accepted

Why wouldn't you do the same technique but using words instead of characters.

Word-based models are used just as often as character-based ones. See an example in [this question](https://stackoverflow.com/q/42064690/712995). But there several important differences between the two:

* Character-based model is more flexible and can learn rarely used words and punctuation. And [Andrej Karpathy's post](http://karpathy.github.io/2015/05/21/rnn-effectiveness/) shows how effective this model can be. But this is also a downside, because this model can produce complete nonsense sometimes.
* Character-based models have much smaller vocabulary, which makes it easier and faster to train. Since one-hot encoding and softmax loss are working perfectly, there's no need to complicate the model with embedding vectors and specially crafted loss functions (negative sampling, NCE, ...)
* Word-based models can't generate out-of-vocabulary (OOV) words, they are more complex and resource demanding. But they can learn syntactically and grammatically correct sentences and are more robust than character-based ones.

By the way, there are also subword models, which are somewhat in the middle. See ["Subword language modeling with neural networks"](http://www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf)by T. Mikolov at al.

Furthermore, is it possible to create a word prediction RNN but with somehow inputting words pretrained on word2vec, so that the RNN can understand their meaning?

Yes, the [example](https://stackoverflow.com/q/42064690/712995) I referred to above is exactly about this kind of model.

64down voteaccepted

There are some cases where LabelEncoder or DictVectorizor are useful, but these are quite limited in my opinion due to ordinality.

LabelEncoder can turn [dog,cat,dog,mouse,cat] into [1,2,1,3,2], but then the imposed ordinality means that the average of dog and mouse is cat. Still there are algorithms like decision trees and random forests that can work with categorical variables just fine and LabelEncoder can be used to store values using less disk space.

One-Hot-Encoding has a the advantage that the result is binary rather than ordinal and that everything sits in an orthogonal vector space. The disadvantage is that for high cardinality, the feature space can really blow up quickly and you start fighting with the curse of dimensionality. In these cases, I typically employ one-hot-encoding followed by PCA for dimensionality reduction. I find that the judicious combination of one-hot plus PCA can seldom be beat by other encoding schemes. PCA finds the linear overlap, so will naturally tend to group similar features into the same feature.

Hope this helps!

## LSTM word embedding and hidden layer size

It should be remembered that in all of the mathematics above we are dealing with vectors i.e. the input xtxt and ht−1ht−1 are not single valued scalars, but rather vectors of a certain length. Likewise, all the weights and bias values are matrices and vectors respectively. Now, you may be wondering, how do we represent words to input them to a neural network? The answer is word embedding. I’ve written about this extensively in previous tutorials, in particular [**Word2Vec word embedding tutorial in Python and TensorFlow**](http://adventuresinmachinelearning.com/word2vec-tutorial-tensorflow/) and [**A Word2Vec Keras tutorial**](http://adventuresinmachinelearning.com/word2vec-keras-tutorial/). Basically it involves taking a word and finding a vector representation of that word which captures some meaning of the word. In Word2Vec, this meaning is usually quantified by context – i.e. word vectors which are close together in vector space are those words which appear in sentences close to the same words.

The word vectors can be learnt separately, as in [**this tutorial**](http://adventuresinmachinelearning.com/gensim-word2vec-tutorial/), or they can be learnt during the training of your Keras LSTM network. In the example to follow, we’ll be setting up what is called an embedding layer, to convert each word into a meaningful word vector. We have to specify the size of the embedding layer – this is the length of the vector each word is represented by – this is usually in the region of between 100-500. In other words, if the embedding layer size is 250, each word will be represented by a 250-length vector i.e. [x1,x2,x3,…,x250x1,x2,x3,…,x250].

**LSTM hidden layer size**

We usually match up the size of the embedding layer output with the number of hidden layers in the LSTM cell. You might be wondering where the hidden layers in the LSTM cell come from. In my LSTM overview diagram, I simply showed “data rails” through which our input data flowed. However, each sigmoid, tanh or hidden state layer in the cell is actually a set of nodes, whose number is equal to the hidden layer size. Therefore each of the “nodes” in the LSTM cell is actually a cluster of normal neural network nodes, as in each layer of a [**densely connected neural network**](http://adventuresinmachinelearning.com/neural-networks-tutorial/).

<http://fbsight.com/t/rnn-classification/1435/3>

keras==2.1.5