พอดีพึ่งเห็นว่า pdf ที่ส่งไปก่อนหน้า (ส่งทัน) มันบัคทำให้คำตอบบางข้อหายไปเลยครับ เลยหาตัว export อื่นส่งมาอีกรอบ ถ้าไม่หักคะแนนส่งสายจะดีใจมากครับ 😭🙏🙇🙏🙇🙏🙇🙏🙇🙏🙇

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).

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

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.

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.

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

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

Weights and Biases

Weights and Biases (https://docs.wandb.ai/company) (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

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

Setup

1. Register Wandb account (https://wandb.ai/login?signup=true) (and confirm your email)
2. `wandb login` and copy paste the API key when prompt

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

In [1]:
```
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
```

Out[1]:
```
'2.2.1'
```

In [2]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

Out[2]:
```
device(type='cuda')
```

Loading the data

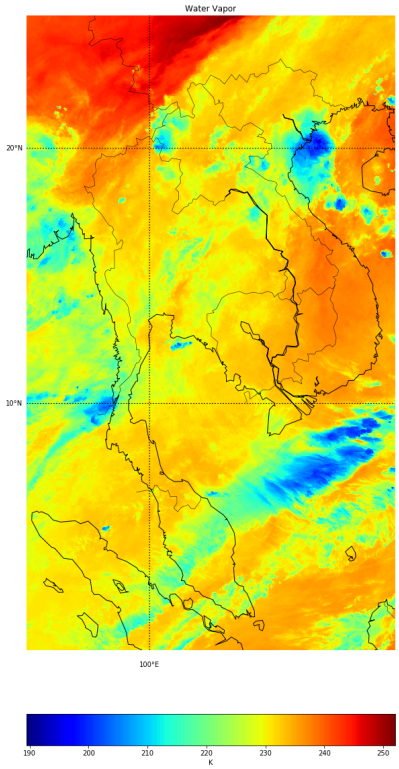
Get the data set by going here (https://drive.google.com/file/d/1NWR22fVVE0tO2Q5EbaPPrRKPhUem-jbw/view?usp=sharing) and click add to drive.

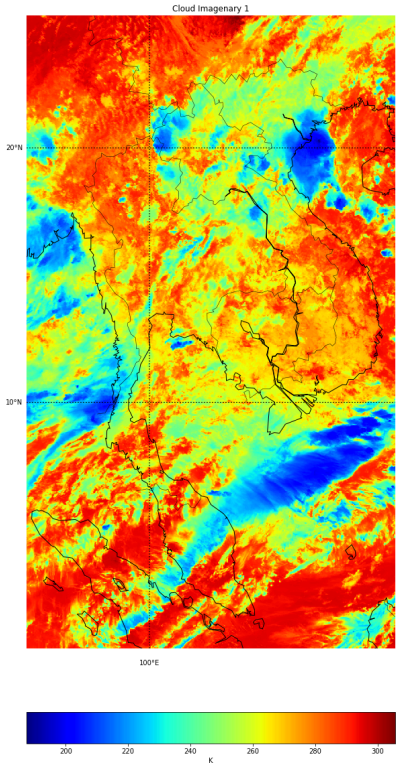
In [6]:
```
# from google.colab import drive
# drive.mount('/content/gdrive/')
```

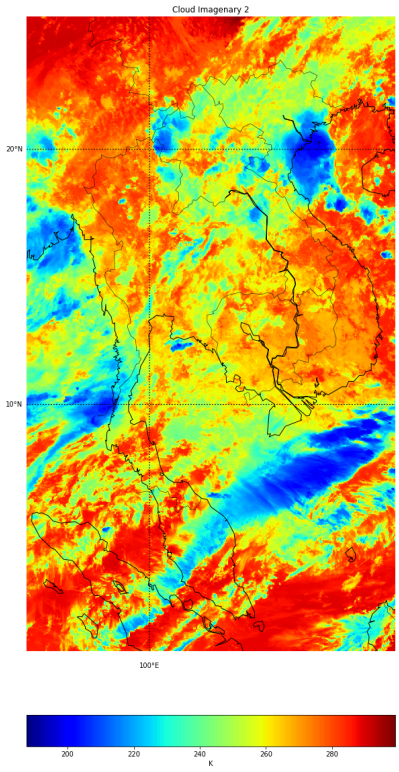
In [7]:
```
# !tar -xvf '/content/gdrive/My Drive/nowcastingHWdataset.tar.gz'
```

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.

Reading data

In [3]:
```
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
```

In [4]:
```
# 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

Three-Layer Feedforward Neural Networks

In [5]:
```
# 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)
```

TODO#1

Explain each line of code in the function preprocess_for_ff()

Ans: flatten data.

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 (https://pytorch.org/docs/stable/data.html#map-style-datasets))

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.

1. 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 (https://developers.google.com/machine-learning/data-prep/transform/normalization) (or google) if you wish to learn more about data normalization.

1. 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 (https://pytorch.org/vision/master/transforms.html) if you want to know more.

In [6]:
```
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]
```

In [7]:
```
def normalizer_std(X):
    scaler = preprocessing.StandardScaler().fit(X)
    return scaler

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

In [8]:
```
normalizer = normalizer_std(x_train_ff) # We will normalize everything based o
n 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)
```

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 (https://pytorch.org/docs/stable/data.html)

In [9]:
```
train_loader = DataLoader(train_dataset, batch_size=1024, shuffle=True, pin_me
mory=True)
val_loader = DataLoader(val_dataset, batch_size=1024, shuffle=False, pin_memor
y=True)
test_loader = DataLoader(test_dataset, batch_size=1024, shuffle=False, pin_mem
ory=True)
```

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 (https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html)

In [10]:
```
loss_fn = nn.MSELoss()
```

TODO#2

Why is the loss MSE?

Ans: Because, this is regression task, and it is easy to find derivative.

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.

In [11]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

Out[11]:
```
device(type='cuda')
```

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.

In [60]:
```
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)
```

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: ReLU, because atmospheric water vapor should not less than 0. So, I think ReLU is the best.

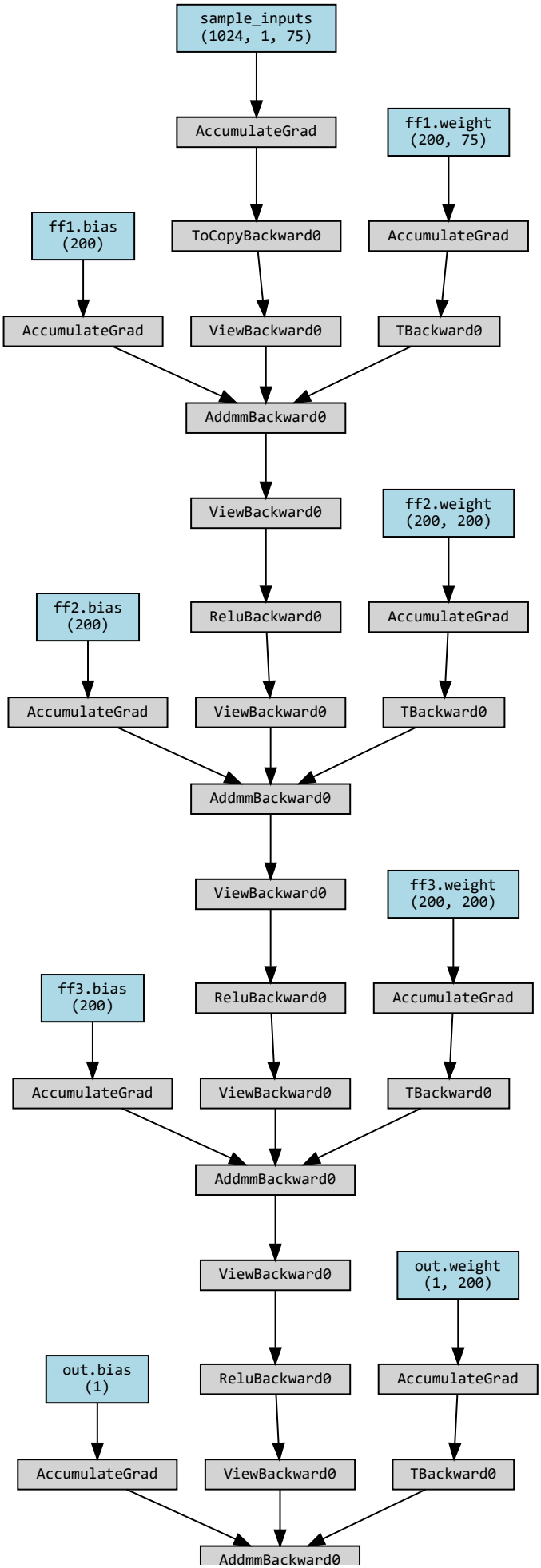
In [61]:

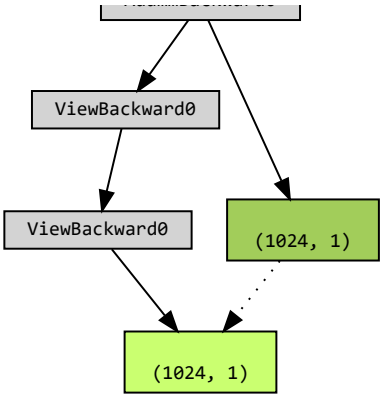
```
# Hyperparameters and other configs
config = {
    'architecture': 'feedforward',
    'lr': 0.01,
    'hidden_size': 200,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 10
}

# 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']
)
```

In [14]:

```
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_inp
uts', sample_inputs)]))
```

Out[14]:


sample_inputs
(1024, 1, 75)
AccumulateGrad
ff1.weight
(200, 75)
ff1.bias
(200)
ToCopyBackward0
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



In [15]:
```
summary(model_ff, input_size=(1024, 75))
```

Out[15]:
```
==========================================================================
=============
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 (Units.MEGABYTES): 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
==========================================================================
=============
```

TODO#4

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

Ans: Because, input has 75 features map to 200 node in first layer. Therefore, it has 75 x 200 = 15200 parameters.

Training

In [19]:

```
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')

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_ff2.pth.tar')

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

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: demonstem. Use `wandb login --relogin` to forc
e relogin
```

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240319_235818-4q5utifx`

Syncing run **chocolate-sunset-10 (https://wandb.ai/demonstem/precipitation-nowcasting/runs/4q5utifx)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting (https://wandb.ai/demonstem/precipitation-nowcasting)

View run at https://wandb.ai/demonstem/precipitation-nowcasting/runs/4q5utifx (https://wandb.ai/demonstem/precipitation-nowcasting/runs/4q5utifx)

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


Epoch 1 train loss: 1.9268
Validating epoch 1


Epoch 1 val loss: 1.6583
Saving new best model..
Training epoch 2...
Current LR: 0.01


Epoch 2 train loss: 1.9210
Validating epoch 2


Epoch 2 val loss: 1.6590
Training epoch 3...
Current LR: 0.01


Epoch 3 train loss: 1.9232
Validating epoch 3


Epoch 3 val loss: 1.6614
Training epoch 4...
Current LR: 0.01


Epoch 4 train loss: 1.9234
Validating epoch 4


Epoch 4 val loss: 1.6632
Training epoch 5...
Current LR: 0.002
```

```
Epoch 5 train loss: 1.9229
Validating epoch 5

Epoch 5 val loss: 1.6587
Training epoch 6...
Current LR: 0.002

Epoch 6 train loss: 1.9195
Validating epoch 6

Epoch 6 val loss: 1.6577
Saving new best model..
Training epoch 7...
Current LR: 0.002

Epoch 7 train loss: 1.9185
Validating epoch 7

Epoch 7 val loss: 1.6566
Saving new best model..
Training epoch 8...
Current LR: 0.002

Epoch 8 train loss: 1.9182
Validating epoch 8

Epoch 8 val loss: 1.6566
Saving new best model..
Training epoch 9...
Current LR: 0.002

Epoch 9 train loss: 1.9181
Validating epoch 9

Epoch 9 val loss: 1.6561
Saving new best model..
Training epoch 10...
Current LR: 0.002

Epoch 10 train loss: 1.9183
Validating epoch 10

Epoch 10 val loss: 1.6562
```

Run history:

Run summary:

lr	0.002
train_loss	1.91827
val_loss	1.65616

View run **chocolate-sunset-10** at: https://wandb.ai/demonstem/precipitation-nowcasting/runs/4q5utifx (https://wandb.ai/demonstem/precipitation-nowcasting/runs/4q5utifx)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240319_235818-4q5utifx\logs`

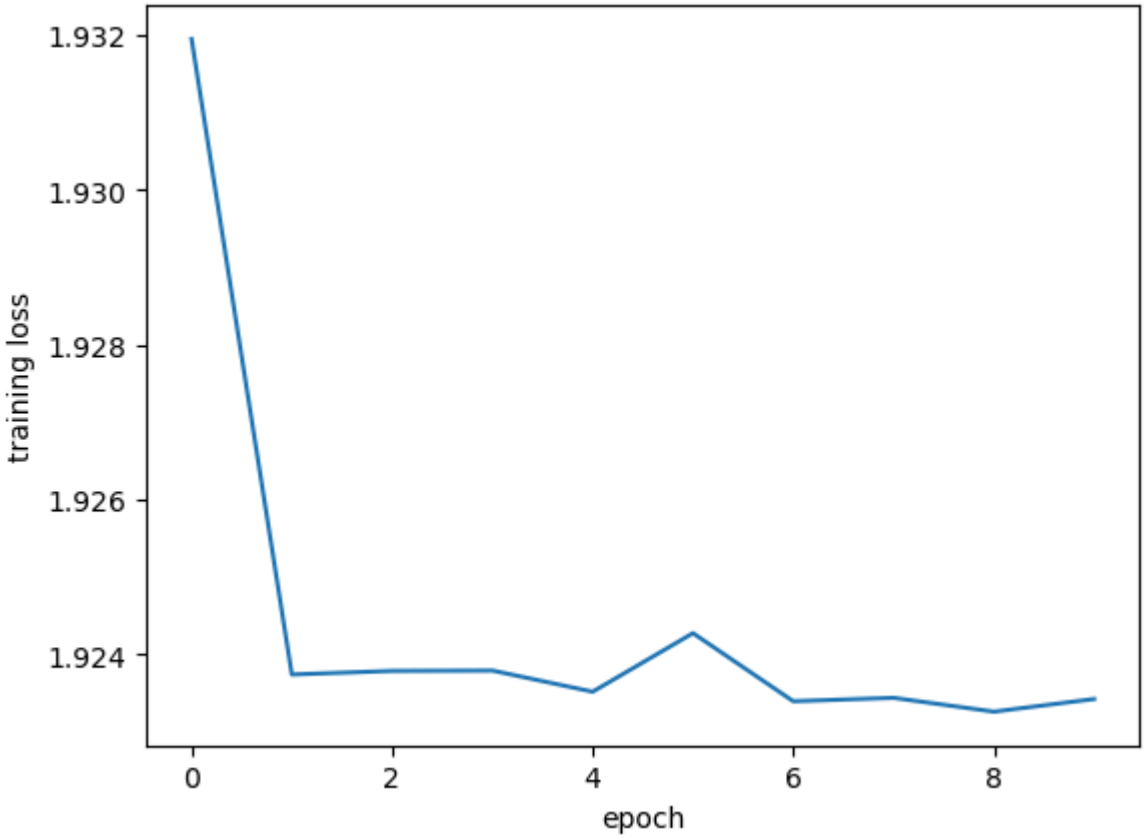
```
Finished Training
```

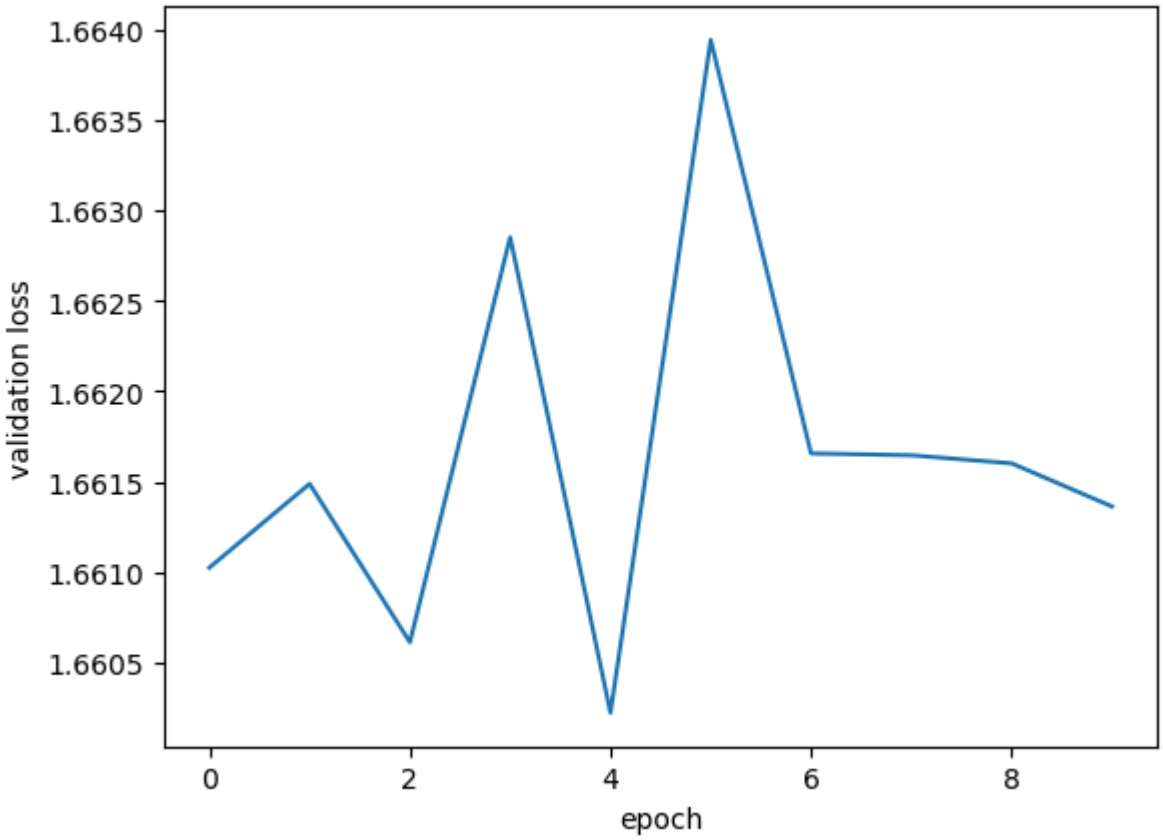
TODO#5

Plot loss and val_loss as a function of epochs.

In [17]:

```
plt.plot(train_losses)
plt.xlabel('epoch')
plt.ylabel('training loss')
plt.show()
plt.plot(val_losses)
plt.xlabel('epoch')
plt.ylabel('validation loss')
plt.show()
```


1.932
1.930
1.928
1.926
1.924
training loss
0
2
4
6
8
epoch


1.6640
1.6635
1.6630
1.6625
1.6620
1.6615
1.6610
1.6605
validation loss
0
2
4
6
8
epoch

TODO#6

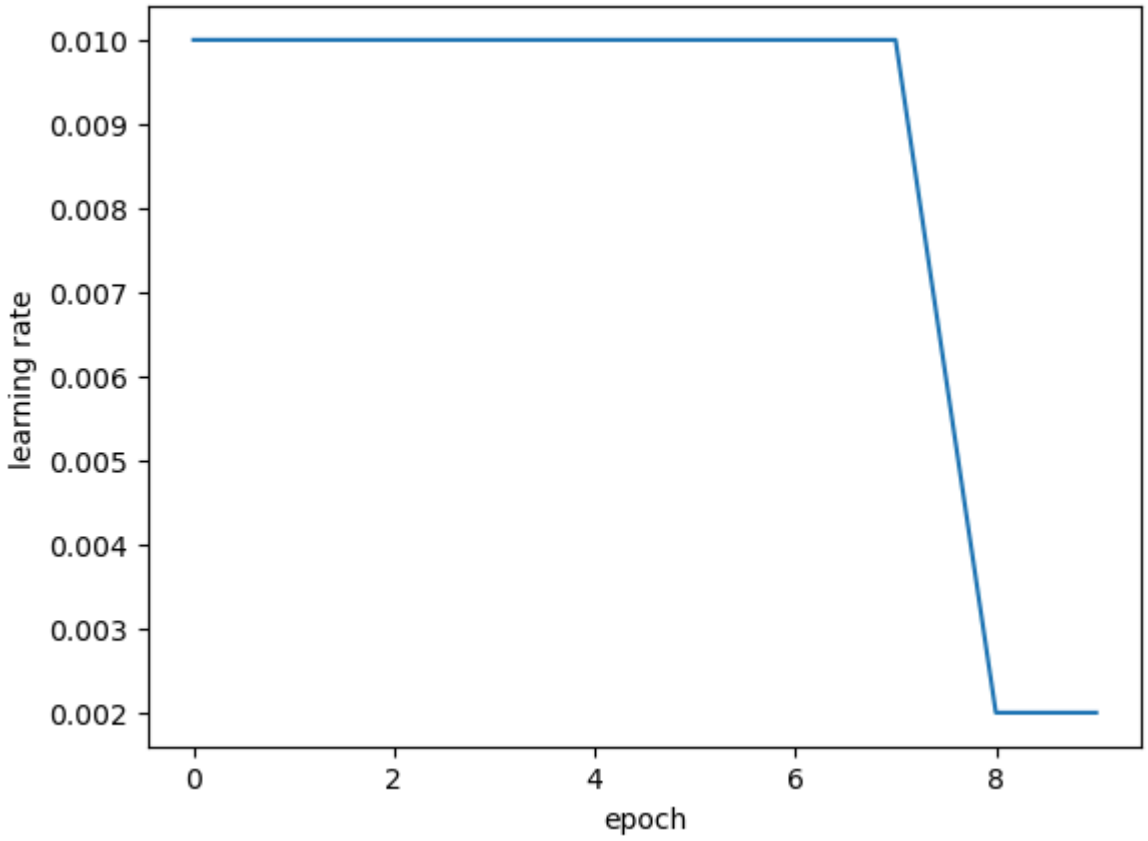
When does the model start to overfit?

Ans: at epoch 1 (observe from both train loss and val loss).

TODO#7

Plot the learning rate as a function of the epochs.

```
In [18]: plt.plot(learning_rates)
         plt.xlabel('epoch')
         plt.ylabel('learning rate')
         plt.show()
```



TODO#8

What makes the learning rate change? (hint: try to understand the scheduler ReduceLROnPlateau (https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html))

Ans: learning rate is reduced when val_loss is not getting better.

Load Model

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

In [15]:
```
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 4 model
```

In [16]:
```
model_ff = loaded_model
```

A more complex scheduling

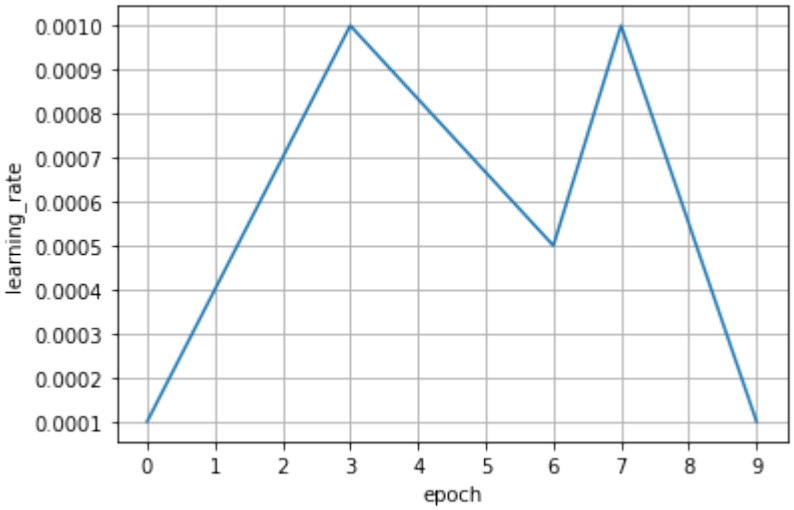
The scheduler can be very complicated and you can write your own heuristic for it.

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 (https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate)](https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate) in order to build your own.

Learning rate should be function of epoch.



In [17]:

```python
# Implement scheduler here
class MyScheduler():
    def __init__(self, optimizer: torch.optim.Optimizer):
        self.slope = [0.0003, 0.0003, 0.0003, -0.0005/3, -0.0005/3, -0.0005/3,
0.0005, -0.0009/2, -0.0009/2]
        self.lr = 0.0001
        self.optimizer = optimizer
        # for param_group in self.optimizer.param_groups:
        #         param_group['lr'] = self.lr
    def step(self, epoch):
        if epoch >= len(self.slope):
            print('!exceed 10 epoch!')
        else:
            self.lr += self.slope[epoch]
            for param_group in self.optimizer.param_groups:
                param_group['lr'] = self.lr
```

In [18]:

```
# Now train with your scheduler
# my_scheduler = MyScheduler(...)

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

model_custom_scheduler = FeedForwardNN(hidden_size=custom_scheduler_config['hi
dden_size'])
model_custom_scheduler = model_custom_scheduler.to(device)

optimizer = torch.optim.Adam(model_custom_scheduler.parameters(), lr=custom_sc
heduler_config['lr'])
my_scheduler = MyScheduler(optimizer)

train_losses_cs = []
val_losses_cs = []
learning_rates_cs = []

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

# Log parameters and gradients
wandb.watch(model_custom_scheduler, log='all')

for epoch in range(custom_scheduler_config['epochs']):  # loop over the datase
t multiple times

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

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

```
        # Predict
        y_pred = model_custom_scheduler(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_cs.append(avg_train_loss)

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

    # Validation
    model_custom_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_custom_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_cs.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        # LR adjustment with scheduler
        my_scheduler.step(epoch)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses_cs[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': model_custom_scheduler.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_custom_scheduler.pth.tar')

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

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: demonstem. Use `wandb login --relogin` to forc
e relogin
```

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240318_230820-0ly9qv4s`

Syncing run **lilac-monkey-6 (https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler/runs/0ly9qv4s)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler (https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler/runs/0ly9qv4s (https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler/runs/0ly9qv4s)

```
Training epoch 1...
Current LR: 0.0001


Epoch 1 train loss: 1.9192
Validating epoch 1


Epoch 1 val loss: 1.6572
Saving new best model..
Training epoch 2...
Current LR: 0.00039999999999999996


Epoch 2 train loss: 1.9189
Validating epoch 2


Epoch 2 val loss: 1.6579
Training epoch 3...
Current LR: 0.00069999999999999999


Epoch 3 train loss: 1.9183
Validating epoch 3


Epoch 3 val loss: 1.6563
Saving new best model..
Training epoch 4...
Current LR: 0.00099999999999999998


Epoch 4 train loss: 1.9192
Validating epoch 4
```

```
Epoch 4 val loss: 1.6587
Training epoch 5...
Current LR: 0.0008333333333333332

Epoch 5 train loss: 1.9184
Validating epoch 5

Epoch 5 val loss: 1.6565
Training epoch 6...
Current LR: 0.0006666666666666665

Epoch 6 train loss: 1.9179
Validating epoch 6

Epoch 6 val loss: 1.6560
Saving new best model..
Training epoch 7...
Current LR: 0.0004999999999999999

Epoch 7 train loss: 1.9176
Validating epoch 7

Epoch 7 val loss: 1.6563
Training epoch 8...
Current LR: 0.001

Epoch 8 train loss: 1.9183
Validating epoch 8

Epoch 8 val loss: 1.6569
Training epoch 9...
Current LR: 0.00055

Epoch 9 train loss: 1.9175
Validating epoch 9

Epoch 9 val loss: 1.6566
Training epoch 10...
Current LR: 0.00010000000000000005

Epoch 10 train loss: 1.9175
Validating epoch 10

Epoch 10 val loss: 1.6557
!exceed 10 epoch!
Saving new best model..
```

Run history:

lr 

train_loss 

val_loss 

Run summary:

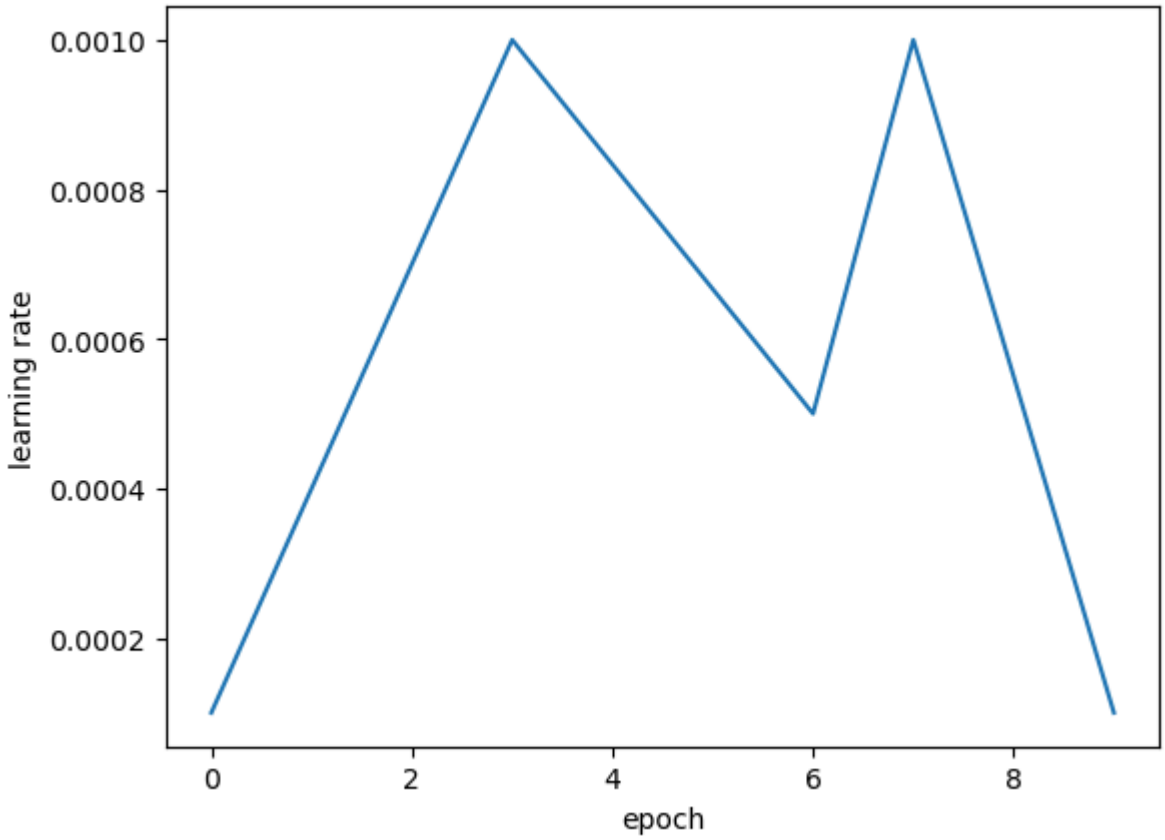
lr	0.0001
train_loss	1.9175
val_loss	1.6557

View run **lilac-monkey-6** at: https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler/runs/0ly9qv4s (https://wandb.ai/demonstem/precipitation-nowcasting-custom-scheduler/runs/0ly9qv4s)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240318_230820-0ly9qv4s\logs`

```
Finished Training
```

In [19]:
```
plt.plot(learning_rates_cs)
plt.xlabel('epoch')
plt.ylabel('learning rate')
plt.show()
```



[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.

Wandb Observation

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: it overfitted at epoch 1 since the training loss did not increased but the validation loss is increased.

Evaluation

In [71]:
```
################################################################################
##
# 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):
    """
    Evaluate model on validation data given by data_loader
    """
    # write code here
    losses = []
    for inputs, y_true in tqdm(data_loader):
        inputs, y_true = inputs.to(device), y_true.to(device)
        y_pred = model(inputs)
        losses.append(loss_fn(y_pred, y_true))
    return sum(losses) / len(losses)
```

In [21]:
```
# 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
```

In [47]:
```
print('baseline')
print('train', majority_baseline(y_train))
print('validate', majority_baseline(y_val))
```

```
baseline
train [1.94397725]
validate [1.6746546]
```

In [24]:
```
print('FF-model')
model_ff = model_ff.to(device)
print('train', evaluate(train_loader, model_ff).item())
print('validate', evaluate(val_loader, model_ff).item())
```

```
FF-model

train 1.9241724014282227

validate 1.6602238416671753
```

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 (https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html)

In [41]:
```
################################################################################
##
# TODO#11:
#
# Write a feedforward model with dropout
#
################################################################################
##
#                              WRITE YOUR CODE BELOW
#
################################################################################
##
class DropOutNN(nn.Module):
    def __init__(self, hidden_size=200):
        super(DropOutNN, 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)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = F.relu(self.ff1(x))
        x = self.dropout(x)
        x = F.relu(self.ff2(x))
        x = self.dropout(x)
        x = F.relu(self.ff3(x))
        x = self.dropout(x)
        x = self.out(x)
        return x.reshape(-1, 1)
```

In [42]:

```
################################################################################
##
# TODO#12:
#
# Complete the code to train your dropout model
#
################################################################################
##
print('start training ff dropout')
################################################################################
##
#                               WRITE YOUR CODE BELOW
#
################################################################################
##
# Hyperparameters and other configs
dropout_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_dropout = DropOutNN(hidden_size=dropout_config['hidden_size'])
model_dropout = model_dropout.to(device)
optimizer_dropout = torch.optim.Adam(model_dropout.parameters(), lr=dropout_co
nfig['lr'])
scheduler_dropout = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_dropout,
    'min',
    factor=dropout_config['scheduler_factor'],
    patience=dropout_config['scheduler_patience'],
    min_lr=dropout_config['scheduler_min_lr']
)

# Training
train_losses_dropout = []
val_losses_dropout = []
learning_rates_dropout = []

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

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

for epoch in range(dropout_config['epochs']):  # loop over the dataset multipl
e times
```

```
    # Training
    train_loss = []
    current_lr = optimizer_dropout.param_groups[0]['lr']
    learning_rates_dropout.append(current_lr)

    # Flag model as training. Some layers behave differently in training and
    # inference modes, such as dropout, BN, etc.
    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)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        # Reset the gradient
        optimizer_dropout.zero_grad()

        # Predict
        y_pred = model_dropout(inputs)

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Compute gradient
        loss.backward()

        # Update parameters
        optimizer_dropout.step()

        # Log stuff
        train_loss.append(loss)

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

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

    # Validation
    model_dropout.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_dropout(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_dropout.append(avg_val_loss)
        print(f"Epoch {epoch+1} val loss: {avg_val_loss:.4f}")

        # LR adjustment with scheduler
        scheduler_dropout.step(epoch)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses_dropout[:-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.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.finish()
print('Finished Training')
```

```
start training ff dropout
```

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240319_015528-cfyc2qhk`

Syncing run **wandering-sound-1 (https://wandb.ai/demonstem/precipitation-nowcasting-dropout/runs/cfyc2qhk)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-dropout) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-dropout (https://wandb.ai/demonstem/precipitation-nowcasting-dropout)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-dropout/runs/cfyc2qhk (https://wandb.ai/demonstem/precipitation-nowcasting-dropout/runs/cfyc2qhk)

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


Epoch 1 train loss: 1.9359
Validating epoch 1


Epoch 1 val loss: 1.6621
Saving new best model..
Training epoch 2...
Current LR: 0.01


Epoch 2 train loss: 1.9233
Validating epoch 2


Epoch 2 val loss: 1.6605
Saving new best model..
Training epoch 3...
Current LR: 0.01


Epoch 3 train loss: 1.9237
Validating epoch 3


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


Epoch 4 train loss: 1.9235
Validating epoch 4


Epoch 4 val loss: 1.6606
Training epoch 5...
Current LR: 0.002


Epoch 5 train loss: 1.9235
Validating epoch 5
```

```
Epoch 5 val loss: 1.6627
Training epoch 6...
Current LR: 0.002

Epoch 6 train loss: 1.9240
Validating epoch 6

Epoch 6 val loss: 1.6609
Training epoch 7...
Current LR: 0.002

Epoch 7 train loss: 1.9233
Validating epoch 7

Epoch 7 val loss: 1.6610
Training epoch 8...
Current LR: 0.0004

Epoch 8 train loss: 1.9236
Validating epoch 8

Epoch 8 val loss: 1.6616
Training epoch 9...
Current LR: 0.0004

Epoch 9 train loss: 1.9233
Validating epoch 9

Epoch 9 val loss: 1.6610
Training epoch 10...
Current LR: 0.0004

Epoch 10 train loss: 1.9233
Validating epoch 10

Epoch 10 val loss: 1.6613
```

Run history:

Run summary:

lr	0.0004
train_loss	1.92333
val_loss	1.66125

View run **wandering-sound-1** at: https://wandb.ai/demonstem/precipitation-nowcasting-dropout/runs/cfyc2qhk (https://wandb.ai/demonstem/precipitation-nowcasting-dropout/runs/cfyc2qhk)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

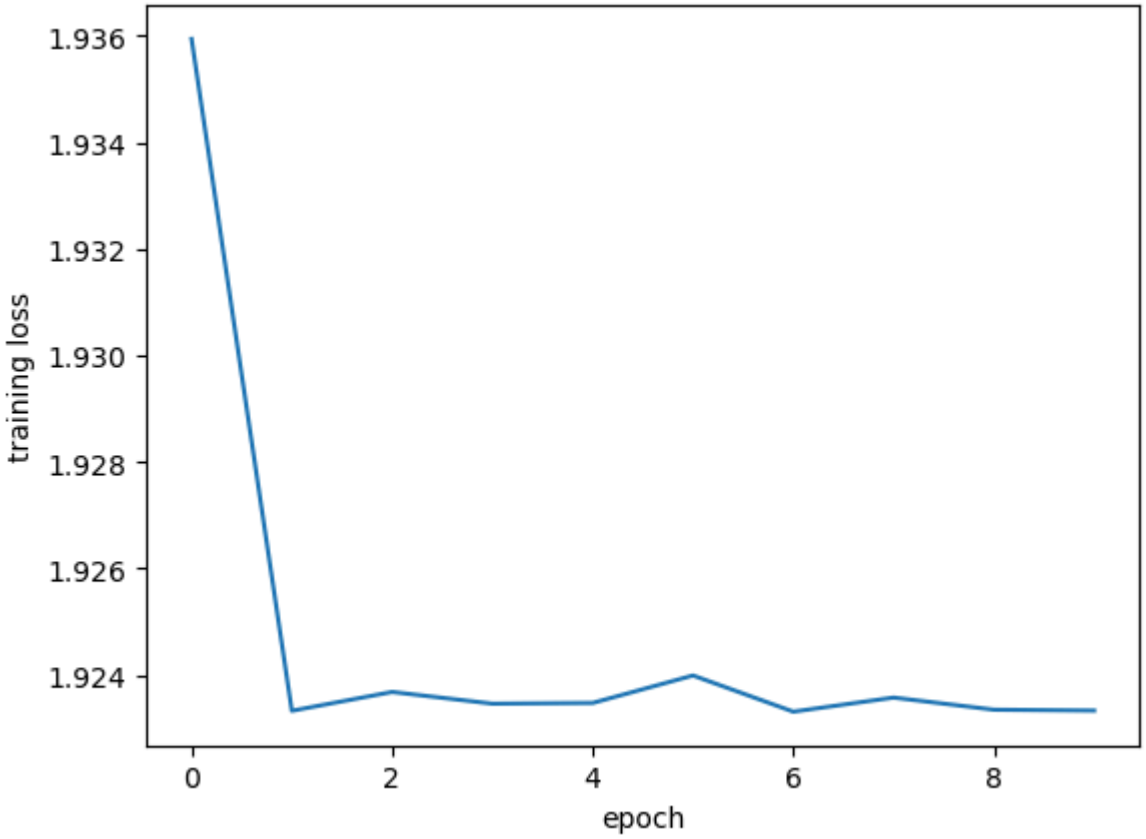
Find logs at: `.\wandb\run-20240319_015528-cfyc2qhk\logs`

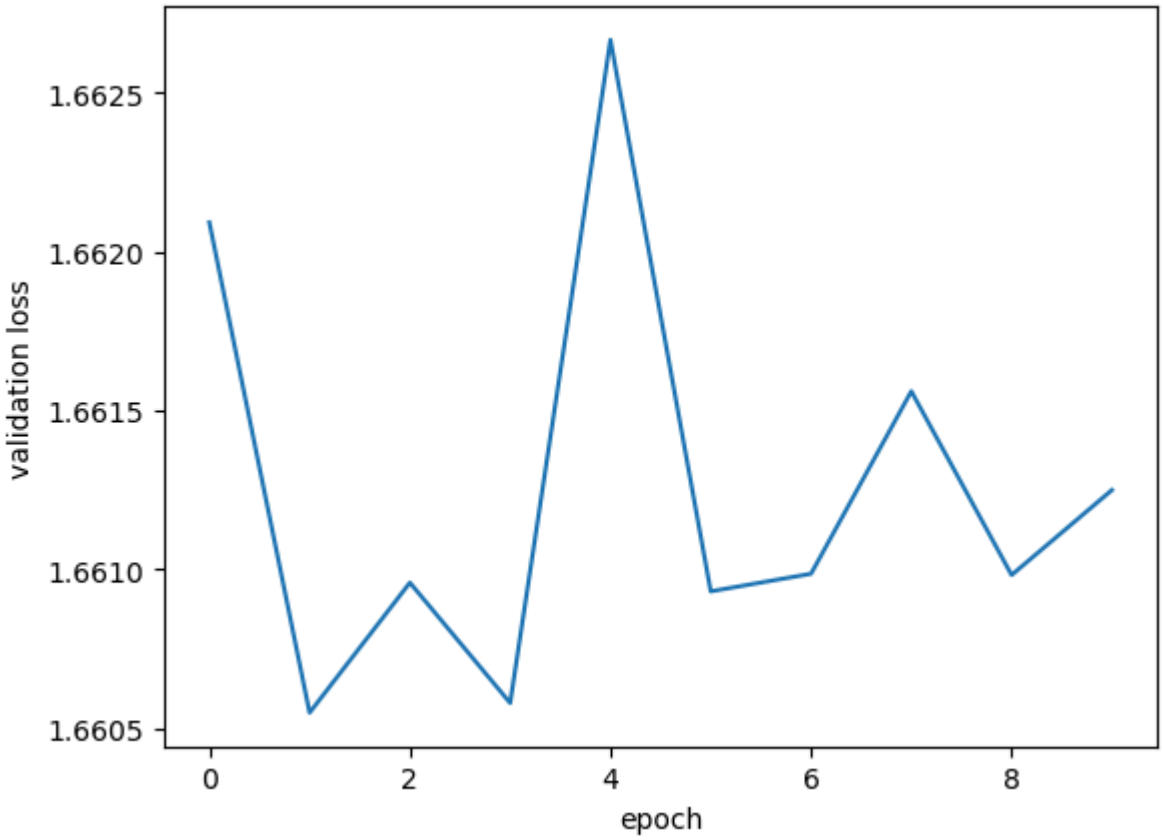
```
Finished Training
```

TODO#13

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

In [64]:
```
# Plot here
plt.plot(train_losses_dropout)
plt.xlabel('epoch')
plt.ylabel('training loss')
plt.show()
plt.plot(val_losses_dropout)
plt.xlabel('epoch')
plt.ylabel('validation loss')
plt.show()
```


1.936
1.934
1.932
1.930
1.928
1.926
1.924
training loss
0
2
4
6
8
epoch


1.6625
1.6620
1.6615
1.6610
1.6605
validation loss
0
2
4
6
8
epoch

In [63]:
```
# Evaluate
print('Dropout-model')
print('train', evaluate(train_loader, model_dropout).item())
print('validate', evaluate(val_loader, model_dropout).item())
```

```
Dropout-model

train 1.92324960231781

validate 1.6612495183944702
```

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.

In [63]:

```
################################################################################
##
# 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_cnn_inputs(x):
    return torch.reshape(x, (-1, 3, 5, 5))
```

In [64]:

```
################################################################################
##
# 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(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=hidden_size, kernel
_size=3)
        self.ff1 = nn.Linear(hidden_size * 3 * 3, hidden_size)
        self.ff2 = nn.Linear(hidden_size, hidden_size)
        self.ff3 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.flatten(x, 1)
        x = torch.relu(self.ff1(x))
        x = torch.relu(self.ff2(x))
        x = self.ff3(x)
        return x

# Hyperparameters and configs
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
model_cnn = CNN(hidden_size=config_cnn['hidden_size'])
model_cnn = model_cnn.to(device)
summary(model_cnn, input_size=(1024, 3, 5, 5))
```

Out[64]:
```
==========================================================================
=============
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 (Units.MEGABYTES): 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
==========================================================================
=============
```

In [21]:

```
##############################################################################
##
# TODO#16:
#
# Complete the code to train your cnn model
#
##############################################################################
##
print('start training conv2d')
##############################################################################
##
#                               WRITE YOUR CODE BELOW
#
##############################################################################
##
optimizer_cnn = torch.optim.Adam(model_cnn.parameters(), lr=config_cnn['lr'])
scheduler_cnn = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_cnn,
    'min',
    factor=config_cnn['scheduler_factor'],
    patience=config_cnn['scheduler_patience'],
    min_lr=config_cnn['scheduler_min_lr']
)

train_losses_cnn = []
val_losses_cnn = []
learning_rates_cnn = []

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

# Log parameters and gradients
wandb.watch(model_cnn, log='all')

for epoch in range(config_cnn['epochs']):  # loop over the dataset multiple ti
mes

    # Training
    train_loss = []
    current_lr = optimizer_cnn.param_groups[0]['lr']
    learning_rates_cnn.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_cnn_inputs(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_cnn.append(avg_train_loss)

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

    # Validation
    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_cnn_inputs(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_cnn.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_cnn[:-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.finish()
print('Finished Training')
```

```
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: demonstem. Use `wandb login --relogin` to forc
e relogin
```

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_002541-9uheewvx`

Syncing run **polished-puddle-2 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/9uheewvx)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-cnn) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-cnn (https://wandb.ai/demonstem/precipitation-nowcasting-cnn)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/9uheewvx (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/9uheewvx)

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


Epoch 1 train loss: 2.1246
Validating epoch 1


Epoch 1 val loss: 1.6603
Saving new best model..
Training epoch 2...
Current LR: 0.01


Epoch 2 train loss: 1.9218
Validating epoch 2


Epoch 2 val loss: 1.6595
Saving new best model..
Training epoch 3...
Current LR: 0.01


Epoch 3 train loss: 1.9210
Validating epoch 3


Epoch 3 val loss: 1.6581
Saving new best model..
Training epoch 4...
Current LR: 0.01


Epoch 4 train loss: 1.9201
Validating epoch 4
```

```
Epoch 4 val loss: 1.6604
Training epoch 5...
Current LR: 0.01

Epoch 5 train loss: 1.9207
Validating epoch 5

Epoch 5 val loss: 1.6582
Training epoch 6...
Current LR: 0.01

Epoch 6 train loss: 1.9203
Validating epoch 6

Epoch 6 val loss: 1.6605
Training epoch 7...
Current LR: 0.002

Epoch 7 train loss: 1.9200
Validating epoch 7

Epoch 7 val loss: 1.6581
Training epoch 8...
Current LR: 0.002

Epoch 8 train loss: 1.9199
Validating epoch 8

Epoch 8 val loss: 1.6590
Training epoch 9...
Current LR: 0.002

Epoch 9 train loss: 1.9198
Validating epoch 9

Epoch 9 val loss: 1.6587
Training epoch 10...
Current LR: 0.0004

Epoch 10 train loss: 1.9196
Validating epoch 10

Epoch 10 val loss: 1.6583
```

Run history:

Run summary:

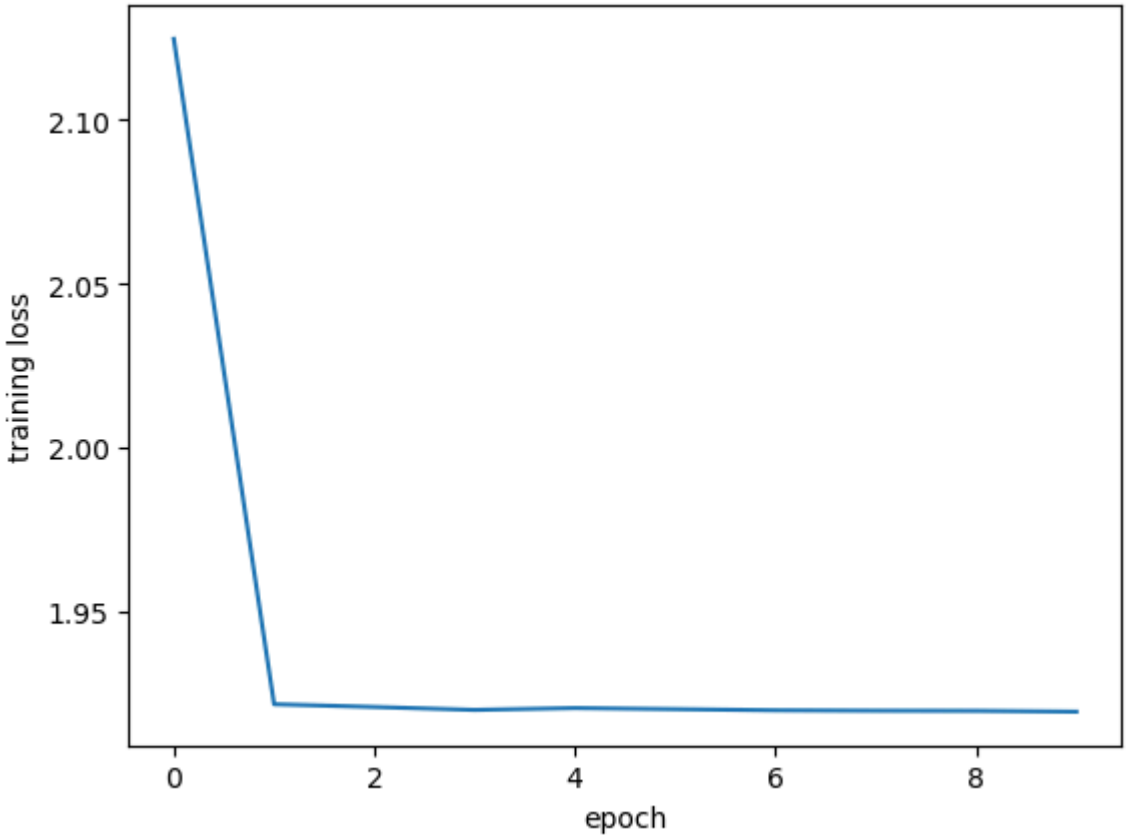
lr	0.0004
train_loss	1.91957
val_loss	1.65834

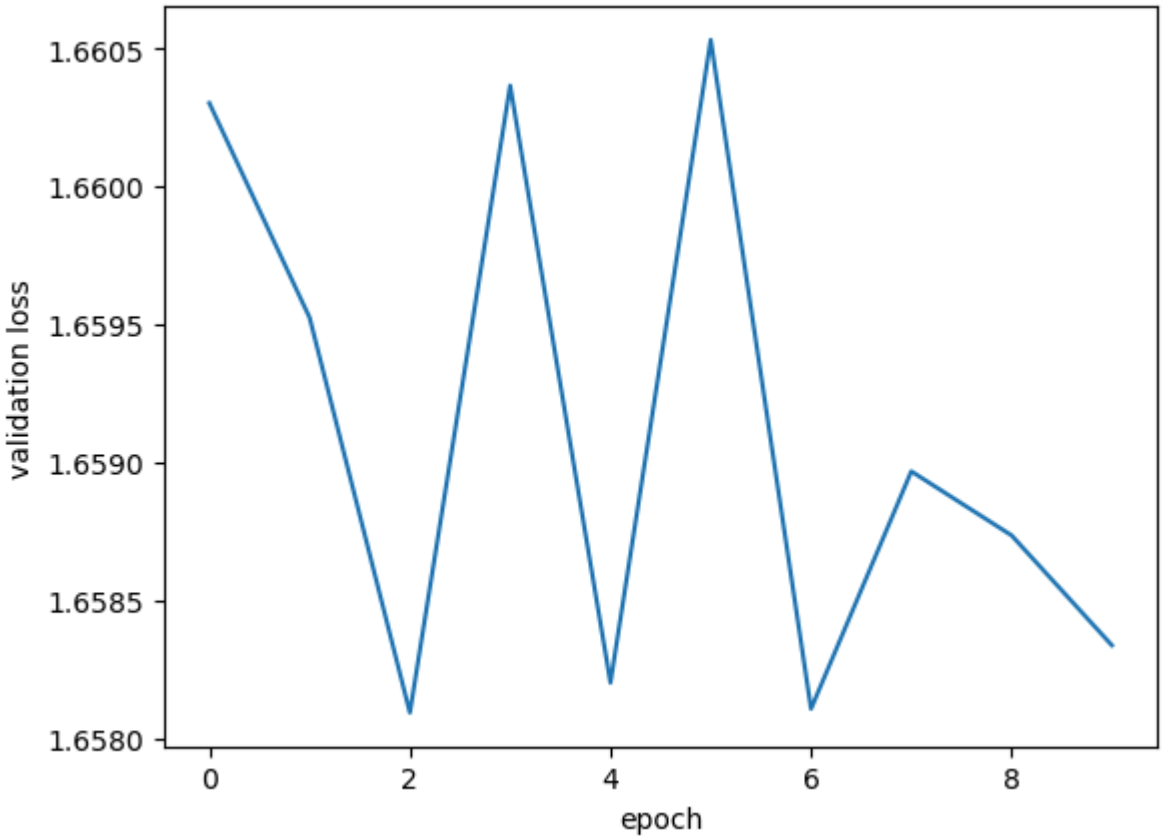
View run **polished-puddle-2** at: https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/9uheewvx (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/9uheewvx)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_002541-9uheewvx\logs`

```
Finished Training
```

In [22]:
```
# Plot losses
plt.plot(train_losses_cnn)
plt.xlabel('epoch')
plt.ylabel('training loss')
plt.show()
plt.plot(val_losses_cnn)
plt.xlabel('epoch')
plt.ylabel('validation loss')
plt.show()
```


training loss
2.10
2.05
2.00
1.95
0
2
4
6
8
epoch


validation loss
1.6605
1.6600
1.6595
1.6590
1.6585
1.6580
0
2
4
6
8
epoch

In [15]:
```
checkpoint = torch.load('model_cnn.pth.tar')
loaded_model = CNN(hidden_size=config_cnn['hidden_size']) # Create model objec
t
loaded_model.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
model_cnn = loaded_model
model_cnn.to(device)
```

```
Loaded epoch 2 model
```

Out[15]:
```
CNN(
  (conv1): Conv2d(3, 200, kernel_size=(3, 3), stride=(1, 1))
  (ff1): Linear(in_features=1800, out_features=200, bias=True)
  (ff2): Linear(in_features=200, out_features=200, bias=True)
  (ff3): Linear(in_features=200, out_features=1, bias=True)
)
```

In [65]:
```
def evaluate_cnn(data_loader, model):
    wandb.init(
        project='precipitation-nowcasting-cnn',
    )

    # Log parameters and gradients
    wandb.watch(model, log='all')
    with torch.no_grad():
      train_losses = []
      for i, (inputs, y_true) in enumerate(tqdm(data_loader)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        # Predict
        y_pred = model(to_cnn_inputs(inputs))

        # Calculate loss
        loss = loss_fn(y_pred, y_true)

        # Log stuff
        train_losses.append(loss)

      mse = torch.stack(train_losses).mean()
      wandb.finish()
      return mse
```

In [66]:

```
# Evaluate
evaluated_train_cnn = evaluate_cnn(train_loader, model_cnn).item()
evaluated_val_cnn = evaluate_cnn(val_loader, model_cnn).item()

print('cnn-model')
print('train', evaluated_train_cnn)
print('validate', evaluated_val_cnn)
```

Finishing last run (ID:72op9xla) before initializing another...

View run **light-glitter-6** at: https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/72op9xla (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/72op9xla)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_212520-72op9xla\logs`

Successfully finished last run (ID:72op9xla). Initializing new run:

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_212641-uliy2zl5`

Syncing run **hearty-firefly-7 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/uliy2zl5)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-cnn) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-cnn (https://wandb.ai/demonstem/precipitation-nowcasting-cnn)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/uliy2zl5 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/uliy2zl5)

View run **hearty-firefly-7** at: https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/uliy2zl5 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/uliy2zl5)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_212641-uliy2zl5\logs`

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_212852-2r3l4brd`

Syncing run **sparkling-vortex-8 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/2r3l4brd)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-cnn) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-cnn (https://wandb.ai/demonstem/precipitation-nowcasting-cnn)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/2r3l4brd (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/2r3l4brd)

View run **sparkling-vortex-8** at: https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/2r3l4brd (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/2r3l4brd)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_212852-2r3l4brd\logs`

```
cnn-model
train 1.9381484985351562
validate 1.6678866147994995
```

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 (https://pytorch.org/docs/stable/generated/torch.nn.GRU.html).

In [24]:

```
################################################################################
##
# 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 preprocess_for_gru(x_train, y_train, x_val, y_val):
  x_train_gru = x_train.reshape((-1, 5, 5*5*3))
  y_train_gru = y_train.reshape((-1, 5))
  x_val_gru = x_val.reshape((-1, 5, 5*5*3))
  y_val_gru = y_val.reshape((-1, 5))
  x_test_gru = x_test.reshape((-1, 5, 5*5*3))
  y_test_gru = y_test.reshape((-1, 5))

  return x_train_gru, y_train_gru, x_val_gru, y_val_gru, x_test_gru, y_test_gr
u


x_train_gru, y_train_gru, x_val_gru, y_val_gru, x_test_gru, y_test_gru = prepr
ocess_for_gru(x_train, y_train, x_val, y_val)


class RainfallDatasetGRU(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
        x = x.reshape(5, 75)
        y = self.y[index]
        return x, y

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

normalizer_gru = normalizer_std(x_train_gru.reshape(-1, 5*5*5*3)) # We will no
rmalize everything based on x_train
train_dataset_gru = RainfallDatasetGRU(x_train_gru, y_train_gru, normalizer_gr
u)
val_dataset_gru = RainfallDatasetGRU(x_val_gru, y_val_gru, normalizer_gru)
test_dataset_gru = RainfallDatasetGRU(x_test_gru, y_test_gru, normalizer_gru)
```

```
train_loader_gru = DataLoader(train_dataset_gru, batch_size=1024, shuffle=Tru
e, pin_memory=True)
val_loader_gru = DataLoader(val_dataset_gru, batch_size=1024, shuffle=False, p
in_memory=True)
test_loader_gru = DataLoader(test_dataset_gru, batch_size=1024, shuffle=False,
pin_memory=True)
```

```
(229548, 5, 75)
(229548, 5)
(92839, 5, 75)
(92839, 5)
(111715, 5, 75)
(111715, 5)
```

In [56]:

```
##############################################################################
##
# 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):
        super(GRU, self).__init__()
        self.gru = nn.GRU(3 * 5 * 5, hidden_size, 5)
        self.ff1 = nn.Linear(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, 1)

    def forward(self, x):
        gru = self.gru(x)
        hd1 = F.relu(self.ff1(gru[0]))
        y = self.out(hd1)
        return y
```

In [57]:

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

In [20]:

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

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

# Log parameters and gradients
wandb.watch(model_gru, log='all')

for epoch in range(config_gru['epochs']):  # loop over the dataset multiple ti
mes

    # Training
    train_loss = []
    current_lr = optimizer_gru.param_groups[0]['lr']
    learning_rates_gru.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
        y_pred = torch.reshape(y_pred, (-1, 5))
```

```
        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_gru.append(avg_train_loss)

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

    # Validation
    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
            y_pred = torch.reshape(y_pred, (-1, 5))
            loss = loss_fn(y_pred, y_true)

            # Log stuff
            val_loss.append(loss)

        avg_val_loss = torch.stack(val_loss).mean().item()
        val_losses_gru.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_gru[:-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.finish()
print('Finished Training')
```

```
start training gru
```

```
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: demonstem. Use `wandb login --relogin` to forc
e relogin
```

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_162959-vym8rb1y`

Syncing run **curious-snowball-1 (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/vym8rb1y)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-gru) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-gru (https://wandb.ai/demonstem/precipitation-nowcasting-gru)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/vym8rb1y (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/vym8rb1y)

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


Epoch 1 train loss: 1.9387
Validating epoch 1


Epoch 1 val loss: 1.6765
Saving new best model..
Training epoch 2...
Current LR: 0.01


Epoch 2 train loss: 1.9148
Validating epoch 2


Epoch 2 val loss: 1.6743
Saving new best model..
Training epoch 3...
Current LR: 0.01


Epoch 3 train loss: 1.9165
Validating epoch 3


Epoch 3 val loss: 1.6741
Saving new best model..
Training epoch 4...
Current LR: 0.01


Epoch 4 train loss: 1.9227
Validating epoch 4
```

```
Epoch 4 val loss: 1.6757
Training epoch 5...
Current LR: 0.01

Epoch 5 train loss: 1.9193
Validating epoch 5

Epoch 5 val loss: 1.6752
Training epoch 6...
Current LR: 0.01

Epoch 6 train loss: 1.9192
Validating epoch 6

Epoch 6 val loss: 1.6731
Saving new best model..
Training epoch 7...
Current LR: 0.01

Epoch 7 train loss: 1.9178
Validating epoch 7

Epoch 7 val loss: 1.6730
Saving new best model..
Training epoch 8...
Current LR: 0.01

Epoch 8 train loss: 1.9172
Validating epoch 8

Epoch 8 val loss: 1.6757
Training epoch 9...
Current LR: 0.01

Epoch 9 train loss: 1.9148
Validating epoch 9

Epoch 9 val loss: 1.6742
Training epoch 10...
Current LR: 0.002

Epoch 10 train loss: 1.9313
Validating epoch 10

Epoch 10 val loss: 1.6741
```

Run history:

Run summary:

lr	0.002
train_loss	1.9313
val_loss	1.67414

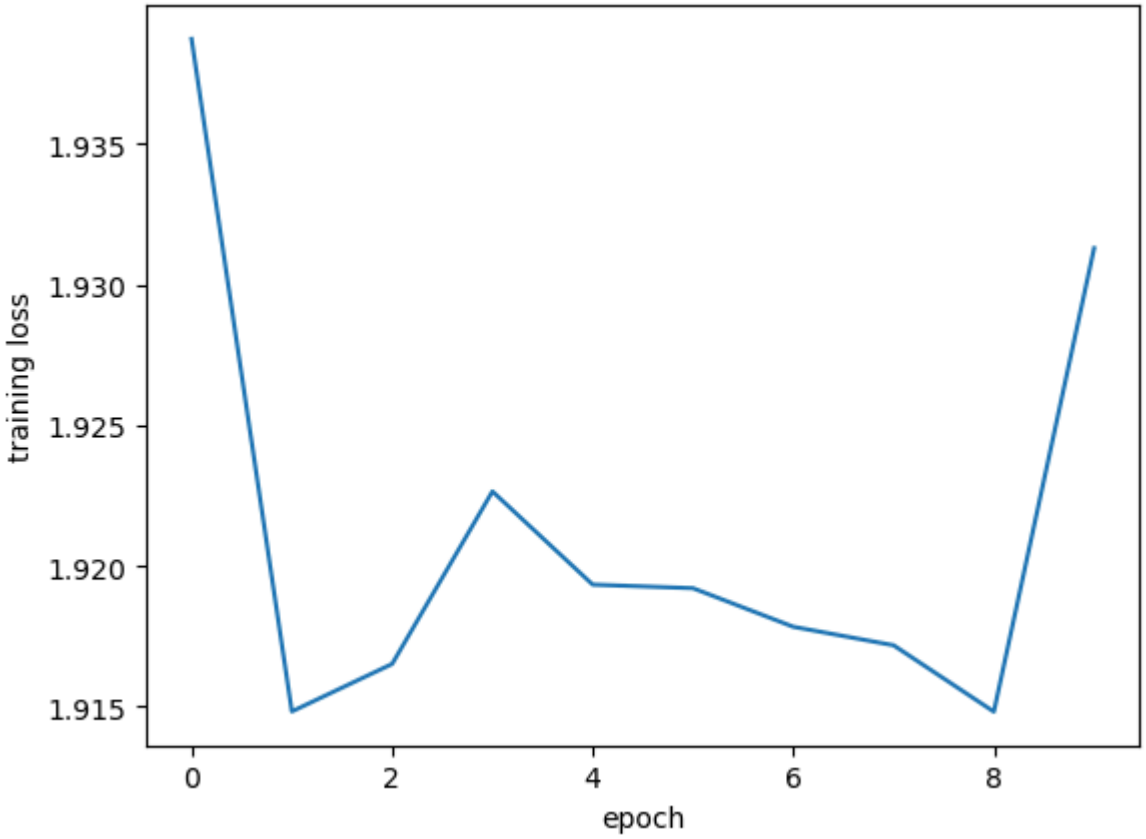
View run **curious-snowball-1** at: https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/vym8rb1y (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/vym8rb1y)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

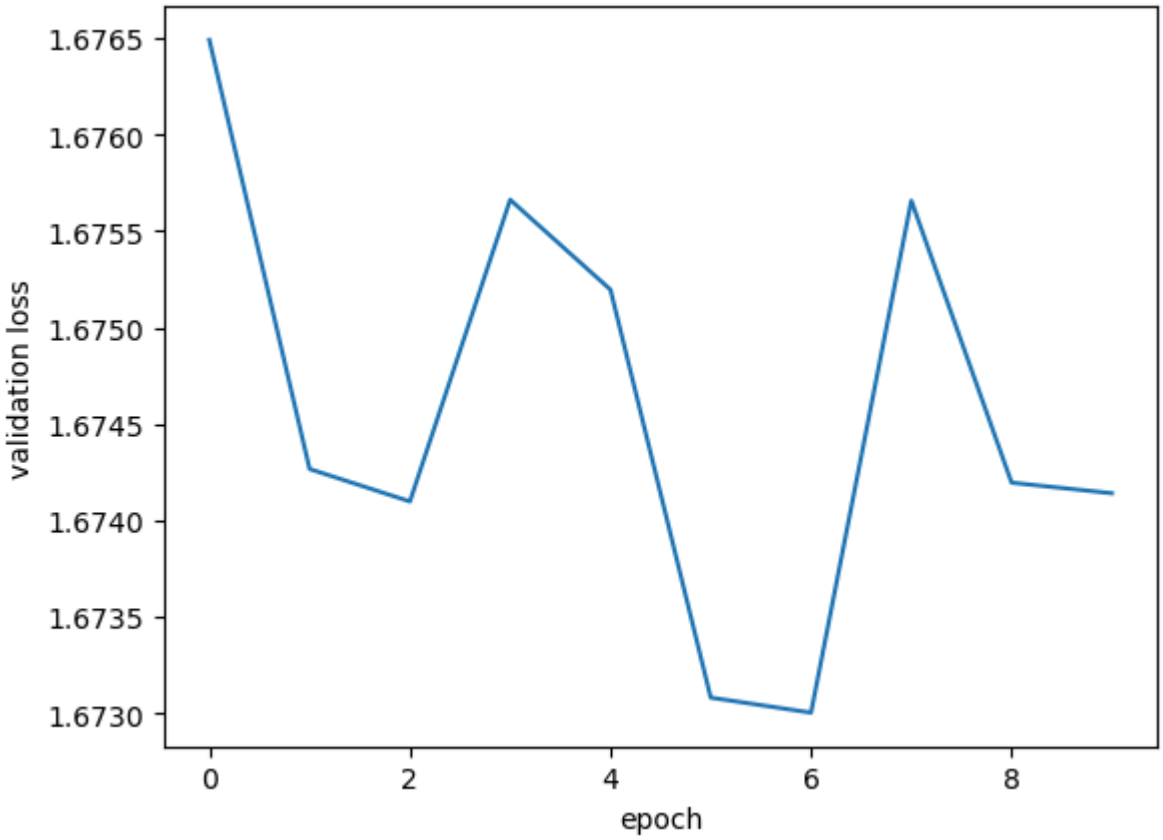
Find logs at: `.\wandb\run-20240320_162959-vym8rb1y\logs`

```
Finished Training
```

In [21]:

```
# Plot
plt.plot(train_losses_gru)
plt.xlabel('epoch')
plt.ylabel('training loss')
plt.show()
plt.plot(val_losses_gru)
plt.xlabel('epoch')
plt.ylabel('validation loss')
plt.show()
```


training loss
1.935
1.930
1.925
1.920
1.915
0
2
4
6
8
epoch


validation loss
1.6765
1.6760
1.6755
1.6750
1.6745
1.6740
1.6735
1.6730
0
2
4
6
8
epoch

In [16]:
```
# Evaluate
def evaluate_gru(data_loader, model):
    """
    Evaluate model on validation data given by data_loader
    """
    # write code here
    # Start wandb run
    wandb.init(
        project='precipitation-nowcasting-gru',
    )

    # Log parameters and gradients
    wandb.watch(model, log='all')
    with torch.no_grad():
      train_losses = []
      for i, (inputs, y_true) in enumerate(tqdm(data_loader)):
        # Transfer data from cpu to gpu
        inputs = inputs.to(device)
        y_true = y_true.to(device)

        # Predict
        y_pred = model(inputs)

        # Calculate loss
        y_pred = torch.reshape(y_pred, (-1, 5))
        loss = loss_fn(y_pred, y_true)

        # Log stuff
        train_losses.append(loss)

      mse = torch.stack(train_losses).mean()
      wandb.finish()
      return mse
```

In [22]:
```
checkpoint = torch.load('model_gru.pth.tar')
loaded_model = GRU(hidden_size=config_gru['hidden_size']) # Create model objec
t
loaded_model.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
model_gru = loaded_model
model_gru.to(device)
```

```
Loaded epoch 6 model
```

Out[22]:
```
GRU(
  (gru): GRU(75, 200, num_layers=5)
  (ff1): Linear(in_features=200, out_features=200, bias=True)
  (out): Linear(in_features=200, out_features=1, bias=True)
)
```

In [25]:
```
evaluated_train_gru = evaluate_gru(train_loader_gru, model_gru).item()
evaluated_val_gru = evaluate_gru(val_loader_gru, model_gru).item()

print('gru-model')
print('train', evaluated_train_gru)
print('validate', evaluated_val_gru)
```

```
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: demonstem. Use `wandb login --relogin` to forc
e relogin
```

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_170013-l1822t8v`

Syncing run **comic-dream-3 (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/l1822t8v)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-gru) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-gru (https://wandb.ai/demonstem/precipitation-nowcasting-gru)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/l1822t8v (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/l1822t8v)

View run **comic-dream-3** at: https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/l1822t8v (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/l1822t8v)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_170013-l1822t8v\logs`

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_170140-cv0m1ltu`

Syncing run **twilight-field-4 (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/cv0m1ltu)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-gru) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-gru (https://wandb.ai/demonstem/precipitation-nowcasting-gru)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/cv0m1ltu (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/cv0m1ltu)

View run **twilight-field-4** at: https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/cv0m1ltu (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/cv0m1ltu)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_170140-cv0m1ltu\logs`

```
gru-model
train 1.9142119884490967
validate 1.673003077507019
```

Transformer

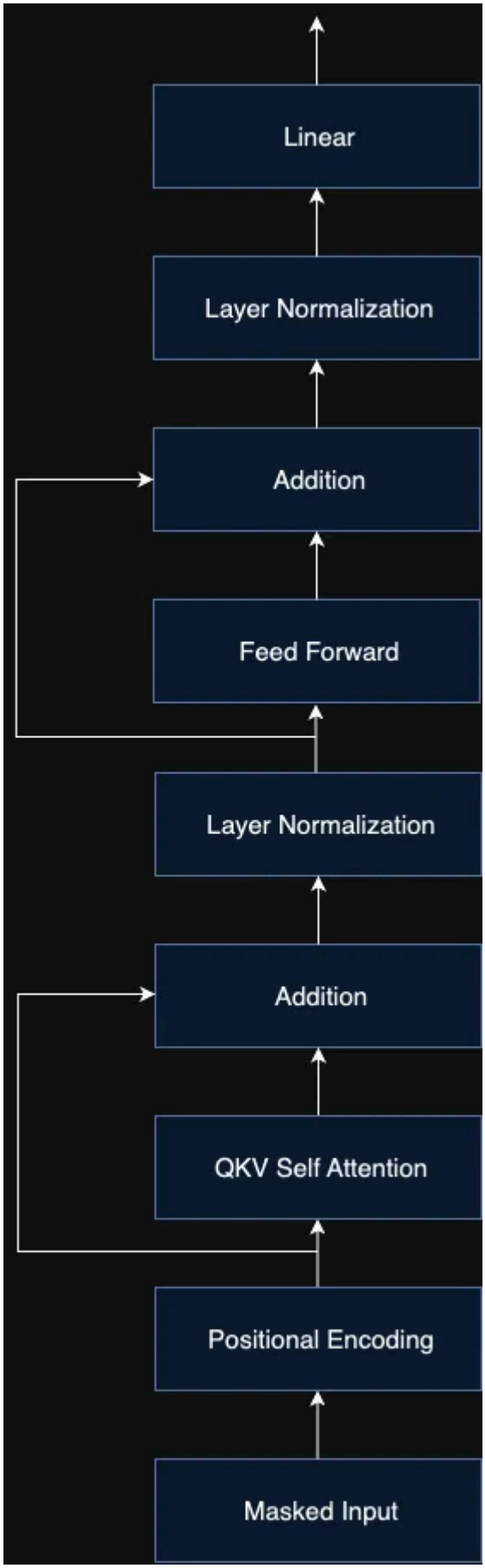
Welcome to the beginning of the real world! The aboved models are not usually used in practice due to its limited capability. Transformers are generally used by computer vision, natural language processing, and speech processing (almost every big AI fields).

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.

In [ ]:
```
##############################################################################
##
# 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
#
##############################################################################
##
```

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.


Linear
Layer Normalization
Addition
Feed Forward
Layer Normalization
Addition
QKV Self Attention
Positional Encoding
Masked Input

```
==========================================================================================
=======
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
==========================================================================================
=======
```

In [ ]:

```
#################################################################################
##
# 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):
    pass

  def forward(self, x):
    pass
```

In [ ]:
```
################################################################################
##
# 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):
    pass

  def forward(self, x):
    pass

class TransformerModel(nn.Module):
  def __init__(self):
    pass

  def forward(self, x):
    pass
```

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

In [ ]:
```
# Plot
```

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

In [ ]:
```
# Evaluate
```

Final Section

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.

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

In [109]:

```
################################################################################
##
# 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 Fancy(nn.Module):
    def __init__(self, hidden_size=200):
        super(Fancy, self).__init__()
        self.gru = nn.GRU(3 * 5 * 5, hidden_size, 5)
        self.ff1 = nn.Linear(hidden_size, hidden_size)
        self.ff2 = nn.Linear(hidden_size, hidden_size)
        self.ff3 = nn.Linear(hidden_size, hidden_size)
        self.dropout = nn.Dropout(0.5)
        self.out = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x = self.gru(x)
        x = F.relu(self.ff1(x[0]))
        x = self.dropout(x)
        x = F.relu(self.ff2(x))
        x = self.dropout(x)
        x = F.relu(self.ff3(x))
        y = self.out(x)
        return y
```

In [110]:

```
fancy_config = {
    'architecture': 'gru',
    'lr': 0.01,
    'hidden_size': 100,
    'scheduler_factor': 0.2,
    'scheduler_patience': 2,
    'scheduler_min_lr': 1e-4,
    'epochs': 5
}

# Model
model_fancy = Fancy(hidden_size=fancy_config['hidden_size'])
model_fancy = model_fancy.to(device)
optimizer_fancy = torch.optim.Adam(model_fancy.parameters(), lr=fancy_config
['lr'])
scheduler_fancy = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer_fancy,
    'min',
    factor=fancy_config['scheduler_factor'],
    patience=fancy_config['scheduler_patience'],
    min_lr=fancy_config['scheduler_min_lr']
)
```

In [ ]:

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


# Training
train_losses_fancy = []
val_losses_fancy = []
learning_rates_fancy = []

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

# Log parameters and gradients
wandb.watch(model_fancy, log='all')

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

    # Training
    train_loss = []
    current_lr = optimizer_fancy.param_groups[0]['lr']
    learning_rates_fancy.append(current_lr)

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

        # Predict
        y_pred = model_fancy(inputs)
```

```
        # Calculate loss
        y_pred = torch.reshape(y_pred, (-1, 5))
        loss = loss_fn(y_pred, y_true)

        # Compute gradient
        loss.backward()

        # Update parameters
        optimizer_fancy.step()

        # Log stuff
        train_loss.append(loss)

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

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

    # Validation
    model_fancy.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_fancy(inputs)

            # Calculate loss
            y_pred = torch.reshape(y_pred, (-1, 5))
            loss = loss_fn(y_pred, y_true)

            # Log stuff
            val_loss.append(loss)

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

        # LR adjustment with scheduler
        scheduler_fancy.step(epoch)

        # Save checkpoint if val_loss is the best we got
        best_val_loss = np.inf if epoch == 0 else min(val_losses_fancy[:-1])
        if avg_val_loss < best_val_loss:
            # Save whatever you want
            state = {
                'epoch': epoch,
                'model': model_fancy.state_dict(),
                'optimizer': optimizer_fancy.state_dict(),
                'scheduler': scheduler_fancy.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_fancy4.pth.tar')

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

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

In [111]:
```
checkpoint = torch.load('model_fancy2.pth.tar')
loaded_model = Fancy(hidden_size=fancy_config['hidden_size']) # Create model object
loaded_model.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
model_fancy = loaded_model
model_fancy.to(device)
```

```
Loaded epoch 0 model
```

Out[111]:
```
Fancy(
  (gru): GRU(75, 100, num_layers=5)
  (ff1): Linear(in_features=100, out_features=100, bias=True)
  (ff2): Linear(in_features=100, out_features=100, bias=True)
  (ff3): Linear(in_features=100, out_features=100, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (out): Linear(in_features=100, out_features=1, bias=True)
)
```

In [112]:
```
# Evaluate best model on validation and test set
evaluated_test_fancy = evaluate_gru(test_loader_gru, model_fancy).item()
print('my-best-model', evaluated_test_fancy)
```

Finishing last run (ID:cl5ibdo8) before initializing another...

View run **silvery-leaf-12** at: https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/cl5ibdo8 (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/cl5ibdo8)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_232633-cl5ibdo8\logs`

Successfully finished last run (ID:cl5ibdo8). Initializing new run:

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_232934-sbzd3n4g`

Syncing run **atomic-night-13 (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/sbzd3n4g)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-gru) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-gru (https://wandb.ai/demonstem/precipitation-nowcasting-gru)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/sbzd3n4g (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/sbzd3n4g)

View run **atomic-night-13** at: https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/sbzd3n4g (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/sbzd3n4g)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_232934-sbzd3n4g\logs`

```
my-best-model 1.1568195819854736
```

In [68]:
```
# Also evaluate your fully-connected model and CNN/GRU/Transformer model on th
e test set.
checkpoint = torch.load('model_cnn.pth.tar')
loaded_model = CNN(hidden_size=config_cnn['hidden_size']) # Create model objec
t
loaded_model.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
model_cnn = loaded_model
model_cnn.to(device)

checkpoint = torch.load('model_gru.pth.tar')
loaded_model = GRU(hidden_size=config['hidden_size']) # Create model object
loaded_model.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
model_gru = loaded_model
model_gru.to(device)

checkpoint = torch.load('model_ff.pth.tar')
loaded_model = FeedForwardNN(hidden_size=config_gru['hidden_size']) # Create m
odel object
loaded_model.load_state_dict(checkpoint['model']) # Load weights
print(f"Loaded epoch {checkpoint['epoch']} model")
model_ff = loaded_model
model_ff.to(device)
```

```
Loaded epoch 2 model
Loaded epoch 6 model
Loaded epoch 4 model
```

Out[68]:
```
FeedForwardNN(
  (ff1): Linear(in_features=75, out_features=200, bias=True)
  (ff2): Linear(in_features=200, out_features=200, bias=True)
  (ff3): Linear(in_features=200, out_features=200, bias=True)
  (out): Linear(in_features=200, out_features=1, bias=True)
)
```

In [69]:
```
evaluated_test_cnn = evaluate_cnn(test_loader, model_cnn).item()
print('cnn-model', evaluated_test_cnn)
```

Finishing last run (ID:wxbzbqxb) before initializing another...

View run **glad-aardvark-9** at: https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/wxbzbqxb (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/wxbzbqxb)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_213133-wxbzbqxb\logs`

Successfully finished last run (ID:wxbzbqxb). Initializing new run:

Tracking run with wandb version 0.16.4

Run data is saved locally in `c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_213329-wmx4g350`

Syncing run **major-armadillo-10 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/wmx4g350)** to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-cnn) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-cnn (https://wandb.ai/demonstem/precipitation-nowcasting-cnn)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/wmx4g350 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/wmx4g350)

View run **major-armadillo-10** at: https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/wmx4g350 (https://wandb.ai/demonstem/precipitation-nowcasting-cnn/runs/wmx4g350)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20240320_213329-wmx4g350\logs`

```
cnn-model 1.161057710647583
```

In [72]:
```
evaluated_test_ff = evaluate(test_loader, model_ff).item()
print('ff-model', evaluated_test_ff)
```

```
ff-model 1.1636040210723877
```

In [74]:
```
evaluated_test_gru = evaluate_gru(test_loader_gru, model_gru).item()
print('gru-model', evaluated_test_gru)
```

```
Finishing last run (ID:epualq7v) before initializing another...

View run exalted-surf-5 at: https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/epualq7v (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/epualq7v)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: .\wandb\run-20240320_213851-epualq7v\logs

Successfully finished last run (ID:epualq7v). Initializing new run:

Tracking run with wandb version 0.16.4

Run data is saved locally in c:\Users\Tonza\Desktop\Code\Pattern Recognition\HW5\wandb\run-20240320_214042-1iooyesd

Syncing run different-plasma-6 (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/1iooyesd) to Weights & Biases (https://wandb.ai/demonstem/precipitation-nowcasting-gru) (docs (https://wandb.me/run))

View project at https://wandb.ai/demonstem/precipitation-nowcasting-gru (https://wandb.ai/demonstem/precipitation-nowcasting-gru)

View run at https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/1iooyesd (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/1iooyesd)

View run different-plasma-6 at: https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/1iooyesd (https://wandb.ai/demonstem/precipitation-nowcasting-gru/runs/1iooyesd)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: .\wandb\run-20240320_214042-1iooyesd\logs

gru-model 1.1548452377319336
```

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

TODO#22

Explain what helped and what did not help here

Ans: add more hidden layer does not help much, what I did are adding dropouts, add more layers and decrease lr, and epoch for gru model. But got 1.156 > 1.154(old gru), therefore adding dropouts in this model does not help much either.

[Optional] Augmentation using data loader

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.

In [ ]:
```
# Write Dataset/DataLoader with noise here
```

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

In [ ]:
```
# Evaluate the best model trained with noise on validation and test set
```