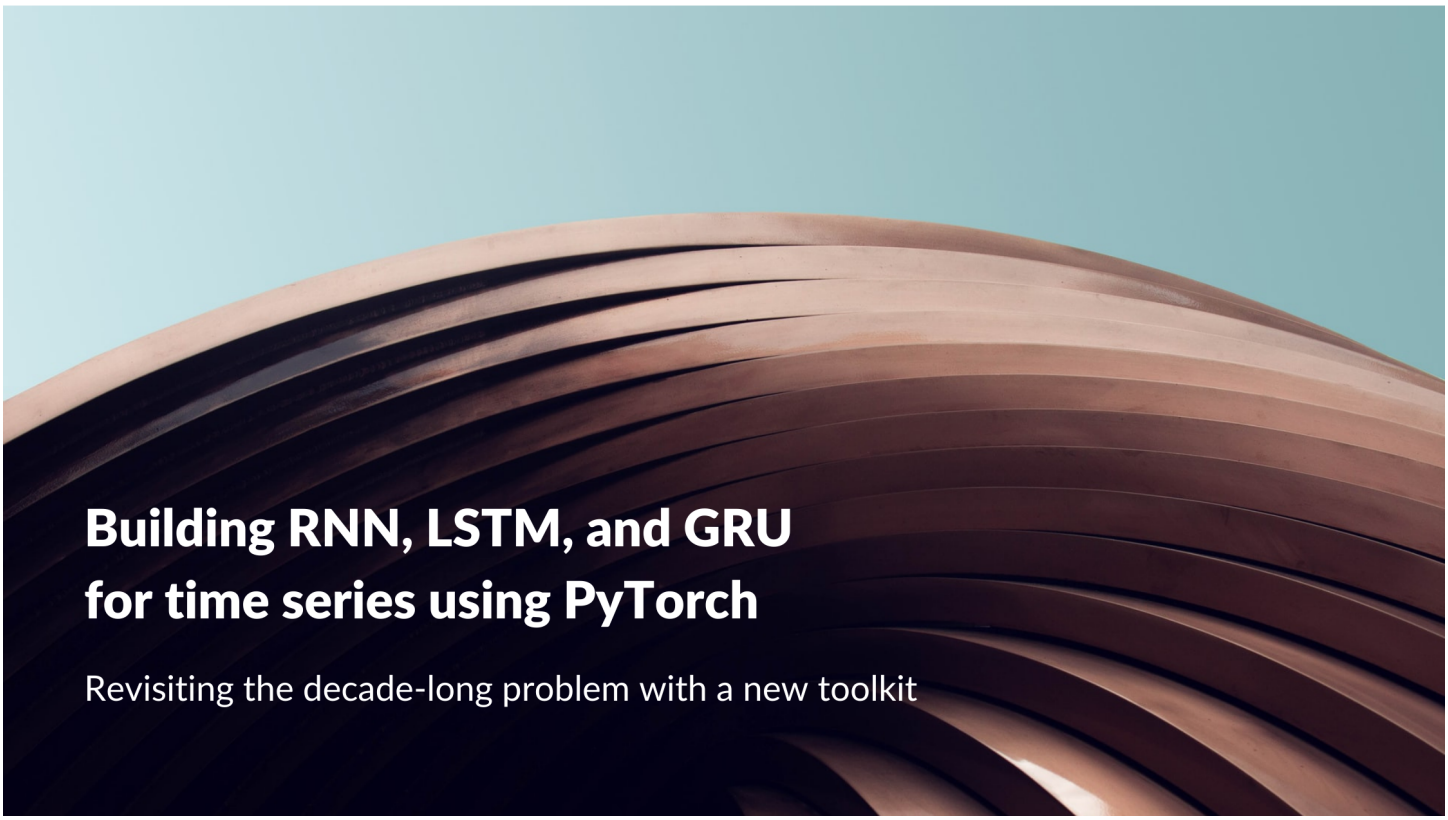

Photo by Nkululeko Jonas.

Building RNN, LSTM, and GRU for time series using PyTorch

Revisiting the decade-long problem with a new toolkit

Kaan Kuguoglu Apr 14, 2021 · 17 min read ★

Historically, time-series forecasting has been dominated by linear and ensemble methods since they are well-understood and highly effective on various problems when supported with feature engineering. Partly for this reason, Deep Learning has been

With the emergence of Recurrent Neural Networks (RNN) in the '80s, followed by more sophisticated RNN structures, namely Long-Short Term Memory (LSTM) in 1997 and, more recently, Gated Recurrent Unit (GRU) in 2014, Deep Learning techniques enabled learning complex relations between sequential inputs and outputs with limited feature engineering. In short, these RNN techniques and the like hold great potential for analyzing large-scale time series in ways that were not previously practical.

In this post, I'd like to give you a bit of an introduction to some of the RNN structures, such as RNN, LSTM, and GRU, and help you get started building your deep learning models for time-series forecasting. Though not the focus of this article, I'll provide some of the feature engineering techniques that are widely applied in time-series forecasting, such as one-hot encoding, lagging, and cyclical time features. I'll use Scikit-learn, Pandas, and PyTorch, an open-source machine learning library primarily developed by Facebook's AI Research lab. While the former two have long been a sweetheart of data scientists and machine learning practitioners, PyTorch is relatively new but steadily growing in popularity. Due to its recency, though, it has been somewhat difficult for me to find the relevant pieces of information and code samples from the get-go, which is usually a bit easier with frameworks that have been around for a while, say TensorFlow. So, I decided to put together the things I would have liked to know earlier. Less talk, more work: where do we start?

Where's the data?

Well, I suppose we need some time-series data to start with. Be it payment transactions or stock exchange data, time-series data is everywhere. One such public dataset is PJM's Hourly Energy Consumption data, a univariate time-series dataset of 10+ years of hourly observations collected from different US regions. I'll be using the PJM East region data, which originally has the hourly energy consumption data from 2001 to 2018, but any of the datasets provided in the link should work.

Given that there are heaps of blogs on data visualizations out there, I'll keep the exploratory data analysis (EDA) part very short. For those interested, I can recommend

Estimated energy consumption (MW) in the PJME region from 2012 to 2018

Estimated energy consumption (MW) in the PJME region from July 2017 to September 2017

The next step is to generate feature columns to transform our univariate dataset into a multivariate dataset. We will convert this time series into a supervised learning problem if you will. In some datasets, such features as hourly temperature, humidity, or precipitation, are readily available. However, in our data set, no extra information could help us predict the energy consumption is given. So, it falls to our lot to create such predictors, i.e., feature columns.

I'll show you two popular ways to generate features: passing lagged observations as features and creating date time features from the DateTime index. Both approaches have their advantages and disadvantages, and each may prove more useful depending on the task at hand.

Using lagged observations as features

Let's start with using time steps as features. In other words, we're trying to predict the next value, $X(t+n)$, from the previous n observations $Xt, X+1, \ldots,$ and $X(t+n-1)$. Then, what we need to do is simply create n columns with the preceding observations. Luckily, Pandas provides the method *shift()* to shift the values in a column. So, we can write a for loop to create such lagged observations by shifting the values in a column by n times and removing the first n columns. Lagging is simple yet a good starting point, especially if you don't have many features to work with at the start.

After setting the number of input features, i.e., lagged observations, to 100, we get the following DataFrame with 101 columns, one for the actual value and the rest for the preceding 100 observations at each row.

Despite its name, feature engineering is generally more art than science. Nonetheless, some rules of thumb can guide data scientists and the like. My goal in this section is not to go through all such practices here but to demonstrate a couple of them and let you experiment on your own. In effect, feature engineering is very much dependent on the domain you're working in, possibly requiring the creation of a different set of features for the task at hand.

Having a univariate time-series dataset, it seems logical to generate date and time features. As we have already converted its index into Pandas' DatetimeIndex type, a series of DateTime objects, we can easily create new features from the index values, like the hour of the day, the day of the month, the month, the day of the week and the week of the year, as follows.

Although passing date and time features to the model without any touch may work in practice, it would be harder for the model to learn interdependencies between these features. For us humans, it is rather straightforward to see the hours, days, weeks, and months follow somewhat cyclical patterns. While it is trivial for us to say that December is followed by January, it may not be apparent that the algorithms to understand the first month of the year come after the 12th one. One can easily come up with many more examples, for that matter. This makes good feature engineering crucial for building deep learning models, even more so for traditional machine learning models.

One-hot Encoding

One way to encode DateTime features is to treat them as categorical variables and add a new binary variable for each unique value, widely known as one-hot encoding. Suppose you applied one-hot encoding on your month column, which ranges from 1 to 12. In this case, 12 new month columns are created, say [Jan, Feb, … Dec], and only one of such columns has the value 1 while the rest is zeroed. For instance, some DateTime values from February should have the second of these encoded columns as 1, as in *[0, 1, … 0]*.

Using Pandas' get_dummies method, we can quickly create one-hot encoded columns from a given dataset.

Alternatively, you may wish to use Scikit-learn's OneHotEncoder to encode a column in your DataFrame using the ColumnTransformer class. Unlike the Pandas way, ColumnTranformer outputs a Numpy array when it is called to fit the DataFrame.

Though quite useful to encode categorical features, one-hot encoding does not fully capture the cyclical patterns in DateTime features. It simply creates categorical buckets, if you will, and lets the model learn from these seemingly independent features. Encoding the day of the week similarly, for instance, loses the information that Monday is closer to Tuesday than Wednesday.

For some use cases, this may not matter too much, indeed. In fact, with enough data, training time, and model complexity, the model may learn such relationships between such features independently. But there is also another way.

Generating cyclical time features

As with all the data we have worked on until now, some data is inherently cyclical. Be it hours, days, weeks, or months, they all follow periodic cycles. Again, this is trivial for us to see, but not so much for machine learning models. How can we tell algorithms that hours 23 and 0 are as close as hour 1 is to hour 2?

The gist is to create two new cyclical features, calculating sine and cosine transform of the given DateTime feature, say the hour of the day. Instead of using the hour's original value, the model then uses its sine transform, preserving its cyclicality. To see how and why it works, feel free to refer to Pierre-Louis'or David's blog post on the matter, which explains the concept more in detail.

two would perform in decision tree-based models (Random Forest, Gradient Boosted Trees, and XGBoost). These features form their splits according to one feature at a time, meaning that it will fail to use two-time features, e.g., a sine and cosine transform, simultaneously. Usually, these models are robust enough to handle such splits, but it's certainly food for thought.

What about other features?

Considering we are now working with the energy consumption data, one might ask whether holidays in a year affect the energy consumption patterns. Indeed, very likely. For such binary variables, i.e., 0 or 1, we can generate extra columns with binary values to denote if a given date is actually a holiday. Remembering all the holidays or manually defining them is a tedious task, to say the least. Fortunately, a package, called holidays, does what it promises to do.

In feature engineering, possibilities are seemingly limitless, and there is certainly room for some experimentation and creativity. The good news is there are already quite a few packages out there that do the job for us, like *meteostat* for historical weather data or *yfinance* for stock market data. The bad one is, there is generally no clear answer to which additional features may improve the model performance without actually trying them out.

Just an idea, one can also try to include weather data, e.g., temperature, humidity, precipitation, wind, rain, snow, and so on, to learn how the weather affects the energy consumption in a given hour, day, week and month.

Splitting the data into train, validation, and test sets

After creating feature columns, be it time-lagged observations or date/time features, we split the dataset into three different datasets: training, validation, and test sets. Since we're dealing with time-dependent data, it is crucial to keep the time sequences intact, unshuffled if you will. You can easily do so by setting the parameter *shuffle* to *false,* avoiding shuffling while splitting into sets.

Scaling the values in your dataset is a highly recommended practice for neural networks, as it is for other machine learning techniques. It speeds up the learning by making it easier for the model to update the weights. You can easily do that by using Scikit-learn's scalers, MinMaxScaler, RobustScaler, Standard Scaler, and so on. For more information on the effects of each scaler, please refer to the official documentation.

And, here is a cool trick if you're looking for a way to switch between scalers quickly. Get yourself comfortable with the switcher function; we may use it again later on.

Loading the datasets into DataLoaders

After you standardize your data, you are usually good to go. Not so fast, this time. After spending quite some time working with PyTorch and going through others' code on the internet, I noticed most people ended up doing the matrix operations for mini-batch training, i.e., slicing the data into smaller batches, using NumPy. You may think that's what NumPy is for; I get it. But there is also a more elegant PyTorch way of doing it, which certainly gets much less attention than it should, in my opinion.

PyTorch's DataLoader class, a Python iterable over Dataset, loads the data and splits them into batches for you to do mini-batch training. The most important argument for the DataLoader constructor is the Dataset, which indicates a dataset object to load data from. There are mainly two types of datasets: map-style datasets and iterable-style datasets.

I'll use the latter in this tutorial, but feel free to check them out in the official documentation. It is also possible to write your own Dataset or DataLoader classes for your requirements, but that's definitely beyond the scope of this post, as the built-in constructors would do more than suffice. But here's a link to the official tutorial on the topic.

For now, I'll be using the class called TensorDataset, a dataset class wrapping the tensors. Since Scikit-learn's scalers output NumPy arrays, I need to convert them into Torch tensors to load them into TensorDatasets. After creating Tensor datasets for each data set, I'll use them to create my DataLoaders.

You may notice an extra DataLoader with the batch size of 1 and wonder why the hell we need it. The short answer, it's not a must-have but a nice-to-have. Like mini-batch training, you can also do mini-batch testing to evaluate the model's performance, which is likely much faster due to tensor operations. However, doing so will drop the last time steps that couldn't make up a batch, leading to losing these data points. This will not likely result in a significant change in your error metrics unless your batch size is huge.

From a more aesthetical standpoint, if one is interested in making predictions into the future, discarding the last time steps may cause a discontinuity from the test set to forecasted values. As for the training and validation DataLoaders, this effect can be tolerated due to significant performance improvement batching provides in training and such dropped time steps being less visible.

Building Recurrent Neural Networks (RNN)

I don't think I can ever do justice to RNNs if I try to explain the nitty-gritty of how they work in just a few sentences here. Fortunately, there are several well-written articles on

LSTM networks, and Michael Phi's Illustrated Guide to LSTM's and GRU's: A step by step explanation are a few that come to mind.

If you've watched the movie Memento -definitely a great watch btw- you may already have an idea about how hard it could be to make predictions with memory loss. While your memory is being reset every couple of minutes, it can easily become impossible to tell what is happening, where you are going, or why -let alone tracking down your wife's murderer. The same can also be said for working with sequential data, be it words or retail sales data. Generally speaking, having the preceding data points helps you understand patterns, build a complete picture and make better predictions.

However, traditional neural networks can't do this, and they start from scratch every time they are given a task, pretty much like Leonard, you see. RNN addresses this shortcoming. To make a gross oversimplification, they do so by looping the information from one step of the network to the next, allowing information to persist within the network. This makes them a pretty strong candidate to solve various problems involving sequential data, such as speech recognition, language translation, or time-series forecasting, as we will see in a bit.

Vanilla RNN

By extending PyTorch's nn.Module, a base class for all neural network modules, we define our RNN module as follows. Our RNN module will have one or more RNN layers connected by a fully connected layer to convert the RNN output into desired output shape. We also need to define the forward propagation function as a class method, called *forward()*. This method is executed sequentially, passing the inputs and the zero-initialized hidden state. Nonetheless, PyTorch automatically creates and computes the backpropagation function *backward()*.

Vanilla RNN has one shortcoming, though. Simple RNNs can connect previous information to the current one, where the temporal gap between the relevant past information and the current one is small. As that gap grows, RNNs become less capable of learning the long-term dependencies. This is where LSTM comes for help.

Long Short-Term Memory (LSTM)

Long Short-Term Memory, LSTM for short, is a special type of recurrent network capable of learning long-term dependencies and tends to work much better than the standard version on a wide variety of tasks. RNNs on steroids, so to speak.

The standard version's main difference is that, in addition to the hidden state, LSTMs have the cell state, which works like a conveyor belt that carries the relevant information from the earlier steps to later steps. Along the way, the new information is added to or removed from the cell state via input and forget gates, two neural networks that determine which information is relevant.

From the implementation standpoint, you don't really have to bother with such details. All you need to add is a cell state in your *forward()* method.

Gated Recurrent Unit (GRU)

Gated Recurrent Units (GRU) is a slightly more streamlined variant that provides comparable performance and considerably faster computation. Like LSTMs, they also capture long-term dependencies, but they do so by using reset and update gates without any cell state.

While the update gate determines how much of the past information needs to be kept, the reset gate decides how much of the past information to forget. Doing fewer tensor operations, GRUs are often faster and require less memory than LSTMs. As you see below, its model class is almost identical to the RNN's.

Similar to the trick we do with scalers, we can also easily switch between these models we just created.

Now, it seems like we got everything ready to train our RNN models. But where do we start?

Training the model

Let's start by creating the main framework for training the models. There are probably heaps of ways to do this, and one of them is to use a helper, or a wrapper, class that holds the training, validation, and evaluation methods. First, we need to have a model class, a loss function to calculate the losses, and an optimizer to update the network's weights.

If you're familiar with neural networks, you already know that training them is a rather repetitive process, looping back and forth between forward-prop and back-prop. I find it useful to have one level of abstraction, a train step function or wrapper, to combine these repetitive steps.

After defining one proper training step, we can now move onto writing the training loop where this step function will be called at each epoch. During each epoch in training, there are two stages: training and validation. After each training step, the network's weights are tweaked a bit to minimize the loss function. Then, the validation step will evaluate the current state of the model to see if there has been any improvement after the most recent update.

I'll be using mini-batch training, a training technique where only a portion of data is used at each epoch. Given a large enough batch size, the model can learn and update its weights more efficiently by only learning a sample of the data. This usually requires reshaping each batch tensor into the correct input dimensions so that the network can use it as an input. In order to reap the computational benefits of tensor operations, I defined our RNN models to operate with 3D input tensors earlier, unless you haven't noticed already. So, you may think of each batch as packages of data, like boxes in a warehouse, with dimensions of *batch size, sequence length, and input_dim*.

There are also two for loops for each stage where a model is trained and validated batch by batch. It is important to activate the *train()* mode during training and the *eval()* mode during the validation. While the *train()* mode allows the network's weights to be updated, the *eval()* mode signals the model that there is no need to calculate the gradients. Hence, the weights get updated or stay the same depending on the operation.

Now, we can finally train our model. However, without evaluating these models with a separate test set, i.e., a hold-out set, it would be impossible to tell how the model performs compared to other models we're building. Much similar to the validation loop in the *train()* method, we'll define a testing method to evaluate our models as follows.

During the training, the loss function outputs are generally a good indicator of whether the model is learning, overfitting, or underfitting. For this reason, we'll be plotting simple loss figures by using the following method.

Training

So far, we have prepared our dataset, defined our model classes and the wrapper class. We need to put all of them together. You can find some of the hyperparameters defined in the code snippet below, and I encourage you to play with them as you wish. The code below will build an LSTM model using the module we defined earlier. You can also build RNN or GRU models by quickly changing the input of the function *get_model* from "*lstm*" to a model of your choice. Without further ado, let's start training our model.

As you may recall, we trained our network with standardized inputs; therefore, all the model's predictions are also scaled. Also, after using batching in our evaluation method, all of our predictions are now in batches. To calculate error metrics and plot these predictions, we need first to reduce these multi-dimensional tensors to a one-dimensional vector, i.e., flatten, and then apply *inverse_transform()* to get the predictions' real values.

After flattening and de-scaling the values, we can now calculate error metrics, such as mean absolute error (MAE), mean squared error (MSE), and root mean squared error (RMSE).

0.64 R2-score… Not great, not terrible. As you can also see in the next section, there is room for improvement, which you can achieve by engineering better features and trying out different hyper-parameters. I'll leave that challenge to you.

Generating baseline predictions

Having some sort of baseline model helps us compare how our models actually do at prediction. For this task, I've chosen good old linear regression, good enough to generate a reasonable baseline but simple enough to do it fast.

Visualizing the predictions

Last but not least, visualizing your results helps you better understand how your model performs and adds what features would likely improve it. I'll be using Plotly again, but feel free to use a package that you are more comfortable with.

Final words

I'd like to say that was all, but there's and will be certainly more. Deep learning has been one of if not, the most fruitful, research areas in machine learning. The research on the sequential deep learning models is growing and will likely keep growing in the future. You may consider this post as the first step into exploring what these techniques have to offer for time series forecasting. For those who enjoyed this one and are interested in predicting future values using these algorithms, I have recently published another article on the topic.

Here's a link to Google Colab notebook for this post, if you'd like to have a look at the complete notebook and play with it. If there are things that don't add up or you disagree, please reach out to me or let me know in the comments. That also goes for all sorts of feedback you may have.

There are still a few more topics that I'd like to write about, like forecasting into the future time steps using time-lagged and DateTime features, regularization techniques, some of which we have already used in this post, and more advanced deep learning architectures for time series. And, the list goes on. Let's hope my motivation lives up to such ambitions.

But, for now, that's a wrap.