Damian Hupka
801091092
ECGR 4090 - Real-Time AI

Homework #3

Problem 1

The code provided by the "Deep Learning with PyTorch" Github repository was used as a base framework for completing this problem. Throughout all of this problem as well as the extra points section, the autograd library was used in derivative computation to increase ease of implementation in this problem. Similarly, the temperature prediction example as provided in the book was used for the input dataset, and the model was trained over 5000 epochs to iteratively reduce the loss. Varying optimizers were used to observe associated loss with each. Figure 1 below shows the loss of the original, linear model, with the SGD optimizer over the 5000 epochs. The "hw3_originalmodel.ipnyb", "hw3p1.ipynp", and "hw3p1extra.ipynb" notebooks provided in the submission package were used in generating the data, and eventually modifying the model. The notebook "hw3_originalmodel.ipynb" was used for gathering the data associated with Figures 1 & 2 below.

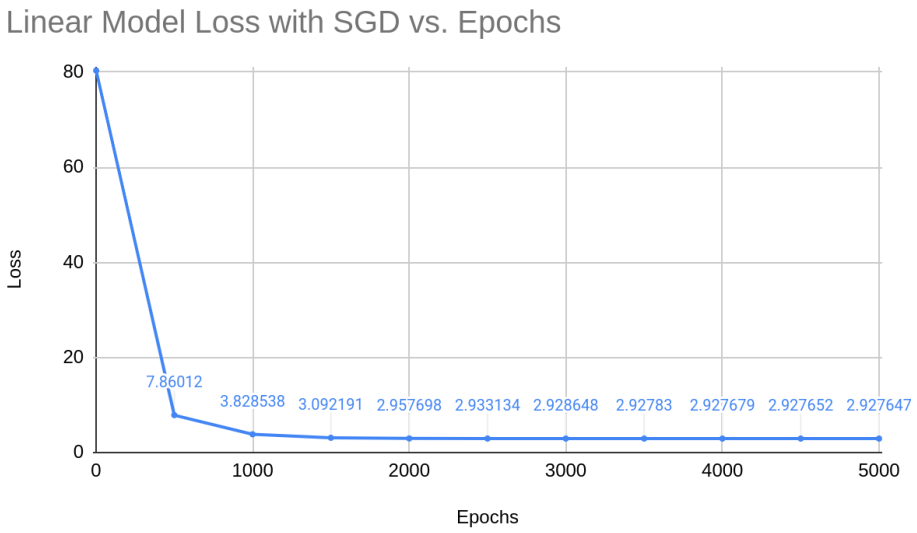

Figure 1: Linear Model Loss vs. Epochs (SGD)

As in the figure above, it is observed that the loss is initially a very high value of 80, but within 500 epochs this value reaches approximately eight and will gradually decrease over the next 4500 epochs to approximately three. Next, the Adam optimizer was used with the linear model and is shown in Figure 2 below.

Damian Hupka
801091092
ECGR 4090 - Real-Time AI

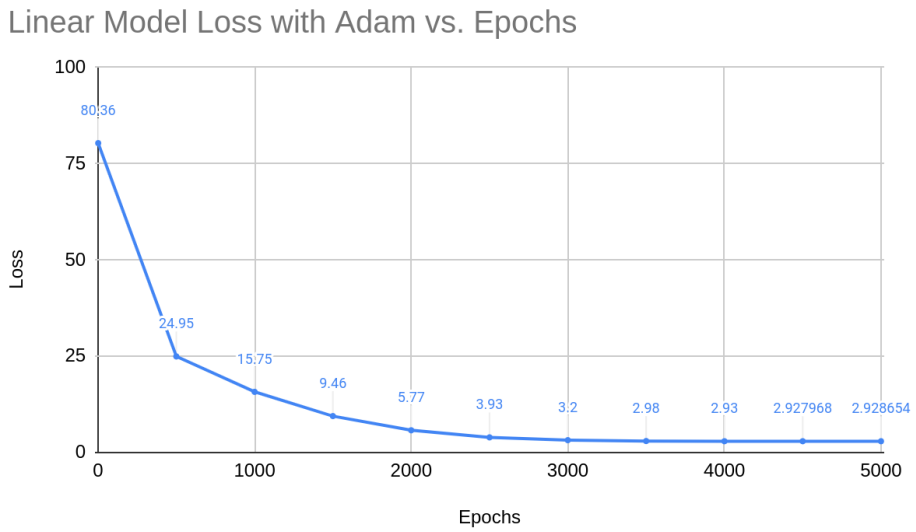

Figure 2: Linear Model Loss vs. Epochs (Adam)

Much like the SGD optimizer, the loss is initially 80 and significantly decreases to 25 over the first 500 epochs; however, with this optimizer, the loss does not stabilize to near the minimum value until around 3500 epochs. But, the loss of each of the optimizers with the linear model is approximately the same.

The linear model that was provided in the temperature prediction example was modified to the following non-linear model: $w_2 * t_u^{\,2} + w_1 * t_u + b$ and similar experiments were conducted to examine the loss related to the non-linear model and the comparison to the original linear model loss as above. The original data from the "hw3_originalmodel.ipnyb" and figures associated with such as well as the data within the "hw3p1.ipynp" notebook with the non-linear model were the sources of generating the data for Figures 3 - 7 below. Initially, the loss of the non-linear model using the SGD optimizer was examined over 5000 epochs and the results for such are below in Figure 3.

Damian Hupka
801091092
ECGR 4090 - Real-Time AI

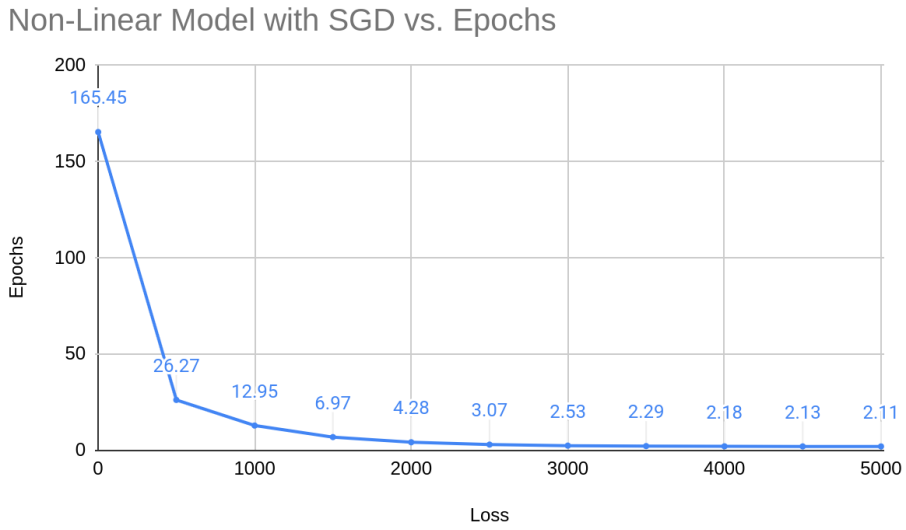

Figure 3: Non-Linear Model Loss vs. Epochs (SGD)

The figure above shows the loss beginning at a value of 165 and within 500 epochs this value drastically decreases to approximately 26, then over the next 2000 epochs the loss reaches three, and lastly will eventually reach approximately two over the remaining epochs.

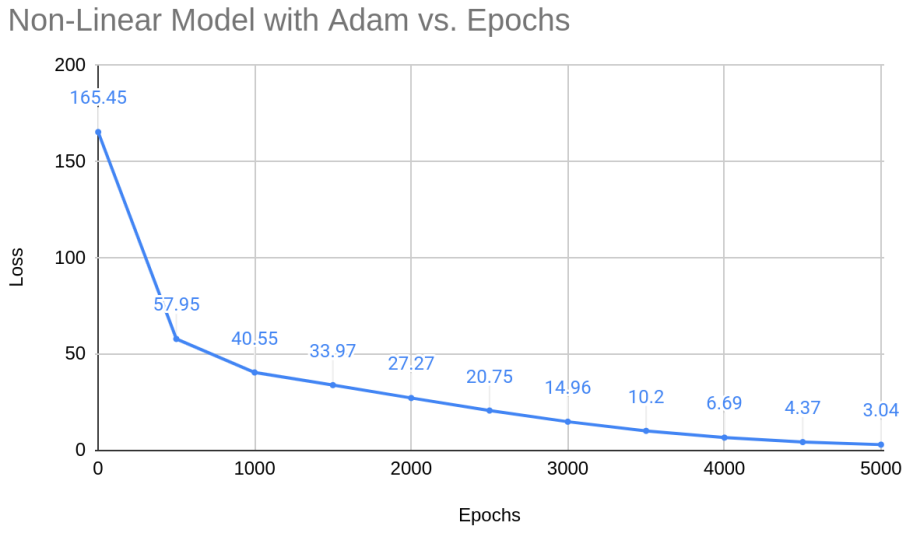

Figure 4: Non-Linear Model Loss vs. Epochs (Adam)

Similar to the linear model with the Adam optimizer, the loss will reach near the minimum value much later, with a smaller decrease in each loss between every 500 epochs. Figure 5 below shows a comparison of the loss of the linear vs. non-linear model with each using the SGD optimizer.

Damian Hupka
801091092
ECGR 4090 - Real-Time AI

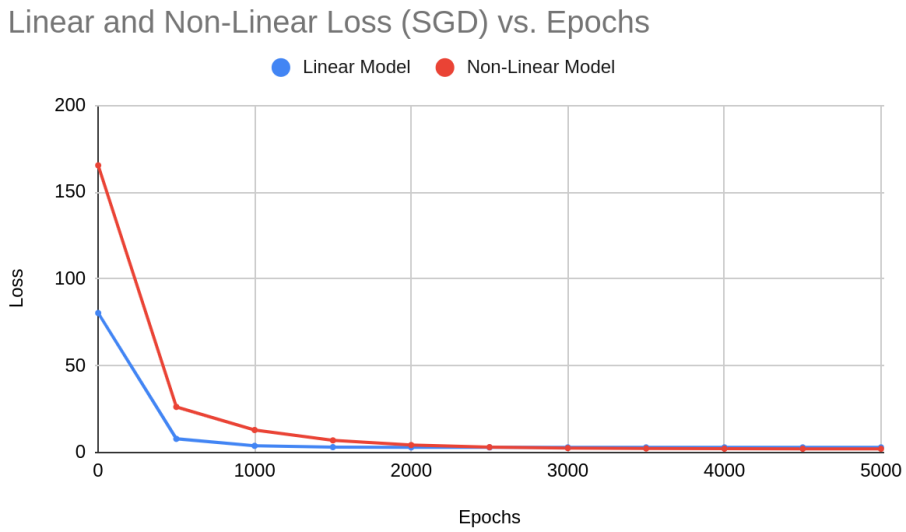

Figure 5: Linear Model vs Non-Linear Model Loss Comparison (SGD)

The figure above shows the non-linear model exhibiting a loss of approximately double that of the linear model at the first epoch. However, both models reach near the respective minimum value in roughly the same amount of epochs. Furthermore, it is difficult to visualize in the plot above, but based upon the previous figures the final loss of the non-linear model was approximately one less than that of the linear model. Thus, the non-linear model performed marginally better using the SGD optimizer. The same comparison was conducted as in Figure 6 below, but with each model using the Adam optimizer.

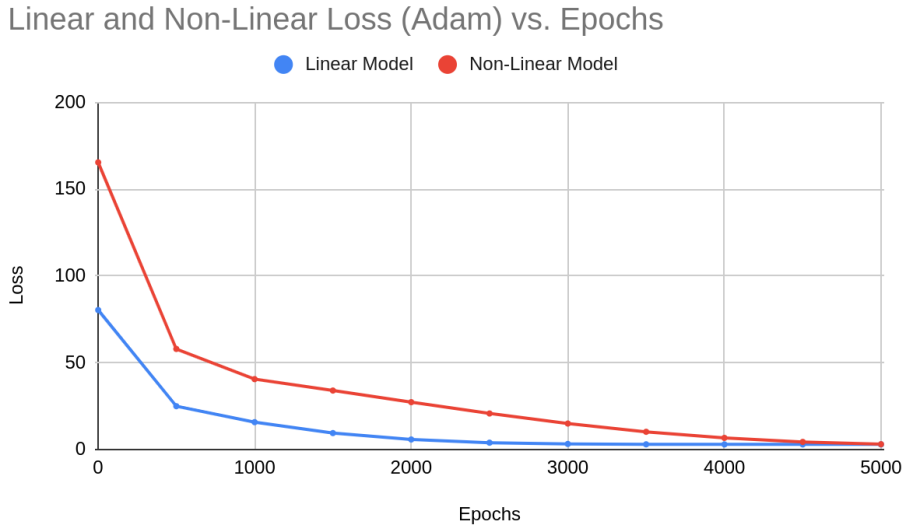

Figure 6: Linear Model vs Non-Linear Model Loss Comparison (Adam)

As in Figures 2 & 4 separately, relating to the linear and non-linear model respectively. The Adam optimizer takes a greater number of epochs to reach the near minimum value for both models. Thus, due to the loss of the non-linear model being so much larger than the linear model initially, for around the first 4000 epochs the loss of the non-linear is substantially greater than the linear model. But, once 5000 epochs has been reached the loss for the non-linear and the linear model is nearly identical which can be observed in analyzing the data at 5000 epochs for figures 2 & 4. Lastly, the two models were plotted over the input dataset and are shown below in Figure 7 and 8.

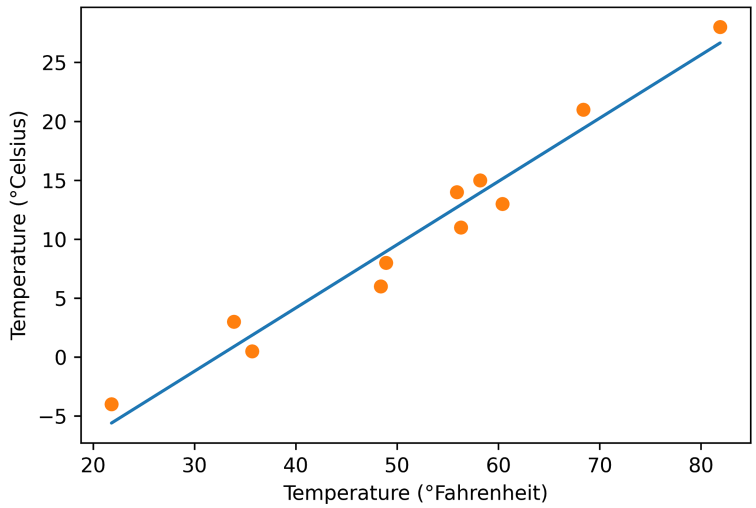

Figure 7: Linear Model vs Input Dataset

The linear model fits the dataset relatively well; however, if new data was used it is unlikely this model would exhibit such accuracy. Thus, figure 8 below shows the non-linear model against the input dataset.

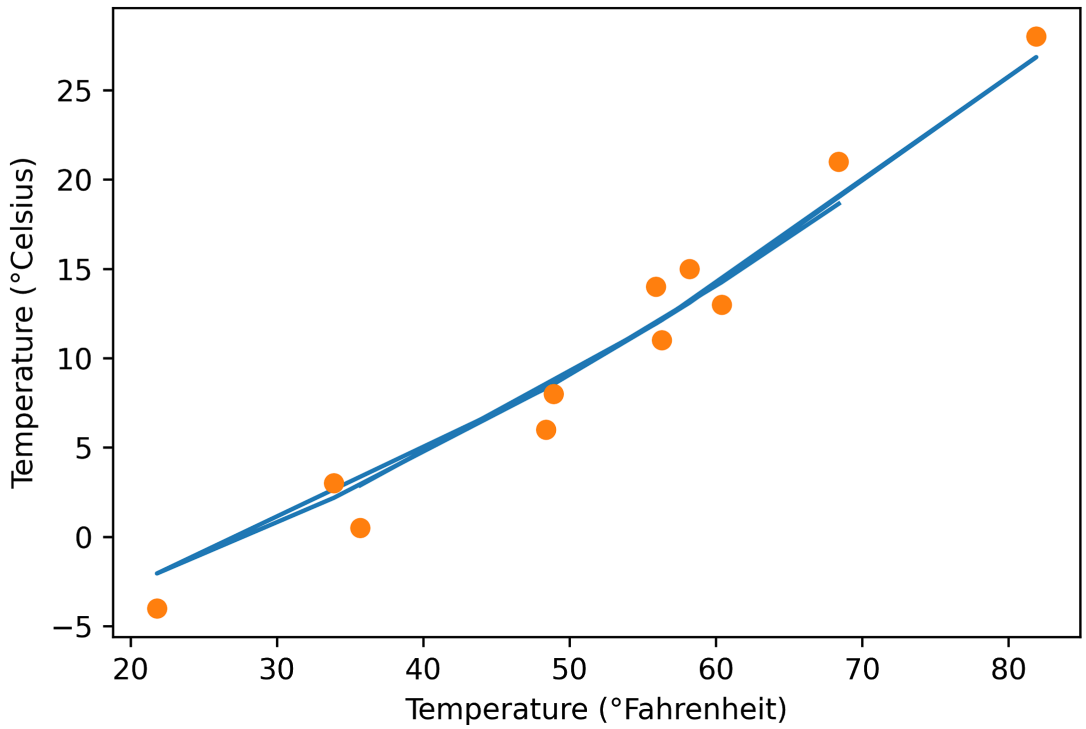

Figure 8: Non-Linear Model vs Input Dataset

Due to the functionality of the "matplotlob" library the non-linear model is not appearing as a smooth parabola as expected, but the model will still be fit in the same way. With this non-linear model, the inference is marginally closer to that of the ground truth values, and this model should be more apt to handle new data that was not within the training dataset simply due to the parabolic nature. Overall, it was observed that the non-linear model performs marginally better than the linear model in most cases. Furthermore, it is likely that the non-linear model will perform better for new data.

Code referenced from the provided git repository by Deep Learning With Pytorch here: https://github.com/deep-learning-with-pytorch/dlwpt-code/tree/master/p1ch5.

Problem 1 Extra Points

This problem was completed similar to that of the original problem 1; however, the training dataset was randomly split with 80 percent of the data still being used for training, but 20 percent being used for validation. Doing this split can help to verify that the model is not overfitting, and will be more apt to predict new data that is not included in the dataset in the future. Figure 8 below shows the training and validation loss of the non-linear model using the SGD optimizer over 5000 epochs.

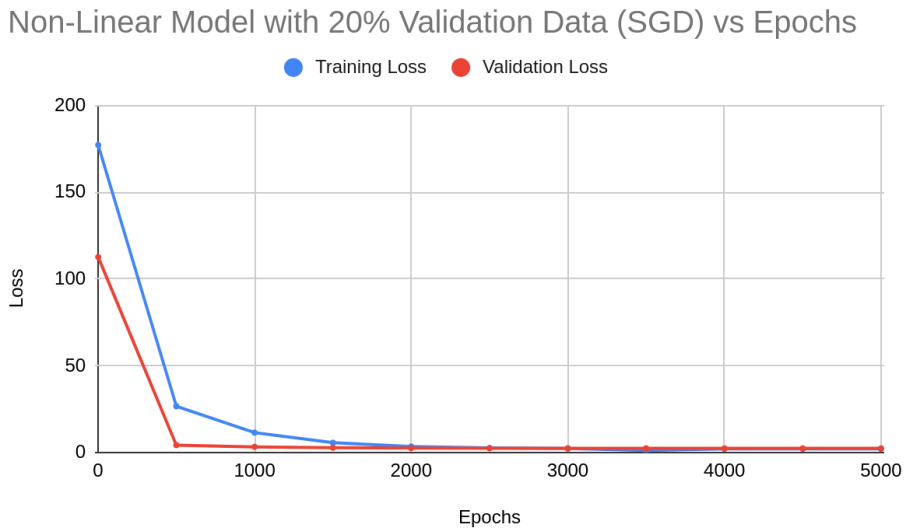

Figure 8: Non-Linear Model Loss (Training & Validation) vs. Epochs (SGD)

The figure above shows both the training and validation loss remaining relatively similar after the first 500 epochs, which is to be expected, and both approach similar nearly minimum values (within .1 of one another) after only 1500 epochs with the SGD optimizer. The same analysis was performed but using the Adam optimizer. These results are below in figure 9.

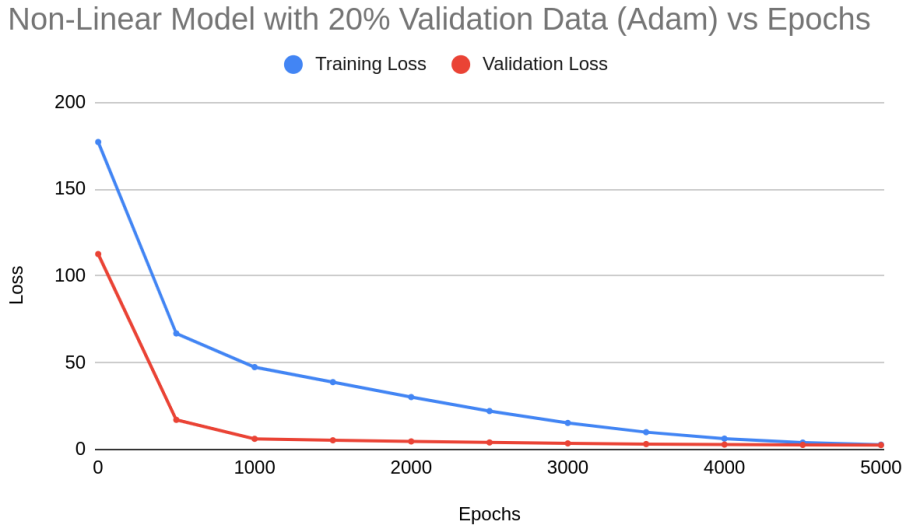

Figure 9: Non-Linear Model Loss (Training & Validation) vs. Epochs (Adam)

As observed when the Adam optimizer is used, it takes a much greater number of epochs for both the training and validation loss to reach a near minimum value. But, when this occurs around 4500-5000 epochs the loss is also within 10 percent of each other, which is again to be expected for the training and

validation loss to be similar at the end of training. Lastly, the loss of the non-linear model which had split the training data for validation, and the loss of the non-linear model that used all of the data in training was compared using both optimizers as in Figures 10 and 11 below.

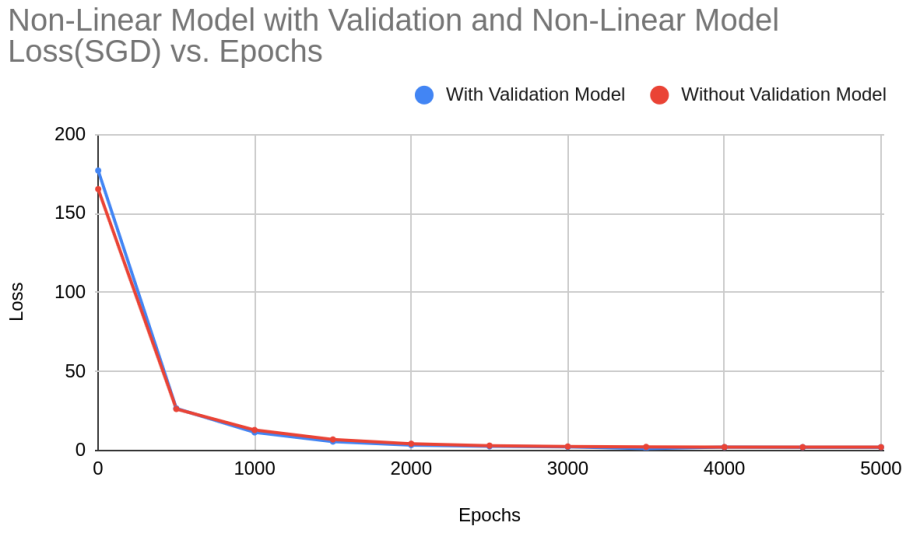

Figure 10: Non-Linear with Validation vs Non-Linear Loss (SGD)

As observed, the loss when reserving some of the data for validation is very similar to that of the loss with the full dataset for training. Thus, it can be concluded that with this dataset it should be encouraged to reserve a portion of the data for validation as it can help to prevent overfitting and ensure the model will be able to accurately predict new data.

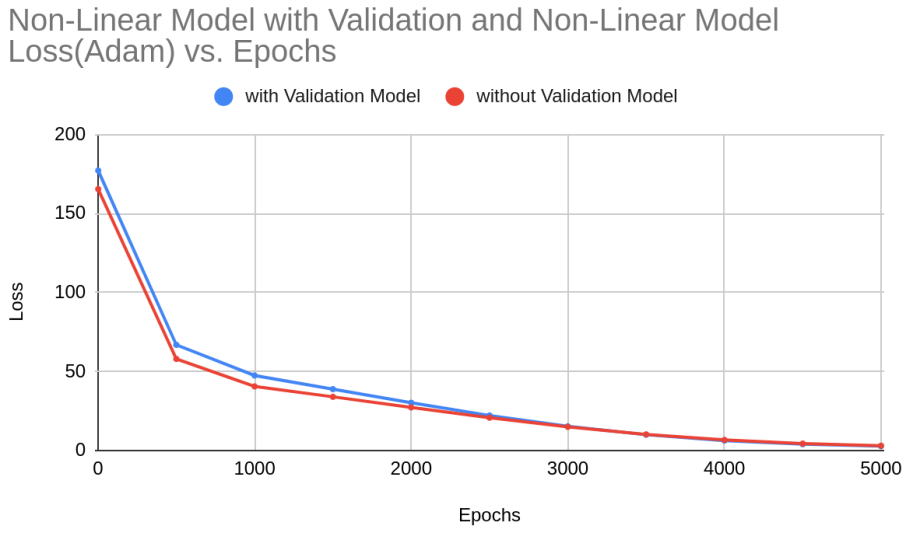

Figure 10: Non-Linear with Validation vs Non-Linear Loss (Adam)

Figure 10 above exhibits the same relationship between the plotted losses with both being similar when the Adam optimizer is used. However, the loss for each of these is marginally greater than the loss when using the SGD optimizer. Thus, it is unlikely that for this application the Adam optimizer is to be used, but if so, splitting the dataset for validation is entirely valid and will not substantially affect overall loss so long as the portion to be reserved is relatively small.

Problem 2

The Human Key-point estimation algorithm provided by the NVIDIA AI IOT "trt_pose" repository was implemented for this problem. The example demonstration that was provided on this repository was modified in a few ways to fit the desired implementation and the required code to run such inference can be found in the submission package as the "live_demo.py" file (modifications can be made to fit the desired model to be used).The "live_demo.ipynb" was changed to be a simple python script, this modification was done to ensure minimal memory strain on the NVIDIA Jetson Nano, as launching a Jupyter Notebook on the Chromium based browser imposes a significant load on the board alone. Furthermore, the FPS benchmark measurement differs from the original as the benchmark provided will be highly inaccurate as it calculates the average FPS when the Jetson Nano is under the most load. Thus, the FPS is measured and displayed on the OpenCV window capture on a frame by frame basis, and the average is observed in this way. With this measurement method, the FPS will likely be very low initially, but as the load on the board stabilizes this value will increase to an accurate, acceptable maximum. Also, for providing a sample video to the model, the OpenCV library was used instead of the Jetcam/IPython widgets. The model inference was conducted on a video of a mother and a child slowly walking towards the camera. **An example output frame is provided below in Figure 11.**

Figure 11: Human Key-Point Inference.

The trt_pose library includes a number of models to be used for inference. This experiment utilized the ResNet 18 and the DenseNet 121 models. TensorRT provides functionality for generating an optimized

model on a system-by-system basis. However, due to the poor processing performance, and extremely limited memory availability (even with a 6GB swapfile) building these optimized models were exceedingly time intensive. Building the optimized ResNet18 model took approximately two hours, and then completed the video inference. The building of the optimized Densenet 121 model took approximately 5 hours, with the program crashing immediately after saving the model. Luckily, optimizing these models only needs to be completed once so once these were completed measuring the various frame rates associated with each model was much more fluid. Table 1 below shows the FPS results of the two models with the unoptimized and optimized versions respectively.

	Frames Per Second (FPS)
ResNet 18 (unoptimized)	7.45
ResNet 18 (optimized)	9.06
DenseNet 121 (unoptimized)	.5
DenseNet 121 (optimized)	7.31

Table 1: Model Inference FPS of Pose Estimation

Interestingly, for the ResNet 18 model the optimized version only performs a few frames better than that of the unoptimized model. However, the unoptimized DenseNet 121 model performs substantially worse than the optimized, and is nearly inoperable. This poor performance is likely due to the DenseNet 121 Model having many more layers, with each of these layers being fully-connected, dense layers which requires much more memory per inference than that of a smaller model. The larger memory requirement is due to the generation of more parameters and requiring more MACs. However, when optimizing for the NVIDIA Jetson Nano running inference on the model will not be reaching the memory bottleneck as readily as before. As expected by the provided documentation on the trt_pose repository the FPS of the optimized ResNet 18 model is greater than that of the DenseNet 121 model. However, the Jetson is unable to reach even close to the 22 or 12 FPS benchmark for ResNet 18 and DenseNet 121 respectively documented in the repository. This is likely due to the Jetson Nano not operating in most efficient conditions for factors such as power, memory usage, and temperature.

Code referenced from the provided git repository by NVIDIA-AI-IOT for TensorRT pose estimation: https://github.com/NVIDIA-AI-IOT/trt_pose.

Damian Hupka
801091092
ECGR 4090 - Real-Time AI

Problem 3

Problem 3.1

This problem implemented the YOLOv5 Object detection with the human pose estimation provided by trt_pose on the NVIDIA Jetson Nano. The small YOLOv5 model was used and inference was conducted on a batch size of one video. The required code to run inference on a video is provided in the "hw3p3.py" file, and a sample video is also included in the submission package to be run without any modifications so long as the file directory structure is spared. The FPS of this inference using both models and drawing to each frame was calculated and found to be **approximately 1.5 FPS**. As expected, this observed value is substantially less than that of YOLOv5 or trt_pose running inference exclusively. This decrease in performance is caused by that for each frame of the video it is necessary to do two inferences (YOLOv5 and trt_pose) which is increasing the load on the memory of the Jetson Nano. An example inference frame is shown below in figure 12.

Figure 11: Human Key-Point Inference and YOLOv5.s Object Detection

Problem 3.2

Similar to problem 3.1, both YOLOv5 and the trt_pose human pose estimation models were to be used simultaneously. However, instead of a preexisting video, a live camera feed from a USB camera would be used for inference but, due to limitations of the available memory capacity on the NVIDIA Jetson Nano, this portion could not be completed. Furthermore, based on the previous problem 3.1 we can assume that the real-time, live video inference would offer an even more substantially decreased FPS as using the NVIDIA Jetcam library is similarly taxing on the Nano. The only necessary change to conduct real-time

inference from a live camera feed would be to use the Jetcam library to read each frame from the connected USB camera, and pass each frame into the two models, and draw this inference to an output feed.