Neocis - Software Assessment Report

Approach

Problem statement - From the image of a drill bit, identify the drill size. This task has been solved as a supervised learning - classification problem wherein we train the model with different frames and provide it the expected drill sizes. During testing, we provide the frame to the model and predict which drill bit size it belongs to.

Brief description of code implementation

Frame creation from video data

I utilized 'ffmpeg' to generate frames from the video data provided. Next, I created a bash script to run ffmpeg on all videos and store the frames in the directory structure shown below. Within every directory, the frames for that particular video are stored.

```
total 704
drwxrwx--- 20 adiyer adiyer   20 Nov 17 12:39 .
drwxrwx--- 11 adiyer adiyer   35 Nov 18 12:42 ..
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:37 20x26dark
drwxrwx---  2 adiyer adiyer 1619 Nov 17 12:37 20x26light
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:37 20x28dark
drwxrwx---  2 adiyer adiyer  981 Nov 17 12:37 20x28light
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:37 28x22dark
drwxrwx---  2 adiyer adiyer  972 Nov 17 12:37 28x22light
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:38 35x19dark
drwxrwx---  2 adiyer adiyer  966 Nov 17 12:38 35x19light
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:38 35x22dark
drwxrwx---  2 adiyer adiyer  972 Nov 17 12:38 35x22light
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:38 35x28dark
drwxrwx---  2 adiyer adiyer  966 Nov 17 12:38 35x28light
drwxrwx---  2 adiyer adiyer 1202 Nov 17 12:39 35x30dark
drwxrwx---  2 adiyer adiyer  975 Nov 17 12:39 35x30light
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:39 42x22dark
drwxrwx---  2 adiyer adiyer  972 Nov 17 12:39 42x22light
drwxrwx---  2 adiyer adiyer 1213 Nov 17 12:39 42x30dark
drwxrwx---  2 adiyer adiyer  981 Nov 17 12:39 42x30light
```

ML Pipeline

Dataset Creation

The entry point for training the model is the main.py file wherein we take multiple input parameters to configure the model parameter settings. First, we create the dataloaders from the extracted frames. This is done in the 'createDataLoader' function. We split the entire training data into training, validation and testing for evaluating the model performance. No additional preprocessing has been performed on the images except for resizing the frames to 1280x960. Preprocessing such as transforming the pixels to lie in a [0,1] range and then by normalizing the images to zero mean and unit variance could be done in the future. Once the dataloaders have been created, we identify the type of model which has to be instantiated i.e the model with a basic LeNet5 architecture or a LeNet5 model enhanced with Batch Normalization layers. Post this, we enter the training or testing stages of the code depending on the arguments passed while running main.py.

Training the model

First, we create an object of the optimizer (optimizer type is provided as part of the input arguments to the main.py). Second, we begin the training loop of the model. We utilize CrossEntropyLoss for this task. The training loop iterates over epochs and batches of data provided by the training dataloader. For every epoch, once it completes a training cycle - we evaluate the model on the validation dataloader to evaluate the performance of the model. Third, once the training loop is completed we save the model locally. Next, we generate predictions on the model by utilizing the test dataset and compute the accuracy score of the model. Finally, we plot the training vs validation loss curves.

Model Architecture

For the purpose of this assessment, I experimented with two architectures. Both architectures are convolutional neural networks based on the LeNet5 architecture.

1. Basic LeNet5 - This neural architecture uses two convolutional layers followed by three affine/fully connected layers. It also utilizes max pooling between the convolutional layers.
2. BatchNorm LeNet5 - This architecture is quite similar to the previous one - however a key difference is that it utilizes batch normalization after the convolutional layers. The reasoning behind using this was that batch normalization helps accelerate convergence and overcomes the internal covariate shift problem in such networks. However, batch normalization is usually more effective for deeper networks so we might not observe significant differences in performance with this enhancement.

Implementation Details

For the experiments, all models were trained for 5 epochs (enough for convergence). Training duration per model took around 30 minutes. All experiments used models with a batch size of 32. I experimented with different optimizers such as Adam, AdamW, SGD with Momentum, RMSProp on both the architectures. Moreover, the learning rate was also modified to see how it affects the model performance. These results have been tabulated and presented in the next section with analysis.

Model Inference

A separate python file has been created for the purpose of model inference which takes as input the location of the unseen data as required by the assessment. Images can also be passed to the inference.py script by using the 'inferenceDataset' parameter. Here, it uses the saved model file to generate prediction results on the unseen data.

Results and Analysis

Optimizer Name	Accuracy
adam	0.9997
adamw	0.9995
Sgd with momentum	0.1410
rmsprop	0.9975

Table 1 - Effect of optimizer on 'Basic' LeNet5 model

Optimizer Name	Accuracy
adam	0.9881
adamw	1
Sgd with momentum	1
rmsprop	0.141

Table 2 - Effect of optimizer on 'BatchNorm' LeNet5 model

Learning Rate	Accuracy
1e-2	0.156
1e-3	0.9997
1e-4	0.9995

Table 3 - Effect of learning rate on 'Basic' LeNet5 model - using adam optimizer

Note - Dataset creation methodology

I devised a script *'frameExtractor.sh'* to run ffmpeg for frame creation from the videos provided. This saves all possible frames of the video in a .jpg format (as opposed to .png) for storage optimization. Utilizing these .jpg files, we create the dataloaders for training, testing and validation of the model. However, during inference it will read .png files as required by the assessment. There isn't a significant difference in training with .jpg and evaluating .png files which led to this design decision.

While creating the splits for training, testing and validation, I used the in-built *'torch.utils.data.random_split'*. While this creates separate non-overlapping datasets - given the nature of our video data, many frames look exactly the same which could mean that while the frame itself is different, the model might have been exposed to the content of the frame during training which creates an implicit leak in test and validation. This could be addressed by selecting frames of the test and validation from an interval of time which hasn't been included in the train set to prevent such data leaks. This could be improved in the future.

Network Architecture Analysis

The proposed solution experiments with two main neural network architectures based on the LeNet5 model. One of the architectures utilizes batch normalization while the other does not. Both models have two convolutional layers followed by three affine/fully connected layers. From the results obtained - there is not much of a significant difference in the accuracy scores due to reasons mentioned above. However, it could also signify that the network is powerful enough to capture rich features from the images of the different drill bits provided to it.

Regarding alternative network architectures, we could utilize a deeper convolutional neural network or try state-of-the-art vision transformers to observe how it would work. While a deeper network performs better in such image classification scenarios, the current model architecture performs relatively well for this task. Another point which can be improved by further analysis is the training time for this model as it takes about 30 mins for 5 epochs.

Model Hyperparameter Tuning

As explained in Table 1, I experimented with a few different optimization algorithms such as Adam, AdamW, Stochastic gradient descent with momentum and RMSProp. Moreover, I tried a few different learning rates to see how it affected the performance of the models. From the results obtained it can be observed that the model trained with AdamW, learning rate 1e-3, batch size 32 and enhanced with Batch normalization gave the best performance. This model will be used to run inferences on unseen data. Overall, there are many other hyperparameters that I would have liked to improve such as the number of epochs of training, batch size of the data loaders, introducing weight decay for regularization etc - however, due to the time constraint of this assessment and the run time of models - I was able to experiment only with the models

mentioned above.

Model Performance

A few loss plots and confusion matrices have been displayed in this report to provide more insights into the model training. From the results and plots, it can be seen that RMSProp and SGD with momentum give inconsistent results. For instance, when the model is trained without batch normalization and optimized with SGD momentum, it has a poor accuracy score. Similarly, the model trained with batch normalization and RMSProp has a poor accuracy score as well. For these reasons, these optimizers were not used in developing the final model.

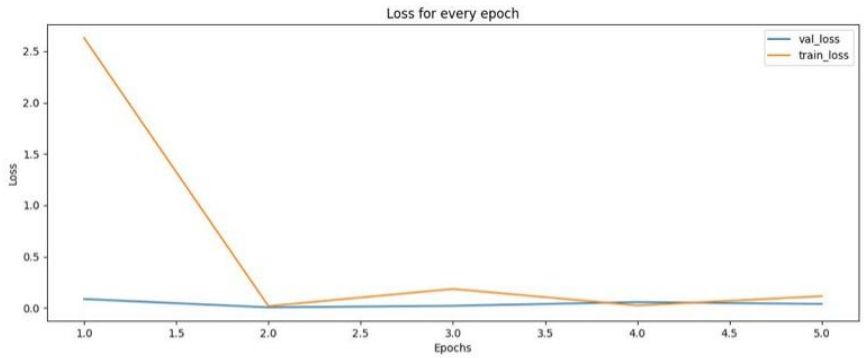

Fig. 1 - Loss vs Epoch curve for 'BatchNorm' LeNet5 model with Adam optimizer

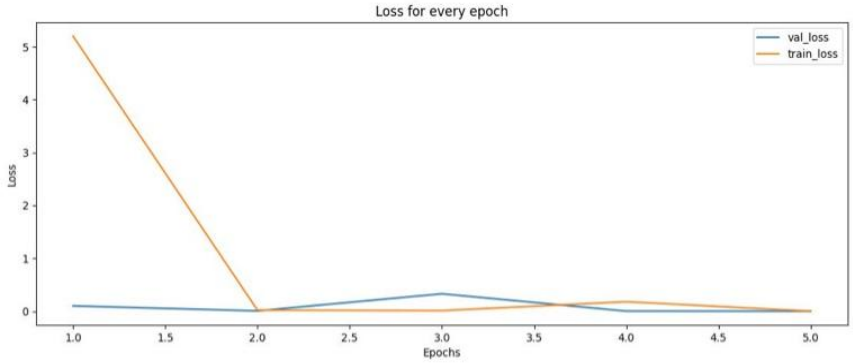

Fig. 2 - Loss vs Epoch curve for 'BatchNorm' LeNet5 model with AdamW optimizer

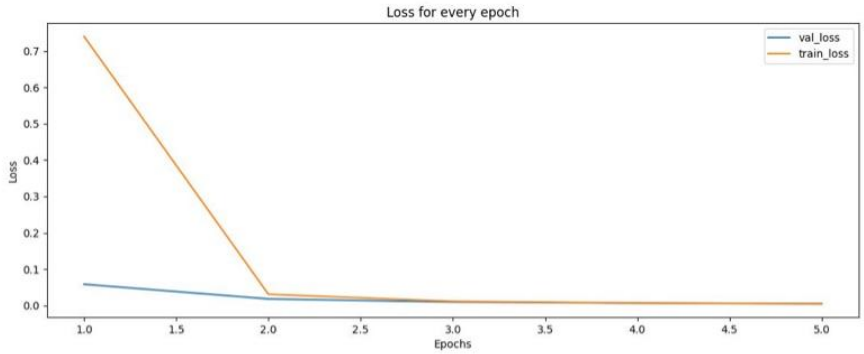

Fig. 3 - Loss vs Epoch curve for 'BatchNorm' LeNet5 model with SGDM optimizer

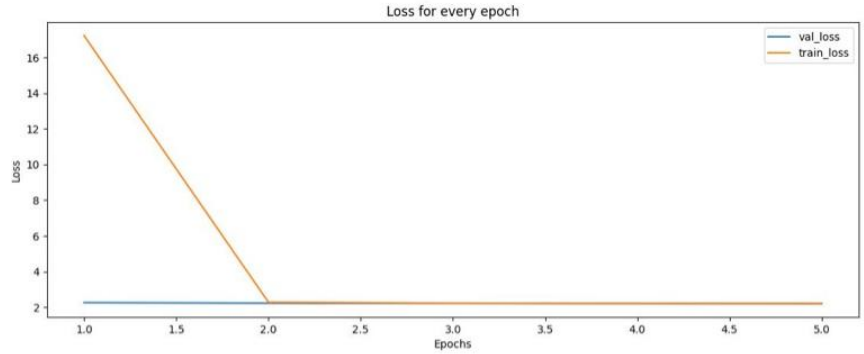

Fig. 4 - Loss vs Epoch curve for 'BatchNorm' LeNet5 model with RMSProp optimizer

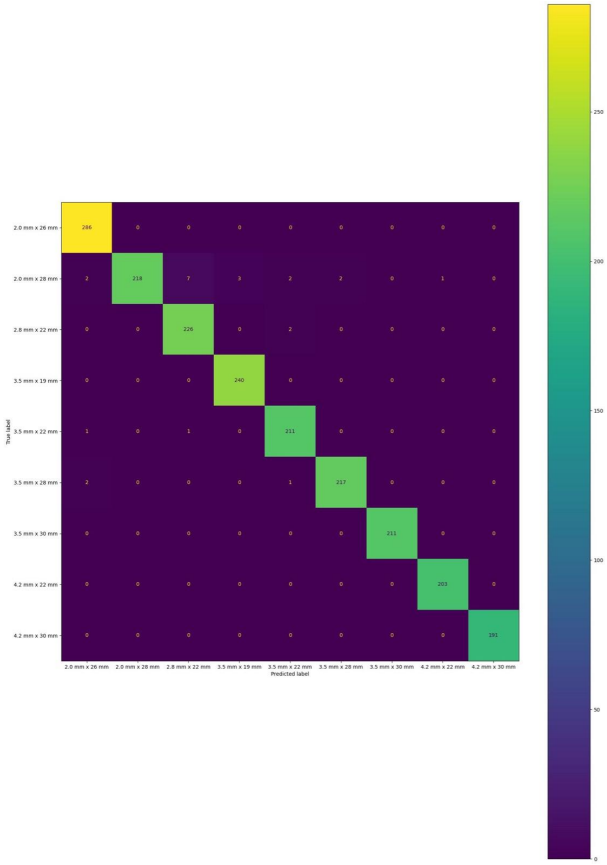

Fig. 5 - Confusion Matrix for 'BatchNorm' LeNet5 model with Adam optimizer

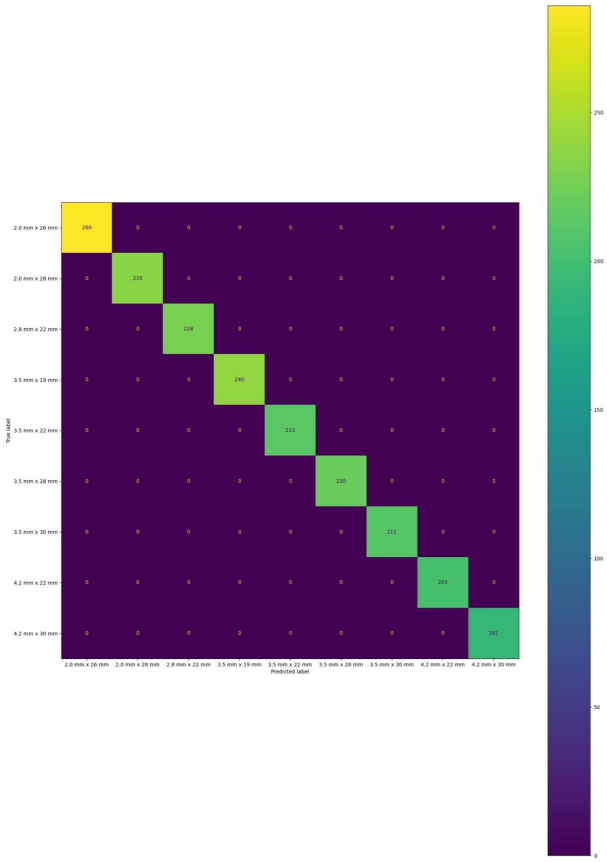

Fig. 6 - Confusion Matrix for 'BatchNorm' LeNet5 model with AdamW optimizer

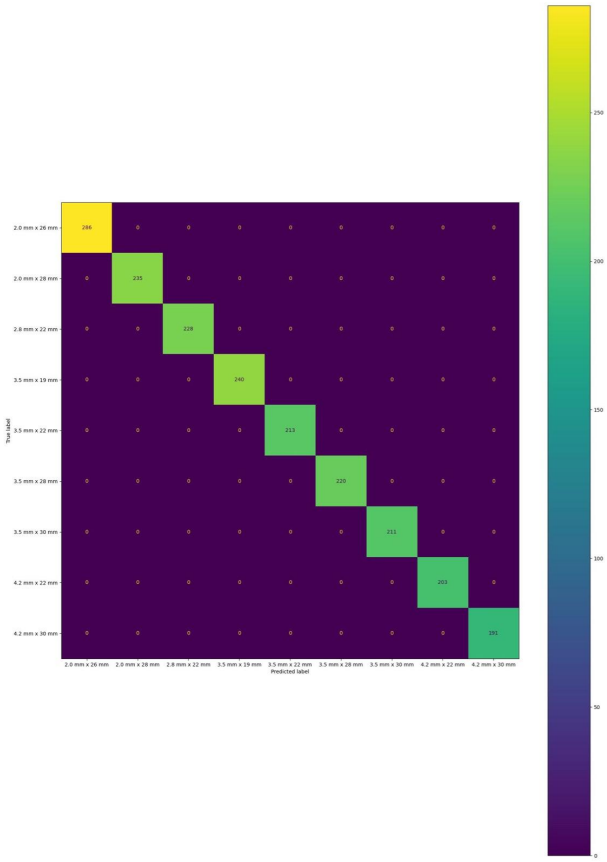

Fig. 7 - Confusion Matrix for 'BatchNorm' LeNet5 model with SGDM optimizer

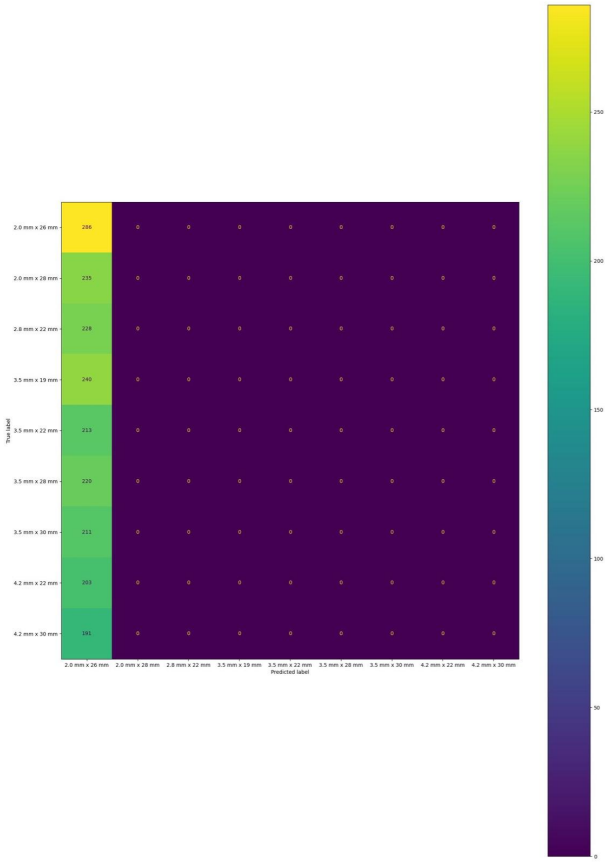

Fig. 8 - Confusion Matrix for 'BatchNorm' LeNet5 model with RMSProp optimizer

Conclusion

Overall, this was an interesting task. It can be observed that a LeNet5 architecture enhanced with Batch normalization performs well for this task. Moreover, I do believe that this can be further improved in different aspects - some of which were mentioned in the previous sections.