Notes 2025-03-31

Carrie Hashimoto

Table of contents

This week

- ☐ Train unet with 24 time steps, 15 days apart, and test on the next year with same time spacing
- ☐ Give stats for time and accuracy for different model runs
- ☐ Find NLDAS nc files instead of grib
- ☐ Figure out how to regrid, either using ESMF or WRF-Hydro tutorial tools
- ☐ Figure out how to replace the precipitation with nexrad (4 km, whereas NLDAS2 is 12 km) – for now, if you figure out met data regridding, try running with NLDAS
- ☐ Check what NLDAS uses
- ☐ Check units on all met data

WRF-Hydro

Getting netcdf NLDAS data:

The dataset linked in the tutorial, where I got data from at first:

- Download data for 2011-08-26 to 2011-09-02 since that's what the tutorial needs

```
def nldas_urls(dates):
    """
    Get a list of two string dates with format 'YYYYMMDD.hhmm' and create urls to NLDAS_FORA0125 data for t
    """

    base_url = "https://hydro1.gesdisc.eosdis.nasa.gov/data/NLDAS/NLDAS_FORA0125_H.2.0"
    urls = []

    # Take the floor of the start day hour and ceiling of the end day hour
    # Since we have NLDAS data on the hour only
    start_day_num = datetime.strptime(dates[0], "%Y%m%d.%H%M")
    end_day_num = datetime.strptime(dates[1], "%Y%m%d.%H%M")

    start_day_num = start_day_num.replace(minute=0)

    if end_day_num.minute != 0:
        end_day_num = end_day_num.replace(minute=0, hour=end_day_num.hour + 1)

    times = pd.date_range(start_day_num,end_day_num,freq='h')

    # return times
    # # Make the urls
    urls = [os.path.join(base_url, str(x.year), str(x.timetuple().tm_yday),
                         ("NLDAS_FORA0125_H.A" + str(x.year) + str(x.month).rjust(2, '0') + str(x.day).rju
                          "." + str(x.hour).rjust(2, '0').ljust(3, "0") + str(x.second) + ".020.nc")).repl
    return urls
```

NLDAS_FORA0125_H dataset

- 1/8th degree ~ 12 km
- Non-precipitation data come from NCEP North American Regional Reanalysis data, which have 32-km resolution and 3-hour frequency
- NARR data used in NLDAS-2 are interpolated to 1/8 degree and disaggregated to 1-hour frequency
- Surface pressure, surfae downward longwave radiation, near-surface air temprature, and near-surface specific humidity are adjusted vertically to account for differences in terrain height between NARR and NLDAS

NLDAS_FORA0125_H dataset

- Surface downward shortwave radiation is a bias-corrected version of NARR surface downward shortwave radiation, incorporating information from 1996-2000 GOES-based surface downward shortwave radiation
- Potential evaporation field is from NARR, which uses modified Penman equation
- Precipitation is generated from temporal disaggregation of gauge-only Climate Prediction Center analysis of daily precipitation on NLDAS grid with orthographic adjustment using PRISM
  - Disaggregated using WSR-88D Doppler radar precip. estimates, 8-km CMORPH hourly precip. analysis, or NARR-simulated precip.

  - These data sources are only to get weights to disaggregate; their precip. values aren't used directly

NLDAS_FORA0125_H dataset

- They store the data in year > date (n/365) > page with download links for hourly data
- Made a script to convert desired date range to urls
- Downloaded date range of interest for example case (2011-08-26 to 2011-09-02)

NLDAS data and units vs needed data

Existing:

- time_bnds
- Tair (K)
- Qair (kg kg-1)
- PSurf (Pa)
- Wind_E (m s-1)
- Wind_N (m s-1)
- LWdown (W m-2)
- CRainf_frac
- CAPE
- PotEvap
- Rainf (kg m-2)
- SWdown (W m-2)

NLDAS data and units vs needed data

Needed:

- SWDOWN: shortwave rad (w m-2)
- LWDOWN: longwave rad (w m-2)
- Q2D: specific humidity (kg/kg)
- T2D: air temperature (K)
- PSFS: surface pressure (Pa)
- U2D: near surface wind u dir (m/s)
- V2D: near surface wind v dir (m/s)
- RAINRATE: precip rate (kg/m2/s or mm/s)

Time units are in 1 hour

Met forcing regridding

- NLDAS variables have different names than expected, even though NLDAS ncs are supposed to be the input
- Supposed to have a config file in the tutorial docker, but the data are missing
- Find some other source for config file: WrfHydroForcing repo has lots of config files, so select one and adapt it to this example

Ex: template_forcing_engine_Short.config

UNet

- Edit the data loader so that instead of loading a continuous period of data, you choose the indices of both time and tiles manually

```
class CombinedDataset(Dataset):
    def __init__(self, root_dir, time_inds=None, listoftiles=None, listofstatic=None, transform=None):

        # Assuming list of tiles has tile naming system present in the directory (I'm using tile_X_Y)
        self.root_dir = root_dir

        # If no tiles list provided, use all
        if not listoftiles:
            listoftiles = sorted(os.listdir(root_dir))

        self.tiles = listoftiles # List of tile directories

        if not time_inds:
#             print(os.path.join(root_dir, self.tiles[0]))
            ds = xr.open_dataset(os.path.join(root_dir, self.tiles[0], "dynamic.nc"))
            time_inds = list(range(0,(len(ds.time.values))))

        self.time_inds = time_inds

#         print(self.time_inds)
        self.listofstatic = listofstatic if listofstatic else [] # if none give, just don't subset later
        self.transform = transform

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

    def load_static_nc(self, path, static_vars=None):
        ds = xr.open_dataset(path)
        if static_vars:
            ds = ds[static_vars]
        data = ds.to_array().values  # shape: (C_static, H, W)
        ds.close()
        return data

    def load_dynamic_nc(self, path, time_inds):
        ds = xr.open_dataset(path)
        ds = ds.isel(time=time_inds)
        data = ds.to_array().values  # shape: (num_dynamic_vars, T, H, W)
        ds.close()
        return data

    def load_target_nc(self, path, time_inds):
        ds = xr.open_dataset(path)
        ds = ds.isel(time=time_inds)
        data = ds.to_array().values  # shape: (C_target, H, W)
        ds.close()
```

```
        return data

    def __getitem__(self, idx):
        tile_name = self.tiles[idx]
        tile_path = os.path.join(self.root_dir, tile_name)

        # Paths to NetCDF files
        dynamic_path = os.path.join(tile_path, "dynamic.nc")
        static_path = os.path.join(tile_path, "static.nc")
        target_path = os.path.join(tile_path, "target.nc")

        # Load data using the specific methods
        dynamic_data = self.load_dynamic_nc(dynamic_path, self.time_inds)   # (C_dyn, T, H, W)
        static_data = self.load_static_nc(static_path, self.listofstatic)     # (C_static, H, W)
        target_data = self.load_target_nc(target_path, self.time_inds)         # (C_target, H, W)

        # Return them separately (they'll be fused in the model)
        static_data = torch.tensor(static_data, dtype=torch.float32)
        dynamic_data = torch.tensor(dynamic_data, dtype=torch.float32)
        target_data = torch.tensor(target_data, dtype=torch.float32)

        if self.transform:
            static_data = self.transform(static_data)
            dynamic_data = self.transform(dynamic_data)
            target_data = self.transform(target_data)

        return static_data, dynamic_data, target_data
```

- Edit the training script so that it takes in training and target data with any time and tile subset

Training approaches

- In order to use multiple time steps, just treat each time step / tile combo as a separate sample

1. 23 timesteps selected from 2016 for train; 23 timesteps selected from 2017 for target
2. 46 timesteps selected from 2016 and 2017 split into a checkerboard with one half for training and one half for target

```
workspace = os.path.dirname(os.getcwd())
ds = xr.open_dataset(os.path.join(workspace, "data", "combined_output", "tiles2", "tile_0_0", "dynamic.nc"
time_inds = list(range(0,(len(ds.time.values))))
years = ds.time.values.astype('datetime64[Y]').astype(int) + 1970
train_time_inds = list(range(list(years).index(2016),list(years).index(2017), 10))
print(ds.time.values[train_time_inds])
```

```
test_time_inds = list(range(list(years).index(2017),list(years).index(2018), 10))
print(ds.time.values[test_time_inds])
```

2016/2017 data split

Figure 1: Error plot

2016/2017 data split

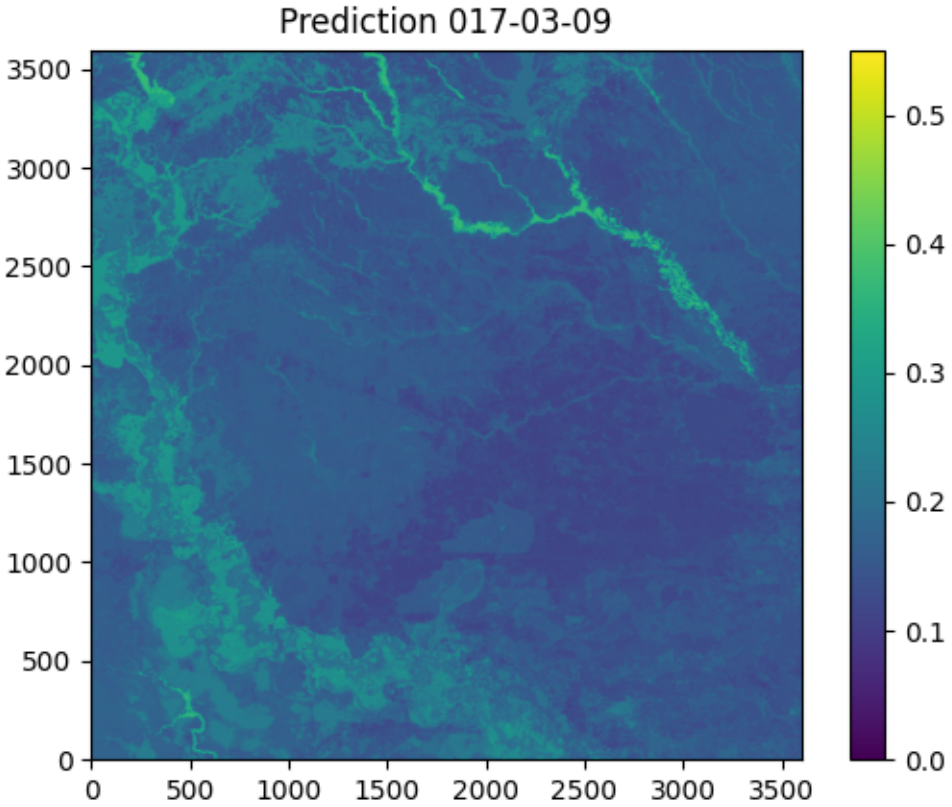

Figure 2: Example prediction

2016/2017 data split

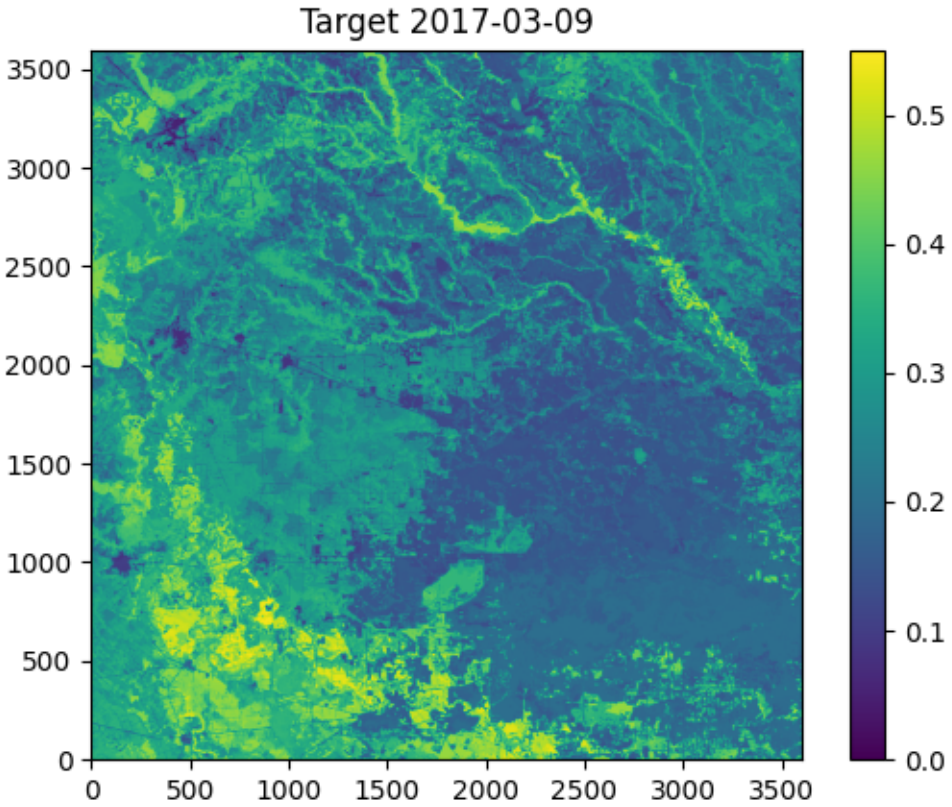

Figure 3: Example target data

2016/2017 data split

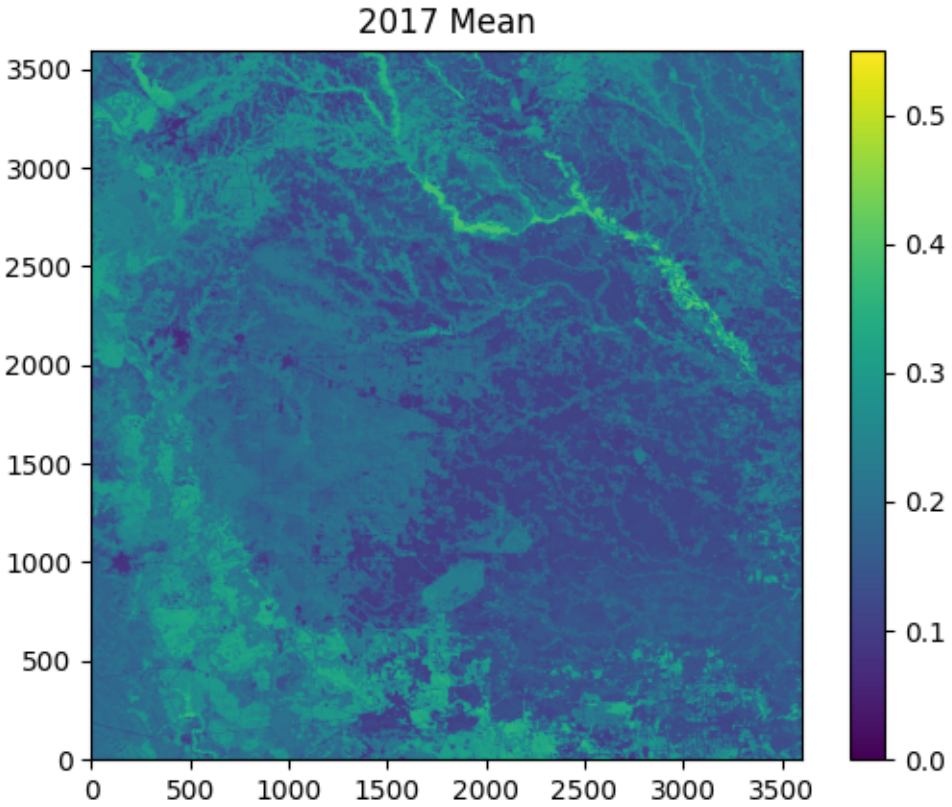

Figure 4: Predictions are close to mean

Checkerboard data split

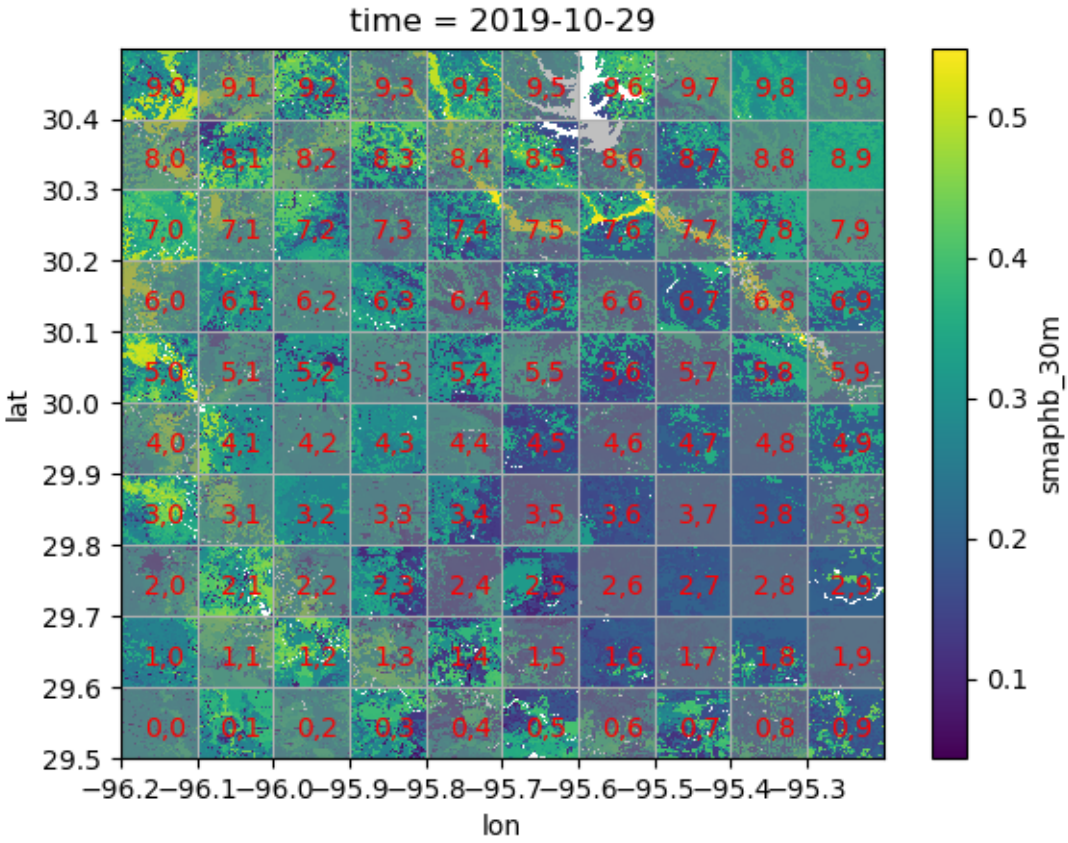

Figure 5: Checkerboard

Training approaches - troubleshooting

Why isn't the temporal component showing up? Maybe an issue with fusion with or matching of dynamic and static data?

```
import torch
import torch.nn as nn
import torch.nn.functional as F

def pad_to_target(x, target_size):
    _, _, h, w = x.shape
    target_h, target_w = target_size
    pad_h = target_h - h
    pad_w = target_w - w
    pad_top = pad_h // 2
    pad_bottom = pad_h - pad_top
```

```
    pad_left = pad_w // 2
    pad_right = pad_w - pad_left
    return F.pad(x, [pad_left, pad_right, pad_top, pad_bottom])

class ConvBlock(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.conv1 = nn.Conv2d(in_c, out_c, kernel_size=3, padding='same')
        self.conv2 = nn.Conv2d(out_c, out_c, kernel_size=3, padding='same')

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        return x

class DownBlock(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.conv_block = ConvBlock(in_c, out_c)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        conv_out = self.conv_block(x)
        if conv_out.size(-2) > 1 and conv_out.size(-1) > 1:
            downsampled = self.pool(conv_out)
        else:
            downsampled = conv_out
        return downsampled, conv_out

class UpBlock(nn.Module):
    def __init__(self, in_channels_up, skip_channels, out_channels):
        super().__init__()
        self.upconv = nn.ConvTranspose2d(in_channels_up, out_channels,
                                         kernel_size=2, stride=2)
        self.conv_block = ConvBlock(out_channels + skip_channels, out_channels)

    def forward(self, x, skip):
        x = self.upconv(x)

        if x.size(2) != skip.size(2) or x.size(3) != skip.size(3):
            x = pad_to_target(x, (skip.size(2), skip.size(3)))

        # Now both tensors should have the same H×W
        x = torch.cat([skip, x], dim=1)
        x = self.conv_block(x)
        return x


class UNetFusion(nn.Module):
    def __init__(self, in_channels_static=86, time_steps=0, in_channels_dynamic=2, out_channels=1):
```

```
        super().__init__()

        self.down_channels = [64, 128, 256, 512, 512, 512, 512, 512, 512]

        in_c = in_channels_static
        self.down_blocks = nn.ModuleList()
        for out_c in self.down_channels:
            self.down_blocks.append(DownBlock(in_c, out_c))
            in_c = out_c

        self.fuse_conv = nn.Sequential(
            nn.Conv2d(self.down_channels[-1] + in_channels_dynamic, self.down_channels[-1], kernel_size=1)
            nn.ReLU(inplace=True)
        )

        self.up_blocks = nn.ModuleList()
        reversed_channels = list(reversed(self.down_channels))

        for i in range(len(reversed_channels) - 1):
            up_in   = reversed_channels[i]       # channels from the previous up block
            skip_c  = reversed_channels[i + 1]   # channels from the skip
            out_c   = reversed_channels[i + 1]   # final output
            self.up_blocks.append(UpBlock(up_in, skip_c, out_c))

        self.out_conv = nn.Conv2d(self.down_channels[0], out_channels, kernel_size=1)

    def forward(self, static_x, dynamic_x):

        skips = []
        x = static_x
        for down in self.down_blocks:
            x, skip = down(x)
            skips.append(skip)
#             print(f"DownBlock: x shape: {x.shape}, skip shape: {skip.shape}")

        x = torch.cat([x, dynamic_x], dim=1)
        x = self.fuse_conv(x)

        used_skips = skips[:-1] # discard last skip
        used_skips = used_skips[::-1]

        decoded = x
        for i in range(len(self.up_blocks)):
            skip = used_skips[i]
            decoded = self.up_blocks[i](decoded, skip)
#             print(f"UpBlock: decoded shape: {decoded.shape}, skip shape: {skip.shape}")

        out = self.out_conv(decoded)
        return out
```

This week

- ☐ Update GitHub
- ☐ Clean up tile naming system: currently directories named tile_Y_X – switch to be tile_X_Y?
- ☐ Update geospatial_env environment yaml to include new packages I installed: hvplot, pyyaml, jupyter_bokeh (check if there are others)