COMP3055 Machine Learning

Coursework

Junyu Liu
20216355

Hardware and Software Settings

Hardware for Task 1 and Task 2: CPU and GPU: Apple M1 Pro chip, 10 cores, 32G RAM;
Hardware for Task 3: CPU: Intel(R) Xeon(R) CPU @ 2.30GHz, GPU: Tesla T4 with High RAM, configured in Google Colab Pro+;
Python packages: python version 3.8, torch, torchvision, sklearn, ssl, tqdm, warnings, collections.

How to Run the Code

Open terminal/command line, load python environments with all the above-mentioned package installed, and run with "python task_1.py" (Take Task 1 as an example here). Note that when running the code, some computers' GPU may run out of memory. In such a case, please change the batch size in the code. Current settings in the code are well-evaluated on Tesla T4 with High RAM in Google Colab Pro+.

Pre-trained and test results

All results for Task 2 are stored in "SVM_results" folder (note that by running "task2.py", the newly generated results will overwrite the pre-uploaded ones). All results for Task 3 are stored in "CNN_results" folder (these results won't be overwritten when running "task3.py" since the newly generated results will be located at the root folder for easy marking).

Task 1

The Principal Component Analysis (PCA) is achieved by using the *sklearn.decompoisition* package. Different amount of information could be kept by changing the parameter *n_components* in the PCA constructor, which specifies the number of components a PCA reduction need to keep in the original data. In my code (*task_1.py*), I apply PCA reduction on the original CIFAR10 dataset through batches (i.e., 64), with *n_components* = $\{1, 2, 4, 8, 16\}$. The results for the information kept rate and corresponding *n_components* are shown in Table 1.

Table 1: Information kept in feature vectors under different *n_components* settings

n_components	1	2	4	8	16
Information Kept (%)	32.60	42.83	56.88	70.71	83.08
Noise Variance	8.01	6.91	5.38	3.92	2.64

Task 2

In this section, I will demonstrate the implementation details and results comparison of task 2.

Implemenration details

The Support Vector Machine (SVM) is achieved by using the *sklearn.decompoisition* package. Two types of kernels (i.e., *linear* kernel and *RBF* kernel) are specified when initializing the SVM classifier, each with 5 different C value setting: $C = \{0.1, 1, 10, 100, 1000\}$. The data includes 5 different PCA reduced features (in Task 1) and the original input (non-PCA). During training, in order to do 5-fold cross-validation, *cross_validate* is imported from *sklearn.model_selection* to train the SVM and output the trained model, with scoring metrics including 'precision_macro', 'recall_macro', and 'f1_macro'. After training, test data are fed into the trained SVM classifier with evaluation metrics of precision, recall, f1 values (for each class), and accuracy. All results (both training and testing) are stored as files in folder *SVM_result*.

Test Results Comparison

Linear Kernel

I select the results when $C = 1$ to showcase how different feature dimensions can affect the final results. Figure 1 shows the comparison of different feature dimensions under 4 different evaluation metrics. Generally speaking, when feature dimensions are higher, the classification results are better. However, the results of PCA *n_components* = 16 (83.08% information kept) show slightly better results than using the original images, possibly indicating that such PCA reduction removes unnecessary information from images.

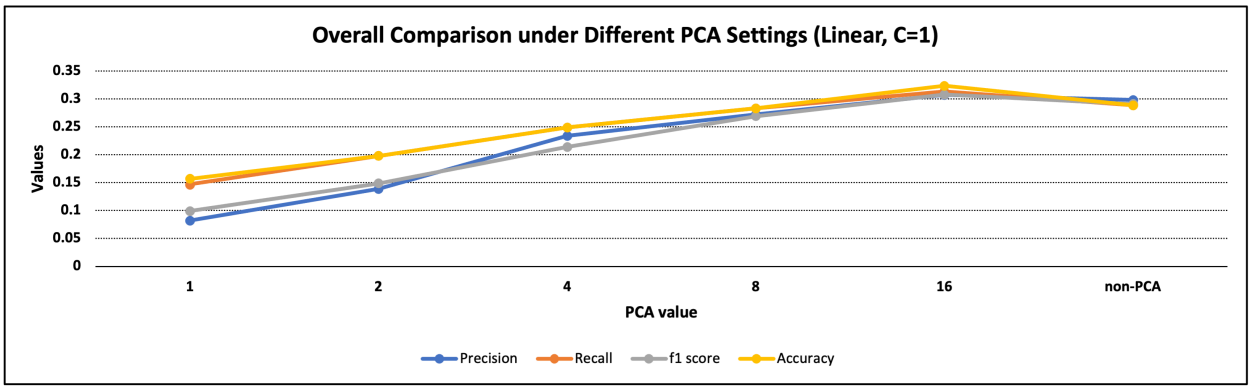

Figure 1: Overall Comparison under Different PCA Settings (Linear, C=1).

Figure 2, Figure 3, and Figure 4 illustrate the comparison of different feature dimensions in each class evaluated by precision, recall, and f1 scores. In most classes, when feature dimensions are higher, the classification results are better. However, there exist several classes that don't show such

Task 2 continued on next page. . .

increasing trends. Another thing worth pointing out is that when features are greatly reduced (e.g., *n_components* = $\{1, 2, 4\}$), several test scores are 0 in some classes.

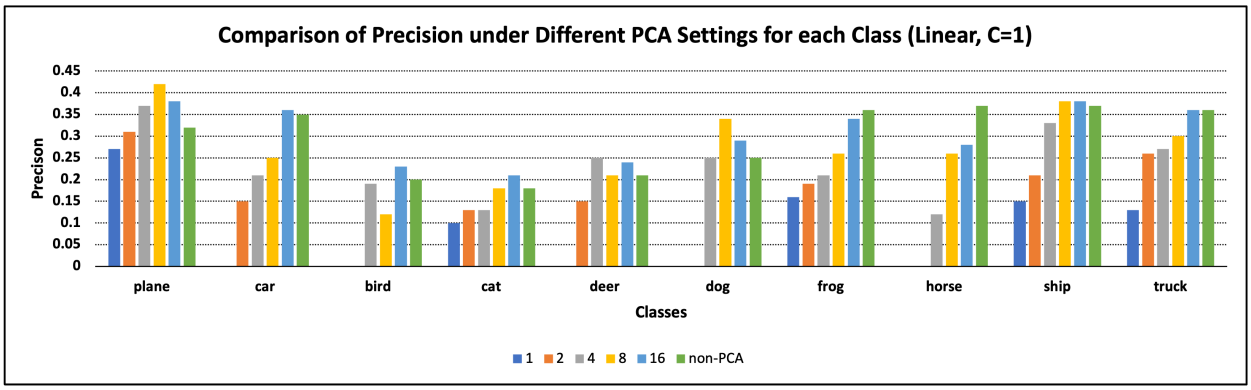

Figure 2: Comparison of Precision under Different PCA Settings for each Class (Linear, C=1).

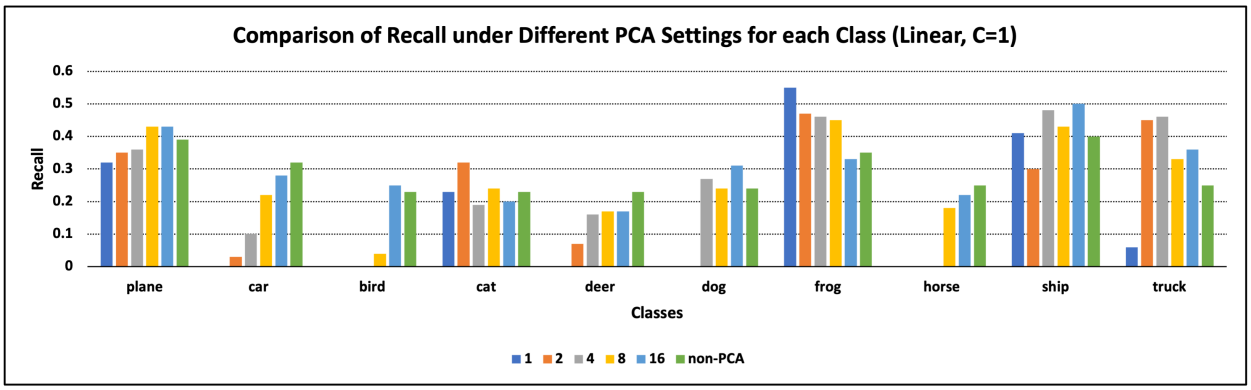

Figure 3: Comparison of Recall under Different PCA Settings for each Class (Linear, C=1).

RBF Kernel

I select the results when PCA *n_components* = 16 to showcase how different hyperparameters C can affect the final results. Figure 5 shows the comparison of different feature dimensions under 4 different evaluation metrics. The best classification results happen when $C = 1$. Generally speaking, the performance under different C values doesn't vary much.

Figure 6, Figure 7, and Figure 8 illustrate the comparison of different hyperparameters C in each class evaluated by precision, recall, and f1 scores. In most classes, classification results achieve the best results when $C = 1$. Generally speaking, the performance under different C values doesn't vary much.

Task 3

For Task 3, I designed 4 different CNN models with inspiration from several existing networks (e.g., AlexNet [1], VGGNet [2], GoogleNet [3], and MobileNet [4]). A detailed explanation of the CNN

Task 3 continued on next page. . .

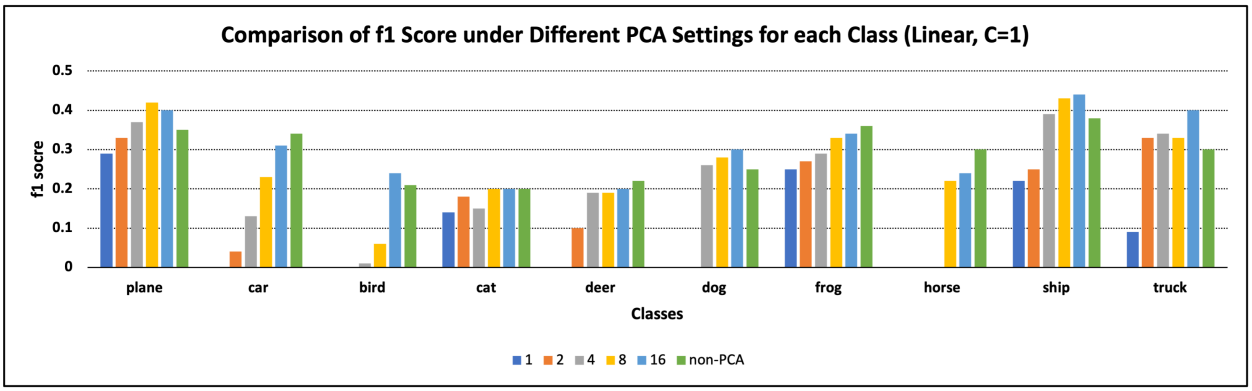

Figure 4: Comparison of f1 score under Different PCA Settings for each Class (Linear, C=1).

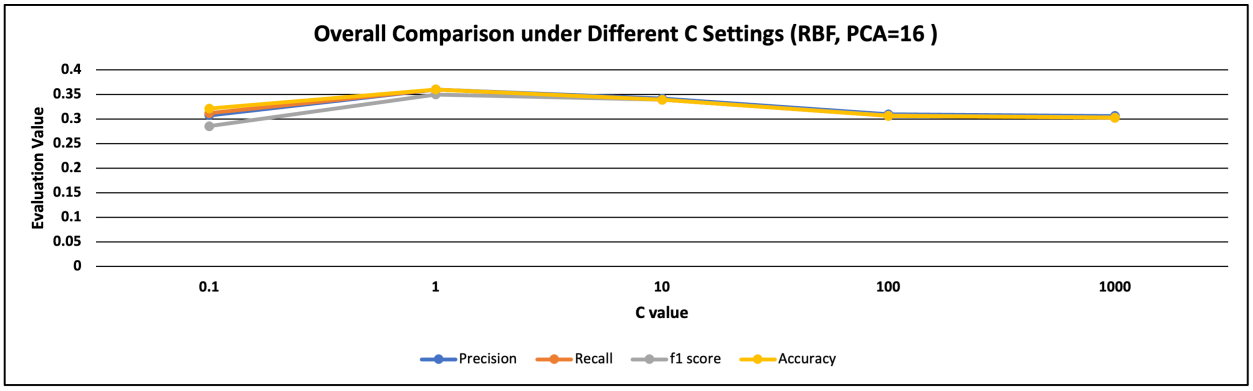

Figure 5: Overall Comparison under Different C Settings (RBF, PCA=16).

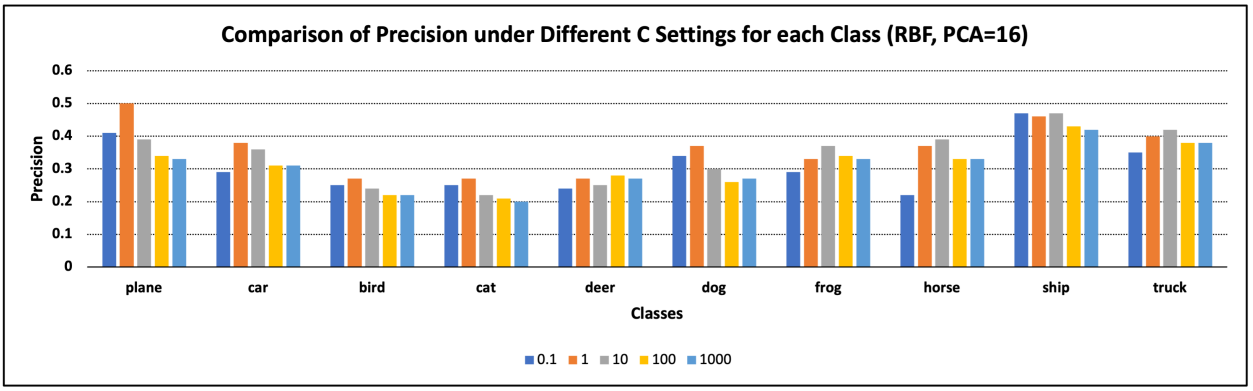

Figure 6: Comparison of precision under Different C Settings for each Class (RBF, PCA=16).

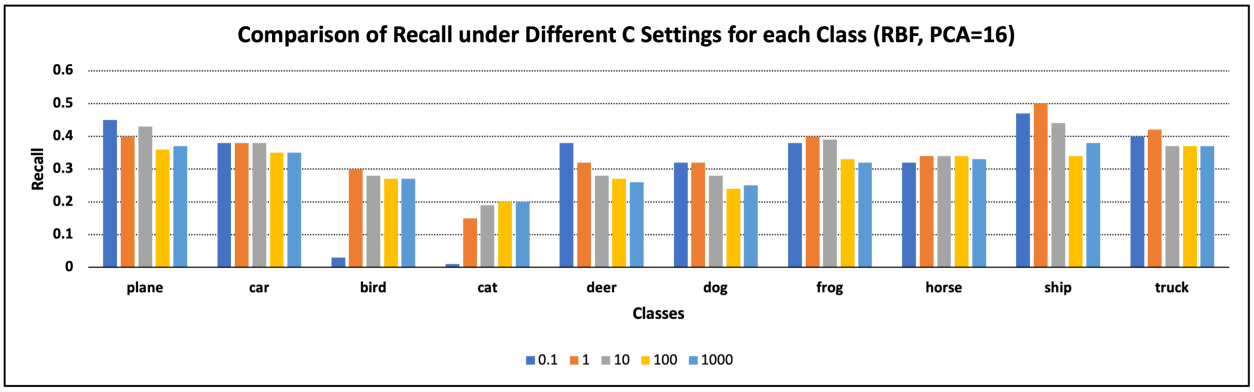

Figure 7: Comparison of recall under Different C Settings for each Class (RBF, PCA=16).

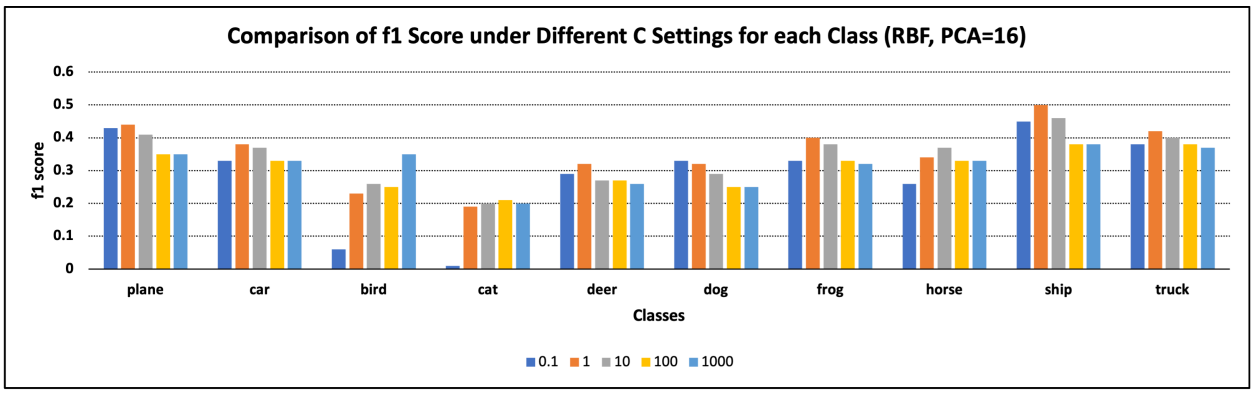

Figure 8: Comparison of f1 Score under Different C Settings for each Class (RBF, PCA=16).

Task 3 continued on next page. . .

model structures and experiment results are introduced as follows.

CNN1

The backbone of CNN2 is inspired by the design of GoogleNet [3], which introduced Inception module to distribute different number of features to different convolutional layers. I utilize this idea to construct the basic structure of CNN1 model 9 and add several critical changes to enable decent performance of the network:

1. At all convolution blocks, only the last activation function is set as ReLU, and all the others are set by LeackyReLU, to keep as much information as possible.
2. Redesigned a classifier:

a. Use two Dropout to mitigate the overfitting problem.

b. Use three connected linear layers to smoothly compute the class label from features.

c. Use ReLU and two LeakyReLU as activation functions instead of the original Softmax to enhance the classification accuracy (such improvement is observed from experiments).

3. Adjust several kernel sizes and layer numbers.

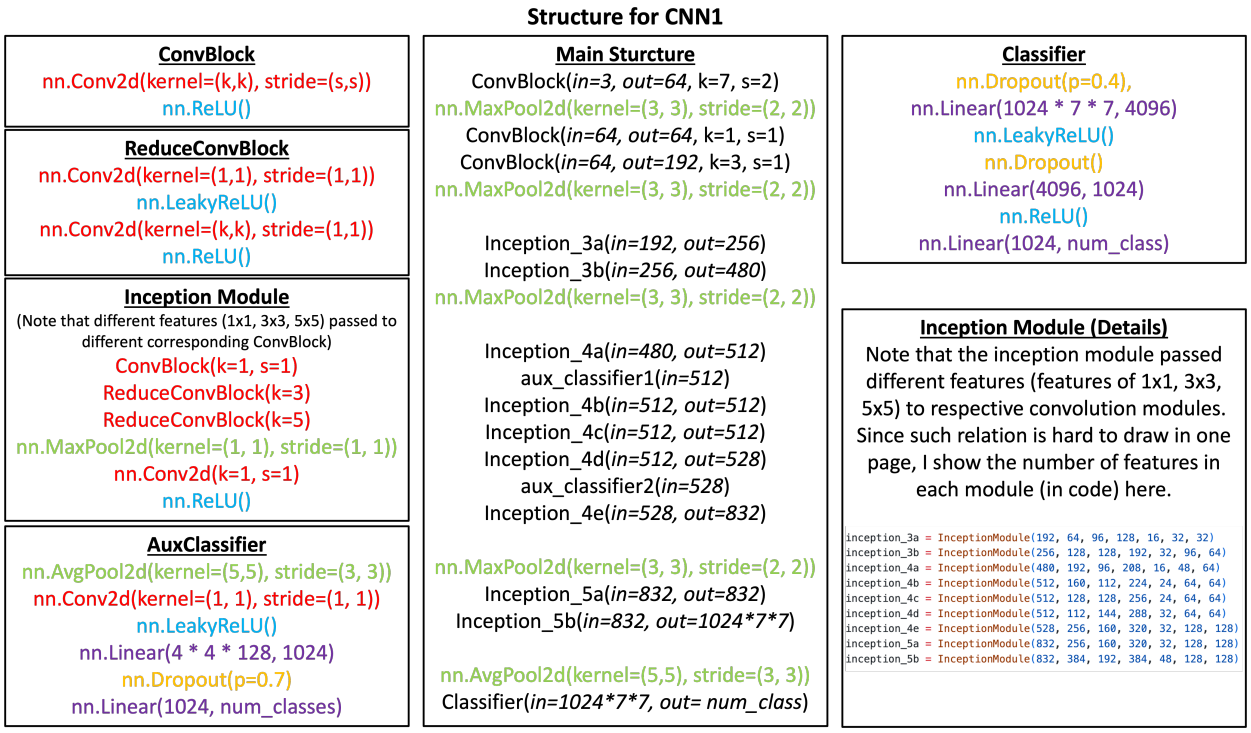

Figure 9: Structure of CNN1

CNN2

The backbone of CNN2 is inspired by the design of MobileNet [4], which introduced Depth-wise convolutional blocks to explore more features and Point-wise convolutional blocks to fine-grain features. I utilize this idea to construct the basic structure of CNN2 model 10 and add several critical changes to enable decent performance of the network:

Task 3 continued on next page. . .

1. Add batch normalization before ReLU at each convolutional block to fit the convoluted results better to ReLU.
2. Redesigned a classifier:
a. Use three Dropout to mitigate the overfitting problem.
b. Use four connected linear layers to smoothly compute the class label from features.
c. Use ReLU and LeakyReLU as activation functions instead of the original Softmax to enhance the classification accuracy (such improvement is observed from experiments).
3. Adjust several kernel sizes and layer numbers.

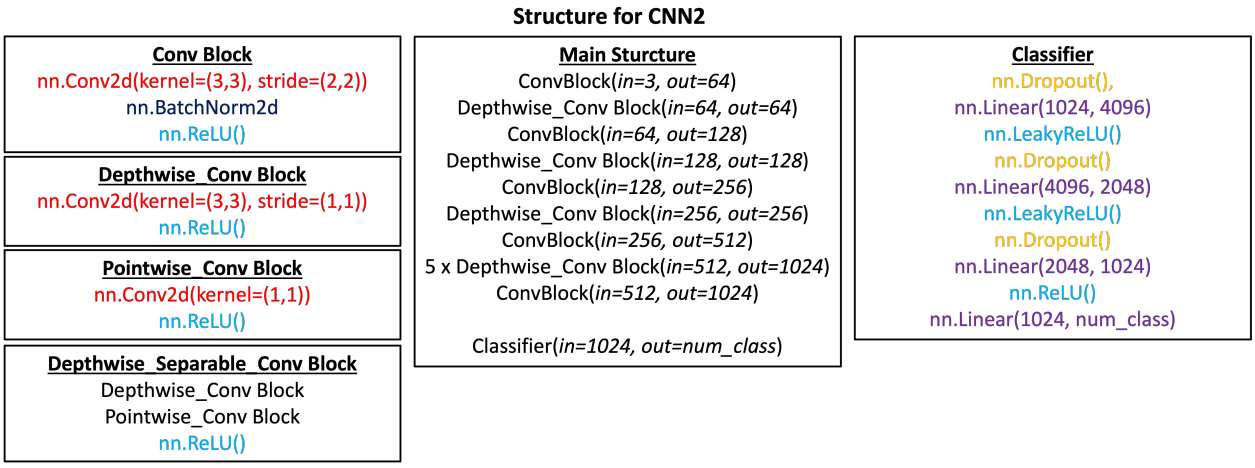

Figure 10: Structure of CNN2

CNN3

The backbone of CNN3 is inspired by the design of AlexNet [1], which is simple, fast, and straightforward to implement. I utilize this idea to construct the basic structure of CNN3 model 11 and add several critical changes to enable decent performance of the network:
1. Add batch normalization before ReLU at each convolutional block to fit the convoluted results better to ReLU.
2. Replace half of the activation function by LeackyReLU to keep as much features as possible.
3. Redesigned a classifier:
a. Use three Dropout to mitigate the overfitting problem.
b. Use four connected linear layers to smoothly compute the class label from features.
c. Use ReLU and LeakyReLU as activation functions to enhance the classification accuracy (such improvement is observed from experiments).
4. Adjust several kernel sizes and layer numbers.

CNN4

The backbone of CNN4 is inspired by the design of VGGNet [2], which is can handle large-scale image recognition with very deep learning. I utilize this idea to construct the basic structure of

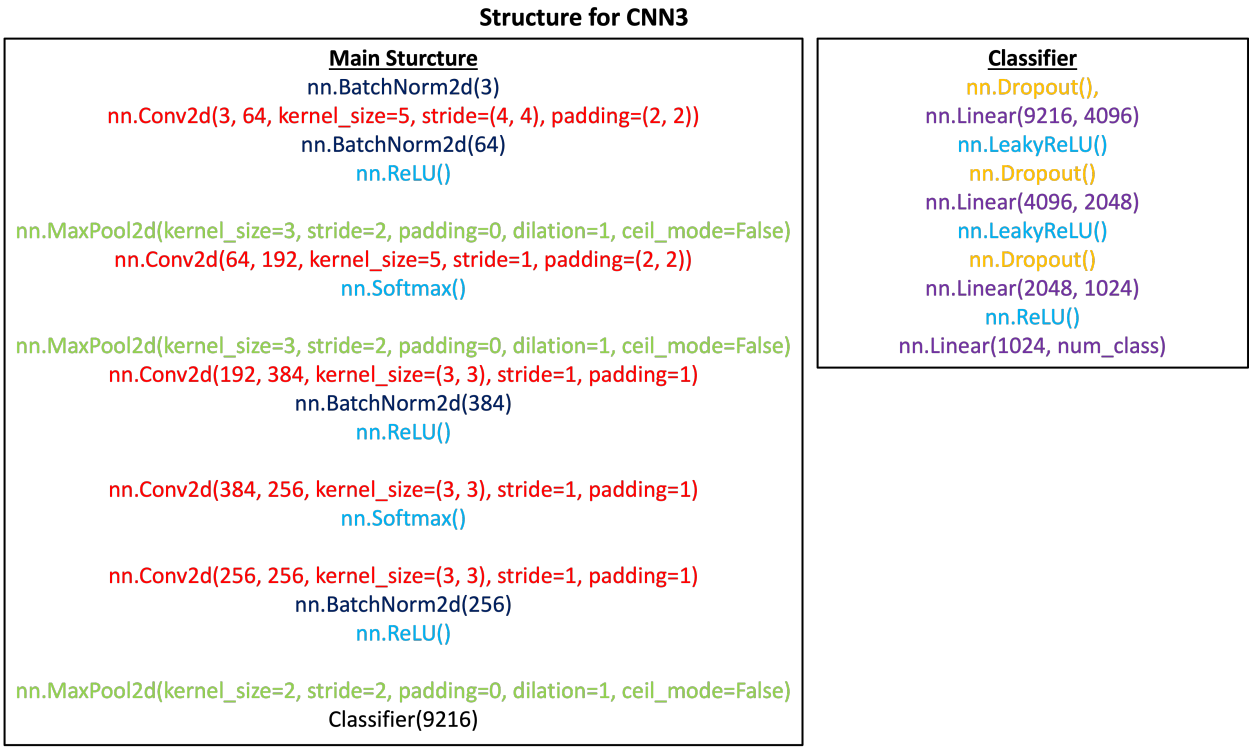

Figure 11: Structure of CNN3

CNN4 model 12 and add several critical changes to enable decent performance of the network:
1. Add pooling to reduce the feature dimension of the networks, since this network is deep and originally very slow.
2. Add batch normalization before ReLU at each convolutional block to fit the convoluted results better to ReLU.
3. Replace half of the activation function with LeackyReLU to keep as many features as possible.
4. Redesigned a classifier:
a. Use three Dropout to mitigate the overfitting problem.
b. Use four connected linear layers to smoothly compute the class label from features.
c. Use ReLU and LeakyReLU as activation functions to enhance the classification accuracy (such improvement is observed from experiments).
5. Adjust several kernel sizes and layer numbers.

Performance Comparison

An overall comparison of the performance for the 4 CNN models (on test dataset) is illustrated in Figure 13, and the per-class comparison is illustrated in Figure 14. The performance is measured by the metrics including precision, recall, and f1 score. From the results, we can conclude that CNN2 generally achieves the best performance among the 4 CNNs and CNN3 generally perform the worst.

Structure for CNN4

Main Sturcture

nn.BatchNorm2d(3)
nn.Conv2d(3, 64, kernel_size=3, padding=1)
nn.ReLU()

nn.MaxPool2d(kernel_size=2, stride=2)
nn.Conv2d(64, 128, kernel_size=3, padding=1)
nn.BatchNorm2d(128),
nn.ReLU()

nn.MaxPool2d(kernel_size=2, stride=2)
nn.Conv2d(128, 256, kernel_size=3, padding=1)
nn.LeakyReLU()

nn.Conv2d(256, 256, kernel_size=3, padding=1)
nn.BatchNorm2d(256)
nn.ReLU()

nn.MaxPool2d(kernel_size=2, stride=2)
nn.Conv2d(256, 512, kernel_size=3, padding=1)
nn.LeakyReLU()

nn.Conv2d(512, 512, kernel_size=3, padding=1)
nn.BatchNorm2d(512)
nn.ReLU()

nn.MaxPool2d(kernel_size=2, stride=2)
nn.Conv2d(512, 512, kernel_size=3, padding=1)
nn.LeakyReLU()

nn.Conv2d(512, 512, kernel_size=3, padding=1)
nn.BatchNorm2d(512)
nn.ReLU()

nn.MaxPool2d(kernel_size=2, stride=2)
Classifier(512*7*7)

Classifier

nn.Dropout(),
nn.Linear(1024*7*7, 4096)
nn.LeakyReLU()
nn.Dropout()
nn.Linear(4096, 2048)
nn.LeakyReLU()
nn.Dropout()
nn.Linear(2048, 1024)
nn.ReLU()
nn.Linear(1024, num_class)

Figure 12: Structure of CNN4

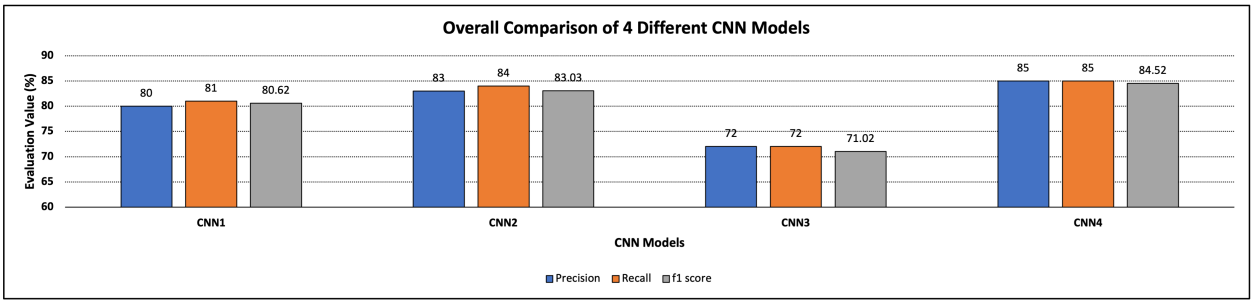

Figure 13: Overall Comparison of 4 Different CNN Models

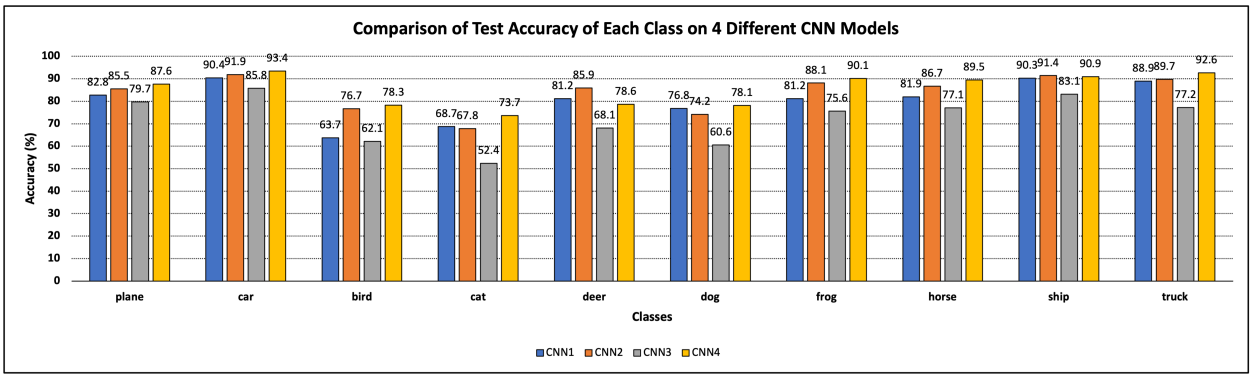

Figure 14: Comparison of Test Accuracy of Each Class on 4 Different CNN Models

Task 4

Accuracy Comparison

The evaluation results with metrics (precision, f1 scores, and recalls) for SVM are illustrated in Figure 1 and Figure 5, and the results for CNNs are illustrated in Figure 13. It's straightforward to observe that CNNs perform much better than SVM under all parameter settings. CNNs can generally achieve higher than 70% precision, f1 scores, and recalls while SVMs can only achieve less than 40%. Thus, in terms of accuracy, CNNs are better choices for image classification tasks.

Overfitting Analysis

Both trained SVM models and CNN models show different amounts of overfitting. Figure 15 shows the performance of SVM models on train dataset and test dataset. For SVM trained with linear kernel, the overfitting rate is small, at around 4.6%. For SVM trained with RBF kernel, the overfitting is negligible since the model has almost equal performance on both the train and test dataset.

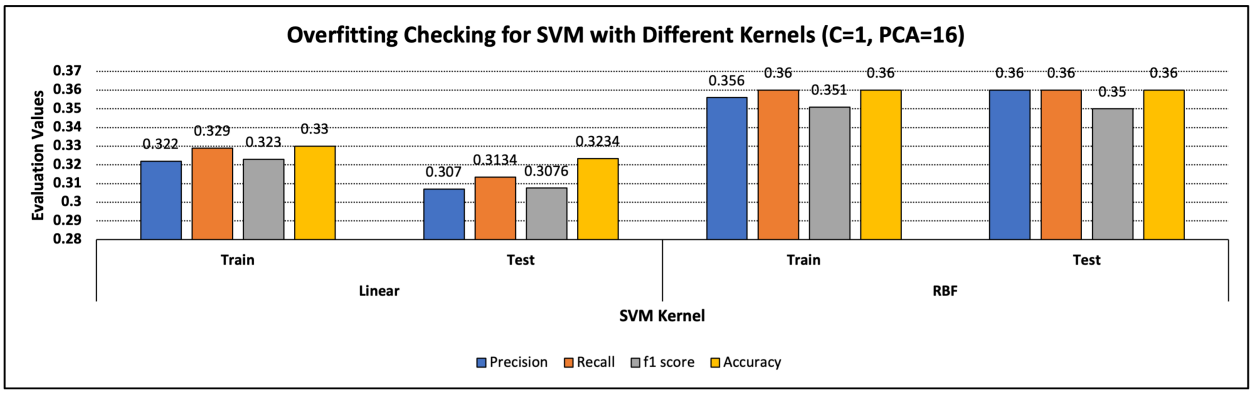

Figure 15: Overfitting Checking for 2 SVM Models

Figure 16 shows the performance of 4 CNN models introduced in Task 3 on train dataset and test dataset. All models show some overfitting since the performance on train dataset approaches 100%

while the performance on test dataset stays around 80%. CNN3 shows the worst overfitting rate, at around 25%, while CNN4 shows the best overfitting rate, at around 14%.

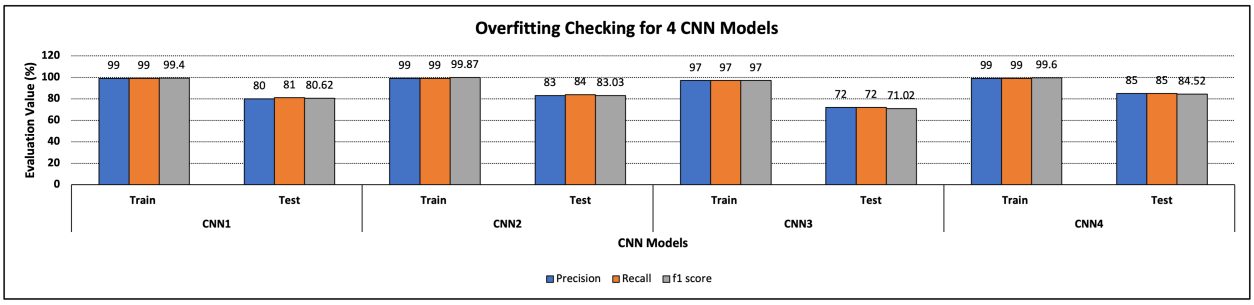

Figure 16: Overfitting Checking for 4 CNN Models

Time Consumption Analysis

Figure 17 shows the computation time for one training fold under different PCA parameter *n_components* settings. The computation time arises when the feature dimension increases (e.g., *n_components* from 1 to 16). When choosing RBF kernel for SVM, training time is shorter than the SVM with linear kernel under the same PCA parameter *n_components* setting. Figure 18 shows the computation time for each epoch of the 4 CNN models, in which CNN3 use the least computation time and CNN4 use the longest computation time. The main reason contributing to this situation is the different complexity and depth of the CNN models. Note that CNN4 achieves the best results while CNN achieves the worst, thus the tradeoff between good classification results and efficient computing is still worth exploring.

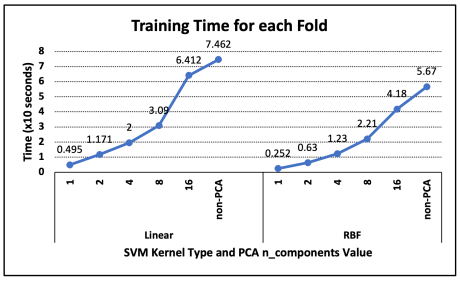

Figure 17: SVM training Time for each Fold

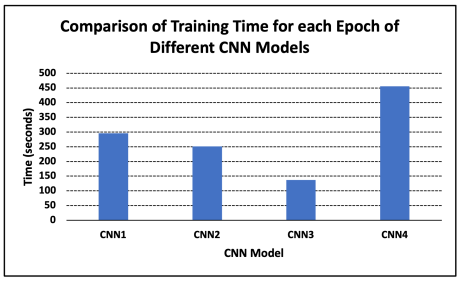

Figure 18: Comparison of Training Time for each Epoch of Different CNN Models

When comparing the training process of SVM and CNNs, one prominent difference is that CNNs are well-supported by GPU acceleration while SVMs are not. Thus, the overall training time for CNN models are comparable to the training time of SVMs. In addition, CNNs could achieve much better results than SVMs, thus CNNs are generally more suitable for image classification tasks.

Reference

[1] Krizhevsky, A., Sutskever, I., Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90.

[2] Simonyan, K., Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[3] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[4] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., ... Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.