Date Format Conversion with Encoder-Decoder

1. Introduction

The problem we are trying to solve is that of converting from one date format to another using an encoder-decoder neural network. More specifically, we will be focusing on conversion from “MMMM DD, YYYY” to “YYYY-MM-DD”; for example, from "April 22, 2019" to "2019-04-22".

The dataset was created by randomly sampling dates between 1000-01-01 and 9999-12-31. To feed it to a NN, the dates were treated as a sequence of characters, and each character was assigned a unique id. A dataset of dates was then converted to be represented using the ids and were zero-padded to match the longest date in the set.

While a regex can be used to easily solve this problem, we will use neural networks to build familiarity with them using a simple dataset.

2. Encoder-Decoder Model

We will use an encoder-decoder model for the problem. This architecture lends itself quite naturally to this problem as this is a sequence-to-sequence problem: taking in a sequence of character ids and producing another sequence of character ids.

We will use teacher forcing to train the model: we feed the decoder the target sequence, shifted by one time step to the right. This way, at each time step the decoder will know what the previous target character was. Since the first output character of each target sequence has no previous character, we will need a new token to represent the start-of-sequence (sos).

During inference, we won’t have the target sequence, instead we can just predict one character at a time, starting with a sos token, then feeding the decoder all the characters that were predicted so far.

The encoder’s internal state will be used to initialize the decoder’s internal state.

2.1 Model Architecture

The model architecture is as follows: it is an encoder model, followed by a decoder model. The encoder model consists of an Embedding layer of 32 followed by a single recurrent layer (e.g. LSTM, GRU, or RNN). The decoder model similarly consists of an Embedding layer followed by a single recurrent layer, which is then followed by a Dense layer with a softmax activation that outputs the probability for each of the character being the next one. Cross-entropy was used as the cost function.

The final hyperparameters used are:

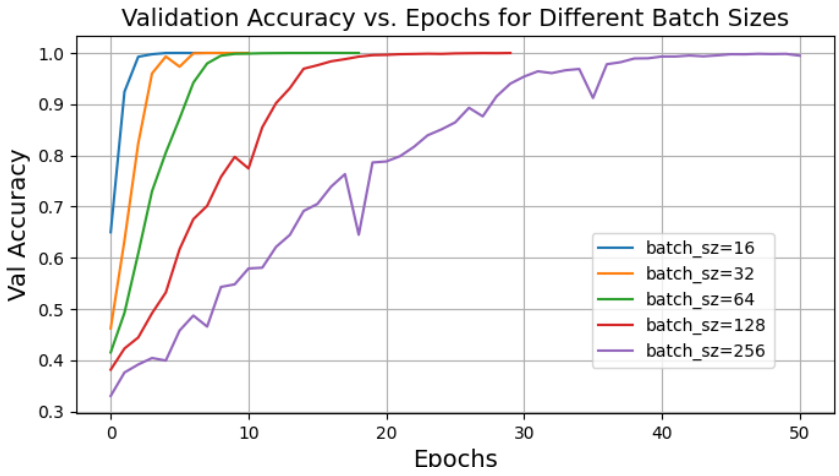
- Training set size: 10,000 examples. Validation set size: 2000. Test set size: 2000.
- The output dimensions for the Embedding layers in both the encoder and decoder is 32. To simplify search, the same dimension is used for both the layers.
- The recurrent layer type use was LSTM.
- The number of neurons in the layer was 128.
- The batch size used during training was 32.
- The Nadam optimizer was used with a learning rate of 1e-3.

3. Hyperparameter Selection

The convergence criterion used was early stopping with a patience of 3 while monitoring the validation accuracy.

3.1 Batch Size

The general trend was that using a smaller batch size led to faster convergence. This makes sense because with smaller batch size we are allowed to rapidly compute approximate estimates of the gradient and do more updates rather than slowly computing the exact gradient.



Recall that the gradient is the arithmetic mean of all the gradients of each individual example in the batch. With a larger batch size, we can compute a more accurate estimate of the gradient. However, by adding more examples to the batch, there is a less than liner return in the decreased standard deviation of the computed gradient. This is because the standard deviation of the gradient is given by:

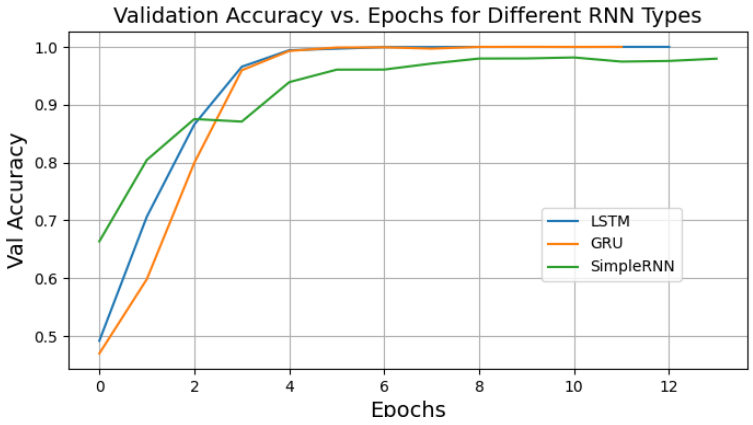
$$\frac{\sigma}{\sqrt{n}}$$

As we can see, as we add more examples to the batch, the standard deviation of the gradient only reduces by $\sqrt{n}$. Therefore, using larger batch sizes is not necessarily more advantageous and is in fact slower to converge.

Batch size of 32 was used for the final model as it strikes a good balance between speed of convergence and stability.

3.2 Embedding Size

The larger the embedding size used, the faster the convergence. This makes sense since with a larger embedding vector there is increased representation capacity, each character can be represented with much richer information. However, increasing the embedding size more than 32 lead to diminishing returns. This might be because the number of characters to represent is quite small, and the model does not need extra representational power.



3.3 Recurrent Layer Type

Three different recurrent layers were tried: RNN, GRU, and LSTM. Both LSTM and GRU had similar performance and converged around the same time, but GRU was much faster taking an average of 13s/epoch while LSTM took about 17s/epoch. This makes sense since GRUs are simplified versions of LSTMs with fewer gates. The RNN layer performed the works both in terms of generalization error (it never reached 100% validation accuracy) and convergence speed.