MagNet Competition

Belinda Trotta

February 15, 2021

1 Folder contents

- `train.py` Training code. (This has been modified slightly from the version I used to train my submission: I removed some testing code, and added some more explanatory comments. However the code that produces the trained models is unchanged.)
- `predict.py` Prediction code.
- 10 tensorflow model files in `.h5` format, generated by `train.py`.
- `norm_df.csv` Median and quartiles for the model features, used to normalise the data before prediction, generated by `train.py`. Required for submission.
- `requirements_gpu.txt` Requirements for running the training code on a GPU machine.
- `requirements_cpu.txt` Requirements for running the training code on a CPU machine.
- `Readme.pdf` This file. Contains instructions for running the code to reproduce my submission. Also includes the model documentation, with detailed description of the modelling approach.
- `LICENSE.txt` MIT license.

2 Reproducing the submission

The code can be run on GPU or CPU. I used GPU for training my submission, but CPU takes only slightly more time (16 minutes vs 11 minutes) and is probably easier to reproduce, so I recommend CPU for reproducing the submission. I've tested that the instructions below for CPU can be used to build a working environment starting with a plain Ubuntu instance on AWS (in contrast, for the GPU training, I started from a pre-built instance with the Python environment already set up and I have not tested building the environment from `requirements_gpu.txt`).

2.1 Environment for CPU (recommended)

Training on CPU takes around 16 minutes.

The following instructions can be used to set up the Python environment, starting from a basic Ubuntu Linux install. I tested these instructions on an AWS Ubuntu Linux cloud instance (instance type t2.2xlarge).

1. Download and install Anaconda:

   ```
   wget https://repo.anaconda.com/archive/Anaconda3-2020.11-Linux-x86_64.sh
   bash Anaconda3-2020.11-Linux-x86_64.sh
   ```

2. Create a new conda environment and activate it:

   ```
   conda create --name tf python=3.8
   conda activate tf
   ```

3. Install `gcc` if not already installed:

   ```
   sudo apt-get install gcc
   ```

4. Install the pip packages from the requirements file:

   ```
   python -m pip install -r requirements_cpu.txt
   ```

2.2 Environment for GPU

I used an AWS Ubuntu Linux cloud instance (instance type p2.xlarge). I used a pre-built image with the Python environment already set up (but I think I also changed some package versions). Packages are in `requirements_gpu.txt`. Training takes around 11 minutes.

2.3 Running the code

The input `csv` files should go in a subdirectory called `data`. To train the models run `train.py`, which generates 10 model files and `norm_df.csv`. The files needed to submit an entry are `predict.py`, the 10 model files, and `norm_df.csv`.

3 Model documentation and write-up

3.1 Bio

I'm a senior software engineer at the Bureau of Meteorology in Australia, working on a system for post-processing weather forecasts. In my spare time I'm a maintainer for the machine learning package LightGBM. I have a PhD in pure mathematics from La Trobe University.

3.2 Motivation

I've been competing in machine learning competitions for a few years, to sharpen my skills and for fun. My favorites are the science-themed competitions like this one. I like learning a little about a new subject area, and I hope that I can contribute in a small way to scientific progress.

3.3 Approach

3.3.1 Features used and preprocessing

I used only the solar wind and sunspots data; I found that the satellite data didn't help my model. From `solar_wind.csv`, I used the following columns: `bt`, `density`, `speed`, `bx_gsm`, `by_gsm`, `bz_gsm`. The features `temperature` and `source` did not seem to improve the model. I used the data in the (x, y, z) coordinate system, rather than the angular coordinate systems, because it would be harder to calculate mean and standard deviations for angle data in degrees.

I excluded from training periods where the temperature is < 1. Missing data is filled by linear interpolation (to make the result less noisy, we interpolate using a smoothed rolling average, rather than just the 2 points immediately before and after the missing part).

Data is normalized by subtracting the median and dividing by the inter-quartile range (I used this approach rather than the more usual mean and standard deviation because some of the variables have asymmetric distributions with long tails).

I aggregated the training data in 10-minute increments, taking the mean and standard deviation of each feature in the increment. This could have alternatively been done as the first layer of the neural network, but it's faster to do it as a preprocessing step.

3.3.2 Neural network model

The model is a convolutional neural network with rectified linear activations. It uses tensorflow with the keras framework.

The model consists of a set of layers which apply convolutions to detect patterns at progressively longer time spans. At each convolutional layer (except the last), a convolution is applied having size 6 with stride 3, which reduces the size of the output data relative to the input. (The last convolution has small input size, so it just convolves all the 9 input points together.) Thus the earlier layers recognize lower-level features on short time-spans, and these are aggregated into higher-level patterns spanning longer time ranges in the later layers. Cropping is applied at each layer which removes a few data points at the beginning at the sequence to ensure the result will be exactly divisible by 6, so that the last application of the convultional filter will capture the data at the very end of the sequence, which is more important to the prediction.

After all the convolutional layers is a layer which concatenates the last data point of each of the convolution outputs (acheived by left-cropping all but the last point). This concatenation is then fed these into a dense layer. The idea of taking the last data point

of each convolution is that it should give a representation of the features at different timespans leading up to the prediction time. For example, the last data point of the first layer gives the features of the hour before the prediction time, then the second layer gives the last 6 hours, etc.

3.3.3 Ensembling

I trained an ensemble of 5 models, by excluding 20% of the training data each time. This was done by splitting the training data into months and randomly selecting 20% of the months to hold out. I split by months rather than hours because subsequent hours are likely to be correlated, meaning test and train sets would be more similar, reducing the benefit of the ensemble. The prediction code averages the output of the 5 models.

I trained a separate set of models for time t and $t+1$, so there are 10 models in total.

3.4 Important parts of the code

- The code of the neural network model. The code is quite simple and short, and achieves good results without the need for much feature engineering.

```
inputs = tf.keras.layers.Input((6*24*7, 13))
conv1 = tf.keras.layers.Conv1D(50, kernel_size=6, strides=3,
    activation='relu')(inputs)
trim1 = tf.keras.layers.Cropping1D((5, 0))(conv1)
conv2 = tf.keras.layers.Conv1D(50, kernel_size=6,
    strides=3, activation='relu')(trim1)
trim2 = tf.keras.layers.Cropping1D((1, 0))(conv2)
conv3 = tf.keras.layers.Conv1D(30, kernel_size=6,
    strides=3, activation='relu')(trim2)
trim3 = tf.keras.layers.Cropping1D((5, 0))(conv3)
conv4 = tf.keras.layers.Conv1D(30, kernel_size=6,
    strides=3, activation='relu')(trim3)
conv5 = tf.keras.layers.Conv1D(30, kernel_size=9,
    strides=9, activation='relu')(conv4)
# extract last data point of previous convolutional layers
comb1 = tf.keras.layers.Concatenate(axis=2)([conv5,
    tf.keras.layers.Cropping1D((334, 0))(conv1),
    tf.keras.layers.Cropping1D((108, 0))(conv2),
    tf.keras.layers.Cropping1D((34, 0))(conv3),
    tf.keras.layers.Cropping1D((8, 0))(conv4)])
dense = tf.keras.layers.Dense(50, activation='relu')(comb1)
output = tf.keras.layers.Flatten()(
    tf.keras.layers.Dense(1)(dense))
model = tf.keras.Model(inputs, output)
```

- Filling missing training data. This is important because there are many gaps in the data. To make the result less noisy, we interpolate using a smoothed rolling average, rather than just the 2 points immediately before and after the missing part

```
solar['month'] = solar['month'].fillna(method='ffill')
train_cols = ['bt', 'density', 'speed', 'bx_gsm', 'by_gsm',
    'bz_gsm', 'smoothed_ssn']
train_short = [c for c in train_cols if c != 'smoothed_ssn']
for p in ['train_a', 'train_b', 'train_c']:
    curr_period = solar['period'] == p
    solar.loc[curr_period, 'smoothed_ssn'] \
        = solar.loc[curr_period, 'smoothed_ssn'].fillna(
            method='ffill', axis=0).fillna(method='bfill', axis=0)
    roll = solar[train_short].rolling(
        window=20, min_periods=5).mean().interpolate('linear', axis=0)
    solar.loc[curr_period, train_short] \
    = solar.loc[curr_period, train_short].fillna(roll)
    solar.loc[curr_period, train_short] \
        = solar.loc[curr_period, train_short].fillna(method='ffill', axis=0)
```

- Aggregating the training data in 10-minute increments, taking the mean and standard deviation of each feature in the increment. This could have alternatively been done as the first layer of the neural network, but it's faster to do it as a preprocessing step.

```
# aggregate features in 10-minute increments
new_cols = [c + suffix for suffix in ['_mean', '_std']
    for c in train_short]
train_cols = new_cols + ['smoothed_ssn']
mean_cols = [i for i, c in enumerate(train_cols) if '_mean' in c]
new_df = pd.DataFrame(index=solar.index, columns=new_cols)
for p in ['train_a', 'train_b', 'train_c']:
    curr_period = solar['period'] == p
    new_df.loc[curr_period] \
        = solar.loc[curr_period, train_short].rolling(window=10,
    min_periods=1, center=False).agg(['mean', 'std']).values
    new_df.loc[curr_period] = new_df.loc[curr_period].fillna(
        method='ffill').fillna(method='bfill')
solar = pd.concat([solar, new_df], axis=1)
solar[train_cols] = solar[train_cols].astype(float)
```

  This enables us to reduce the size of the data 10-fold before training:

```
# sample at 10-minute frequency
solar = solar.iloc[10::10].reset_index()
```

3.5 Machine specifications

Training can be done on CPU or GPU. Training on CPU takes only slightly longer (16 minutes vs 11 minutes), so I'd recommend using that.

For my submission, I used a GPU, mainly because I was also experimenting with more complex neural networks that took longer to train, and for these the GPU was worthwhile.

3.5.1 GPU specifications

I used an AWS Ubuntu Linux instance (instance type p2.xlarge)

- CPU: Can't find this information on AWS website
- GPU: NVIDIA K80
- Memory: 61 Gb
- OS: Ubuntu Linux
- Training duration: approximately 11 minutes
- Inference time: approximately 170 minutes

3.5.2 CPU specifications

I used an AWS Ubuntu Linux instance (instance type t2.2xlarge). See Section 2 for how to set up the environment.

- CPU: Can't find this information on AWS website
- GPU: None
- Memory: 32 Gb
- OS: Ubuntu Linux
- Training duration: approximately 16 minutes
- Inference time: approximately 170 minutes

3.6 Other things tried

I started with a tree-boosting model using LightGBM. But this required calculating lots of engineered aggregate features (e.g. mean and standard deviation for different time ranges), which means infererence is slow and training requires a lot of memory (since features must be calculated before training for all time points). I found that the neural network gave better results with fast inference time. I also think it's a simpler and more elegant approach. I experimented with more complex neural networks (more layers and convolutions, more channels in the convolutions), but this simple one performed best on the leaderboard.

3.7 Other tools used for exploration

I did everything in Python, plotting with `matplotlib`.

3.8 Evaluating model performance

My model is an ensemble, trained on 5 subsets of the training data. I looked at out-of-sample performance for each of these slices. During development, I also kept a hold-out sample of the training data to evaluate performance of the combined 5-model ensemble (of course for submission I retrained the final set of models using all training data). I evaluated using the RMSE metric, since that's what's used on the leaderboard.

3.9 Things to watch out for

Nothing I can think of; the model doesn't require a lot of memory or long training time. Results will probably not reproduce exactly due to randomness effects, but should be very close.

3.10 Visualisations

I plotted a lot of graphs as I explored the data, but I didn't keep them. Plotting the data revealed that it is very noisy and seems to contain some spurious values, and that the distributions of some features are not Gaussian: they are asymmetic and some have very long tails. I found the following paper provided valuable background context on this, including many useful graphs:
Solar Wind and Interplanetary Magnetic Field: A Tutorial by C. T. Russell, Space Weather Geophysical Monograph 125, 2001
(https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/GM125p0073)

3.11 Further work

I noticed that there seemed to be systematic differences between the data produced by the 2 different satellites (it seemed there was a step-change in the mean and standard deviation when the data switched from Ace to Discover in period b). I tried to control for this but wasn't able to do it successfully. I think this is because we don't have much data for Discover, and so we can't see how it behaves across the different training periods and with different sunspot numbers. But if we could obtain more training data for Discover, it may be possible to correct for these systematic differences, which could make the model more accurate.

I didn't use the temperature data because it didn't seem to improve the model. However someone with more knowledge of the physics might be able to make use of it, and maybe also derive engineered features based on the underlying physical model of the system.

It would be useful to have sensor data that indicates when sensors were malfunctioning. From plotting the data, there seemed to be many anomalous periods where the data didn't look plausible, but I couldn't find a systematic way to filter these out, so I just left them unchanged (with the exception of a long period where the temperature is < 1, which is clearly bad data and which I excluded).