Ex04: Comparison of different frameworks - ('Implementing a neural part-of-speech tagger')

Models: DyNet 2.0.3, PyTorch 0.4.1, Tensorflow 1.9.0 & **Model Design (same for all approaches):** word embeddings initialized using GloVe, one-layer bi-directional LSTM, matrix multiplication to produce scores for tags

Difference	**DyNet**	**PyTorch**	**TensorFlow**
Random Seed Initialization: Guaranteeing reproducible results	random seed needs to be set in dynet_config before the actual import of DyNet + setting seed ensures reproducible results; random number generator is initialized with the same starting value	manual_seed(0) sets the random seed after import of PyTorch + setting seed ensures reproducible results; random number generator is initialized with the same starting value	no random seed setting done in this project => but tf.set_random_seed() exists in TensorFlow 1.9 - defining a random seed at the start is important to produce reproducible results. - if no random seed is set TensorFlow automatically picks a random seed (doesn't guarantee reproducibility since it changes)
Consistent Size of Input Vectors	no special padding symbol required but kept in code for consistency => done by DyNet automatically + practical that vectors are automatically padded	Special PAD symbol introduced - manual padding required not bad but additional effort, user needs to be aware of this	Special PAD symbol introduced - manual padding required not bad but additional effort, user needs to be aware of this
Model Definition	Model parameters (e.g., layers, optimizer) defined first, and the computations are only defined when required (during training i.e. computation graph is built in do_pass function) + logical definition (1st design the model, 2nd computations = optimize/train the model) - LSTM initialization parameters are done a lot more explicit than in other models	Model parameters defined as part of constructor of a class, the computations are defined inside functions of the class (e.g., forward() function). + training = method, classification task = class => analogy to object-oriented programming + reusing structures - unsure whether the class design has any real benefits	Both the parameters and the computations must be defined at the start. Starts with new computation graph and placeholders for variables fed into graph during computation. + More definition power (I would assume since every variable is defined manually) - Very verbose: placeholders + all the definitions at the start make it harder to understand at the beginning. More code required to do the same than with other frameworks.
Gradient Clipping	Automatically built-in and activated (disable manually in this implementation) + ensures that exploding gradients problem is automatically taken care of - automatic activation can also be negative (user needs to be aware of this fact, otherwise he searches for other reasons first)	PyTorch Utitilities has two methods: clip_grad_norm and clip_grad_value which would allow to clip the gradients manually. +/- no per default gradient clipping + easier than in TensorFlow - user needs to check gradients if training diverges (optimizer oscillates due to too large gradient steps)	Gradient clipping needs to be performed manually: - compute gradients - clip/modify gradients - apply modified gradients +/- no per default gradient clipping - user needs to check gradients if training diverges (optimizer oscillates due to too large gradient steps)

Difference	DyNet	PyTorch	TensorFlow
LSTM Cell Definition	Requires two LSTM cells to create a bi-directional RNN. + more definition power (cells could potentially differ) - requires the definition of two separate cells +/- initialization is a bit more effort but potentially more definition freedom (different initializations)	Single LSTM cell is sufficient. Bi-directionality can be specified via an input parameter in the constructor of the LSTM class. + efficient and convenient for the end-user - less definition power/freedom	To create a bi-directional RNN two separate LSTM cells need to be defined (a forward and a backward LSTM cell) + more definition power / possibilities for the end-user - for any action the LSTM cells need to be wrapped - more work than PyTorch
Learning Rate Scheduling	Computation manually defined and updated during each epoch of the training + less boilerplate code than PyTorch, update is defined, directly computed and then reassigned	Scheduler: defined together with the optimizer and the initial learning rate. Defined as a lambda function which receives the current number of epochs at each iteration and then multiplies it by the initial learning rate. + once defined, method invocation at the start of each epoch - boilerplate code (more lines of code to achieve the same compared to TensorFlow or DyNet)	Computation manually defined and updated during each epoch of the training + less boilerplate code than PyTorch, update is defined, directly computed and then reassigned
Recurrent Dropout Regularization	Can be set directly on each LSTM (e.g., b_lstm.set_dropouts(0.0) => disable input and output variational dropout of this cell) + very simple to define like TensorFlow directly on LSTM cell	PyTorch 0.4 doesn't support recurrent dropout directly => use toolkit (WeightDrop class) - no support for recurrent dropout, only via separate class	Can be specified via parameter of Dropout wrapper cell for each LSTM cell. + simple to define directly on LSTM cell like DyNet
Input / Output Dropout	Is applied manually to input (dy.dropout(weights, 1-PROB)) and automatically rescales the outputs + very practical: only needs to be applied during training since DyNet automatically rescales the outputs by 1/PROB	Input (Word) and Output (LSTM) dropout are defined in the constructor of the TaggerModel class and applied during training (calls of the forward method) +/- (neutral – not very bad): dropout is manually applied before and after LSTM cell	Basic LSTM cell wrapped in a Dropout Wrapper Cell which applies the dropout to both input and output. Dropout probabilities for input and output can be set individually. - dropout is always applied and cannot be turned off during prediction. **Reason:** The computation graph cannot be changed. **Solution:** dropout probability = 1.0 during prediction to keep all values.
Weight Decay	Defined globally in dynet_config (at import time) => contains parameter weight_decay + allows for weight decay - fixed weight decay (rescales weights with the same factor at each update)	Directly defined inside the optimizer initialization as a constant + allows for weight decay, defined logically => in optimizer - fixed weight decay (same decay at each update)	TensorFlow 1.9 doesn't support weight decay yet => fixed by pull-request in TensorFlow 1.10 - no weight decay possible