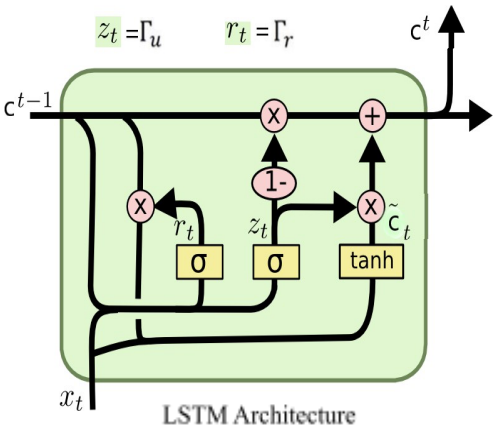
24-hour temperature prediction using

Recurrent Neural Networks and Transformer

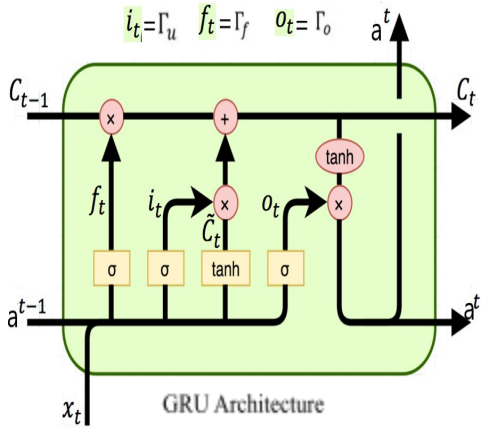
Understanding of RNN

Recurrent Neural Networks are built to handle sequential data. Sequential data (which can be time series) can take the form of text, audio, video, and so on. RNN generates the current output by utilizing prior information in the sequence. I'll provide an example sentence to help you understand - How are you?. The initial step is to feed the word "How" into the network at the time (T0). The RNN generates an output. At the time (T1), we supply the word "are" and the activation value from the previous phase. The RNN now includes information on both the terms "how" and "are." This method is repeated until all words in the phrase have been supplied. The RNN now contains information about all of the preceding words. The major issue is that RNNs find it difficult to train to store information across numerous timesteps. The hidden state in vanilla RNN is continually rebuilt. RNNs have a short-term memory issue. RNN suffers from vanishing gradient more than other neural network designs as it processes more steps.



LSTM Architecture

Long Short-Term Memory: The purpose of LSTMs is to overcome the vanishing gradient problem. LSTM networks are well-suited to categorizing, analyzing, and forecasting time series data. The memoizing process is controlled by a gating mechanism in LSTMs. LSTMs store, write and read information via gates that open and close. These gates store memory in analogue format, implementing element-wise multiplication with sigmoid ranges ranging from 0 to 1. There are 3 gates - Input gate, Forget gate and Output gate.



GRU Architecture

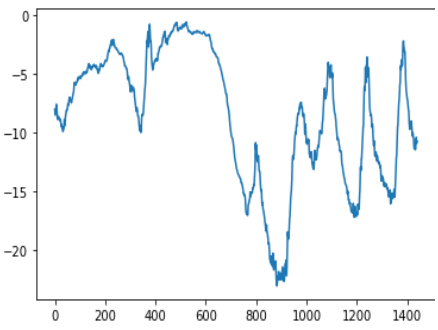
Gated Recurrent Units: The GRU, like the LSTM unit, includes gating units that influence the flow of information inside the unit, but it lacks distinct memory cells. The workflow of GRU is similar to that of RNN, however the distinction is in the operations performed within the GRU unit. GRU is unique in that it can be trained to retain knowledge from long ago without washing it away with time or deleting information that is unrelated to the prediction. In GRU, there are 2 gates - Reset gate and Update gate.

Dataset

Throughout several years, fourteen distinct values were recorded in this dataset. This dataset is ideal for learning how to work with numerical timeseries. We will use it to create a model that takes data from the past as input and predicts the temperature 24 hours in the future.

24-hour temperature prediction using LSTM and GRU

Data Preprocessing: The **layer is the fundamental building element of neural networks**; it is a data-processing module that may be thought of as a **data filter**. Each timeseries in the data is scaled differently. As a result, we shall **normalize each timeseries** separately such that they all accept tiny values on a comparable scale. We **preprocess the data by dividing the standard deviation by the mean of each timeseries**. We want to **use 50% of the initial dataset as training data**, so we compute the **mean and standard deviation** on that portion of the data. **Using keras.timeseries_dataset_from_array**, we can extract **batches of data** from the recent past as well as a **target temperature** in the future from our present array of float data.



Because the **samples in our dataset are substantially redundant** (eg, sample N and sample N+1 will have the majority of their timesteps in common), manually allocating every sample would be quite inefficient. Instead **we create the samples** on the fly **using the original data**. It returns a tuple (samples, targets), where **samples is a batch of input data and targets is an array of target temperatures**. We will use this to create **three generators: one for training** (50%), **one for validation** (25%), **and one for testing** (25%).

Training Optimizations

Before we use pricey machine learning models to tackle our temperature forecast problem, let's attempt a **basic common-sense method and inexpensive machine learning models (such as small densely-connected networks).** It will **act as a sanity check and provide a baseline** for more complex and costly machine learning models like **RNNs**. We can greatly **outperform the common sense baseline** with a **basic LSTM or GRU** model. Here, **training and validation curves show that our model is overfitting**, the training and validation losses begin to diverge significantly after a few epochs.

Dropout: Regularization method to overcome overfitting. It randomly **zeroes off a layer's input units** in order **to disrupt accidental correlations in the training data** to which the layer is exposed. Instead of a dropout mask that varies randomly from timestep to timestep, the **dropout mask is applied at each timestep**. Used **layers.Dropout to regularize the outputs** created by the recurrent gates of layers such as GRU and LSTM, a **temporally constant dropout mask was added to the layer's inner recurrent activations** which permits the network to appropriately transmit its learning error across time; a temporally **random dropout mask**, on the other hand, would disrupt this error signal and **damage the learning process**.

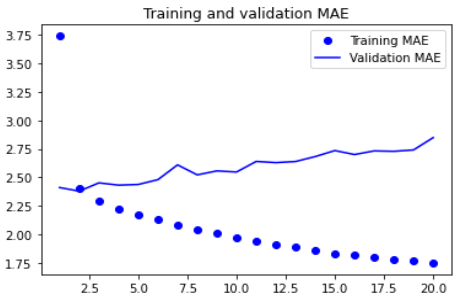
Stacking: Increasing network capacity is accomplished **by increasing the number of units in the layers** or by adding more levels. To stack recurrent layers, all intermediary **layers** should **return their series of**

outputs (**a 3D tensor**) rather than their output at the final timestep. **return_sequences=True** is used to do this in the LSTM or GRU layer. We can observe that the **additional layers enhance our findings slightly**, but not considerably. We could **safely raise the size of our layers in search of a bit of validation loss improvement because we are still not overfitting** too significantly.

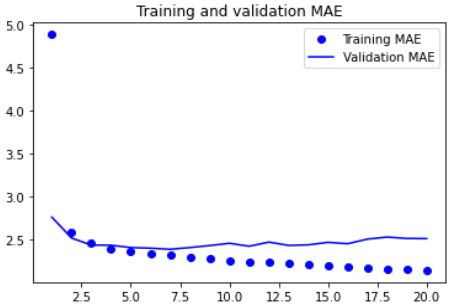
Bidirectional RNNs: It takes **use of RNNs' order sensitivity**. It is just **two conventional RNNs layers,** such as the GRU or LSTM, **processing the input sequence in one direction (chronologically and anti-chronologically) before combining their representations**. Using **layers. Bidirectional** helps **detect patterns** that a one-direction RNN may have missed **by analyzing a sequence both ways**.

Layer Normalization: Normalization technique for intermediate layer distributions. It allows for **smoother gradients, faster training, and accurate generalization**. It **calculates the normalization statistics directly from the summed inputs to the neurons inside a hidden layer**, thus the normalization introduces no additional dependencies across training examples. Used **layers. LayerNormalization normalizes activations in the direction of the feature** rather than the direction of the mini-batch and **normalizes each characteristic of the activations to zero mean and unit variance**.

Model Compilation: As part of the compilation stage, we need to choose three additional items to prepare our network for training: (A) **Loss Function (MSE):** How the network will be able to assess how well it is performing on its training data and so steer itself in the proper direction. (B) **Optimizer (RMSprop):** Technique that the network will use to update itself depending on the data it observes and its loss function. (C) **Metrics (MAE):** To keep an eye on things throughout training and testing. We will be concerned with mean absolute error in this case. Using different parameters, we got different MAEs and **lowest MAEs of 2.48** on 20 epochs.



Worst MAE - 2.83

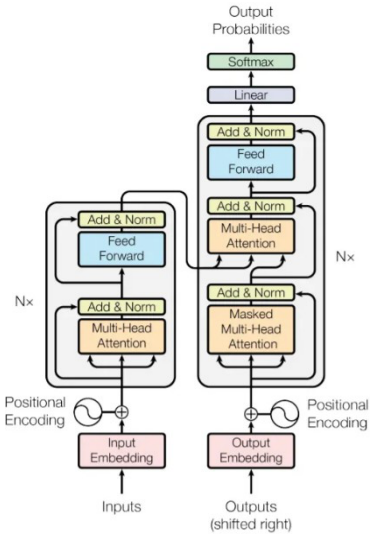


Best MAE - 2.46

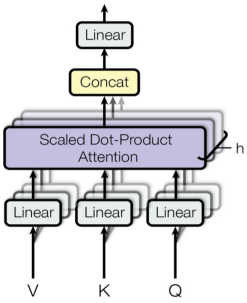
Model	MAE
Simple LSTM	2.58
Dropout-regularized LSTM	2.57
Layer normalized LSTM	2.58
Stacked LSTM	2.56
Bidirectional LSTM	2.75
Simple GRU	2.48
Dropout-regularized GRU	2.49
Layer normalized GRU	2.50
Stacked GRU	2.46
Bidirectional GRU	2.83

How Transformer Works

Despite RNNS being so good at what it does, there are limitations: Dealing with long-range dependencies and the sequential nature of the model architecture prevents parallelization. The Transformer design tries to **address sequence-to-sequence challenges while also dealing with long-term interdependence**. Transformer was proposed in the paper **Attention Is All You Need**. **Transformers are used to process sequential data**. RNNs are outperformed by transformers because they are not recurrent and may exploit prior time step characteristics with no information loss. One significant advantage of the transformer is that we have **direct access to all of the previous processes at each step (self-attention), which almost eliminates information loss**. Furthermore, we may **examine both future and past components at the same time**, which provides the benefit of bidirectional RNNs **without the 2x computation**. And, all of this **occurs in parallel (non-recurrent), making training faster**. The transformer design **consists of an Encoder and a Decoder, both of which contain Multi-Head Attention modules and FeedForward modules**. The model concatenates with results and is normalized to optimize training. The attention module and feedforward module sublayers are then repeated N times for both Encoder and Decoder.



A transformer model's central **principle is self-attention**, or the **capacity to attend to multiple locations in an input sequence in order to calculate a representation of that sequence**. Instead of RNNs, a transformer model **handles variable-sized input by stacking self-attention layers**. Transformer architecture has a number of advantages: (A) It makes **no assumptions about the data's temporal connections**. This is **perfect for dealing with a collection of feature objects**. (B) In contrast to an RNN, **layer outputs can be computed simultaneously**. (C) **Distant objects can influence each other's output** without having to go through a lot of RNN steps. (D) It is capable of **learning long-term dependence**. **Downside** is the **output for a time step** in a time-series is **calculated using the complete history** rather than inputs and present hidden state.



Multi-head Attention: Multi-head attention mechanism is the transformer's main component. The encoded representation of the input is seen by the transformer as a **collection of key-value pairs**, both the **keys and values are encoder hidden states**. The previous **output is compressed into a query in the decoder**, and the next **output is generated by mapping this query and the collection of keys and values together**. The **scaled dot-product attention is used by the transformer**: the output is a weighted sum of the values, with the **weight assigned to each value decided by the dot-product of the query with all the keys**. The **multi-head mechanism passes through the scaled**

dot-product attention several times in parallel, rather than just once. Simply, **concatenates the separate attention outputs and linearly translates them into the projected dimensions**. The **model attends to input from distinct representation subspaces at different points using multi-head attention**. Each multi-head attention block gets **three inputs; Q (query), K (key), V (value)**. These are put through linear (Dense) layers before the multi-head attention function. Instead of one single attention head, **Q, K, and V are split into multiple heads** because it allows the model to jointly **attend to information from different representation subspaces** at different positions.

Model Training: Using dataset and data preprocessing, same as above, we implemented a transformer based forecasting approach. Used tensorflow.keras library to implement layers. Used **multi-head attention layers to split information into query, key and value**. Used **layers.MultiHeadAttention to implement multi-head attention**. While **each attention head attends to the feature sets that are relevant to each target**, the model may accomplish this for different definitions of relevance with numerous attention heads. **Multi-head attention network creates output vectors along with layer normalization (layers.LayerNormalization) and dense layers (layers.Dense)**. Also, using **Conv1D layers (layers.Conv1D) and dropout (layers.Dropout) in feed forward network to convolve a vector into a shorter vector and prevent overfitting**. Here, we use a transformer encoder to fulfill the multi-head attention for the forecasting training. Each **transformer encoder is made up of two key parts: a self-attention mechanism and a feed-forward neural network**. To create output encodings, the **attention mechanism receives input encodings from the preceding encoder and balances their importance to each other.** Each **output encoding is then processed independently by the feed-forward neural network**. These o**utput encodings are then passed to the next encoder as its input**. Used **layers.GlobalAveragePooling1D on each feature dimension to average among all time steps**. The output shape you get is equal to (batch_size, features). Used **ELU as activation function** on dense and convolutional layers which **handles negative values which allows them to push mean unit activations closer to zero**. **Built this model and compiled using Adam optimizer, loss function as MSE and metric as MAE**. Achieved **lowest MAE of 2.43** on test data on 20 epochs. Multi-head attention based transformers perform better than LSTM and GRU and are faster in training.

```
Epoch 18/20
1150/1150 [==============================] - 168s 146ms/step -
Epoch 19/20
1150/1150 [==============================] - 168s 146ms/step -
Epoch 20/20
1150/1150 [==============================] - 168s 146ms/step -
245/245 [==============================] - 14s 55ms/step - loss
Test MAE: 2.43
```