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) The 2D dropout technique is implemented by torch.nn.Dropout2d(p=0.5, inplace=False)

(b) 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 worse 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.
Dropping out 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 hyperparameter 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 minibatch 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 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 possible 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 and the number of iterations (epochs) during training. However training time for each epoch is less. 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.

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). Minibatch will have a size ranging from 2 to 32 (see Revisiting Small Batch Training for Deep Neural Networks). Minibatch, compared to SGD, can still be parallelized.

(b) Batch normalization reduces the *"Internal Covariate Shift"* which is defined in Batch Normalization: Accelerating Deep Network 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 non-linear activation resulting in vanishing gradient, were usually addressed by using Relu, small learning rates or care-

ful initialization of the weights. 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. The parameters γ and β are learned per feature map. BN is a differentiable transformation and using the chain rules the gradient of loss can be explicitly formalized and it can be shown that backpropagation for BN is unaffected by the scale of its parameters and BN will stabilize the parameter growth.

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).

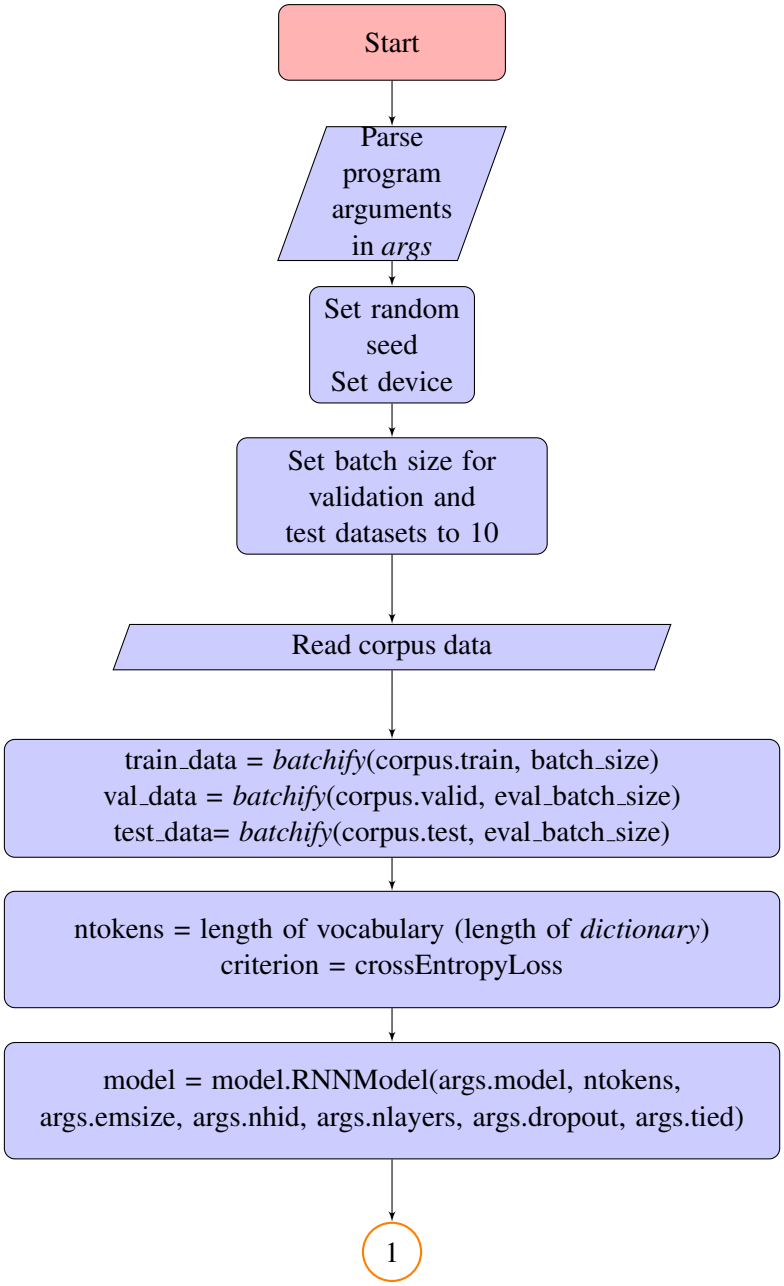
Main Component

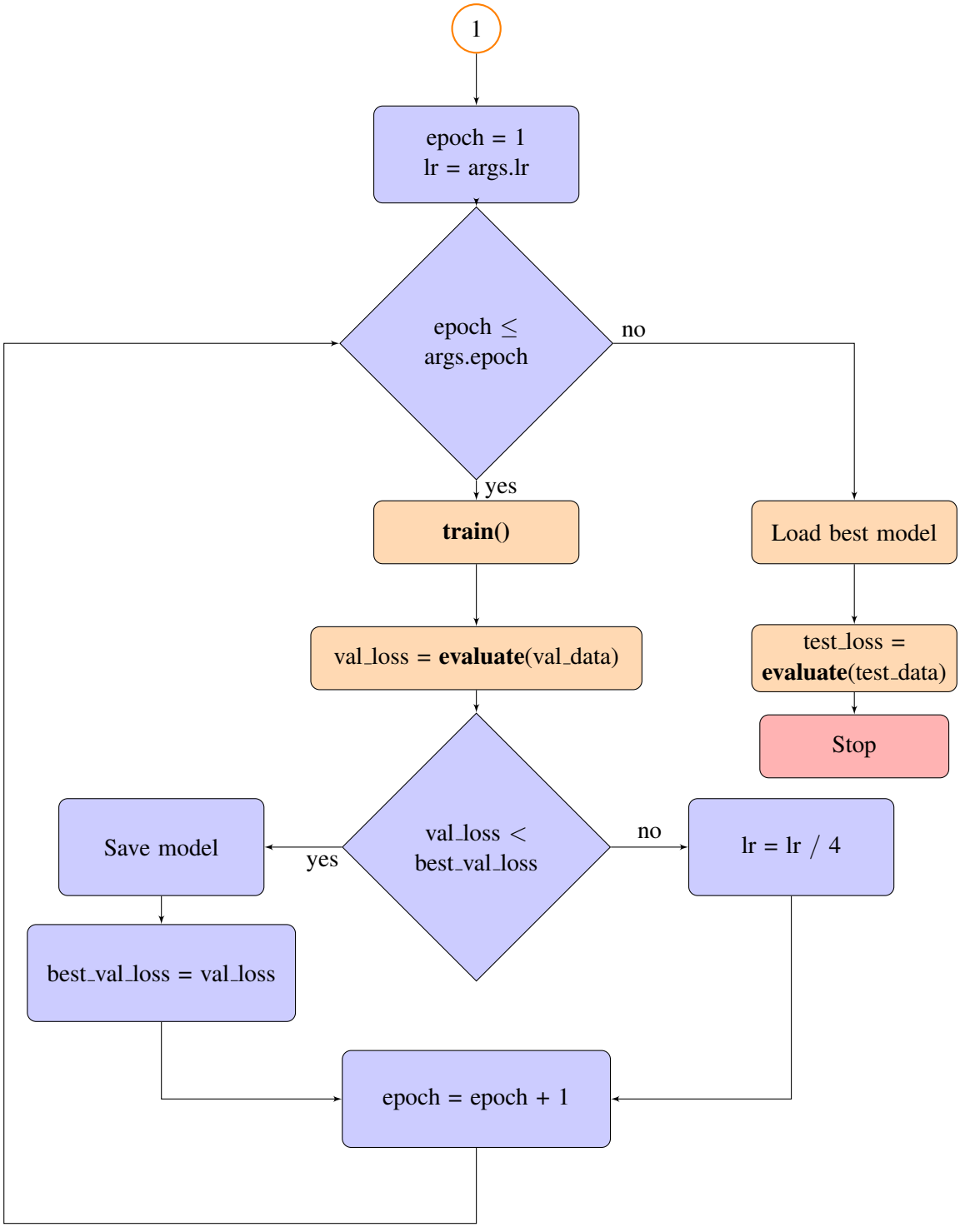
We start to describe the major functions used in the main component flow chart.

- **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

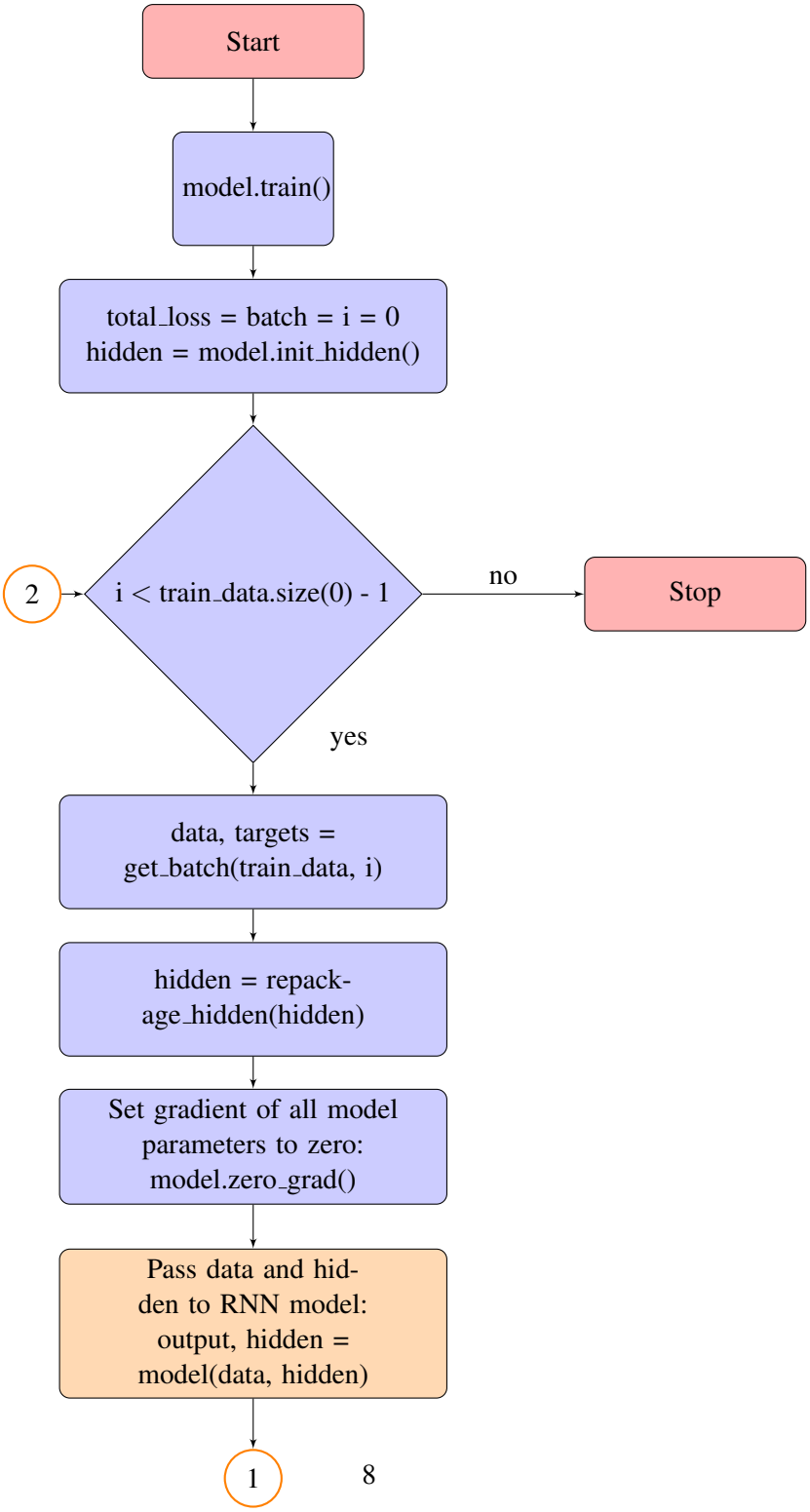
Main.train() Component

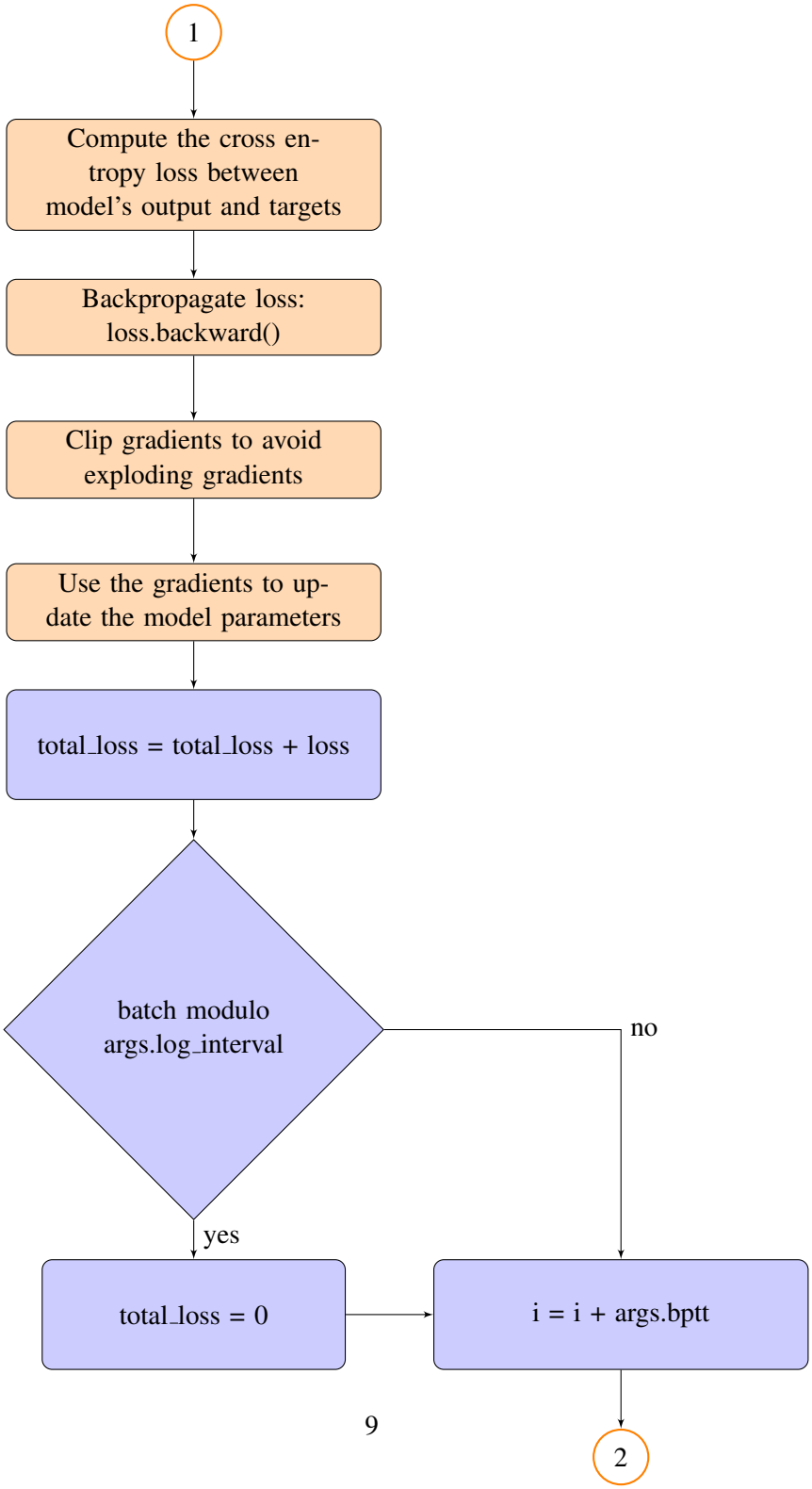
The main functions of the train function are:

- **model.init_hidden()** assigns 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).
- **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) Wit a bptt of 2 and reusing the previous example of the output of the batchify function, the function generates:

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

- **repackage_hidden(hidden)** At each 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 backpropagated (BPTT) through the RNNs computational graph. From one iteration of the loop to the next iteration, if the hidden states from the previous iteration were still have a reference to the graph, the gradients will be backpropagated 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 wraps these hidden states into brand new tensors that have no history. This allows the previous graph to go out of scope and free up the memory for the next iteration.
- **Clipping the gradients:** is performed using torch.nn.utils.clip_grad_norm. During the backpropagation, gradients are not clipped until the backward pass is completed and before the model parameters are updated.

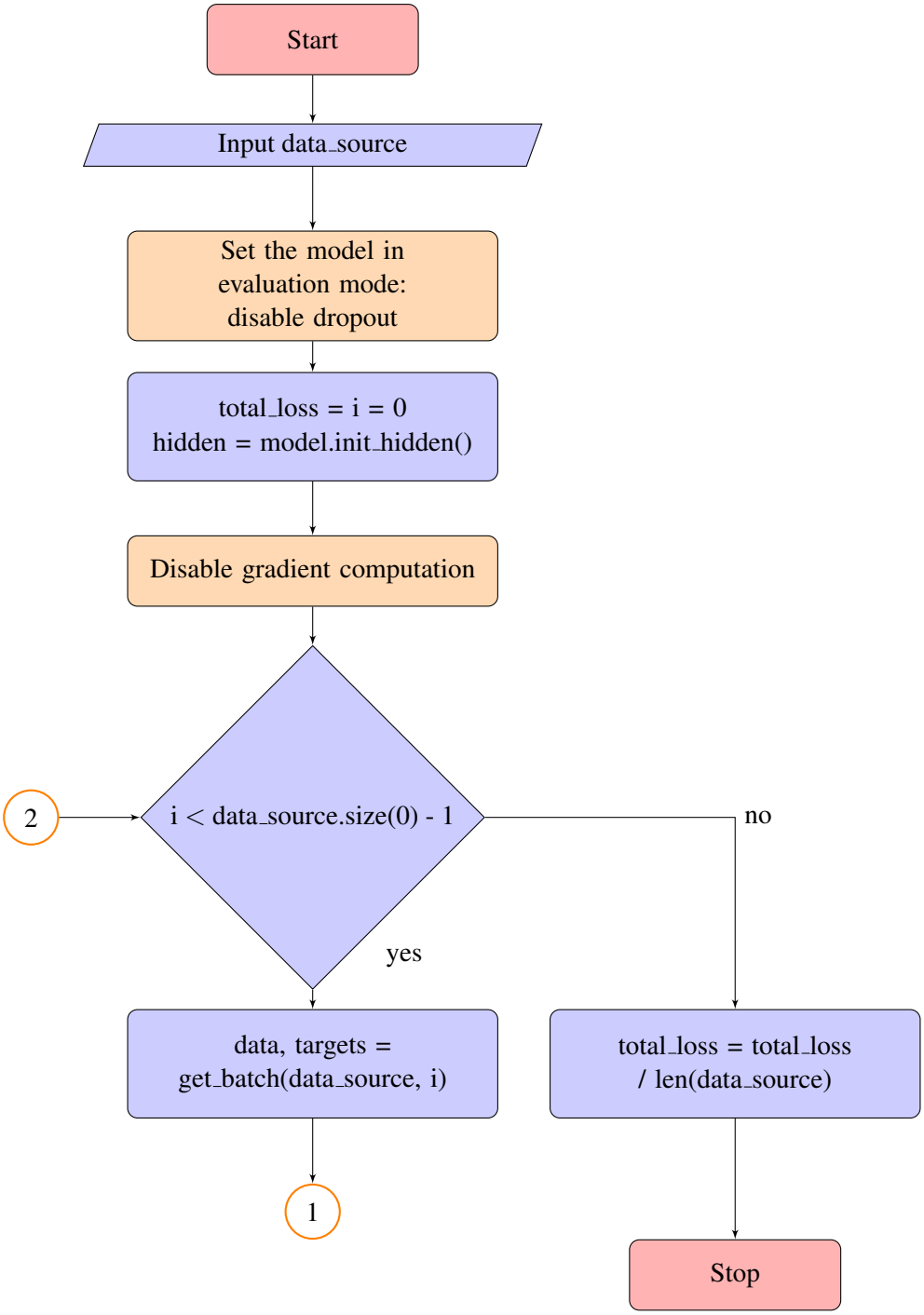

Start
model.train()
total_loss = batch = i = 0
hidden = model.init_hidden()
2
i < train_data.size(0) - 1
no
Stop
yes
data, targets =
get_batch(train_data, i)
hidden = repack-
age_hidden(hidden)
Set gradient of all model
parameters to zero:
model.zero_grad()
Pass data and hid-
den to RNN model:
output,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

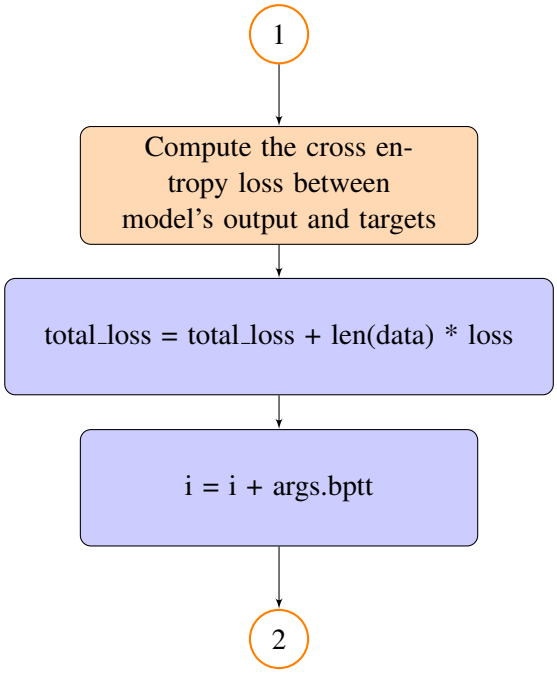

1
Compute the cross entropy loss between model's output and targets
Backpropagate loss: loss.backward()
Clip gradients to avoid exploding gradients
Use the gradients to update the model parameters
total_loss = total_loss + loss
batch modulo args.log_interval
no
yes
total_loss = 0
i = i + args.bptt
2

Main.evaluate() Component

There are not important functions in the evaluate() function which have not already been covered. The major differences with the train function are:

1. No gradients are calculated, no backprogation 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



(b) Find and explain where and how the back-propagation through time (BPTT) 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	Output Shape
Linear	(args.nlayers, ntokens)
Recurrent Network	(args.emsize, args.nlayers)
Dropout	(ntokens, args.emsize)
Embedding	(ntokens, args.emsize)

Recurrent Network is either LSTM, RNN_TANH, RNN_RELU or GRU. The back-propagation will trickle down starting from the loss, to the linear layer, through the recurrent network unrolled computational graph up to and including the embedding layer. BPTT refers to the back-propagation specific to the recurrent network seen as unrolled computational graph. It stops at line 158 with the line hidden = repackage_hidden(hidden) where the hidden states of the model are detached from the computational graph.

(c) Describe why we need the repackage_hidden(h) function, and how it works. At each 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 backpropagated (BPTT) through the RNNs computational graph. From one iteration of the loop to the next iteration, if the hidden states from the previous iteration were still have a reference to the graph, the gradients will be backpropagated 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 wraps these hidden states into, *fresh*, new tensors that have no history. This allows the previous graph to go out of scope and free up the memory for the next iteration.

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

- The encoder which takes a one hot vector of an input word, multiplies it by the a matrix to give a word embedding.
- The decoder which 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 (args.emsize). 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 (reduces 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).

(f) Why do we compute performance on a test set as well? What is this number good for?