PyTorch Guide for Programming Assignment 2

1. Neural Network Models

In PyTorch, the neural network models are defined under the **torch.nn** package. It is imported into the system by the following line of code:

```
import torch.nn as nn
```

This package consists of a number of neural network models, including embedding, feed-forward linear transformation, convolutional neural network (CNN), and long short-term memory (LSTM). These models can be composed to form a single model for POS tagging. The key models implemented in the **torch.nn** package are:

```
# embedding layer, i.e., character -> vector or word -> vector
nn.Embedding(num_embeddings, embedding_dim, padding_idx=None,
      max_norm=None, norm_type=2, scale_grad_by_freq=False, sparse=False,
      _weight=None)
```

Hyper-parameters:

- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **padding_idx** (*int, optional*) – If given, pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index.
- **max_norm** (*float, optional*) – If given, will renormalize the embedding vectors to have a norm lesser than this before extracting.
- **norm_type** (*float, optional*) – The p of the p-norm to compute for the max_norm option. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – if given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** (*bool, optional*) – if `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

```
# feed-forward linear transformation
nn.Linear(in_features, out_features, bias=True)
```

Hyper-parameters:

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to False, the layer will not learn an additive bias. Default: `True`

```
# 1D convolutional neural network
nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0,
      dilation=1, groups=1, bias=True)
```

Hyper-parameters:

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0

- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

```
# long short-term memory (LSTM)
nn.LSTM(input_size, hidden_size, num_layers=1, bias=True,
      batch_first=False, dropout=0, bidirectional=False)
```

Hyper-parameters:

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in the outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, a bidirectional LSTM is used. Default: `False`

The **torch.nn** package also defines a number of loss functions. A useful loss function is the negative log-likelihood which takes as input the log-probabilities stored in a tensor:

```
# negative log-likelihood loss (all parameters are optional)
nn.NLLLoss(weight=None, ignore_index=-100, reduction='elementwise_mean')
```

2. Optimizer

The optimizer is implemented from the **torch.optim** package. The import syntax is:

```
import torch.optim as optim
```

Two best-known examples are stochastic gradient descent (SGD) and Adam:

```
optim.SGD(params, lr, momentum=0, weight_decay=0, dampening=0,
      nesterov=False)
```

Hyper-parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

```
optim.Adam(params, lr=0.001, betas=(0.9,0.999), eps=1e-08, weight_decay=0,
      amsgrad=False)
```

Hyper-parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups

- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **amsgrad** (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm (From the ICLR 2018 paper titled "On the Convergence of Adam and Beyond").

3. Non-Linear Activation Functions

Non-linear activation functions are defined in the **torch.nn.functional** package, which can be imported by

```
import torch.nn.functional as F
```

Examples of these functions are:

```
# rectified linear units (ReLU)
F.relu(input, inplace=False) → Tensor
```

Function parameters:

- **inplace** – can optionally do the operation in-place. Default: False

```
# softmax to make vector values sum to 1
F.softmax(input, dim=None)
```

Function parameters:

- **dim** (*int*) – A dimension along which Softmax will be computed (so every slice along dim will sum to 1).

4. Loading and Saving Models

PyTorch supports loading and saving model parameters using its built-in functions. A sample usage of saving the vocabulary (string to index mapping) and the model files:

```
vocab = {} # mapping token string to integer index
model = Model()
# model training
# …
torch.save((vocab, model.state_dict()), model_file)
```

A sample usage of loading a pre-trained model:

```
model = Model()
vocab, model_state_dict = torch.load(model_file)
model.load_state_dict(model_state_dict)
```