Deep learning model to estimate land-surface temperature in Israel

Student Name: Itay Barkai

Advisor: Dr. Oren Glickman (CS, BIU).

Project Description

The goal of this project is to build a deep-learning model to estimate land surface temperature in Israel at a 1x1 km resolution. The Land Surface Temperature (LST) is the radiative skin temperature of the land surface, as measured in the direction of the remote sensor. Historical LST data is available at a 1x1 km resolution, however we would like to see if we can use this data as a target for a deep learning model to learn to downscale temperature data to a finer resolution.

Available Data (For Israel):

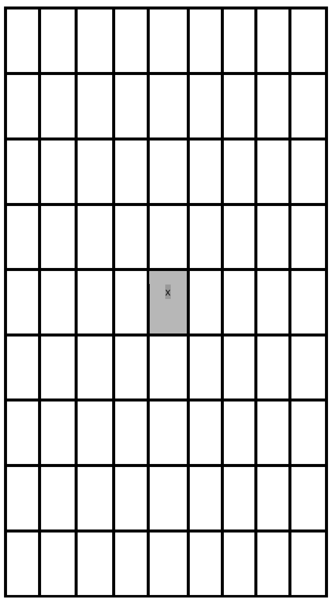
- Historical Hourly LST data for the past 20 years. Data includes:
    - lat, long of observation
    - Date & Hour in day
    - LST temperature
- Topography:
    - lat, long
    - height (above sea level)
- Vegetation index from satellite imaging (NDVI) - at a 1x1 resolution
- Additional information (e.g. 9x9 Air temperature, land use, ...) may be also available - to be possibly used in the project

shilo.shiff(at)biu.ac.il - will provide the data for this project.

Learning Task:

The goal of the deep learning model is to estimate the 1x1 km resolution land surface temperature given the surrounding topography.

For every such 1x1 km pixel, a 9x9 ‘patch’ surrounding the pixel as in the following illustration:



The overall LST of the 9x9 grid at a given time will be calculated as the average of the 81 LST observations at the same time. Similarly, the average height of the 9x9 grid will be calculated.

The difference between the 9x9 LST and center pixel 1x1 LST will be the target value to predict. Evaluation of the regression problem will be measured via RMSE.

For every such observation, the input to the model will include:

- The Average height of the 9x9 grid
- The average LST of the 9x9 grid
- 81 height diffs (difference between pixel height to the average height for every 1x1 pixel in the grid)

- The day in the the year (0-365, consider transform via sine and cosine)
- The time in the day (0-24, consider transform via sine and cosine)
- lat, long of center pixel
- We may add additional features as we progress (e.g. NDVI

Project work includes finding the optimal feature representation and deep learning neural net architecture (# layers etc,) as well as analysis of the results.

It is important to test the model on **unseen (held out) pixels and dates.**

Research Questions:

1) Can you improve RMSE predictions results over the simple baseline of always predicting 0 (temp of 1x1 center pixel identical to the 9x9 average).
2) What is the optimal DL model
3) How the various features impact the prediction

Intro

- All of the project's code, documents, and files is located at my public git repository at: https://github.com/itaybarkai/Deep-learning-land-surface-temperature-in-Israel/tree/develop

- Code is written in Python 3.8.8 with Tensorflow 2.13

- Libraries requirements are elaborated in the "requirements.txt" file.

- MLOps – use "MLFlow" for automated experiment logging (params and metrics), experiment reviewing and graphs displays.

- Running on DSI servers, using **self-made Docker image** as the environment.

- Data sources are:

1. LST dataset - https://zenodo.org/record/4533677#.ZFDyHHZBwuX

2. Topography data – From "Google Earth Engine" with this short script I wrote: (…Git/Data/Topography/ Topography_Israel_earth_engine.txt)

```
// lat,long taken from min/max of lst dataset

var israel = ee.Geometry.BBox(33.20034536095077, 34.00777509644337,
36.59654466075468, 28.996716325654262)

var dataset = ee.Image('USGS/SRTMGL1_003');

var i_data = dataset.clipToBoundsAndScale(israel, 409, 603)

Export.image.toDrive({

   'image': i_data,

   'fileFormat': 'GeoTIFF',

   'folder': 'MasterProject',

   'fileNamePrefix': 'my_topo',

   'description': 'my_topo',

   'dimensions': '409x603'

});
```

The LST data

Consists of NetCDFs for each year between 2002-2020.

Each one should have 2 samples per day of the LST on each 1X1 block in Israel.

Practically each NetCDF has 5 variables:

1. "y" = latitude, **603** points between 28.996716325654262 and 34.00777509644337

2. "x" = longitude, **409** points between 33.20034536095077 and 36.59654466075468

3. "band" = 4 values (channels):

NIGHT_LST = 0

DAY_LST = 1

DAILY_LST = 2 (average of 2 previous ones)

QA = 3

QA means if there were clouds that interrupted the actual sample and the data is generated from a previous model: (https://zenodo.org/record/4533677#.ZFkQRHZBwuX).
The values in the QA band represents:

NO_DATA_BOTH_DAY_NIGHT = 0

NO_DATA_DAY = 1

NO_DATA_NIGHT = 2

BOTH_VALID = 3

4. "time" = day in the year for the sample – from 0 to 365 (366 in leap years).

There might be skips in time in days that for some reason did not have a sample.

5. "__xarray_dataarray_variable__" = The actual LST value for each combination of the above variables.

Has shape of (364, 4, 603, 409)

* Notice the time variable has 364 instead of 366 (leap year in 2020) because 2 days samples are missing.

This is a **numpy masked array** which means it may have invalid data (like NaN).

For example in a single specific day there were 170k out of 240k valid values.

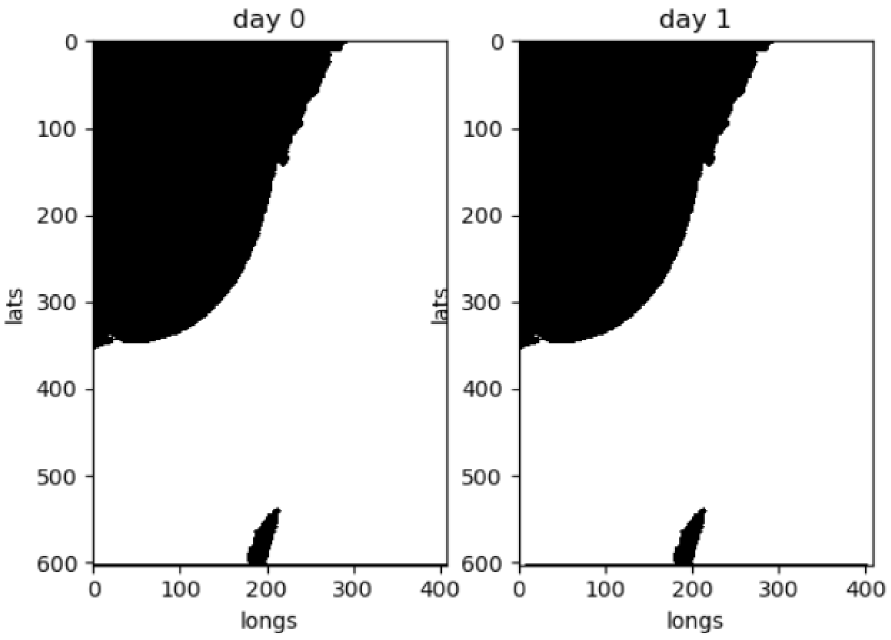
The units of the LST are 0.02 Kelvin, with values in range 7500-65535.

LST data Invalidity

As said, the data might be invalid/missing in some 1x1 spots on the grid.

We need to decide how to treat such data points, while we want to make samples out of each 1x1 spot and its 9x9 surrounding cell.

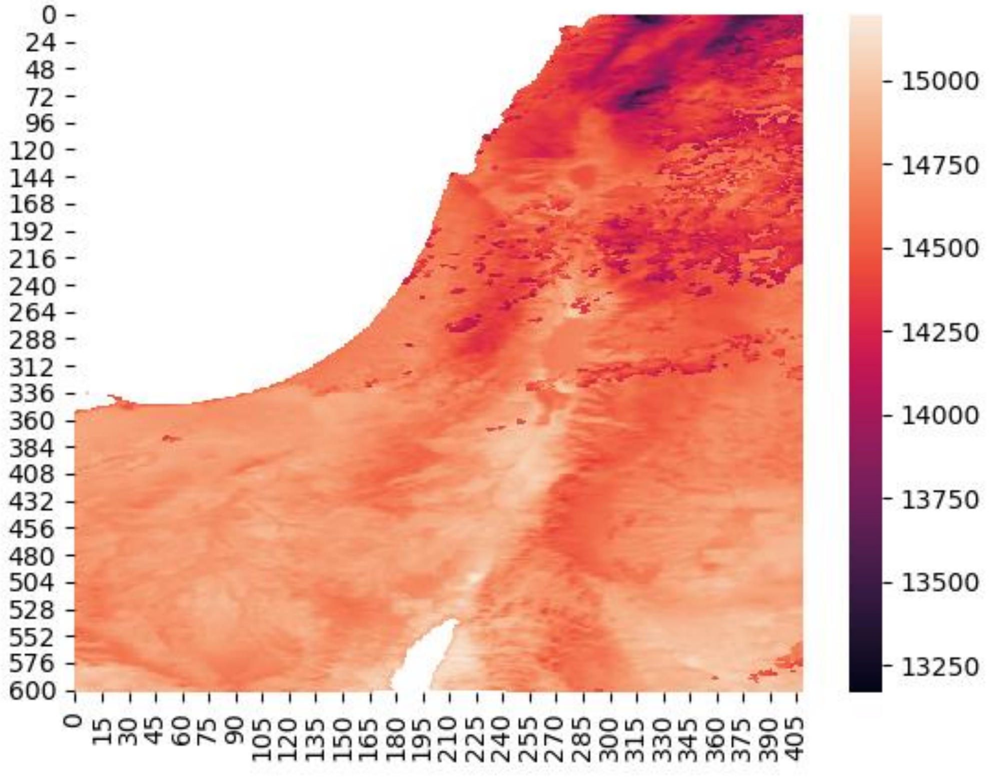
After printing the validity of the masked array's values of "True/False" as White/Black I got:



Aside from about 10-30 points the validity map is the same between days and it looks like a ground-sea map of Israel.

Overall it means we can use only 1x1 spots with a valid 9x9 surrounding and it should work fine.

Example day temp (0.02 Kelvin units)



The Topography data

Queried and exported from "google earth engine", in .tiff format that has a numpy array. Can be read with python's tifffile.imread.

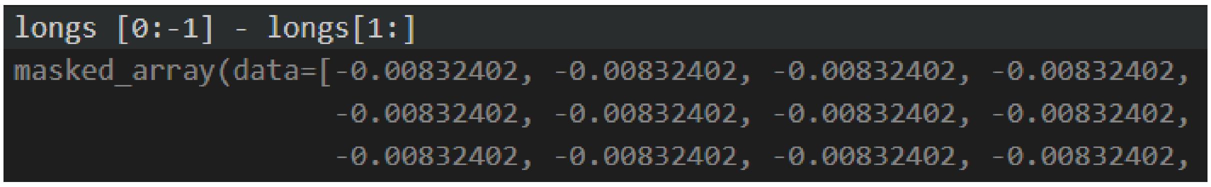
The dataset I received had the wrong resolution, as its shape was 379x559, when the LST's shape was 409x603. That would make it impossible to align the correct elevation to a sample.

In order to generate a correct resolution topography dataset I had to figure exactly how the LST dataset was made to recreate it in the google earth engine query.

In the LST data:
latitude, 603 points between 28.996716325654262 and 34.00777509644337
longitude, 409 points between 33.20034536095077 and 36.59654466075468

Found the length of the north and south borders to be 313km and 330km respectively, meaning the distance between each LST sample is changing.

After checking the difference between each longitude/latitude (in degrees), they were all equal ~1/120 of a degree :



```
longs [0:-1] - longs[1:]
masked_array(data=[-0.00832402, -0.00832402, -0.00832402, -0.00832402,
                   -0.00832402, -0.00832402, -0.00832402, -0.00832402,
                   -0.00832402, -0.00832402, -0.00832402, -0.00832402,
```

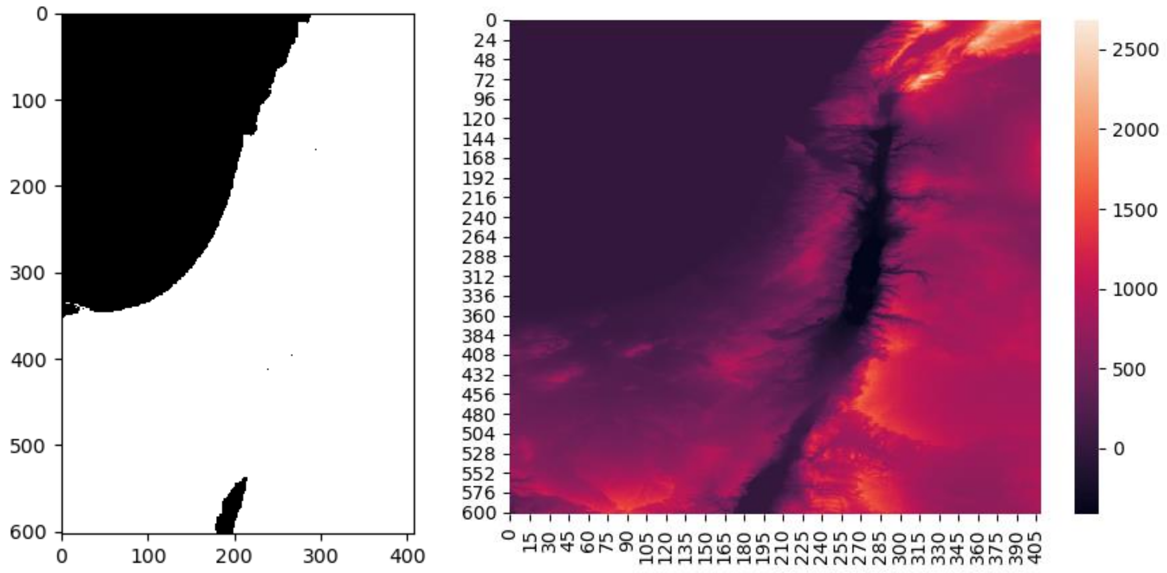
From the commonly used NASA topography dataset "USGS/SRTMGL1_003" it is documented to have resolution of arc-seconds (1/3600 of a degree), so rescaling 30x30 pixels to each output pixel is about right and possible.

I confirmed it by clipping the dataset to a rectangle within the bounding degrees of the LST data, and received shape (12228,18041), that matches: (max_degree-min-digree)/pixels ~= 1/3600.

I wrote a script in Earth Engine to scale this rectangle topography to dimensions of 409x603 just like the LST shape and exported a good looking .tiff file. (Script is saved near the data file)

Its min height is -415 and max height is 2687, which correctly describe the Dead Sea and the Hermon heights.

Topography binary image (left, sea values are 0) and heat map (right)



ax.imshow(image==0, cmap=plt.cm.binary, interpolation='nearest')

sns.heatmap(image_f)

Data Normalization and Feature Scaling

After parsing the raw data, and forming a dataset with each sample having the model inputs as described in the projects description. We have the following columns:

- day of the year
- longitude
- latitude
- LST average on 9x9 grid
- height average on 9x9 grid
- 81 columns of each cell's height diff from the average of its 9x9 grid

The question is how to tackle normalization without ruining the relations between the features (ex. different height diffs features getting differently normalized might affect the "grid" they should from)

<u>**1. "Day of the year"**</u> has a problem with not being cyclic – day 0 and day 364 should be close. We could use a cyclic function like sin() but it would create repeats (intermediate value theorem proves there are 2 different values that would map to the same output).
Therefor we must split the day to 2 cyclic columns, such as sin(day), cos(day),

representing the day of the year as a **Unit Circle**, thus cyclic without repeating outputs.

2. "long" and "lat" are evenly spaced in a segment. Intuitively it is common to use Normalization [(x – x_min) / (x_max – x_min)] for this data type.

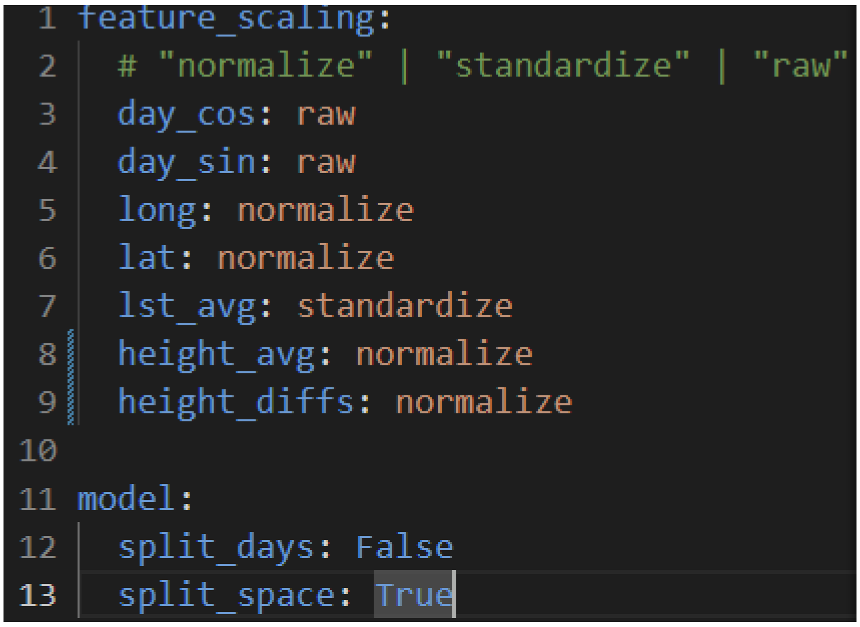
3. "Height Diffs" – since each one of the 81 columns represents the same data type and units, it is essential to scale them the same way, thus fitting the scaler on all these columns and then transforming them with the same parameters.

The specific Scaling method is left to experiment and is configurable.

Config

In order to test various hyper parameters such as the feature scaling methods and to split the training data by days or by locations (to make sure there are no leaks to the test set), There is a config file in ".yaml" format.

For example:



```yaml
feature_scaling:
  # "normalize" | "standardize" | "raw"
  day_cos: raw
  day_sin: raw
  long: normalize
  lat: normalize
  lst_avg: standardize
  height_avg: normalize
  height_diffs: normalize

model:
  split_days: False
  split_space: True
```

Currently the configs directory is ./Configs

Git Structure

All relevant files are in the git repository mentions in the "Intro" section:

- ./**Configs** – contains .yaml config files as described at the "Config" section
- ./**Data** –
    - LST – (This is in .git_ignore, very heavy folder)
        - downloaded – Downloaded .zip files from the link in "intro"
        - unprocessed – Extracted .nc – raw LST files
    - Processed – Cached dataset after initial processing (samples list)
    - Topography – Heights data, and generating Google Earth Engine script.
- ./**Documents** – Mostly contains this document (which includes the rest)
- ./**mlruns** – See "MLFlow" section
- ./**Source** - Code
- ./ – Contains docker build and run bash scripts, see "Docker" section.

Code documentation

config.py – Parsing of config .yaml files

consts.py – Consts of file paths, data and dataset

getTiffHandler.py – Handler class for .tiff files

netCDFHandler.py - Handler for .nc files of LST

preprocessing.py – Creates the dataset (samples and targets) from the raw data, and handles the caching of processed data. Also handles the dataset preprocessing, the feature scaling.

main.py – The file to run, an example usage of the project code.

Models/trivial_model.py – Model that predicts 0 always – the baseline.

Models/deep_model.py – Our Model class, defines the architecture.

Models/utils.py – Splitting train-test by days or by space.

Processing and Data Caching

The processing of the data goes as follow:

- For every pixel (1x1 km) we pick the 9x9 grid around it, calculating its average lst and average height.

- For each 1x1 spot in that grid, we calculate its height diff from the average.

- We calculate the cos and sin of the "day of the year"

And the result is:

sample = [day_cos,

day_sin,

middle_long,

middle_lat,

lst_grid_avg,

height_grid_avg,

81 height diffs columns]

target = lst_grid_middle

Caching: The resulted dataset is saved under "./Data/Processed/",
in .npz format by the "year" and the "number of days processed".

If a file with these same parameters is requested to process, it is instead loaded from its cached file, making it much fast (almost instant, while processing takes a while).

It makes multiple runs, (eg re-training using different model parameters) much faster.

Infrastructure + Docker

The running environment I've used is a docker container running tensorflow on the DSI gpu servers (currently on dsicsgpu05, all files are in the server's storage).

The relevant scripts in order to do it are in the git root ./ folder.

Dockerfile – To create the image that has tensorflow and ./run_main.sh script
Also puts and runs "pip install –r ./requirements.txt"

*** needs to have requirements.txt and run_main.sh in the same folder**

docker_build.sh – run on the server next to Dockerfile to build the image

docker_run.sh – run the container after building its image.
Also maps the project folder into the container

*** replace "~/project" with the location of the files from the git repository.**

run_main.sh – will be in "/" in the container, run it to run your **main.py**

Running new code:

After doing changes to the code or configs use MobaxTerm to copy the "Source" and "Configs" folders to replace the existing ones.
Because these folders are mapped to the container, you can just run "run_main.sh" again and the new files will be used.

Summarized guide:

- Connect to the server

- Use git to pull the project (preferably in ~/project)

  - If develop branch is ahead, checkout develop.

- Put the raw LST data In the Data folder:

  - LST in ./Data/LST/unprocessed (the .nc files)

- Run docker_builds.sh (only once!)

- Run docker_run.sh

- In the opened container run "run_main.sh"

- Do changes to the code (on the server or copy the new ones)

- re-run "run_main.sh"

* Docker image already exists in "dsicsgpu05" server

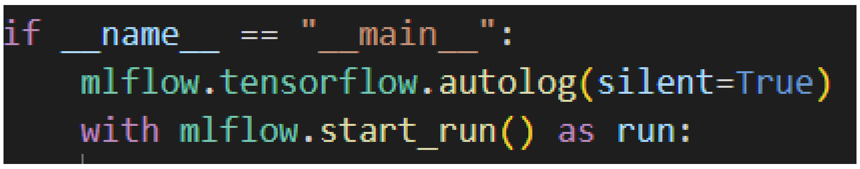
MLFlow

MLflow is an open source platform to manage the ML lifecycle, including experimentation, reproducibility, deployment, and a central model registry.

I am using the component "MLflow Tracking" which is an API and UI for logging parameters, code versions, metrics, and output files when running your machine learning code and for later visualizing the results.

Most of the parameters tracking, some metrics and the model itself are logged using the automatic mlflow-tensorflow integration with:

(The ML code in the block of start_run() is recorded to that run)



```python
if __name__ == "__main__":
    mlflow.tensorflow.autolog(silent=True)
    with mlflow.start_run() as run:
```

Additional metrics are logged using mlflow.log_metric such as "Trivial Model MSE", "Final Model MSE", "MSE Improvement", "MSE Improvement Percent".

The config file is also logged as an artifact.

mlruns moving files problem:

Couldn't find a simple solution for moving the mlruns output files from the server to my computer. Because:

Each run has a metadata file at:
./mlruns/0/run_id/meta.yaml
./mlruns/0/meta.yaml

It has a parameter names "artifact_uri" which is in local file:// protocol (should lead to a common storage server)

You can run the mlflow backend server in a way that could reach the same storage, but if you want to move the files to your filesystem you have to change these paths to match their new location. (They can't be relative).

To fix this problem, you can use mlf-core script:

Installation: https://mlf-core.readthedocs.io/en/latest/installation.html

Usage: https://mlf-core.readthedocs.io/en/latest/fix_artifact_paths.html

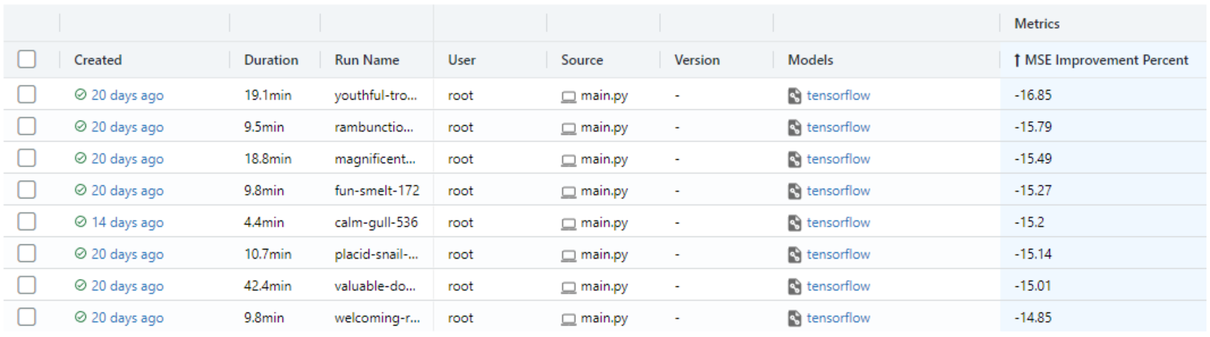
mlf-core fix-artifact-paths <PATH>
PATH is the directory that contains the mlruns directory (the git folder)

MLFlow ui:

in the git folder, run in cmd: "mlflow ui" to run the server on localhost:5000,

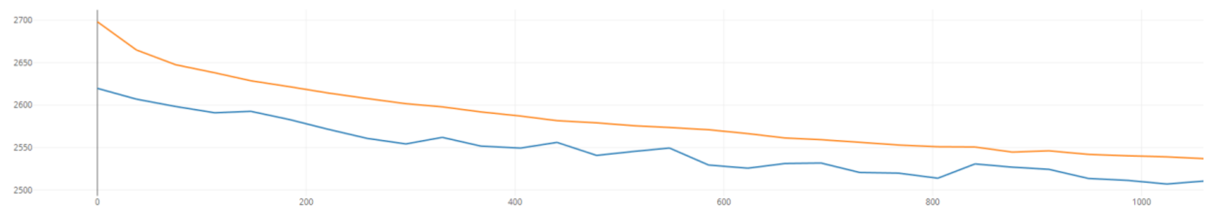
then in your browser head to 127.0.0.1:5000.

- Sort button on the main experiment page, to sort by "MSE Improvement Percent" for example.



								Metrics
☐	Created	Duration	Run Name	User	Source	Version	Models	↑ MSE Improvement Percent
☐	20 days ago	19.1min	youthful-tro...	root	main.py	-	tensorflow	-16.85
☐	20 days ago	9.5min	rambunctio...	root	main.py	-	tensorflow	-15.79
☐	20 days ago	18.8min	magnificent...	root	main.py	-	tensorflow	-15.49
☐	20 days ago	9.8min	fun-smelt-172	root	main.py	-	tensorflow	-15.27
☐	14 days ago	4.4min	calm-gull-536	root	main.py	-	tensorflow	-15.2
☐	20 days ago	10.7min	placid-snail-...	root	main.py	-	tensorflow	-15.14
☐	20 days ago	42.4min	valuable-do...	root	main.py	-	tensorflow	-15.01
☐	20 days ago	9.8min	welcoming-r...	root	main.py	-	tensorflow	-14.85

- Can view a chosen model's parameters, architecture and metrics



- Model can be loaded from its artifact using MLFlow API

* Some parameter logging was added after some runs were done, so it might be missing from old runs

Models and Learning Process

For the learning process to not leak information from the train set to the test set, it is important to make sure the split of the dataset is fair:

1. Split by day: Complete day's data is chosen for train/test, so that same day information is not leaked between different locations.

2. Split by space: Each space (location) is chosen for train/test, so that information on this location from different days is not leaking.

I have used about 30-70% split.

This is addressed as a **regression** problem, which goal is to predict the LST in the middle of the grid out of the LST average of its surrounding 9x9 block.

The **loss** used is MSE, while training and both MSE and RMSE while testing.

Models tested were simple deep fully connected networks (2-5 layers).

<u>Hyper Parameters and techniques tested were:</u>

Architecture – Did not really matter, as long as it have enough capacity.
Got to 1024->512->256->128->1, it can easily overfit the data.

Optimizer – Adam worked well

Batch size – 128 worked well and faster training than 64

Shuffle data each epoch – obviously important

Early Stopping – Saving and keeping the model from the epoch with the best validation loss, to help with over-fitting.

Learning Rate – Much improvement moving from constant LR to decaying LR, used a decay factor of / (2^0.1) which halves LR every 10 epochs.

Dropout – Improved handling of overfit, but lowers the model accuracy if using too much. About 0.2 seems in the middle, but I'm not sure if it exact. Also I can question about using it on all vs some of the layers.

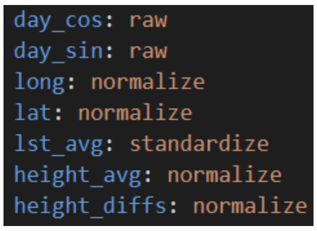
Batch normalization – while almost doubling each step training time, it seems to improve convergence time in epochs. (This layer comes before Activation)

Activation function – ReLU worked fine (about the same is leaky relu)

Weight initialization – used he_normal, but do not see much difference from other methods.

Epochs – 50 or 60, changes in according to other parameters.

Feature Scaling – This was the best working configuration:



```
day_cos: raw
day_sin: raw
long: normalize
lat: normalize
lst_avg: standardize
height_avg: normalize
height_diffs: normalize
```

Also mention "Number of days" of data, and train/test split method as non-parameters that affect the model performance

Results

Evaluation is calculated on the MSE of a trivial model that always predicts 0 on the validation set, versus the validation MSE of our model. In raw number and in percentage of improvement.

<u>Data of 2 days in 2020 split by **space** (mostly searched for hyper parameters)</u>

With 0.2 dropouts:

Trivial MSE – 3219, Model MSE – 2522, Improvement – 696 = **21.6%**

Without dropout:

Trivial MSE – 3219, Model MSE – 2428, Improvement – 790 = **24.56%**

* Train MSE is 1988 vs 1522, seems like the more overfitting model also gets better validation loss. But might be less generalizing.

Data of 5 days in 2020 split by **space** (only test - unchanged model parameters)

With 0.2 dropouts:

Trivial MSE – 3241, Model MSE – 2637, Improvement – 603 = **18.62%**

Without dropouts:

Trivial MSE – 3241, Model MSE – 2504, Improvement – 737 = **22.74%**

Data of 5 days in 2020 split by **days** (only test - unchanged model parameters)

With 0.2 dropouts:

Not learning well in validation – **6.5%** and getting worse while training

Without dropouts:

Not learning well in validation – **8%** and getting worse while training

Data of 10 days in 2020 split by **days** (only test - unchanged model parameters)

With **0.3** dropouts:

Trivial MSE – 2495, Model MSE – 2100, Improvement – 394 = **15.8%**

* Splitting by days seems a bit harder to get the model to converge and learn but seems possible with better fine tuning. And probably easier when using larger training dataset.

This might point to an inherent situation where same day cells are correlated enough to share LST-Topography relations, which makes the problem easier even when holding out pixels.

Conclusion

The 2 main goals of this project were:

1 - Prove with a POC that there is enough information in surrounding Topography to predict the LST in a smaller area than available from satellite sensors

2 – Provide an easy to use infrastructure for running experiments with this organized data, while allowing the addition of more data features for more advanced research.

The results show existing correlation and proven ability to learn the relation between the surrounding Topography and the LST of a smaller area.

Also, the codebase is very approachable and makes it easier to conduct further research, with many quality-of-life features for efficient training cycle and MLOps tools to track experiments, all which can be used in other projects as well.