Geomagnetic storms

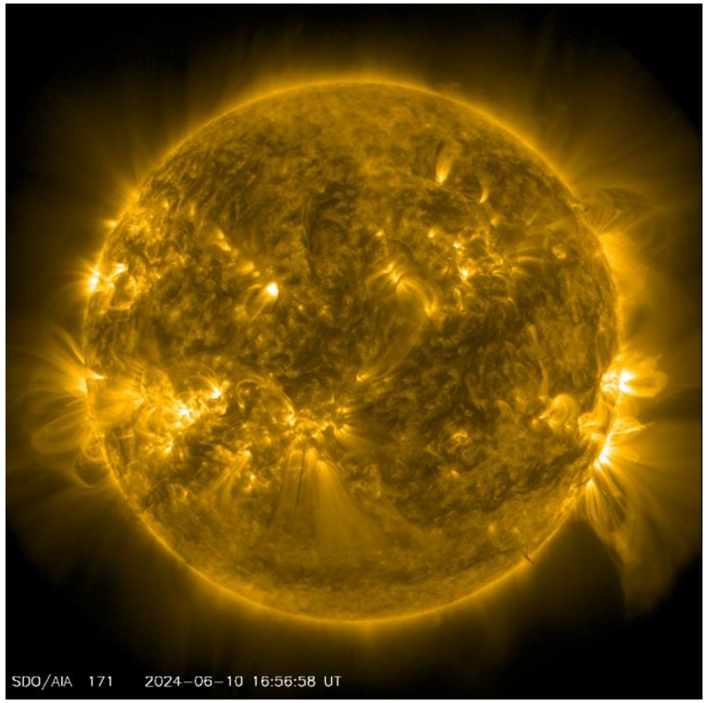
BY: ALI AHMAD (XGH224) & FLORENT I. MUSTAFAJ (WQB322)

All participants contributed evenly to the project

Introduction

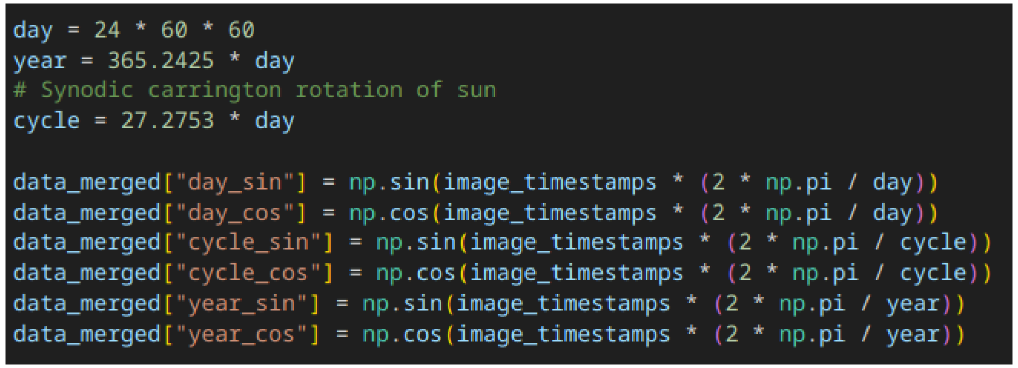
Goal: Predicting geomagnetic storms from solar (image) data

- SDO Image Data & Kp-index (3hr intervals)
  - AIA 171
  - Kp: Maximum magnetic field disturbance
  - Kp is normalised (calm day)
- Sunspots
- Flares & CMEs (different timescales)
- Solar rotation (Carrington rotation)
- Earthly consequences



Data & Preprocessing

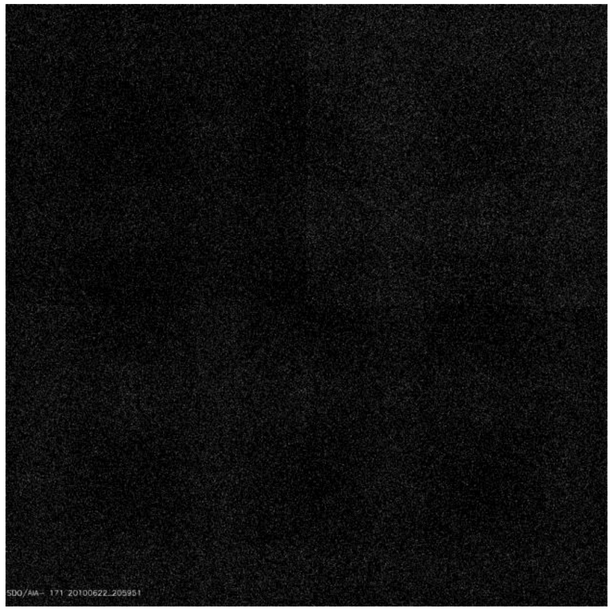
- Data sources:
  - Image data (NASA)
  - Kp + SN (Helmholtz Center, Potsdam)
- Webscraping
  - 8 images extracted per day (~3hr interval)
  - Request limitations (data gathering ~30 hrs)
  - Data from May 2010 to May 2024 (~40000 images & Kp datapoints)
- Data preprocessing
  - Kp-interpolation (CubicSpline)
  - Datascaling (MinMax), Imagescaling (Range(-1, 1), Mean=0.5, Std=0.5, all RGB channels), Resizing
  - Sequence creation (n_seq = 7)
  - Timestamp transformations
  - Remove unwanted images

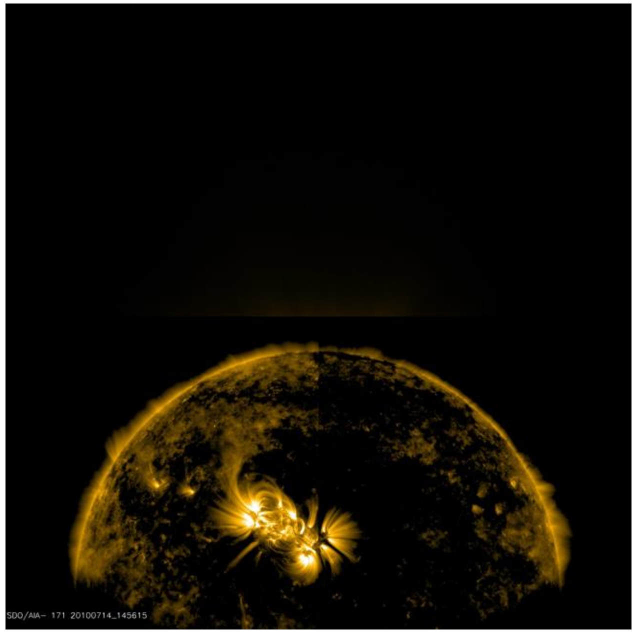


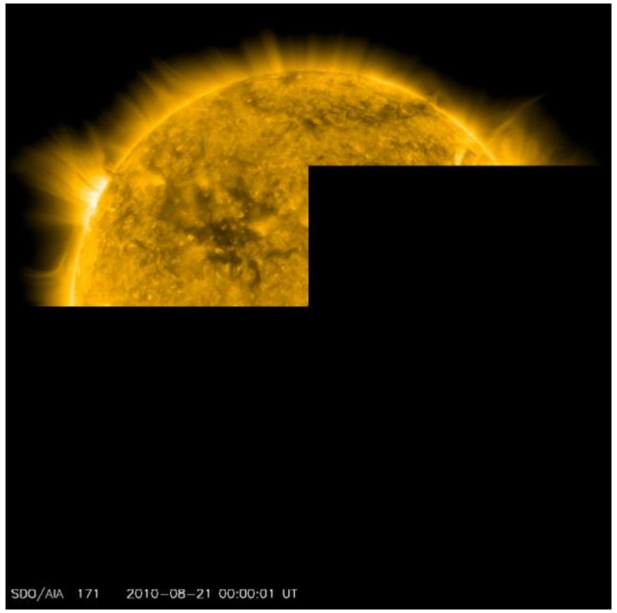
```python
day = 24 * 60 * 60
year = 365.2425 * day
# Synodic carrington rotation of sun
cycle = 27.2753 * day

data_merged["day_sin"] = np.sin(image_timestamps * (2 * np.pi / day))
data_merged["day_cos"] = np.cos(image_timestamps * (2 * np.pi / day))
data_merged["cycle_sin"] = np.sin(image_timestamps * (2 * np.pi / cycle))
data_merged["cycle_cos"] = np.cos(image_timestamps * (2 * np.pi / cycle))
data_merged["year_sin"] = np.sin(image_timestamps * (2 * np.pi / year))
data_merged["year_cos"] = np.cos(image_timestamps * (2 * np.pi / year))
```

Corrupted/unwanted images - Examples





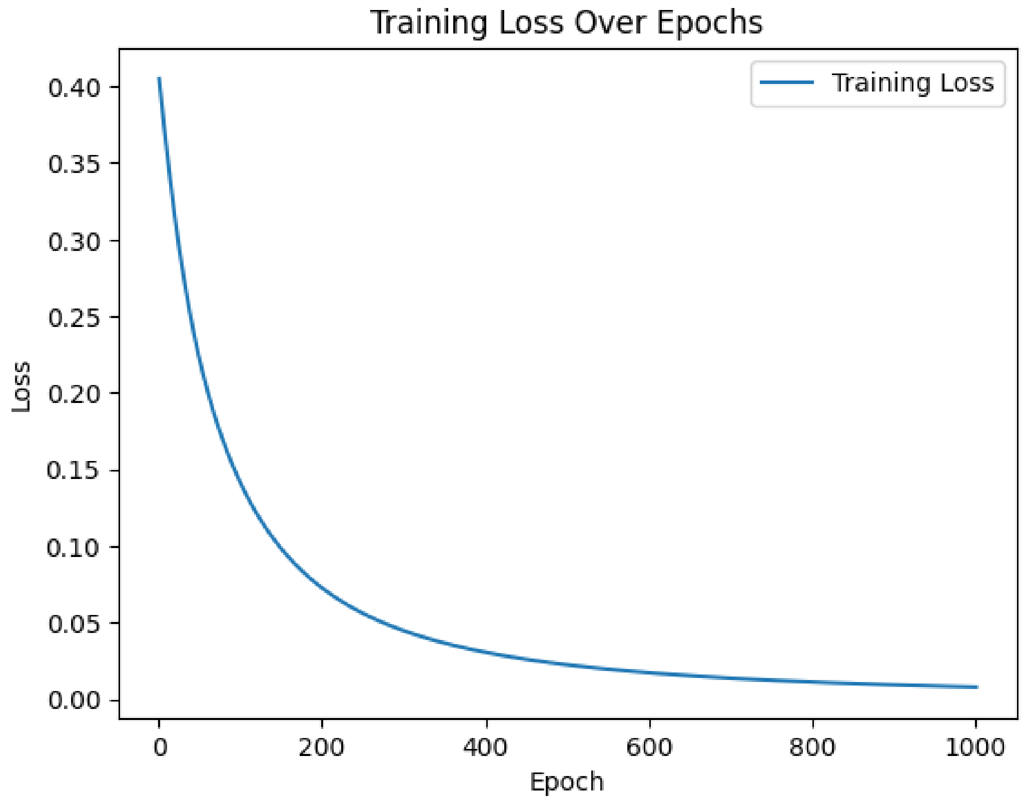


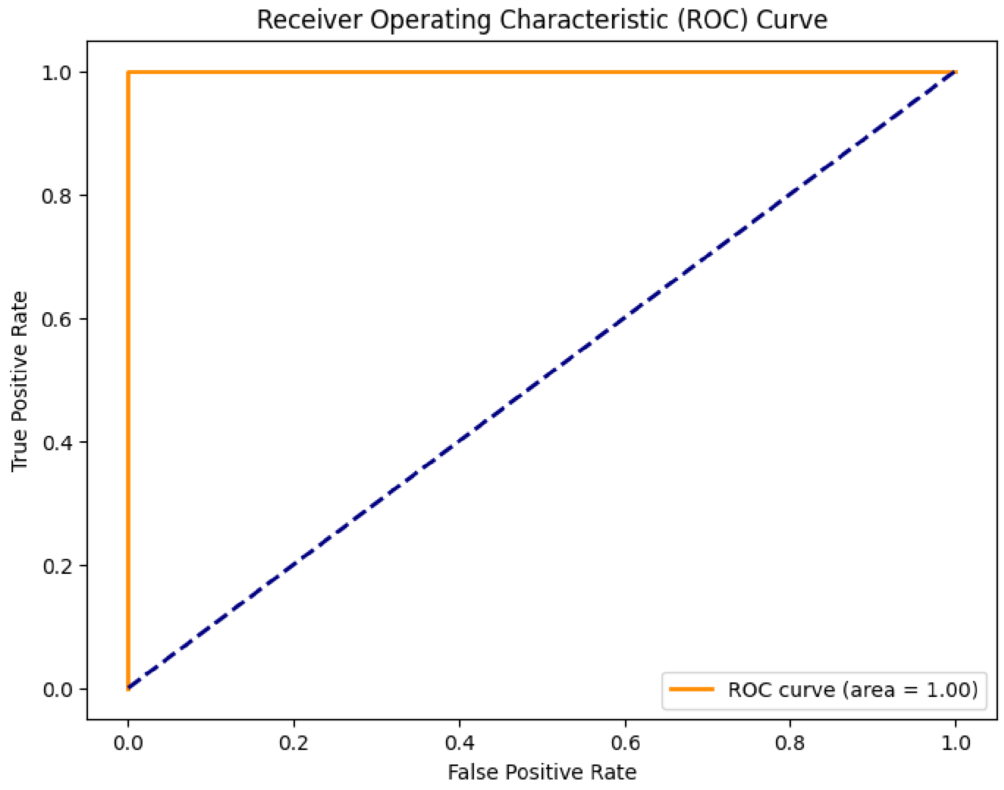
Sunspot model

Simple models based on sunspot number (which is measured once a day)

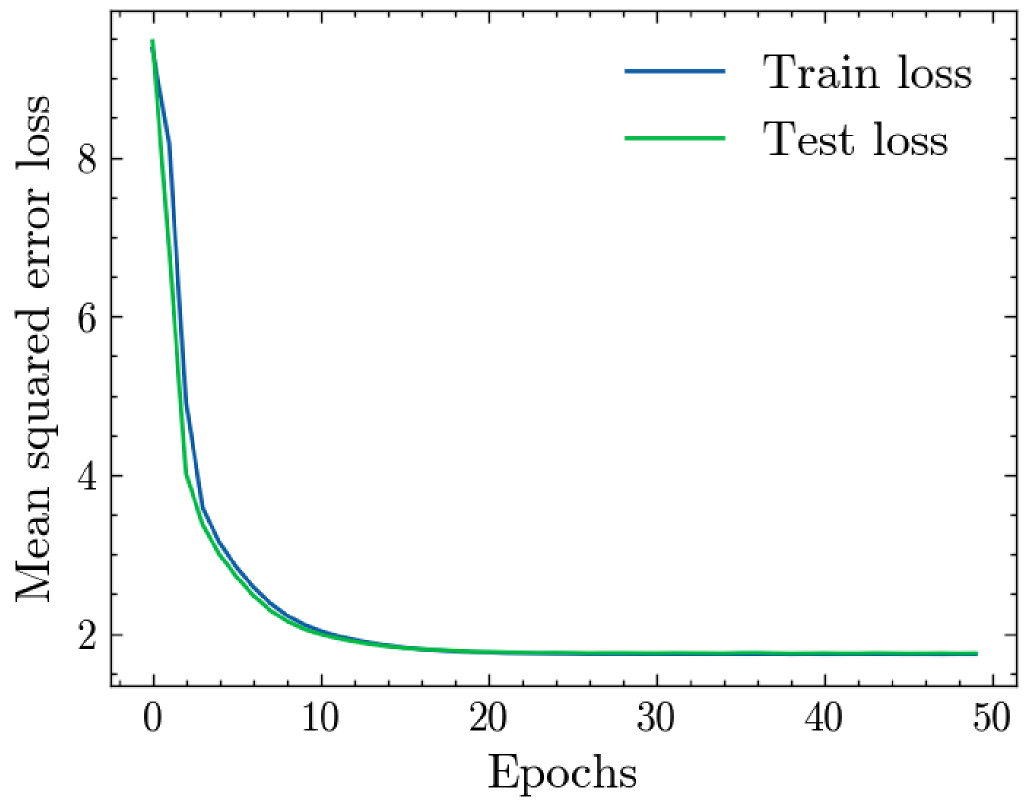
- Simple classification (Predict storm as Kp>=5, no time series)
  - 1-layer FF + Sigmoid
- Time series/regression (Predict max daily Kp-index)
- LSTM (hidden_size = 50, num_layers=5) + Linear layer
  - Lookback: 1 week
- Evaluation:
  - Classification – Perfect, but useless!
  - Time series – Poor, too little information, infrequent measurements

Classification





Time series/Regression



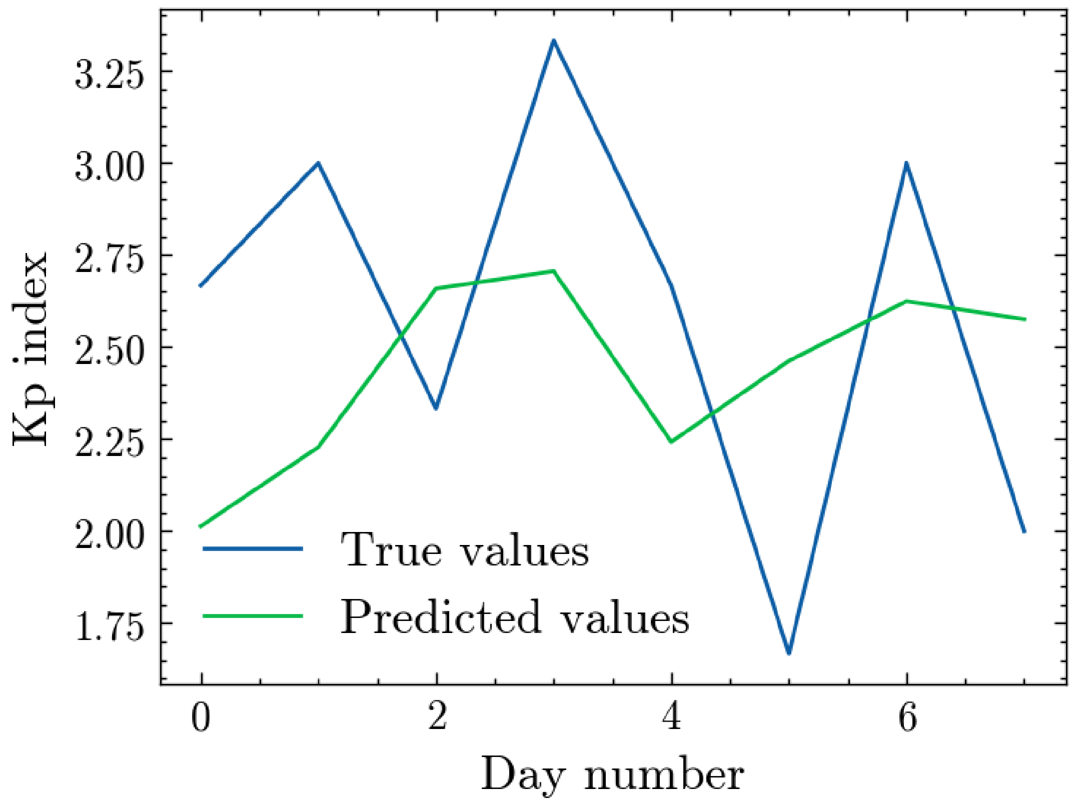


Image model: Architechture Overview

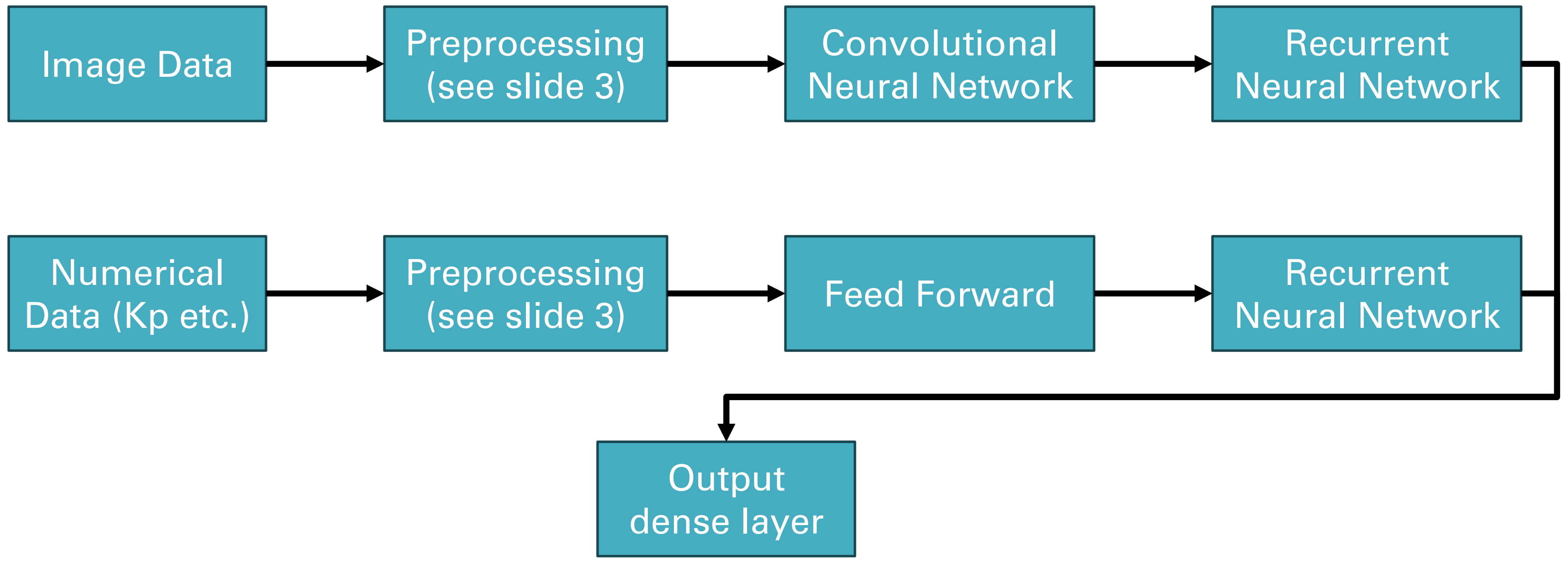


Image model: Architechture details

- Convolutional neural network
  - Consists of three 3x3 kernel Conv2d layers (input RGB): (3, 8) → (8, 16) → (16, 32)
  - RELU + MaxPool2d between each layer
  - Final dense layer with 4 output features

  Note: Each image ran through CNN separately, then outputs are stacked

- Feed-forward neural network (8 numerical features)
  - Two layers: (8, 16) → (16, 32)

- Recurrent neural networks
  - Numerical features: (num_layers = 1, hidden_size = 32)
  - CNN output features: (num_layers = 1, hidden_size = 32)

- Output dense layer (Stack both RNN outputs, dense layer: (64, 1))

- Optimizer: Adam (default settings, $LR_{class} = 5 \cdot 10^{-4}$ & $LR_{regr} = 1.5 \cdot 10^{-3}$)

Performance & Optimisation

Challenge: 40K total images – Each batch $(8 \times 7 \times 3 \times 512 \times 512\ ,\ 8 \times 7 \times 8\ ,\ 8 \times 1)$

→ **Things get inefficient <u>really</u> fast!**

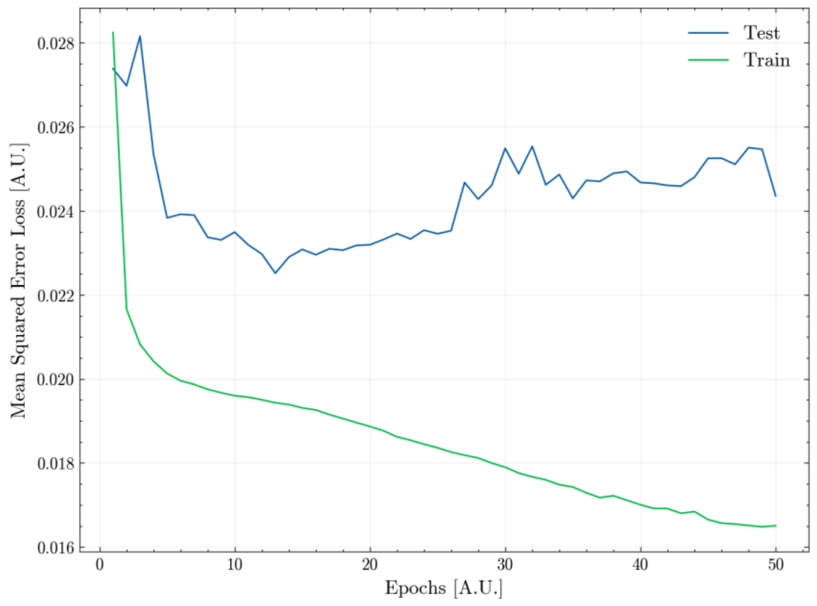
- In the beginning:
  - Slow performance
  - Scaled badly with batchsize
  - Slow model evaluation

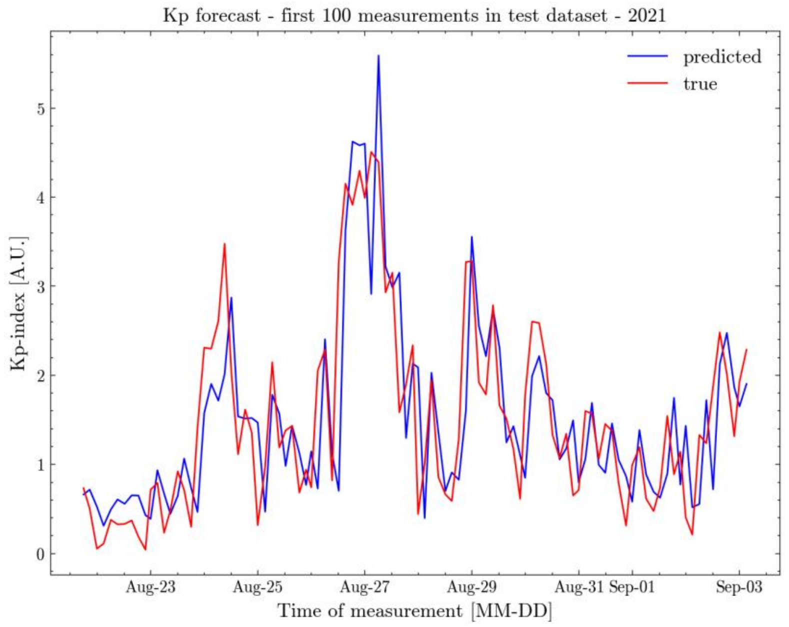
- Debugging:
  - Main culprit – get_item very slow (tried many things)
  - Main fix: Multithreading + CUDA (6x speedup)
  - Example (1% of data, per epoch, only dataloading): ~35s to 6s

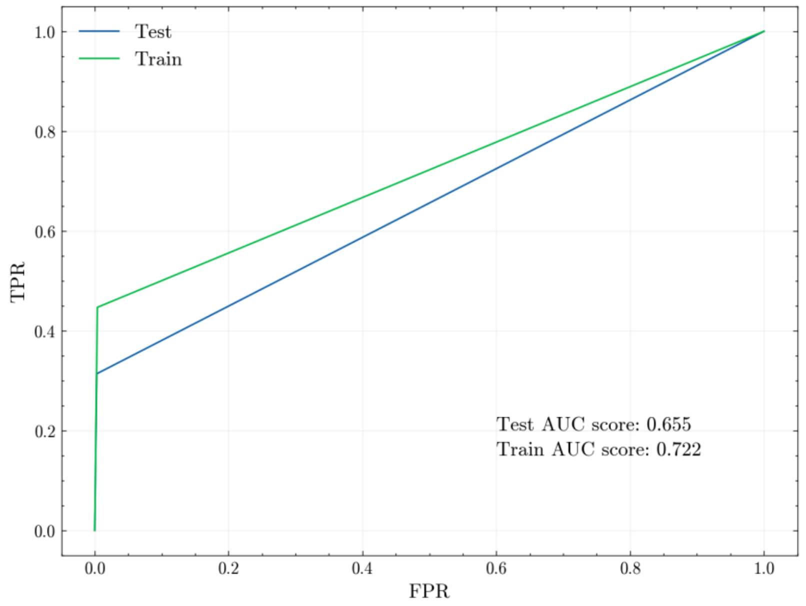
- Further speedup: Decrease model size (see appendix for details)

Result: 25% of data (w/ no validation): 14 hours → 80% of data (w/ validation): 11 hours

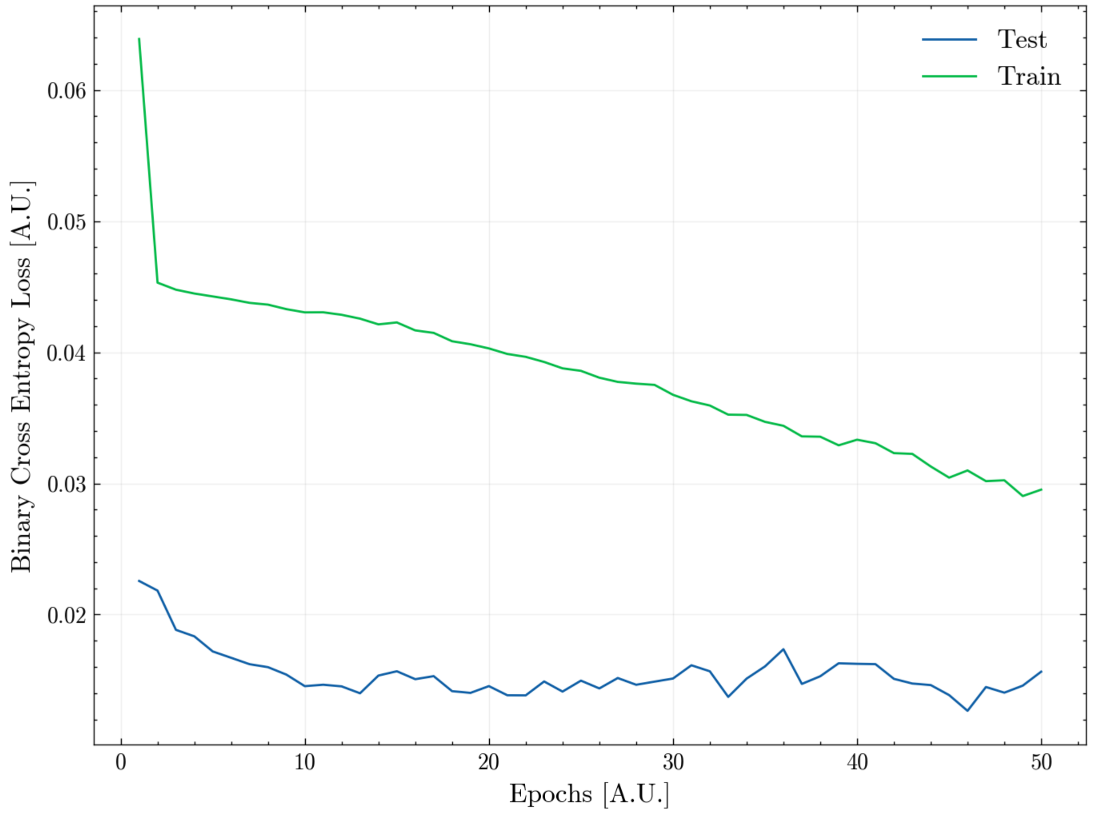
Results of regression – 80% train & 50 epochs

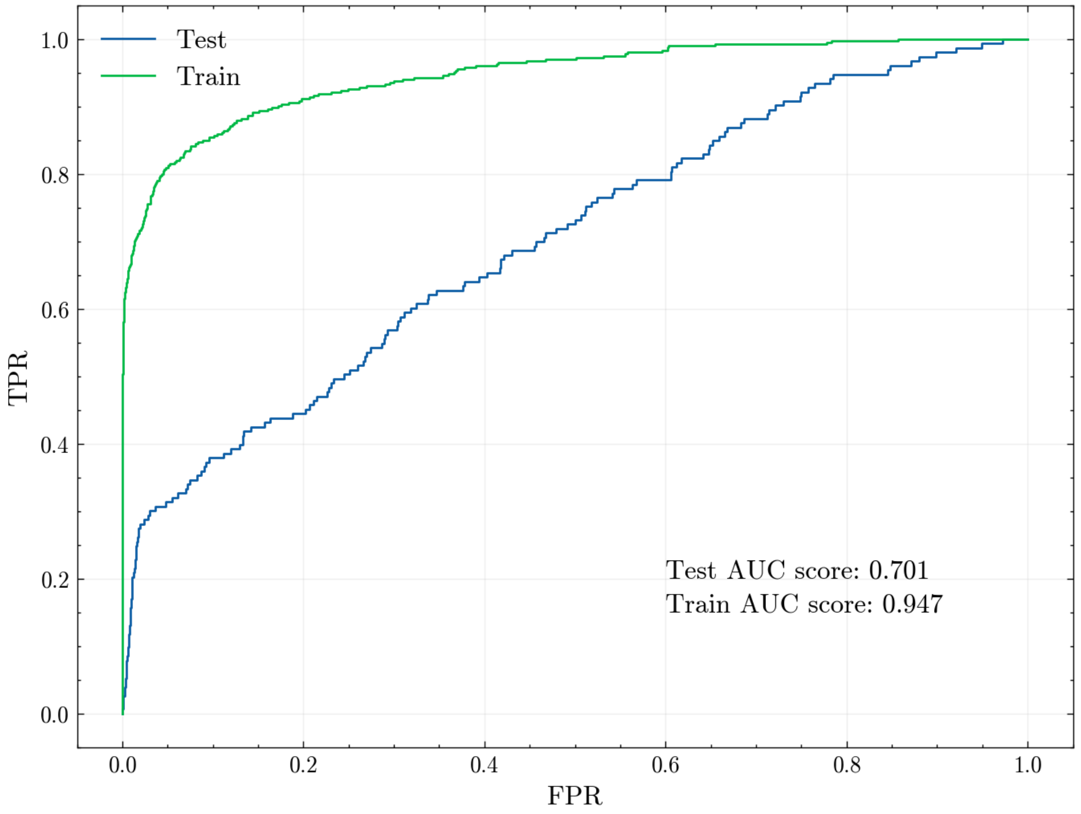






Results of classification – 80% train & 50 epochs





Conclusion

- Model quite good for Kp-forecasting – not quite as good for storm prediction
- Forecasting – Possible alternatives
  - Forecasting further into the future (Longer sequences/lookback)
  - Auto-regression not possible
  - Dumb implementation: Predict all inputs. Smart implementation: SEQ2SEQ models from NLP
- Storm prediction – Possible alternatives
  - Longer sequences/lookback (Capture storm ”brewing” timescale correctly)
  - Low number of storms: (possibly) Anomaly Detection models (?)

Both improvements require longer lookback → VRAM issues. See appendix for possible fix

(Or buy a bigger and better graphics card)

Thanks for listening!

Special thanks: Troels Petersen, TAs & Morten Holm

Appendix

Link to GitHub Repo

https://github.com/AliAhmad02/AppliedML-Final-Project

A: Optimisations

Next are various optimizations we tried to make throughout the project

Dataloader optimization – Failed attempts

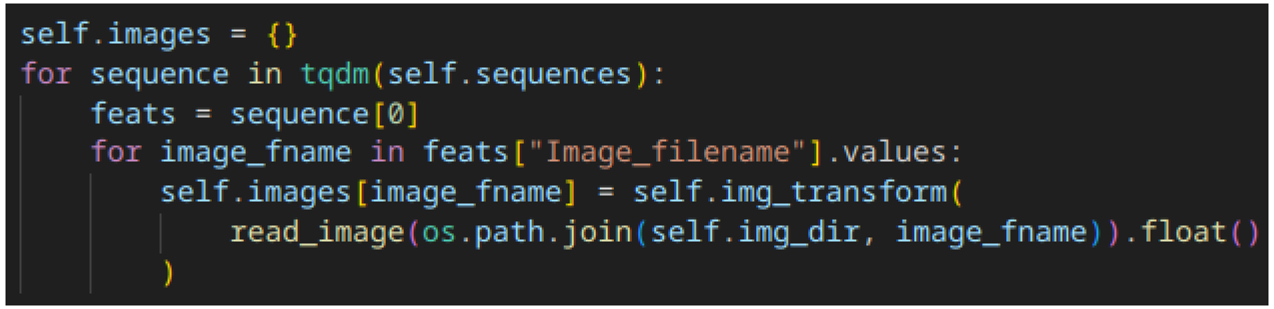
Attempt 1: Pre-Reading all tensors and storing them into memory

Using torchvision.io.read_image

Result: Didnt work because tensors

are too memory-inefficient (114GB RAM

Needed to be allocated for 7GB data)



```
self.images = {}
for sequence in tqdm(self.sequences):
    feats = sequence[0]
    for image_fname in feats["Image_filename"].values:
        self.images[image_fname] = self.img_transform(
            read_image(os.path.join(self.img_dir, image_fname)).float()
        )
```

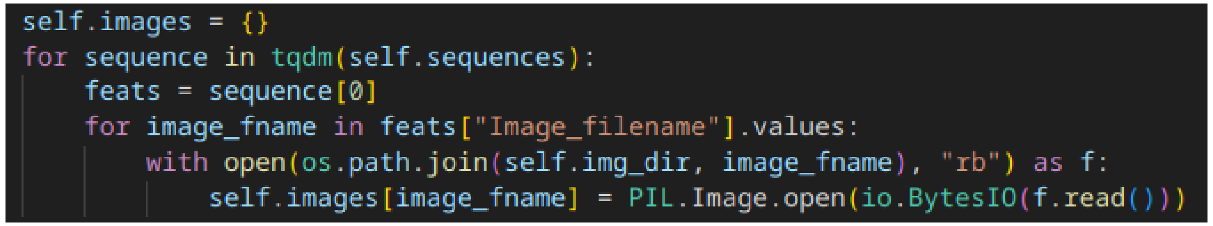
Attempt 2: Reading images into memory as PIL binary object

Result: No speed-up, turns out reading images from file is

quick, however conversion to pytorch tensor is the

most time consuming part

(and this did not address that issue)



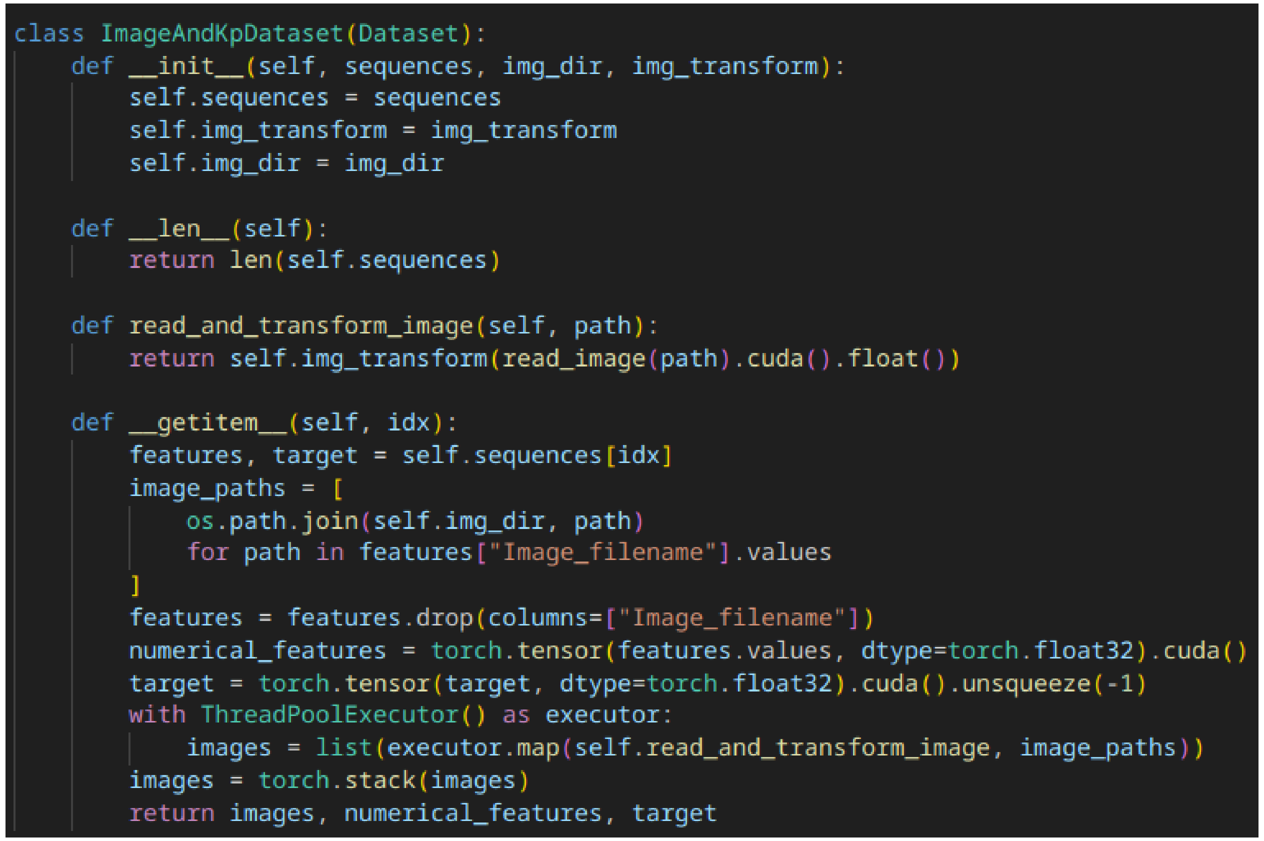
```
self.images = {}
for sequence in tqdm(self.sequences):
    feats = sequence[0]
    for image_fname in feats["Image_filename"].values:
        with open(os.path.join(self.img_dir, image_fname), "rb") as f:
            self.images[image_fname] = PIL.Image.open(io.BytesIO(f.read()))
```

Dataloader optimization – Working solution

Fix 1: Use of python multithreading functionalities to read in the sequence of images simultaneously

Fix 2: Further speed-up from converting each tensor to CUDA tensor immediately in the dataloader.

Note: Typically, one converts tensors to CUDA in the training loop. However, doing it in the dataloader immediately after reading them in gave a significant speedup in our case. This is likely because we do a few operations in the dataloader such as stacking images and transforming them.



```python
class ImageAndKpDataset(Dataset):
    def __init__(self, sequences, img_dir, img_transform):
        self.sequences = sequences
        self.img_transform = img_transform
        self.img_dir = img_dir

    def __len__(self):
        return len(self.sequences)

    def read_and_transform_image(self, path):
        return self.img_transform(read_image(path).cuda().float())

    def __getitem__(self, idx):
        features, target = self.sequences[idx]
        image_paths = [
            os.path.join(self.img_dir, path)
            for path in features["Image_filename"].values
        ]
        features = features.drop(columns=["Image_filename"])
        numerical_features = torch.tensor(features.values, dtype=torch.float32).cuda()
        target = torch.tensor(target, dtype=torch.float32).cuda().unsqueeze(-1)
        with ThreadPoolExecutor() as executor:
            images = list(executor.map(self.read_and_transform_image, image_paths))
        images = torch.stack(images)
        return images, numerical_features, target
```

Memory optimization

As mentioned, increasing the sequence length can quickly lead to memory issues.

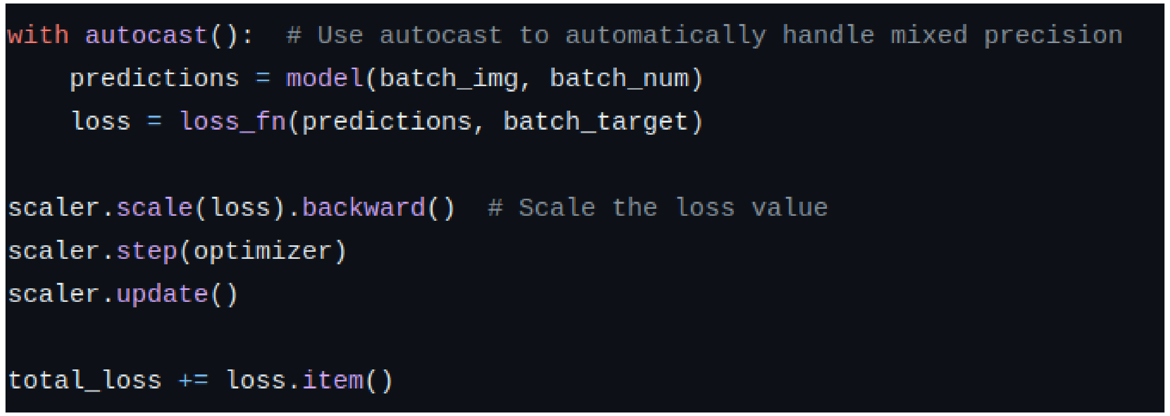
Some fixes were looked into and implemented, but ultimately not used due to time limitations in running with longer sequences.

The following slides show some of the implementations

Memory optimization: Mixed precision training

Normally, we store floats as 32-bits. Using 16-bits will reduce the memory usage by one half, however this can lead to issues.

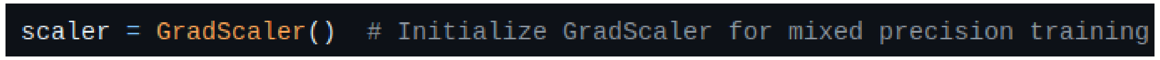
Mixed precision training automatically determines when we can use 16-bit, and when we have to use 32-bit.



```
with autocast():  # Use autocast to automatically handle mixed precision
    predictions = model(batch_img, batch_num)
    loss = loss_fn(predictions, batch_target)

scaler.scale(loss).backward()  # Scale the loss value
scaler.step(optimizer)
scaler.update()

total_loss += loss.item()
```



```
scaler = GradScaler()  # Initialize GradScaler for mixed precision training
```

Memory optimization: AutoEncoder

The majority of the memory problems comes from the fact that we have to have many images in sequence, and the images themselves are quite large.

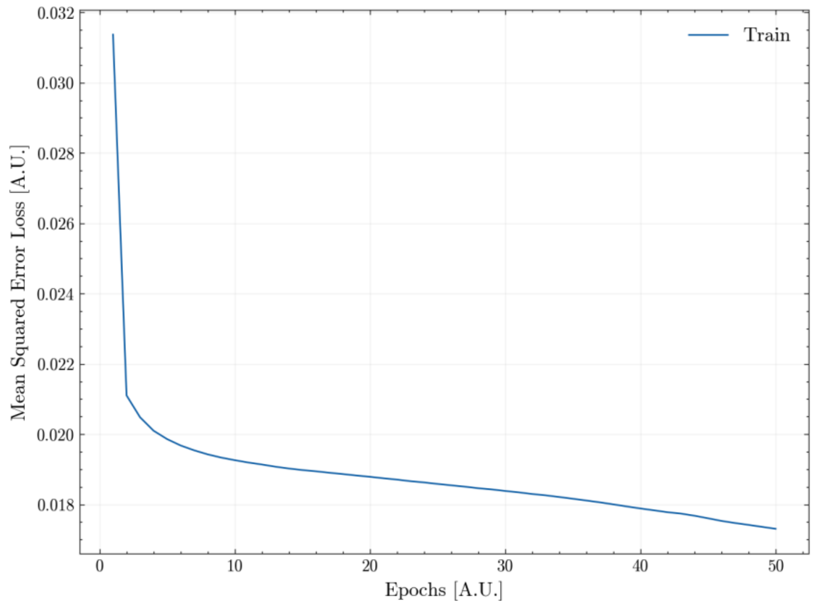
One fix is to run the images through an autoencoder, to reduce their dimensionality to, say, 4 numbers. With this, we can easily have a very large number of images in memory at once.

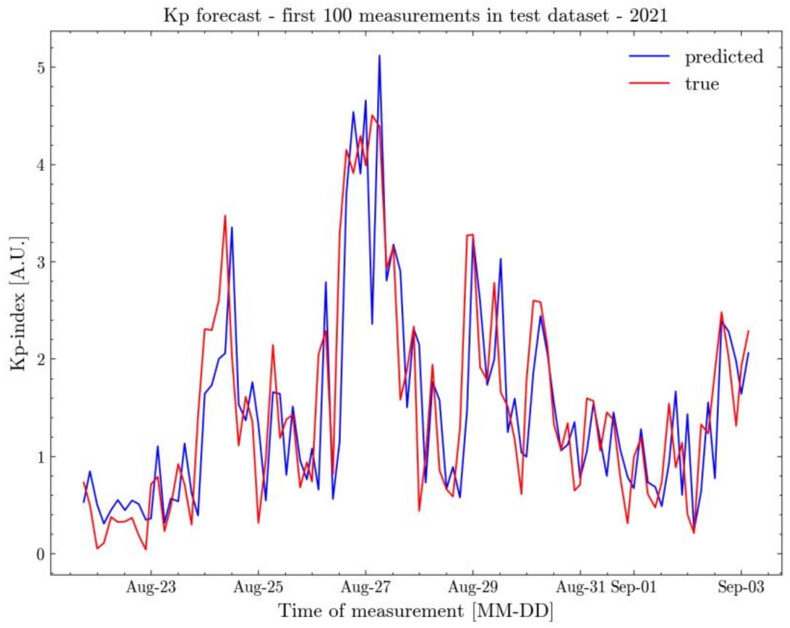
https://github.com/AliAhmad02/AppliedML-Final-Project/blob/main/autoencoder.py

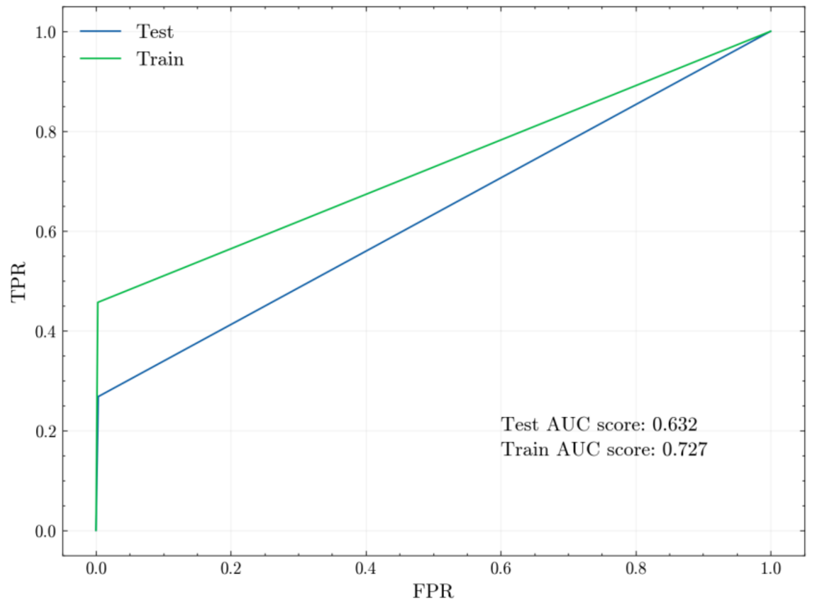
B: Additional plots

Next slides show additional plots generated using our image model.

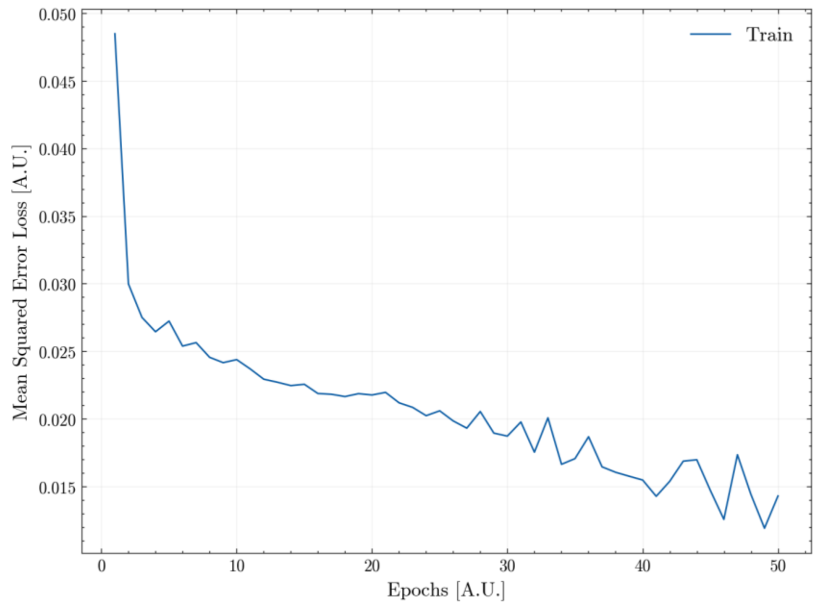
Results of regression – 80% train & 50 epochs – $LR = 5 \cdot 10^{-4}$

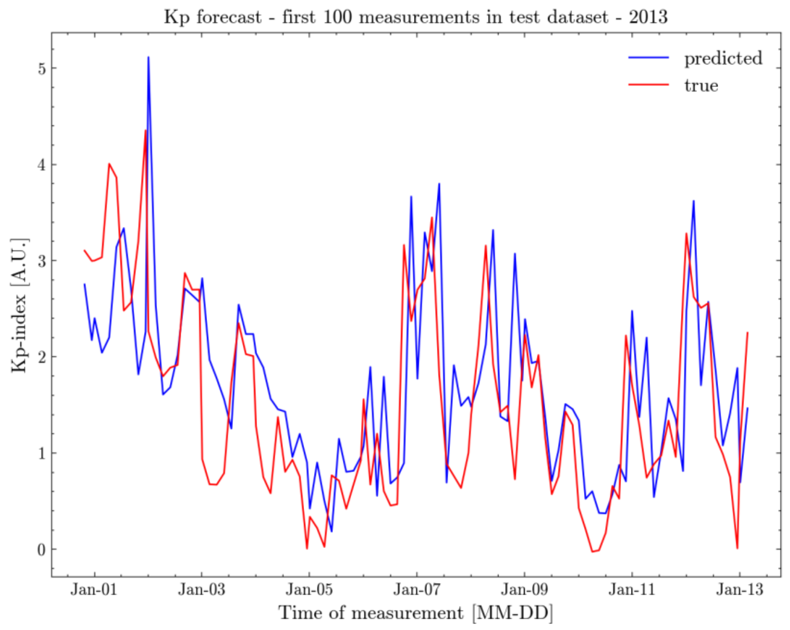


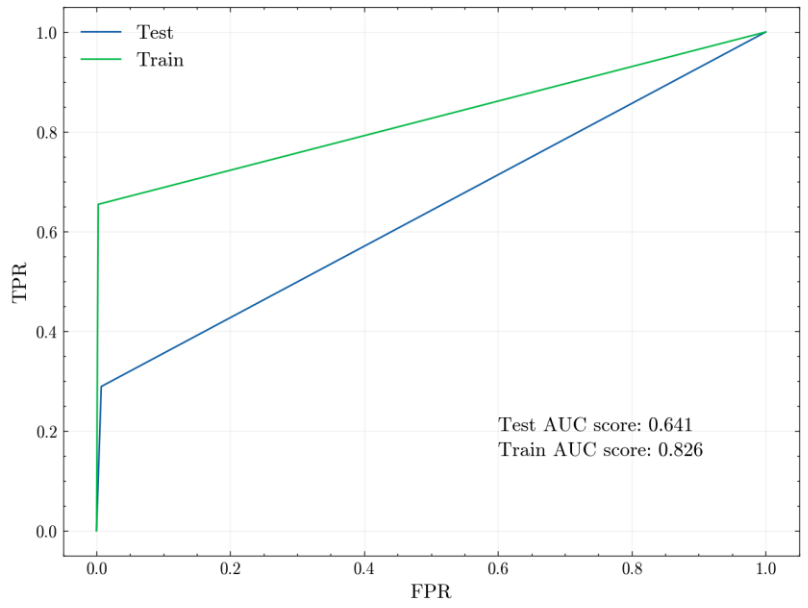




Results of regression – 25% train & 50 epochs – $LR = 10^{-3}$







Note: *This is a larger CNN model, and was ran before we implemented optimizations.*

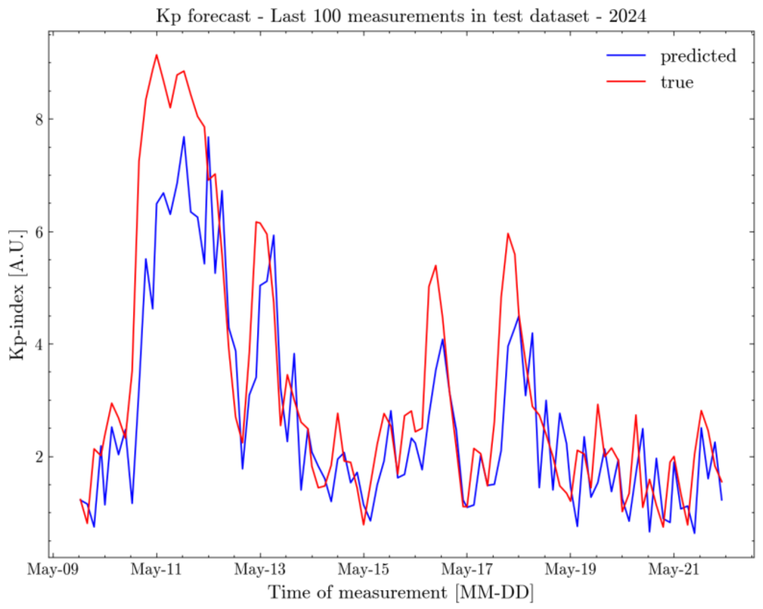
Forecasts made further into the future

The forecasts we showed in the main slideshow were made for the first 100 measurements in the test dataset, i.e. right after the train dataset.

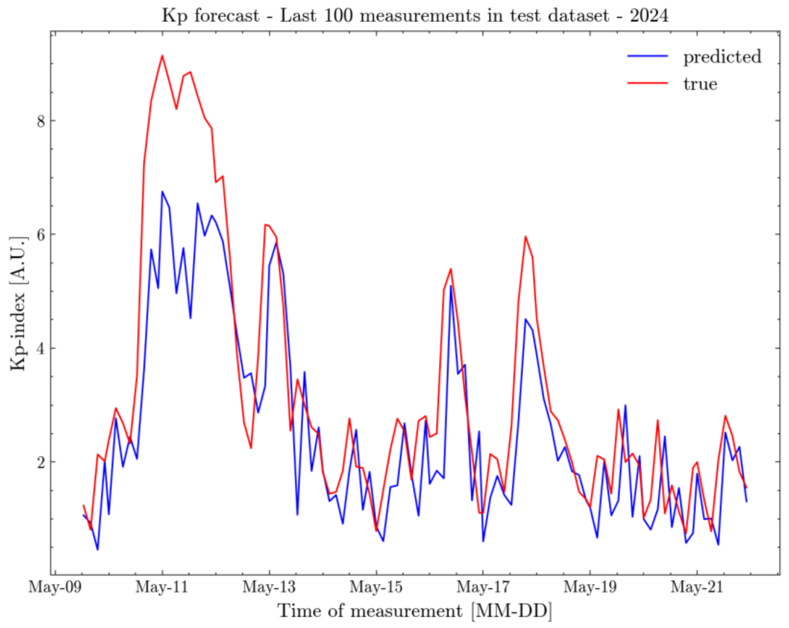
In the next slides we show forecasts that have been made for measurements that are more recent, i.e. for the last 100 measurements in the test dataset (may 2024). We show that the model can still make good forecasts years away from the training dataset.

Note: 11th-13th may 2024 had a historical solar storm, that is a bit difficult for the model to fit. However, given our criteria of Kp>=5 for the classification of a storm, the model correctly predicts (depending on model) that the storm is present within this time period.

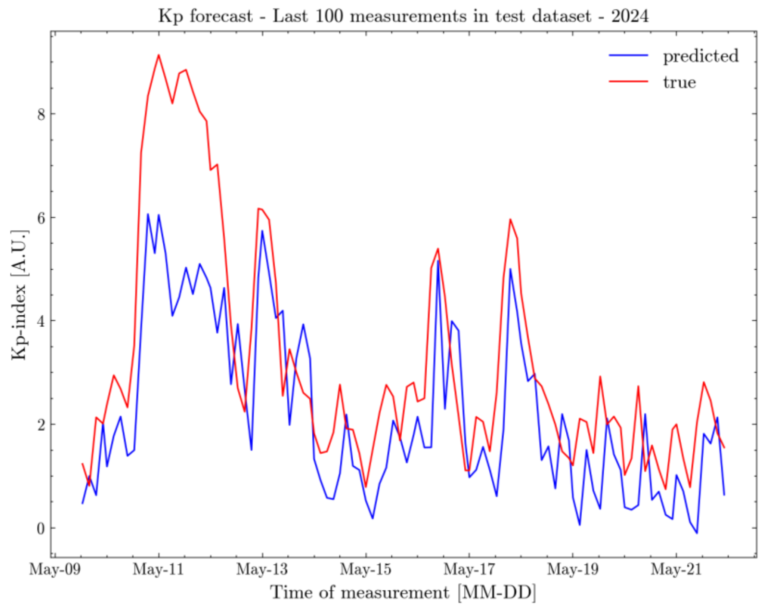
Forecasts made further into the future



Small regression model
80% train, 50 epochs, $LR = 1.5 \cdot 10^{-3}$



Small regression model
80% train, 50 epochs, $LR = 5 \cdot 10^{-4}$



Large regression model
25% train, 50 epochs, $LR = 10^{-3}$