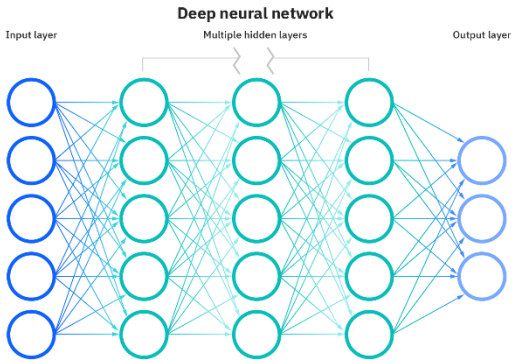
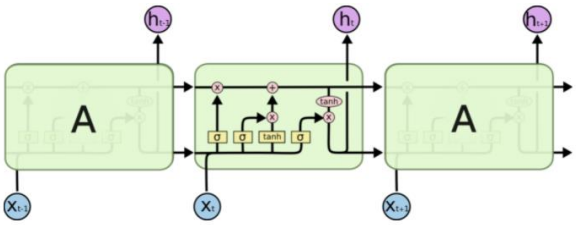
Temperature Prediction Using RNN

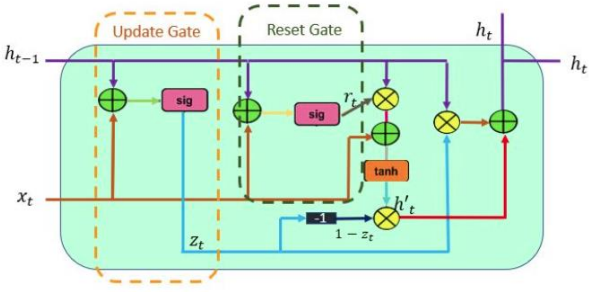
Long-Term Short-Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Transformers

INTRODUCTION TO RECURRENT NEURAL NETWORKS

A neural network is a set of algorithms that tries to recognise underlying correlations in a dataset by mimicking how the brain works. In deep learning, there are three main types of neural network models: ANN, CNN, and RNN. This report deals with using Recurrent Neural Networks to predict 24-hour temperature and modifying different parameters to get the best results. A recurrent neural network is a type of algorithm used to analyse time series data, event history, or temporal ordering in applications including language translation, natural language processing (NLP), and speech recognition. A neural network is made up of three layers: an input layer, a processing layer (hidden layer), and an output layer. RNN is a type in which the output of the previous step is used as the input for the current step. Normally, the inputs and outputs of a neural network are independent, but in some instances, the subsequent output is dependent on the prior input's output. As a result, RNN was developed to remember the previous output and use it as input for the next iteration using a hidden layer. Furthermore, because of the enormous amount of data available, algorithms like LSTM and GRU play a critical role in bringing RNNs to the forefront. A long-term sequence learning RNN is known as an LSTM. It has been meticulously designed to avoid long-term dependency difficulties. It works by remembering big sequences for long periods of time. The LSTM method is substantially more involved; as seen in the accompanying architecture, it accepts input from three different states at any given time: the current input state, the previous cell's short term memory, and finally the long term memory. Before delivering long and short term data to the next cell, these cells use gates to govern whether information is kept or discarded during loop operation. The three gates used by LSTM are the Input Gate, Forget Gate, and Output Gate. These gates function as filters, removing unwanted and irrelevant information. GRU, on the other hand, is very similar to RNN in terms of functionality and the gates that correlate to each GRU cell. The GRU uses a multi-gate method termed the 'Update and Reset Gate' to circumvent the limitations of regular RNNs. The update gate is responsible for determining how much previous data must be passed through to the next state. The reset gate is used in the model to determine how much previous data should be discarded, or whether the previous cell state is significant. LSTM and GRU also differ on conceptual scales. A GRU uses only two gates and has no internal memory, as well as no output gate. In GRU, the input and target gates are taken over by the update gate, in addition to the direct application of the reset gate to the previous concealed state. This guarantees the model's speed.

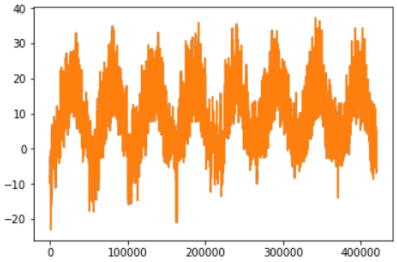


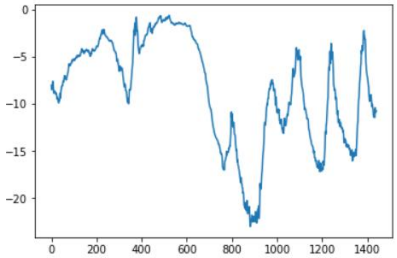




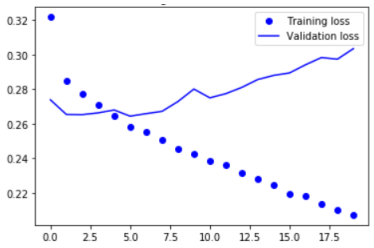
TEMPERATURE PREDICTION USING RECURRENT NETWORKS

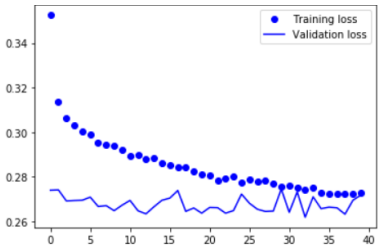
The report primarily deals with the implementation of a 24-hour temperature predictor using LSTM and GRU. The prediction becomes possible given a time series dataset, consisting of 14 different features such as air temperature, atmospheric pressure, humidity, wind direction, etc. These attributes are recorded every 10 minutes, over a period of time. The data we use in building the model is the recordings from 2009-16. Being a numerical time series dataset, it is ideal to work with and, the model being built needs to make sure to take some data as input from recent past, and predict the air temperature 24 hours in the future. Initially, after looking into the data, it needs to uncompressed. There are 420,551 lines of data, each line being a time stamp, with a date and 14 climate-associated features. The entire set of timestamps needs to be converted into NumPy arrays. Following the conversion, yearly and daily periodicity of temperature data can be noticed on the line plots. The plot for the initial 10 days is slightly tapered, owing to a supposedly colder month, in addition to the date being recorded every 10 minutes, giving 144 data points or timestamps per day. The temperature looks much messy based on daily periodicity but the purpose here is to conduct a time series prediction on a daily scale which would be challenging in comparison to the monthly temperature prediction based on yearly periodicity. For the purpose of building the models, 60% of the dataset will be used for training, 20% of the remaining data for testing and validation each, respectively. Ideally, using the latest data for validation and testing is pivotal when dealing with time series, for better future prediction given the past. To get started data needs to be pre-processed, however, vectorization can be skipped since the data is already numerical. Still, data needs to be normalized independently to take smaller values on a single scale, since each time series data is on a distinct scale. A python generator needs to be compiled to yield a set of data from the latest recordings, by using a current array of raw data. Generating the samples on the go using the original data would be better, instead of explicitly allocating every sample as most of their time stamps are same. We divide the data by the standard deviation after subtracting the mean of each time series. We only compute the mean and standard deviation on the first 60% time steps because we aim to use these as training data. We'll use abstract generator function to incorporate 3 datasets, for training, validation and testing. The dataset generator yields a tuple (samples, targets), where samples is one batch of input data and targets is the corresponding array of the target temperature, using several parameter values like *sampling_rate* (sampling of data as 1 data point per hour), *sequence_length* (observations will go back 5 days), *delay* (targets will be 24 hours in the future). Before we conduct deep-learning models of RNN (LSTM and GRU), we could build a model using basic ML algorithms. During the construction of various models, we would use Mean Absolute Error (MAE) as a metric to evaluate the models. Firstly, a basic machine learning model can be examined, that uses dense layers, can help identify complexities involved. In this approach, we could use Mean Squared Error (MSE) as the loss, instead of MAE, which is quite useful for gradient descent. This model uses the properties of a regression function, resembled through the activation function absence in the last dense layer. A test MAE of 2.64 is rendered





from this approach and it can be noticed that some of the validation losses close to the no-learning baseline, but not quite reliably. Furthermore, the 1D convolutional layer produced an MAE of 3.13, way redundant in comparison to the previous approach. This method relies on 1D windows that slide across the input sequences and cubic windows that slide across input volumes. Starting with an initial window length of 24 hours, we decrease this size as we go down layers. The densely connected approach flattens the time series and the convolutional approach on the other hand obliterated the sequence data, leaving behind a poor performance of their models. Regardless, a recurrent baseline approach is ideal for our problem, as we have a sequence data where causality matters and temporal ordering of data points is not disturbed. As a part, an LSTM layer can be applied, which significantly defeats the previous two models with an MAE of 2.58, as well as depicting the superiority of recurrent networks compared to sequence-flattening dense networks on this type of task. However, the model faces the problem of overfitting at the first instance. The divergence between training and validation losses significantly rises after a few epochs. A traditional overfitting fighting technique, dropout, can be used which randomly destroy the correlations in the training data and removes the input units that layer is exposed to. There are two types of





arguments – dropout, a floating integer, determining the rate at which input units of layer drop and recurrent dropout, determining the dropout rate of recurrent units. The MAE of the LSTM model after applying the dropouts is 2.53, resulting in no overfitting in the first 10 epochs. Since, the obstacle of overfitting is dealt with, we can increase the capacity of our network, by stacking recurrent layers on top of each other in Keras, mostly importantly, this can achieved by specifying *retun_sequences = True.* In this approach, a GRU layer can be used, for a stack of two dropout optimised layers. The model achieves a MAE of 2.42, a subtle improvement. Theoretically, GRU is much sophisticated and executes faster as it uses less training parameters, which is ideal when the memory consumption needs to be less. However LSTM has the benchmark when dealing with large sequences and accuracy is concerned. All the models have fetched competent results, but the extremities of different layers can be tested by conducting several experiments using various tricks/techniques on certain impactful factors of the models. These tweaks display the evident changes in test MAEs of the LSTM and GRU layers. For this purpose, we have decided to change the dropout values, recurrent dropout values, sequence length, mixing of LSTM and GRU layers, and the split of the data used for training, validation and testing. Initially, the data split used in the original model

```
num_train_samples = int(0.47568 * len(raw_data))
num_val_samples = int(0.23784 * len(raw_data))
num_test_samples = len(raw_data) - num_train_samples - num_val_samples
```

```
sampling_rate = 6
sequence_length = 240
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256
```

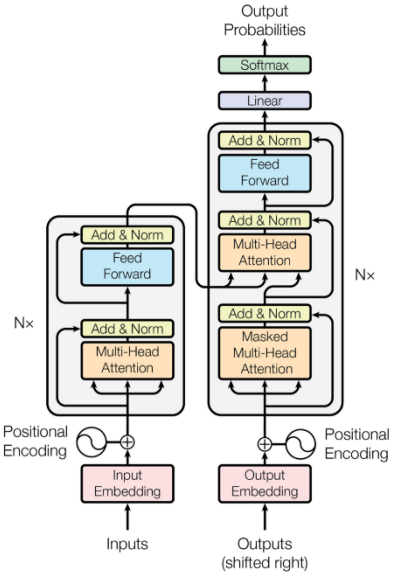
has been altered to 47.57% (200,000) for training, 23.78% (100,000) for validation and rest (120551) for testing. Secondly, the length of sequence has been changed to 240 (obvs. go back to 10 days), instead of the length in the original model, i.e., 120 (obvs. go back to 5 days). Thirdly, the dropout values have been changed to notice the changes in the MAE owing to the overfitting

```
x = layers.LSTM(32, recurrent_dropout=0)(inputs)
x = layers.Dropout(0.25)(x)
```

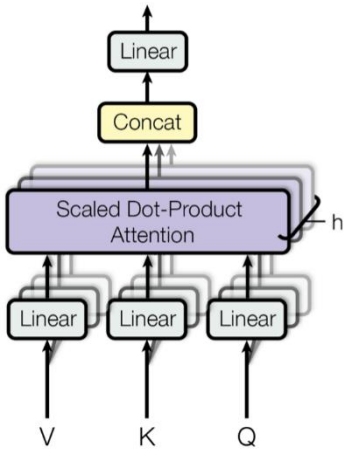
```
x = layers.GRU(32, recurrent_dropout=0, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0)(x)
x = layers.Dropout(0.25)(x)
```

problem. In both the LSTM and GRU layers, the value of recurrent dropout = 0, decreased by 0.25 and dropout = 0.25, decreased by 0.25 as well. After conducting all the tricks and altering the parameters, the LSTM and GRU models achieved a 2.61 and 2.46 MAE respectively, without recurrent drop out. The models comparatively took longer execution time with not much significant change in the values.

TRANSFORMER-BASED TEMPERATURE PREDICTION



A transformer is a deep learning algorithm that uses the self-attention methodology to independently evaluate the significance of each element of the incoming data. For popular natural language processing (NLP) and computer vision (CV) tasks, this neural network technique has proven to be effective. RNN-style transformers are designed to process sequential input data and have a wide range of applications in translation and text summarization. Transformers employ a simple technique known as 'neural attention,' which could serve as a reliable sequence model builder without the usage of recurrent or convolutional layers. Transformers employ an attention mechanism that examines an input sequence and determines which parts are essential at each step. Transformer, like LSTM, is an architecture for converting one sequence into another using two parts: encoder and decoder. It is possible that only attention processes, not RNNs, can improve results in many NLP tasks. Different types of attention start with calculating the important scores for a set of features, with higher scores for more relevant items and lower scores for less relevant ones. The representation of a token is modulated by self-attention by employing the representation of related tokens in the sequence. This results in token representations that are context aware. The value of the relevant query vectors is provided by *Attention (Q, K, V) = V.softmax (score (Q, K)* in general, given Q (query), K (key), and V (value). *Q = K = V* is the formula for self-awareness. i.e., by comparing all vectors, we compute a new value for each vector. Multi-head attention, on the other hand, combines the knowledge discovered by numerous heads rather than just one, as is the case with traditional attention. Multi-head is a variation on the self-awareness process. The self-attention layer's output space is divided into a collection of distinct sub-spaces that are learned separately, with the initial query, key, and value processed through three different sets of dense projections, resulting in three different vectors. Natural attention processes all three vectors, resulting in three outputs that are concatenated back into a single output sequence. Each subspace is referred to as a head. The transformer's entire architecture is built around two crucial components, the transformer encoder being the most important for our model. The encoder is crucial because it is used for text



categorization, a generic module that takes a sequence and learns how to transform it into a more useful representation. With a few tweaks to the metrics, we could utilise an encoder to solve our prediction problem. The transformer model, like any other ML or DL technique, necessitates data pre-processing, and the amount of data divided for training, validation, and testing is 50 percent, 25%, and 25%, respectively. Additionally, the data must be normalised to ensure that it is organised and appears on the same scale for all aspects. Unlike traditional classification transformers, our prediction model requires a function defined generator with the same parameter values as RNN models. *Sequence_length = 120, delay = 6 * (sequence_length + 24 – 1), batch_size = 256*. Our model works with a form tensor (batch size, sequence length, features), where sequence length refers to the number of time steps and features refers to each input time series. Multi-head attention is used in the encoder layer, which is followed by residual connections, the normalisation layer, and dropouts. Multiple encoder blocks and the generated layers can be stacked together. Apart from the dense layer stack, the encoder's output tensor must be decreased. As a pooling layer, the build model employs a GlobalAveragePooling1D, which is sufficient. *LayerNormalization* also pools data inside each sequence independently, which is more suitable for sequence data. After the multi-head attention layer, a fully connected feed-forward network with two linear transformations and ReLU activations is used. After that, a second residual connection and normalising layer is added. The input shape, head size, num_heads, ff_dim, num_transformer blocks, and drop out parameters define the build model. The model is evaluated using MSE as the loss, rmrsprop as the optimizer, and MAE as metrics over 20 epochs with a batch size of 32. Although the transformer algorithm produced an MAE of 2.48, which is substantially competent with RNN layers, a transformer model is not practical for predicting and is well suited for classification tasks, in addition to being expensive and time-consuming. The prediction problem is best served by LSTM and GRU.