UDACITY

Return to "Deep Learning" in the classroom

DISCUSS ON STUDENT HUB

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

You have done a tremendous job on the project!! Keep it up.

Few things you could try in your free time:

1. Play with seq length(Think of seq length as average sentence length in the training corpus)
2. Try with GRU and play with hyperparameters accordingly

Please find the pointers to interesting articles on applications of LSTM's

1. Language translation: https://chunml.github.io/ChunML.github.io/project/Sequence-To-Sequence/
2. Sequence tagging: https://www.depends-on-the-definition.com/sequence-tagging-lstm-crf/

and a link to the deep learning dictionary: https://towardsdatascience.com/the-deep-learning-ai-dictionary-ade421df39e4
Good luck :)

PyTorch cheatsheet:

Cheat Sheet

Pytorch is an machine learning library based on Torch built by facebook.Two important features:

- Tensor computation (like NumPy) with strong GPU acceleration.
- Provide Dynamic computation graphs with imperative paradigm.

1 TENSOR

Make use of the following alias to import the libraries.

```
>>> import torch
>>> import torchvision
```

Tensor is a multidimensional array of numbers.There are some basic tensor operations.

```
>>> x = torch.randn(3, 2)
>>> y = torch.randn(2, 3)
>>> x + y
>>> z = x.mm(y)
>>> z.t()
>>> torch.ones(3,3)
>>> z[1,]
>>> z[:,1]
>>> z[:,1] = z[:,1] + 1
>>> z[1:2,1:2]
>>> z.view(9)
>>> z.view(-1,9)
>>> z.numpy()
```

2 AUTOGRAD

Mechanism of error gradient calculation and back-propagate the error through computation graph is called Autograd in pytorch.

```
>>> from torch.autograd import Variable
>>> x = Variable(torch.ones(2, 2) * 2, requires_grad=True)
>>> z = 2 * (x * x) + 5 * x
>>> z.backward(torch.ones(2,2))
>>> z.grad_fn
>>> z.grad_fn.next_functions[0]
>>> print(x.grad)
```

1. First Create a Variable class which wrap the tensor and specifies variable requires a gradient to be True then allow gradient computation with .backward() function.
2. dz/dx calculated to be 4x+5.all elements of x are 2,so gradient dz/dx to be a (2,2) shape tensor with values 13.

3 NN MODULE

The nn Package defines the set of modules think as a neural network layer produces output from input and have some trainable weights.

```
>>> model = torch.nn.Sequential(
 torch.nn.Linear(input_num_units, hidden_num_units),
 torch.nn.ReLU(),
 torch.nn.Linear(hidden_num_units, output_num_units),
 )
>>> model[0].weight
>>> model[0].bias
```

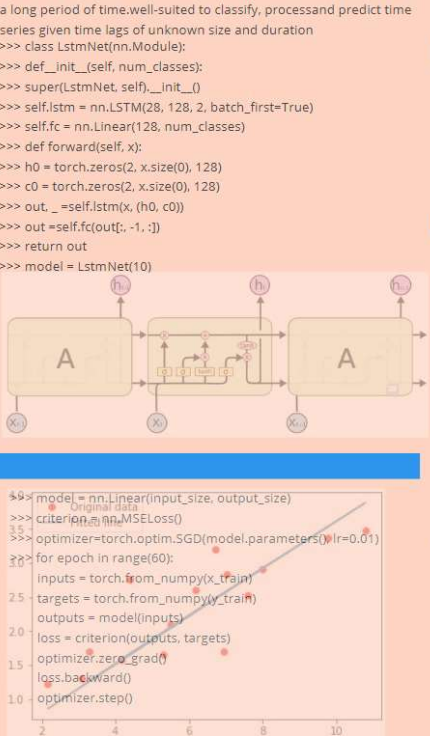
For more complex models than sequence of existing models create custom neural network class by inheriting nn.Module class.

```
>>> class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.linear = nn.Linear(1, 1)
        def forward(self, x):
            y =self.linear(x)
            return y
```

4 LINEAR REGRESSION

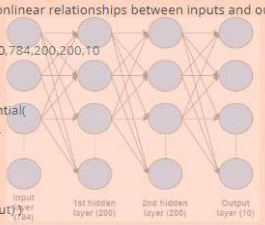
Simple linear regression is useful for finding relationship between two continuous variables. One is predictor or independent variable and other is response or dependent variable

```
>>> import torch.nn as nn
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x_train = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168], [9.779], [6.182], [7.59], [2.167], [7.042], [10.791], [5.313], [7.997], [3.1]], dtype=np.float32)
>>> y_train = np.array([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573], [3.366], [2.596], [2.53], [1.221], [2.827], [3.465], [1.65], [2.904], [1.3]], dtype=np.float32)
```




```
>>> model = nn.Linear(input_size, output_size)
>>> criterion = nn.MSELoss()
>>> optimizer=torch.optim.SGD(model.parameters(),lr=0.01)
>>> for epoch in range(60):
    inputs = torch.from_numpy(x_train)
    targets = torch.from_numpy(y_train)
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

modeling and processing nonlinear relationships between inputs and outputs in parallel.

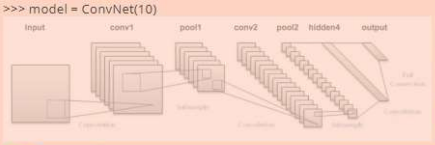



```
>>> import torch
>>> N,D_in,H1,H2,D_out = 50,784,200,200,10
>>> x=torch.randn(N,D_in)
>>> y=torch.randn(N,D_out)
>>> model=torch.nn.Sequential(
    torch.nn.Linear(D_in,H),
    torch.nn.ReLU(),
    torch.nn.Linear(H,H1),
    torch.nn.reLU(),
    torch.nn.Linear(H1,D_out))
>>> loss=torch.nn.MSELoss(size_average=False)
>>> lr=1e-4
>>> optim=torch.optim.SGD(model.parameters(),lr=l
>>> for t in range(500):
>>> y_pred=model(x)
>>> loss1=loss(y_pred,y)
>>> print(t,loss1)
>>> optim.zero_grad()
>>> loss.backward()
>>> optimizer.step()
```

6 CNN

CNN or ConvNet is a class of feed-forward artificial neural networks, most commonly applied to analyzing visual imagery.

```
>>> class ConvNet (nn.Module):
>>> def__init__(self, num_classes=10):
>>> super(ConvNet, self).__init__()
>>> self.layer1 = nn.Sequential(
>>> nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
>>> nn.MaxPool2d(kernel_size=2, stride=2))
>>> self.layer2 = nn.Sequential(
>>> nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
>>> nn.MaxPool2d(kernel_size=2, stride=2))
>>> self.fc = nn.Linear(7*7*32, num_classes)
>>> def forward (self, x) :
>>> out =self.layer1(x)
>>> out =self.layer2(out)
>>> out = out.reshape(out.size(0), -1)
>>> return self.fc(out)
>>> model = ConvNet(10)
```



7 LSTM

LSTM is a model for the short-term memory which can last for a long period of time.well-suited to classify, processand predict time series given time lags of unknown size and duration

```
>>> class LstmNet(nn.Module):
>>> def__init__(self, num_classes):
>>> super(LstmNet, self).__init__()
>>> self.lstm = nn.LSTM(28, 128, 2, batch_first=True)
>>> self.fc = nn.Linear(128, num_classes)
>>> def forward(self, x):
>>> h0 = torch.zeros(2, x.size(0), 128)
>>> c0 = torch.zeros(2, x.size(0), 128)
>>> out, _ =self.lstm(x, (h0, c0))
>>> out =self.fc(out[:, -1, :])
>>> return out
>>> model = LstmNet(10)
```


OPTIM MODULE

Optimization algorithms like stochastic gradient descent,AdaGrad, RMSProp, Adam, etc.can be used to update the weights and change the learning rate dynamically.

```
>>> optimizer = Adam(model.parameters(),lr=lr)
```

LOSS FUNCTION

Various loss functions available are:
NLLLoss,MSELoss,CrossEntropyLoss

```
>>> loss=torch.nn.MSELoss(size_average=False)
```

PRETRAINED MODELS

Use Pre-trained models from torchvision package.
some available models are:
VGG,ResNet,DenseNet,Inception

```
>>> import torchvision.models as models
>>> resnet = models.resnet18(pretrained=True)
>>> for param in resnet.parameters():
>>> param.requires_grad =False
>>> resnet.fc = nn.Linear(resnet.fc.in_features, 100)
```

freeze the weights for all the layers except the last layer.

SAVE AND LOAD

```
>>> torch.save(resnet, 'model.ckpt')
>>> model = torch.load('model.ckpt')
```

For only saving the model parameters

```
>>> torch.save(resnet.state_dict(), 'params.ckpt')
>>> resnet.load_state_dict(torch.load('params.ckpt'))
```

Data Augmentation

Data augmentation using torchvision transforms:

```
>>> transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.Scale(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    ])
```

SAMPLE DATASETS

There are some sample datasets are available in torchvision.datasets:
MNIST,CIFAR,Imagenet

```
>>> torchvision.datasets.CIFAR10(root='/data',
    train=True,
    download=True,
    transform=transforms.ToTensor())
```

DataLoader

Load the data in batches using torch util.

```
>>> import torch.utils.data as data_utils
>>> train = data_utils.TensorDataset(features, targets)
>>> train_load = data_utils.DataLoader(train,
batch_size=50, shuffle=True)
```

8 SOURCES

- https://pytorch.org/docs/master/index.html
- https://pytorch.org/tutorials/index.html
- https://github.com/pytorch/examples
- https://www.udemy.com/practical-deep-learning-with-pytorch/
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Rednivrug

All Required Files and Tests

The project submission contains the project notebook, called “dlnd_tv_script_generation.ipynb”.

All the unit tests in project have passed.

Pre-processing Data

The function `create_lookup_tables` create two dictionaries:

- Dictionary to go from the words to an id, we'll call vocab_to_int
- Dictionary to go from the id to word, we'll call int_to_vocab

The function `create_lookup_tables` return these dictionaries as a tuple (vocab_to_int, int_to_vocab).

Good use of dictionary comprehensions.

The function `token_lookup` returns a dict that can correctly tokenizes the provided symbols.

Batching Data

The function `batch_data` breaks up word id's into the appropriate sequence lengths, such that only complete sequence lengths are constructed.

In the function `batch_data`, data is converted into Tensors and formatted with TensorDataset.

Finally, `batch_data` returns a DataLoader for the batched training data.

In the batch data function,
sequencing is done perfectly
the code is clean and readable
data is converted to tensors and
Data loader is returned successfully.

Build the RNN

The RNN class has complete `__init__`, `forward`, and `init_hidden` functions.

The RNN must include an LSTM or GRU and at least one fully-connected layer. The LSTM/GRU should be correctly initialized, where relevant.

RNN class contains **init**, forward and init_hidden functions which are implemented perfectly.
LSTM layers are stacked and fully connected layers are used on top of it.
Good work!

Further Resource: Keras has a neat API for visualizing the architecture, which is very helpful while debugging your network. pytorch-summary is a similar project in PyTorch.

RNN Training

- **Enough epochs to get near a minimum in the training loss, no real upper limit on this. Just need to make sure the training loss is low and not improving much with more training.**
- **Batch size is large enough to train efficiently, but small enough to fit the data in memory. No real "best" value here, depends on GPU memory usually.**
- **Embedding dimension, significantly smaller than the size of the vocabulary, if you choose to use word embeddings**
- **Hidden dimension (number of units in the hidden layers of the RNN) is large enough to fit the data well. Again, no real "best" value.**
- **n_layers (number of layers in a GRU/LSTM) is between 1-3.**
- **The sequence length (seq_length) here should be about the size of the length of sentences you want to look at before you generate the next word.**
- **The learning rate shouldn't be too large because the training algorithm won't converge. But needs to be large enough that training doesn't take forever.**

Perfect choice of hyperparameters. Nothing negative to call out here.

The printed loss should decrease during training. The loss should reach a value lower than 3.5.

There is a provided answer that justifies choices about model size, sequence length, and other parameters.

Nice to see that you have played with different hyperparameter settings and you were able to get the optimal hyperparameters and meet the project requirements.

Generate TV Script

The generated script can vary in length, and should look structurally similar to the TV script in the dataset. **It doesn't have to be grammatically correct or make sense.**
Generated script looks good to me.

DOWNLOAD PROJECT

RETURN TO PATH