# Tutorial 11: Simple recurrent network (SRN)

**Tutorial objectives:**

- **Hands on practice with recurrent neural networks**
- **Experience practical machine learning, where the challenge is not implementing the learner, but to convert data into the format that the learner can work with**
- **Train a text-predicting/generating recurrent neural network**
- **Implement a simple text-predictor/generator**

## Sequential environment concepts

1. How does a sequential environment differ from the episodic environment?
2. In an SRN with $K$ inputs and $K$ softmax outputs trained to predict the next word in a sequence of one-hot encoded words, what does the output after the softmax activation convey?
3. How can we an agent, trained to predict the next word in a sentence, be used to generate arbitrary long sentences?

## Text prediction/generation with SRN

In this tutorial you will train a Simple recurrent network (SRN) to predict and generate text. The Sklearn library doesn't provide a ready made SRN classifier, so one was prepared 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. This time around though, there is am extra twist – you need to bring the data in and convert it into the format that the SRN classifier can work with. This is a common challenge for machine learning in practice – the learner is implemented for you and the hard part is formatting of the data into something that the learner can work with.

*class* `WordEnc`(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_words**
Maximum number of words to use when fitting.

**Methods**

`fit`(X)
Create dictionary from text string.

**Parameters:** **X: string**
Text string to create the dictionary from; process the first `self.max_words` from the string if `self.max_words` is not None.

`transform`(X)
Transform string to one-hot encoding.

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

**Returns:** **Numpy array 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. Processes only the first *self.max_words* from *X* if *self.max_words* not set to None.

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

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

**Returns:** string
Returns X as a string of *n_words* words.

**Exercise 1:** In this exercise you need to create an encoder/decoder (not quite like the encoder/decoder we talked about for dimensionality reduction in unsupervised learning) that converts text into one hot encoding and converts one hot encoding back to text. Create a new project in PyCharm (remember to select cosc343 environment for the interpreter), and create a new script named `WordEnc.py`, inside of which you should implement a class `WordEnc` that follows the API shown above. The encoder should implement a `fit` method, which creates a dictionary of words/tokens from the text given and assign unique index to each word. Its `transform` method should convert a string of words to an $N \times K$ Numpy array, where $N$ is the number of words in the string, $K$ is the number of words in the dictionary and the $n^{th}$ row is a one-hot encoding of the $n^{th}$ word according to the index determined in the `fit` method. It's up to you which symbols in the text string your encoder supports, and which ones are ignored. For instance, you might choose to ignore all punctuation, and then your predictor will not include punctuation. Similarly, you may choose to differentiate or not between the upper and lower case words.

When your encoder works as intended, it should work (in another script) something along those lines:

```
from WordEnc.py import WordEnc

# Create the encoder
my_word_enc=WordEnc(max_words=10000)

# Create the dictionary inside the encoder
my_word_enc.fit(some_string_with_text)

# Convert text to one-hot encoded matrix
X = my_word_enc.transform(some_sring_with_text)

# Convert one-hot encoded matrix back to a string
a_string= my_word_enc.inverse_transform(X)
```

**Exercise 2:** In this exercise you need to create a script that trains an SRN on a piece of text. Download `SRNClassifier.py` rom Blackboard into your project folder. 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*). The SRNClassifier, all implemented for you, accepts one-hot encoded input and one-hot encoded target output for training. It works exactly like any Sklearn learner (which you should be familiar with form the previous tutorials). The detailed API of the `SRNClassifier` is given below.

*class* `SNRClassifier`(hidden_layer_size=100, activation='tanh',
solver='sgd', learning_rate_init=0.001,
max_iter=200, verbose=False)

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='sgd'**
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.

**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.

**Methods**

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

**Parameters:** **X: Numpy array of shape (n_samples, n_features)**
The input data in one-hot encoding.

**y: Numpy array of shape (n_samples, n_outputs)**
The target values in one-hot encoding.

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

`predict`(X)
Output of the SRN classifier in one-hot encoding.

**Parameters:** **X: Numpy array of shape (n_samples, n_features)**
The input data in one-hot encoding.

**Returns:** **y: Numpy array of shape (n_samples, n_outputs)**
The predicted classes in one-hot encoding.

`predict_proba`(X)
Output of the SRN classifier as probability distribution over classses.

**Parameters:** **X: Numpy array of shape (n_samples, n_features)**
The input data in one-hot encoding.

**Returns:** **y: Numpy array of shape (n_samples, n_outputs)**
Predicted probability distribution of outputs (each rows is a probability distributions over predicted outputs) .

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

**Returns:** **None**

Once you have trained the SRN, 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_trained.save", 'w') as f:
    pickle.dump((srn,word_enc), f)
```

where `srn` and `my_word_enc` are your SRN model and word encoder respectively, and "srn_trained.save" is the name of the file.

**Hints!**

- It might be a good idea to limit the encoder to first 10K words (otherwise training will take forever).
- Given `X`, and Numpy of shape `N×n_featurers`, you can select the set of `N-1` samples starting at index `0` like so: `X[:-1]`; similarly you can select the set of (`N-1`) samples starting at index `1` like so: `X[1:]`).
- It's a good idea to enable verbose mode in SRN and monitor the loss (it should be going down) and the score/accuracy (it should be going up) as it trains. If it's training rather slow, try switching from the 'sgd' solver to 'adam'. The 'adam' solver is a version of 'sgd' that essentially adjust the learning rate to take larger steps in the loss surface when it's flat. The learning rate you specify in `learning_rate_init` is still used as the base learning grate, so you still need to set this up correctly – not too big, and not too small. The 'adam' solver should allow faster convergence to a better accuracy.

**Exercise 3:** In this exercise you need to build a text predictor that works on top of the SRN you trained in the previous exercise. Create a new Python script, and load the trained SRN model along with the word encoder like so:

```
import pickle, gzip

with gzip.open("srn_trained.save") as f:
    srn, my_word_enc = pickle.load(f)
```

Create a priming sequence of words, pass them through SRN, and use the output after the last words to predict what words that should follow. Pick the four candidates that the network considers most likely words to follow, and print them as suggestions

after the priming sentence. For instance, the output for the priming sequence "It is time to" should be something like this:

| It is time to [go/talk/work/maybe] |
|---|

**Hints!**

- If you're priming your SRN with several sequences (say in a loop), you can use the model's `reset` method to reset the context;
- Remember, you have to convert strings to one hot encoding before feeding them into the model and back to string from model's output;
- The `predict` method of SRN gives you one hot encoding indicating the most likely word to follow the sequence of words processed by the model; you need to use the output of `predict_proba`, which gives the softmax output of the SRN before converting to one-hot encoding, and then you can find the indices of the four most likely candidate words...and then you still need to express them in one-hot encoding and convert them to strings.

**Exercise 4:** In this exercise you need to setup SRN to generate text. As in the previous exercise load the trained SRN and the word encoder. Then prime SRN with a string of your choosing, something like "It is time to", and from then on use the output of the model as the next input...and keep going for some number of words. Convert all the output to text and display the generated text.

If the results are not great, maybe you need to retrain SRN to a better accuracy, or on a different text to obtain different style of text generation.