Tutorial 10: Recurrent neural networks

Tutorial objectives:

- **Hands on practice with recurrent neural networks**
- **Train a text-generating recurrent neural network**

Simple recurrent neural networks

Exercise 1: For this exercise you will train a Simple Recurrent Network (SRN) to generate text. The sklearn library doesn't provide a ready made SRN classifier, but we have prepared one for you that mimics sklearn's API. As in the previous tutorial, the objective of this exercise is not to get you to implement a learning algorithm, but rather to have you figure out how to use ready made tools for the task at hand.

Create a new project in PyCharm (remember to select cosc343 environment for the interpreter), download `COSC343SRNClassifier.py` and `COSC343WordEnc.py` from Blackboard into your project folder.

In this part of the exercise you will create a script that trains your SRN and saves it to file. Create a new Python script for SRN training. You need training data. Find a piece of text to train the network on – should be something at least few thousand words long (if you're too lazy to find your own, you can download pride.txt from Blackboard, which contains the text of Jane Austen's *Pride and Prejudice*). To read-in a text file (found in the same folder as your script) into a Python string you can do:

```
with open(filename, "r") as f:
    all_text = f.read()
```

where `filename` is the string with the name of your file and `all_text` is the resulting text string. SRN processes vectors, not text, so you need to parse the text, create a dictionary of word tokens and assign one hot encoding vector to each word. To simplify this task for you, we provided a word encoder – import it at the top of your file like so:

```
from COSC343WordEnc import COSC343WordEnc
```

Here's a description of the encoder's API.

class `COSC343WordEnc`(max_text_length=None, max_words=None)

Word to one-hot encoder.

This encoder can transform text into a set of one-hot vectors and do an inverse transform of one-hot vector back to text.

Parameters: **max_text_length: int**
Maximum number of words from text string to use to create the encoding.

max_words
Maximum number of words in the dictionary.

Methods

`fit`(X,y=None)
Create dictionary from text string.

Parameters: **X: string**
Text string to create the dictionary from; only the `max_text_length` words are used if its not None; dictionary is restricted to `max_words` if not None.

y: Ignored

Returns: **self: object**
Returns the instance itself.

`fit_transform`(X,y=None)
Create dictionary from text string and transform the text to one-hot encoding.

Parameters: **X: string**
Text string to create dictionary from and transform; only the `max_text_length` words are used if its not None; dictionary is restricted to `max_words` if not None.

y: Ignored

Returns: **X_new: ndarray of shape (n_words, n_dict_size)**
Returns X as a one-hot encoding, where *n_words* is the number of converted tokens, and *n_dict_size* is the size of the dictionary.

`transform`(X,y=None)
Transform string to one-hot encoding.

Parameters: **X: string**
Text string to transform

y: Ignored

Returns: **X_new: ndarray of shape (n_words, n_dict_size)**
Returns X as a one-hot encoding, where *n_words* is the number of converted tokens, and *n_dict_size* is the size of the dictionary.

`inverse_transform`(X)
Transform one-hot encoding back to words.

Parameters: **X: ndarray of shape (n_words, n_dict_size)**
Text string to transform

y: Ignored

Returns: **X_new: string**
Returns X as a text string.

For example, the code

```
word_enc=COSC343WordEnc(max_text_length=10000)
X = word_enc.fit_transform(all_text)
```

will create encoder from the first 10,000 word tokes found in `all_text` (encoder treats punctuation as separate tokens) and return a one-hot encoded vector of X of shape (10000,`num_unique_tokens`), where `num_unique_tokens` is the number of unique tokens found in the first 10000 tokens of `all_text`.

At the top of your script import the `COSC343SRNClassifier`:

```
from COSC343SRNClassifier import COSC343SRNClassifier
```

Use it to create and fit the SRN model for text prediction. `COSC343SRNClassifier` uses a subset of sklearn's MLPClassifier's API, so you should know (if you've done the Perceptron and Multi-layer Perceptron labs) how to use it. Here's the detailed description of its API

class `COSC343SNRClassifier`(hidden_layer_size=100, activation='relu', solver='adam', batch_size='auto', learning_rate_init=0.001, max_iter=200, verbose=False, momentum=0.9, beta_1=0.9, beta_2=0.999, epsilon=1e-08)

Simple recurrent network classifier.

This model optimises the cross-entropy loss over softmax otuput using stochastic gradient descent or the adam optimiser.

Parameters: **hidden_layer_size: int, default=100**
Number of neurons in the single hidden layer.

activation: {'identity', 'logistic', 'tanh', 'relu'}, default='tanh'
Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns $f(x) = x$
- 'logistic', the logistic sigmoid function, returns $f(x) = 1/(1 + e^{-x})$.
- 'tanh', the hyperbolic tan function, returns $f(x) = \tanh(x)$.
- 'relu', the rectified linear unit function, returns $f(x) = \max(0, x)$.

solver : {'sgd', 'adam'}, default='adam'
The solver for weight optimisation.

- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimiser proposed by Kingma, Diederik, and Jimmy Ba.

batch_size: int, default='auto'
Size of minibatches for stochastic optimisers. When set to 'auto', `batch_size=min(200, n_samples)`.

learning_rate_init: double, default=0.001
The learning rate that controls the step-size in updating the weights.

max_iter: int, default=200
This determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

verbose: bool, default=False
Whether to print progress messages to stdout.

momentum: float, default=0.9
Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

beta_1: float, default=0.9
Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver='adam'.

beta_2: float, default=0.999
Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver='adam'.

epsilon: float, default=1e-8
Value for numerical stability in adam. Only used when solver='adam'.

Methods

`fit`(X,y)
Fit the model to data matrix X and target(s) y.

Parameters: **X: ndarray of shape (n_samples, n_features)**
The input data.

y: ndarray of shape (n_samples, n_outputs)
The target values.

Returns: **self: a trained SRN model**.

`predict`(X,sampled=1)
Predict using the SRN classifier.

Parameters: **X: ndarray of shape (n_features,) representing a single sample, or ndarray of (n_samples, n_features)**
The input data.

sampled: int, default=1
Number of top neurons to sample for declaring output class.

Returns: **y: ndarray, shape (n_outputs,) or (n_samples, n_outputs)**
The predicted classes.

`reset()`
Resets the SRN context vector to zero.

Returns: **None**

Decide on the architecture and hyper-parameters of your SRN, instantiate it and train it on the encoded text to predict the next work in the text sequence. It might be a good idea to limit the encoder to first 10K words (otherwise training will take forever). **Hint!** Given `X`, and ndarray of shape `n_samples,n_featurers`, you can select the set of `n_samples-1` samples starting at index `0` like so: `X[:-1]`; similarly you can select the set of `n_samples-1` samples starting at index `1` like so: `X[1:]`).

You need to save the trained classifier along with the encoder (since a given SRN is trained for a specific encoder). You can pickle them both into one file like so:

```
import pickle, gzip
.
.
with gzip.open("srn\_training.save", 'w') as f:
    pickle.dump((net,word_enc), f)
```

where `net` and `word\_end` are your SRN model and word encoder respectively, and "srn_training.save" is the name of the file.

It's a good idea to create a separate script for text prediction. This way you can run training script once, and then run the text generation script a number of times. Create a new Python script, and load the trained SRN model along with the word encoder like so:

```
import pickle, gzip

with gzip.open("model.save") as f:
    net, word_enc = pickle.load(f)
```

Use SRN's `reset` method to reset the context vector. Then create a primer string of your choosing (a string of few words that can be found in the dictionary), covert it to one-hot encoding and pass it through the model using its `predict` method. The last vector of the output will be the one-hot encoded vector representing word that should follow the last word in the primer string (as predicted by the model). The context vector is kept intact between calls to `predict` method, so you can call it in a loop, where the the last predicted word is used as input (in the first iteration of the loop the context will be related to the primer string). Use the encoder's `inverse_transform` method to convert the prediction back to text and observe the

results. Hint! You can use the `sampled` argument of the `predict` method to return a random choice of the next word precition based on top `sampled` outputs of softmax. For instance, if `sampled=3`, the SNR model will return one of three top next predictions. This way you can generate different text for the same primer.

Try training network on different text styles and see how to generation goes.