CS 5600/6600: F23: Intelligent Systems
Project 1

Vladimir Kulyukin
Department of Computer Science
Utah State University

October 11, 2023

Introduction

This project has three objectives. The first objective is for you to design, train, evaluate ANN, ConvNet, and LSTM models to predict time series. The second objective is for you to train and test ResNet50 and YOLO, two very popular ConvNet models, on an image dataset. The third objective is to practice technical writing by documenting your results. The time series part will be done with Keras. The image classification – with PyTorch, which will give you an opportunity to work with two popular machine learning libraries.

Time Series

A time series is a sequence of 2-tuples $(t_1, o_1), (t_2, o_2), \ldots, (t_n, o_n)$, where t_i, $1 \leq i \leq n$, is some measurement of time and o_i is some observation recorded at t_i. There are two common things that an analyst does with a time series: description and prediction. Description involves computing means, medians, distributions, plots, etc. Prediction invloves building models to predict the future on the assumption that it will be like the past. The main objective of descriptive analysis is to develop models of feasible descriptions of the sample data. If CS5600/6600 were a statistics course, we would spend a lot of time on description. But, since it is a CS course on intelligent systems, we'll focus on predictive modeling. In some technical literature, predictive modeling is called *time series forecasting*. I prefer *predictive modeling*, because it sounds less pompous to me.

The value of a predictive model is how well it can predict future observations and how far into the future it can predict. There are, generally speaking, two types of predictive modeling – *uniseries* and *multiseries*. The former focuses on predicting values of one time series and the latter – on predicting values of some series from other series. In the simplest case of multiseries prediction, we are predicting the values of one series from another series. Of course, a fundamental, frequently implict, assumption in multiseries prediction is that the series are related.

There are several issues that the analyst must consider. First, the amount of data. The more is not always better, especially if the data are not curated. Synthentic augmentation is fine, but is no substitute for real data. The amount of data determines the scope of the analysis. Second, the time horizon of the precition. How far into the future does the model need to predict? Shorter time horizons are typically easier and more reliable. There is a tradeoff between the length of the horizon and accuracy. Third, are prediction updates possible? If new data become availabe, can the model be retrained to update its predictions? This is useful, but leads dynamic changeable accuracies. Fourth, does the model need to predict at the original sampling frequency or is it possible to change the frequency? For example, if the sampling is done every 5 minutes, can we change the model to predict the value at the end of the next hour? Fifth, can we pre-process the data? For example, can we remove outliers?

Uniseries Precition: A Case Study with $f(x) = x + 1$

Let's start with a lab on uniseries prediction. You don't have to turn anything in for this part of the project. The souce code is in `cs5600_6600_f23_project_1_time_series.py`. If you do not have Keras installed on your computer, you can follow the instructions on this site to get it installed. We will also need matplotlib installed. In short, make sure that the following imports work on your computer.

```
import numpy as np
import math
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import LSTM
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
import matplotlib.pyplot as plt
```

Let's consider $f(x) = x + 1$, where x is a natural number, i.e., a non-negative integer. Recall that we modeled this function in a previous assignment with a neural network. In computability theory, this function is called *successor*, because it can successfully generate all natural numbers from 0. The successor function can be treated as a time series so that a time 0, the observation is f(0), at time 1, the observation is f(1), etc.

Let's define the function and a simple plotting utility. The plotting utility will also us to plot the ground truth, i.e., `y`, and our model's predictions, i.e., `yhat`, on the same plot.

```
def successor(x):
    return x + 1

def plot_preds(plot_title, xlabel, ylabel, y, yhat):
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.plot(y, 'r--')
    plt.plot(yhat, 'bs--')
    plt.legend(['y', 'yhat'], loc='best')
    plt.title(plot_title)
    plt.show()
```

To train a predictive model we must define the input and output sequences. In this case, the input sequence consists of non-negative integers and the output sequence is the values of the successor function for the corresponding elements of the input sequence. In the code segment below, the input sequence starts at 0 and ends at 99. Both the input and output sequences are converted to numpy arrays and reshaped to have a row-column structure needed for Keras network models. The re-shaped numpy arrays are stacked to form a dataset, another typical pattern for Keras models.

```
    a = [i for i in range(100)]
    b = [i+1 for i in a]
    in_seq  = np.array(a)
    out_seq = np.array(b)

    # convert to rows-columns structure
    in_seq  = in_seq.reshape((len(in_seq), 1))
    out_seq = out_seq.reshape((len(out_seq), 1))

    # stack columns horizontally
    dataset = np.hstack((in_seq, out_seq))
```

The dataset must be partitioned into training samples. This is where we have to decide how many values of the input sequence the model will need to see to predict the next output value. This number of values of the input sequence the model needs to see is called the *number of steps*. Below is a function that takes the stacked dataset and the number of steps and returns two numpy arrays `X` and `y`. The first contains the input sequence partitioned into segments, each of which has the length equal to the number of steps. The values in `y` are the values of the successor function that we want to compute from a given sample of steps. The partition function is implemented in aux_funs.py.

```
from aux_funs import partition_dataset_into_samples
def partition_dataset_into_samples(dataset, num_steps):
    X, y = [], []
    for i in range(len(dataset)):
        end_ix = i + num_steps
        if end_ix > len(dataset):
           break
        seq_x, seq_y = dataset[i:end_ix, :-1], dataset[end_ix-1, -1]
        X.append(seq_x)
        y.append(seq_y)
    return np.array(X), np.array(y)
```

Let's assume that `num_steps=5` in the following code segment.

```
    X, y = partition_dataset_into_samples(dataset, num_steps)
    num_features = X.shape[2]
```

Then the first three values of `X` are

```
array([[0],
       [1],
       [2],
       [3],
       [4]])
array([[1],
       [2],
       [3],
       [4],
       [5]])
array([[2],
       [3],
       [4],
       [5],
       [6]]).
```

The first three values of `y` are

```
5
6
7.
```

In other words, we want our model (whatever it is), on seeing the sample $(0, 1, 2, 3, 4)$, to predict 5, on seeing the sample $(1, 2, 3, 4, 5)$, to predict 6, and to predict 7 from the sample $(2, 3, 4, 5, 6)$.

The next component of the predictive model is called the *number of features*. This is the number of outputs that we want the model to predict. Since we're modeling the successor function, the number of features is 1. Once the dataset is partitioned into `X` and `y`, the number of features can be obtained as `X.shape[2]`.

Model 1: Artificial Neural Network

We are ready to define our first model that we will be training to predict the next value of the successor function from the previous five values. Here it is. The complete source code for this model is in `test_ann_successor()` in `cs5600_6600_f23_project_1_time_series.py`.

```
    model = Sequential()
    model.add(Dense(5, input_shape=(num_steps, num_features),
              activation='relu'))
    model.add(Flatten())
    model.add(Dense(num_features))
    model.compile(optimizer='adam', loss='mse')
```

This is is a simple ANN model whose hidden layer one consists of 5 nodes fully connected (hence Dense) to the input layer of dimensions `num_steps` by `num_features`. In Keras models, the shape of the input is always num_steps by num_features, because we take it a sample with num_steps values and predict from it num_features values.

Then we add a flat layer that flattens the previous value into a one dimenstional array, which is a common technique in ANN building, and connect it to a fully connected output layer with the num_features nodes. Then we specify the gradient descent option as adam and the loss function as mean squred error (mse).

We are ready to train, i.e., fit, and save the model. We train the model for 2K epochs. If we set verbose to 1, then the loss values will be dislayed for every epoch. The fitted model will be saved in the current directory as ann_successor.h5. The format h5 is a zip format Keras uses to persist models. After the model is saved it can be loaded with the function load_model().

```
model.fit(X, y, epochs=2000, verbose=0)
model.save('ann_successor.h5')
loaded_model = load_model('ann_successor.h5')
```

Now we need to test the trained model on the data that it has not seen. Testing on on the near future, e.g., the next five values, is not that interesting. Many models work well on the near future. Let's test it on the data from som distant future, e.g., on the range from 1000 upto 1020. We are essentially repeating the same steps we did when we were preparing the dataset for training the model on the range from 0 upto 99. I use different variables, e.g., a2, b2, to emphasize that this is the testing stage.

```
a2 = [i for i in range(1000, 1021)]
b2 = [i+1 for i in a2]
in_seq2  = np.array(a2)
out_seq2 = np.array(b2)
in_seq2  = in_seq2.reshape((len(in_seq2), 1))
out_seq2 = out_seq2.reshape((len(out_seq2), 1))
dataset2 = np.hstack((in_seq2, out_seq2))
X2, y2 = partition_dataset_into_samples(dataset2, num_steps)
num_features2 = X2.shape[2]
```

Now we can test the loaded trained model, print the mean squared error, and plot the predictions and ground truths. The plot for this test is in Figure 1.

```
loaded_model = load_model('ann_successor.h5')
ground_truth = []
preds        = []
for i in range(len(X2)):
    x_input_2 = X2[i].reshape((1, num_steps, num_features))
    y_hat_2 = loaded_model.predict(x_input_2)
    preds.append(y_hat_2[0][0])
    ground_truth.append(y2[i])
mse = (np.array(ground_truth) - np.array(preds))**2
mse = np.mean(mse)
print(mse)
plot_preds('ANN: f(x)=x+1; mse={}'.format(mse), 'x',
           'y/yhat', ground_truth, preds)
```

If you're new to builing predictive models for time series, congratulations on building your first predictor. The logical steps of building predictors are always the same: 1) decide on the number of steps and features; 2) prepare the dataset; 3) construct a mode; 4) train the model; 5) test the model.

Model 2: Convolutional Network

Let's now train and test a simple convolutional network model on the same data. Here's the model. The complete source code is in test_convnet_successor(). The preparation of the train and test datasets is the same as for the ANN model in the previous section.

```
model = Sequential()
model.add(Conv1D(filters=5, kernel_size=2, activation='relu',
```

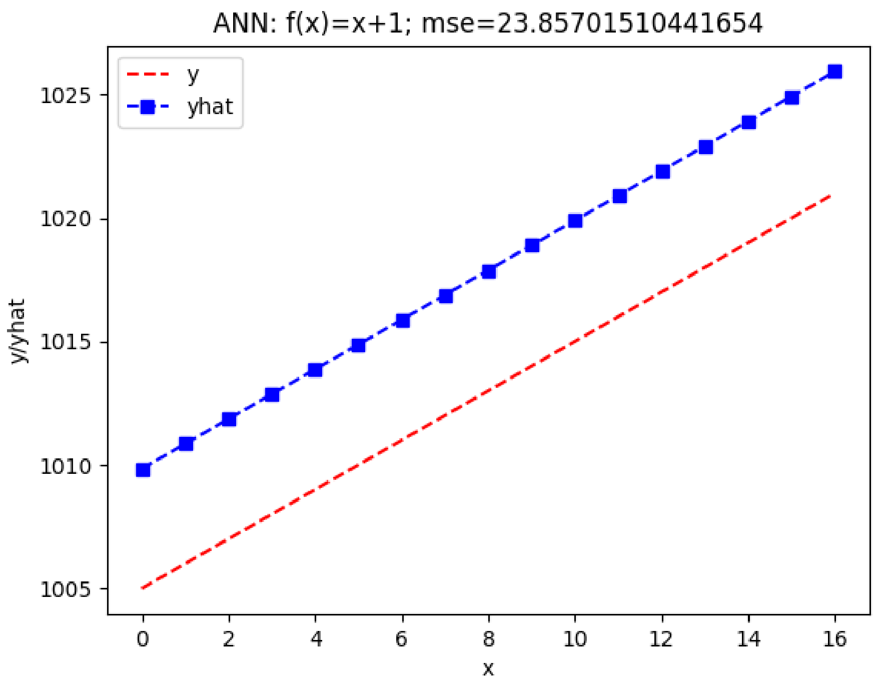

Figure 1: Ground truths and predictions for ann_successor.h5 trained on the interval [0, 99] and tested the interval [1000, 1020].

```
                 input_shape=(num_steps, num_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(10, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Figure 2 shows the performance of this network on a distant future.

Model 3: Long Short-Term Memory Network

Let's try the LSTM model. The source code is in `test_lstm_successor()`.

```
model = Sequential()
model.add(LSTM(10, activation='relu',
               input_shape=(num_steps, num_features)))
model.add(Dense(num_features))
model.compile(optimizer='adam', loss='mse')
```

Figure 3 shows the LSTM network on a distant future.

It looks like the ConvNet model performed best in terms of mean squared error (MSE). The ANN model was second best. The LSTM model was worst. One caveat is that I trained each model only once. Repeated training of each model typically improves results for all models.

Multiseries Prediction: Predicting $2sin(x)$ from $sin(x)$

Let's now use the same three models, i.e., ANN, ConvNet, and LSTM, in multiseries prediction. We will attempt to predict $2sin(x)$ from $sin(x)$. We can follow the same steps to prepare our dataset.

```
a = [math.sin(i) for i in range(100)]
```

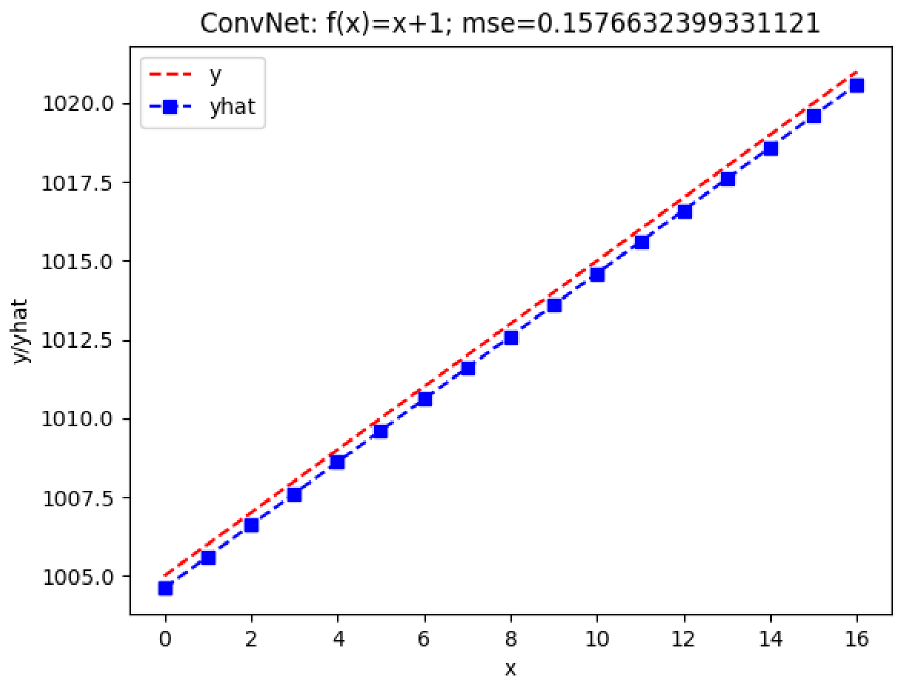

Figure 2: Ground truths and predictions for convnet_successor.h5 trained on the interval [0, 99] and tested the interval [1000, 1020].

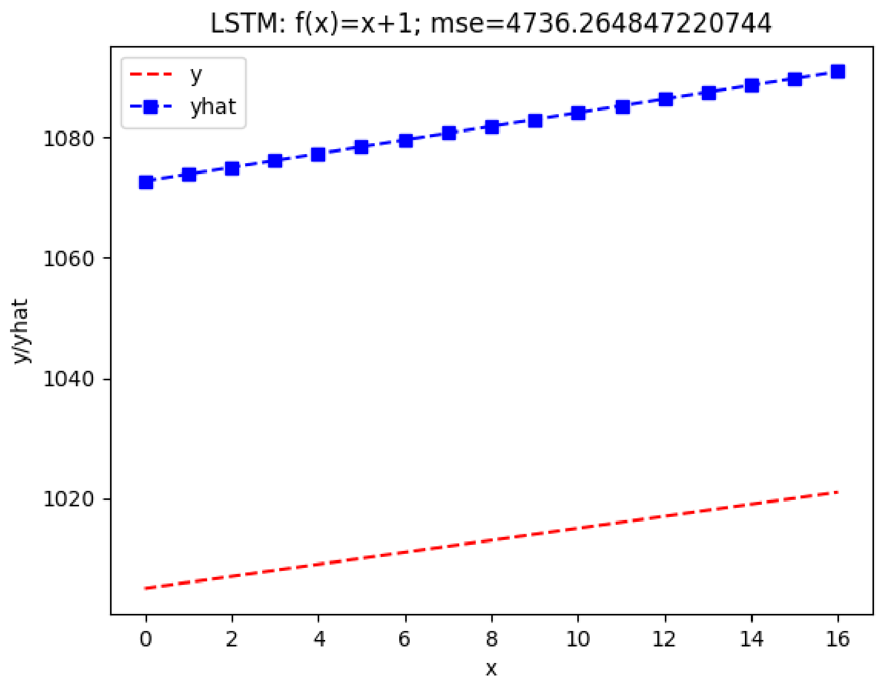

Figure 3: Ground truths and predictions for lstm_successor.h5 trained on the interval [0, 99] and tested the interval [1000, 1020].

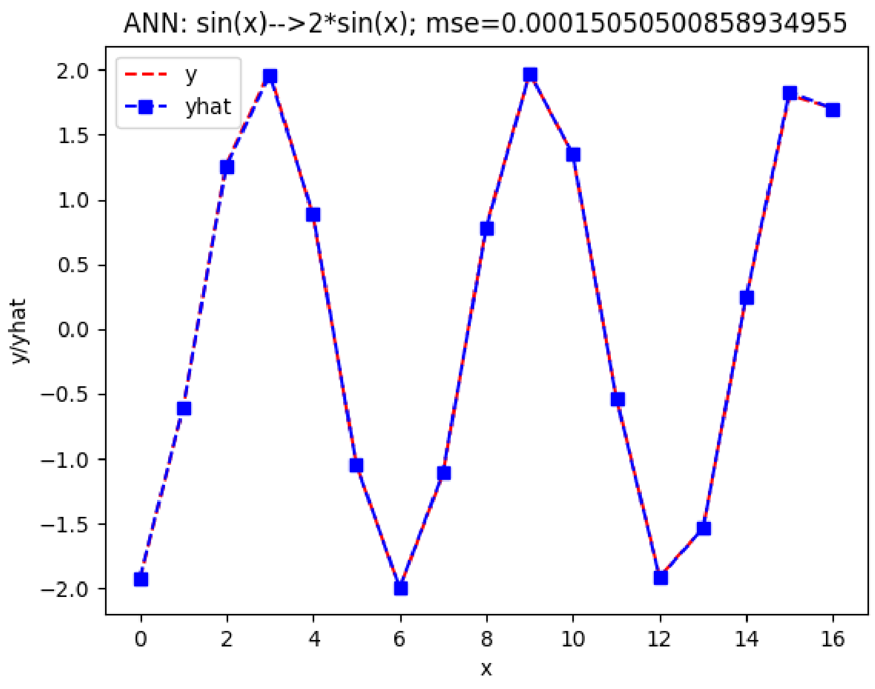

Figure 4: Ground truths and predictions for ann_2sinx.h5 trained on the interval [0, 99] and tested the interval [1000, 1020].

```
b = [2*i for i in a]
in_seq  = np.array(a)
out_seq = np.array(b)
in_seq  = in_seq.reshape((len(in_seq), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
dataset = np.hstack((in_seq, out_seq))
num_steps = 5
X, y = partition_dataset_into_samples(dataset, num_steps)
```

In the training dataset, we are predicting $2sin(x)$ from $sin(x)$, where $0 \leq x \leq 99$. I coded the functions `test_ann_2sinx()`, `test_convnet_2sinx()`, `test_lstm_2sinx()` to trained the same three models.

Let's test each model on a distant future. We will attempt to predict $2sin(x)$ from $sin(x)$, where $1000 \leq x \leq 1020$. Figures 4, 5, 6 show the performance of these nets on this future. It looks like all three models did well in terms of mse.

Project 1: Part 1: Time Series Prediction

The directory `periods_p1_p2_p3_p4_p5` contain time-aligned temperature and weight measurements for 10 honey bee hives at a USDA research apiary in Tucson, AZ collected in 2022. For this project, we will focus only on hive 2059. If you look at the names of the csv files in the 2059 directory, you will notice abbreviations such as P1, P2, P3, P4, and P5. These are the names of the time periods into which each 24-hour period is partitioned using the 24-hour time notation.

Period	Start (hour:min)	End (hour:min)
P1	06:00	11:55
P2	12:00	15:55
P3	16:00	19:55
P4	20:00	23:55
P5	00:00	05:55

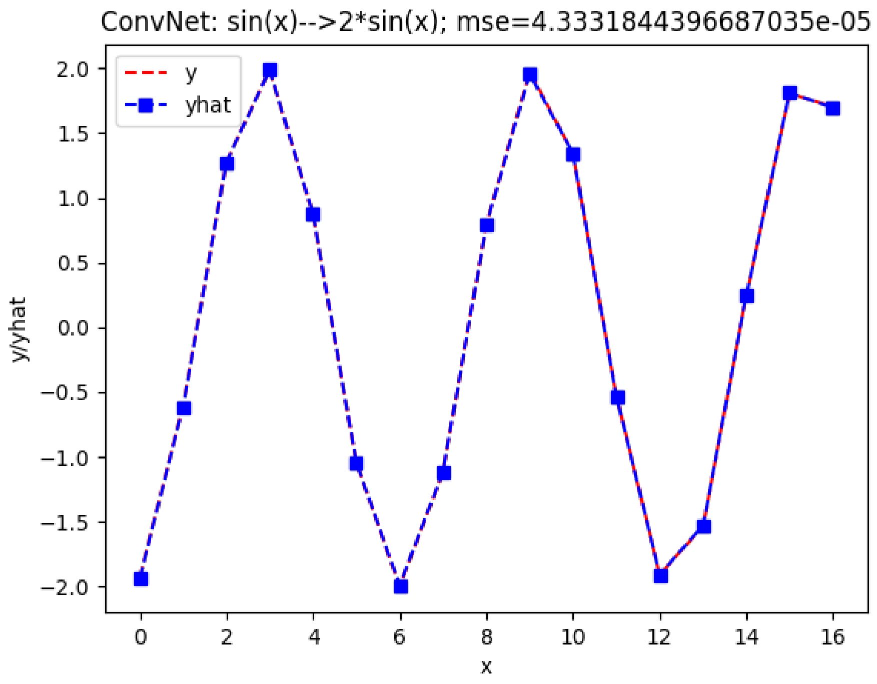

Figure 5: Ground truths and predictions for convnet_2sinx.h5 trained on the interval [0, 99] and tested the interval [1000, 1020].

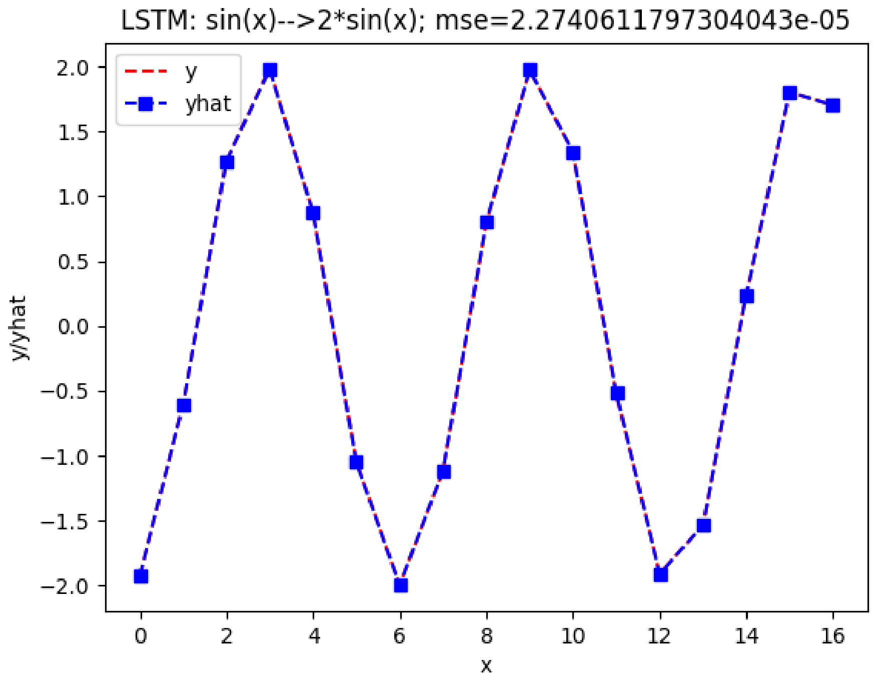

Figure 6: Ground truths and predictions for lstm_2sinx.h5 trained on the interval [0, 99] and tested the interval [1000, 1020].

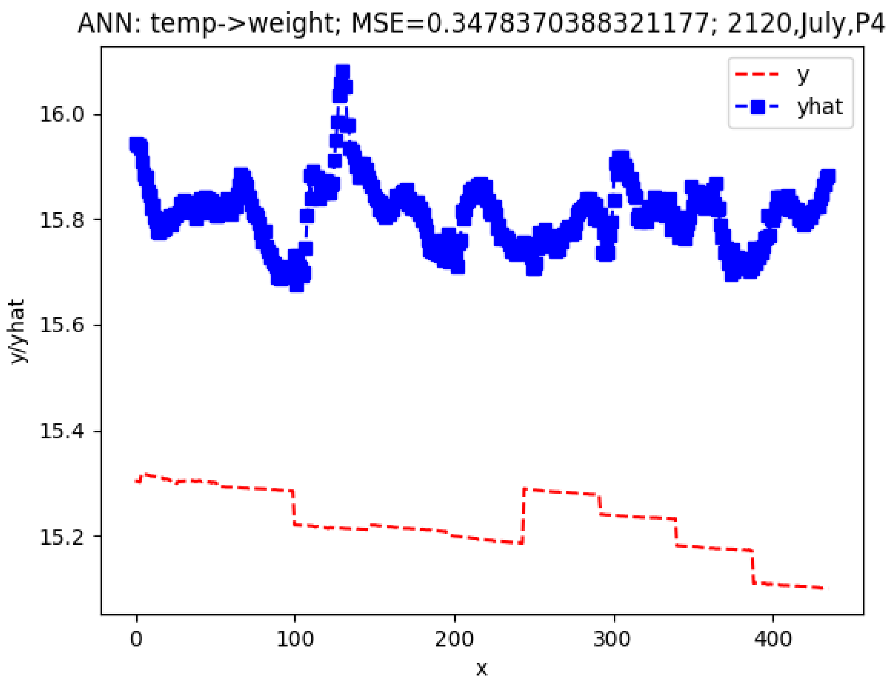

Figure 7: Ground truths and predictions for temp_weight_ann_2120.h5 for hive 2120, July, P4.

The time period in the name of each csv file is preceeded with the month, i.e., June, July, August, and September. If you open a csv file, you'll see that it's organized into 3 columns. For this project, we will focus only on the month of June and the P3 period and ignore all other months and periods.

```
Date,Temp,Weight
8/1/2022 6:0,35.18,16.1555
8/1/2022 6:5,35.22,16.1601
8/1/2022 6:10,35.22,16.1626
8/1/2022 6:15,35.18,16.1656
....
```

The objective of the time series in Part 1 of Project 1 is for you to build and submit 3 predictive models, i.e., ANN, ConvNet, LSTM, that predict the weight of hive 2059 from its internal temperature for the month of June and the time period P3.

You need to train, persist, and submit 3 models, i.e., ann_2059.h5, convnet_2059.h5, lstm_2059.h5, and your source code in cs5600_6600_proj_1_f23_temp_weight_predict.py with your model definitions. In addition to the three networks, your project report should contain three plots similar to the plots in Figures 7, 8, 9 which I generated with simple models for hive 2120, July, P4.

The only functions you should modify in cs5600_6600_proj_1_f23_temp_weight_predict.py are predict_temp_weight_ann, predict_temp_weight_convnet, predict_temp_weight_lstm. Each of these functions has the same parameters.

1. `train_temps` – the temperature series values used in training;
2. `train_weights` – the weight series values used in training;
3. `test_temps` – the temperature series values used in testing;
4. `test_weights` – the weight series values used in testing;
5. `num_steps` – number of steps in predictive model;
6. `num_epochs` – number of epochs to train predictive model;

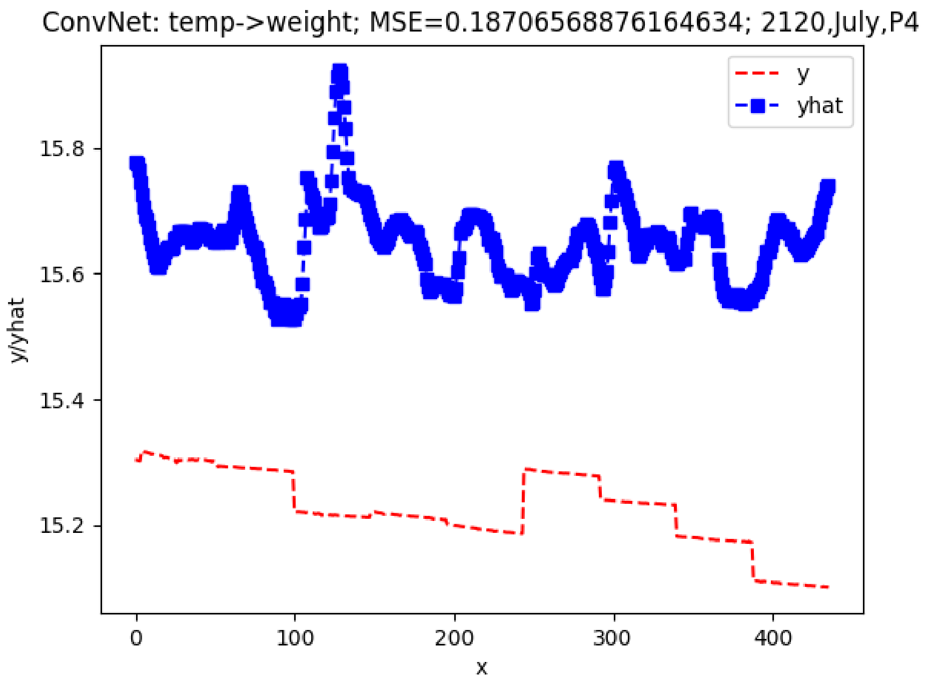

Figure 8: Ground truths and predictions for temp_weight_convnet_2120.h5 for hive 2120, July, P4.

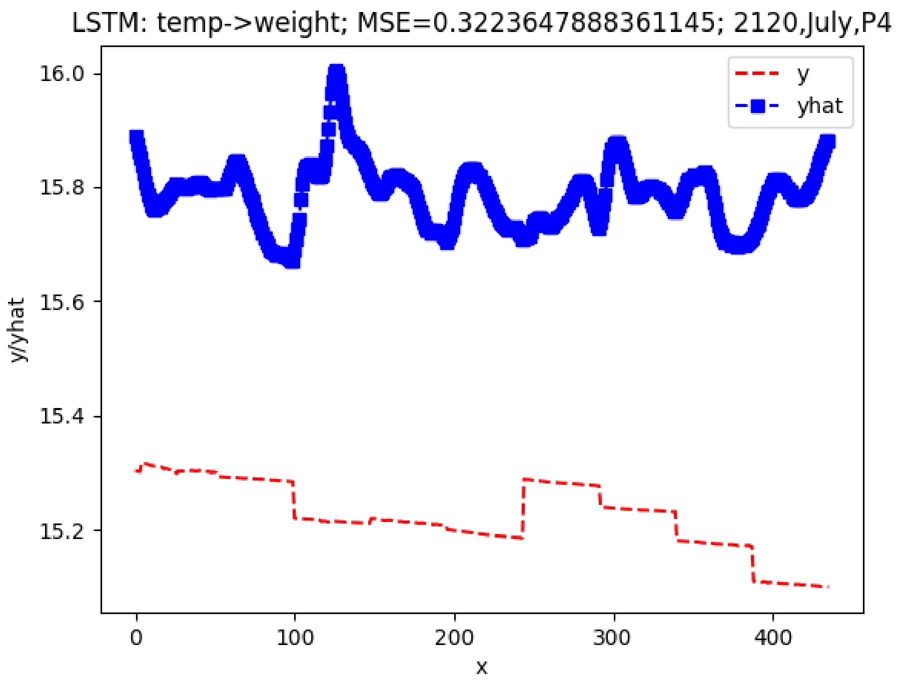

Figure 9: Ground truths and predictions for temp_weight_lstm_2120.h5 for hive 2120, July, P4.

7. `hiveid` – hive id number (should be fixed to 2059);
8. `monthid` – month id (should be fixed to `'June'`);
9. `period` – period of the day (should be fixed to `'P3'`);
10. `saved_model_name` – the file path where the trained model is persisted (e.g., `'ann_tp_wt.h5'`).

The file `aux_funs.py` contains all the csv processing functions need to get the data from csv files. So you shouldn't spend time on csv processing.

Project 2: Image Classification

We will work with the images in the BEE4 dataset. This dataset has an interesting history. In 2017, I ran my first open science crowdfunding project *BeePi: A Multisensor Electronic Beehive Monitor* on Kickstarter to crowdfund som hardware needs for my beehive monitoring research. Since this was my first crowdfunder, my target goal was a modest $1,000 to purcahse some raspberry pi computers, cameras, and USB disks. I was genuinely amazed that I got $2,940. That sum brought my project to life. In 2019, I ran another crowdfunder on Kickstarter *BeePi: Honeybees Meet AI: Stage 2* with the target goal of $5,000. I was again pleasantly surprised that 2 months later I raised $5,753, which allowed me to buy more hardware, some bee packages, and hive woodenware. In 2021, I ran a third science crowdfunding project on Kickstarter *BeePi: Honeybees Meet AI: Stage 3* with the target goal of $10,000. I was again pleasantly surprised that 2 months later I raised $12,327, which, after the Kicstarter took its small fees (7% at the time), I donated to USU to buy more hardware and to pay an hourly data manager to help me curate videos.

All BEE4 images were obtained from the data captured by on-hive video traffic monitors purchased with the Kistarter funds and deployed on beehives in Logan and North Logan, UT in 2017 – 2020. The images are labeled as bee or no-bee. If an image is labeled as bee, it has at least one bee object. Otherwise, it's labeled as no-bee. The BEE4 dataset is in the data directory in the project zip. BEE4/train directory has training images and BEE4/valid directory contains images we can use for testing. Each train and valid subdirectory, in turn, contains the subdirectories bee and nobee.

For this part of the project, you will train ResNet50 and YOLO, two popular ConvNet models, to classify images with the dataset. The files reset50_image.py and yolo_image.py some starter code you can use to start training and testing these models right away. I will not spend too much time describing these files, because they are very similar to the Python scripts we used to train LeNet on KMNIST in a previous homework.

In resnet50_image.py, you can play with loss functions, optimizers, and number of training epochs in the call to train_model function. When writing this document, I set the number of epochs to 3. You'll need to raise this number to improve your accuracy and loss.

```
model_trained = train_model(model, criterion, optimizer,
                            num_epochs=3)
```

The trained model is persisted with torch.save and then loaded and tested on a small set of the validation images to give you an example of how to create validation image sets in order to test your model.

```
model = models.resnet50(pretrained=False).to(device)
model.fc = nn.Sequential(nn.Linear(2048, 128),
                         nn.ReLU(inplace=True),
                         nn.Linear(128, 2)).to(device)
model.load_state_dict(torch.load('resnet50_bee4.h5'))

validation_img_paths = ['data/BEE4/valid/bee/32_1180_yb.png',
                        'data/BEE4/valid/bee/80_46_yb.png',
                        'data/BEE4/valid/nobee/192_168_4_5-2017-05-13_16-38-06_27_223_18.png',
                        'data/BEE4/valid/nobee/192_168_4_8-2017-05-08_15-45-28_187_261_195.png']
img_list = [Image.open(img_path) for img_path in validation_img_paths]
validation_batch = torch.stack([data_transforms['validation'](img).to(device)
                                for img in img_list])
pred_logits_tensor = model(validation_batch)
```

```
pred_probs = F.softmax(pred_logits_tensor, dim=1).cpu().data.numpy()
for i, img in enumerate(img_list):
    title_str = f'{100*pred_probs[i,0]:.0f}% Bee, {100*pred_probs[i,1]:.0f}%noBee'
    print(title_str)
```

Here's my output after 3 epochs. The loss is pretty high. So, more bulldozing is needed.

```
Epoch 1/3
----------
train loss: 0.3980, acc: 0.8199
validation loss: 0.3152, acc: 0.8630
Epoch 2/3
----------
train loss: 0.2226, acc: 0.9045
validation loss: 0.4494, acc: 0.8295
Epoch 3/3
----------
train loss: 0.2068, acc: 0.9120
validation loss: 0.3043, acc: 0.8830

100% Bee, 0%noBee
100% Bee, 0%noBee
1% Bee, 99%noBee
13% Bee, 87%noBee
------
bee: 79.62%
true: nobee
------
bee: 78.08%
true: nobee
------
bee: 82.44%
true: bee
------
bee: 83.02%
true: bee
------
bee: 79.04%
true: nobee
------
bee: 81.90%
true: bee
------
bee: 79.59%
true: nobee
------
bee: 78.36%
true: nobee
------
bee: 82.32%
true: bee
------
bee: 81.65%
true: bee
Done...
```

In yolo_image.py, we have a different way to train a network through a configuration table where all parameters are defined.

```
CONFIG_DICT = {
    'model': 'yolo_model.pth',
    'plot':  'yolo_plot.png',
    'debug': True,
    'INIT_LR': 1e-3,
    'BATCH_SIZE': 64,
    'EPOCHS': 3,
    'TRAIN_SPLIT': 0.75,
}
```

That file also contains a YOLO class that initializes a version of the YOLO archicture and has an implementation of the forward method. It then trains this YOLO network in the same way that we trained LeNet, prints a classification report and persists the model. Here's my output after training the network for 3 epochs.

```
<DBG> training YOLO...
<DBG> EPOCH: 1/3
Train loss: 0.7235, Train accuracy: 0.4730
Val loss: 0.7914, Val accuracy: 0.5000
<DBG> EPOCH: 2/3
Train loss: 0.7217, Train accuracy: 0.5097
Val loss: 0.7904, Val accuracy: 0.4900
<DBG> EPOCH: 3/3
Train loss: 0.7197, Train accuracy: 0.5404
Val loss: 0.7884, Val accuracy: 0.5140
<DBG> total time taken to train the model: 4.75s
<DBG> evaluating network...
<DBG> rand_idx:  [ 3 31 26  0  3  0  0  8  4 11]
Predicted: 0
 Actual: 0
Predicted: 0
 Actual: 0
Predicted: 0
 Actual: 0
Predicted: 0
 Actual: 0
Predicted: 0
 Actual: 0
Predicted: 0
 Actual: 1
Predicted: 0
 Actual: 1
              precision    recall  f1-score   support

         bee       0.52      0.99      0.68      1000
       nobee       0.87      0.07      0.13      1000

    accuracy                           0.53      2000
   macro avg       0.69      0.53      0.40      2000
weighted avg       0.69      0.53      0.40      2000
```

For the second part of the project, you should submit two trained PyTorch network files, i.e., resnet50_image.pth and yolo_image.pth. You should train the nets on the train images and validate them on all validation images. Include the classification reports for both networks in the project report where you have documented your results for Part 1 of this project.

What to Submit

1. Part 1: ann_2059.h5, convnet_2059.h5, lstm_2059.h5, cs5600_6600_proj_1_f23_temp_weight_predict.py with your model definitions;
2. Part 2: resnet50_image.pth, yolo_image.pth;

3. project_1_report.pdf with your documented results for Part 1 and Part 2.

Happy Hacking!