HW5_Precipitation_Nowcasting_PyTorch_Student_2024

March 19, 2024

1 Precipitation Nowcasting using Neural Networks

In this exercise, you are going to build a set of deep learning models on a real world task using PyTorch. PyTorch is an open source machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR).

1.1 Setting up to use the gpu

Before we start, we need to change the environment of Colab to use GPU. Do so by:

Runtime -> Change runtime type -> Hardware accelerator -> GPU

1.2 Deep Neural Networks with PyTorch

To complete this exercise, you will need to build deep learning models for precipitation nowcasting. You will build a subset of the models shown below: - Fully Connected (Feedforward) Neural Network - Two-Dimentional Convolution Neural Network (2D-CNN) - Recurrent Neural Network with Gated Recurrent Unit (GRU)

and one more model of your choice to achieve the highest score possible.

We provide the code for data cleaning and some starter code for PyTorch in this notebook but feel free to modify those parts to suit your needs. Feel free to use additional libraries (e.g. scikit-learn) as long as you have a model for each type mentioned above.

This notebook assumes you have already installed PyTorch with python3 and had GPU enabled. If you run this exercise on Colab you are all set.

1.3 Precipitation Nowcasting

Precipitation nowcasting is the the task of predicting the amount of rainfall in a certain region given some kind of sensor data. The term nowcasting refers to tasks that try to predict the current or near future conditions (within 6 hours).

You will be given satellite images in 3 different bands covering a 5 by 5 region from different parts of Thailand. In other words, your input will be a 5x5x3 image. Your task is to predict the amount of rainfall in the center pixel. You will first do the prediction using just a simple fully-connected neural network that view each pixel as different input features.

Since the your input is basically an image, we will then view the input as an image and apply CNN to do the prediction. Finally, we can also add a time component since weather prediction can benefit greatly using previous time frames. Each data point actually contain 5 time steps, so each input data point has a size of 5x5x5x3 (time x height x width x channel), and the output data has a size of 5 (time). You will use this time information when you work with RNNs.

Finally, we would like to thank the Thai Meteorological Department for providing the data for this assignment.

[1]:
```
!nvidia-smi
```

```
Fri Mar 15 16:46:47 2024
+-------------------------------------------------------------------------------
--------+
| NVIDIA-SMI 535.129.03             Driver Version: 535.129.03   CUDA Version:
12.2     |
|-----------------------------------------+----------------------+--------------
--------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
MIG M. |
|=========================================+======================+==============
========|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
0 |
| N/A   39C    P8               9W /  70W |      0MiB / 15360MiB |      0%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+--------------
--------+
|   1  Tesla T4                       Off | 00000000:00:05.0 Off |
0 |
| N/A   39C    P8               9W /  70W |      0MiB / 15360MiB |      0%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+--------------
--------+

+-------------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                            GPU
Memory |
|        ID   ID
Usage      |
```

```
|===============================================================================
========|
|  No running processes found
|
+-------------------------------------------------------------------------------
--------+
```

[2]:
```
# For summarizing and visualizing models
!pip install torchinfo
!pip install torchviz
```

```
Requirement already satisfied: torchinfo in /opt/conda/lib/python3.10/site-
packages (1.8.0)
Collecting torchviz
  Downloading torchviz-0.0.2.tar.gz (4.9 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: torch in /opt/conda/lib/python3.10/site-
packages (from torchviz) (2.1.2)
Requirement already satisfied: graphviz in /opt/conda/lib/python3.10/site-
packages (from torchviz) (0.20.1)
Requirement already satisfied: filelock in /opt/conda/lib/python3.10/site-
packages (from torch->torchviz) (3.13.1)
Requirement already satisfied: typing-extensions in
/opt/conda/lib/python3.10/site-packages (from torch->torchviz) (4.9.0)
Requirement already satisfied: sympy in /opt/conda/lib/python3.10/site-packages
(from torch->torchviz) (1.12)
Requirement already satisfied: networkx in /opt/conda/lib/python3.10/site-
packages (from torch->torchviz) (3.2.1)
Requirement already satisfied: jinja2 in /opt/conda/lib/python3.10/site-packages
(from torch->torchviz) (3.1.2)
Requirement already satisfied: fsspec in /opt/conda/lib/python3.10/site-packages
(from torch->torchviz) (2024.2.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/opt/conda/lib/python3.10/site-packages (from jinja2->torch->torchviz) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in /opt/conda/lib/python3.10/site-
packages (from sympy->torch->torchviz) (1.3.0)
Building wheels for collected packages: torchviz
  Building wheel for torchviz (setup.py) ... done
  Created wheel for torchviz: filename=torchviz-0.0.2-py3-none-any.whl
size=4131
sha256=f4a2d03634b603c4dfd5287046f0a3f6b7a874c8d14b6c031d74023b605e2017
  Stored in directory: /root/.cache/pip/wheels/4c/97/88/a02973217949e0db0c9f4346
d154085f4725f99c4f15a87094
Successfully built torchviz
Installing collected packages: torchviz
Successfully installed torchviz-0.0.2
```

1.4 Weights and Biases

Weights and Biases (wandb) is an experiment tracking tool for machine learning. It can log and visualize experiments in real time. It supports many popular ML frameworks, and obviously PyTorch is one of them. In this notebook you will learn how to log general metrics like losses, parameter distributions, and gradient distribution with wandb.

To install wandb, run the cell below

[3]:
```
!pip install wandb
```

```
Requirement already satisfied: wandb in /opt/conda/lib/python3.10/site-packages
(0.16.3)
Requirement already satisfied: Click!=8.0.0,>=7.1 in
/opt/conda/lib/python3.10/site-packages (from wandb) (8.1.7)
Requirement already satisfied: GitPython!=3.1.29,>=1.0.0 in
/opt/conda/lib/python3.10/site-packages (from wandb) (3.1.41)
Requirement already satisfied: requests<3,>=2.0.0 in
/opt/conda/lib/python3.10/site-packages (from wandb) (2.31.0)
Requirement already satisfied: psutil>=5.0.0 in /opt/conda/lib/python3.10/site-
packages (from wandb) (5.9.3)
Requirement already satisfied: sentry-sdk>=1.0.0 in
/opt/conda/lib/python3.10/site-packages (from wandb) (1.40.5)
Requirement already satisfied: docker-pycreds>=0.4.0 in
/opt/conda/lib/python3.10/site-packages (from wandb) (0.4.0)
Requirement already satisfied: PyYAML in /opt/conda/lib/python3.10/site-packages
(from wandb) (6.0.1)
Requirement already satisfied: setproctitle in /opt/conda/lib/python3.10/site-
packages (from wandb) (1.3.3)
Requirement already satisfied: setuptools in /opt/conda/lib/python3.10/site-
packages (from wandb) (69.0.3)
Requirement already satisfied: appdirs>=1.4.3 in /opt/conda/lib/python3.10/site-
packages (from wandb) (1.4.4)
Requirement already satisfied: protobuf!=4.21.0,<5,>=3.19.0 in
/opt/conda/lib/python3.10/site-packages (from wandb) (3.20.3)
Requirement already satisfied: six>=1.4.0 in /opt/conda/lib/python3.10/site-
packages (from docker-pycreds>=0.4.0->wandb) (1.16.0)
Requirement already satisfied: gitdb<5,>=4.0.1 in
/opt/conda/lib/python3.10/site-packages (from GitPython!=3.1.29,>=1.0.0->wandb)
(4.0.11)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/conda/lib/python3.10/site-packages (from requests<3,>=2.0.0->wandb) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.10/site-
packages (from requests<3,>=2.0.0->wandb) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/opt/conda/lib/python3.10/site-packages (from requests<3,>=2.0.0->wandb)
(1.26.18)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/conda/lib/python3.10/site-packages (from requests<3,>=2.0.0->wandb)
```

```
(2024.2.2)
Requirement already satisfied: smmap<6,>=3.0.1 in
/opt/conda/lib/python3.10/site-packages (from
gitdb<5,>=4.0.1->GitPython!=3.1.29,>=1.0.0->wandb) (5.0.1)
```

1.5 Setup

1. Register Wandb account (and confirm your email)
2. `wandb login` and copy paste the API key when prompt

[4]:
```
!wandb login
```

```
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server
locally: https://wandb.me/wandb-server)
wandb: You can find your API key in your browser here:
https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter, or press
ctrl+c to quit:
Aborted!
```

[1]:
```python
import os
import numpy as np
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import urllib
import wandb
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms

from sklearn import preprocessing
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torchinfo import summary
from tqdm.notebook import tqdm

torch.__version__ # 1.10.0+cu111
```

[1]: '2.1.1'

1.6 Loading the data

Get the data set by going here and click add to drive.

[ ]:
```python
from google.colab import drive
drive.mount('/content/gdrive/')
```

[7]:
```
!tar -xvf '/content/gdrive/My Drive/nowcastingHWdataset.tar.gz'
```

```
dataset/features-m10.pk

dataset/features-m6.pk

dataset/features-m7.pk

dataset/features-m8.pk

dataset/features-m9.pk

dataset/labels-m10.pk

dataset/labels-m6.pk

dataset/labels-m7.pk

dataset/labels-m8.pk

dataset/labels-m9.pk
```

2 Data Explanation

The data is an hourly measurement of water vapor in the atmosphere, and two infrared measurements of cloud imagery on a latitude-longitude coordinate. Each measurement is illustrated below as an image. These three features are included as different channels in your input data.

We also provide the hourly precipitation (rainfall) records in the month of June, July, August, September, and October from weather stations spreaded around the country. A 5x5 grid around each weather station at a particular time will be paired with the precipitation recorded at the corresponding station as input and output data. Finally, five adjacent timesteps are stacked into one sequence.

The month of June-August are provided as training data, while the months of September and October are used as validation and test sets, respectively.

3 Reading data

[2]:
```
def read_data(months, data_dir='dataset'):
    features = np.array([], dtype=np.float32).reshape(0,5,5,5,3)
    labels = np.array([], dtype=np.float32).reshape(0,5)
    for m in months:
        filename = 'features-m{}.pk'.format(m)
        with open(os.path.join(data_dir,filename), 'rb') as file:
            features_temp = pickle.load(file)
        features = np.concatenate((features, features_temp), axis=0)
```

```
        filename = 'labels-m{}.pk'.format(m)
        with open(os.path.join(data_dir,filename), 'rb') as file:
            labels_temp = pickle.load(file)
        labels = np.concatenate((labels, labels_temp), axis=0)

    return features, labels
```

[3]:
```
# use data from month 6,7,8 as training set
x_train, y_train = read_data(months=[6,7,8])

# use data from month 9 as validation set
x_val, y_val = read_data(months=[9])

# use data from month 10 as test set
x_test, y_test = read_data(months=[10])

print('x_train shape:',x_train.shape)
print('y_train shape:', y_train.shape, '\n')
print('x_val shape:',x_val.shape)
print('y_val shape:', y_val.shape, '\n')
print('x_test shape:',x_test.shape)
print('y_test shape:', y_test.shape)
```

```
x_train shape: (229548, 5, 5, 5, 3)
y_train shape: (229548, 5)

x_val shape: (92839, 5, 5, 5, 3)
y_val shape: (92839, 5)

x_test shape: (111715, 5, 5, 5, 3)
y_test shape: (111715, 5)
```

features - dim 0: number of entries - dim 1: number of time-steps in ascending order - dim 2,3: a 5x5 grid around rain-measued station - dim 4: water vapor and two cloud imagenaries

labels - dim 0: number of entries - dim 1: number of precipitation for each time-step

4 Three-Layer Feedforward Neural Networks

[4]:
```
# Dataset need to be reshaped to make it suitable for feedforword model
def preprocess_for_ff(x_train, y_train, x_val, y_val):
    x_train_ff = x_train.reshape((-1, 5*5*3))
    y_train_ff = y_train.reshape((-1, 1))
    x_val_ff = x_val.reshape((-1, 5*5*3))
    y_val_ff = y_val.reshape((-1, 1))
    x_test_ff = x_test.reshape((-1, 5*5*3))
    y_test_ff = y_test.reshape((-1, 1))
```

```
    return x_train_ff, y_train_ff, x_val_ff, y_val_ff, x_test_ff, y_test_ff

x_train_ff, y_train_ff, x_val_ff, y_val_ff, x_test_ff, y_test_ff =␣
 ↪preprocess_for_ff(x_train, y_train, x_val, y_val)
print(x_train_ff.shape, y_train_ff.shape)
print(x_val_ff.shape, y_val_ff.shape)
print(x_test_ff.shape, y_test_ff.shape)
```

```
(1147740, 75) (1147740, 1)
(464195, 75) (464195, 1)
(558575, 75) (558575, 1)
```

4.0.1 TODO#1

Explain each line of code in the function preprocess_for_ff()

Ans: - for x we reshape(flatten) into each time-step data which having $5 \times 5 \times 3$ features which is 5x5 grid and water vapor and two cloud imagenaries(total 3) respectively. - for y just only flatten 5 time-step.

4.1 Dataset

To prepare a DataLoader in order to feed data into the model, we need to create a `torch.utils.data.Dataset` object first. (Learn more about it here)

Dataset is a simple class that the DataLoader will get data from, most of its functionality comes from `__getitem__(self, index)` method, which will return a single data point (both input and label). In real world scenarios the method can do some other stuffs such as

1. Load images

If your input (x) are images. Oftentimes you won't be able to fit all the training images into your RAM. Thus, you should pass an array (or list) of image path into the dataloader, and the `__getitem__` will be the one who dynamically loads the actual image from the harddisk for you.

2. Data Normalization

Data normalization helps improve stability of training. Unnormalized data can cause gradients to explode. There are many variants of normalization, but in this notebook we will use either minmax or z-score (std) normalization. Read this (or google) if you wish to learn more about data normalization.

3. Data Augmentation

In computer vision, you might want to apply small changes to the images you use in training (adjust brightness, contrast, rotation) so that the model will generalize better on unseen data. There are two kinds of augmentation: static and dynamic. Static augmentation will augment images and save to disk as a new dataset. On the other hand, rather than applying the change initially and use the same change on each image every epoch, dynamic augmentation will augment each data differently for each epoch. Note that augmentation is usually done on the CPU and you might be bounded by the CPU instead. PyTorch has a dedicated documentation about data augmentation if you want to know more.

[5]:
```
class RainfallDatasetFF(Dataset):
    def __init__(self, x, y, normalizer):
        self.x = x.astype(np.float32)
        self.y = y.astype(np.float32)
        self.normalizer = normalizer
        print(self.x.shape)
        print(self.y.shape)

    def __getitem__(self, index):
        x = self.x[index] # Retrieve data
        x = self.normalizer.transform(x.reshape(1, -1)) # Normalize
        y = self.y[index]
        return x, y

    def __len__(self):
        return self.x.shape[0]
```

[6]:
```
def normalizer_std(X):
    scaler = preprocessing.StandardScaler().fit(X)
    return scaler

def normalizer_minmax(X):
    scaler = preprocessing.MinMaxScaler().fit(X)
    return scaler
```

[7]:
```
normalizer = normalizer_std(x_train_ff) # We will normalize everything based on␣
 ↪x_train

train_dataset = RainfallDatasetFF(x_train_ff, y_train_ff, normalizer)
val_dataset = RainfallDatasetFF(x_val_ff, y_val_ff, normalizer)
test_dataset = RainfallDatasetFF(x_test_ff, y_test_ff, normalizer)
```

```
(1147740, 75)
(1147740, 1)
(464195, 75)
(464195, 1)
(558575, 75)
(558575, 1)
```

4.2 DataLoader

DataLoader feeds data from our dataset into the model. We can freely customize batch size, data shuffle for each data split, and much more with DataLoader class. If you're curious about what can you do with PyTorch's DataLoader, you can check this documentation

[8]:

```
train_loader = DataLoader(train_dataset, batch_size=1024, shuffle=True,␣
 ↪pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=1024, shuffle=False,␣
 ↪pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=1024, shuffle=False,␣
 ↪pin_memory=True)
```

4.3 Loss Function

PyTorch has many loss functions readily available for use. We can also write our own custom loss function as well. But for now, we will use PyTorch's built-in mean squared error loss

[9]:
```
loss_fn = nn.MSELoss()
```

4.3.1 TODO#2

Why is the loss MSE?

Ans: because we predict amount of rainfall which is numerical data so we use MSE to measure what difference our prediction and ground truth plus MSE is easier to differentiate.

4.4 Device

Unlike Tensorflow/Keras, PyTorch allows user to freely put any Tensor or objects (loss functions, models, optimizers, etc.) in CPU or GPU. By default, all objects created will be in CPU. In order to use GPU we will have to supply `device = torch.device("cuda")` into the objects to move it to GPU. You will usually see the syntax like `object.to(device)` for moving CPU object to GPU, or `o = Object(..., device=device)` to create the object in the GPU.

[10]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

4.5 Model

Below, the code for creating a 3-layers fully connected neural network in PyTorch is provided. Run the code and make sure you understand what you are doing. Then, report the results.

[11]:
```
class FeedForwardNN(nn.Module):
    def __init__(self, hidden_size=200):
        super(FeedForwardNN, self).__init__()
        self.ff1 = nn.Linear(75, hidden_size)
        self.ff2 = nn.Linear(hidden_size, hidden_size)
        self.ff3 = nn.Linear(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, 1)

    def forward(self, x):
        hd1 = F.relu(self.ff1(x))
        hd2 = F.relu(self.ff2(hd1))
        y = F.relu(self.ff3(hd2))
        y = self.out(y)
```

```
        return y.reshape(-1, 1)
```

4.5.1 TODO#3

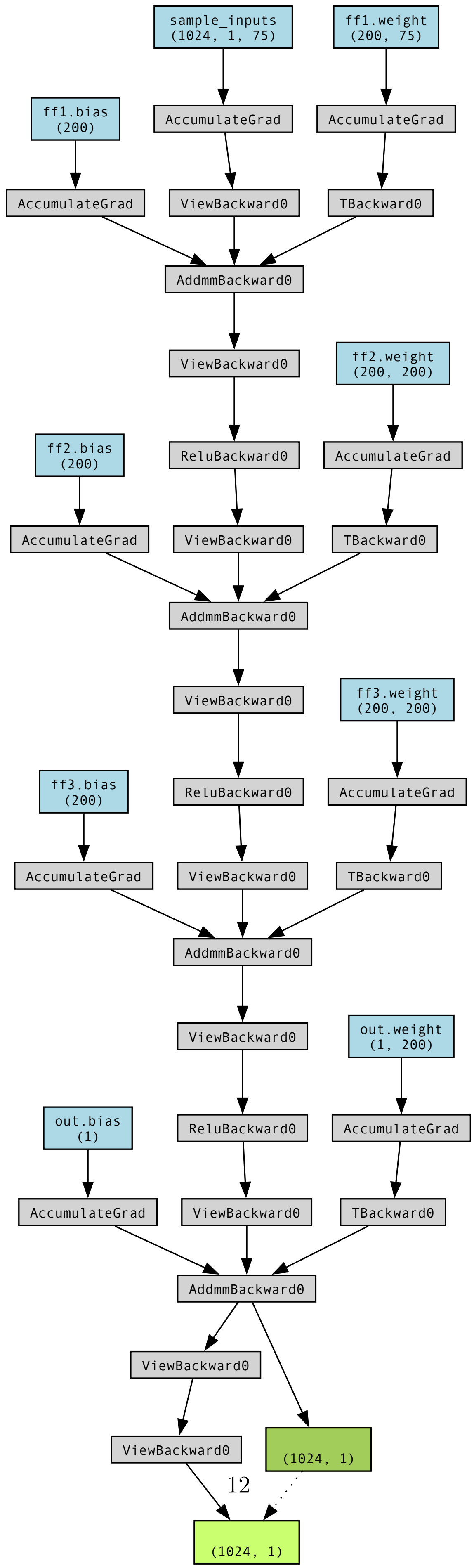
What is the activation function in the final dense layer? and why? Do you think there is a better activation function for the final layer?

Ans: linear layer. I think linear layer is appropriate because our objective is to predict numerical data.

[91]:
```
# Hyperparameters and other configs
config = {
    'architecture': 'feedforward',
    'lr': 0.01,
    'hidden_size': 200,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10
}

# Model
model_ff = FeedForwardNN(hidden_size=config['hidden_size'])
model_ff = model_ff.to(device)
optimizer = torch.optim.Adam(model_ff.parameters(), lr=config['lr'])
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    'min',
    factor=config['scheduler_factor'],
    patience=config['scheduler_patience'],
    min_lr=config['scheduler_min_lr']
)
```

[92]:
```
from torchviz import make_dot
# Visualize model with torchviz
sample_inputs = next(iter(train_loader))[0].requires_grad_(True)
sample_y = model_ff(sample_inputs.to(device))
make_dot(sample_y, params=dict(list(model_ff.
 ↪named_parameters())+[('sample_inputs', sample_inputs)]))
```

[92]:


sample_inputs
(1024, 1, 75)
ff1.weight
(200, 75)
ff1.bias
(200)
AccumulateGrad
AccumulateGrad
AccumulateGrad
ViewBackward0
TBackward0
AddmmBackward0
ViewBackward0
ff2.weight
(200, 200)
ff2.bias
(200)
ReluBackward0
AccumulateGrad
AccumulateGrad
ViewBackward0
TBackward0
AddmmBackward0
ViewBackward0
ff3.weight
(200, 200)
ff3.bias
(200)
ReluBackward0
AccumulateGrad
AccumulateGrad
ViewBackward0
TBackward0
AddmmBackward0
ViewBackward0
out.weight
(1, 200)
out.bias
(1)
ReluBackward0
AccumulateGrad
AccumulateGrad
ViewBackward0
TBackward0
AddmmBackward0
ViewBackward0
ViewBackward0
(1024, 1)
(1024, 1)

[93]:
```
summary(model_ff, input_size=(1024, 75))
```

[93]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
FeedForwardNN                             [1024, 1]                 --
Linear: 1-1                              [1024, 200]               15,200
Linear: 1-2                              [1024, 200]               40,200
Linear: 1-3                              [1024, 200]               40,200
Linear: 1-4                              [1024, 1]                 201
================================================================================
==========
Total params: 95,801
Trainable params: 95,801
Non-trainable params: 0
Total mult-adds (M): 98.10
================================================================================
==========
Input size (MB): 0.31
Forward/backward pass size (MB): 4.92
Params size (MB): 0.38
Estimated Total Size (MB): 5.61
================================================================================
==========
```

4.5.2 TODO#4

Explain why the first linear layer has number of parameters = 15200

Ans: The input has 75 featues ($5 \times 5 \times 3$) first hidden layer has 200 neurons $\Rightarrow 75 \times 200 + 200 = 15,200$

5 Training

[94]:
```
train_losses = []
val_losses = []
learning_rates = []

# Start wandb run
wandb.init(
    project='precipitation-nowcasting',
    config=config,
)
```

```
# Log parameters and gradients
wandb.watch(model_ff, log='all', log_freq=1)

for epoch in range(config['epochs']):  # loop over the dataset multiple times

    # Training
    train_loss = []
    current_lr = optimizer.param_groups[0]['lr']
    learning_rates.append(current_lr)

    # Flag model as training. Some layers behave differently in training and
    # inference modes, such as dropout, BN, etc.
    model_ff.train()

    print(f"Training epoch {epoch+1}...")
    print(f"Current LR: {current_lr}")

    for i, (inputs, y_true) in enumerate(tqdm(train_loader)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        # Reset the gradient
        optimizer.zero_grad()

        # Predict
        y_pred = model_ff(inputs)

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Compute gradient
        loss.backward()

        # Update parameters
        optimizer.step()

        # Log stuff
        train_loss.append(loss)

    avg_train_loss = torch.stack(train_loss).mean().item()
    train_losses.append(avg_train_loss)

    print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

    # Validation
    model_ff.eval()
```

```
with torch.no_grad(): # No gradient is required during validation
    print(f"Validating epoch {epoch+1}")
    val_loss = []
    for i, (inputs, y_true) in enumerate(tqdm(val_loader)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        # Predict
        y_pred = model_ff(inputs)

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Log stuff
        val_loss.append(loss)

    avg_val_loss = torch.stack(val_loss).mean().item()
    val_losses.append(avg_val_loss)
    print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

    # LR adjustment with scheduler
    scheduler.step(avg_val_loss)

    # Save checkpoint if val_loss is the best we got
    best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
    if avg_val_loss < best_val_loss:
        # Save whatever you want
        state = {
            'epoch': epoch,
            'model': model_ff.state_dict(),
            'optimizer': optimizer.state_dict(),
            'scheduler': scheduler.state_dict(),
            'train_loss': avg_train_loss,
            'val_loss': avg_val_loss,
            'best_val_loss': best_val_loss,
        }

        print(f"Saving new best model..")
        torch.save(state, 'model_ff.pth.tar')

wandb.log({
    'train_loss': avg_train_loss,
    'val_loss': avg_val_loss,
    'lr': current_lr,
})
```

```python
wandb.unwatch()
wandb.finish()
print('Finished Training')
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Training epoch 1...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 1 train loss: 1.9245
Validating epoch 1

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 1 val loss: 1.6585
Saving new best model..
Training epoch 2...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 2 train loss: 1.9211
Validating epoch 2

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 2 val loss: 1.6594
Training epoch 3...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 3 train loss: 1.9206
Validating epoch 3

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 3 val loss: 1.6610
Training epoch 4...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 4 train loss: 1.9208
Validating epoch 4

  0%|          | 0/454 [00:00<?, ?it/s]
```

```
Epoch 4 val loss: 1.6585
Training epoch 5...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 5 train loss: 1.9200
Validating epoch 5

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 5 val loss: 1.6583
Saving new best model..
Training epoch 6...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 6 train loss: 1.9198
Validating epoch 6

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 6 val loss: 1.6591
Training epoch 7...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 7 train loss: 1.9198
Validating epoch 7

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 7 val loss: 1.6587
Training epoch 8...
Current LR: 0.0004

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 8 train loss: 1.9184
Validating epoch 8

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 8 val loss: 1.6565
Saving new best model..
Training epoch 9...
Current LR: 0.0004

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 9 train loss: 1.9183
Validating epoch 9

  0%|          | 0/454 [00:00<?, ?it/s]
```

```
Epoch 9 val loss: 1.6566
Training epoch 10...
Current LR: 0.0004
```

```
  0%|          | 0/1121 [00:00<?, ?it/s]
```

```
Epoch 10 train loss: 1.9185
Validating epoch 10
```

```
  0%|          | 0/454 [00:00<?, ?it/s]
```

```
Epoch 10 val loss: 1.6564
Saving new best model..
```

```
VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
↪FloatProgress(value=1.0, max=1.0)))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

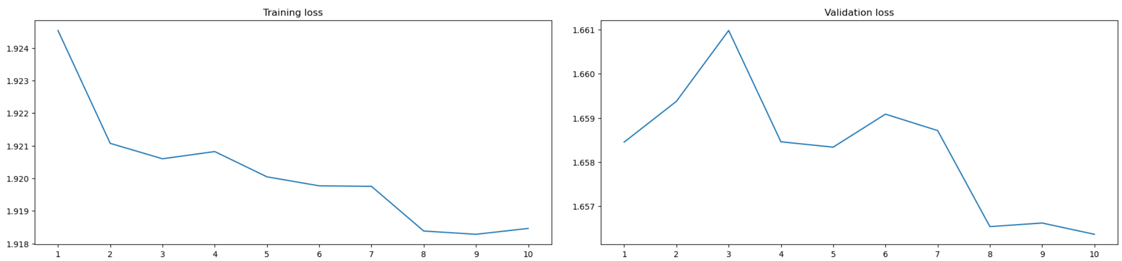
```
<IPython.core.display.HTML object>
```

```
Finished Training
```

5.0.1 TODO#5

Plot loss and val_loss as a function of epochs.

[95]:
```
fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.tight_layout(pad=3.0)

ax[0].set_xticks(range(1,11))
ax[1].set_xticks(range(1,11))

ax[0].set_title('Training loss')
ax[1].set_title('Validation loss')

ax[0].plot(range(1, 11), train_losses)
ax[1].plot(range(1, 11), val_losses)

plt.show()
```



5.0.2 TODO#6

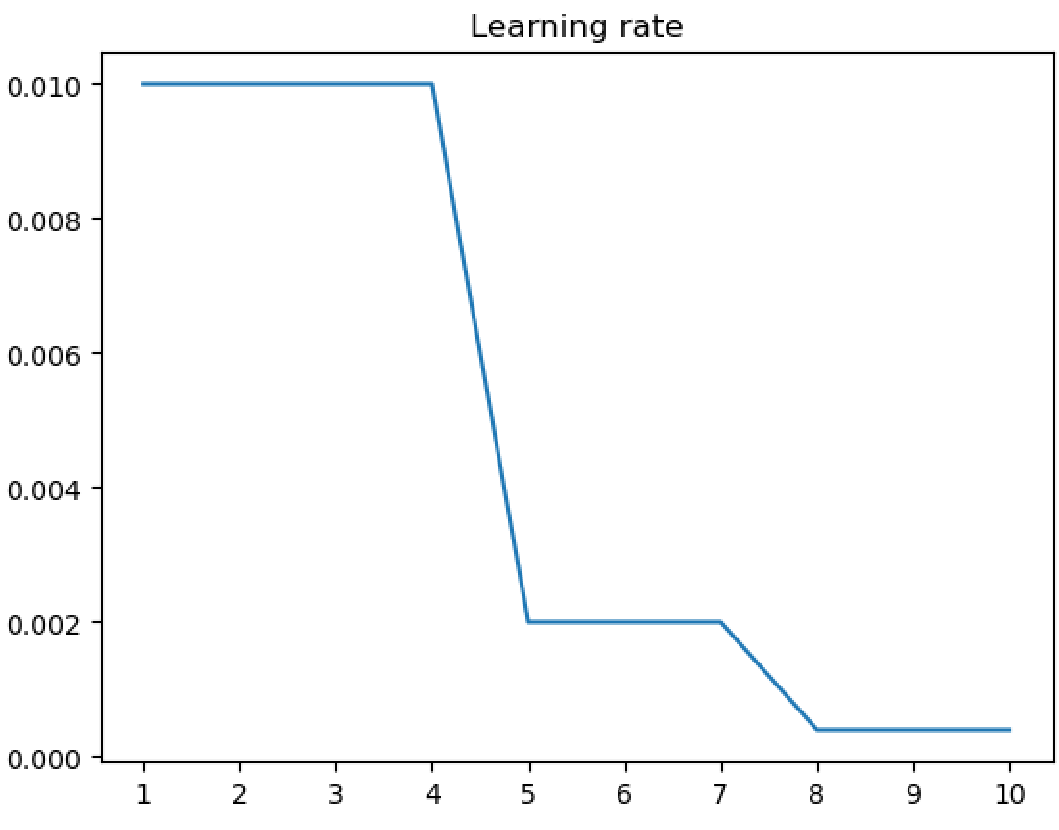
When does the model start to overfit?

Ans: epochs 3 and 6 due to training loss plateu and while validation loss is increasing

5.0.3 TODO#7

Plot the learning rate as a function of the epochs.

[96]:
```
plt.plot(range(1, 11), learning_rates)
plt.xticks(range(1, 11))
plt.title('Learning rate')
plt.show()
```



5.0.4 TODO#8

What makes the learning rate change? (hint: try to understand the scheduler ReduceLROnPlateau)

Ans: When there is no (less) improvement of our objective function(metrics).

6 Load Model

Use the code snippet below to load the model you just trained

[97]:
```python
checkpoint = torch.load('model_ff.pth.tar')
loaded_model = FeedForwardNN(hidden_size=config['hidden_size']) # Create model␣
↪object
loaded_model.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
```

```
Loaded epoch 9 model
```

7 A more complex scheduling

The scheduler can be very complicated and you can write your own heuristic for it.

7.0.1 TODO#9

Implement a custom learning rate scheduler that behaves like the following graph.

You might want to learn how to use PyTorch's built-in learning rate schedulers in order to build your own.

Learning rate should be function of epoch.

https://raw.githubusercontent.com/pjumruspun/ComProg2021-Workshop/main/graph.png

[41]:
```python
# Implement scheduler here
class MyScheduler():
    def __init__(self, optimizer: torch.optim.Optimizer):
        self.optimizer = optimizer

    def step(self, epoch):
        # Changes the learning rate here
        if epoch <= 3:
          lr = 1e-4 + epoch * 3e-4
        elif epoch <= 6:
          lr = 1e-3 - (epoch - 3) * (5e-4 / 3)
        elif epoch == 7:
          lr = 1e-3
        else:
          lr = 1e-3 - (epoch - 7) * (9e-4 / 2)

        for param_groups in self.optimizer.param_groups:
          param_groups['lr'] = lr
```

```
        return lr
```

[42]:
```
# Now train with your scheduler

config_my_scheduler = {
    'architecture': 'feedforward_w_my_scheduler',
    'lr': 0.0001,
    'hidden_size': 200,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10
}

model_ff_my_scheduler =␣
↪FeedForwardNN(hidden_size=config_my_scheduler['hidden_size'])
model_ff_my_scheduler = model_ff_my_scheduler.to(device)
optimizer_my_scheduler = torch.optim.Adam(model_ff_my_scheduler.parameters(),␣
↪lr=config_my_scheduler['lr'])
my_scheduler = MyScheduler(optimizer_my_scheduler)
```

[43]:
```
train_losses = []
val_losses = []
learning_rates = []

# Start wandb run
wandb.init(
    project='precipitation-nowcasting',
    config=config_my_scheduler,
)

# Log parameters and gradients
wandb.watch(model_ff_my_scheduler, log='all', log_freq=1)

for epoch in range(config['epochs']):  # loop over the dataset multiple times

    # Training
    train_loss = []
    current_lr = optimizer_my_scheduler.param_groups[0]['lr']
    learning_rates.append(current_lr)

    # Flag model as training. Some layers behave differently in training and
    # inference modes, such as dropout, BN, etc.
    model_ff_my_scheduler.train()

    print(f"Training epoch {epoch+1}...")
```

```
    print(f"Current LR: {current_lr}")

    for i, (inputs, y_true) in enumerate(tqdm(train_loader)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        # Reset the gradient
        optimizer_my_scheduler.zero_grad()

        # Predict
        y_pred = model_ff_my_scheduler(inputs)

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Compute gradient
        loss.backward()

        # Update parameters
        optimizer_my_scheduler.step()

        # Log stuff
        train_loss.append(loss)

    avg_train_loss = torch.stack(train_loss).mean().item()
    train_losses.append(avg_train_loss)

    print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

    # Validation
    model_ff_my_scheduler.eval()
    with torch.no_grad(): # No gradient is required during validation
        print(f"Validating epoch {epoch+1}")
        val_loss = []
        for i, (inputs, y_true) in enumerate(tqdm(val_loader)):
            # Transfer data from cpu to gpu
            inputs = inputs.to(device)
            y_true = y_true.to(device)

            # Predict
            y_pred = model_ff_my_scheduler(inputs)

            # Calculate loss
            loss = loss_fn(y_pred, y_true)

            # Log stuff
```

```
            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        # LR adjustment with scheduler
        my_scheduler.step(epoch+1)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': model_ff_my_scheduler.state_dict(),
                'optimizer': optimizer_my_scheduler.state_dict(),
                # 'scheduler': my_scheduler.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss,
                'best_val_loss': best_val_loss,
            }

            print(f"Saving new best model..")
            torch.save(state, 'model_ff_my_scheduler.pth.tar')

    wandb.log({
        'train_loss': avg_train_loss,
        'val_loss': avg_val_loss,
        'lr': current_lr,
    })

wandb.finish()
print('Finished Training')
```

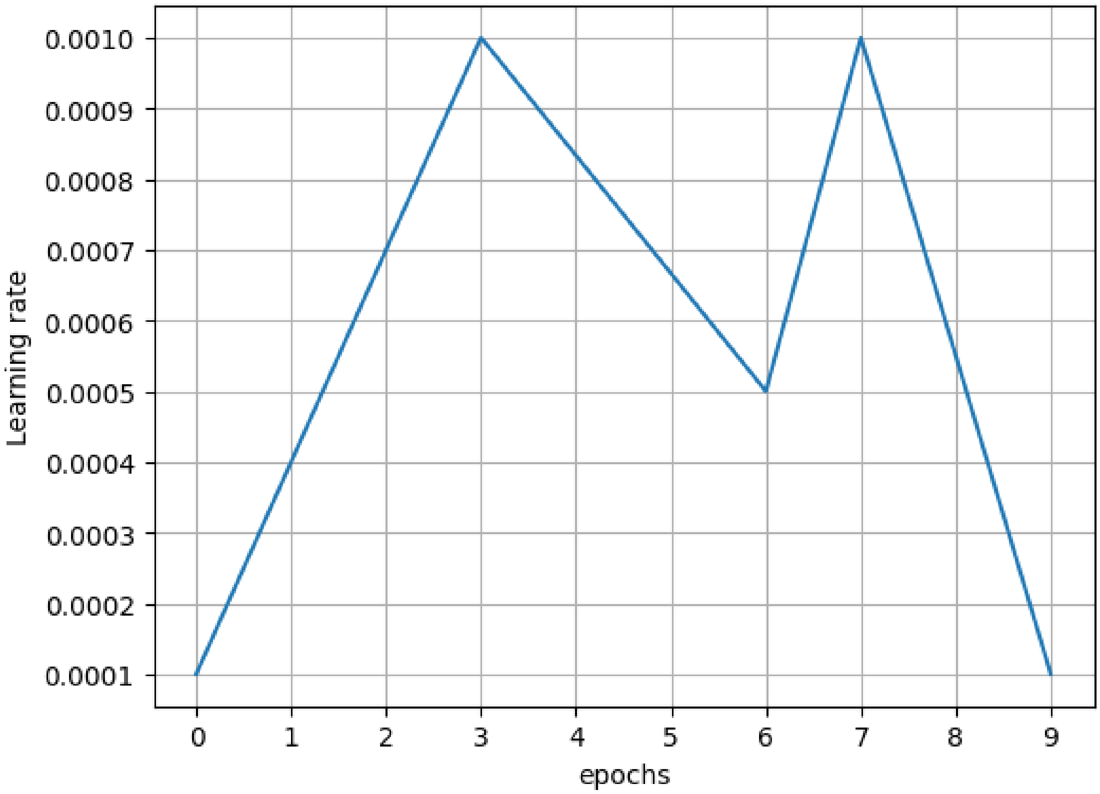
```
<IPython.core.display.HTML object>

VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
↪FloatProgress(value=1.0, max=1.0)))

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Training epoch 1...

Current LR: 0.0001

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 1 train loss: 1.9192

Validating epoch 1

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 1 val loss: 1.6557

Saving new best model..

Training epoch 2...

Current LR: 0.00039999999999999996

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 2 train loss: 1.9187

Validating epoch 2

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 2 val loss: 1.6565

Training epoch 3...

Current LR: 0.0007

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 3 train loss: 1.9185

Validating epoch 3

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 3 val loss: 1.6579

Training epoch 4...

Current LR: 0.001

  0%|          | 0/1121 [00:00<?, ?it/s]
```

```
Epoch 4 train loss: 1.9186

Validating epoch 4

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 4 val loss: 1.6564

Training epoch 5...

Current LR: 0.0008333333333333334

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 5 train loss: 1.9182

Validating epoch 5

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 5 val loss: 1.6568

Training epoch 6...

Current LR: 0.0006666666666666668

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 6 train loss: 1.9181

Validating epoch 6

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 6 val loss: 1.6566

Training epoch 7...

Current LR: 0.0005

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 7 train loss: 1.9176

Validating epoch 7

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 7 val loss: 1.6561

Training epoch 8...

Current LR: 0.001

  0%|          | 0/1121 [00:00<?, ?it/s]
```

```
Epoch 8 train loss: 1.9182

Validating epoch 8

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 8 val loss: 1.6559

Training epoch 9...

Current LR: 0.00055

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 9 train loss: 1.9179

Validating epoch 9

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 9 val loss: 1.6559

Training epoch 10...

Current LR: 0.00010000000000000005

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 10 train loss: 1.9171

Validating epoch 10

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 10 val loss: 1.6561

VBox(children=(Label(value='0.001 MB of 0.028 MB uploaded\r'),␣
↪FloatProgress(value=0.04377337581382463, max=1....

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Finished Training
```

[44]:
```
plt.plot(range(10), learning_rates)
plt.ylabel("Learning rate")
plt.xlabel("epochs")

plt.xticks(range(10))
plt.yticks(np.arange(1e-4, 1e-3+1e-4, 1e-4))
plt.grid()
plt.show()
```



8 [Optional] Wandb

You should now have a project in wandb with the name `precipitation-nowcasting`, which you should see the latest run you just finished inside the project. If you look into the run, you should be able to see plots of learning rate, train loss, val loss in the `Charts` section. Below it should be `Gradients` and `Parameters` section.

9 Wandb Observation

9.0.1 Optional TODO#1

Write your own interpretation of the logs from this example. A simple sentence or two for each section is sufficient.

Your answer: `losses` and `learning rate` is like our own plot with matplotlib. The `Gradient` is smaller, `Parameters` has a high variance from graph it look like a Mixtured distribution

10 Evaluation

[12]:
```
################################################################################
# TODO#10:                                                                     #
# Write a function to evaluate your model. Your function must predicts         #
# using the input model and return mean square error of the model.             #
#                                                                              #
# Hint: Read how to use PyTorch's MSE Loss                                     #
# https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html              #
################################################################################
#                            WRITE YOUR CODE BELOW                             #
################################################################################
def evaluate(data_loader, model, pipeline=None):
    """
    Evaluate model on validation data given by data_loader
    """
    # write code here

    loss = 0
    model.eval()
    with torch.no_grad():
      for i, (inputs, y_true) in enumerate(tqdm(data_loader)):
          # Transfer data from cpu to gpu
          inputs = inputs.to(device)
          if pipeline != None:
             inputs = pipeline(inputs)
          y_true = y_true.to(device)

          y_pred = model(inputs)
          loss += loss_fn(y_pred, y_true)

    mse = loss / len(data_loader)
    return mse
```

[56]:
```
# We will use majority rule as a baseline.
def majority_baseline(label_set):
    unique, counts = np.unique(label_set, return_counts=True)
    majority = unique[np.argmax(counts)]
    baseline = 0
    label_set = label_set.reshape(-1,1)
    for r in label_set:
        baseline += (majority - r) ** 2 / len(label_set)
    return baseline
```

[53]:
```
print('baseline')
print('train', majority_baseline(y_train))
print('validate', majority_baseline(y_val))
```

```
baseline

train [1.94397725]

validate [1.6746546]
```

[61]:
```python
print('FF-model')
print('train', evaluate(train_loader, model_ff).item())
print('validate', evaluate(val_loader, model_ff).item())
```

```
FF-model

  0%|          | 0/1121 [00:00<?, ?it/s]

train 1.9190120697021484

  0%|          | 0/454 [00:00<?, ?it/s]

validate 1.6571478843688965
```

11 Dropout

You might notice that the 3-layered feedforward does not use dropout at all. Now, try adding dropout (dropout rate of 20%) to the model, run, and report the result again.

To access PyTorch's dropout, use `nn.Dropout`. Read more about PyTorch's built-in Dropout layer here

[11]:
```python
#################################################################################
# TODO#11:                                                                      #
# Write a feedforward model with dropout                                        #
#################################################################################
#                               WRITE YOUR CODE BELOW                           #
#################################################################################
class FeedForwardNNDropout(nn.Module):
    def __init__(self, hidden_size=200, drop_rate = 0.2):
        super().__init__()
        self.ff1 = nn.Linear(75, hidden_size)
        self.dropout1 = nn.Dropout(p=drop_rate)
        self.ff2 = nn.Linear(hidden_size, hidden_size)
        self.dropout2 = nn.Dropout(p=drop_rate)
        self.ff3 = nn.Linear(hidden_size, hidden_size)
        self.dropout3 = nn.Dropout(p=drop_rate)
        self.out = nn.Linear(hidden_size, 1)

    def forward(self, x):
        hd1 = F.relu(self.ff1(x))
        hd1_dropped = self.dropout1(hd1)

        hd2 = F.relu(self.ff2(hd1_dropped))
```

```
        hd2_dropped = self.dropout2(hd2)

        y = F.relu(self.ff3(hd2_dropped))
        y = self.dropout3(y)
        y = self.out(y)
        return y.reshape(-1, 1)
```

[13]:
```
################################################################################
# TODO#12:                                                                     #
# Complete the code to train your dropout model                                #
################################################################################
print('start training ff dropout')
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################

config_dropout = {
    'architecture': 'feedforward_w_dropout',
    'lr': 0.01,
    'hidden_size': 200,
    'dropout_rate': 0.2,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10
}
model_dropout = FeedForwardNNDropout(hidden_size =␣
 ↪config_dropout['hidden_size'], drop_rate = config_dropout['dropout_rate']).
 ↪to(device)
optimizer_dropout = torch.optim.Adam(model_dropout.parameters(), lr =␣
 ↪config_dropout['lr'])
scheduler_dropout = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_dropout,
    'min',
    factor=config_dropout['scheduler_factor'],
    patience=config_dropout['scheduler_patience'],
    min_lr=config_dropout['scheduler_min_lr']
)
loss_fn = nn.MSELoss()
```

```
start training ff dropout
```

[14]:
```
train_losses = []
val_losses = []
learning_rates = []

# Start wandb run
```

```python
wandb.init(
    project='precipitation-nowcasting',
    config=config_dropout,
)

# Log parameters and gradients
wandb.watch(model_dropout, log='all', log_freq=1)

for epoch in range(config_dropout['epochs']):
    train_loss = []
    current_lr = optimizer_dropout.param_groups[0]['lr']
    learning_rates.append(current_lr)

    model_dropout.train()

    print(f"Training epoch {epoch+1}...")
    print(f"Current LR: {current_lr}")

    for i, (inputs, y_true) in enumerate(tqdm(train_loader)):
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        optimizer_dropout.zero_grad()

        y_pred = model_dropout(inputs)

        # Calculate loss
        loss = loss_fn(y_pred, y_true)
        loss.backward()
        optimizer_dropout.step()

        # log
        train_loss.append(loss)

    avg_train_loss = torch.stack(train_loss).mean().item()
    train_losses.append(avg_train_loss)

    print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

    model_dropout.eval()
    with torch.no_grad():
        print(f"Validating epoch {epoch+1}")
        val_loss = []
        for i, (inputs, y_true) in enumerate(tqdm(val_loader)):
            inputs = inputs.to(device)
            y_true = y_true.to(device)
```

```
            y_pred = model_dropout(inputs)

            loss = loss_fn(y_pred, y_true)

            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        scheduler_dropout.step(avg_val_loss)
        best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': model_dropout.state_dict(),
                'optimizer': optimizer_dropout.state_dict(),
                'scheduler': scheduler_dropout.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss,
                'best_val_loss': best_val_loss,
            }

            print(f"Saving new best model..")
            torch.save(state, 'model_dropout.pth.tar')

    wandb.log({
        'train_loss': avg_train_loss,
        'val_loss': avg_val_loss,
        'lr': current_lr,
    })

wandb.unwatch()
wandb.finish()
print('Finished Training')
```

```
Failed to detect the name of this notebook, you can set it manually with the
WANDB_NOTEBOOK_NAME environment variable to enable code saving.
wandb: Currently logged in as: pna-wan. Use `wandb

login --relogin` to force relogin

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

Training epoch 1...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 1 train loss: 1.9277
Validating epoch 1

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 1 val loss: 1.6608
Saving new best model..
Training epoch 2...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 2 train loss: 1.9237
Validating epoch 2

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 2 val loss: 1.6626
Training epoch 3...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 3 train loss: 1.9242
Validating epoch 3

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 3 val loss: 1.6616
Training epoch 4...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 4 train loss: 1.9241
Validating epoch 4

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 4 val loss: 1.6603
Saving new best model..
Training epoch 5...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 5 train loss: 1.9237
Validating epoch 5

  0%|          | 0/454 [00:00<?, ?it/s]
```

```
Epoch 5 val loss: 1.6616
Training epoch 6...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 6 train loss: 1.9234
Validating epoch 6

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 6 val loss: 1.6615
Training epoch 7...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 7 train loss: 1.9236
Validating epoch 7

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 7 val loss: 1.6616
Training epoch 8...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 8 train loss: 1.9233
Validating epoch 8

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 8 val loss: 1.6615
Training epoch 9...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 9 train loss: 1.9234
Validating epoch 9

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 9 val loss: 1.6620
Training epoch 10...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 10 train loss: 1.9236
Validating epoch 10

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 10 val loss: 1.6613
```

```
wandb: WARNING Source type is set to 'repo' but some required information is
missing from the environment. A job will not be created from this run. See
https://docs.wandb.ai/guides/launch/create-job
```

```
VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
↪FloatProgress(value=1.0, max=1.0)))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

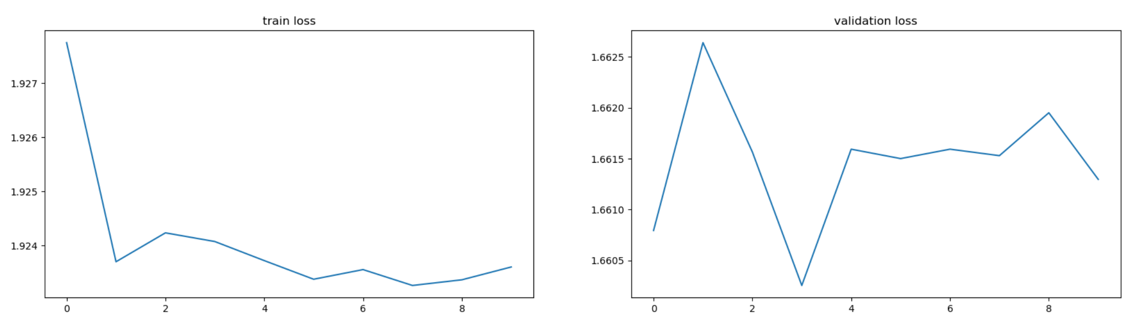
```
<IPython.core.display.HTML object>
```

```
Finished Training
```

11.0.1 TODO#13

Plot the losses and MSE of the training and validation as before. Evaluate the dropout model's performance

[15]:
```python
# Plot here
fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.tight_layout
ax[0].plot(train_losses)
ax[1].plot(val_losses)

ax[0].set_title('train loss')
ax[1].set_title('validation loss')

plt.show()
```



[18]:
```python
# Evaluate
print(evaluate(train_loader, model_dropout).item())
print(evaluate(val_loader, model_dropout).item())
```

```
  0%|          | 0/1121 [00:00<?, ?it/s]
```

```
1.923722505569458
```

```
  0%|          | 0/454 [00:00<?, ?it/s]
```

```
1.6612948179244995
```

12 Convolution Neural Networks

Now let's try to incorporate the grid sturcture to your model. Instead of passing in vectors, we are going to pass in the 5x5 grid into the model (5lat x 5long x 3channel). You are going to implement you own 2d-convolution neural networks with the following structure.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Conv2DNN                                  --                        --
Conv2d: 1-1                             [1024, 200, 3, 3]         5,600
Linear: 1-2                             [1024, 200]               360,200
Linear: 1-3                             [1024, 200]               40,200
Linear: 1-4                             [1024, 1]                 201
==========================================================================================
Total params: 406,201
Trainable params: 406,201
Non-trainable params: 0
```

These parameters are simple guidelines to save your time.
You can play with them in the final section which you can choose any normalization methods, activation function, as well as any hyperparameter the way you want.

Hint: You should read PyTorch documentation to see the list of available layers and options you can use.

[13]:
```
################################################################################
# TODO#14:                                                                     #
# Complete the code for preparing data for training CNN                        #
# Input for CNN should not have time step.                                     #
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################
def to_grid_channel(x:torch.Tensor):
    return x.view(-1, 3, 5, 5)
```

[14]:
```
################################################################################
# TODO#15:                                                                     #
# Write a PyTorch convolutional neural network model.                          #
# You might want to use the layer torch.flatten somewhere                      #
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################
class CNN(nn.Module):
    def __init__(self, hidden_size=200):
        super().__init__()
        self.cv = nn.Conv2d(3, hidden_size, 3)
```

```python
        self.ff1 = nn.Linear(hidden_size*3*3, hidden_size)
        self.ff2 = nn.Linear(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, 1)

    def forward(self, x:torch.Tensor):
        cv = F.relu(self.cv(x))
        cv_flatten = cv.flatten(1)
        fc1 = F.relu(self.ff1(cv_flatten))
        fc2 = F.relu(self.ff2(fc1))
        y = self.out(fc2)
        return y
```

[15]:
```python
config_cnn = {
    'architecture': 'cnn',
    'lr': 0.01,
    'hidden_size': 200,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10,
}

model_cnn = CNN(hidden_size=config_cnn['hidden_size']).to(device)
optimizer_cnn = torch.optim.Adam(model_cnn.parameters(), lr=config_cnn['lr'])
scheduler_cnn = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_cnn,
    mode='min',
    factor=config_cnn['scheduler_factor'],
    patience=config_cnn['scheduler_patience'],
    min_lr = config_cnn['scheduler_min_lr']
)

summary(model_cnn, input_size=(1024, 3, 5, 5))
```

[15]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
CNN                                       [1024, 1]                 --
Conv2d: 1-1                              [1024, 200, 3, 3]         5,600
Linear: 1-2                              [1024, 200]               360,200
Linear: 1-3                              [1024, 200]               40,200
Linear: 1-4                              [1024, 1]                 201
================================================================================
==========
Total params: 406,201
```

```
Trainable params: 406,201
Non-trainable params: 0
Total mult-adds (M): 461.83
================================================================================
==========
Input size (MB): 0.31
Forward/backward pass size (MB): 18.03
Params size (MB): 1.62
Estimated Total Size (MB): 19.96
================================================================================
==========
```

[14]:

```
################################################################################
# TODO#16:                                                                     #
# Complete the code to train your cnn model                                    #
################################################################################
print('start training conv2d')
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################

train_losses = []
val_losses = []
learning_rates = []

wandb.init(
    project='precipitation-nowcasting',
    config=config_cnn,
)
wandb.watch(model_cnn, log='all', log_freq=1)

for epoch in range(config_cnn['epochs']):
    # Training
    train_loss = []
    current_lr = optimizer_cnn.param_groups[0]['lr']
    learning_rates.append(current_lr)

    # Flag model as training. Some layers behave differently in training and
    # inference modes, such as dropout, BN, etc.
    model_cnn.train()

    print(f"Training epoch {epoch+1}...")
    print(f"Current LR: {current_lr}")

    for i, (inputs, y_true) in enumerate(tqdm(train_loader)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
```

```
        y_true = y_true.to(device)

        # Reset the gradient
        optimizer_cnn.zero_grad()

        # Predict
        y_pred = model_cnn(to_grid_channel(inputs))

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Compute gradient
        loss.backward()

        # Update parameters
        optimizer_cnn.step()

        # Log stuff
        train_loss.append(loss)

    avg_train_loss = torch.stack(train_loss).mean().item()
    train_losses.append(avg_train_loss)

    print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

    model_cnn.eval()
    with torch.no_grad(): # No gradient is required during validation
        print(f"Validating epoch {epoch+1}")
        val_loss = []
        for i, (inputs, y_true) in enumerate(tqdm(val_loader)):
            # Transfer data from cpu to gpu
            inputs = inputs.to(device)
            y_true = y_true.to(device)

            # Predict
            y_pred = model_cnn(to_grid_channel(inputs))

            # Calculate loss
            loss = loss_fn(y_pred, y_true)

            # Log stuff
            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")
```

```
        # LR adjustment with my_scheduler
        scheduler_cnn.step(avg_val_loss)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': model_cnn.state_dict(),
                'optimizer': optimizer_cnn.state_dict(),
                'scheduler': scheduler_cnn.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss,
                'best_val_loss': best_val_loss,
            }

            print(f"Saving new best model..")
            torch.save(state, 'model_cnn.pth.tar')

    wandb.log({
        'train_loss': avg_train_loss,
        'val_loss': avg_val_loss,
        'lr': current_lr,
    })



wandb.unwatch()
wandb.finish()
```

```
start training conv2d

Failed to detect the name of this notebook, you can set it manually with the
WANDB_NOTEBOOK_NAME environment variable to enable code saving.
wandb: Currently logged in as: pna-wan. Use `wandb

login --relogin` to force relogin

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Training epoch 1...
Current LR: 0.01
```

```
  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 1 train loss: 2.1787
Validating epoch 1

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 1 val loss: 1.6570
Saving new best model..
Training epoch 2...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 2 train loss: 1.9218
Validating epoch 2

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 2 val loss: 1.6639
Training epoch 3...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 3 train loss: 1.9200
Validating epoch 3

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 3 val loss: 1.6590
Training epoch 4...
Current LR: 0.01

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 4 train loss: 1.9198
Validating epoch 4

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 4 val loss: 1.6597
Training epoch 5...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 5 train loss: 1.9188
Validating epoch 5

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 5 val loss: 1.6567
Saving new best model..
Training epoch 6...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]
```

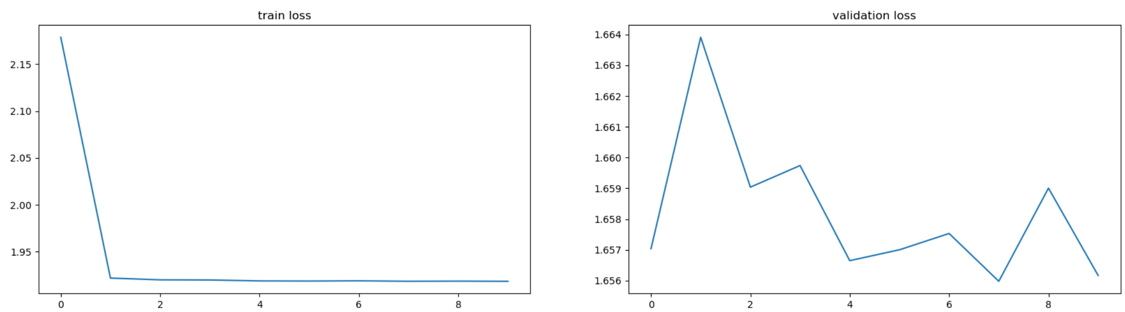
```
Epoch 6 train loss: 1.9187
Validating epoch 6

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 6 val loss: 1.6570
Training epoch 7...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 7 train loss: 1.9189
Validating epoch 7

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 7 val loss: 1.6575
Training epoch 8...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 8 train loss: 1.9184
Validating epoch 8

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 8 val loss: 1.6560
Saving new best model..
Training epoch 9...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 9 train loss: 1.9186
Validating epoch 9

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 9 val loss: 1.6590
Training epoch 10...
Current LR: 0.002

  0%|          | 0/1121 [00:00<?, ?it/s]

Epoch 10 train loss: 1.9184
Validating epoch 10

  0%|          | 0/454 [00:00<?, ?it/s]

Epoch 10 val loss: 1.6562

wandb: WARNING Source type is set to 'repo' but some required information is
missing from the environment. A job will not be created from this run. See
https://docs.wandb.ai/guides/launch/create-job

VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),
 ↪FloatProgress(value=1.0, max=1.0)))
```

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

[15]:
```
# Plot losses
fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.tight_layout
ax[0].plot(train_losses)
ax[1].plot(val_losses)

ax[0].set_title('train loss')
ax[1].set_title('validation loss')

plt.show()
```



[31]:
```
# Evaluate
print(evaluate(train_loader, model_cnn, to_grid_channel).item())
print(evaluate(val_loader, model_cnn, to_grid_channel).item())
print(evaluate(test_loader, model_cnn, to_grid_channel).item())
```

```
  0%|          | 0/1121 [00:00<?, ?it/s]

1.9181199073791504

  0%|          | 0/454 [00:00<?, ?it/s]

1.656167984008789

  0%|          | 0/546 [00:00<?, ?it/s]

1.1611506938934326
```

13 Gated Recurrent Units

Now, you want to add time steps into your model. Recall the original data has 5 time steps per item. You are going to pass in a data of the form 5 timesteps x 75data. This can be done using a GRU layer. Implement you own GRU network with the following structure.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
GRUModel                                  --                        --
GRU: 1-1                                 [1024, 5, 200]            166,200
Linear: 1-2                              [1024, 5, 200]            40,200
Linear: 1-3                              [1024, 5, 1]              201
==========================================================================================
Total params: 206,601
Trainable params: 206,601
Non-trainable params: 0
```

These parameters are simple guidelines to save your time.
You can play with them in the final section which you can choose any normalization methods, activation function, as well as any hyperparameter the way you want.
The result should be better than the feedforward model and at least on par with your CNN model.

Do consult PyTorch documentation on how to use GRUs.

[16]:
```python
################################################################################
# TODO#17:                                                                     #
# Complete the code for preparing data for training GRU                        #
# GRU's input should has 3 dimensions.                                         #
# The dimensions should compose of entries, time-step, and features.           #
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################
def to_time_step(x:torch.Tensor):
    return x.view(-1, 5, 5*5*3)

normalizer_gru = normalizer_std(x_train.reshape(-1, 5*5*5*3))
train_loader_gru = DataLoader(RainfallDatasetFF(x_train.reshape(-1, 5*5*5*3),␣
↪y_train.reshape(-1, 5), normalizer_gru), batch_size=1024, shuffle=True,␣
↪pin_memory=True)
val_loader_gru = DataLoader(RainfallDatasetFF(x_val.reshape(-1, 5*5*5*3), y_val.
↪reshape(-1, 5), normalizer_gru), batch_size=1024, shuffle=True,␣
↪pin_memory=True)
test_loader_gru = DataLoader(RainfallDatasetFF(x_test.reshape(-1, 5*5*5*3),␣
↪y_test.reshape(-1, 5), normalizer_gru), batch_size=1024, shuffle=True,␣
↪pin_memory=True)
```

```
(229548, 375)
(229548, 5)
(92839, 375)
(92839, 5)
(111715, 375)
(111715, 5)
```

[17]:

```
################################################################################
# TODO#18                                                                      #
# Write a PyTorch GRU model.                                                   #
# Your goal is to predict a precipitation of every time step.                  #
#                                                                              #
# Hint: You should read PyTorch documentation to see the list of available     #
# layers and options you can use.                                              #
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################
class GRU(nn.Module):
    def __init__(self, hidden_size = 200, pipe=None):
        super().__init__()
        self.pipe = pipe
        self.gru = nn.GRU(5*5*3, hidden_size, batch_first=True)
        self.ff1 = nn.Linear(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, 1)

    def forward(self, x):
        if self.pipe:
            x = self.pipe(x)
        gru, _ = self.gru(x)
        fc1 = self.ff1(gru)
        y = self.out(fc1)
        return y.reshape(-1, 5)
```

[18]:

```
config_gru = {
    'architecture': 'gru',
    'lr': 0.01,
    'hidden_size': 200,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10,
}

model_gru = GRU(hidden_size=config_gru['hidden_size'], pipe=to_time_step).
 ↪to(device)
optimizer_gru = torch.optim.Adam(model_gru.parameters(), lr=config_gru['lr'])
scheduler_gru = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_gru,
    mode='min',
    factor=config_gru['scheduler_factor'],
    patience=config_gru['scheduler_patience'],
    min_lr = config_gru['scheduler_min_lr']
)
```

```
summary(model_gru, input_size=(1024, 5*5*5*3))
```

[18]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
GRU                                       [1024, 5]                 --
GRU: 1-1                                 [1024, 5, 200]            166,200
Linear: 1-2                              [1024, 5, 200]            40,200
Linear: 1-3                              [1024, 5, 1]              201
================================================================================
==========
Total params: 206,601
Trainable params: 206,601
Non-trainable params: 0
Total mult-adds (M): 892.31
================================================================================
==========
Input size (MB): 1.54
Forward/backward pass size (MB): 16.42
Params size (MB): 0.83
Estimated Total Size (MB): 18.79
================================================================================
==========
```

[40]:
```
################################################################################
# TODO#19                                                                      #
# Complete the code to train your gru model                                    #
################################################################################
print('start training gru')
################################################################################
#                             WRITE YOUR CODE BELOW                            #
################################################################################

train_losses = []
val_losses = []
learning_rates = []

wandb.init(
    project='precipitation-nowcasting',
    config=config_gru,
)
wandb.watch(model_gru, log='all', log_freq=1)

for epoch in range(config_gru['epochs']):
    # Training
```

```
train_loss = []
current_lr = optimizer_gru.param_groups[0]['lr']
learning_rates.append(current_lr)

# Flag model as training. Some layers behave differently in training and
# inference modes, such as dropout, BN, etc.
model_gru.train()

print(f"Training epoch {epoch+1}...")
print(f"Current LR: {current_lr}")

for i, (inputs, y_true) in enumerate(tqdm(train_loader_gru)):
    # Transfer data from cpu to gpu
    inputs = inputs.to(device)
    y_true = y_true.to(device)

    # Reset the gradient
    optimizer_gru.zero_grad()

    # Predict
    y_pred = model_gru(inputs)

    # Calculate loss
    loss = loss_fn(y_pred, y_true)

    # Compute gradient
    loss.backward()

    # Update parameters
    optimizer_gru.step()

    # Log stuff
    train_loss.append(loss)

avg_train_loss = torch.stack(train_loss).mean().item()
train_losses.append(avg_train_loss)

print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

model_gru.eval()
with torch.no_grad(): # No gradient is required during validation
    print(f"Validating epoch {epoch+1}")
    val_loss = []
    for i, (inputs, y_true) in enumerate(tqdm(val_loader_gru)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)
```

```
            # Predict
            y_pred = model_gru(inputs)

            # Calculate loss
            loss = loss_fn(y_pred, y_true)

            # Log stuff
            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        # LR adjustment with my_scheduler
        scheduler_gru.step(avg_val_loss)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': model_gru.state_dict(),
                'optimizer': optimizer_gru.state_dict(),
                'scheduler': scheduler_gru.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss,
                'best_val_loss': best_val_loss,
            }

            print(f"Saving new best model..")
            torch.save(state, 'model_gru.pth.tar')

    wandb.log({
        'train_loss': avg_train_loss,
        'val_loss': avg_val_loss,
        'lr': current_lr,
    })


wandb.unwatch()
wandb.finish()
```

```
start training gru
```

```
<IPython.core.display.HTML object>
```

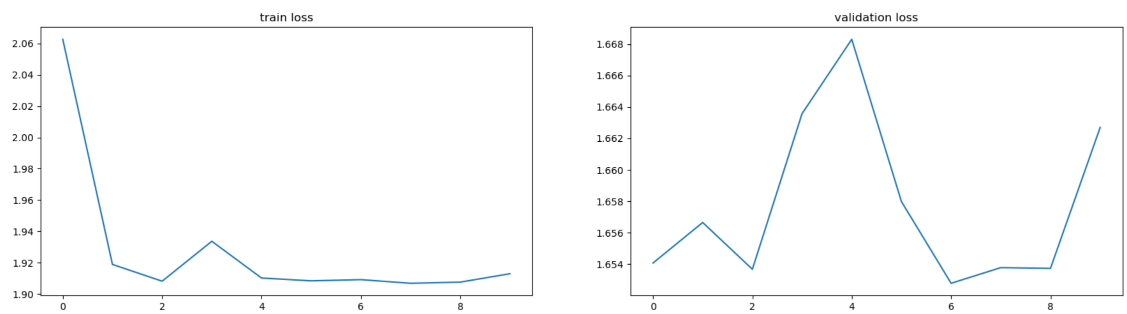
```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Training epoch 1...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 1 train loss: 2.0627
Validating epoch 1

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 1 val loss: 1.6541
Saving new best model..
Training epoch 2...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 2 train loss: 1.9189
Validating epoch 2

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 2 val loss: 1.6566
Training epoch 3...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 3 train loss: 1.9082
Validating epoch 3

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 3 val loss: 1.6537
Saving new best model..
Training epoch 4...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 4 train loss: 1.9337
Validating epoch 4

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 4 val loss: 1.6636
Training epoch 5...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]
```

```
Epoch 5 train loss: 1.9102
Validating epoch 5

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 5 val loss: 1.6683
Training epoch 6...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 6 train loss: 1.9084
Validating epoch 6

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 6 val loss: 1.6580
Training epoch 7...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 7 train loss: 1.9092
Validating epoch 7

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 7 val loss: 1.6528
Saving new best model..
Training epoch 8...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 8 train loss: 1.9068
Validating epoch 8

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 8 val loss: 1.6538
Training epoch 9...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 9 train loss: 1.9076
Validating epoch 9

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 9 val loss: 1.6537
Training epoch 10...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 10 train loss: 1.9129
Validating epoch 10
```

```
  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 10 val loss: 1.6627

wandb: WARNING Source type is set to 'repo' but some required information is
missing from the environment. A job will not be created from this run. See
https://docs.wandb.ai/guides/launch/create-job

VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
↪FloatProgress(value=1.0, max=1.0)))

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[41]:
```
# Plot
fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.tight_layout
ax[0].plot(train_losses)
ax[1].plot(val_losses)

ax[0].set_title('train loss')
ax[1].set_title('validation loss')

plt.show()
```



[42]:
```
# Evaluate
print(evaluate(train_loader_gru, model_gru).item())
print(evaluate(val_loader_gru, model_gru).item())
print(evaluate(test_loader_gru, model_gru).item())
```

```
  0%|          | 0/225 [00:00<?, ?it/s]

1.9080415964126587

  0%|          | 0/91 [00:00<?, ?it/s]

1.661690354347229
```

```
0%|          | 0/110 [00:00<?, ?it/s]
```

```
1.1552120447158813
```

14 Transformer

Welcome to the beginning of the real world! The aboved models are not usually used in practice due to its limited capability. Transformers are generally used by computer vision, natural language processing, and speech processing (almost every big AI fields).

In our dataloader, we will add the output of this timestep (the number of precipitation) as an auxiliary input to predict the next timestep. Thus, input for the model should be [#batch_size, 5, 76] (5 timesteps and the number 76 comes from (3x5x5)+1) and the output for the model should be [#batch_size, 1] which would be the next timestep we want to predict. Additionally, we will mask the input at the dataloader to the attenttion from observing future values. Suppose that we want to predict timestep 3, we will mask the timestep 3, 4 and 5 in our input by setting it to zeros, and we will predict the timestep 3.

In order to get a score on this TODO, students need to implement a dataloader that mask the input correctly.

[52]:
```
################################################################################
# OT#2:                                                                        #
# Complete the code for preparing data for training Transformer                #
# Transformer's input should has 3 dimensions.                                 #
# The dimensions should compose of entries, time-step, and features.           #
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################
class TransfomerDataset(Dataset):
    def __init__(self, x, y):
        x = x.astype(np.float32) # 5 * (3 * 5 * 5)
        y = y.astype(np.float32)
        y = y[:, :, None]
        x = np.concatenate((x, y), axis=2)
        # x = np.hstack( (x.reshape(-1, 5*3*5*5), y.reshape(-1, 5)) )
        self.x = x.reshape(-1, 5, 76)
        self.y = y

    # def __len__(self):
    #     return self.x.shape[0] * (self.x.shape[1]-1) # first time step?

    # def __getitem__(self, index) :
    #     year = index // 4
    #     time_step = index % 4 + 1
    #     x = self.x[year].copy() # Retrieve data
    #     x[time_step:, :] = 0.0
    #     y = self.y[year, time_step]
```

```
    #     return x, y

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, index):
        x = self.x[index].copy()
        time_step = np.random.randint(1, 5)
        x[time_step:, :] = 0.0
        y = self.y[index, time_step]
        return x, y


normalizer_tf = normalizer_std( (np.hstack((x_train.reshape(-1, 5*3*5*5),␣
↪y_train.reshape(-1, 5))) ) )


train_loader_tf = DataLoader(TransfomerDataset(x_train.reshape(-1, 5, 5*5*3),␣
↪y_train.reshape(-1, 5)), batch_size=1024, shuffle=True, pin_memory=True)
val_loader_tf = DataLoader(TransfomerDataset(x_val.reshape(-1, 5, 5*5*3), y_val.
↪reshape(-1, 5)), batch_size=1024, shuffle=True, pin_memory=True)
test_loader_tf = DataLoader(TransfomerDataset(x_test.reshape(-1, 5, 5*5*3),␣
↪y_test.reshape(-1, 5)), batch_size=1024, shuffle=True, pin_memory=True)
```

[47]:
```
data = TransfomerDataset(x_train.reshape(-1, 5, 5*5*3), y_train.reshape(-1, 5))
data[0]
print()
```

In this task, we will implement one encoder layer of Transformer and add the linear layer to make a regression prediction. For the simplicity of the model, we will change the multi-head attention to QKV self-attention (single-head). As a result, our model should look like the diagram below. Since the layer self-attention is not availble in torch, students have to implement it themselves. In Add & Norm layer, students have to do the addition before normalizing. In Layer Normalization, we will normalize across both timesteps and features.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
TransformerModel                          [1024, 1]                 --
PositionalEncoding: 1-1                 [1024, 5, 76]             --
    Dropout: 2-1                       [1024, 5, 76]             --
SelfAttention: 1-2                      [1024, 5, 76]             --
    Linear: 2-2                        [1024, 5, 76]             5,852
    Linear: 2-3                        [1024, 5, 76]             5,852
```

```
   Linear: 2-4                          [1024, 5, 76]            5,852
   Softmax: 2-5                         [1024, 5, 5]             --
LayerNorm: 1-3                           [1024, 5, 76]            760
Linear: 1-4                              [1024, 5, 76]            5,852
LayerNorm: 1-5                           [1024, 5, 76]            760
Linear: 1-6                              [1024, 1]                381
==========================================================================================
Total params: 25,309
Trainable params: 25,309
Non-trainable params: 0
Total mult-adds (M): 25.92
==========================================================================================
Input size (MB): 1.56
Forward/backward pass size (MB): 18.69
Params size (MB): 0.10
Estimated Total Size (MB): 20.34
==========================================================================================
```

[40]:

```
################################################################################
# OT#3                                                                         #
# Write a PyTorch PositionalEncoding model.                                    #
#                                                                              #
# Hint: You should read PyTorch documentation to see the list of available     #
# layers and options you can use.                                              #
################################################################################
#                             WRITE YOUR CODE BELOW                            #
################################################################################

class PositionalEncoding(nn.Module):
  def __init__(self, seq_len, emb_dim, dropout=0.2):
    super().__init__()
    self.dropout = nn.Dropout(p=dropout)

    i = torch.arange(seq_len).unsqueeze(1)
    powers = torch.pow(10_000, -torch.arange(0, emb_dim, 2) / emb_dim)
    PE = torch.zeros(1, seq_len, emb_dim)
    PE[0, :, 0::2] = torch.sin(i * powers)
    PE[0, :, 1::2] = torch.cos(i * powers)
    self.register_buffer('PE', PE)

  def forward(self, x):
    x_mask = (x==0)
    batch_size, time_step, dim = x.shape
    out = self.dropout(x + self.PE[:, :time_step])
    out[torch.where(x)] = 0.0
    return out
```

[141]:

```
################################################################################
# OT#4                                                                         #
# Write a PyTorch Transformer model.                                           #
# Your goal is to predict a precipitation of every time step.                  #
#                                                                              #
# Hint: You should read PyTorch documentation to see the list of available     #
# layers and options you can use.                                              #
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################

class SelfAttention(nn.Module):
  def __init__(self, input_dim):
    super().__init__()
    self.query = nn.Linear(input_dim, input_dim)
    self.key = nn.Linear(input_dim, input_dim)
    self.value = nn.Linear(input_dim, input_dim)
    self.softmax = nn.Softmax(dim=-1)

  def forward(self, x):
    Q = self.query(x)
    K = self.key(x)
    V = self.value(x)
    scale_factor = 1 / np.sqrt(Q.size(-1))
    out = Q @ K.transpose(-2, -1) * scale_factor
    out = self.softmax(out) @ V
    return out

class TransformerModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.pos_enc = PositionalEncoding(5, 76)
    self.self_attn = SelfAttention(76)
    self.layer_norm1 = nn.LayerNorm((5, 76))
    self.layer_norm2 = nn.LayerNorm((5, 76))
    self.ff = nn.Linear(76, 76)
    self.out = nn.Linear(76 * 5, 1)


  def forward(self, x):
    x = self.pos_enc(x)
    attn = self.self_attn(x)
    x = self.layer_norm1(x + attn)
    ff_out = self.ff(x)
    x = self.layer_norm2(ff_out + x)
    out = self.out(x.flatten(1))
    return out
```

[142]:
```
################################################################################
# OT#5                                                                         #
# Complete the code to train your Transformer model                            #
################################################################################
print('start training transformer')
################################################################################
#                           WRITE YOUR CODE BELOW                              #
################################################################################


config_transformer = {
    'architecture': 'transformer',
    'lr': 0.01,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10,
}

transformer = TransformerModel().to(device)
optimizer_transformer = torch.optim.Adam(transformer.parameters(),␣
 ↪lr=config_transformer['lr'])
scheduler_transformer = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_transformer,
    mode='min',
    factor=config_transformer['scheduler_factor'],
    patience=config_transformer['scheduler_patience'],
    min_lr = config_transformer['scheduler_min_lr']
)

transformer = TransformerModel()
summary(transformer, input_size=(1024, 5, 76))
```

```
start training transformer
```

[142]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
TransformerModel                          [1024, 1]                 --
PositionalEncoding: 1-1                 [1024, 5, 76]             --
    Dropout: 2-1                       [1024, 5, 76]             --
SelfAttention: 1-2                      [1024, 5, 76]             --
    Linear: 2-2                        [1024, 5, 76]             5,852
    Linear: 2-3                        [1024, 5, 76]             5,852
    Linear: 2-4                        [1024, 5, 76]             5,852
```

```
    Softmax: 2-5                          [1024, 5, 5]              --
LayerNorm: 1-3                             [1024, 5, 76]             760
Linear: 1-4                                [1024, 5, 76]             5,852
LayerNorm: 1-5                             [1024, 5, 76]             760
Linear: 1-6                                [1024, 1]                 381
================================================================================
==========
Total params: 25,309
Trainable params: 25,309
Non-trainable params: 0
Total mult-adds (M): 25.92
================================================================================
==========
Input size (MB): 1.56
Forward/backward pass size (MB): 18.69
Params size (MB): 0.10
Estimated Total Size (MB): 20.34
================================================================================
==========
```

[54]:

```
train_losses = []
val_losses = []
learning_rates = []

wandb.init(
    project='precipitation-nowcasting',
    config=config_transformer,
)
wandb.watch(transformer, log='all', log_freq=1)

for epoch in range(config_transformer['epochs']):
    # Training
    train_loss = []
    current_lr = optimizer_transformer.param_groups[0]['lr']
    learning_rates.append(current_lr)

    # Flag model as training. Some layers behave differently in training and
    # inference modes, such as dropout, BN, etc.
    transformer.train()

    print(f"Training epoch {epoch+1}...")
    print(f"Current LR: {current_lr}")

    for i, (inputs, y_true) in enumerate(tqdm(train_loader_tf)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)
```

```
        # Reset the gradient
        optimizer_transformer.zero_grad()

        # Predict
        y_pred = transformer(inputs)

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Compute gradient
        loss.backward()

        # Update parameters
        optimizer_transformer.step()

        # Log stuff
        train_loss.append(loss)

    avg_train_loss = torch.stack(train_loss).mean().item()
    train_losses.append(avg_train_loss)

    print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

    transformer.eval()
    with torch.no_grad(): # No gradient is required during validation
        print(f"Validating epoch {epoch+1}")
        val_loss = []
        for i, (inputs, y_true) in enumerate(tqdm(val_loader_tf)):
            # Transfer data from cpu to gpu
            inputs = inputs.to(device)
            y_true = y_true.to(device)

            # Predict
            y_pred = transformer(inputs)

            # Calculate loss
            loss = loss_fn(y_pred, y_true)

            # Log stuff
            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        # LR adjustment with my_scheduler
```

```
        scheduler_transformer.step(avg_val_loss)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': transformer.state_dict(),
                'optimizer': optimizer_transformer.state_dict(),
                'scheduler': scheduler_transformer.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss,
                'best_val_loss': best_val_loss,
            }

            print(f"Saving new best model..")
            torch.save(state, 'transformer.pth.tar')

    wandb.log({
        'train_loss': avg_train_loss,
        'val_loss': avg_val_loss,
        'lr': current_lr,
    })


wandb.unwatch()
wandb.finish()
```

```
<IPython.core.display.HTML object>

VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
↪FloatProgress(value=1.0, max=1.0)))

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

VBox(children=(Label(value='Waiting for wandb.init()...\r'), FloatProgress(value=0.
↪011143120833245727, max=1.0...

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

Training epoch 1...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 1 train loss: 2.1088
Validating epoch 1

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 1 val loss: 1.9829
Saving new best model..
Training epoch 2...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 2 train loss: 2.0157
Validating epoch 2

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 2 val loss: 1.6176
Saving new best model..
Training epoch 3...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 3 train loss: 2.0483
Validating epoch 3

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 3 val loss: 1.8421
Training epoch 4...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 4 train loss: 2.1264
Validating epoch 4

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 4 val loss: 1.8916
Training epoch 5...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 5 train loss: 2.1325
Validating epoch 5

  0%|          | 0/91 [00:00<?, ?it/s]
```

```
Epoch 5 val loss: 1.7252
Training epoch 6...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 6 train loss: 2.0808
Validating epoch 6

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 6 val loss: 1.7952
Training epoch 7...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 7 train loss: 2.1345
Validating epoch 7

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 7 val loss: 1.5484
Saving new best model..
Training epoch 8...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 8 train loss: 1.9650
Validating epoch 8

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 8 val loss: 1.6363
Training epoch 9...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 9 train loss: 2.0653
Validating epoch 9

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 9 val loss: 1.9458
Training epoch 10...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 10 train loss: 2.0601
Validating epoch 10

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 10 val loss: 1.9881
```

```
VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
↪FloatProgress(value=1.0, max=1.0)))
```

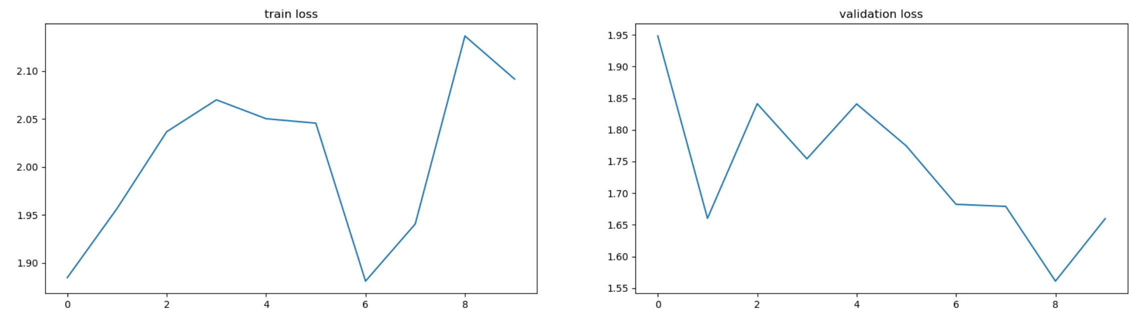
```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[73]:
```
# Plot
fig, ax = plt.subplots(1, 2, figsize=(20, 5))
fig.tight_layout
ax[0].plot(train_losses)
ax[1].plot(val_losses)

ax[0].set_title('train loss')
ax[1].set_title('validation loss')

plt.show()
```



If you implement it correctly, you should evaluate the model in the test dataset and the score should be better than the aboved models.

[61]:
```
# Evaluate
checkpoint = torch.load('transformer.pth.tar')
loaded_transformer = TransformerModel().to(device)
loaded_transformer.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")

print(evaluate(test_loader_tf, loaded_transformer).item())
```

```
Loaded epoch 6 model
```

```
  0%|          | 0/110 [00:00<?, ?it/s]
```

```
1.1332389116287231
```

15 Final Section

16 PyTorch playground

Now, train the best model you can do for this task. You can use any model structure and function available.
Remember that trainig time increases with the complexity of the model. You might find printing computation graphs helpful in debugging complicated models.
Your model should be better than your CNN or GRU model in the previous sections.

Some ideas:

- Tune the hyperparameters
- Adding dropouts
- Combining CNN with GRUs

You should tune your model on training and validation set.
The test set should be used only for the last evaluation.

[ ]:
```
# Prep data as you see fit
```

[115]:
```
################################################################################
# TODO#20                                                                      #
# Write a function that returns your best PyTorch model. You can use anything  #
# you want. The goal here is to create the best model you can think of.        #
#                                                                              #
# Hint: You should read PyTorch documentation to see the list of available     #
# layers and options you can use.                                              #
################################################################################
#                            WRITE YOUR CODE BELOW                             #
################################################################################
class TransformerV2(nn.Module):
  def __init__(self):
    super().__init__()
    self.pos_enc = PositionalEncoding(5, 76)
    self.self_attn = nn.MultiheadAttention(76, 4)
    self.layer_norm1 = nn.LayerNorm((5, 76))
    self.layer_norm2 = nn.LayerNorm((5, 76))
    self.ff = nn.Sequential(
      nn.Linear(76, 150),
      nn.SELU(150),
      nn.Linear(150, 76)
    )
    self.out = nn.Linear(76 * 5, 1)

  def forward(self, x):
    x = self.pos_enc(x)
    attn, _ = self.self_attn(x, x, x)
    x = self.layer_norm1(x + attn)
```

```
        ff_out = self.ff(x)
        x = self.layer_norm2(ff_out + x)
        out = self.out(x.flatten(1))
        return out
```

[116]:
```
config_transformer = {
    'architecture': 'transformerV2',
    'lr': 0.01,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10,
}

transformerV2 = TransformerV2().to(device)
optimizer_transformer = torch.optim.Adam(transformer.parameters(),␣
 ↪lr=config_transformer['lr'])
scheduler_transformer = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_transformer,
    mode='min',
    factor=config_transformer['scheduler_factor'],
    patience=config_transformer['scheduler_patience'],
    min_lr = config_transformer['scheduler_min_lr']
)

summary(transformerV2, input_size=(1024, 5, 76))
```

[116]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
TransformerV2                             [1024, 1]                 --
PositionalEncoding: 1-1                 [1024, 5, 76]             --
    Dropout: 2-1                        [1024, 5, 76]             --
MultiheadAttention: 1-2                 [1024, 5, 76]             23,408
LayerNorm: 1-3                          [1024, 5, 76]             760
Sequential: 1-4                         [1024, 5, 76]             --
    Linear: 2-2                         [1024, 5, 150]            11,550
    SELU: 2-3                           [1024, 5, 150]            --
    Linear: 2-4                         [1024, 5, 76]             11,476
LayerNorm: 1-5                          [1024, 5, 76]             760
Linear: 1-6                             [1024, 1]                 381
================================================================================
==========
Total params: 48,335
Trainable params: 48,335
```

```
Non-trainable params: 0
Total mult-adds (M): 25.53
================================================================================
==========
Input size (MB): 1.56
Forward/backward pass size (MB): 15.49
Params size (MB): 0.10
Estimated Total Size (MB): 17.15
================================================================================
==========
```

[117]:

```
################################################################################
# TODO#21                                                                      #
# Complete the code to train your best model                                   #
################################################################################
print('start training the best model')
################################################################################
#                            WRITE YOUR CODE BELOW                             #
################################################################################

train_losses = []
val_losses = []
learning_rates = []

wandb.init(
    project='precipitation-nowcasting',
    config=config_transformer,
)
wandb.watch(transformerV2, log='all', log_freq=1)

for epoch in range(config_transformer['epochs']):
    # Training
    train_loss = []
    current_lr = optimizer_transformer.param_groups[0]['lr']
    learning_rates.append(current_lr)

    # Flag model as training. Some layers behave differently in training and
    # inference modes, such as dropout, BN, etc.
    transformerV2.train()

    print(f"Training epoch {epoch+1}...")
    print(f"Current LR: {current_lr}")

    for i, (inputs, y_true) in enumerate(tqdm(train_loader_tf)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)
```

```
        # Reset the gradient
        optimizer_transformer.zero_grad()

        # Predict
        y_pred = transformerV2(inputs)

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Compute gradient
        loss.backward()

        # Update parameters
        optimizer_transformer.step()

        # Log stuff
        train_loss.append(loss)

    avg_train_loss = torch.stack(train_loss).mean().item()
    train_losses.append(avg_train_loss)

    print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

    transformerV2.eval()
    with torch.no_grad(): # No gradient is required during validation
        print(f"Validating epoch {epoch+1}")
        val_loss = []
        for i, (inputs, y_true) in enumerate(tqdm(val_loader_tf)):
            # Transfer data from cpu to gpu
            inputs = inputs.to(device)
            y_true = y_true.to(device)

            # Predict
            y_pred = transformerV2(inputs)

            # Calculate loss
            loss = loss_fn(y_pred, y_true)

            # Log stuff
            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        # LR adjustment with my_scheduler
```

```python
        scheduler_transformer.step(avg_val_loss)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': transformerV2.state_dict(),
                'optimizer': optimizer_transformer.state_dict(),
                'scheduler': scheduler_transformer.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss,
                'best_val_loss': best_val_loss,
            }

            print(f"Saving new best model..")
            torch.save(state, 'transformerV2.pth.tar')

    wandb.log({
        'train_loss': avg_train_loss,
        'val_loss': avg_val_loss,
        'lr': current_lr,
    })


wandb.unwatch()
wandb.finish()
```

```
start training the best model

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Training epoch 1...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 1 train loss: 1.9805
Validating epoch 1

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 1 val loss: 1.9284
Saving new best model..
```

```
Training epoch 2...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 2 train loss: 1.9016
Validating epoch 2

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 2 val loss: 1.7492
Saving new best model..
Training epoch 3...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 3 train loss: 2.0304
Validating epoch 3

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 3 val loss: 1.5051
Saving new best model..
Training epoch 4...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 4 train loss: 2.1504
Validating epoch 4

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 4 val loss: 2.0055
Training epoch 5...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 5 train loss: 1.9502
Validating epoch 5

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 5 val loss: 1.6535
Training epoch 6...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 6 train loss: 1.8435
Validating epoch 6

  0%|          | 0/91 [00:00<?, ?it/s]
```

```
Epoch 6 val loss: 2.0199
Training epoch 7...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 7 train loss: 1.9191
Validating epoch 7

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 7 val loss: 1.4063
Saving new best model..
Training epoch 8...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 8 train loss: 2.0019
Validating epoch 8

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 8 val loss: 1.7244
Training epoch 9...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 9 train loss: 2.0396
Validating epoch 9

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 9 val loss: 1.7342
Training epoch 10...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 10 train loss: 1.9262
Validating epoch 10

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 10 val loss: 1.5789

VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded\r'),␣
↪FloatProgress(value=1.0, max=1.0)))

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[128]:
```
# Evaluate best model on validation and test set

checkpoint = torch.load('transformerV2.pth.tar')
loaded_transformer = TransformerV2().to(device)
loaded_transformer.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")


print('val', evaluate(model=loaded_transformer, data_loader=val_loader_tf).
 ↪item())
print('test', evaluate(model=loaded_transformer, data_loader=test_loader_tf).
 ↪item())
```

```
Loaded epoch 6 model

  0%|          | 0/91 [00:00<?, ?it/s]

val 1.600846290588379

  0%|          | 0/110 [00:00<?, ?it/s]

test 0.972445011138916
```

[98]:
```
checkpoint = torch.load('model_ff.pth.tar')
loaded_ff = FeedForwardNN(hidden_size=config['hidden_size']) # Create model␣
 ↪object
loaded_ff.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")

checkpoint = torch.load('model_cnn.pth.tar')
loaded_cnn = CNN(hidden_size=config_cnn['hidden_size']) # Create model object
loaded_cnn.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")

checkpoint = torch.load('model_gru.pth.tar')
loaded_gru = GRU(hidden_size=config_gru['hidden_size'], pipe=to_time_step) #␣
 ↪Create model object
loaded_gru.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")

checkpoint = torch.load('transformer.pth.tar')
loaded_transformer = TransformerModel().to(device)
loaded_transformer.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
```

```
Loaded epoch 9 model
Loaded epoch 7 model
Loaded epoch 6 model
Loaded epoch 6 model
```

[99]:
```
# Also evaluate your fully-connected model and CNN/GRU/Transformer model on the␣
↪test set.
print('ff', evaluate(model=loaded_ff, data_loader=test_loader).item())
```

```
  0%|          | 0/546 [00:00<?, ?it/s]

ff 1.1613706350326538
```

[89]:
```
print('cnn', evaluate(model=loaded_cnn, data_loader=test_loader,␣
↪pipeline=to_grid_channel).item())
```

```
  0%|          | 0/546 [00:00<?, ?it/s]

cnn 1.1606194972991943
```

[90]:
```
print('gru', evaluate(model=loaded_gru, data_loader=test_loader_gru).item())
```

```
  0%|          | 0/110 [00:00<?, ?it/s]

gru 1.152531623840332
```

[88]:
```
print('transformer', evaluate(test_loader_tf, loaded_transformer).item())
```

```
  0%|          | 0/110 [00:00<?, ?it/s]

transformer 1.0053691864013672
```

To get full credit for this part, your best model should be better than the previous models on the **test set**.

16.0.1 TODO#22

Explain what helped and what did not help here

Ans:
help: add positional feed forward, use MultiHeadAttention instead of Self-Attention
didn't help: change start learning rate, initialize paramter

17 [Optional] Augmentation using data loader

17.0.1 Optional TODO#6

Implement a new dataloader on your best model that will perform data augmentation. Try adding noise of zero mean and variance of $10e^{-2}$.

Then, train your model.

[143]:
```
# Write Dataset/DataLoader with noise here
from scipy.stats import norm

def augment_x(x):
    x = x.reshape(-1, 5, 3*5*5)
    return x + norm.rvs(0, np.sqrt(10) * np.exp(-1), x.shape).astype(np.float32)
```

```
train_loader_aug = DataLoader(TransfomerDataset(augment_x(x_train), y_train.
↪reshape(-1, 5)), batch_size=1024, shuffle=True, pin_memory=True)
val_loader_aug = DataLoader(TransfomerDataset(augment_x(x_val), y_val.
↪reshape(-1, 5)), batch_size=1024, shuffle=True, pin_memory=True)
test_loader_aug = DataLoader(TransfomerDataset(augment_x(x_test), y_test.
↪reshape(-1, 5)), batch_size=1024, shuffle=True, pin_memory=True)
```

[144]:
```
print('start training the best model with noise')
################################################################################
#                              WRITE YOUR CODE BELOW                           #
################################################################################

config_transformer = {
    'architecture': 'transformerV2',
    'lr': 0.01,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10,
}

transformerV2 = TransformerV2().to(device)
optimizer_transformer = torch.optim.Adam(transformer.parameters(),␣
↪lr=config_transformer['lr'])
scheduler_transformer = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_transformer,
    mode='min',
    factor=config_transformer['scheduler_factor'],
    patience=config_transformer['scheduler_patience'],
    min_lr = config_transformer['scheduler_min_lr']
)

train_losses = []
val_losses = []
learning_rates = []

wandb.init(
    project='precipitation-nowcasting',
    config=config_transformer,
)
wandb.watch(transformerV2, log='all', log_freq=1)

for epoch in range(config_transformer['epochs']):
    # Training
    train_loss = []
    current_lr = optimizer_transformer.param_groups[0]['lr']
```

```
learning_rates.append(current_lr)

# Flag model as training. Some layers behave differently in training and
# inference modes, such as dropout, BN, etc.
transformerV2.train()

print(f"Training epoch {epoch+1}...")
print(f"Current LR: {current_lr}")

for i, (inputs, y_true) in enumerate(tqdm(train_loader_tf)):
    # Transfer data from cpu to gpu
    inputs = inputs.to(device)
    y_true = y_true.to(device)

    # Reset the gradient
    optimizer_transformer.zero_grad()

    # Predict
    y_pred = transformerV2(inputs)

    # Calculate loss
    loss = loss_fn(y_pred, y_true)

    # Compute gradient
    loss.backward()

    # Update parameters
    optimizer_transformer.step()

    # Log stuff
    train_loss.append(loss)

avg_train_loss = torch.stack(train_loss).mean().item()
train_losses.append(avg_train_loss)

print(f"Epoch {epoch+1} train loss: {avg_train_loss:.4f}")

transformerV2.eval()
with torch.no_grad(): # No gradient is required during validation
    print(f"Validating epoch {epoch+1}")
    val_loss = []
    for i, (inputs, y_true) in enumerate(tqdm(val_loader_tf)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        # Predict
```

```python
            y_pred = transformerV2(inputs)

            # Calculate loss
            loss = loss_fn(y_pred, y_true)

            # Log stuff
            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        # LR adjustment with my_scheduler
        scheduler_transformer.step(avg_val_loss)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': transformerV2.state_dict(),
                'optimizer': optimizer_transformer.state_dict(),
                'scheduler': scheduler_transformer.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss,
                'best_val_loss': best_val_loss,
            }

            print(f"Saving new best model..")
            torch.save(state, 'transformerV2_Noised.pth.tar')

    wandb.log({
        'train_loss': avg_train_loss,
        'val_loss': avg_val_loss,
        'lr': current_lr,
    })


wandb.unwatch()
wandb.finish()
```

```
start training the best model with noise

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Training epoch 1...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 1 train loss: 2.3150
Validating epoch 1

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 1 val loss: 2.0332
Saving new best model..
Training epoch 2...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 2 train loss: 2.2194
Validating epoch 2

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 2 val loss: 2.1802
Training epoch 3...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 3 train loss: 2.3822
Validating epoch 3

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 3 val loss: 1.9123
Saving new best model..
Training epoch 4...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 4 train loss: 2.2238
Validating epoch 4

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 4 val loss: 2.1978
Training epoch 5...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 5 train loss: 2.2590
Validating epoch 5
```

```
  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 5 val loss: 2.0666
Training epoch 6...
Current LR: 0.01

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 6 train loss: 2.2844
Validating epoch 6

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 6 val loss: 2.3246
Training epoch 7...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 7 train loss: 2.2130
Validating epoch 7

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 7 val loss: 1.9791
Training epoch 8...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 8 train loss: 2.1593
Validating epoch 8

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 8 val loss: 2.1944
Training epoch 9...
Current LR: 0.002

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 9 train loss: 2.1421
Validating epoch 9

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 9 val loss: 2.0567
Training epoch 10...
Current LR: 0.0004

  0%|          | 0/225 [00:00<?, ?it/s]

Epoch 10 train loss: 2.3891
Validating epoch 10

  0%|          | 0/91 [00:00<?, ?it/s]

Epoch 10 val loss: 2.0280
```

```
VBox(children=(Label(value='0.039 MB of 0.045 MB uploaded (0.005 MB deduped)\r'),␣
↪FloatProgress(value=0.859578...
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[146]:

```
# Evaluate the best model trained with noise on validation and test set
checkpoint = torch.load('transformerV2.pth.tar')
loaded_transformer = TransformerV2().to(device)
loaded_transformer.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")


print('val', evaluate(model=loaded_transformer, data_loader=val_loader_aug).
↪item())
print('test', evaluate(model=loaded_transformer, data_loader=test_loader_aug).
↪item())
```

```
Loaded epoch 6 model
```

```
  0%|          | 0/91 [00:00<?, ?it/s]
```

```
val 1.6004705429077148
```

```
  0%|          | 0/110 [00:00<?, ?it/s]
```

```
test 1.0926133394241333
```