MULTIVARIATE TEMPORAL AUTOENCODER FOR PREDICTIVE RECONSTRUCTION OF DEEP SEQUENCES

PRESENTATION BASED ON THE PAPER AND CODE BY JAKOB AUNGIERS

Gene Olafsen

ACKNOWLEDGEMENT

- Paper by Jakob Aungiers of Altum Intelligence
- https://altumintelligence.com/assets/multivariate-temporal-autoencoder-for-predictive-reconstruction-of-deep-sequences/MvTAe-ResearchPaper.pdf
- https://github.com/jaungiers/MvTAe-Multivariate-Temporal-Autoencoder

RAINDROP PATH ON A WINDOW

BEYOND THE RAINDROP

- "There are ample problems where a large amount of data can be gathered at very fine points in time and hence a model can be created to forecast the problem process."

TERMS

- Temporal Autoencoder
- Deep Neural Network
- Time Series
- Multivariate Model
- Feature Engineering
- Signal Processing

FORECAST THE PROBLEM PROCESS

Small closed universe of potential drivers

Easier to forecast for greater sequential steps ahead

Great variety of influencing drivers

Exponential decay of accuracy through chaos and as such are only able to be modelled very short sequential steps ahead

The more influencing drivers of a system that can be worked into the model however, the more accurate the prediction process

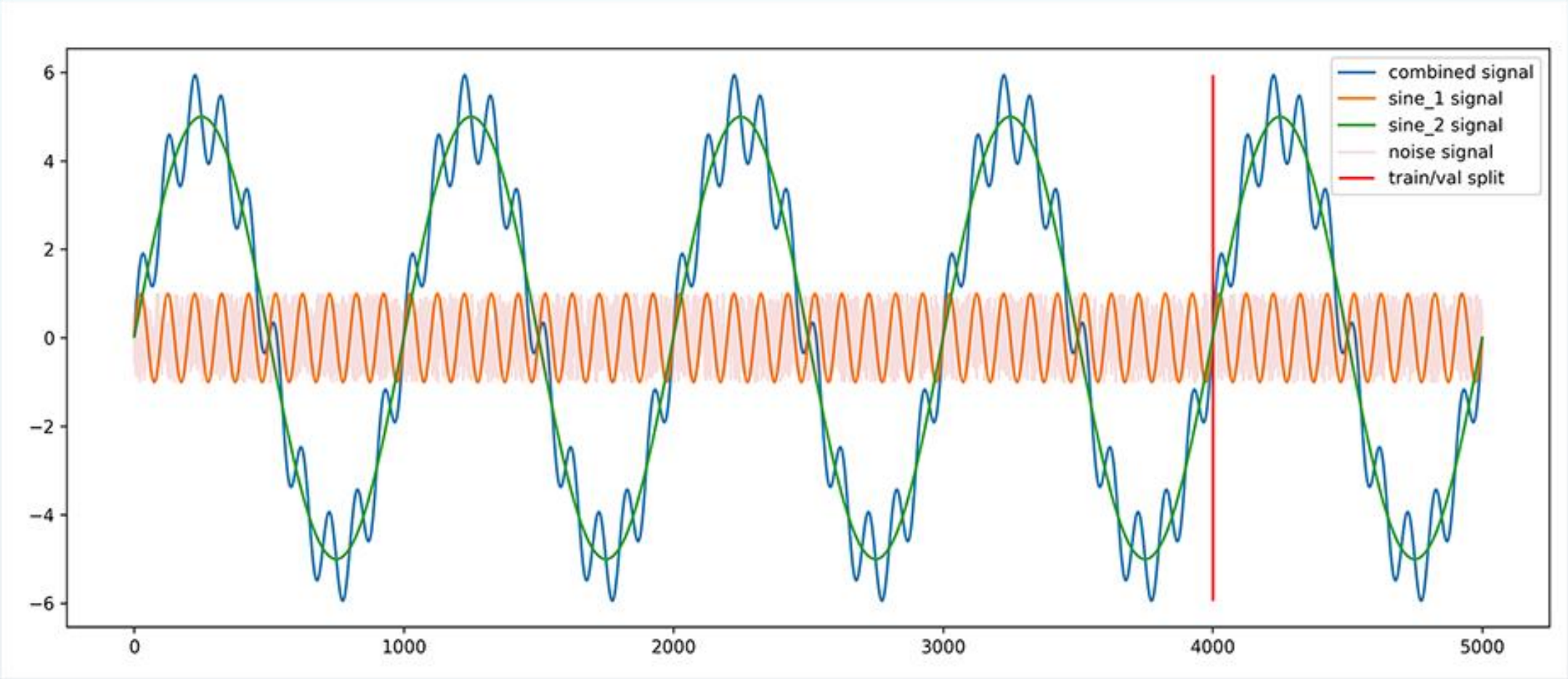
RESEARCH FOCUS

- Build a model which can process multivariate temporal sequences of data.
- Leverage unsupervised learning. Specifically, multi-branch deep neural network approach utilizing autoencoding.
  - Referred to as: Multivariate Temporal Autoencoder (MvTAe)
- The dataset demonstrated is created to be of a toy-dataset nature used to demonstrate the MvTAe model in simple yet fully functional circumstances.

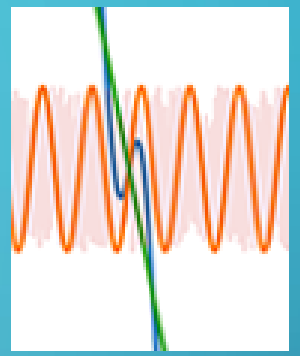
RESEARCH DISCLAIMER

- This research is not concerned with the other major challenge of real-world usage concerning observation, measurement and data processing.

DATASET IMAGE

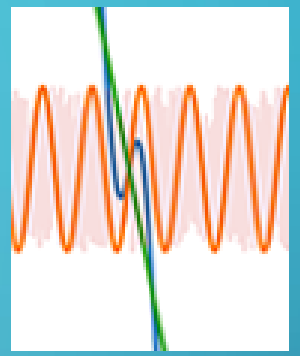


TRAIN AND TEST DATASET



- Sine_1(orange): a sinusoidal wave with a cycle period of 100 timesteps and an amplitude of 1.
  - This is a repeating pattern over time which will test the ability of the model to capture the sequential process of this pattern.
- Sine_2 (green): a sinusoidal wave with a cycle period of 1000 timesteps and an amplitude of 5.
  - This creates a longer term cyclical sequence pattern which the model will not be able to see in full for each training example and hence it tests the models ability to capture cyclical trends.

TRAIN AND TEST DATASET



- Noise (grey): a gaussian distribution of stochastic noise between -1 and +1.
  - Adds an extra dimension of redundant information to test the models ability to identify and disregard dimensions which do not contribute to the latent drivers of the data.
- Combined_signal (blue): a sum of sine_1 and sine_2. This will be used as the Y target variable we are looking to predict and will NOT be included in the X training data that the autoencoder branch of MvTAe sees.
- The red line is the Train/Validation split.

TRAINING DATA SAMPLE

	noise	sine_1	sine_2	combined
0	-0.545357	6.279052e-02	3.141572e-02	9.420624e-02
1	0.151303	1.253332e-01	6.283020e-02	1.881634e-01
2	0.825299	1.873813e-01	9.424220e-02	2.816235e-01
3	-0.486808	2.486899e-01	1.256505e-01	3.743404e-01
4	0.334807	3.090170e-01	1.570538e-01	4.660708e-01
...	...	...	...	...
4995	0.029877	-2.486899e-01	-1.256505e-01	-3.743404e-01
4996	0.687034	-1.873813e-01	-9.424220e-02	-2.816235e-01
4997	0.368473	-1.253332e-01	-6.283020e-02	-1.881634e-01
4998	0.623531	-6.279052e-02	-3.141572e-02	-9.420624e-02
4999	-0.969568	1.959370e-15	1.163780e-14	1.359720e-14

5000 rows × 4 columns

DATA PREPARATION

- The training dataset is split into sliding windows of length N with step S between each window.
- This approach allows the training of our autoencoder branch to lookback across N temporal steps to determine relationship patterns within the temporal sequence.

WINDOW CODE

- The code for creating the normalized sliding windows across dimensions and the accompanying target variable.

```
idx_front = 0
idx_rear = window_size
features_x = ['sine_1', 'sine_2', 'noise']
feature_y = 'combined'

tr_data_windows_size = int(np.ceil((data['sine_1'][:idx_val_split].shape[0]-window_size-1)/step_size))
tr_data_windows = np.empty((tr_data_windows_size, len(features_x), window_size))
tr_data_windows_y = np.zeros(tr_data_windows_size)

i = 0
pbar = tqdm(total=tr_data_windows_size-1, initial=i)
while idx_rear + 1 < data['sine_1'][:idx_val_split].shape[0]:
    # create x data windows
    for j, feature in enumerate(features_x):
        _data_window, _hi, _lo = norm(data[feature][idx_front:idx_rear])
        tr_data_windows[i][j] = _data_window

    # create y along same normalized scale
    _, hi, lo = norm(data[feature_y][idx_front:idx_rear])
    _y = norm(data[feature_y][idx_rear], hi, lo)[0]
    tr_data_windows_y[i] = _y

    idx_front = idx_front + step_size
    idx_rear = idx_front + window_size
    i += 1
    pbar.update(1)
pbar.close()

# reshape input into [samples, timesteps, features]
tr_data_size = tr_data_windows.shape[0]
tr_input_seq = tr_data_windows.swapaxes(1,2)
```

NORMALIZE / DE-NORMALIZE DATA

- Normalize
  - The process deals with temporal data windows along multiple dimensions.
  - It treats each window and each dimension within the window as independent (of other dimensions) in terms of normalization.
- De-normalize
  - The de-normalization process is required to bring data back to the input scale.

MINMAX NORMALIZATION

- With MinMax normalization we normalize data using the min (lo) and max (hi) values of the data window and hence these values created during the normalization process are required for the de-normalization process.

NORMALIZE / DE-NORMALIZE DATA

- Normalize

$$W_{normalized} = \frac{W^k_{(i-N)\to i} - min(W^k_{(i-N)\to i})}{max(W^k_{(i-N)\to i}) - min(W^k_{(i-N)\to i})}$$

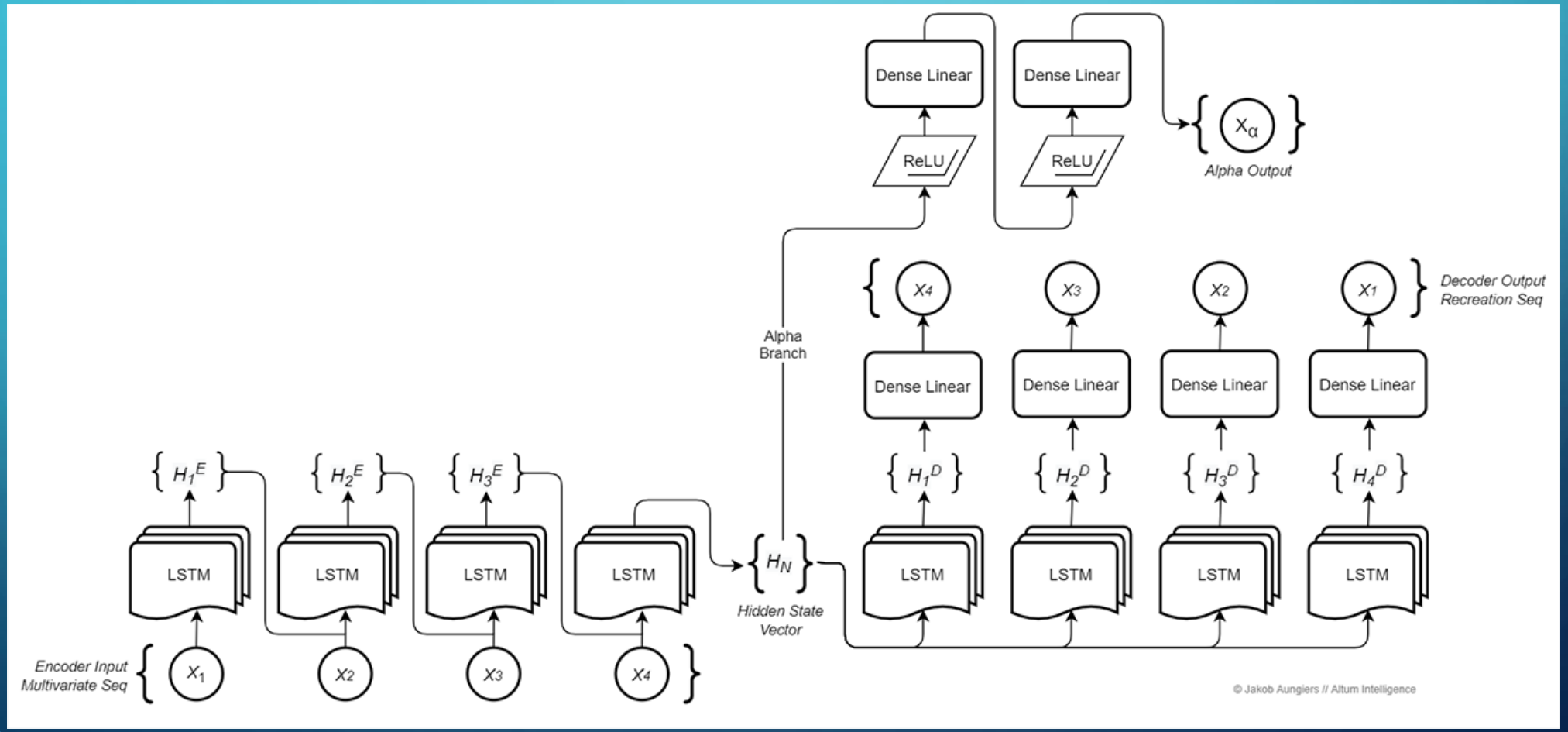
```
def norm(data, hi=None, lo=None):
    hi = np.max(data) if not hi else hi
    lo = np.min(data) if not lo else lo
    if hi-lo == 0:
        return 0, hi, lo
    y = (data-lo)/(hi-lo)
    return y, hi, lo
```

- De-normalize

$$W_{denormalized} = W^k_{(i-N)\to i} \times (hi^k_i - lo^k_i) + lo^k_i$$

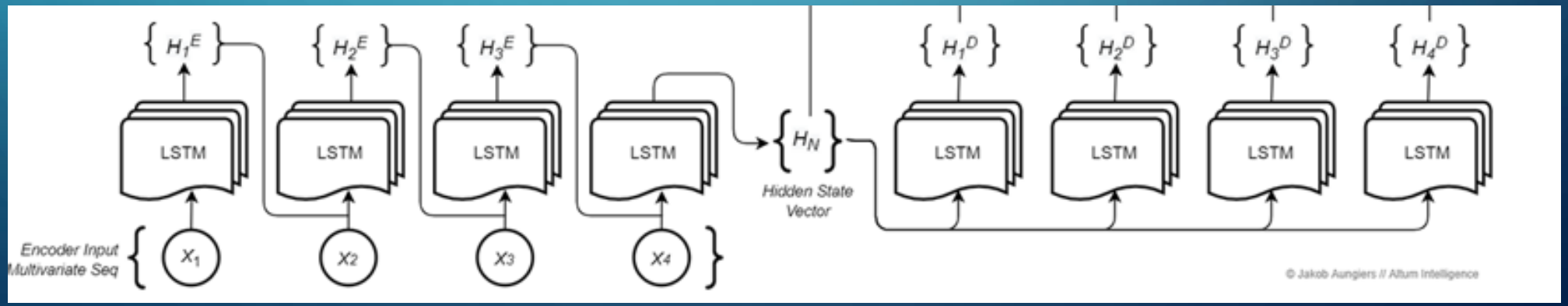
```
def reverse_norm(y, hi, lo):
    x = y*(hi-lo)+lo
    return x
```

MULTIVARIATE TEMPORAL AUTOENCODER MODEL



ENCODER-DECODER BRANCH

- The first branch of the MvTAe model is composed of two parts.
- An encoder transforms the input sequence into the hidden state vector and a decoder which takes a hidden state vector and transforms it back into the original sequence, albeit in reverse. - Typically refered to an an *autoencoder*.

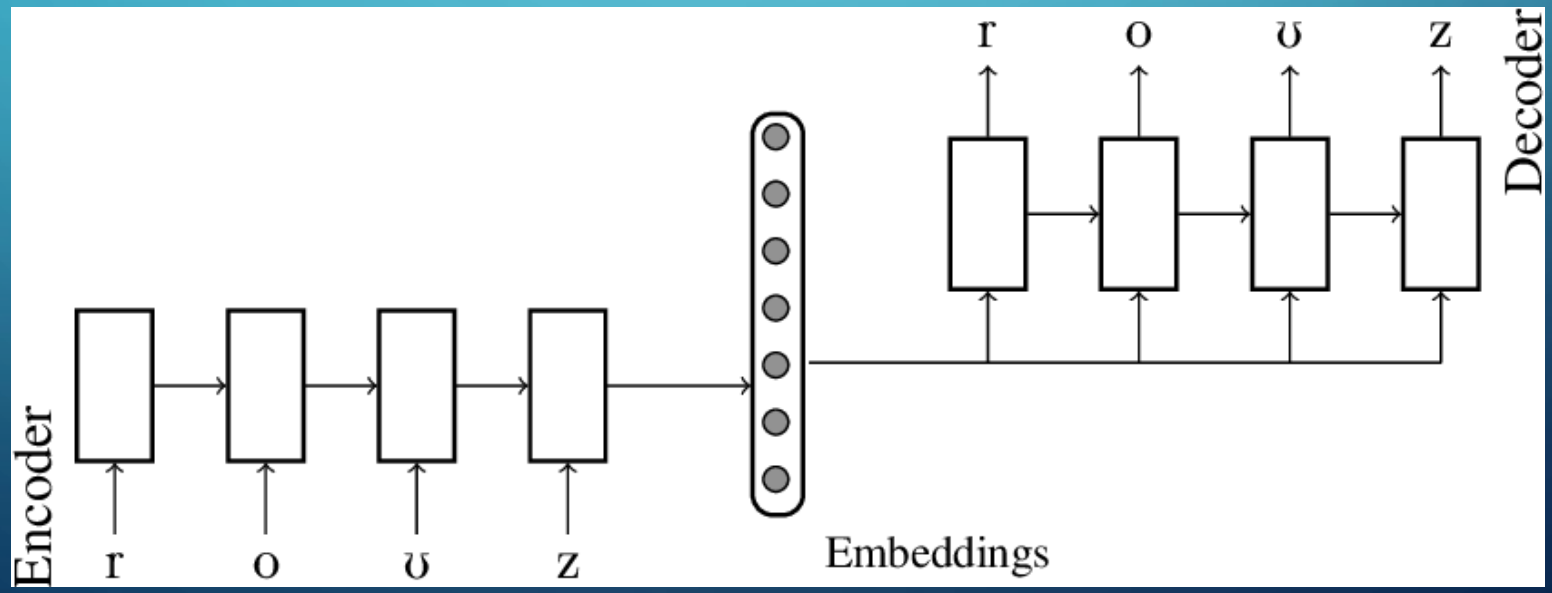


LET'S LOOK INTO LSTM AUTOENCODERS

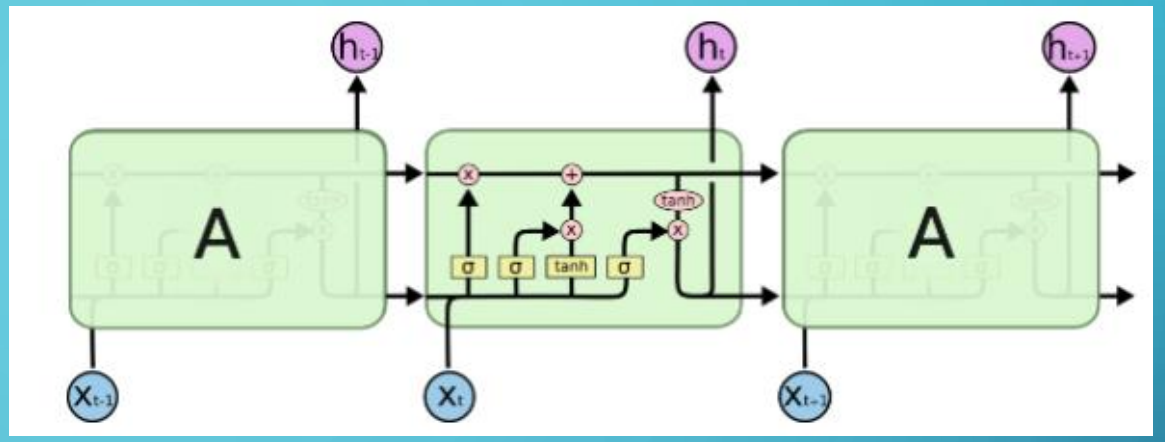
- Autoencoders use an Encoder-Decoder LSTM architecture.
- Long short-term memory (**LSTM**) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. Unlike standard feedforward neural networks, **LSTM** has feedback connections
- Autoencoders are a type of self-supervised learning model that can learn a compressed representation of input data.
- Additionally, LSTM Autoencoders can learn a compressed representation of sequence data and have been used on video, text, audio, and time series data.

ENCODER / DECODER

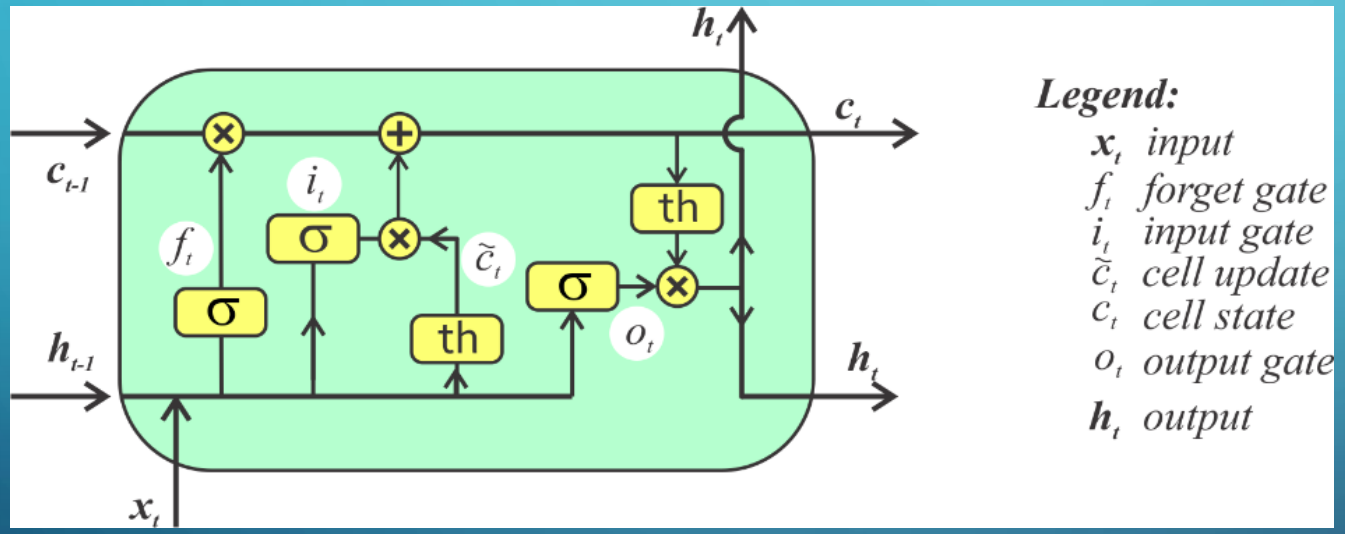
- The design of the autoencoder model purposefully makes training challenging by restricting the architecture to a bottleneck at the midpoint of the model, from which the reconstruction of the input data is performed.



RNN'S

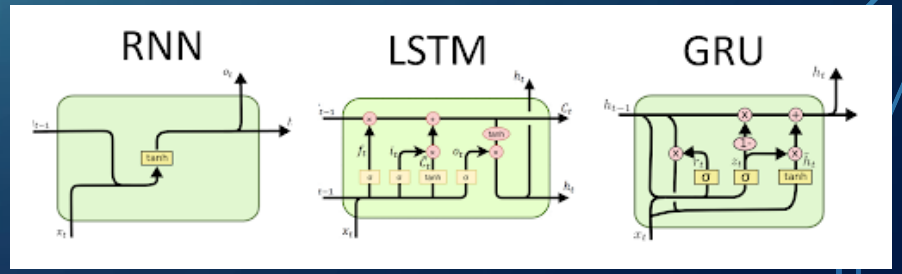


- LSTM



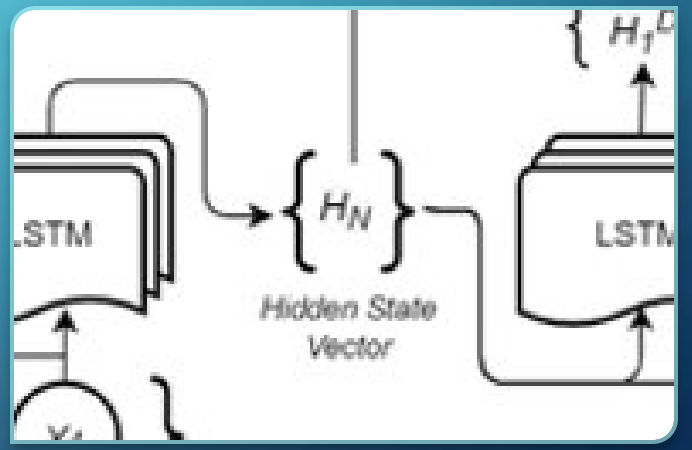
Legend:

x_t *input*
f_t *forget gate*
i_t *input gate*
$\tilde{c}_t$ *cell update*
c_t *cell state*
o_t *output gate*
h_t *output*



HIDDEN STATE VECTOR AND FEATURE ENGINEERING

- This hidden state vector, when the EncoderDecoder is properly trained, can be regarded as a high dimensional approximation of the drivers that make up the full dimensions of the entire input sequence.
- In essence, this is the feature vector that traditional feature engineering aims to create and which is then used with the second branch of the model to predict future sequence steps, however the creation of this feature vector/hidden state vector is done in an unsupervised way by the decoder.



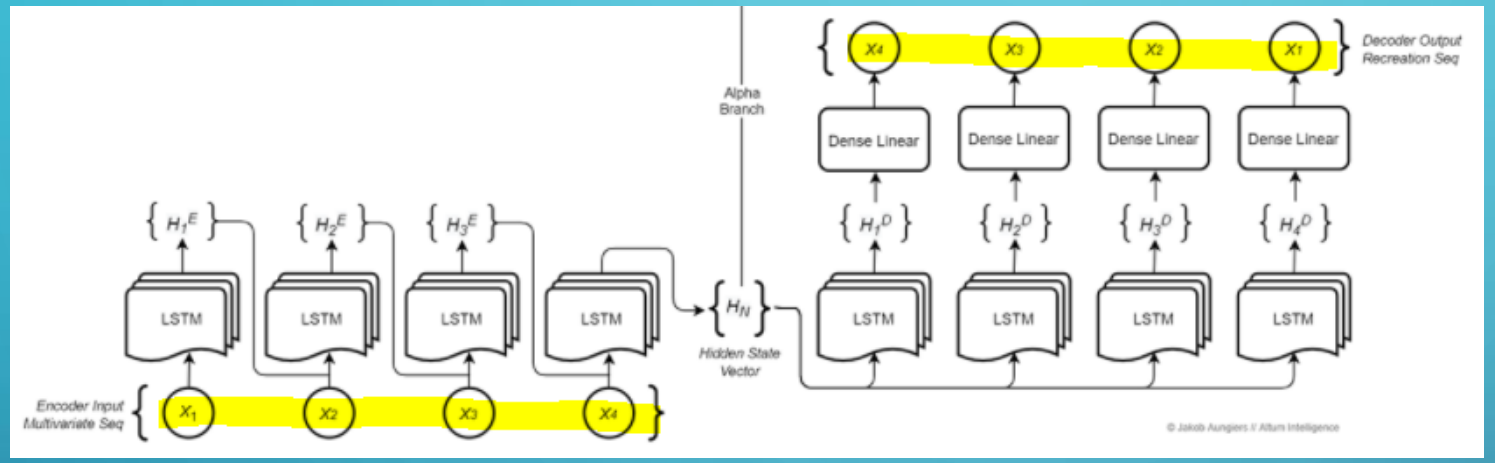
DECODER OVERVIEW

- The decoder structure is similar to the encoder in the sense that it is composed of the same layer of LSTM cells equal to the sequence length. The input to each of these cells is the hidden state vector created from the final context vector of the last encoder LSTM cell, copied across into each decoder cell.

DECODER 'TRICK'

- The decoder output – and what makes this process unsupervised – is the same input as to the encoder, hence the model acts in an autoencoder fashion mapping $X \rightarrow X$.
- Note: The decoder output targets are the reversed input of X (Xˆ) hence $X \rightarrow \hat{X}$. This is done as Sutskever et al. [5] found reversing the decoder targets significantly improves modelling accuracy, likely due to a higher influence of short-term dependencies within the sequences as opposed to longer term patterns.

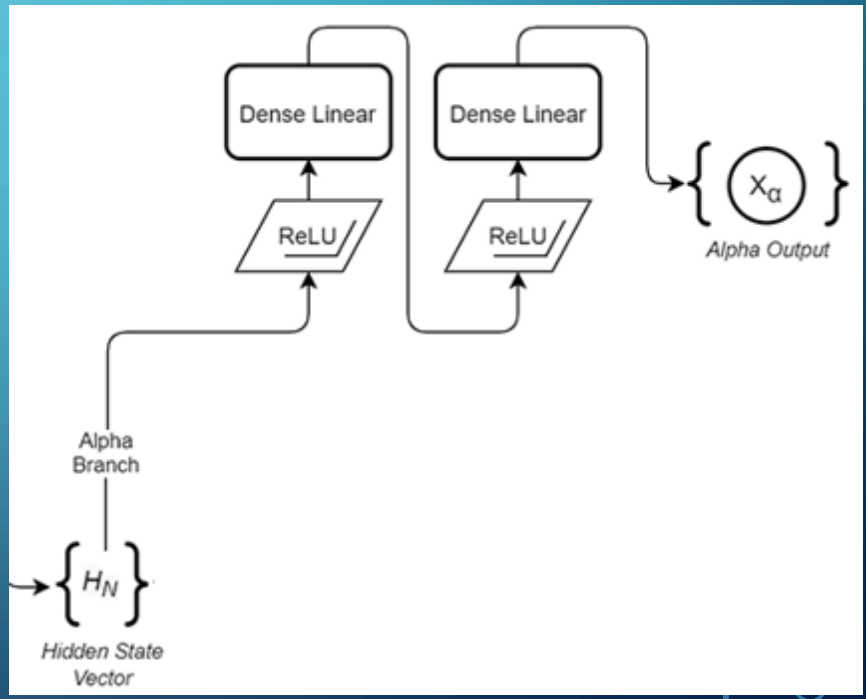
DECODER DETAILS




```
# Encoder
encoder_out, encoder_hidden = self.encoder(x)
hidden_state_vector = encoder_hidden[0]

# Decoder
encoder_hidden_dropout = self.dropout(hidden_state_vector)
decoder_out, decoder_hidden = self.decoder(encoder_hidden_dropout.repeat(self.seq_len, 1, 1))
decoder_output = self.decoder_output(decoder_out.transpose(0,1))
```

PREDICTIVE (ALPHA) BRANCH

- The second branch of the MvTAe model acts as a predictive branch - we call this the Alpha branch, as it generates a predictive alpha signal as its output. (The second branch of the model is expected to predict future sequence steps.)
- Its input is the output of the encoder - the hidden state vector which, when trained sufficiently, represents the underlying context and drivers of the dataset, and hence can be used to train the predictive alpha branch for a forward looking prediction of the dataset.



ALPHA BRANCH CODE STRUCTURE

- The alpha branch has a fully connected structure.
  - Two fully connected hidden layers of neurons.
  - This fully connected layer enables the backpropagation training process to capture higher dimensionality linear functions within the hidden state vector data while allowing the LSTM decoder layer to focus on capturing the non-linear sequential functions within the data.

(continued next slide)

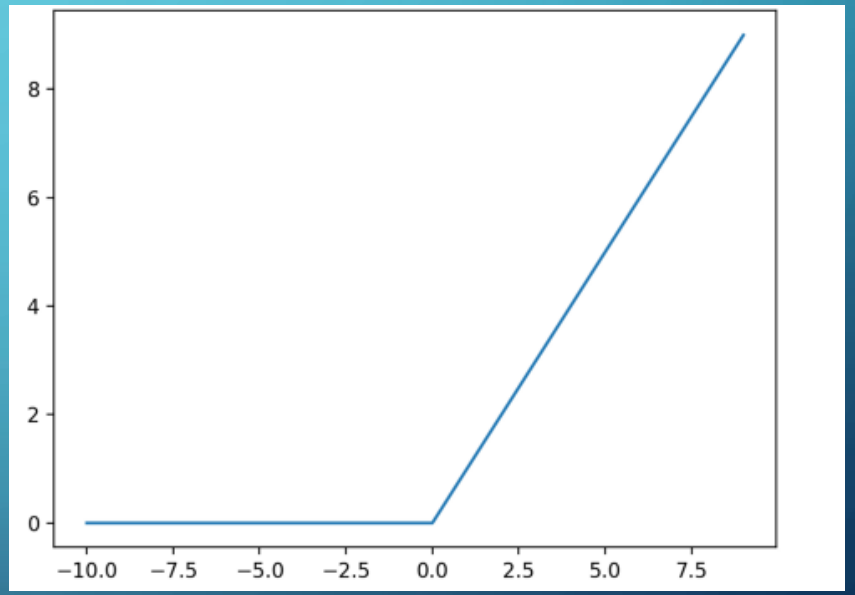
ALPHA BRANCH CODE STRUCTURE

- The alpha branch has a fully connected structure.
  - ReLU (Rectified Linear Units) activation functions are associated with these layers.
  - ReLU provide statibility in modelling non-linearity, which most complex sequential problems require (as shown by Zeiler et al.), and help alleviate the problem of vanishing/exploding gradients in the backpropagation process.

```
self.alpha_hidden_1 = nn.Linear(self.encoder.hidden_size, hidden_alpha_size)
self.alpha_hidden_2 = nn.Linear(hidden_alpha_size, hidden_alpha_size)
self.alpha_out = nn.Linear(hidden_alpha_size, 1)

alpha_hidden_1 = F.relu(self.alpha_hidden_1(hidden_state_vector))
alpha_hidden_1_dropout = self.dropout(alpha_hidden_1)
alpha_hidden_2 = F.relu(self.alpha_hidden_2(alpha_hidden_1_dropout))
alpha_output = self.alpha_out(alpha_hidden_1).squeeze()
```

RELU OVERVIEW

- **Rectified Linear Unit** is an activation function used in nearly all modern neural network architectures.
- ReLU is defined as **max(0, x)**.
- This function that will output the input directly if it is positive, otherwise, it will output zero.
- Similarly, the derivative of the rectified linear function is also easy to calculate.
  - The derivative of the activation function is required when updating the weights of a node as part of the backpropagation of error.
  - The derivative of the function is the slope. The slope for negative values is 0.0 and the slope for positive values is 1.0.



ACTIVATION FUNCTIONS

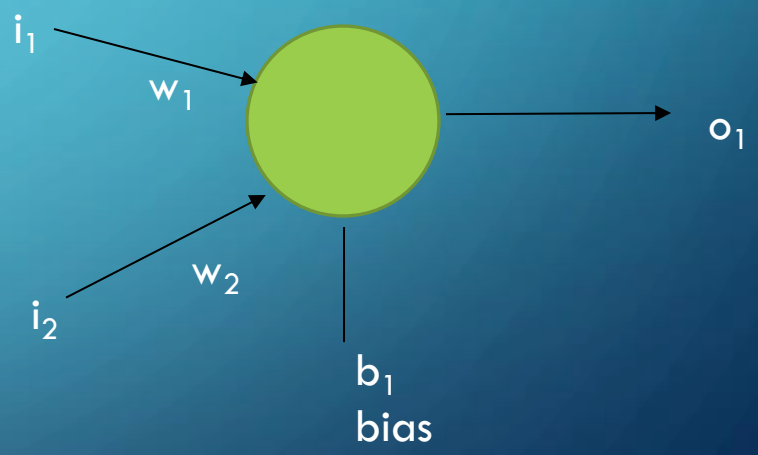
- An activation outputs a small value for small inputs, and a larger value if its inputs exceed a threshold. If the inputs are large enough, the activation function "fires", otherwise it does nothing. In other words, an activation function is like a gate that checks that an incoming value is greater than a critical number.
- "Activation functions are useful because they add non-linearities into neural networks, allowing the neural networks to learn powerful operations. If the activation functions were to be removed from a feedforward neural network, the entire network could be re-factored to a simple linear operation or matrix transformation on its input, and it would no longer be capable of performing complex tasks such as image recognition." --DeepAI

RELU ADVANTAGES

- Computationaly 'cheap' - no exponential calculations: tanh, sigmoid
- Capable of outputting a true zero value. "This is called a sparse representation and is a desirable property in representational learning as it can accelerate learning and simplify the model." -- Page 507, Deep Learning, 2016.

NN.LINEAR

- network = torch.nn.Linear(2,1);
  - Creates a network as show.
  - Weight and Bias values are initialized with random values.

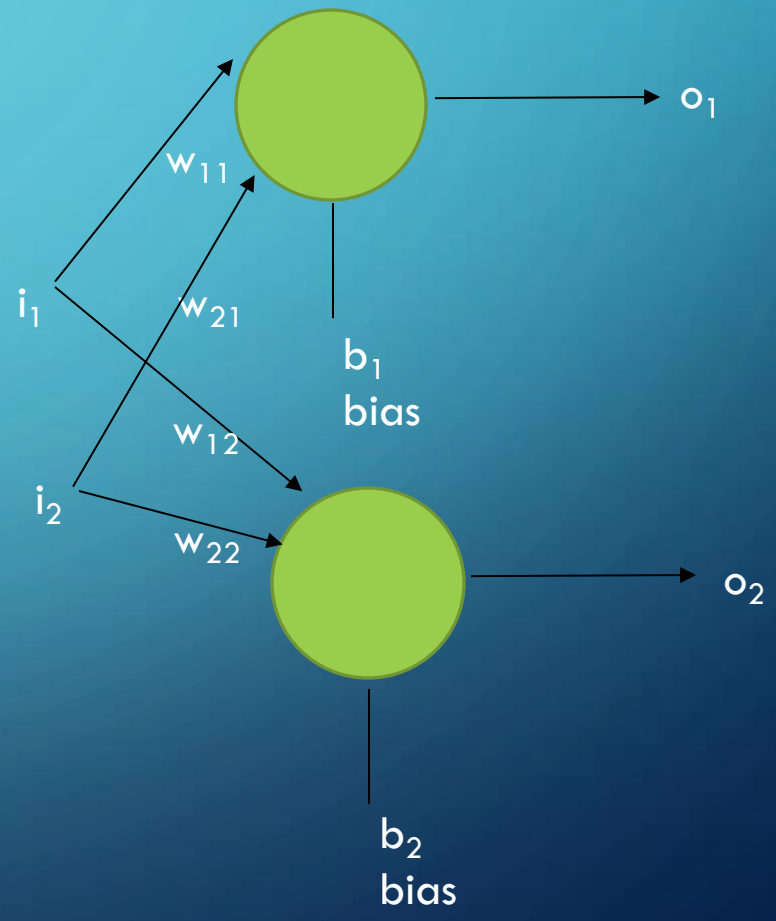


PYTORCH NN.LINEAR

- nn.Linear(n,m) is a module that creates single layer feed forward network with n inputs and m output. Mathematically, this module is designed to calculate the linear equation $Ax = b$ where x is input, b is output, A is weight. This is where the name 'Linear' came from.

NN.LINEAR (2,2)

- network = torch.nn.Linear(2,2);



MVTAE MODEL CONSTRUCTOR

- `nn.Module` (uppercase M) is a PyTorch specific concept. The code will subclass the `nn.Module` (used as a base-class and able to keep track of state).
- At some point later, in the constructor there is a call to a method to build the model.

```
18  class MVTAEModel(nn.Module):
19
20      def __init__(self,
21                   model_save_path,
22                   seq_len, in_data_dims,
23                   out_data_dims,
24                   model_name,
25                   hidden_vector_size,
26                   hidden_alpha_size,
27                   dropout_p,
28                   optim_lr):
29          super(MVTAEModel, self).__init__()
```

```
44
45          self.build_model(hidden_vector_size, hidden_alpha_size, dropout_p, optim_lr)
```

BUILD THE MVTAE MODEL

```
49      def build_model(self, hidden_vector_size, hidden_alpha_size, dropout_p, optim_lr):
50          self.dropout = nn.Dropout(p=dropout_p)
51          self.encoder = nn.LSTM(input_size=self.in_data_dims, hidden_size=hidden_vector_size, batch_first=True)
52          self.decoder = nn.LSTM(input_size=self.encoder.hidden_size, hidden_size=self.encoder.hidden_size, batch_first=False)
53          self.decoder_output = nn.Linear(self.encoder.hidden_size, self.out_data_dims)
54
55          self.alpha_hidden_1 = nn.Linear(self.encoder.hidden_size, hidden_alpha_size)
56          self.alpha_hidden_2 = nn.Linear(hidden_alpha_size, hidden_alpha_size)
57          self.alpha_out = nn.Linear(hidden_alpha_size, 1)
58
59          self.loss_decoder = nn.MSELoss()
60          self.loss_alpha = nn.MSELoss()
61          self.optimizer = optim.Adam(self.parameters(), lr=optim_lr)
```

BUILD A KERAS MODEL

```
10 # define the keras model
11 model = Sequential()
12 model.add(Dense(12, input_dim=8, activation='relu'))
13 model.add(Dense(8, activation='relu'))
14 model.add(Dense(1, activation='sigmoid'))
```

KERAS VS PYTORCH

- Keras is a high-level API capable of running on top of TensorFlow, CNTK, Theano, or MXNet (or as tf.contrib within TensorFlow). Since its initial release in March 2015, it has gained favor for its ease of use and syntactic simplicity, facilitating fast development. It's supported by Google.
- PyTorch, released in October 2016, is a lower-level API focused on direct work with array expressions. It has gained immense interest in the last year, becoming a preferred solution for academic research, and applications of deep learning requiring optimizing custom expressions. It's supported by Facebook.

FORWARD

- The forward function computes **output** Tensors from **input** Tensors. The backward function receives the gradient of the **output** Tensors with respect to some scalar **value**, and computes the gradient of the **input** Tensors with respect to that same scalar **value**.
- The forward function is *user-defined* when deriving from nn.Module.

AUTOGRAD

- The `autograd` package is central to all neural networks in PyTorch.
- The `autograd` package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.
- `torch.Tensor` is the central class of the package. If you set its attribute `.requires_grad` as `True`, it starts to track all operations on it. When you finish your computation you can call `.backward()` and have all the gradients computed automatically. The gradient for this tensor will be accumulated into `.grad` attribute.

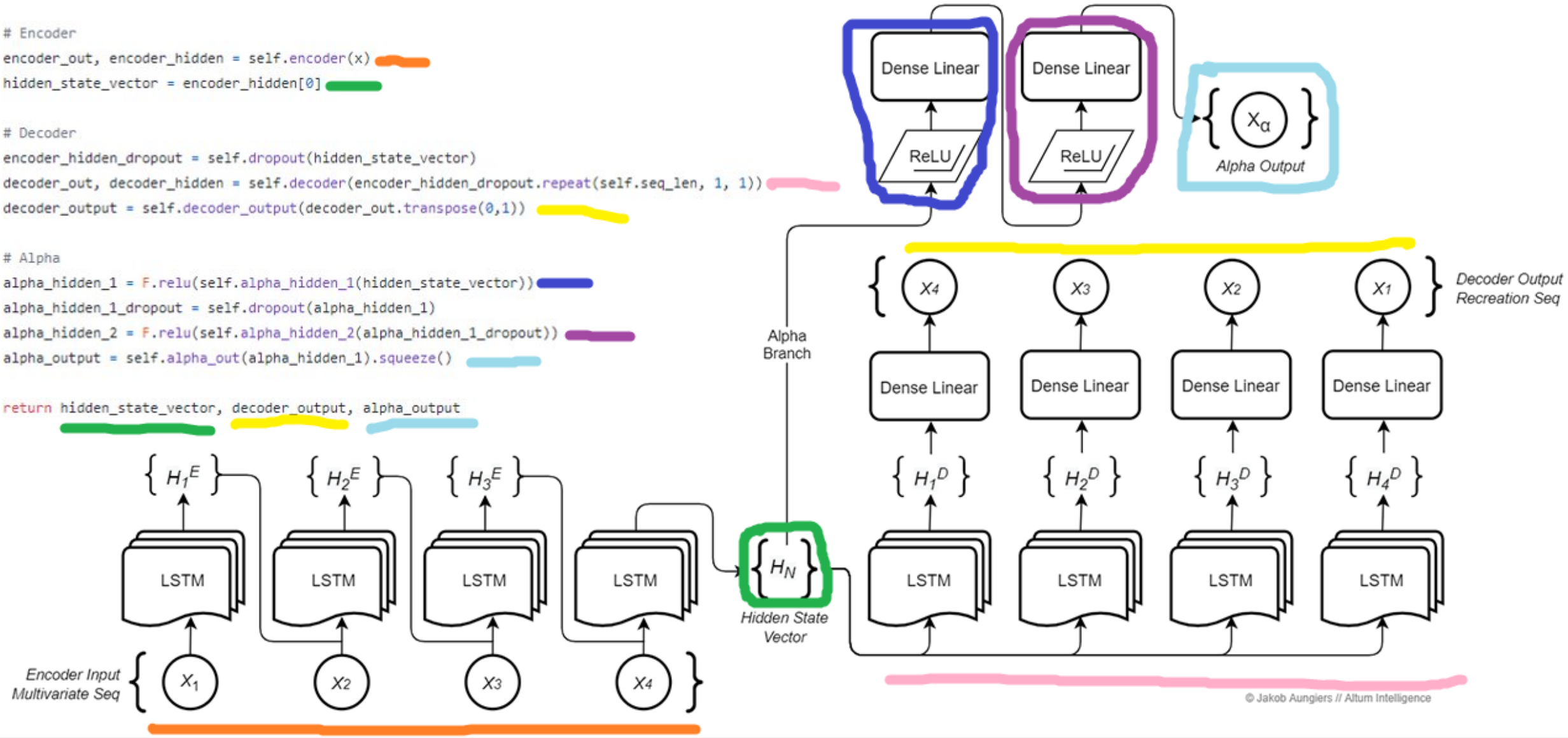
FORWARD METHOD

```
63      def forward(self, x):
64          x = x.to(self.device)
65
66          # Encoder
67          encoder_out, encoder_hidden = self.encoder(x)
68          hidden_state_vector = encoder_hidden[0]
69
70          # Decoder
71          encoder_hidden_dropout = self.dropout(hidden_state_vector)
72          decoder_out, decoder_hidden = self.decoder(encoder_hidden_dropout.repeat(self.seq_len, 1, 1))
73          decoder_output = self.decoder_output(decoder_out.transpose(0,1))
74
75          # Alpha
76          alpha_hidden_1 = F.relu(self.alpha_hidden_1(hidden_state_vector))
77          alpha_hidden_1_dropout = self.dropout(alpha_hidden_1)
78          alpha_hidden_2 = F.relu(self.alpha_hidden_2(alpha_hidden_1_dropout))
79          alpha_output = self.alpha_out(alpha_hidden_1).squeeze()
80
81          return hidden_state_vector, decoder_output, alpha_output
```

SPEED

- Line 64 – Transfers a tensor from *cpu* to *gpu* (if available)
  - `torch.cuda` is used to set up and run CUDA operations. It keeps track of the currently selected GPU, and all CUDA tensors you allocate will by default be created on that device.

```python
# Encoder
encoder_out, encoder_hidden = self.encoder(x)
hidden_state_vector = encoder_hidden[0]

# Decoder
encoder_hidden_dropout = self.dropout(hidden_state_vector)
decoder_out, decoder_hidden = self.decoder(encoder_hidden_dropout.repeat(self.seq_len, 1, 1))
decoder_output = self.decoder_output(decoder_out.transpose(0,1))

# Alpha
alpha_hidden_1 = F.relu(self.alpha_hidden_1(hidden_state_vector))
alpha_hidden_1_dropout = self.dropout(alpha_hidden_1)
alpha_hidden_2 = F.relu(self.alpha_hidden_2(alpha_hidden_1_dropout))
alpha_output = self.alpha_out(alpha_hidden_1).squeeze()

return hidden_state_vector, decoder_output, alpha_output
```



MODEL FITTING

- The PyTorch code for the model fitting process.
- The individual losses for each branch are summed up into a general loss which is then backpropagated.

```
for i in tqdm(range(start_epoch, epochs), disable=not verbose):
self.train()  # set model to training mode
for x_batch, y_batch in data_loader:
    x = x_batch.to(self.device)
    x_inv = x.flip(1) # reversed sequence (dim 1) of x reconstructed on all dimensions
    y = y_batch.to(self.device)

    self.optimizer.zero_grad()
    hidden_state_vector, decoder_output, alpha_output = self(x)

    loss_decoder = self.loss_decoder(decoder_output, x_inv)
    loss_alpha = self.loss_alpha(alpha_output, y)
    loss = loss_decoder + loss_alpha
    loss.backward()

    torch.nn.utils.clip_grad_norm_(self.parameters(), 1.5)
    self.optimizer.step()
```

Forward() method

TQDM

- According to TQDM: "Instantly make your loops show a smart progress meter - just wrap any iterable with `tqdm(iterable)`, and you're done!"
- https://tqdm.github.io/
- https://towardsdatascience.com/progress-bars-in-python-4b44e8a4c482

ZERO OUT THE GRADIENTS

- When using PyTorch, there is a need to set the gradients to zero before starting to do backpropragation because PyTorch accumulates the gradients on subsequent backward passes.

TRAINING

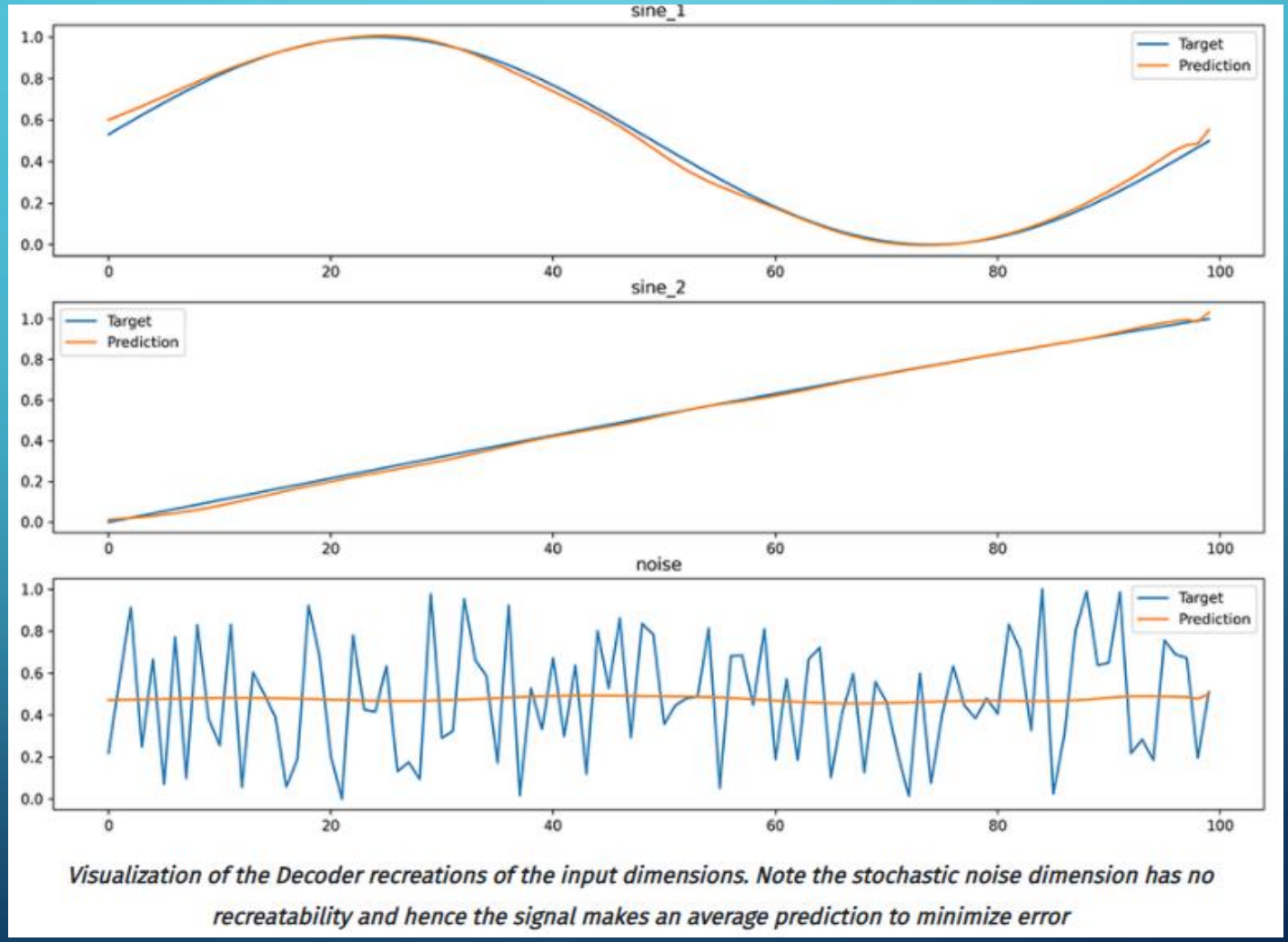
- The EncoderDecoder branch and the Alpha branch are trained using the standard backpropagation algorithm with respect to a mean squared error (MSE) loss function.
- The Adam optimizer function is employed due to the proven optimal convergence of regression problems.

DECODER TARGET RECREATION

```
model.eval()
figsize(15,10)
idx = 0
y_in = tr_input_seq[idx].reshape(1, window_size, tr_input_seq.shape[2])
hidden_state_vector, decoder_output, alpha_output = model(from_numpy(y_in).float())

subplot(3,1,1)
plot(tr_input_seq[idx,:,0], label='Target')
plot(decoder_output[0,:,0].detach().cpu().numpy()[::-1], label='Prediction')
title('sine_1')
legend()

subplot(3,1,2)
plot(tr_input_seq[idx,:,1], label='Target')
plot(decoder_output[0,:,1].detach().cpu().numpy()[::-1], label='Prediction')
title('sine_2')
legend()

subplot(3,1,3)
plot(tr_input_seq[idx,:,2], label='Target')
plot(decoder_output[0,:,2].detach().cpu().numpy()[::-1], label='Prediction')
title('noise')
legend()
show()
```

`model.eval()` can be thought of as a switch for some specific layers/parts of the model that behave differently during training and inferencing time. E.g. Dropouts Layers, BatchNorm Layers etc. These are turned off during model evaluation.

DECODER TARGET OUTPUT GRAPH



Visualization of the Decoder recreations of the input dimensions. Note the stochastic noise dimension has no recreatability and hence the signal makes an average prediction to minimize error

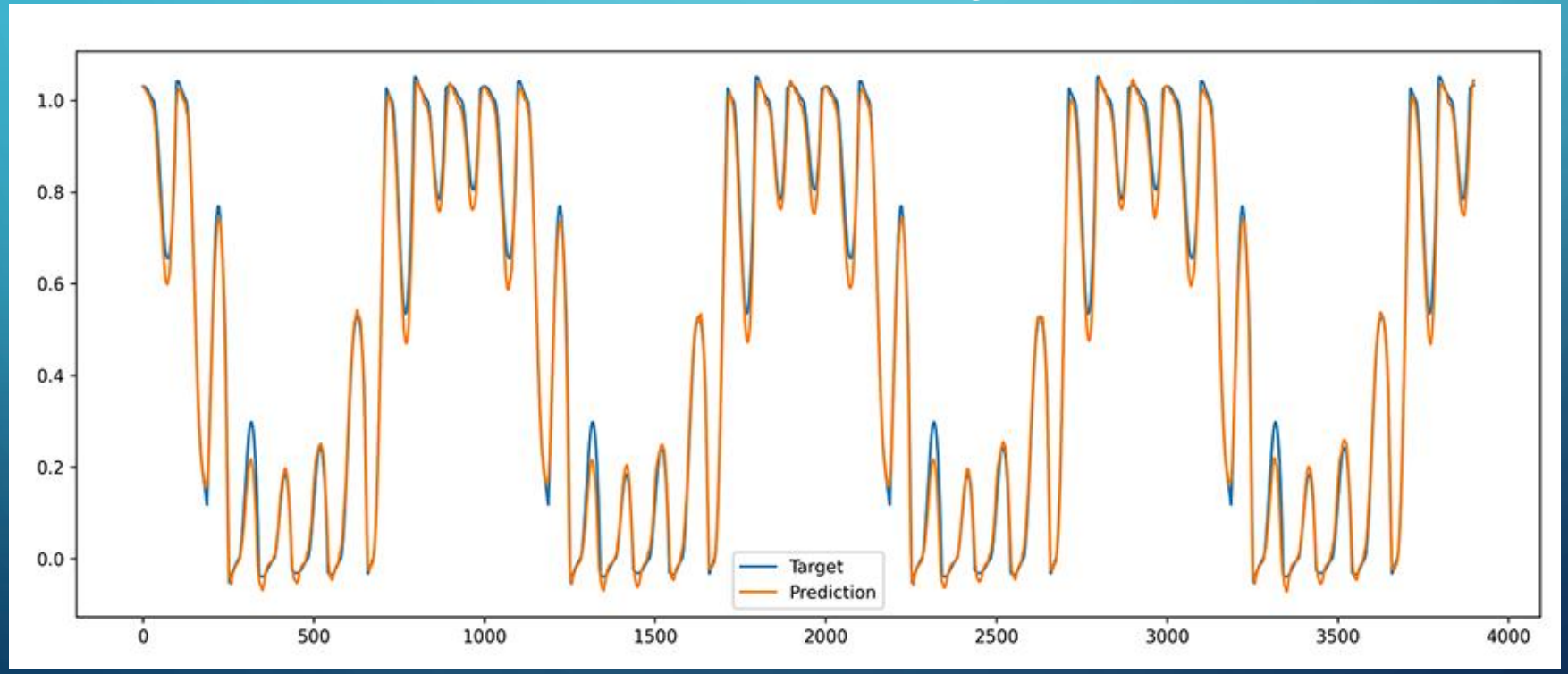
ALPHA TARGET BRANCH PREDICTION

```
model.eval()
figsize(15,6)
_,_ , alpha_output = model(from_numpy(tr_input_seq).float())
alpha_output = alpha_output.flatten().detach().cpu().numpy()

print('### Error/Accuracy Metrics ###')
print('MSE:\t', mean_squared_error(tr_data_windows_y, alpha_output))
print('MAE:\t', mean_absolute_error(tr_data_windows_y, alpha_output))
print('R²:\t', r2_score(tr_data_windows_y, alpha_output))

plot(tr_data_windows_y, label='Target')
plot(alpha_output, label='Prediction')
legend()
show()
```

```
### Error/Accuracy Metrics ###
MSE:     0.0010554455800590521
MAE:     0.023022483376210708
R²:      0.9935462829031128
```

ALPHA TARGET BRANCH PREDICTION GRAPH

- Normalized Predictions vs Normalized Targets



MODEL INSTANCING

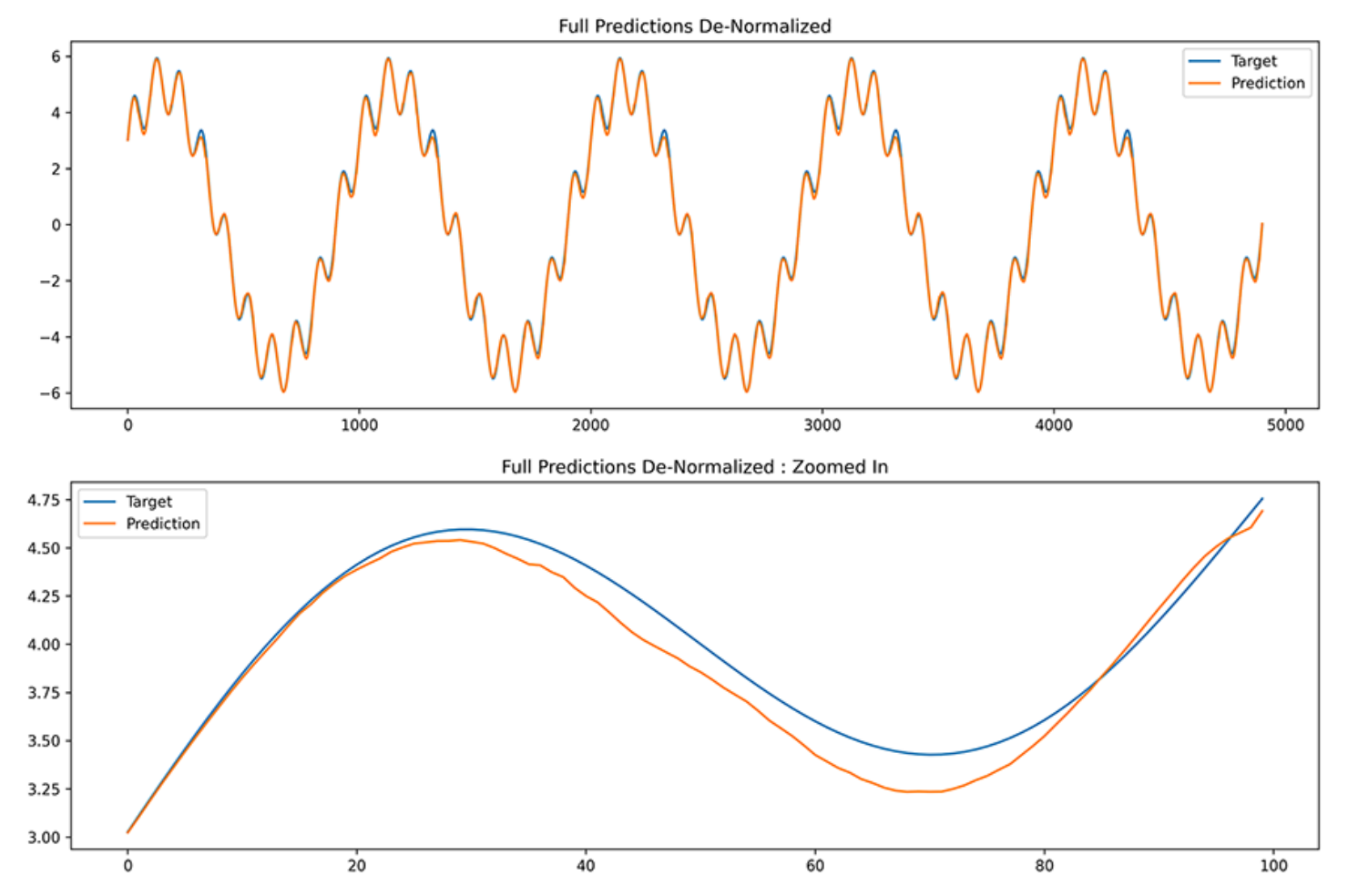
- Produce de-normalized predictions along the original reference data scale:

1. Create a data window on the fly
2. Store the hi, lo values for that normalized data window
3. Leverage the hi, lo values to de-normalize the final prediction output to plot against the original un-normalized data window.

PREDICTION

```
model.eval()
true = []
pred = []
for i in tqdm(range(data.shape[0])):
    if i < window_size:
        continue
    data_window = data[i-window_size:i]
    input_seq = np.zeros((1, window_size, len(features_x)))

    for j, feature in enumerate(features_x):
        _data_window, _, _= norm(data_window[feature])
        input_seq[0,:,j] = _data_window
    _, hi, lo = norm(data_window[feature_y])

    x_hidden_vector, decoder_output, alpha_output = model(from_numpy(input_seq).float())
    abs_pred = reverse_norm(alpha_output.squeeze().detach().cpu().numpy(), hi, lo)

    true.append(data[feature_y][i])
    pred.append(abs_pred)

figsize(15,10)
subplot(2,1,1)
plot(true, label='Target')
plot(pred, label='Prediction')
title('Full Predictions De-Normalized')
legend()

subplot(2,1,2)
zoom = 100
plot(true[:zoom], label='Target')
plot(pred[:zoom], label='Prediction')
title('Full Predictions De-Normalized : Zoomed In')
legend()
show()
```

PREDICTION GRAPH



EXPERIMENTS

- Result accuracy is measured using Mean Squared Error (MSE), Mean Absolute Error (MAE) and an R2 value to measure the correlation between the predictions and targets.
- In each MSE and MAE we look to minimize the error in the first instance and maximize the R2 value in the second instance by tuning the three primary drivers of our model:
  - batch size
  - hidden vector size
  - data window size.
- Many other hyperparameters such as learning rate, activation function values, and neural layer sizes can also be explored, however in these experiments we only show the three drivers mentioned above which were shown to have the greatest varying influence on accuracy and the other hyperparameters are left generally optimized.

RESULTS

- The results of the experiments with show, through a short parameter search along three primary hyperparameters of batch size, hidden vector size and data window size for 100 epochs, that the most optimal of these parameters are: batch size = 8, hidden vector size = 128, window size = 100.
- It is observed that there exist these optimal parameter states below which the full representation of the data cannot be captured and above which the representation is overly complex which leads to instability in accuracy.

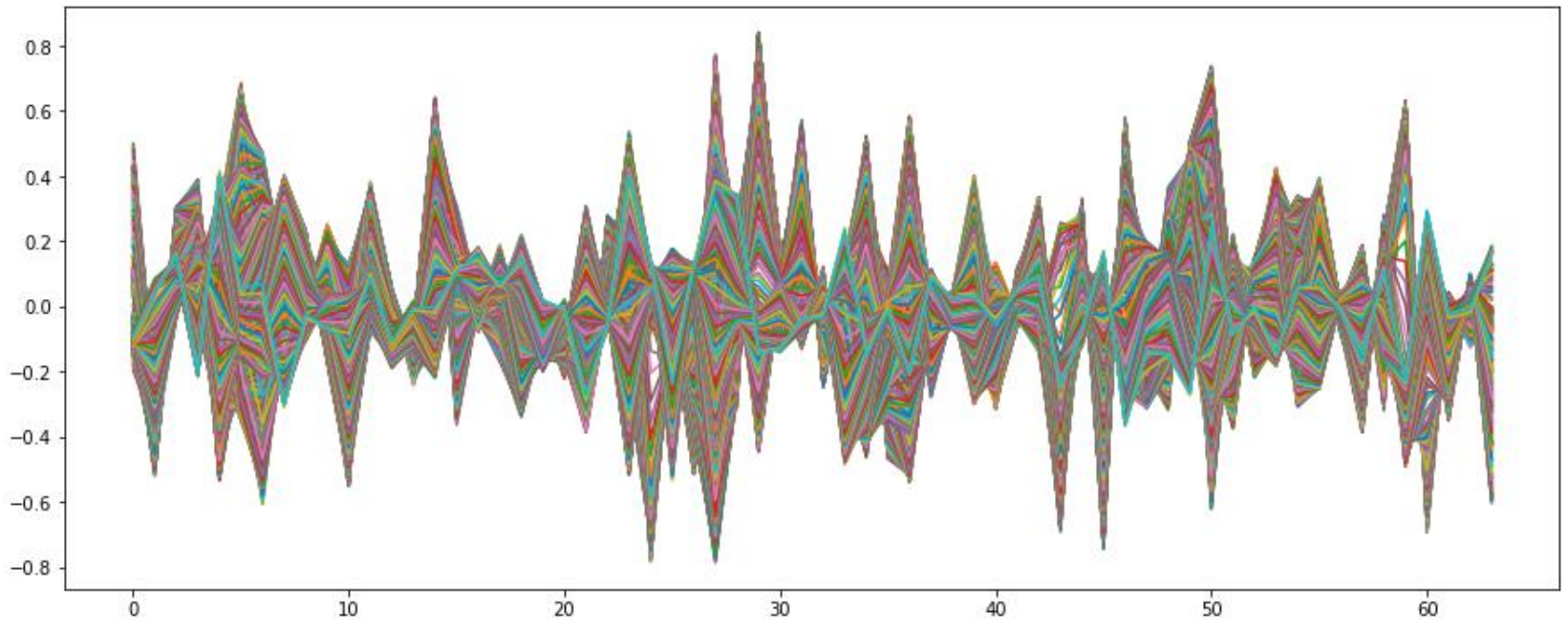
Batch Size	MSE	MAE	R^2
1	0.00552	0.05372	96.62%
2	0.00435	0.04771	97.34%
4	0.00255	0.03621	98.44%
8	0.00165	0.02728	99.01%
16	0.00319	0.04205	98.05%
32	0.00954	0.07259	94.16%
64	0.02436	0.12341	85.10%
128	0.03692	0.16036	77.43%
Hidden Vector Size	**MSE**	**MAE**	**R^2**
8	0.01738	0.09427	89.37%
16	0.00940	0.07003	94.25%
32	0.00393	0.04480	97.60%
64	0.00165	0.02728	99.01%
128	0.00167	0.02967	98.65%
256	0.00323	0.04403	98.02%
512	0.00359	0.04621	97.80%
1024	0.00862	0.07046	94.73%
Window Size	**MSE**	**MAE**	**R^2**
5	0.12368	0.15590	79.58%
10	0.04560	0.10294	88.04%
25	0.01669	0.06574	93.80%
50	0.01242	0.06452	93.93%
100	0.00165	0.02728	99.01%
200	0.00311	0.03918	98.16%
400	0.00165	0.02833	98.99%
800	0.00259	0.03608	97.52%

RESULTS – HIDDEN VECTOR

- Even with a very limited hidden vector size (8) a reasonably accurate data window representation can be created.
- Despite the target signal being composed of 100 sequential steps of multiple dimensions, the representation of the full dimensionality of the data window can be compressed within a hidden state vector of size 8 and still retain 89.37% accuracy.

HIDDEN STATE VECTOR VISUALIZATION

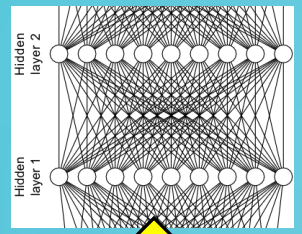
```
model.eval()
figsize(15,6)
zoom_lim = 400 # limit num of hidden vector examples to show
hidden_state_vector, decoder_output, alpha_output = model(from_numpy(tr_input_seq[:zoom_lim]).float())

# visualise hidden vector
for i in range(tr_input_seq[:zoom_lim].shape[0]):
    plot(hidden_state_vector[0, i].detach().cpu())
show()
```



RESULTS - LSTM

- Due to the nature of LSTM cells the temporal memory is reasonably short and has a tendency to decay exponentially for longer term sequences. In this model this effect is dampened through the use of inverse target sequences in the EncoderDecoder branch, however this has the negative effect of diminishing long term dependencies if they exist.

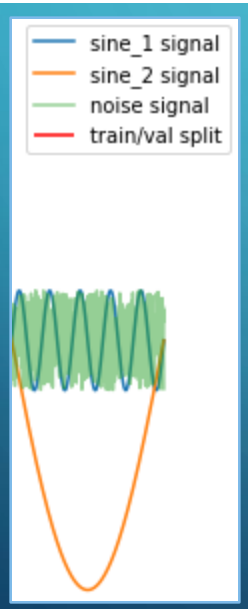
ONE SLIDE



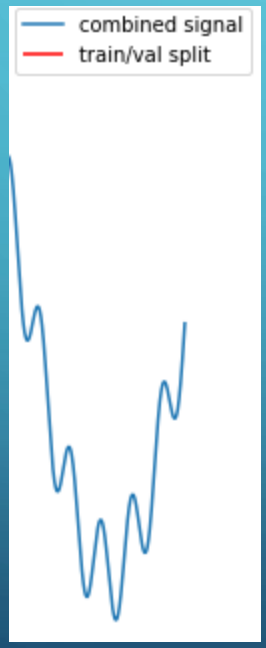
Train a Dense Neural Network

Train on This

noise	sine_1	sine_2
-0.545357	6.279052e-02	3.141572e-02
0.151303	1.253332e-01	6.283020e-02
0.825299	1.873813e-01	9.424220e-02
-0.486808	2.486899e-01	1.256505e-01
0.334807	3.090170e-01	1.570538e-01



Calculate a Loss Function On This



Prediction

Instance Using a 'Window' of Time Series Data

noise	sine_1	sine_2
-0.545357	6.279052e-02	3.141572e-02
0.151303	1.253332e-01	6.283020e-02
0.825299	1.873813e-01	9.424220e-02
-0.486808	2.486899e-01	1.256505e-01
0.334807	3.090170e-01	1.570538e-01

Predict the next value.