2021 Fall STATSM231A/CSM276A: Pattern Recognition and Machine Learning

Yaxuan Zhu
Department of Statistics
University of California, Los Angeles
9401 Bolter Hall., Los Angeles, California 90095
`yaxuanzhu@g.ucla.edu`

Abstract

In this course, you will learn multiple algorithms in machine learning. In general, for each homework, you are required to write the corresponding codes. We will provide a structure of code and a detailed tutorial online. Your job is (1) to deploy and understand the code. (2) Run the code and output reasonable results. (3) Make ablation study and parameter comparison. (4) Write the report with algorithms, experiment results and conclusions. For detailed instructions, please check the specification of each homework.

Homework submission forms

You have to submit both code and reports. For the code, zip everything into a single zip file. For the report, use this latex template. When submitting, no need to submit the tex file. Submit one single pdf file only.

1 Coding part: Neural Network (Fully connected Layer and CNN)

1.1 Problems

In the first part of the homework, we are tasked with loading several variations of the ResNet model, more specifically Resnet18, Resnet34 and Resnet 50 trained on the CIFAR-10 dataset and compare the accuracy of each of the loaded models. Here, the objective is to learn how to make use of pretrained models and test them on different datasets. Since the models are used to determine the class of CIFAR-10 test images, this problem is a classification problem.

In the second part of the homework, we are tasked with training and testing several recurrent neural network models such as Recurrent Neural Networks (RNNs), Long-short Term Memory (LSTM) models and Gated Recurrent Unit (GRU) architectures. Here, we train the models to be able to predict energy usage in the next hour when given data from previous time steps. Since the models give out a value instead of a class, this can be described as a regression task.

1.2 Introduction: ResNet

ResNet, short for Residual Network, is a specific type of neural network that was introduced in 2015 by Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun in their paper "Deep Residual Learning for Image Recognition". The ideas introduced in the paper have been successfully used to win several data science competitions.

The main benefit that ResNets offer comes from the residual connections present in their architectures. As the tasks to solve get more complex, neural network models tend to become deeper with a higher amount of layers. However, as the amount of layers increases, the information propagated to the first layers during backpropagation decreases, which limits the learning capabilities of a model.

With the introduction of the residual blocks employed in ResNet, the problems that arise from training deep networks have been alleviated. Residual connections offer an alternative path for the gradient to flow through during training, which helps the model learn the identity function and leads to an increase in performance on deep architectures.

1.2.1 ResNet Key Equations

The residual connections in ResNet can be visualised as below:

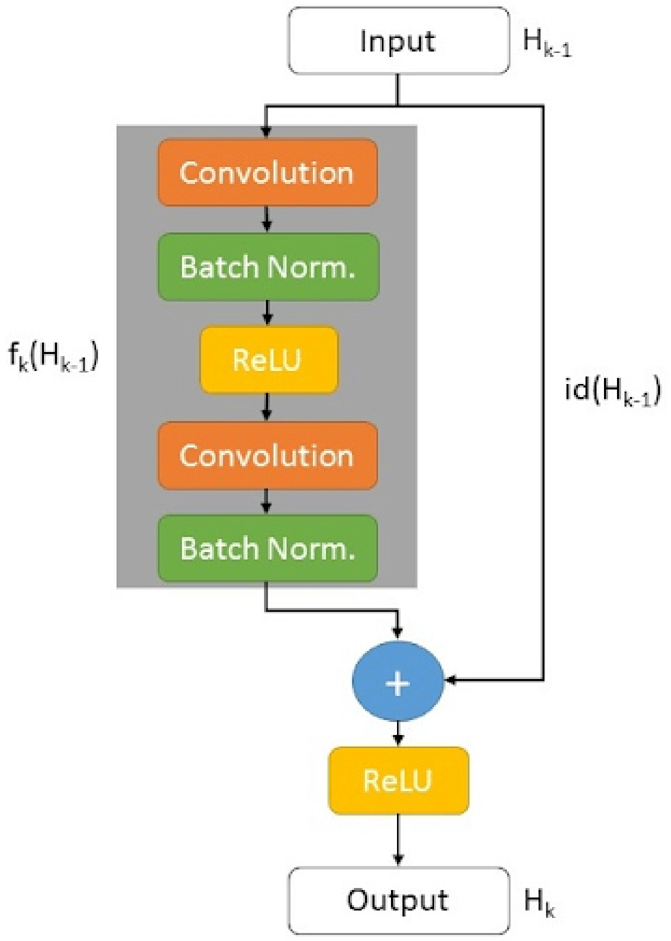

Figure 1: The depiction of the residual connections in ResNet

Here, the input is passed directly to the end and summed up with the output of the weight layer. If we denote the operations performed by the convolution, batch normalization and ReLu on input x as $f_k(x)$, then the output will be:

$$s = f_k(x) + x$$
$$Output = ReLU(s)$$

where s is the output after the residual operation.

The rest of the ResNet is made up of convolutinal layers, pooling layers and nonlinearities. As these were covered in the last homework they will not be explained here again. However, batch normalization is a new concept that should be introduced.

Batch normalization is a technique that aims to make training neural networks faster and more stable. Here, the output of each neuron that is given to the batch normalization layer is rescaled using the mean and variance of the data in a minibatch.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x_i}^{(k)} = \frac{\hat{x_i}^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)2}} + \epsilon}$$

where x_i is the ith input to the batch normalization layer in a minibatch, m is the number of total elements in the minibatch, μ_B is the minibatch mean, σ_B^2 is the minibatch variance and $\hat{x_i}^{(k)}$ is the kth normalized activation after batch normalization.

1.2.2 Structure of ResNet 50

Let us define the architecture of ResNet 50. If the code available at Pytorch_CIFAR10 repository is examined, we can see that the architecture is composed of several Bottleneck blocks. Each block has the following structure:

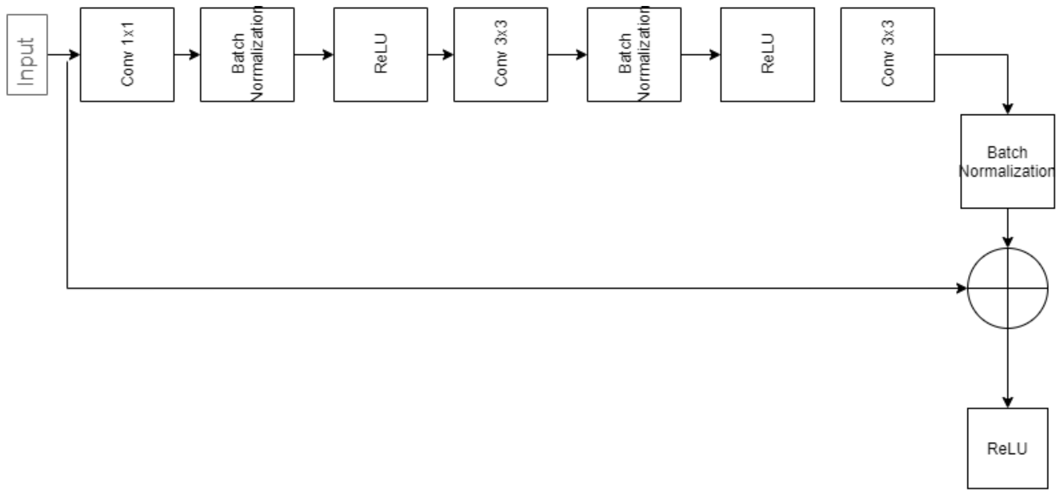

Figure 2: The structure of each Bottleneck block

Since the dimension of the batch norm is simply the output dimension of the previous convolutional layer, let us denote each Bottleneck layer with Bottleneck(a,b,c) where a is the output dimension of the 1x1convolution, b is the output dimension of the first 3x3 convolution and c is the output dimension of 3x3 convolution. Now, we can define the entire architecture as follows:

```
Input (dim = 3*32*32) -->
convLayer (OutputChannel = 64; Kernel = 3*3) -->
BatchNorm2D-->
ReLU -->
convLayer (OutputChannel = 16; Kernel = 5*5; Stride = 1) -->
Relu -->
Bottleneck(a=64,b=64,c=256) x3 -->
Bottleneck(a=128,b=128,c=512) x4 -->
Bottleneck(a=256,b=256,c=1024) x6 -->
Bottleneck(a=512,b=512,c=2048) x3 -->
```

```
AdaptiveAvgPool2d -->
Fully Connected(Input neuron size=2048, output neuron size = 10) -->
Softmax -->
```

1.3 Introduction: RNN/GRU/LSTM

RNN/GRU/LSTM are examples of recurrent neural networks, where data is passed as a sequence and the model makes use of the information learned from the previous steps to make a decision about the present. They are capable of doing this by storing an internal state that serves as their memory which aids them in the decision process. This makes them suitable for applications where there is time sensitive or sequential data in fields ranging from finance to natural language processing.

1.3.1 RNN Key Equations

Introduced in the paper "Learning Internal Representations by Error Propagation" in 1985, the basic RNN is one of the earliest models that are capable of learning from sequential data. The inner workings of RNNs are as follows:

$$hidden_t = tanh(weight_{hidden} * hidden_{t-1} + weight_{input} * input_t)$$
$$output_t = weight_{output} * hidden_t$$

where $hidden_t$ is the hidden state at time t, $weight_{hidden}$ is the weight for the hidden state, $weight_{input}$ is the weight for the input, $weight_{output}$ is the output weight and $output_t$ is the output at time t.

As can be understood from the set of equations, RNNs work by processing both the information propagated from the previous time steps and the current input to generate the current hidden state. It then uses the hidden state to generate the output at each timestep.

1.3.2 LSTM Key Equations

Although RNNs work for short sequences, they do not tend to perform well on longer sequences. This is because the gradient needs to go through continuous matrix multiplication through time during backpropagation, which causes the gradients either to vanish or explode. This inhibits RNNs from learning long-term dependencies.

Introduced in the paper "Long Short-Term Memory" in 1997, LSTMs are capable of retaining both short term and long term dependencies by utilising the different gates within their structure. Unlike RNNs, LSTM has an additional internal state called the cell state to preserve long term information. The value of the hidden (h_t) and cell (c_t) states are controlled using the gates in the network architecture. LSTMs have three gates which perform the following operations:

Input Gate Equations

$$i_1 = \sigma(W_{i_1}.(h_{t-1}, x_t) + bias_{i_1})$$
$$i_2 = tanh(W_{i_2}.(h_{t-1}, x_t) + bias_{i_2}))$$
$$i_{input} = i_1 \odot i_2$$

Forget Gate Equations

$$f = \sigma(W_{forget}.(h_{t-1}, x_t) + bias_{forget})$$

Output Gate Equations

$$c_t = c_{t-1} \odot f + i_{input}$$
$$o_1 = \sigma(W_{output_1}.(h_{t-1}, x_t) + bias_{output_1})$$
$$o_2 = tanh(W_{output_2}.c_t + bias_{output_2})$$
$$h_t = o_1 \odot o_2$$

$$y_t = o_1 \odot o_2$$

where y_t is the output of the model at time t, h_t is the hidden state at time t and c_t is the cell state at time t. The W and bias terms are the different parameters of the model that are updated using backpropagation during learning.

1.3.3 GRU Key Equations

Introduced in the paper "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation" in 2014, the relatively young GRU architecture is another RNN structure used to learn sequential data patterns. Although it is similar to LSTMs in terms of structure, what sets GRUs apart is that they are faster to train and evaluate. This is because GRUs have two gates instead of the three present in LSTMs. Furthermore, instead of storing both a hidden state and a cell state to capture dependencies, GRUs are capable of storing both short and long term information using a single hidden state (h_t). The gates in a GRU perform the following equations:

Reset Gate Equations

$$gate_{reset} = \sigma(W_{input_{reset}}.x_t + W_{hidden_{reset}}.h_{t-1})$$
$$r = tanh(gate_{reset} \odot (W_{h_1}.h_{t-1} + W_{x_1}.x_t)$$

Update Gate Equations

$$gate_{update} = \sigma(W_{input_{update}}.x_t + W_{hidden_{update}}.h_{t-1})$$
$$u = gate_{update} \odot h_{t-1}$$

The final hidden state and output is calculated as follows:

$$h_t = r \odot (1 - gate_{update}) + u$$
$$y_t = r \odot (1 - gate_{update}) + u$$

where h_t is the hidden state at time t, y_t is the output of the model and W are the model parameters.

1.3.4 RNN Structure

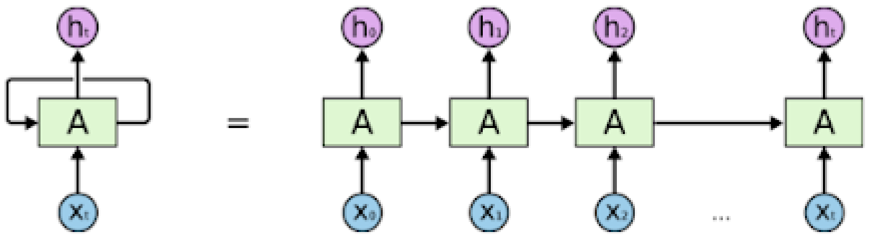

Figure 3: The basic structure of an RNN

As can be seen from the image, the regular RNN does not have any gates within itself. It just makes use of the hidden state to store past information and the current input provided to the model to generate an output.

1.3.5 LSTM Structure

Unlike RNNs, LSTMs make use of gates to control the gradient and decide which part of the past information must be propagated to the next time step.

LSTMs have 2 states. The hidden state is responsible for retaining short term information while the cell state is responsible for carrying long term information.

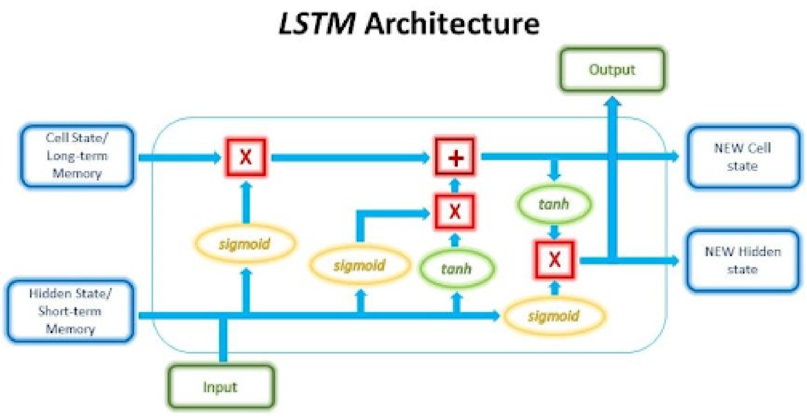

Figure 4: The structure of an LSTM cell

LSTM has three gates. The forget gate, which is represented by the leftmost multiplication operation in Fig. 4, is responsible for deciding which information from the cell state will remain. The input gate, which is represented by the multiplication in the middle of the figure, is responsible for determining which parts of the new information will be used to modify the existing long term memory. Finally, the output gate, represented by the rightmost multiplication in the figure, is responsible for generating the output and the next short term memory by utilising the current input, the previous short-term memory, and the newly computed long-term memory

1.3.6 GRU Structure

Unlike LSTMs, GRUs only have 2 gates to control the flow of information and one hidden state to store dependencies. This state is responsible for carrying information from both short term and long term dependencies on a single vector.

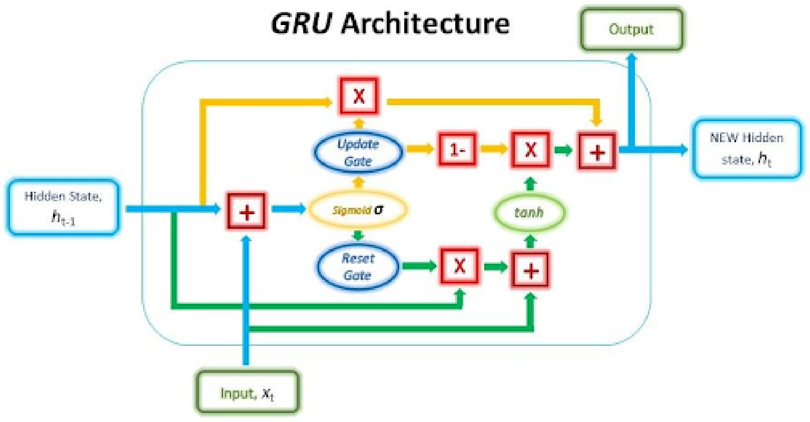

Figure 5: The structure of a GRU cell

There are only 2 gates present in a GRU, the reset gate and the update gate. The reset gate is responsible for deciding which portions of the previous hidden state are to be combined with the current input to propose a new hidden state. On the other hand, the update gate is responsible for determining how much of the previous hidden state is to be retained and what portion of the new proposed hidden state (derived from the reset gate) is to be added to the final hidden state. This also allows the gate to control information about long term dependencies.

1.4 Experiments

1.4.1 ResNet Experiments

How to run The code for this section can be found on the PartA subfolder of the submitted homework. The code was written using the Spyder IDE, so to be able to run the code in sections, the code must be opened using Spyder. Here, the actual code is located at the code.py folder. There is no code for downloading the weights because that was done using the command prompt. Second section of the code.py folder loads the models, 5th section tests the models and 6th section plots the graph.

For this part of the homework, the ResNet18, ResNet36, ResNet50 models pretrained on the CIFAR-10 dataset were loaded using the GitHub repository provided in the homework description. The pretrained models were then tested on the test dataset composed of CIFAR-10 images. The accuracies of each model on this test set is depicted below:

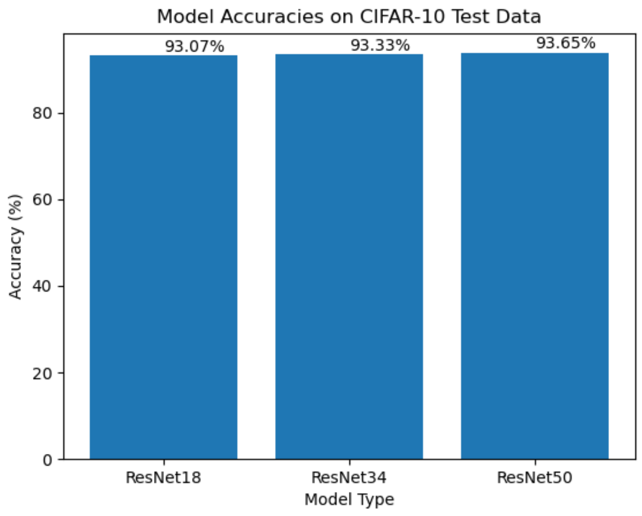

Figure 6: The accuracies of the three pretrained models on CIFAR-10 test set

As can be seen, ResNet 50 performs the best. It is followed by Resnet34 and ResNet18 in terms of performance. From here, it can be inferred that the model with more layers performs better on the test set. However, there is not much difference between the models in terms of performance.

1.4.2 RNN/GRU/LSTM Experiments

How to run The code for this section can be found on the main.ipynb folder in the Part B subfolder of the homework. The code was written using Google Colab, so some of the accessed files were located in my Drive folder. Each section of the code is briefly explained using comments. To train the 2 layer GRU, run the 12th section. To train the 2 layer LSTM, 3 layer GRU and 3 layer LSTM, run section 13. To train the 2 layer RNN, run section 14. Finally, to train the 3 layer RNN, run section 15.To test the 2 layer GRU, run section 16. To test the 2 layer LSTM, run section 17. To test the 2 layer RNN, run section 18. To test the 3 layer GRU, run section 19. To test the 3 layer LSTM, run section 20. Finally, to test the 3 layer RNN, run section 21.

For this part of the experiment, we were tasked with training GRU, LSTM and RNN models on the Hourly Energy Consumption dataset so that we may have a model that can predict future energy consumption based on past data. To do so, 12 csv files were gathered to create the test and train datasets. Then, each model were trained on the training data and tested on the test data. The training/evaluation time and the Symmetric Mean Absolute Percentage Error (sMAPE) on the test data was recorded. The formula for sMAPE is as follows:

$$sMAPE = \frac{100}{n} \sum_{t=1}^{n} \frac{|F_t - A_t|}{(|F_t + A_t|)/2}$$

where A_t is the actual value and F_t is the predicted value of a signal.

Due to hardware limitations, it was noted that the training of an individual model took too long. In order to train the models by the deadline of the homework, the models were trained for only 2 epochs as allowed. Furthermore, during training the error "RuntimeError: CUDA error: out of memory" was commonly encountered. After some research, it was understood that this error was caused by having the batch size too large, which made it difficult for the computer to store all the operations in memory during model training. To overcome this issue, the batch size was reduced to 200 from the default value of 1024.

The following models were created for testing:

GRU 2 Layer: This was a GRU model with 2 layers. The learning rate was set to 0.001for training.

LSTM 2 Layer: This was an LSTM model with 2 layers. The learning rate was set to 0.001 for training.

GRU 3 Layer: This was a GRU model with 3 layers. The learning rate was set to 0.001 for training.

LSTM 3 Layer: This was an LSTM model with 3 layers. The learning rate was set to 0.001 for training.

RNN 2 Layer: This was a regular RNN model with 2 layers. The learning rate was set to 0.001 for training.

RNN 3 Layer: This was a regular RNN model with 3 layers. The learning rate was set to 0.001 for training.

The results obtained after training each model and testing them on the test set is as follows:

Table 1: Experimentation results of different models

Model	**Training Time (s)**	**Evaluation Time (s)**	**Evaluation sMAPE**
GRU Model 2 layer	763.7	7.13	0.239%
LSTM Model 2 layer	908.6	9.56	0.266%
RNN 2 layer	310.1	2.68	0.297%
GRU 3 layer	1302.2	11.72	0.261%
LSTM 3 layer	1512.3	16.04	0.283%
RNN 3 layer	415.5	4.26	0.294%

As expected, the RNN model is fastest to train and evaluate, followed by the GRU and LSTM. This is expected since training and testing time depend on the number of gates. For both 2 and 3 layered models, best performance was obtained when GRU was utilised. However, the performance of the other models were also similar. An interesting observation is that having more layers actually resulted in a decrease in performance. However, this can be attributed to the amount of epochs that we trained each model. More sophisticated models need more training to update their parameters. Since we only trained for 2 epochs, we did not give the three layered models adequate time to update their parameters, which resulted in a less than ideal performance.

Let us visualise some of the testing results.

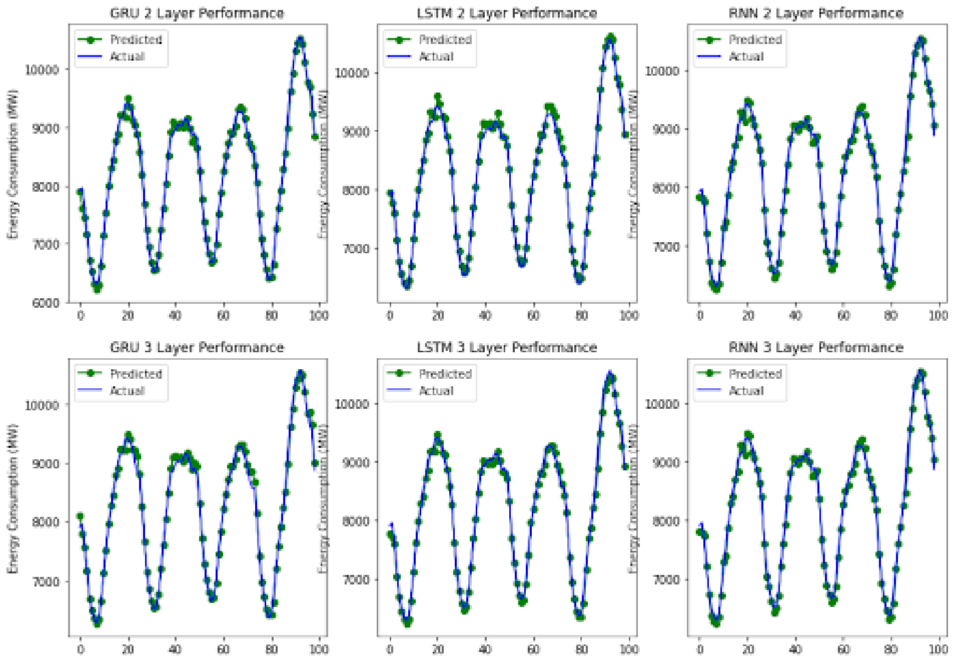

Figure 7: The visualised testing results on the last 100 points of the first test for all trained models

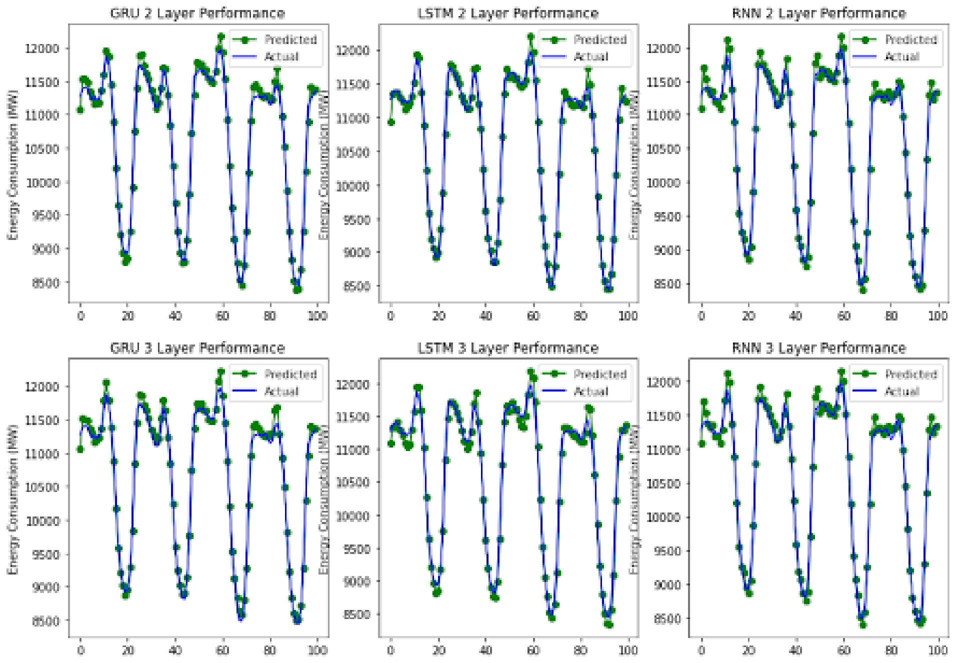

Figure 8: The visualised testing results on the first 100 points of the fourth test for all trained models

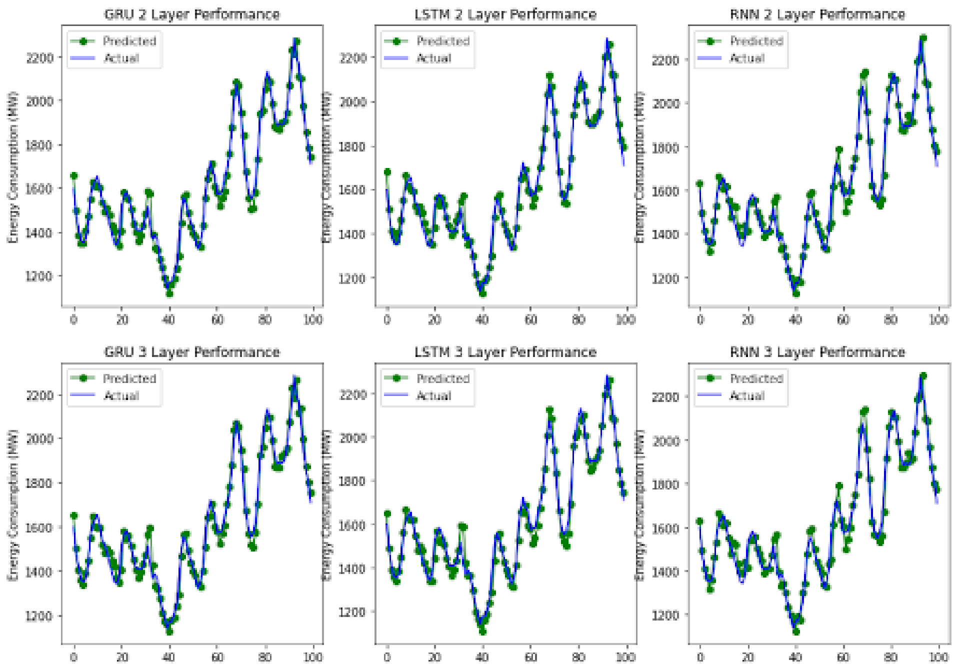

Figure 9: The visualised testing results on the first 100 points of the tenth test for all trained models

As can be seen, all of the trained models perform reasonably well on this test dataset. Although there are some errors present, overall the model is capable of generating realistic predictions.

Let us now plot the change in sMAPE loss on the test data as each model trains.

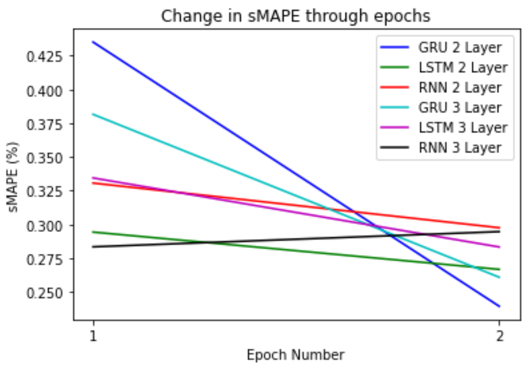

Figure 10: The change in sMAPE loss on the test data as each model trains

As can be seen, for all models except the 3 layered RNN, the test loss decreases as epochs progress. Here, it is interesting to note that although the training error was decreasing during training of the 3 layered RNN, the test error slightly increased. Training for more epochs would have probably led to a better decrease in loss. Furthermore, it appears that the initial error of the 2 layered GRU is the highest. However, it quickly diminishes and the 2 layered GRU becomes the best performing model in our set of models. The performance of the RNNs seem to be the worst among both the 2 layered models and the 3 layered models at the end of 2 epochs. This is expected since RNNs have difficulty learning long term dependencies. Since the LSTM and GRU models do not suffer from this, they provide a better performance. However, it should be noted that since training was only done for 2 epochs, it is difficult to accurately interpret the loss curves.

1.5 Conclusion

In conclusion, we managed to load pretrained ResNet models and test these models on the test set for CIFAR-10. Furthermore, we were able to train and test RNNs, LSTMs and GRUs on the Hourly Energy Consumption dataset and got models that can accurately predict future energy consumption based on test data. Overall, it can be said that the homework was successfully completed.

During the implementation of the homework, there were numerous issues. First of all, loading the pretrained weights took a long time. Furthermore, training the RNN models for the second part of the homework proved to be challenging. The slow training times forced me to lose a lot of time if something went wrong with the computer or the code. Furthermore, figuring out the cause of the memory error that came up during training took a long time as well.

Reference

He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

Cho, Kyunghyun, et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." arXiv preprint arXiv:1406.1078 (2014).

Loye, Gabriel. “Gated Recurrent Unit (GRU) with Pytorch.” FloydHub Blog, FloydHub Blog, 10 Feb. 2020, https://blog.floydhub.com/gru-with-pytorch/.

Kumra, Sulabh and Kanan, Christopher. (2017). Robotic Grasp Detection using Deep Convolutional Neural Networks. 10.1109/IROS.2017.8202237.