TensorFlow

NOTES FROM 3 SPECIALIZATIONS FROM COURSERA

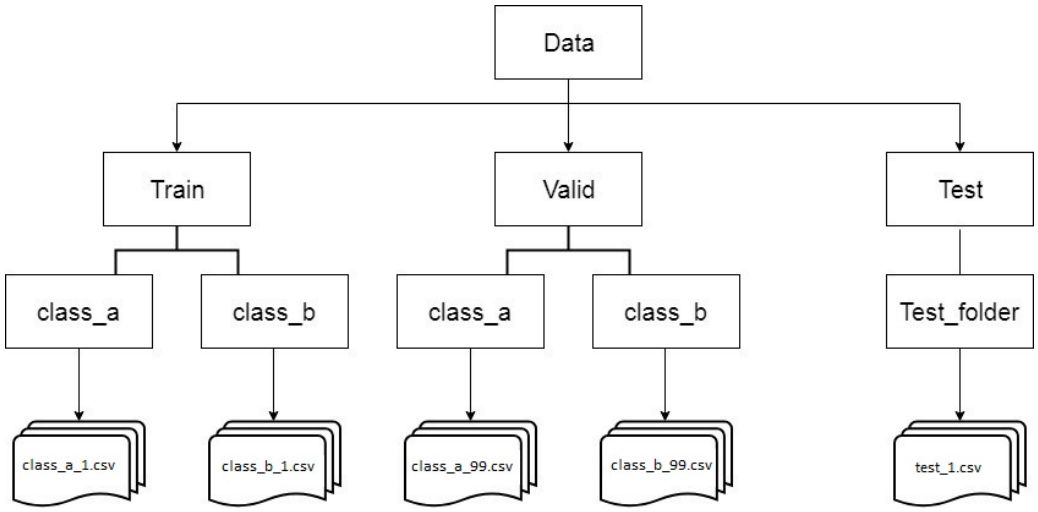
1 | TensorFlow: Developer Professional Certificate
2 | TensorFlow: Advanced Techniques Specialization
3 | TensorFlow: Data and Deployment Specialization

1. Introduction to TensorFlow for AI:

1. Convolution layer in TensorFlow (Conv2D) expects all the data at once. E.g. if you have 1000 images of 28*28*1 gray scale images to train the model. Then to pass it to the convolution layer we need to reshape it to 1000*28*28*1. We have made a batch of 1000 28*28*1 images. Now the convolution layer takes all the images and trains on it. Same goes with the test images. If you don't do this, you'll get an error when training as the Convolutions do not recognize the shape. (This is because TensorFlow works in batches, so always try to have batches of data to get fast result in TensorFlow).
2. Pooling just creates magic. The features remains same but the size of the image decreases up to 25%. Check out this notebook.
3. There are a few rules about the filter:
    a. Its size has to be odd, so that it has a center, for example 3x3, 5x5 and 7x7 are ok.
    b. It doesn't have to, but the sum of all elements of the filter should be 1 if you want the resulting image to have the same brightness as the original.
    c. If the sum of the elements is larger than 1, the result will be a brighter image, and if it's smaller than 1, a darker image. If the sum is 0, the resulting image isn't necessarily completely black, but it'll be very dark.

IMAGE/DATA GENERATOR API IN KERAS:

One feature of the image generator is that you can point it at a directory and then the sub-directories of that will automatically generate labels for you. So for example if we point to the 'Train' directory in the image, the labels will be class a, class b and all of the images in each directory will be loaded and labeled accordingly. Similarly if we point one at the validation directory, the same thing will happen.



The code for Image generator is:

```
from tensorflow.keras.preprocessing.image
import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255)
train_generator = train_datagen.flow_from_directory(
                      train_dir,
                      target_size = (300, 300)
                      bactch_size = 128,
                      class_mode = 'binary'
```

After importing it we create an object of ImageDataGenerator i.e. train_datagen. The rescale parameter is to normalize the images on fly (so that every image that it reads from the disk is normalized automatically). The 'train_dir' is the directory path where training data is present in our example we need to give the address of 'Train' directory. 'target_size' is the size that the image must be converted when loading from the directory (again helps to do the job on fly, without writing more code for transformations). 'batch_size' is the amount of examples to consider in one go (it improves the speed rather than taking one at a time), 'class_mode' is the parameter which signifies whether it is a binary classification or something else.

Same goes the validation set, the only change in the code will be the directory to the validation set.

Next after defining a model, we need to train it. Instead of using `model.fit` we use `model.fit_generator`

```
history = model.fit_generator(
          train_generator,
          steps_per_epoch = 8,
          epochs = 15,
          validation_data = validation_generator,
          validation_steps =8,
          verbose = 2)
```

The first parameter is the `train_generator` that we setup earlier. This streams the images from the training directory. There are 1,024 images in the training directory, so we're loading them in 128 at a time. So in order to load them all, we need to do 8 batches. So we set `the steps_per_epoch` to cover that (1024/128 = 8). `epochs` is just the normal epochs as used in fit function. The next parameter is `validation_data` which needs to be pointed to the validation directory. And now we specify the `validation_steps`, we have 256 images in validation set, and we wanted to handle them in batches of 32, so we will do 8 steps (256/32 = 8). The `verbose` parameter specifies how much to display while training is going on.

4. If we compact (decrease) the size of the image to the model, the training time may decrease, but the accuracy might also go down.

2. CNN in TensorFlow:

- Data Augmentation is a great technique to increase the performance of the model on validation dataset and decrease overfitting in the model. The image augmentation introduces a random element to the training images but if the validation set doesn't have the same randomness, then its results can fluctuate and might not get better results in the validation data set. So bear in mind that you don't just need a broad set of images for training, but you also need them for testing or the image augmentation won't help you very much.

  Image augmentation in TensorFlow with keras API can be done using the same class as used for generating images from the disk i.e. `ImageDataGenerator.`

  For example:

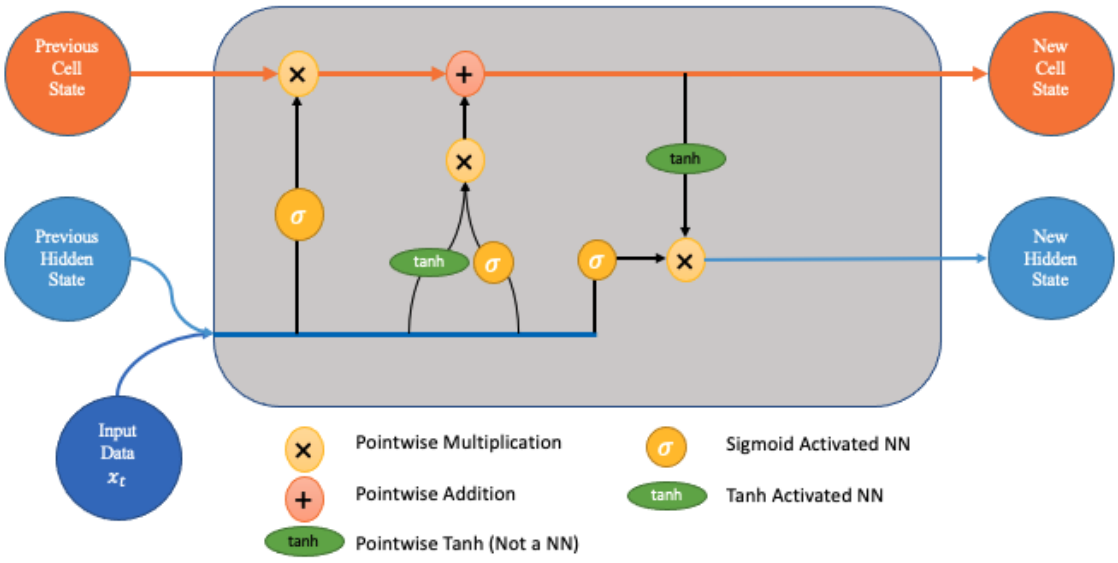
```
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

- Transfer learning using keras. See the code here.
- Dropout: The idea behind Dropouts is that they remove a random number of neurons in your neural network. This works very well for two reasons: The first is that neighboring neurons often end up with similar weights, which can lead to overfitting, so dropping some out at random can remove this. The second is that often a neuron can over-weigh the input from a neuron in the previous layer, and can over specialize as a result. Thus, dropping out can break the neural network out of this potential bad habit!
- Moving from binary class to multi class classification:
    - Change the class mode in generator from `binary` to `categorical`
    - In the output layer we need to change the `number of neurons` and activation function from `sigmoid` to `softmax`.
    - The loss parameter while compiling the model must be changed from `binary_crossentropy` to `categorical_crossentropy`

3. NLP in TensorFlow:

- Tokenizing the unique words in a collection of sentences and creating a vocabulary. Converting the sentences into sequences of tokens. Padding the sentence to have same size of the sentence. Check out the code in this notebook.
- Embedding: Words and associated words are clustered as vectors in a multi-dimensional space.

- There is a library in TensorFlow called TensorFlow Data Services or TFDS for short and it contains many data sets and lots of different categories.
- Subword Text encoder: Instead of tokenizing every word in the sentence, we tokenize multiple subparts of the word. By doing this we increase the size of the vocabulary and also the accuracy goes down because we need to give the model something so that it can remember sequence. Therefore RNN's are introduced.
- LSTM (Long Short Term Memory) is used to store information for long time and therefore this can act as a solution for the above problem. LSTM has its own cell state which can be both unidirectional and bidirectional.



Single Layer LSTM code:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

In the following code `tf.keras.layers.Embedding` is the layer which embeds the all the unique tokenized words into multi-dimensional vectors. The two arguments given to this layer is `input_dim` (size of the vocabulary) and `output_dim` (number of dimensions the vectors should

be created in).
`tf.keras.layers.Bidirectional` is used to create a bidirectional layer, since we know that LSTM has cell state which is bidirectional. Inside this layer we create a single layer of LSTM with 64 units which defines the number of outputs we need from the LSTM layer. (Since we are using bidirectional layer therefore the number of units when the model gets built gets doubled i.e. 2 * 64 = 128).

Multi-Layer LSTM code:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,
return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32))
,
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

We can also stack LSTMs like any other keras layer by using code shown above. But when we feed an LSTM into another one, you do have to put the `return_sequences` equal true parameter into the first LSTM layer. This ensures that the outputs of the LSTM match the desired inputs of the next one.

It is seen that more the LSTM layers in the sequence the more will be accuracy and better will be the model.
LSTM models mostly overfit on the dataset due to the reason that in the validation dataset there are more number of <oov> tokens due to which the model overfits.

4. Sequences, Time Series and Prediction:

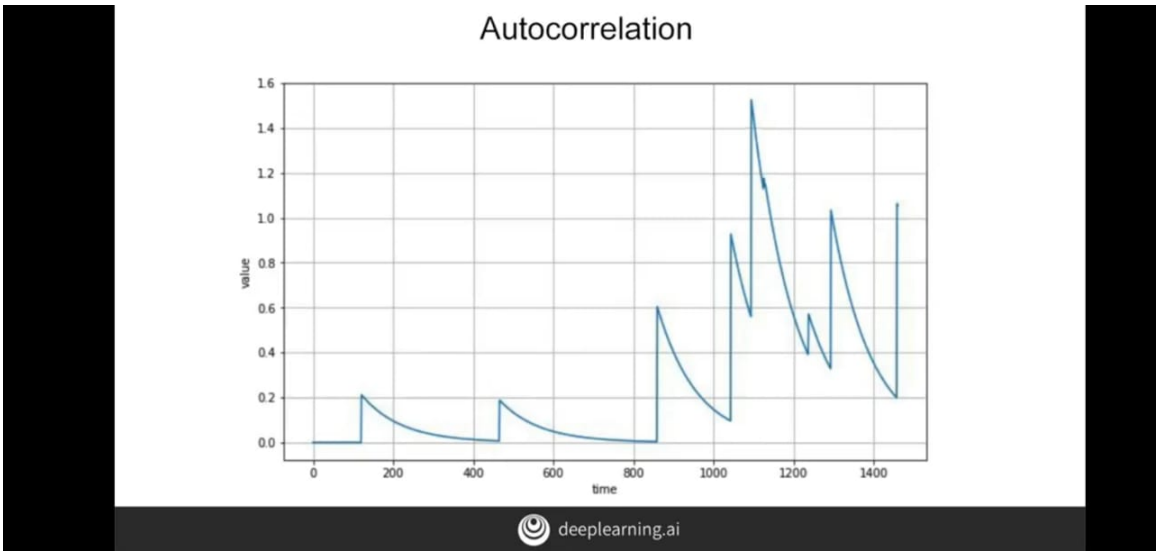
- Just about anything that has a time factor in it can be analyzed using time series. The first and most obvious is prediction of forecasting based on the data. The second is project back into the past to see how we got to where

we are now. This process is called imputation. We might just fill holes in the data for what data doesn't exist, with imputation we can fill these holes. Time series prediction can also be used to detect anomalies. The other option is to analyze the time series to spot patterns in them that determine what generated the series itself. A classic example of this is to analyze sound waves to spot words in them which can be used as a neural network for speech recognition.

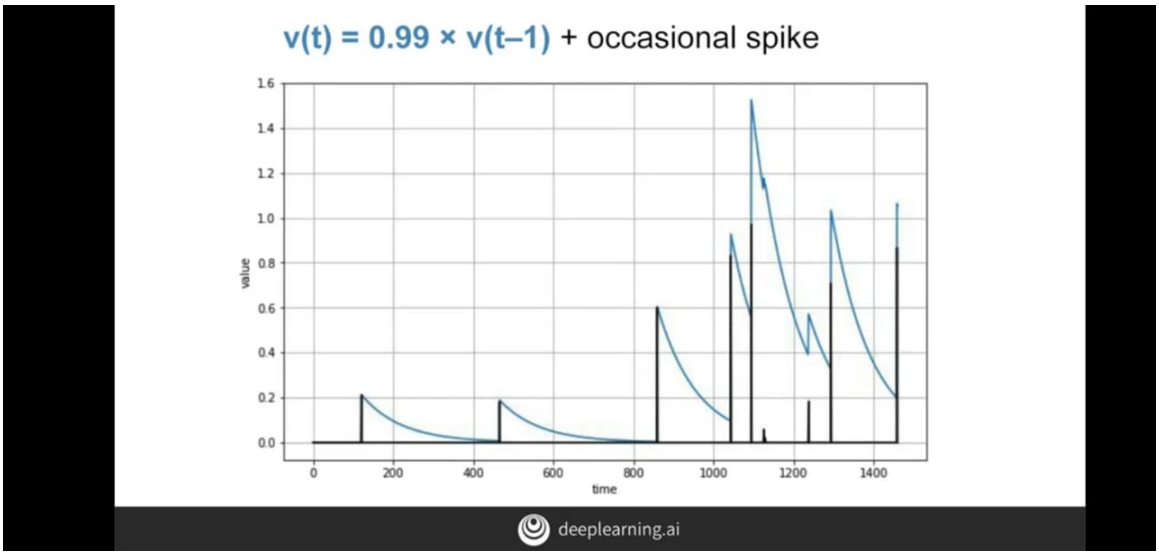
- Types of Time Series:
    - Trend: Time series have a specific direction that they're moving.
    - Seasonality: When patterns repeat at predictable intervals.

    There are also some scenarios where we see both trend and seasonality at once.

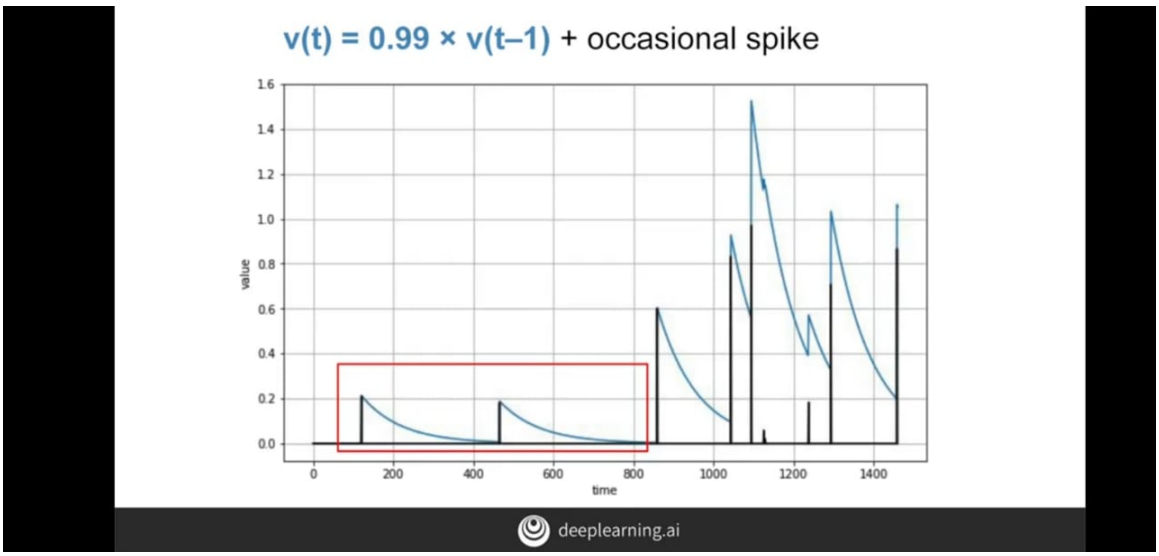
    - White Noise: Time series which are not predictable at all and just a complete set of random values. There's not a whole lot you can do with this type of data.
    - Consider the given time series:



There's no trend and there's no seasonality. The spikes appear at random timestamps. We can't predict when that will happen next or how strong they will be. But clearly, the entire series isn't random. Between the spikes there's a very deterministic type of decay.



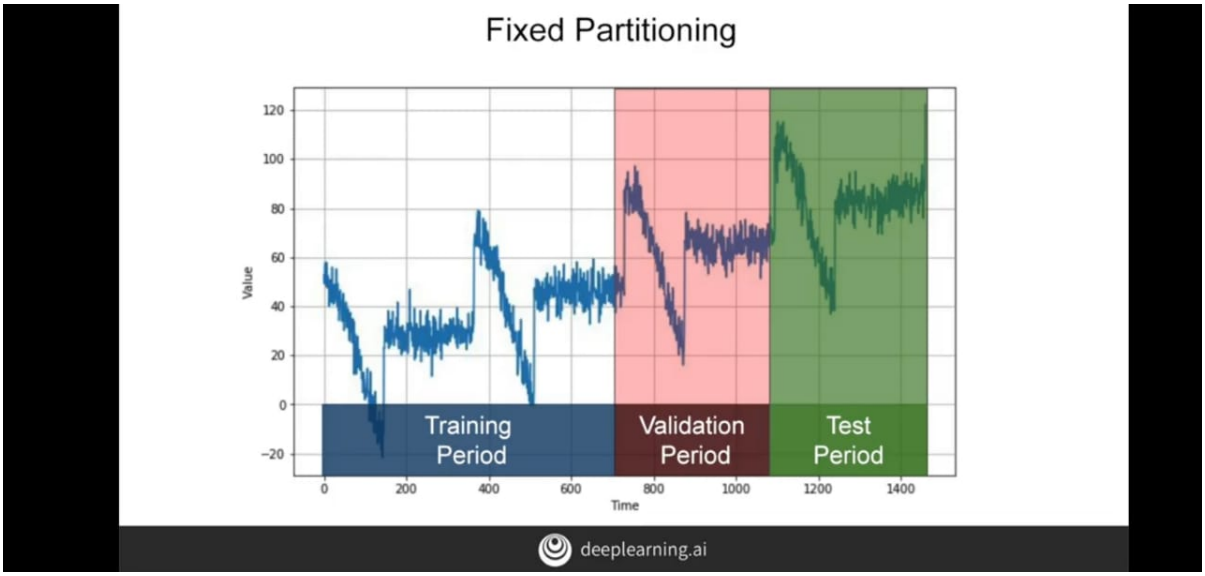
We can see here that the value of each time step is 99% of the value of the previous time step plus an occasional spike. This is an auto correlated time series. Namely it correlated with a delayed copy of itself often called a lag.



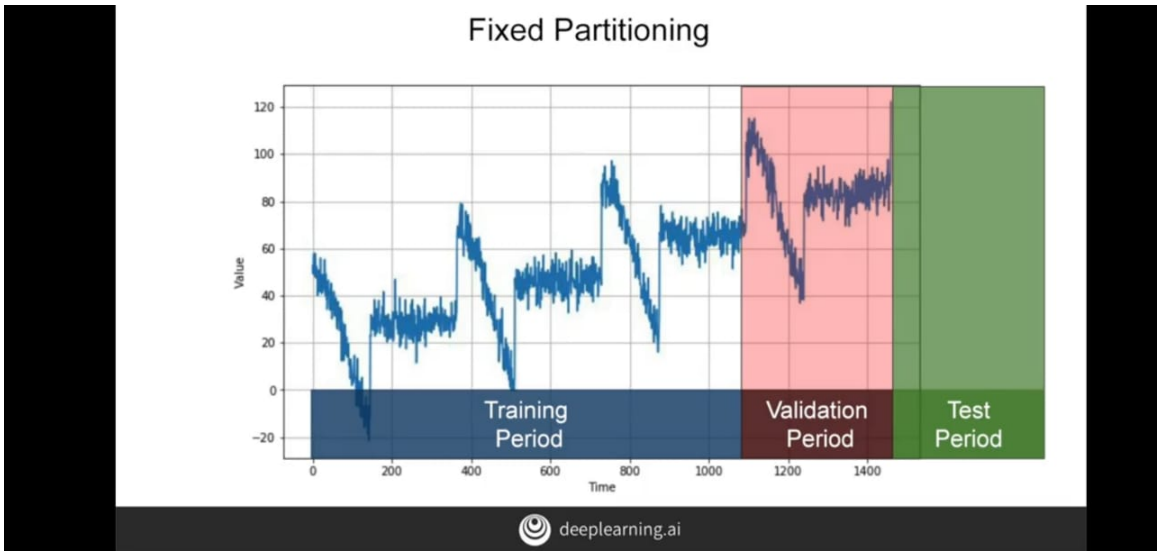
In this example you can see at lag 1 there's a strong autocorrelation. Often a time series like this is described as having memory as steps are dependent on previous ones. The spikes which are unpredictable are often called Innovations. In other words, they cannot be predicted based on past values.

Time series in the real world probably have a bit of each of these features: trend, seasonality, autocorrelation, and noise.

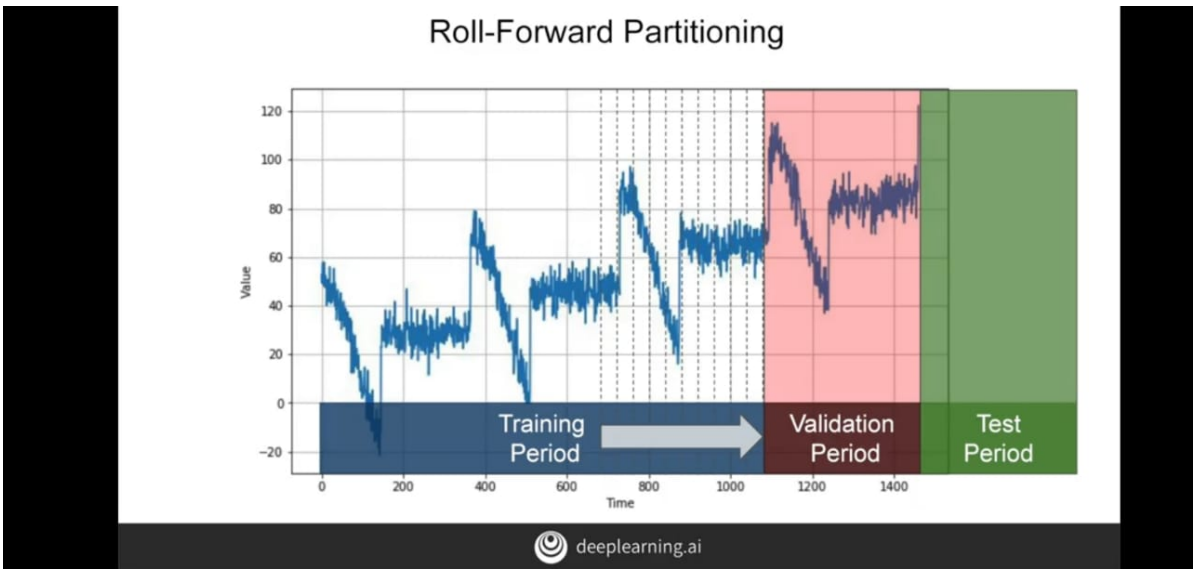
- Data Split:
  - We typically want to split the time series into a training period, a validation period and a test period. This is called fixed partitioning.



If the time series has some seasonality, we generally want to ensure that each period contains a whole number of seasons.
Next we'll train our model on the training period, and we'll evaluate it on the validation period. Here's is where we can experiment to find the right architecture and hyper parameters for training. Often, once we've done that, we can retain using both the training and validation data. And then test on the test period to see if our model performs well.
And if does, then we could take the unusual step of retraining again, using also the test data. But why would we do that?
Well, it's because the test data is the closest data we have to the current point in time. And as such it's often the strongest signal in determining future values. If your model is not trained using that data too, then it may not be optimal.
Due to this, it's actually quite common to forgo a test set all together. And just train, using a training period and a validation period, and the test set is in the future.

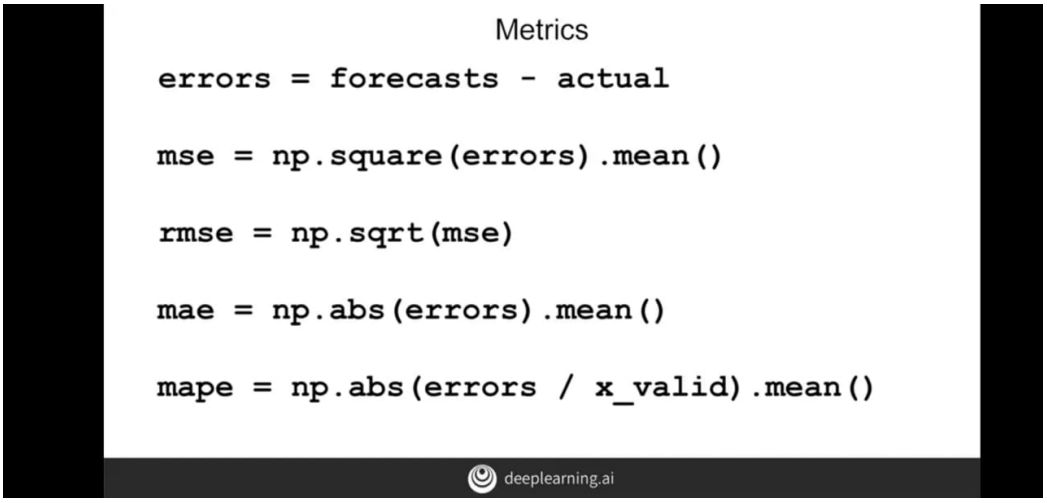


- Fixed partitioning is very simple and very intuitive, but there's also another way called Roll-forward partitioning.



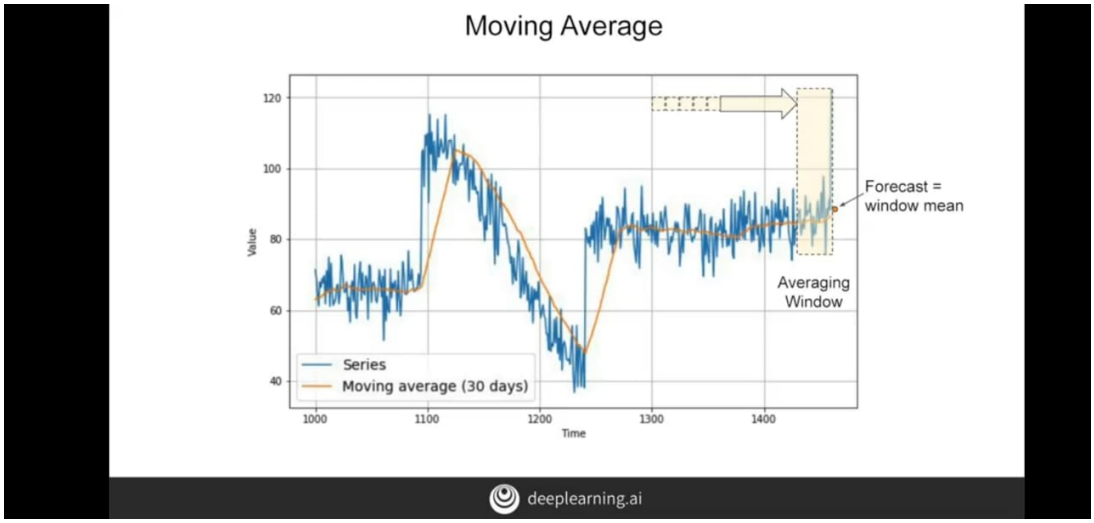
We start with a short training period, and we gradually increase it, say by one day at a time, or by one week at a time. At each iteration, we train the model on a training period. And we use it to forecast the following day, or the following week, in the validation period.
It is like doing fixed partitioning a number of times, and then continually refining the model as such.

- Metrics:



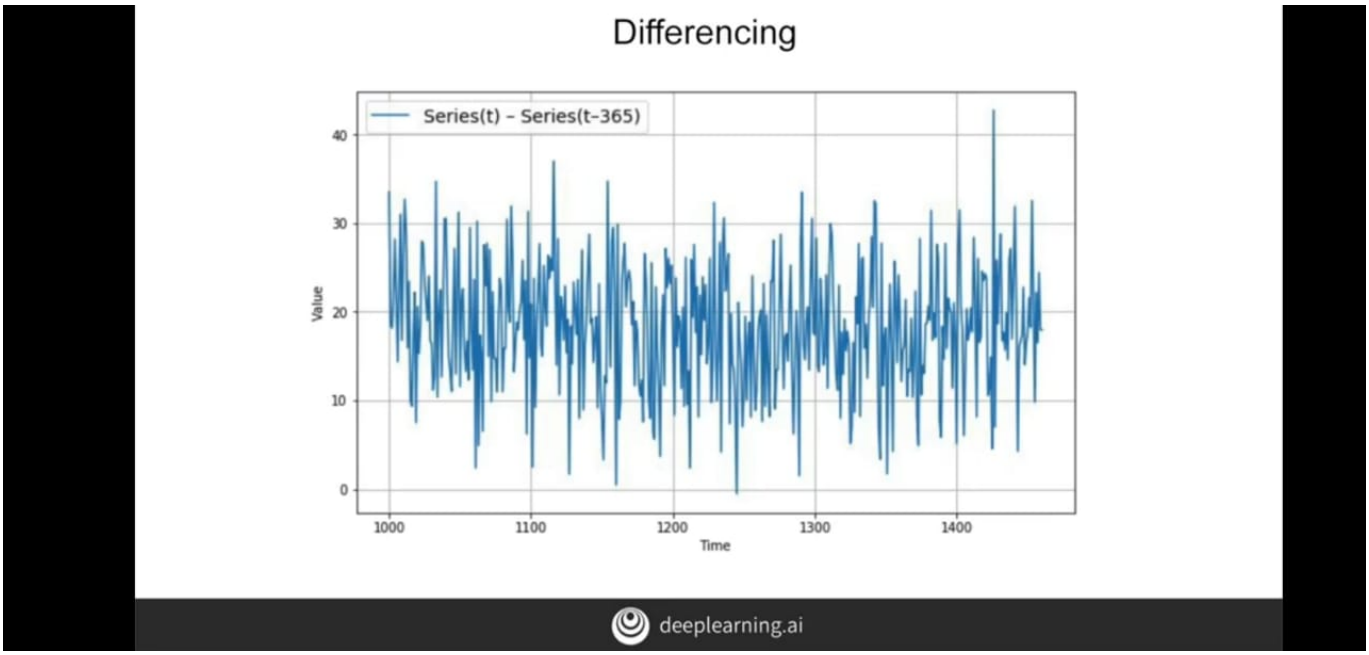
The most common metric to evaluate the forecasting performance of a model is the mean squared error or mse where we square the errors and then calculate their mean.
Another common metric and one of the favorite is the mean absolute error or mae, and it's also called the mean absolute deviation or mad. Instead of squaring to get rid of negatives, it just uses their absolute value. This does not penalize large errors as much as the mse does.
Depending on your task, you may prefer the mae or the mse. For example, if large errors are potentially dangerous and they cost you much more than smaller errors, then you may prefer the mse. But if your gain or our loss is just proportional to the size of the error, then the mae may be better.

- Moving Average:

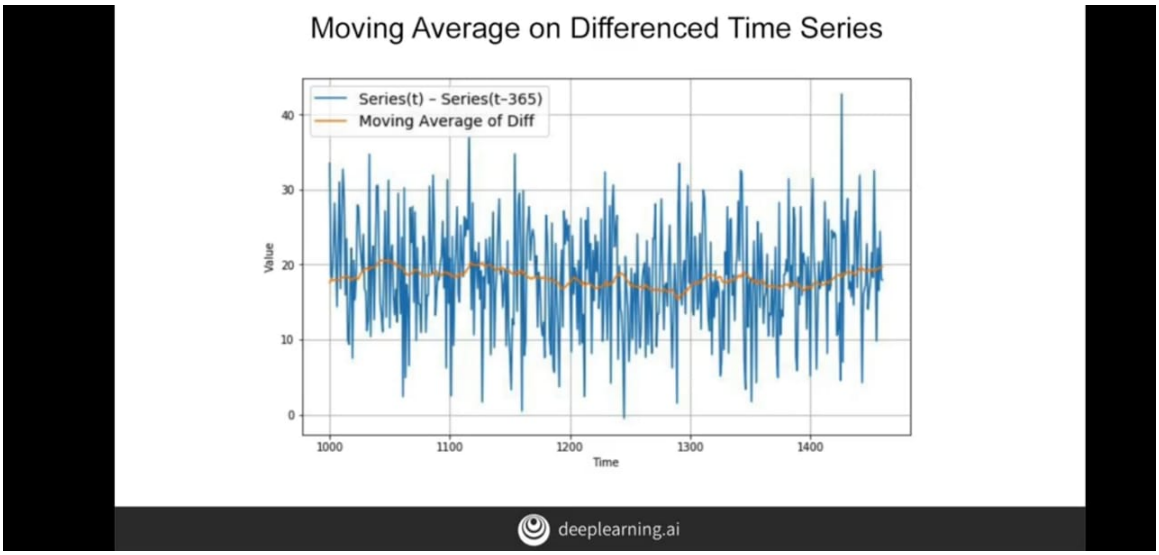


A common and very simple forecasting method is to calculate a moving average. The idea here is that the yellow line is a plot of the average of the blue values over a fixed period called an averaging window, for e.g. 30 days. Now this nicely eliminates a lot of the noise and it gives us a curve roughly emulating the original series, but it does not anticipate trend or seasonality. Depending on the current time i.e. the period after which we want to forecast for the future, it can actually end up being worse than a naïve forecast (baseline model). In this case, for example, we got a mean absolute error of about 7.14

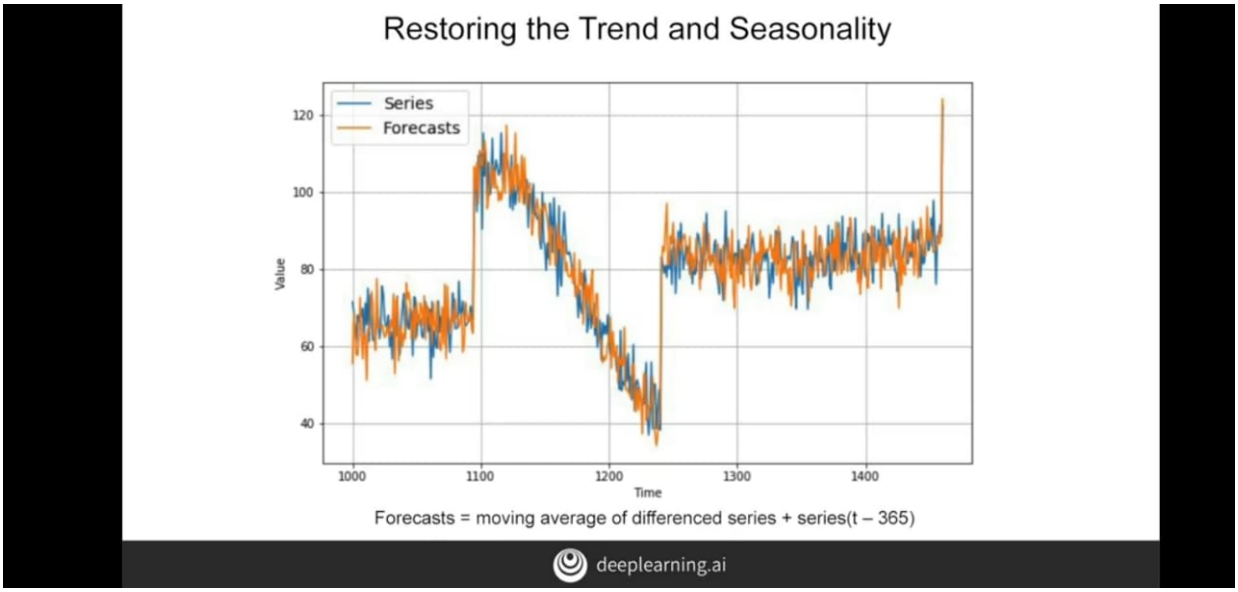
One method to avoid this is to remove the trend and seasonality form the time series with a technique called differencing. So instead of studying the time series itself, we study the difference between the value at time T and the value at an earlier period (depending on the time of our data, that period might be a year, a day, a month or whatever). Let's look at a year earlier. So for this data, at time T minus 365, we'll get this difference time series which has no trend and no seasonality.



We can then use a moving average to forecast this time series which gives us these forecasts.

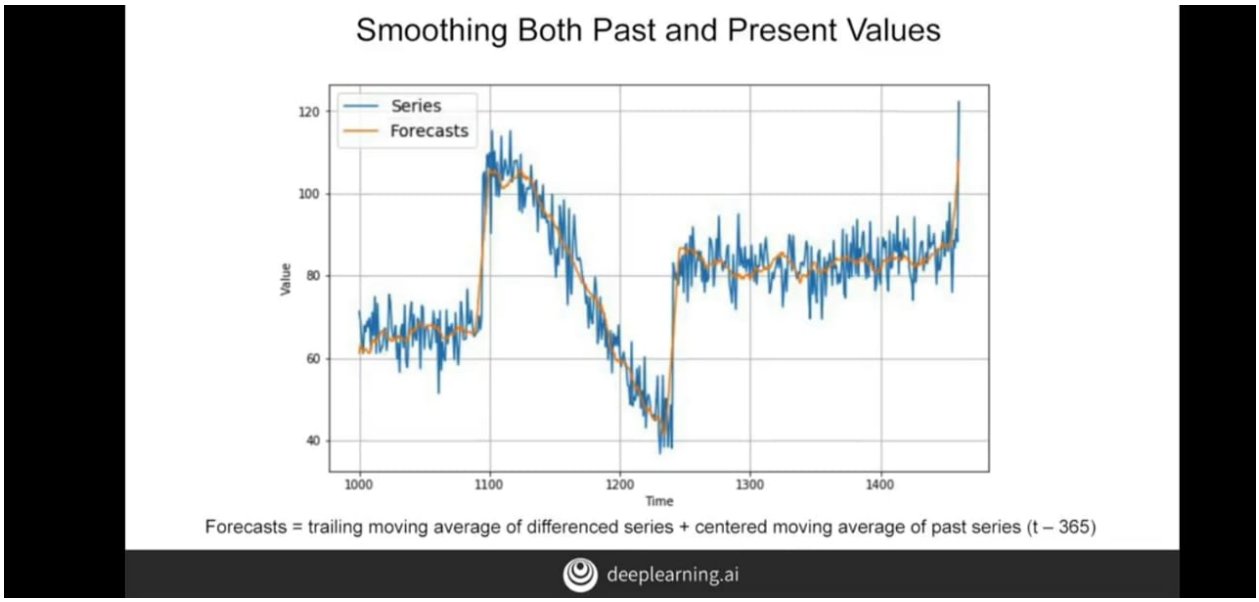


But these are just forecasts for the difference time series, not the original time series.
To get the final forecasts for the original time series, we just need to add back the value at time T minus 365, and we'll get these forecasts.



They look much better. If we measure the mae on the validation period, we get about 5.8. So it's slightly better than naïve forecasting but not tremendously better. We know that moving average removes a lot of noise but our final forecasts are still pretty noisy. Where does that noise come from?
It's coming from the past values that we added back into our forecasts. So

we can improve these forecasts by also removing the past noise using a moving average on that.



If we do that we get much smoother forecasts. In fact, this gives us a mean squared error over the validation period of just about 4.5. Now that's much better than all of the previous methods.

There was no Machine Learning till now, we were solving time series using Statistics. But how do we solve using ML. We'll see now.

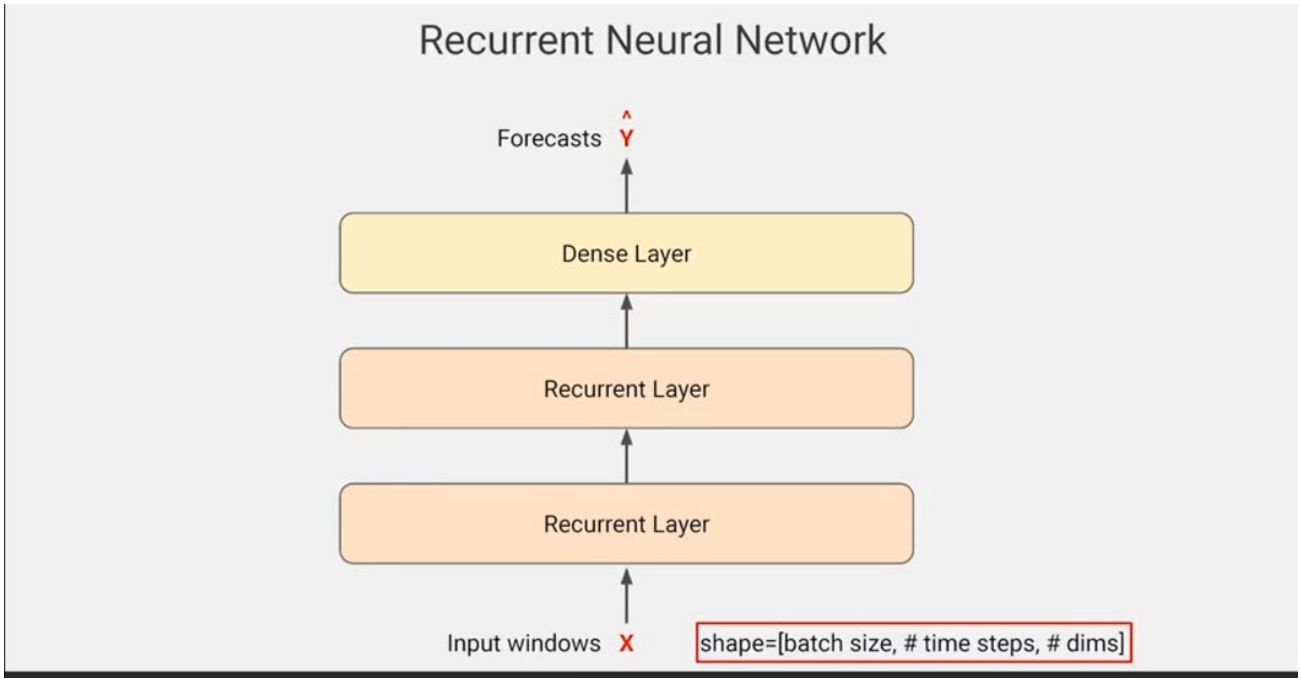
- Preparing features and labels for time series:
  First of all, as with any other ML problem, we have to divide our data into features and labels. In this case our feature is effectively a number of values in the series, with our label being the next value. We'll call that number of values that will treat as our feature, the window size, where we're taking a window of the data and training an ML model to predict the next value.
  So for example if we take our time series data, say 30 days at a time, we'll use 30 values as the feature and the next value is the label. Then over time, we'll train a neural network to match the 30 features to the single label. The code to create a synthetic data for training a neural network can be found in this notebook.
- Once we have the data ready to feed into the network, we can use either a linear regression (single layer, single neuron to predict) or deepNN to train on the data.

- To find the optimal learning rate for the optimizer (adam, SGD, etc.), we use a technique called callbacks while training the network on a fixed number of epochs.

```
model = tf.keras.Sequential(#layers)

lr_schedule = tf.keras.callbacks.LearningRateScheduler(
                lambda epoch: 1e-8 * 10 **(epoch/20))

optimizer = tf.keras.optimizers.SGD(lr = 1e-8, momentum = 0.9)
model.compile(loss = 'mse', optimizer = optimizer)

model.fit(dataset, epochs = 100, callbacks = [lr_schedule])
```

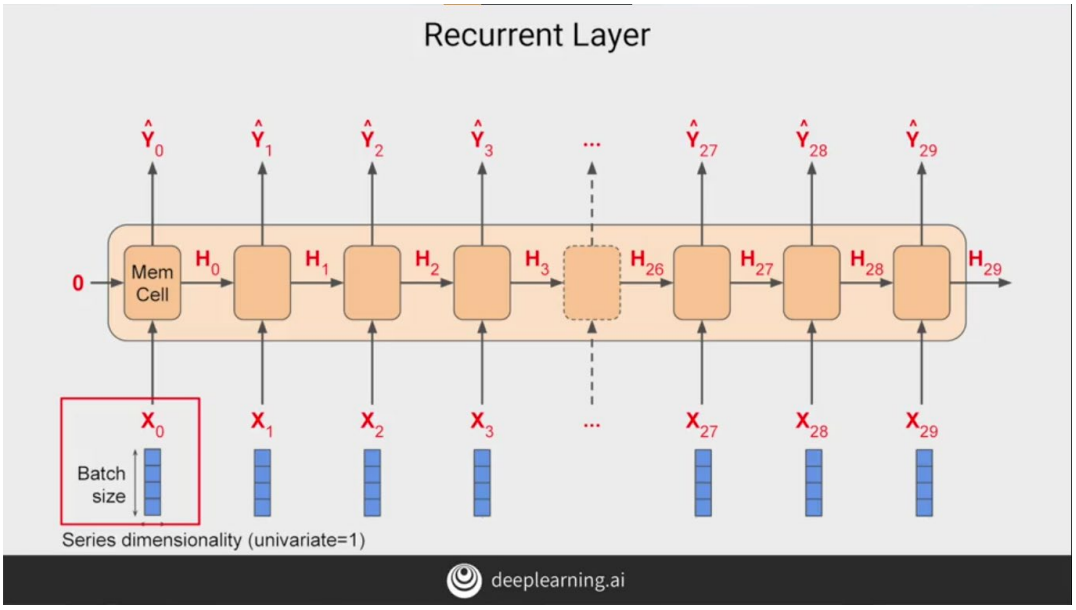
For complete code and how to use this, view this notebook.

- Shape of the input to the RNN layer:

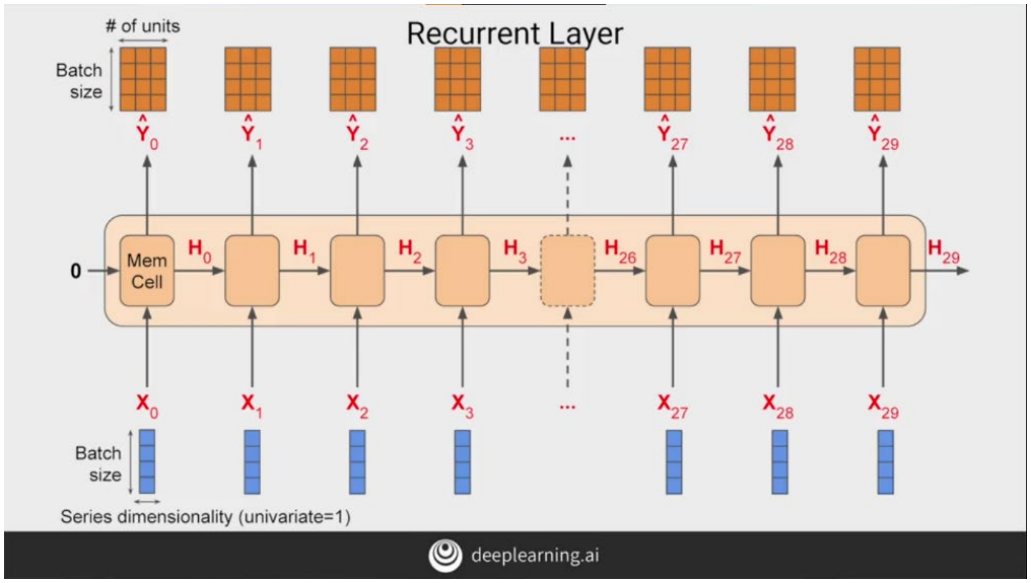


The input shape when using RNN is 3 dimensional. The first dimension will be the batch size, the second will be the timestamps, and the third is the dimensionality of the inputs at each time step (e.g. if it's a univariate time series, this value will be 1, for multivariate it'll be more). So for example, if we have a window size of 30 timestamps and we're batching them in sizes of 4, then the shape will be 4 * 30 * 1.

- More on input shape to the RNN layer:

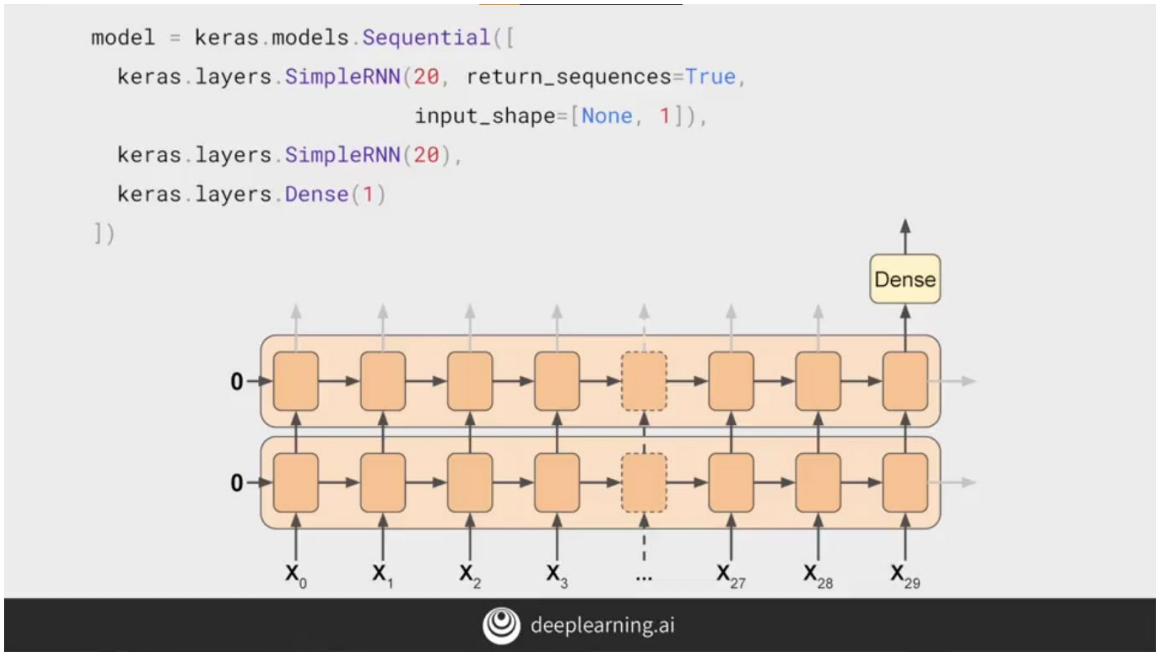


For example, if we have a window size of 30 timestamps and we're batching them in sizes of 4, the shape will be 4*30*1 and at each timestamp, the memory cell input will be a 4*1 matrix (like shown in the image above). The cell will also take the input of the state matrix from the previous step. But of course in this case, in the first step, this will be 0. For subsequent ones, it'll be the output from the previous memory cell. But other than the state vector (Ho in the image), the cell also output a Y value, which we can see in the below image.

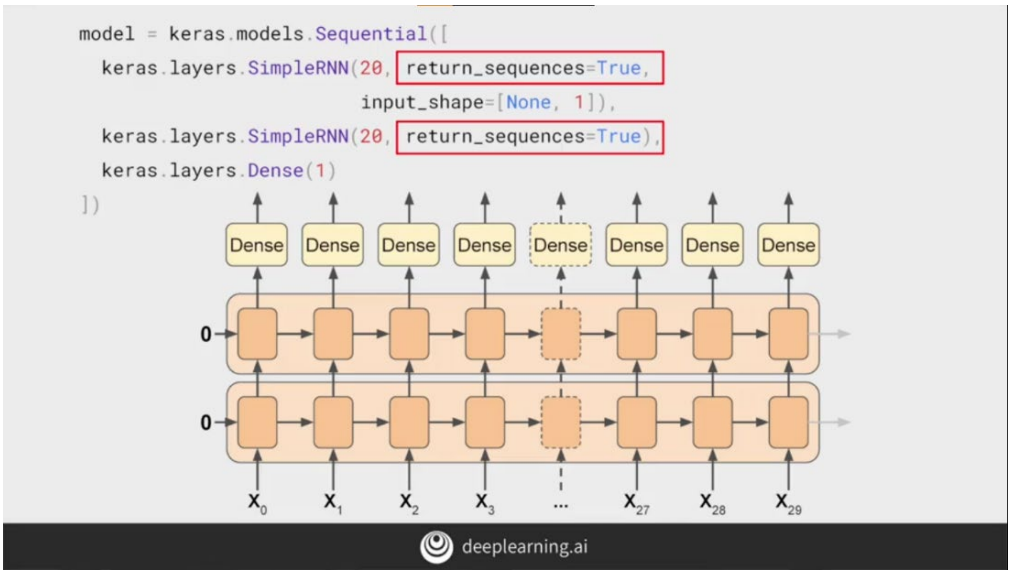


If the memory cell is comprised of 3 neurons, then the output matrix will be 4*3 because the batch size coming in was 4 and the number of neurons is 3. So the full output of the layer is 3D, in this case 4*30*3. With 4 being the batch size, 3 being the number of units and 30 being the number of overall steps (timestamps).

- Outputting a Sequence from RNN:



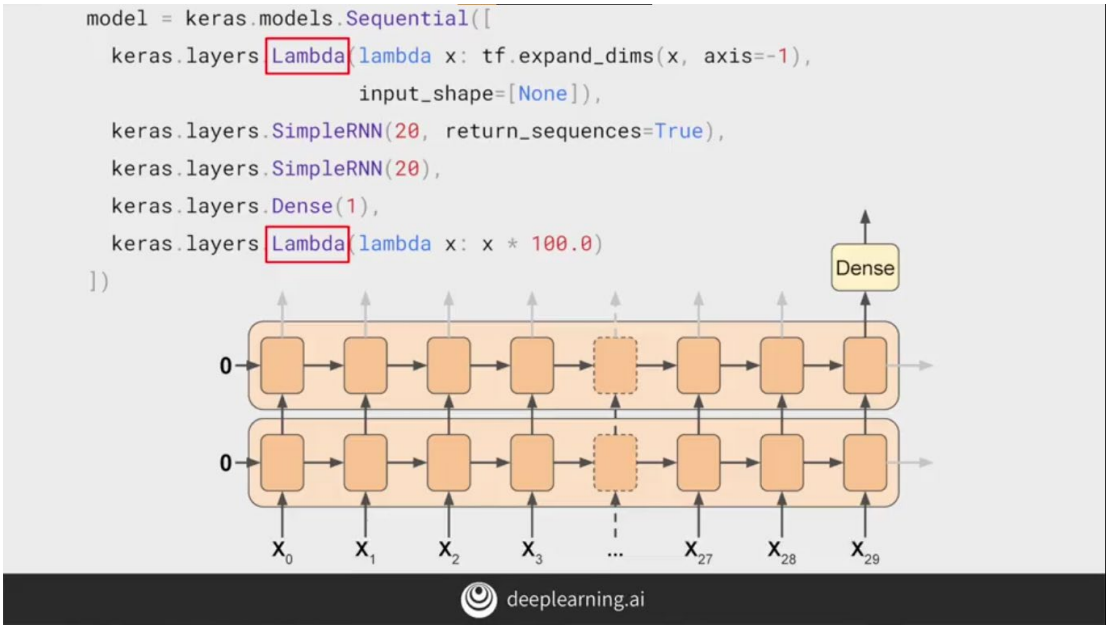
Consider the above RNN, this has two recurrent layers and the first has `return_sequence = True`. It will output a sequence which is fed to the next layer. The next layer does not `have return_sequence` that's set to `True`, so it will only output to the final step (sequence to vector or many to one layer). But notice the `input_shape`, its set to `(None, 1)`. TensorFlow assumes that the first dimension is the batch size, and that it can have any size at all so you don't need to define it. Then the next dimension is the number of timestamps, which we can set to `None`, which means that the RNN can handle sequences of any length. The last



dimension is just 1 because we're using a univariate time series.

If we set `return_sequence = True` for all recurrent layers (as shown above), then they will output sequences and the dense layer will get a sequence as its inputs. Keras handles this by using the same dense layer independently at each time stamp. It might look like multiple ones here but it's the same one that's being reused at each time step. This gives us what is called a sequence to sequence (or many to many) RNN. It's fed a batch of sequences and it returns a batch of sequences of the same length.

- Lambda Layers:



Lambda layer is a type of layer which allows to perform arbitrary operations to effectively expand the functionality of Tensor Flow's Keras and we can do this within the model definition itself.
So the first Lambda layer will be used to help us with our dimensionality. Since the window dataset helper function returns 2D batches of windows on the data, with the first being the batch size and the second the number of timestamps. But an RNN expects 3 dimensions: batch size, the number of timestamps and the series dimensionality (univariate or multivariate). With the Lambda layer, we can fix this without rewriting our window dataset helper function. Using the Lambda we just expand the array by one dimension. By setting input shape to `None`, we're saying that the model can take sequences of any length.
Similarly, if we scale up the outputs by 100, we can help training. The default activation function in the RNN layers is `tanh` which is the hyperbolic tangent activation function. This outputs values between -1 and 1. Therefore we will scale these outputs by multiplying it by 100, since most of the time series values lies in the order of 100s.

- Huber Loss Function:
  The Huber loss is a loss function used in robust regression that is less sensitive to outliers in data than the squared error loss.