Author: Ana-Maria Vintila, based off work from Srijith Rajamohan based off the work by Robert Guthrie

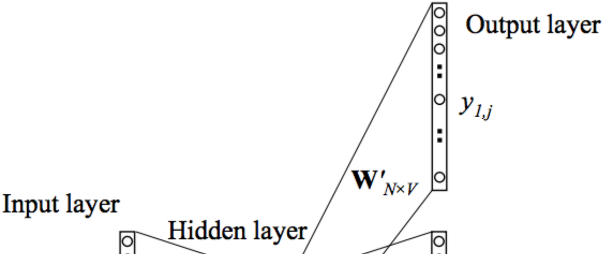
Source: https://srijithr.gitlab.io/post/word2vec/

```
import os
from IPython.display import Image

pth = os.getcwd()
pth
```

```
'/content/gdrive/My Drive/StatFitScholarshipProject/PythonN
```

```
Image(filename=pth + '/src/NLPstudy/images/Skip-gram.png')
```



Step 1: Initialization

Here we set the context window size to 3 words and the word embedding dimension to 10, and also pass in the text corpora from which we build vocabulary.

Tokenizing the text occurs later while reading in the data.

```
CONTEXT_SIZE = 3
EMBEDDING_DIM = 10

testSentence = """Empathy for the poor may not come easily
They may blame the victims and insist their predicament ca
and hard work.
But they may not realize that extreme poverty can be psych
incapacitating - a perpetual cycle of bad diets, health ca
by the shaming and self-fulfilling prophecies that define
Gordon Parks - perhaps more than any artist - saw poverty
afflictions" and realized the power of empathy to help us
abstract problem nor political symbol, but something he en
Kansas and having spent years documenting poverty througho
```

Step 2: Build the *n*-grams

Next we build the *n*-grams, or sequence of words, as a list of tuples.

Each tuple is ([$\texttt{word}_{i-2}$, $\texttt{word}_{i-1}$], `targetWord`)

```
ngrams = []
for i in range(len(testSentence) - CONTEXT_SIZE):
    tup = [testSentence[j] for j in np.arange(i + 1, i + CO
    # skip-gram way of appending:
    ngrams.append( (testSentence[i], tup) )
    # cbow# ngrams.append( (tup, testSentence[i + CONTEXT_S
```

```
ngrams[:20] # showing a few sample n-grams
```

```
[('Empathy', ['for', 'the', 'poor']),
 ('for', ['the', 'poor', 'may']),
 ('the', ['poor', 'may', 'not']),
 ('poor', ['may', 'not', 'come']),
 ('may', ['not', 'come', 'easily']),
 ('not', ['come', 'easily', 'to']),
 ('come', ['easily', 'to', 'people']),
```

Step 3: Create Vocabulary

Create the vocabulary by converting the text into a set to remove duplicate words.

```
vocabulary = set(testSentence)
```

```
len(vocabulary)
list(vocabulary)[:20] # showing first 20 words in vocabular
```

```
['lived',
 'psychologically',
 'prophecies',
 'Gordon',
 'Silva',
 'They',
 'abstract',
 'perpetual',
 'hills',
 'Silva,',
 'power',
 'magazine',
```

Step 4: Create Map of Words to Indices

Creating word to index map that prints the key (word) corresponding to the given index in the dictionary argument. Basically, we get a list of tuples (number, word) from zipping the sequence $0, 1, 2, 3....$ with the vocabulary word list.

```
wordToIndex = {word : i for i, word in enumerate(vocabulary
```

```
# Showing first 20 word to index pairs
len(wordToIndex)
itemsList = list(wordToIndex.items())
itemsList[:20]
```

```
[('lived', 0),
 ('psychologically', 1),
 ('prophecies', 2),
 ('Gordon', 3),
 ('Silva', 4),
 ('They', 5),
 ('abstract', 6),
 ('perpetual', 7),
```

Step 6: Create Skip-Gram Model

The skip-gram neural network has three components:

1. embedding layer, created using pytorch's `nn.Embedding`, to convert tensors into word embeddings.
2. hidden layer, in this case it is a linear layer.
3. output layer, in this case also linear layer.

Forward Pass of Skip-Gram:

1. Convert the tensor `inputs` to word embeddings via the skip-gram's `nn.Embedding` layer
2. Pass the embeddings to the hidden layer and transform the result using the `relu` function
3. Transform the hidden layer results using the output layer.
4. Finally, create a probability distribution over words using the `softmax` function. (Here we actually use the `log_softmax` so the results are log probabilities instead of probabilities.)

Predictions:

Step 7: Train the Skip-Gram Model

Training the model requires the following steps:

1. Convert the context words into integer indices using the `wordToIndex` dictionary, and make their type a `Tensor`.
2. Set the model gradients to zero so they do not accumulate artificially (feature of pytorch)
3. Do the `forward` pass of the Skip-Gram model, resulting in the log probabilities of the context words.
4. For each word in the correct target context wrods, convert it to an index using the `wordToIndex` dictionary and wrap it in a `Tensor` type.
5. Compute the loss between the log probabilities and target contexts.
6. Do the `backward` pass over the neural network to update the gradients by calling `loss.backward()`.
7. Do one step using the optimizer, so that weights are updated using stochastic gradient descent.
8. Increment the total loss by this epoch's current loss.