducing the need for dropout. After the shift and scaling, two learnable parameters, γ and β, are used to avoid the network to undo the normalization and recover the original activations of the network. The output of a BatchNorm layer is given by:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

In PyTorch, by default, the elements of γ are sampled from a uniform distribution $\mathcal{U}(0, 1)$ and the elements of β are set to 0. The mean and standard deviation are computed over the mini-batches and each layer can keep running estimates using a momentum. The running averages for mean and variance are updated using an exponential decay based on the momentum parameter:

- running_mean = momentum * running_mean + (1 - momentum) * sample_mean (sample_mean is the new observed mean)

· running_var = momentum * running_var + (1 - momentum) * sample_var (sample_var is the new observed variance)

momentum=0 means that old information is discarded completely at every time step, while momentum=1 means that new information is never incorporated. For fully connected activation layers, BN is applied separately to each dimension (H, W) with a pair of learned parameters γ and β per dimension. For convolutional layers, BN is applied so that different elements of the same feature map, at different spatial locations, are normalized across the mini-batch. For a mini-batch of size m and features maps of size $p \times q$, means and standard deviations are computed over $m \times p \times q$. The parameters γ and β are learned per feature map. At inference time, BN uses the statistic estimates (mean and variance) computed at training time. To have good estimates BN requires very large batches. Thus BN is less effective for small datasets. BN is a differentiable transformation and using the chain rules the gradient of loss can be explicitly formalized and it can be shown that back-propagation for BN is unaffected by the scale of its parameters and BN will stabilize the parameter growth. Overall BN is great, it makes the loss surface smoother enabling higher learning rates. However it needs to go over a large training data set to have a good estimate of the statistics. There are other normalization methods like layer, instance or group normalization which uses statistics computed from input data in both training and evaluation modes and might be more appropriate depending the characteristics of the data set. Recent papers have explored different normalization methods at different depths of the network.

2. Language Modeling

This exercise explores the code from the word_language_model example in PyTorch.

(a) Go through the code and draw a block diagram / flow chart (see this tutorial) which highlights the main interacting components, illustrates their functionality, and provides an estimate of their computational time percentage (rough profiling).

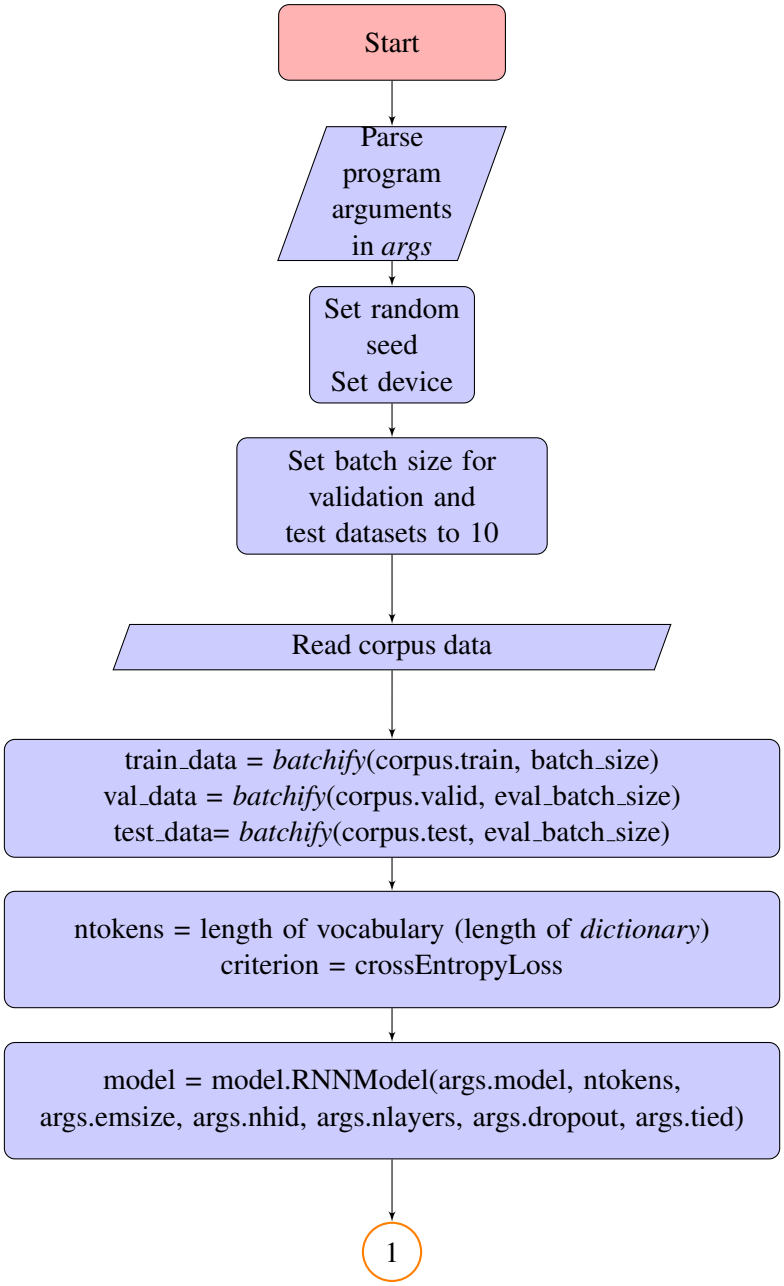
Main Component

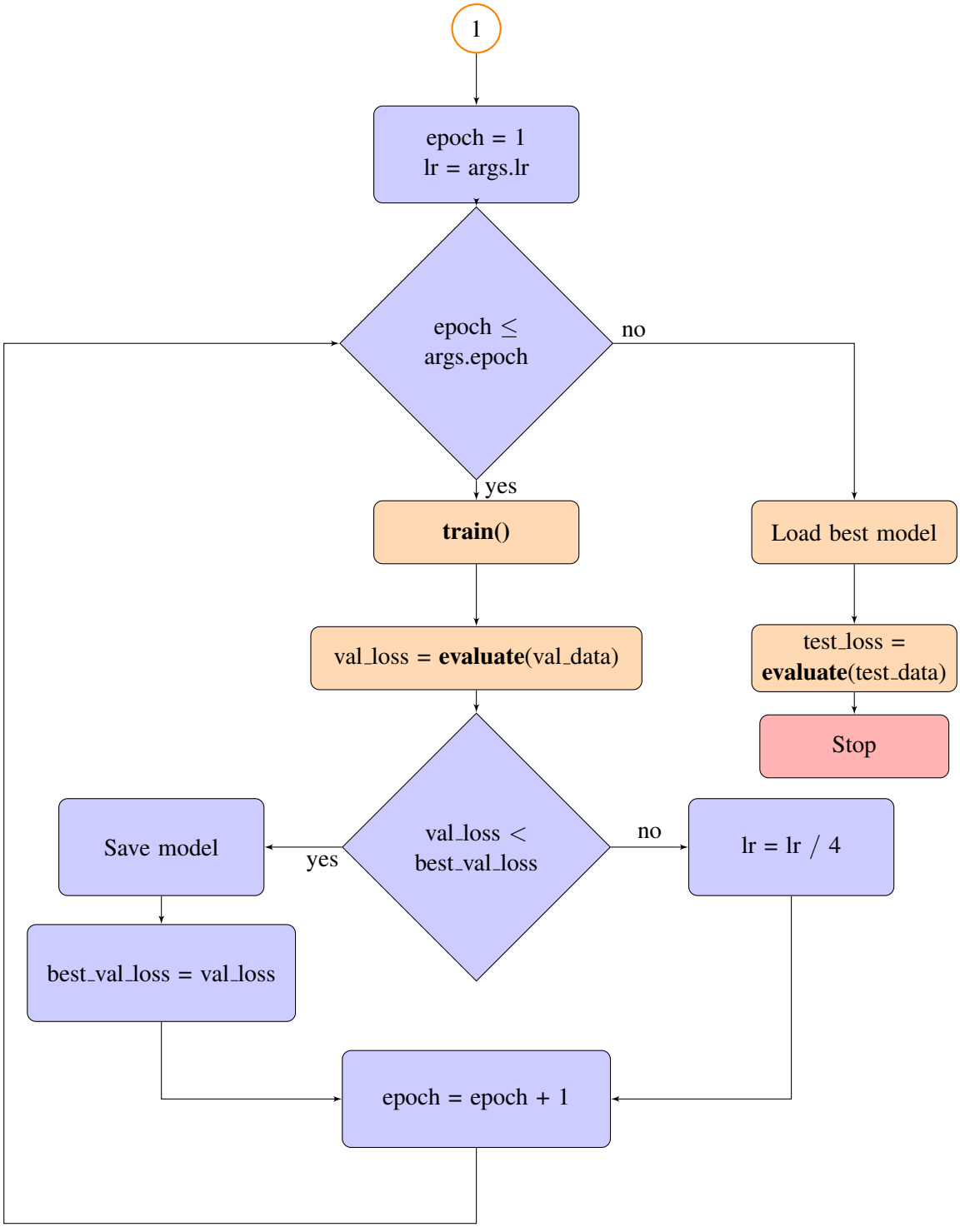
We start to describe the major functions used in the main component flow chart.

- **Read corpus data** creates three dictionaries: train, validation and test from three separate related files (train.txt, valid.txt, test.txt). Each line of a text file is split in words, and each word is added to a dictionary. A dictionary is made of two maps: (1) word to index: word2idx and (2) index to word: idx2word.
- **batchify()** given a tensor of size M, the function creates N batches of size batch_size ($N = \lfloor M/\text{batch\_size} \rceil$), throwing away the data in excess of N. Starting from sequential data, batchify arranges the dataset into columns. For instance, with the alphabet as the sequence and batch size of 4, we will get

$$\begin{bmatrix} a & g & m & s \\ b & h & n & t \\ c & i & o & u \\ d & j & p & v \\ e & k & q & w \\ f & l & r & c \end{bmatrix}$$

  These columns are treated independently without trying to learn the dependency between characters like for e.g. between f and g but allows more efficient batch processing.


Start
Parse program arguments in *args*
Set random seed
Set device
Set batch size for validation and test datasets to 10
Read corpus data
train_data = *batchify*(corpus.train, batch_size)
val_data = *batchify*(corpus.valid, eval_batch_size)
test_data= *batchify*(corpus.test, eval_batch_size)
ntokens = length of vocabulary (length of *dictionary*)
criterion = crossEntropyLoss
model = model.RNNModel(args.model, ntokens, args.emsize, args.nhid, args.nlayers, args.dropout, args.tied)
1


1
epoch = 1
lr = args.lr
epoch ≤
args.epoch
no
yes
train()
Load best model
val_loss = evaluate(val_data)
test_loss =
evaluate(test_data)
Stop
val_loss <
best_val_loss
Save model
yes
no
lr = lr / 4
best_val_loss = val_loss
epoch = epoch + 1

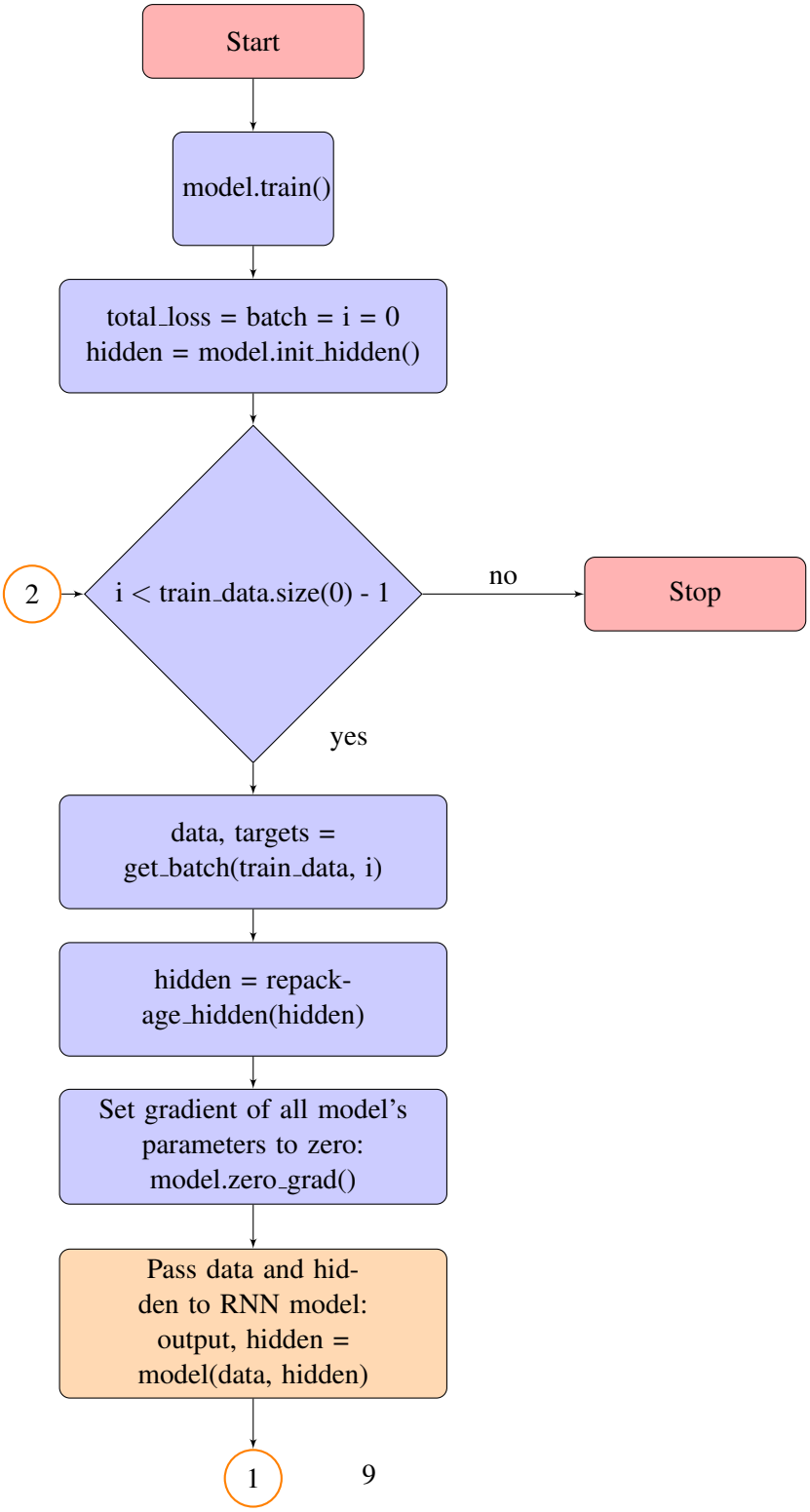
Main.train() Component

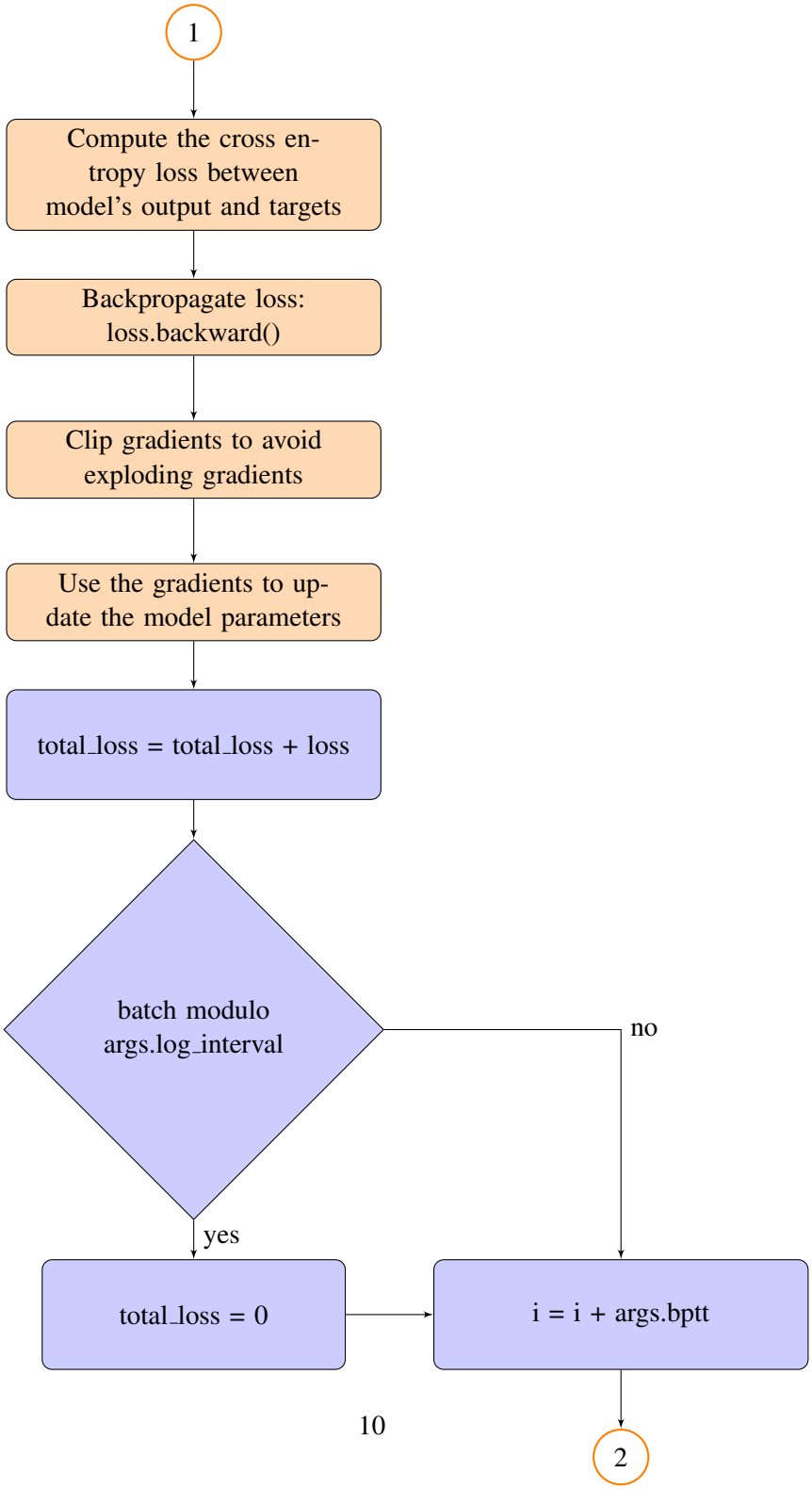
The main functions of the train function are:

- **model.init_hidden()** uses the next set of model parameters in the sequence of all the model parameters since the last call to the function model.init_hidden() to set the recurrent network hidden weights to zero. The main interest of this method is that it follows best practices to assign the initial hidden state of the recurrent neural network to a device (CPU or GPU) if the model and its parameters are on cpu, the same gpu if the model has been transferred with model.cuda() (see device-agnostic code). It is called before the training and evaluation epoch loop where the model is updated via gradient descent.
- **get_batch()** will generate chunks of length args.bptt, which corresponds to the length of the sequence being passed to the RNN model (sequence length) With a *bptt* of 2 and reusing the previous example of the output of the batchify function, for i = 0, the function generates:

$$\text{data} = \begin{pmatrix} a & g & m & s \\ b & h & n & t \end{pmatrix} \text{target} = \begin{pmatrix} b & h & n & t \\ c & i & o & u \end{pmatrix}$$

- **repackage_hidden(hidden)** At each epoch, one iteration of the main loop in the train() function, the RNN model is trained on a sequence of new characters produced by the get_batch() function and the gradients are back-propagated (BPTT) through the RNNs computational graph. From one iteration of the loop to the next iteration, if the hidden states still have a reference to the graph, from the previous iteration, the gradients will be back-propagated through them. This is not the intended behavior as we want the RNN's gradients to be propagated independently for each batch of words (sequence). To get rid of the references of the hidden states, repackage_hidden(hidden) function goes through each hidden state and detaches it from the graph that created it, making it a leaf. This allows the previous graph (the graph which was created in the previous iteration) to go out of scope and free up the memory for the next iteration. The hidden states at epoch (t) are preserved for the next iteration: epoch (t+1).
- **Clipping the gradients:** is performed using torch.nn.utils.clip_grad_norm, they are clipped using their L2 norms. During the back-propagation, gradients are not clipped until the backward pass is completed and before the model parameters are updated.

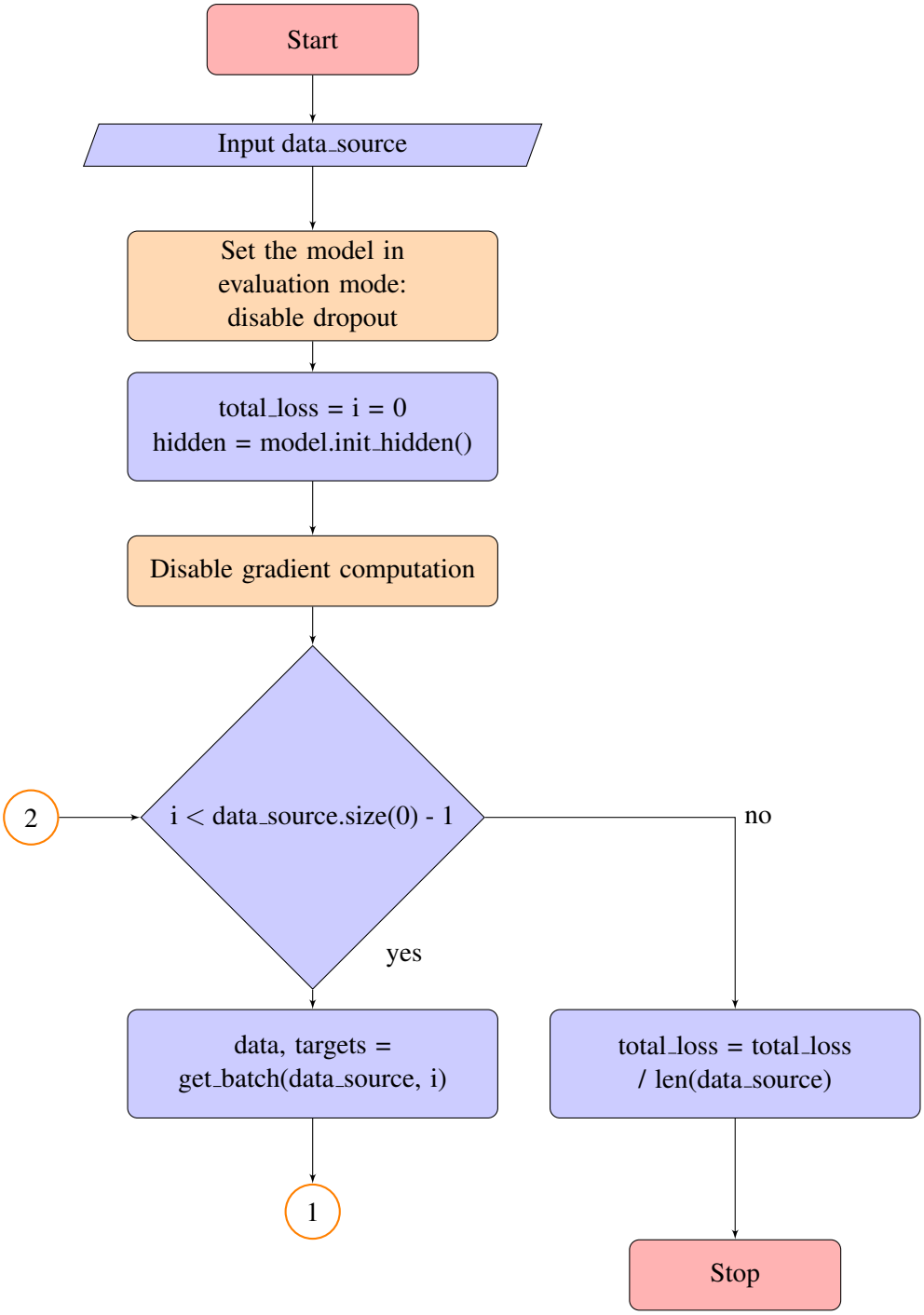

Start
model.train()
total_loss = batch = i = 0
hidden = model.init_hidden()
2
i < train_data.size(0) - 1
no
Stop
yes
data, targets =
get_batch(train_data, i)
hidden = repack-
age_hidden(hidden)
Set gradient of all model's
parameters to zero:
model.zero_grad()
Pass data and hid-
den to RNN model:
output, hidden =
model(data, hidden)
1

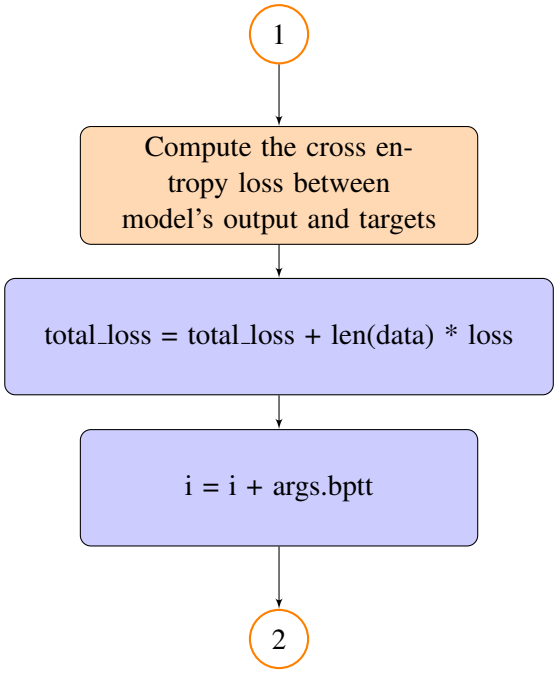

1
Compute the cross entropy loss between model's output and targets
Backpropagate loss: loss.backward()
Clip gradients to avoid exploding gradients
Use the gradients to update the model parameters
total_loss = total_loss + loss
batch modulo args.log_interval
no
yes
total_loss = 0
i = i + args.bptt
2

Main.evaluate() Component

Every important function in the evaluate() function has already been covered. The major differences with the train function are:

1. No gradients are calculated, no back-progation is performed, no gradient clipping, no update of the model parameters with the gradients. Although the repackage_hidden is invoked, it should not be necessary to do so.
2. The function accepts the parameter data source so it can be invoked for the validation and test datasets.
3. At each iteration the loss is multiplied by the sequence length and added to the total loss. At exit, the evaluate function returns the total_loss divided by the total size of the data source returning in effect the updated total loss.


Start
Input data_source
Set the model in evaluation mode: disable dropout
total_loss = i = 0
hidden = model.init_hidden()
Disable gradient computation
2
i < data_source.size(0) - 1
no
yes
data, targets = get_batch(data_source, i)
total_loss = total_loss / len(data_source)
1
Stop



Profiling

batchify

Most of the time as indicated in column %Time is spent in the reshaping of the data (Fig. **??**).

```
Total time: 0.011815 s
File: main.py
Function: batchify at line 79

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    79                                           @profile
    80                                           def batchify(data, bsz):
    81                                               # Work out how cleanly we can divide the dataset into bsz parts.
    82         3         28.0      9.3      0.2      nbatch = data.size(0) // bsz
    83                                               # Trim off any extra elements that wouldn't cleanly fit (remainders).
    84         3         45.0     15.0      0.4      data = data.narrow(0, 0, nbatch * bsz)
    85                                               # Evenly divide the data across the bsz batches.
    86         3      11686.0   3895.3     98.9      data = data.view(bsz, -1).t().contiguous()
    87         3         56.0     18.7      0.5      return data.to(device)
```

Figure 1: batchify.

repackage_hidden

This is a recursive function: the time is equally spent across the function (Fig. **??**).

```
Total time: 0.054145 s
File: main.py
Function: repackage_hidden at line 107

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   107                                           @profile
   108                                           def repackage_hidden(h):
   109                                               """Wraps hidden states in new Tensors, to detach them from their history."""
   110     12924      14292.0      1.1     26.4      if isinstance(h, torch.Tensor):
   111      8616      19565.0      2.3     36.1          return h.detach()
   112                                               else:
   113      4308      20288.0      4.7     37.5          return tuple(repackage_hidden(v) for v in h)
```

Figure 2: repackage_hidden.

get_batch

The computational time is uniformly distributed across the function (Fig. **??**).

```
Total time: 0.112994 s
File: main.py
Function: get_batch at line 126

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   126                                           @profile
   127                                           def get_batch(source, i):
   128      4308      32773.0      7.6     29.0      seq_len = min(args.bptt, len(source) - 1 - i)
   129      4308      32626.0      7.6     28.9      data = source[i:i+seq_len]
   130      4308      45179.0     10.5     40.0      target = source[i+1:i+1+seq_len].view(-1)
   131      4308       2416.0      0.6      2.1      return data, target
```

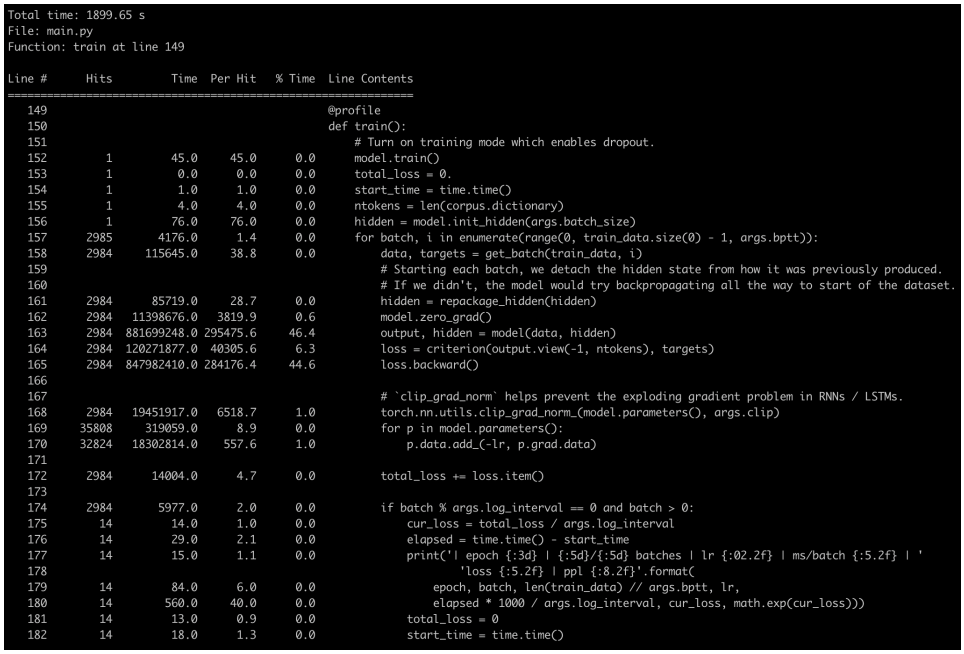
Figure 3: get_batch.

train

The time is split between the forward pass executed by the model (46%) and the back-propagation triggered at line 165: loss.backward() (44%) (Fig. **??**).

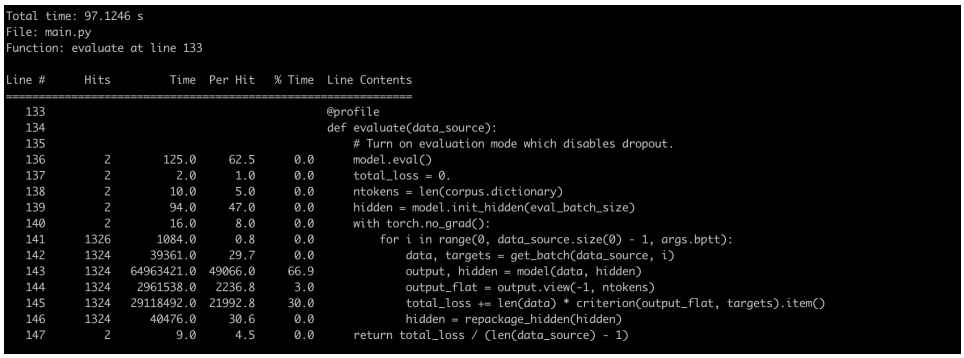
evaluate

66% of the time is spent inside the model processing the data and 30% is taken by the computation of the loss (Fig. **??**).



```
Total time: 1899.65 s
File: main.py
Function: train at line 149

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def train():
   151                                               # Turn on training mode which enables dropout.
   152         1         45.0     45.0      0.0      model.train()
   153         1          0.0      0.0      0.0      total_loss = 0.
   154         1          1.0      1.0      0.0      start_time = time.time()
   155         1          4.0      4.0      0.0      ntokens = len(corpus.dictionary)
   156         1         76.0     76.0      0.0      hidden = model.init_hidden(args.batch_size)
   157      2985       4176.0      1.4      0.0      for batch, i in enumerate(range(0, train_data.size(0) - 1, args.bptt)):
   158      2984     115645.0     38.8      0.0          data, targets = get_batch(train_data, i)
   159                                                   # Starting each batch, we detach the hidden state from how it was previously produced.
   160                                                   # If we didn't, the model would try backpropagating all the way to start of the dataset.
   161      2984      85719.0     28.7      0.0          hidden = repackage_hidden(hidden)
   162      2984   11398676.0   3819.9      0.6          model.zero_grad()
   163      2984  881699248.0 295475.6     46.4          output, hidden = model(data, hidden)
   164      2984  120271877.0  40305.6      6.3          loss = criterion(output.view(-1, ntokens), targets)
   165      2984  847982410.0 284176.4     44.6          loss.backward()
   166
   167                                                   # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
   168      2984   19451917.0   6518.7      1.0          torch.nn.utils.clip_grad_norm_(model.parameters(), args.clip)
   169     35808     319059.0      8.9      0.0          for p in model.parameters():
   170     32824   18302814.0    557.6      1.0              p.data.add_(-lr, p.grad.data)
   171
   172      2984      14004.0      4.7      0.0          total_loss += loss.item()
   173
   174      2984       5977.0      2.0      0.0          if batch % args.log_interval == 0 and batch > 0:
   175        14         14.0      1.0      0.0              cur_loss = total_loss / args.log_interval
   176        14         29.0      2.1      0.0              elapsed = time.time() - start_time
   177        14         15.0      1.1      0.0              print('| epoch {:3d} | {:5d}/{:5d} batches | lr {:02.2f} | ms/batch {:5.2f} | '
   178                                                             'loss {:5.2f} | ppl {:8.2f}'.format(
   179        14         84.0      6.0      0.0                      epoch, batch, len(train_data) // args.bptt, lr,
   180        14        560.0     40.0      0.0                      elapsed * 1000 / args.log_interval, cur_loss, math.exp(cur_loss)))
   181        14         13.0      0.9      0.0              total_loss = 0
   182        14         18.0      1.3      0.0              start_time = time.time()
```

Figure 4: train.



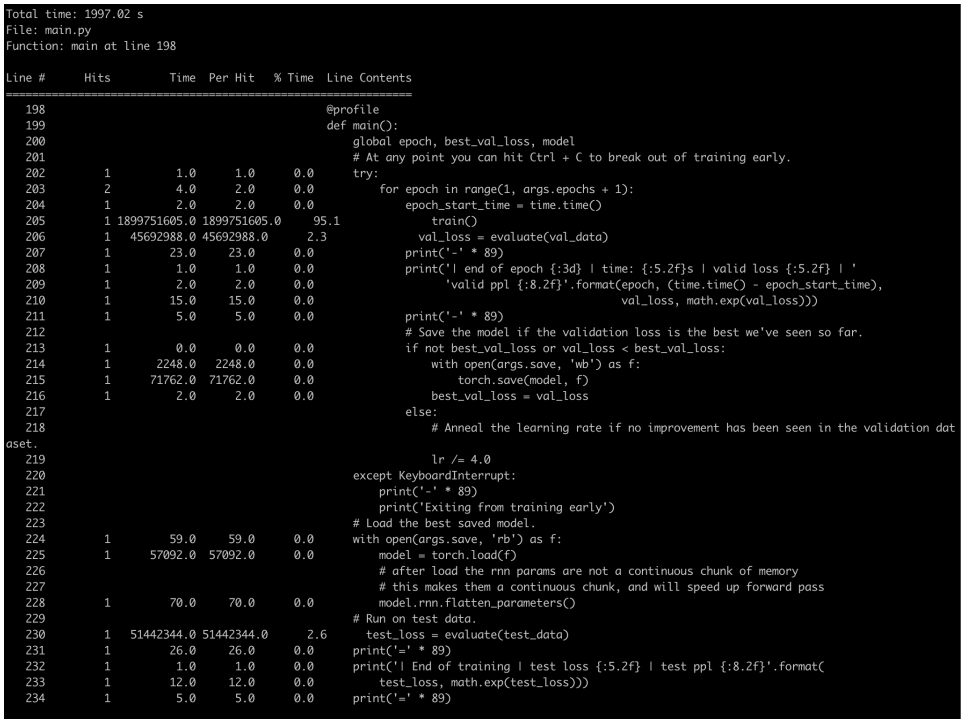
```
Total time: 97.1246 s
File: main.py
Function: evaluate at line 133

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   133                                           @profile
   134                                           def evaluate(data_source):
   135                                               # Turn on evaluation mode which disables dropout.
   136         2        125.0     62.5      0.0      model.eval()
   137         2          2.0      1.0      0.0      total_loss = 0.
   138         2         10.0      5.0      0.0      ntokens = len(corpus.dictionary)
   139         2         94.0     47.0      0.0      hidden = model.init_hidden(eval_batch_size)
   140         2         16.0      8.0      0.0      with torch.no_grad():
   141      1326       1084.0      0.8      0.0          for i in range(0, data_source.size(0) - 1, args.bptt):
   142      1324      39361.0     29.7      0.0              data, targets = get_batch(data_source, i)
   143      1324   64963421.0  49066.0     66.9              output, hidden = model(data, hidden)
   144      1324    2961538.0   2236.8      3.0              output_flat = output.view(-1, ntokens)
   145      1324   29118492.0  21992.8     30.0              total_loss += len(data) * criterion(output_flat, targets).item()
   146      1324      40476.0     30.6      0.0              hidden = repackage_hidden(hidden)
   147         2          9.0      4.5      0.0      return total_loss / (len(data_source) - 1)
```

Figure 5: evaluate.

main flow

Most of the time, as you will expect, is spent within the train() function (95%) and about 5% (2.3% and 2.6%) are consumed by the evaluation on the validation and test datasets (Fig. **??**).

(b) Find and explain where and how the back-propagation through time (BPTT)



```
Total time: 1997.02 s
File: main.py
Function: main at line 198

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   198                                           @profile
   199                                           def main():
   200                                               global epoch, best_val_loss, model
   201                                               # At any point you can hit Ctrl + C to break out of training early.
   202         1          1.0      1.0      0.0      try:
   203         2          4.0      2.0      0.0          for epoch in range(1, args.epochs + 1):
   204         1          2.0      2.0      0.0              epoch_start_time = time.time()
   205         1 1899751605.0 1899751605.0     95.1              train()
   206         1   45692988.0 45692988.0      2.3              val_loss = evaluate(val_data)
   207         1         23.0     23.0      0.0              print('-' * 89)
   208         1          1.0      1.0      0.0              print('| end of epoch {:3d} | time: {:5.2f}s | valid loss {:5.2f} | '
   209         1          2.0      2.0      0.0                    'valid ppl {:8.2f}'.format(epoch, (time.time() - epoch_start_time),
   210         1         15.0     15.0      0.0                                               val_loss, math.exp(val_loss)))
   211         1          5.0      5.0      0.0              print('-' * 89)
   212                                                       # Save the model if the validation loss is the best we've seen so far.
   213         1          0.0      0.0      0.0              if not best_val_loss or val_loss < best_val_loss:
   214         1       2248.0   2248.0      0.0                  with open(args.save, 'wb') as f:
   215         1      71762.0  71762.0      0.0                      torch.save(model, f)
   216         1          2.0      2.0      0.0                  best_val_loss = val_loss
   217                                                       else:
   218                                                           # Anneal the learning rate if no improvement has been seen in the validation dat
aset.
   219                                                           lr /= 4.0
   220                                               except KeyboardInterrupt:
   221                                                   print('-' * 89)
   222                                                   print('Exiting from training early')
   223                                               # Load the best saved model.
   224         1         59.0     59.0      0.0      with open(args.save, 'rb') as f:
   225         1      57092.0  57092.0      0.0          model = torch.load(f)
   226                                                   # after load the rnn params are not a continuous chunk of memory
   227                                                   # this makes them a continuous chunk, and will speed up forward pass
   228         1         70.0     70.0      0.0          model.rnn.flatten_parameters()
   229                                               # Run on test data.
   230         1   51442344.0 51442344.0      2.6      test_loss = evaluate(test_data)
   231         1         26.0     26.0      0.0      print('=' * 89)
   232         1          1.0      1.0      0.0      print('| End of training | test loss {:5.2f} | test ppl {:8.2f}'.format(
   233         1         12.0     12.0      0.0          test_loss, math.exp(test_loss)))
   234         1          5.0      5.0      0.0      print('=' * 89)
```

Figure 6: main.

takes place (you may need to delve into PyTorch source code). The back-propagation through time starts in the train function at the line: loss.backward(). The generalized back-propagation algorithm is applied to the unrolled computational graph $\boldsymbol{h}(t) = f(\boldsymbol{h}^{t-1}, \boldsymbol{x}^t; \theta)$ where $\boldsymbol{h}^t$ represents the hidden state a time t and $\boldsymbol{x}^t$ the input $\boldsymbol{x}$ a time t and θ the model parameters. The RNN-Model architecture is:

Layer	**Parameters**
Linear	(args.nhid, ntokens)
Recurrent Network	(args.emsize, args.nhid, args.nlayers, dropout)
Dropout	(dropout)
Embedding	(ntokens, args.emsize)

Recurrent Network is either LSTM, RNN_TANH, RNN_RELU or GRU. All these neural networks in PyTorch extend the base class RNNBase which extends in turn the class Module. The sequence of layers which constitutes the recurrent network is created in the RNNBase code on line 60 (Fig **??**). Every layer is wrapped in a tensor, which is, in turn plugged into a Parameter,

which has by default the parameter *require_grad* set to true. Autograd creates a DAG (Direct Acyclic Graph) when creating these tensors as a graph of Functions which can be **apply**()ed to compute the result of evaluating the graph during the forward pass. During the forward pass, autograd builds up a graph representing the functions that computes the gradient (the *.grad_fn* attribute of each torch.Tensor is an entry point into this graph) (see How autograd encodes the history). When the line 160 of the main.py script is executed: **loss.backward()**, the BPTT happens. This graph is evaluated to compute the gradients. The computation stops on the next batch ("next time step") at line 156: hidden = repackage_hidden(hidden), when the hidden states are detached from the graph and the graph is freed. The hidden states with their states from the previous iteration are then used when a new graph is computed for the next args.bptt chunk of data.

```
self._all_weights = []
for layer in range(num_layers):
    for direction in range(num_directions):
        layer_input_size = input_size if layer == 0 else hidden_size * num_directions

        w_ih = Parameter(torch.Tensor(gate_size, layer_input_size))
        w_hh = Parameter(torch.Tensor(gate_size, hidden_size))
        b_ih = Parameter(torch.Tensor(gate_size))
        b_hh = Parameter(torch.Tensor(gate_size))
        layer_params = (w_ih, w_hh, b_ih, b_hh)

        suffix = '_reverse' if direction == 1 else ''
        param_names = ['weight_ih_l{}{}', 'weight_hh_l{}{}']
        if bias:
            param_names += ['bias_ih_l{}{}', 'bias_hh_l{}{}']
        param_names = [x.format(layer, suffix) for x in param_names]

        for name, param in zip(param_names, layer_params):
            setattr(self, name, param)
        self._all_weights.append(param_names)
```

Figure 7: Main loop of the RNNBase's __init__ function.

(c) Describe why we need the repackage_hidden(h) function, and how it works. At each epoch, one iteration of the main loop in the train() function, the RNN model is trained on a sequence of new characters produced by the get_batch() function and the gradients are back-propagated (BPTT) through the RNNs computational graph. From one iteration of the loop to the next iteration, if the hidden states still have a reference to the graph, from the previous iteration, the gradients will be back-propagated through them. This is not the intended behavior as we want the RNN's gradients to be propagated independently for each batch of words (sequence). To get rid of the references of the hidden states, repackage_hidden(hidden) function goes through each hidden state and detaches it from the graph that created it, making it a leaf.

This allows the previous graph (the graph which was created in the previous iteration) to go out of scope and free up the memory for the next iteration. The hidden states at epoch (t) are preserved for the next iteration: epoch (t+1).

(d) Why is there a –tied (tie the word embedding and softmax weights) option? The model can be separated into two components:

- The **encoder**: takes a one hot vector of an input word, multiplies it by a matrix to give a word embedding.
- The **decoder**: multiplies the word embedding by another matrix resulting to an output embedding vector. This vector is then passed through a cross entropy loss, normalizing its values into a probability distribution.

Input embedding and output embedding have few common properties. The first property they share, is that they are both of the same size (number of hidden units per layer is the same as the size of word embeddings) And when the *tied* option is used, encoder and decoder share the same weights . The second property is that they will show similar behavior. The input embedding words with similar meanings, will be represented by similar vectors (in term of cosine similarity). Given the representation from the RNN, the decoder would like to assign similar probabilities to similar words. Therefore similar words are represented by similar vectors in the output embedding. Experiments have also shown that the word representations in the output embedding are of much higher quality than the ones in the input embedding. In a weight tied model, a single high quality embedding matrix is used in two places in the model. In addition weight tying reduces the number of parameters (increases training speed), and has a regularization effect as the model has less capacity to overfit.

(e) Compare LSTM and GRU performance (validation perplexity and training time) for different values of the following parameters: number of epochs, number of layers, hidden units size, input embedding dimensionality, BPTT temporal interval, and non- linearities (pick just 3 of these parameters and experiment with 2-3 different values for each).

I ran three set of experimentations for the two models LSTM and GRU for a total of 3 epochs for each run and each time recording the training time and validation perplexity:

1. Increasing the number of number of hidden units for the recurrent network by 200: 200, 400, 600.

200 hidden units:

Epoch	1	2	3
LSTM - Training time (s)	137.61	137.48	137.81
GRU - Training time (s)	134.96	135.29	135.28
LSTM - Validation perplexity	254.85	207.8	176.86
GRU - Validation perplexity	274.56	223.76	203.47

	LSTM	**GRU**
Total Training Time (s)	412.9	405.53
Total Training Time (mn)	7	7
Average Training Time (s)	137.63	135.18
Average Training Time (mn)	2	2
Min.Validation Perplexity	176.86	203.47
Max.Validation Perplexity	254.85	274.56
Average Validation Perplexity	213.17	233.93

400 hidden units:

Epoch	1	2	3
LSTM - Training time (s)	210.33	211.63	213.38
GRU - Training time (s)	204.34	205.51	204.79
LSTM - Validation perplexity	242.96	178.86	147.99
GRU - Validation perplexity	266.78	214.04	185.36

	LSTM	**GRU**
Total Training Time (s)	635.34	614.64
Total Training Time (mn)	11	10
Average Training Time (s)	211.78	204.88
Average Training Time (mn)	4	3
Min.Validation Perplexity	147.99	185.36
Max.Validation Perplexity	242.96	266.78
Average Validation Perplexity	189.94	222.06

600 hidden units:

Epoch	1	2	3
LSTM - Training time (s)	137.61	137.48	137.81
GRU - Training time (s)	277.55	278.68	278.9
LSTM - Validation perplexity	254.85	207.8	176.86
GRU - Validation perplexity	253.95	200.07	179.45

	LSTM	**GRU**
Total Training Time (s)	412.9	835.13
Total Training Time (mn)	7	14
Average Training Time (s)	137.63	278.38
Average Training Time (mn)	2	5
Min.Validation Perplexity	176.86	179.45
Max.Validation Perplexity	254.85	253.95
Average Validation Perplexity	213.17	211.16

2. Increasing the sequence length *bptt* from 35 to 55 and 75.

35 bptt:

Epoch	1	2	3
LSTM - Training time (s)	137.61	137.48	137.81
GRU - Training time (s)	134.96	135.29	135.28
LSTM - Validation perplexity	254.85	207.8	176.86
GRU - Validation perplexity	274.56	223.76	203.47

	LSTM	**GRU**
Total Training Time (s)	412.9	405.53
Total Training Time (mn)	7	7
Average Training Time (s)	137.63	135.18
Average Training Time (mn)	2	2
Min.Validation Perplexity	176.86	203.47
Max.Validation Perplexity	254.85	274.56
Average Validation Perplexity	213.17	233.93

55 bptt:

Epoch	1	2	3
LSTM - Training time (s)	129.66	129.92	129.99
GRU - Training time (s)	127	127.54	127.6
LSTM - Validation perplexity	278.91	203.52	186.98
GRU - Validation perplexity	280.53	233.69	185.08

	LSTM	**GRU**
Total Training Time (s)	389.57	382.14
Total Training Time (mn)	6	6
Average Training Time (s)	129.86	127.38
Average Training Time (mn)	2	2
Min.Validation Perplexity	186.98	185.08
Max.Validation Perplexity	278.91	280.53
Average Validation Perplexity	223.14	233.1

75 bptt:

Epoch	1	2	3
LSTM - Training time (s)	128.62	129.15	129.27
GRU - Training time (s)	125.53	126.14	126.18
LSTM - Validation perplexity	328.97	222.96	175.43
GRU - Validation perplexity	325.23	220.94	196.32

	LSTM	**GRU**
Total Training Time (s)	387.04	377.85
Total Training Time (mn)	6	6
Average Training Time (s)	129.01	125.95
Average Training Time (mn)	2	2
Min.Validation Perplexity	175.43	196.32
Max.Validation Perplexity	328.97	325.23
Average Validation Perplexity	242.45	247.5

3. Increasing the number of recurrent networks starting at 2,4, and 6.

2 recurrent networks:

Epoch	1	2	3
LSTM - Training time (s)	137.61	137.48	137.81
GRU - Training time (s)	134.96	135.29	135.28
LSTM - Validation perplexity	254.85	207.8	176.86
GRU - Validation perplexity	274.56	223.76	203.47

	LSTM	**GRU**
Total Training Time (s)	412.9	405.53
Total Training Time (mn)	7	7
Average Training Time (s)	137.63	135.18
Average Training Time (mn)	2	2
Min.Validation Perplexity	176.86	203.47
Max.Validation Perplexity	254.85	274.56
Average Validation Perplexity	213.17	233.93

4 recurrent networks:

Epoch	1	2	3
LSTM - Training time (s)	207.91	207.91	208.35
GRU - Training time (s)	200.64	200.73	135.28
LSTM - Validation perplexity	467.44	272	200.63
GRU - Validation perplexity	188785902.8	61978.5	429628533.5

	LSTM	**GRU**
Total Training Time (s)	624.17	536.65
Total Training Time (mn)	10	9
Average Training Time (s)	208.06	178.88
Average Training Time (mn)	3	3
Min.Validation Perplexity	200.63	61978.5
Max.Validation Perplexity	467.44	429628533.5
Average Validation Perplexity	313.3566667	206158804.9

6 recurrent networks:

Epoch	1	2	3
LSTM - Training time (s)	281.01	281.8	281.9
GRU - Training time (s)	267.79	267.59	267.83
LSTM - Validation perplexity	1066.02	1059.07	1040.2
GRU - Validation perplexity	31479922593	30779997.28	44318884.84

	LSTM	**GRU**
Total Training Time (s)	844.71	803.21
Total Training Time (mn)	14	13
Average Training Time (s)	281.57	267.74
Average Training Time (mn)	5	4
Min.Validation Perplexity	1040.2	30779997.28
Max.Validation Perplexity	1066.02	31479922593
Average Validation Perplexity	1055.096667	10518340492

Based on these experiments, we can observe that LSTM and GRU have similar training times but the validation perplexity of the LSTM based model is better than the GRU based model. We also notice that as the number of recurrent GRU networks increases, instabilities appear with a validation perplexity going to the roof. At the same time, compared to the GRU model, for the same values of hyper-parameters, the LSTM model is quite stable.

(f) Why do we compute performance on a test set as well? What is this number good for? The performances on the test set is an estimation of the performances of the model on unseen data. It gives a measurement of the true prediction capability of the model.