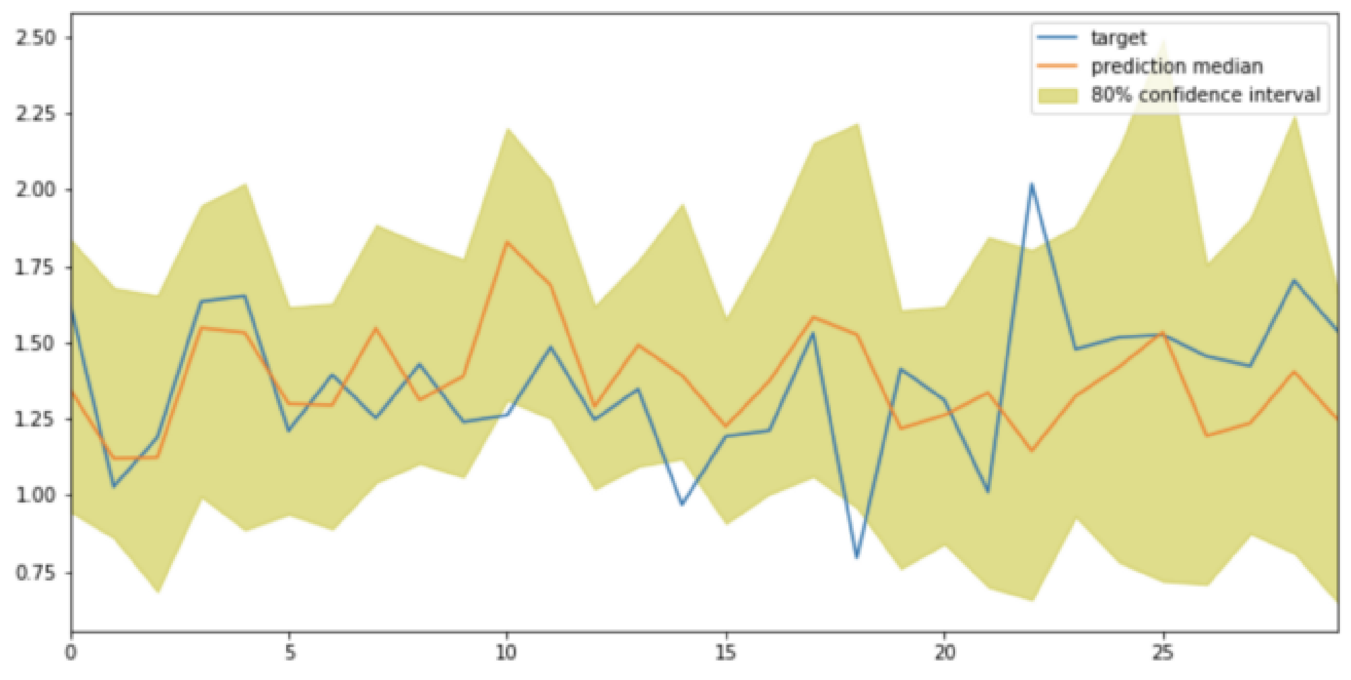
Time series (TS) forecasting is notoriously finicky. That is, until now.



png

Figure 1: DeepAR trained output based on this tutorial. Image by author.

In 2019, Amazon's research team developed a deep learning method called DeepAR that exhibits a ~15% accuracy boost relative to state-of-the-art TS forecasting models. It's robust out-of-the-box and can learn from many different time series', so if you have lots of choppy data, DeepAR could be an effective solution.

From an implementation perspective, DeepAR is more computationally complex than other TS methods. It also requires more data than traditional TS forecasting methods such as ARIMA or Facebook's prophet.

That being said, if you have lots of complex data and need a very accurate forecast, DeepAR is arguably the most robust solution.

Technical TLDR

In short, DeepAR is an LSTM RNN with some bells and whistles to improve accuracy on complex data. There are 4 main advantages to DeepAR relative to traditional TS forecasting methods…

1. **DeepAR is effective at learning seasonal dependencies with minimal tuning.** This out-of-the-box performance makes the model a good jumping-off point for TS forecasting.
2. **DeepAR can use covariates with little training history.** By leveraging *similar* observations and weighted resampling techniques, the model can effectively determine how an infrequent covariate would behave.
3. **DeepAR makes probabilistic forecasts.** These probabilities, in the form of Monte Carlo samples, can be used to develop quantile forecasts.
4. **DeepAR supports a wide range of likelihood functions.** If your dependent variable takes on a non-normal or non-continuous distribution, you can specify the relevant likelihood function.

The model has received lots of attention and is currently supported in PyTorch. Tutorials and code are linked in the comments.

But, what's actually going on?

Ok let's slow down a bit and discuss how Amazon's DeepAR model actually works…

Traditional Time Series Forecasting

Let's start at square one.

As noted above, time series forecasting is notoriously difficult for two main reasons. The first is that most time series models require lots of subject matter knowledge. If you're modeling a stock price with a traditional TS model, it's important to know what covariates impact price, whether there's a delay in the impact of those covariates, if price exhibits seasonal trends, etc.

Often engineers lack the subject matter knowledge required to create effective features.

The second reason is that TS modeling is a pretty niche skillset. Prior outputs of a given timestep are the inputs to the next timestep, so we can't use the usual modeling techniques or evaluation criteria.

So, not only do engineers need in-depth knowledge of the data, they also must have a strong understanding of TS modeling techniques.

Traditional Recurrent Neural Nets (RNNs)

Machine learning can provide alternatives to traditional TS forecasts that are often more accurate and easier to build. The simplest ML algorithm that supports sequential data are recurrent neural nets.

RNNs are essentially a bunch of neural nets stacked on top of each other. The output of the model at h*1* feeds into the next model at *h2* as shown in figure 2.

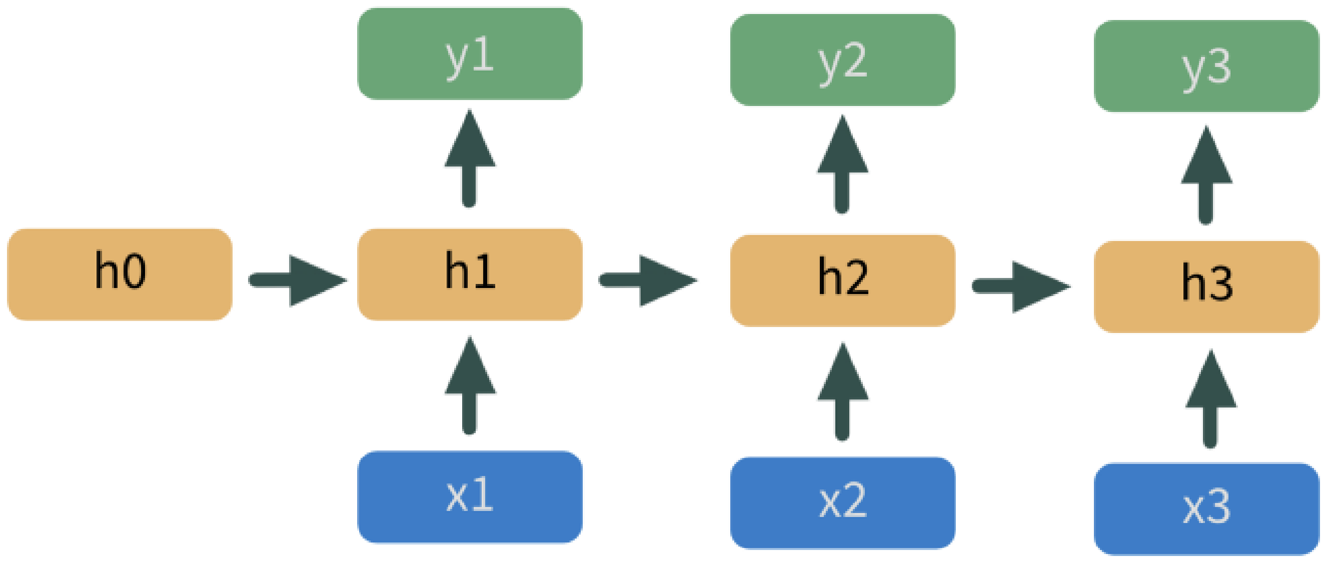


Figure 2: graphical representation of a recurrent neural net. Image by author.

Here, *x*'s in blue are predictor variables, *h*'s in yellow are hidden layers, and *y*'s in green are predicted values. This model architecture automatically handles many of the challenges we discussed above.

But even RNNs fall short in certain areas. First, they're overly simplistic in their assumptions about what should be passed to the next hidden layer. More advanced components of a RNN such as Long Short Term Memory (LSTM) and Gate Recurring Units (GRU) layers provide filters for what information get's passed down the chain. And they often provide better forecasts than vanilla RNNs, but sadly they can fall short too.

Amazon's DeepAR model is the most recent iteration on LSTM and solves two of its key shortcomings:

1. **Outliers are fit poorly due to a uniform sampling distribution.** One of the strengths of DeepAR is that it aggregates information from many time series' to develop a

prediction for a single unit e.g. a user. If all units have an equal chance of being sampled, outliers are smoothed over and our forecasted values become less extreme (and probably less accurate).

2. **RNN's don't handle temporal scaling well.** Over time, time series data often exhibit overall trends — just think about your favorite (or least favorite) stock during COVID. These temporal trends make fitting our model more difficult because it has to remove this trend when fitting, then add it back to our model output. That's a lot of unnecessary work.

DeepAR solves both of these problems.

How does DeepAR Work?

Building on RNN architecture, DeepAR uses LSTM cells to fit our predictor variables to our variable of interest. Here's how…

Sequence to Sequence Encoder-Decoder

First, to make our data more usable we leverage a sequence to sequence encoder-decoder. This method takes a set of n inputs, encodes those inputs with a neural net, then outputs m decoded values.

Fun fact — they're the backbone of all language translation algorithms, such as Google Translate. So, using figure 3 below, let's think about this method through the lens of translating english to Spanish.

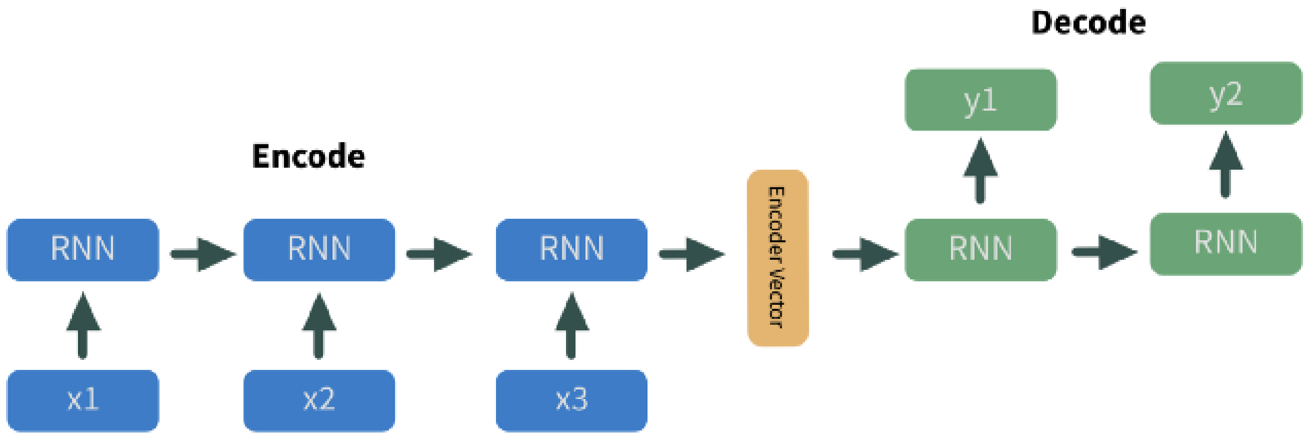


Figure 3: graphical representation of a sequence to sequence encoder-decoder. Image by author.

Here, each of the *x_n* values in blue are inputs to our model i.e. english words. They are sequentially fed into an RNN which encodes their information and outputs it as an encoder vector (in yellow). The information the encoder vector is represented as a weight vector for our hidden state. From there, our encoder vector is fed into our decoder, which is a RNN. The final output, labeled *y_n* in green, are the Spanish words.

Pretty cool, right?

Scale According to our Data

Second, we tackle the two problems that make basic LSTMs inferior to DeepAR: uniform sampling and temporal scaling.

Both of these are handled using a scaling factor. The scaling factor is a user-specified hyperparameter but the recommended value is simply the average value of a given time series:

$$\nu_i = 1 + \frac{1}{t0} \sum_{t=1}^{t0} z_{i,t}$$

Figure 4: formula for the default scaling factor used in DeepAR — the average value of an individual time series. Image by author.

To handle the uneven sampling of extreme values, we make the sampling probability proportional to *v_i*. So, if the average value for a given time series is high, it's more likely to be sampled and visa versa.

To handle temporal scaling, we divide the inputs at each LSTM cell by *v_i*. By scaling our values down before inputting them into each cell, our model can focus on the relationships between our predictor and outcome variables instead of fitting the time trend. After the cell has fit our data we multiply its output by *v_i* to return our data to its original scale.

Less cool but still pretty cool. Just one more section to go…

Fit using a Likelihood Function

Third and finally, with all of our data intricacies handled, we fit by maximizing the conditional probability of our dependent variable given our predictors and model parameters (figure 5). This estimator is called the Maximum Likelihood Estimator (MLE).

$$P(z_t|z_{t-1}, x_t, \theta)$$

Figure 5: simplified DeepAR likelihood function. Image by author.

The simplified expression above is what we are looking to maximize — we want to find the model parameters (θ) that maximize the probability of our dependent variable (z). We also condition on our covariates (*x_t*) and the output of our prior node (*z_t-1*).

So, given our covariates and the predicted value at the prior timestep, we find the parameter values that maximize the likelihood of making the observations given the parameters.

Now the likelihood also includes a data transformation step, but from a comprehension and implementation standpoint, it's not super important. If you're curious, check out the paper linked in the comments.

And there you have it — DeepAR in a nutshell. Before you go here are some practical tips on implementing the method…

Implementation Notes

- The authors suggest standardizing covariates i.e. subtract the mean and divide by the standard deviation.
- Missing data can be a big problem for DeepAR. The author suggest imputing the missing data by sampling from the conditional predictive distribution.
- For the example cited in the paper, the authors created covariates that correspond to temporal information. Some examples were *age* (in days) and *day-of-week*.
- Optimizing parameters using a grid search is an effective way to tune the model. However, learning rate and encoder/decoder length are subject-specific and should be tuned manually.
- To ensure the model is not fitting based on the index of our dependent variable in the TS, the authors suggest training the model on “empty” data prior our start period. Just use a bunch of zeros as our dependent variable.