Machine Learning Project 1

Liong Khai Jiet

1 Introduction

The Street View Hour Numbers (SVHN) Dataset [7] is a dataset compiled by the Stanford.

The SVHN is a 10-class classification problem where one for each digit 0-9. Digit '1' has label 1, '9' has label 9 and '0' has label 10. For the following experiments, the complete 73257 training set and 26032 testing set is used unless stated otherwise. For example, 10000 training samples and 5000 testing samples are randomly selected for the Support Vector Machines (SVM) experiments due to the slow computational of SVM classifiers.

Some of the images and its label are shown in Figure 2.

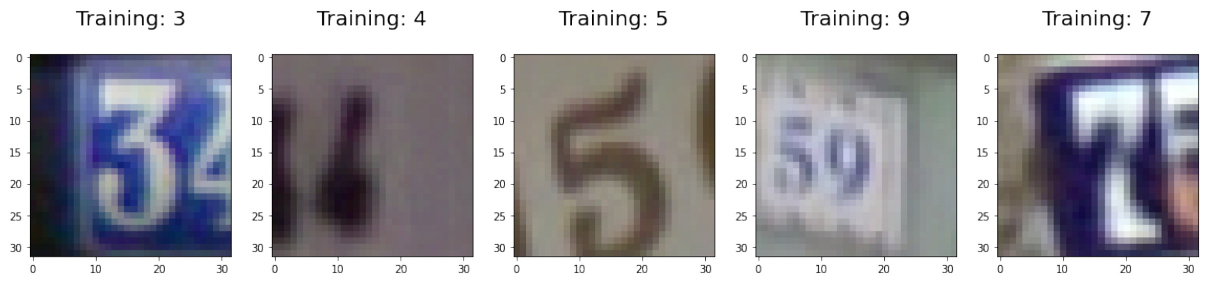


Figure 1: Sub-sample of training data with its label

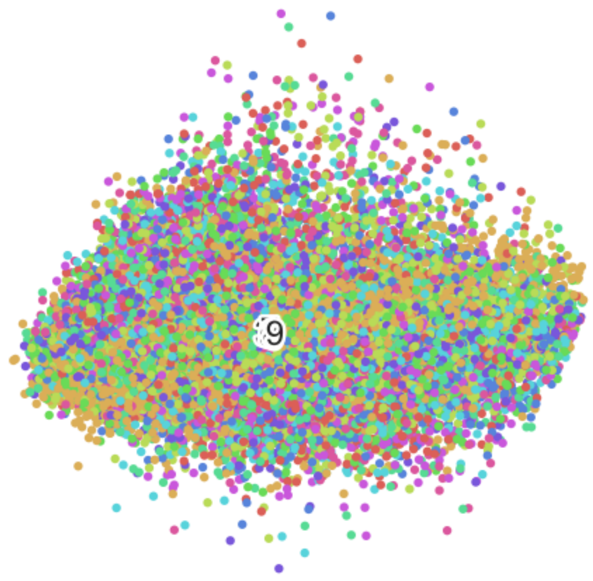


Figure 2: Principal Component Analysis for test data

1.1 Methodology

The experiment is carried out using different type of methods. In this report, the pre-processing methods, algorithm, and the results of the experiment are compiled according to the type of method used as follows. In section 3, the three types of logistic regression model is applied: vanilla logistic regression, logistic regression with L1 loss and logistic regression with L2 loss. In section 4, we use support vector machines to solve the same problem. In section 5, we use deep neural networks, specifically Convolutional Neural Network (CNN) to solve the same problem and apply techniques like fine-tuning a pretrained model and transfer learning in subsection 5.3.

1.2 Setup

The experiments are run using PyTorch Lightning [3] framework is used for the data loading and training loop implementation. Moreover, Scikit-Learn for the implementation of SVM, data preprocessing, and metrics. The code are run using a hosted Jupyter Lab in Google Colab instance with limited free tier GPU resource.

1.3 Preprocessing

The images in the dataset are with the shape of (32,32,3).

All the data are fed into a custom `pytorch.Dataset` to ease the transformation of the images with the help of the `pytorch.vision`, which are:

- `transforms.ToTensor()`. This method helps to convert the image data into Pytorch tensors for computation.
- `transforms.GrayScale()`. This method helps to convert the image data into one color channel for faster computation as shown in Figure 3.
- `transforms.Normalize()`. This method is used to normalize the image data to have 0.5 mean and 0.19 standard deviation for a faster training process.

2 Logistic Regression

Logistic regression normally used as the baseline classification algorithm for machine learning projects. It is a one-layer network which uses the logistic (or Sigmoid) function to model. However, Sigmoid is normally used for binary classification problem.

In the two class classification problem, the Sigmoid function is:

$$P(y_i = 0) = \frac{\exp(-\beta \cdot X_i)}{1 + \exp(-\beta \cdot X_i)}$$

$$P(y_i = 1) = \frac{1}{1 + \exp(-\beta \cdot X_i)}$$

In a K-class problem, Softmax has the formula of:

$$P(y_i = k) = \frac{\beta_k \cdot X_i}{\sum_{0 \le C \le K} (\exp(B_C \cdot X_i)}$$

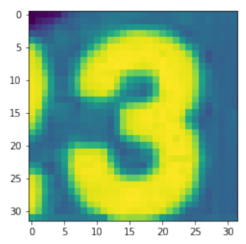


Figure 3: sub-sample of training after preprocessing

which can be rewritten as

When K = 2

$$P(y_i = 0) = \frac{\beta_0 \cdot X_i}{\exp(B_0 \cdot X_i)}$$

We can see that Softmax is a generalized version of Sigmoid. So, instead of using the sigmoid function, `torch.nn.CrossEntropyLoss()`, which has the Softmax and negative log likelihood is used as the loss function which outputs the probability of each class of the digits.

2.1 Baseline

To ensure the algorithim is correctly implemented, `sklearn.LogisticRegression` with the liblinear solver is fitted to the training data and the test score of 21.83% is used as the baseline score for our logistic regression implementation. The results are tabulated in Table 1 along with other methods.

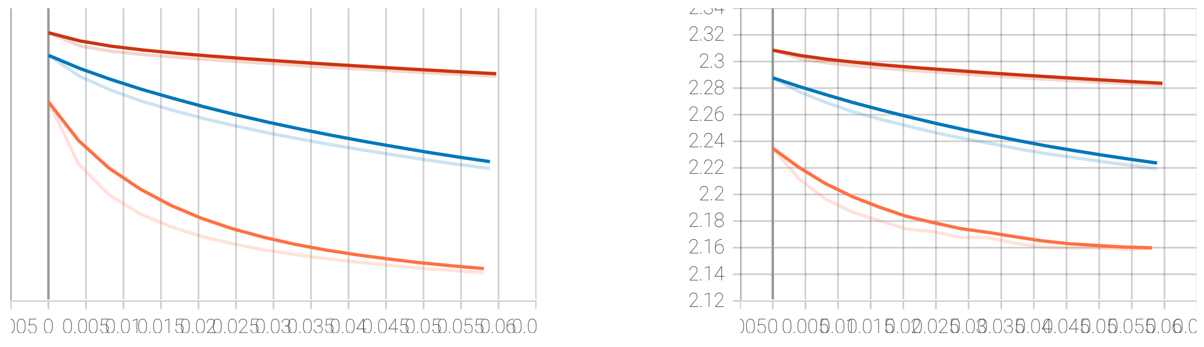
2.2 Vanilla Logistic Regression

Learning rate (α) is chosen between 0.001 - 0.00001. Figure 4 shows that α=0.001 has the best score because the model can achieve a smaller training loss and higher validation accuracy.

Besides, the 3 models are initially trained with 20 epochs but the training loss stop decreasing after epoch 15, so the early stopping technique is applied and the final training epoch is fixed at 15.

For the training batch size, we find that batch size of 32 works well with the stochastic gradient descent (SGD) because of the balance of good training speed while maintaining a lower generalization error. Besides, batch size of 32 is also suggested by the previous studies [1, 6] which showed empirical results that suitable batch size can minimize the noise exist in each data batches.

The vanilla model with α=0.001 can achieve the accuracy of **24.24%**.



(a) Training loss of 3 vanilla logistic model (b) Validation loss of 3 vanilla logistic model

Figure 4: Epoch vs Training loss and Validation loss (Red[1e-5], orange[1e-3], blue[1e-4])

2.3 Logistic regression + Ridge Loss (L2)

$$l(\beta) = ||Y - X\beta||^2 + \lambda_{L2}||\beta||^2$$

where λ_{L2} is a tunable hyperparameter.

We tested another 4 models with hyperparameter fixed at α=0.001 with different λ_{L2} value (1e-3,1e-4,1e-5,1e-6), we found that model with λ_{L2}=1e-5 can achieve the accuracy of **22.9%**.

2.4 Logistic regression + LASSO Loss (L1)

$$l(\beta) = ||Y - X\beta||^2 + \lambda_{L1}|\beta|$$

where λ_{l1} is a tunable hyperparameter.

Similarly, we tested on 4 models with hyperparameter fixed at α=0.001 with different λ_{L1} value (1e-3,1e-4,1e-5,1e-6), we found that model with λ_{L1}=1e-5 can achieve the accuracy of **24.25%**.

2.5 Results

These results in Table 1 shows that logistic regression with LASSO Loss has the best performance compared to other models. This may due to the regularization effect where more significant features are selected and the weights of other features are shrink to zero.

Table 1: Logistic Regression results

Log. Reg Model	Accuracy	Gained %
Baseline	21.83	-
Vanilla	24.24	+ 2.41
with Ridge	22.9	+ 1.07
with LASSO	24.25	+ 2.42

After training, Figure 5 shows the weights learned by the logistic model with LASSO. As we can see in the activation filters shown in Figure 5, logistic regression did a bad job in capturing the edges of the digits, where most of the images have “salt and pepper” noise and blurry shape of the digits.

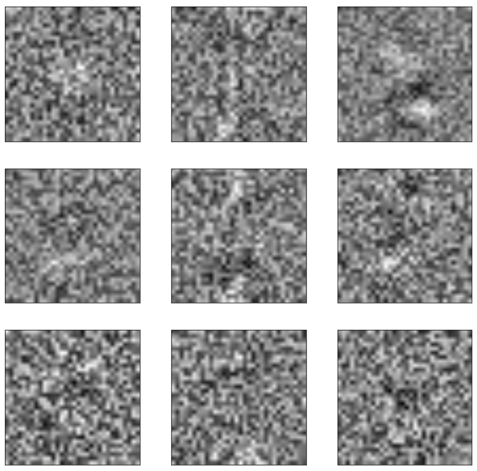


Figure 5: Learned weights after training for digits top-left to bottom-right (0 - 8)

Doing PCA as shown in Figure 6 and 7, we can see that the best logistic regression model is unable to separate the clusters of digits further from one another.

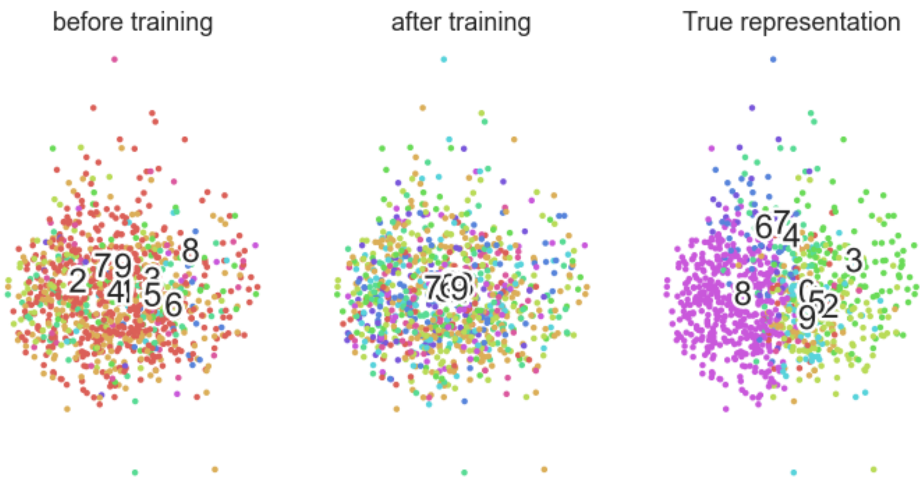


Figure 6: PCA of subset of test data for Logistic Regression Model

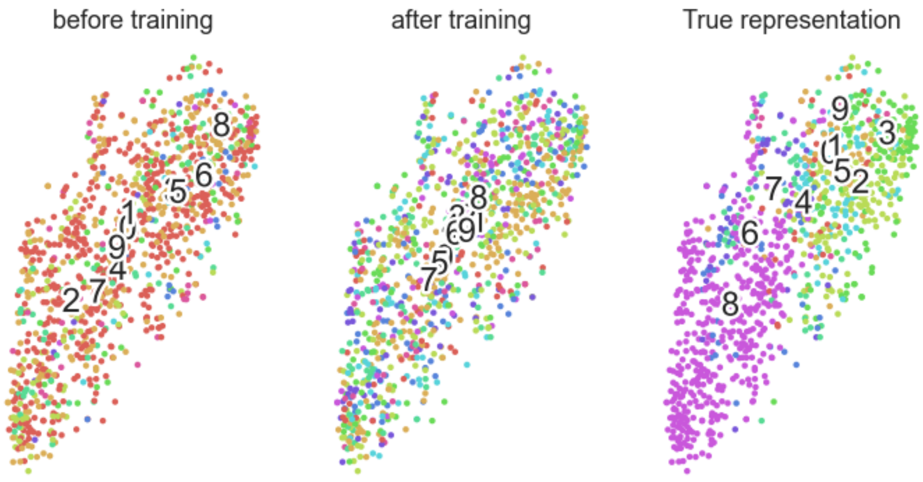


Figure 7: t-SNE of subset of test data for Logistic Regression Model

2.5.1 Logistic Regression on 3-channel image

The preprocessing `transforms.Grayscale()` is remove from the pipeline and the data are feeded into the network.

Compared to the single channel image, RGB images are harder to learn as shown by the experiment results. The results are tabulated in Table 2.

Table 2: Effect of preprocessing 3 channel image to single channel

Log. Reg Model	RGB channel?	Accuracy	Gained (%)
with LASSO	False	24.25	-
with LASSO	True	22.44	- 1.81

This may due to colors are not so relevant in digit identification as the model tries to recognize a digit via the edges and corners of the digit, so the variation added by the additional two color channels only increase the difficulty of the model to learn the feature

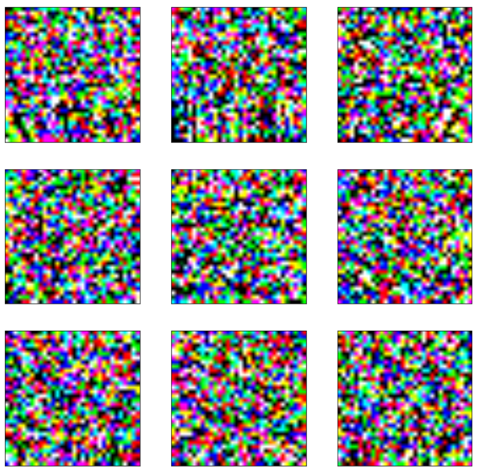


Figure 8: Learned weights after training for digits (RGB) top-left to bottom-right (0 - 8)

representations. Figure 8 shows more noise exist in the feature maps compared to the single channel feature map in Figure 5 .

3 Support Vector Machine

Support Vector Machine (SVM) is also another method for classification. As it is computational expensive with minimum model complexity of n^2 and only runs on CPU, the training data size is reduced to 5000 training samples and 2000 testing samples. The samples are randomly sampled using `sklearn.train_test_split`. The preprocessing steps are the same as stated in section 2.1 with additional preprocessing step of scaling the features within the range [0,1] as suggested by the library author [2, 5]. Standard Scaler is not used because it is very sensitive to outliers existed in the dataset.

The testing data is then plotted using Principal Component Analysis (PCA) with 2 components to reduce the $32 \times 32 = 1024$ features to plot the distributions as shown in Figure 9.

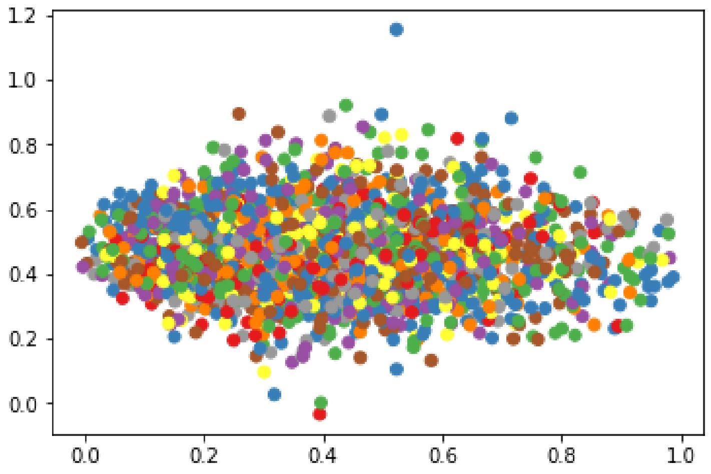


Figure 9: PCA (0.9) plot for random 2000 testing samples (n=2)

Obviously, the figure showed that this is not a linearly separable problem. Therefore, another variant of SVM are used to compare the weights learned by Linear SVM and polynomial kernel SVM.

3.1 Experiment

For linear SVM, `sklearn.LinearSVC` with a hinge loss and one-vs-all strategy is used. For comparison, we also tested with radial basis function (RBF) and polynomial kernel using the same training samples. The results of the classification accuracy is tabulated in Table 3.

Table 3: SVM results

SVM Classifier	Accuracy (%	) Train Time (s)
Linear	23.7	981
rbf kernel	55.55	466
polynomial kernel(=3)	22.85	589
polynomial kernel(=4)	46.3	467
polynomial kernel(=5)	20.5	-

The Linear SVM only achieves the testing accuracy of **23.7%**. It performs the worst with the longest training times. We can see that linear SVM is unable to solve problems that are linearly inseparable. When the data has noise and overlapping points, there is a problem in drawing a clear hyperplane without misclassifying.

In the other hand, SVM with RBF kernel outperforms the other SVM algorithms. This two kernel methods can map the data in very higher dimensional feature space using the kernel trick. In the high dimensional feature space, the classifier is able to find support vectors that can separate the different classes.

We tried a few more hyperparameters for the polynomial svm, we find that the accuracy of the classifier increases at polynomial degree 4 and decreases as the polynomial degree increases. Besides, the polynomial kernel has more hyperparameters than the RBF kernel, so it is not feasible to try other polynomial degree due to the long training times.

4 Deep Neural Network

Unlike the previous traditional machine learning techniques shown in section 3 and 4, deep neural network learn features through its ‘deeper’ hidden layer architecture, where lower layers extract low level features, followed by higher layers which extract high level features. Taking an example in the SVHN problem context, high level features could be the color of the image and overall digit where low level features is the edges found in the different parts of digits.

In this part, we will implement a Convolutional Neural Network (CNN) to solve the classification problem.

4.1 Vanilla CNN (Baseline)

CNN is a neural network that process data with grid like topology. Unlike a fully connected neural network with relatively more parameters, CNN has lesser parameter to train, which leads to faster training times and less prone to over-fitting issues with features like sparse connection and parameter sharing properties. These advantages makes the CNN is good for extracting features in image data.

The CNN Network Architecture used as feature extractor of Vanilla CNN and DANN model in section 5.3 are shown in Listing 1:

Listing 1: First verbatim

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
       ---> Conv2d-1            [32, 16, 28, 28]             448
       BatchNorm2d-2            [32, 16, 28, 28]              32
              ReLU-3            [32, 16, 28, 28]               0
            Conv2d-4            [32, 32, 28, 28]           4,640
       BatchNorm2d-5            [32, 32, 28, 28]              64
              ReLU-6            [32, 32, 28, 28]               0
            Conv2d-7            [32, 64, 28, 28]          18,496
              ReLU-8            [32, 64, 28, 28]               0
            Conv2d-9           [32, 128, 28, 28]          73,856
      BatchNorm2d-10           [32, 128, 28, 28]             256
             ReLU-11           [32, 128, 28, 28]               0
            Conv2d-12          [32, 256, 28, 28]         295,168
             ReLU-13           [32, 256, 28, 28]               0
       ---> Conv2d-14           [32, 16, 28, 28]           4,112
        MaxPool2d-15            [32, 16, 14, 14]               0
          Dropout-16            [32, 16, 14, 14]               0
           Conv2d-17            [32, 32, 12, 12]           4,640
             ReLU-18            [32, 32, 12, 12]               0
           Conv2d-19            [32, 64, 10, 10]          18,496
      BatchNorm2d-20            [32, 64, 10, 10]             128
             ReLU-21            [32, 64, 10, 10]               0
           Conv2d-22             [32, 128, 8, 8]          73,856
             ReLU-23             [32, 128, 8, 8]               0
           Conv2d-24             [32, 256, 6, 6]         295,168
      BatchNorm2d-25             [32, 256, 6, 6]             512
             ReLU-26             [32, 256, 6, 6]               0
      ---> Conv2d-27              [32, 10, 8, 8]           2,570
        AvgPool2d-28              [32, 10, 1, 1]               0
          Flatten-29                    [32, 10]               0
================================================================
    (criterion): CrossEntropyLoss()
```

*The arrow (—¿) indicates the layer used for visualization.

Two vanilla CNN are trained for 5 epochs respectively where one is trained with a fixed learning rate of 0.001 and the other is trained with learning rate of 0.01 with learning rate decay of 0.01. The results are tabulated in Table 4.

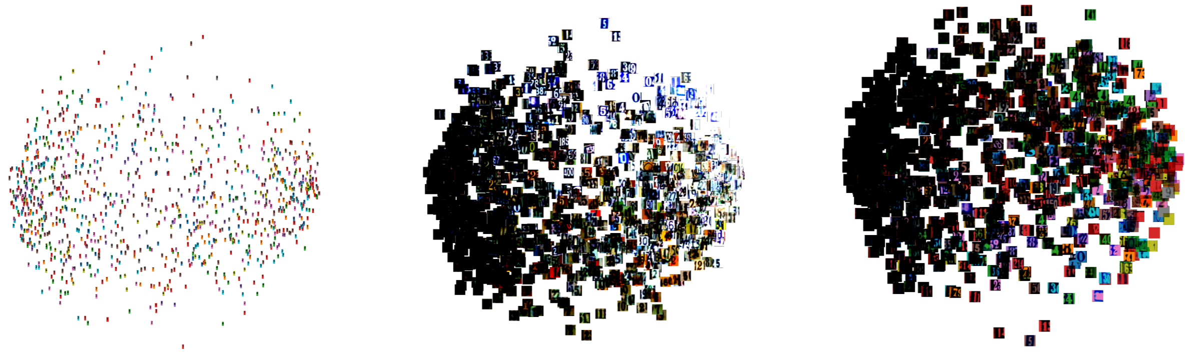
Table 4: Classification accuracy for various cnn model

Model	Description	Batch Size	Train Epoch	Train Time (s)	Accuracy (%)
A	-	93.5	32	3	107
B	with LR decay	32	3	104	94.31
C	with LR decay	128	3	57	94.41
D	with LR decay	256	3	35	94.08
E	with LR decay	512	3	42	92.86

Selecting a batch size is important to balance between reliable gradient and faster training times without hurting the model performance. Our findings showed that batch size with 128 not only improves the model performance but also increases the convergence rate by 54%. The findings are also supported by previous studies [8], which suggested that the training batch size should be set between 64 to 512 with the 2 of power n.

4.1.1 Feature Visualizations

Before training, the PCA and t-SNE of the randomly selected 1024 points are plotted as shown in Figure 10.



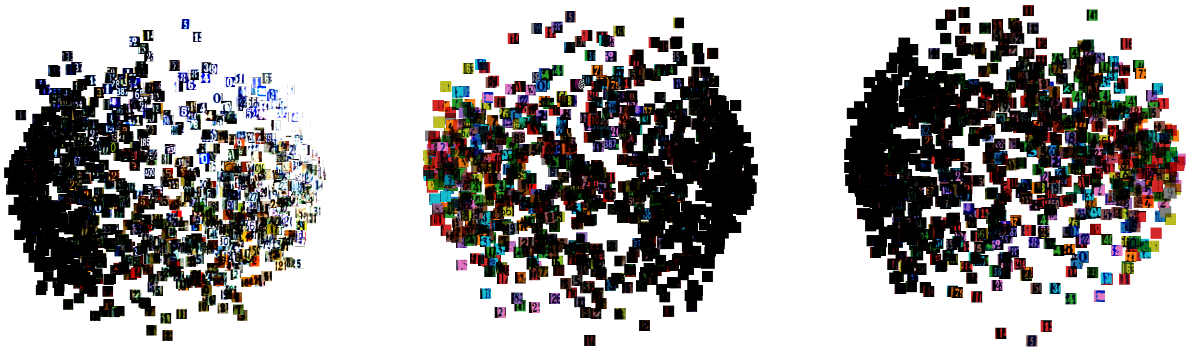
(a) Feature representation before training (colored labeled) (b) Feature representation before training (img) (c) Features representation after training (img)

Figure 10: Feature representation (PCA) of randomly selected 1024 points

Next, we dissect the vanilla CNN network into 3 parts, namely first, middle and last to gather the CNN layer in each part. The layers chosen are `layer-1`, `layer-14` and `layer-27` as labelled in the CNN architecture showed above in Listing 1. For convention, we named the layers `First`, `Middle` and `Last` layer. We then plot the graphs layer by layer to visualize what does each layer is responsible during training.

4.1.2 First Layer

Before training, `First` cov2d layer tries to cluster the images by the general feature (in this case image brightness) where dark images are clustered in the left in Figure 11a and as the network converges, the “darker” images as shown in Figure 12a are distributed in the feature space as shown in Figure 11c.



(a) Epoch 1 (var = 30.1%) (b) Epoch 2 (var = 46.7%) (c) Epoch 3 (var = 42.9%)

Figure 11: PCA for feature representation captured by `layer-1`

(a) Dark figure cluster (b) White figure cluster

Figure 12: Cluster samples captured by `layer-1`

4.1.3 Middle Layer

`Middle` cov2d layer tries to detect features of the digits like the edges and curves in the images. For example,the top nearest Euclidean points digit "6" in original space shows numbers like "3" and "2". As training epoch increases, the digits with similar features like edge and pointy shape are getting closer in the euclidean space as shown in Figure 14.

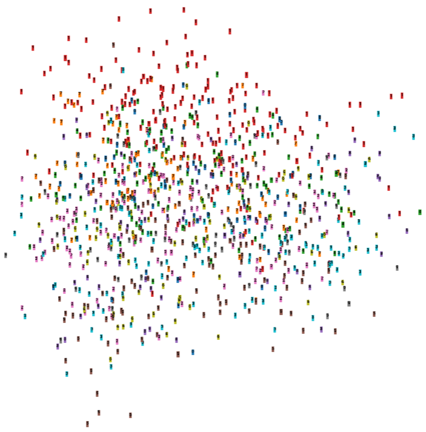


Figure 13: PCA for `layer-14` in 2D

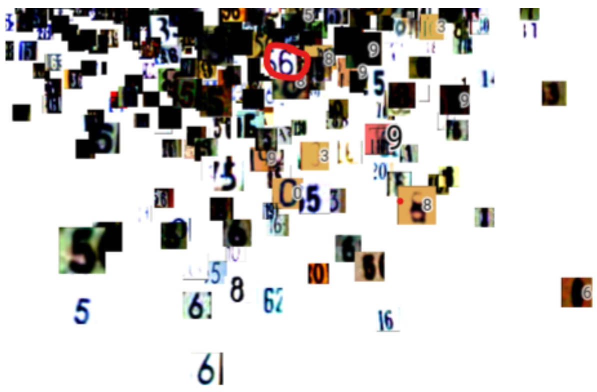
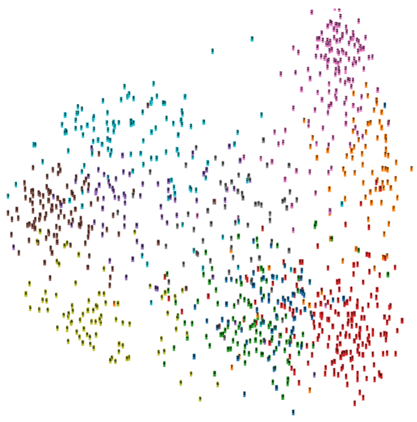


Figure 14: Sample cluster (Digit Chosen:6; Neighbour numbers: 8,5,9)

4.1.4 Last Layer

Features captured by the `Last` layer formed 10 clusters and the network is able to classify the digits with more accuracy.


(a) PCA for middle layer in 2D

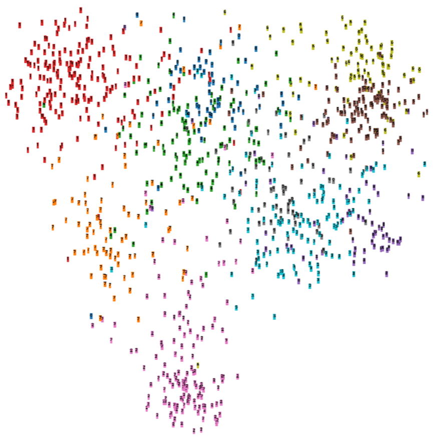

(b) PCA for last layer in 2D

Figure 15: PCA for feature representation generated by `layer-27`

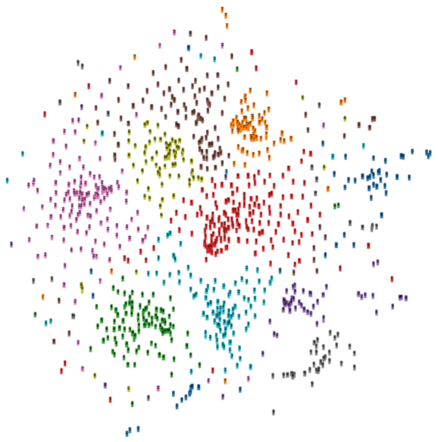


Figure 16: t-SNE of top-2 feature embedding extracted by the network

4.2 Finetuning with AlexNet

The official ResNet model is designed for the classification task on RGB image dataset, e.g. ImageNet. The main difference between ImageNet and SVHN is shown in Table 5:

Table 5: Difference between ImageNet and SVHN

Input Image	ImageNet	SVHN
Channels	3	1
Resolution	224 x 224	28 x 28
Classes	1000	10

Therefore, we did some modifications on the original networks to adapt for SVHN dataset. Specifically, we modified:

- Resize the input dimension from 224 x 244 to 28 x 28
- Change the last fc layer's output features from 1000 to 10

The implementation of loss function Cross Entropy Loss is similar to previous experiments unless specified otherwise. Below are the descriptions of the changes made in an attempt and the results are tabulated in Table 6.

4.2.1 Attempt #1

The initial strategy is to use the pretrained ResNet18 (backbone) as a feature extractor where the last layer of the backbone is fully connected to a dense classifier layer with 10 output. We freeze the backbone and only train the classifier layer. The model performs poorly both on the training data and validation data at about 33% accuracy.

An educated guess is the classifier network parameters is insufficient to learn the features of the digits.

4.2.2 Attempt #2

With the same setting, one more dense layer with 50,100,500 units is added between the backbone and the classification layer with a dropout layer with $p = 0.25$ respectively and all networks show similar performance as Attempt #1.

This remove the possible reason of not having learning capacity of the classifier layer.

4.2.3 Attempt #3

With the similar setting with Attempt #1, the backbone layer is unfreeze and the whole network is trained. The network performance improves significantly and is able to reach a test results of **85%**. However, the Figure 17 shows the network is prone to overfit the training data. To solve this problem, the early stopping strategy is used to choose the optimal epoch, which is 6 in this experiment.

4.2.4 Attempt #4

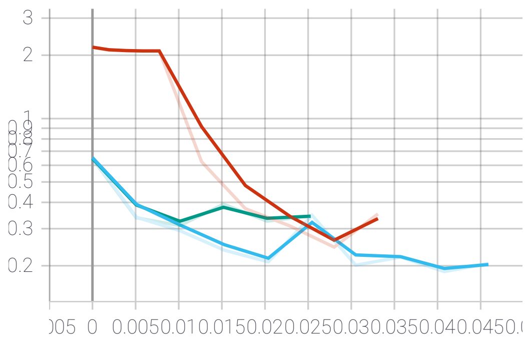
Model A is trained with a similar setting but with the addition of a learning rate decay scheduler with step size = 1 and $\gamma = 0.1$. The performance of the model is negatively impacted with only 55% accuracy on the test dataset.

4.2.5 Results

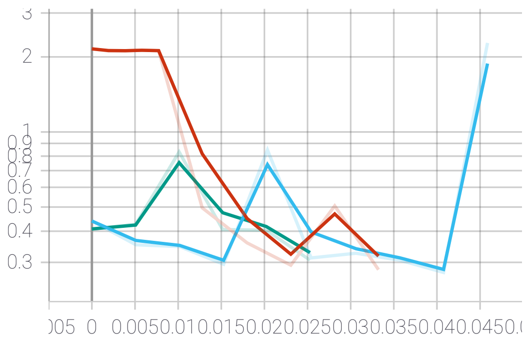
Table 6 shows the results fine-tuned Alex net.

Table 6: Fine tune ResNet18 results

Model	Description	Accuracy (%)	Train Epoch	Train Time (s)
A	Unfreeze after epoch 5	92.4	10	119
B	Unfreeze at epoch 0	80.49	10	165
C	Unfreeze at epoch 0	91.95	6	90



(a) Training loss plot of 3 finetuned ResNet18



(b) Validation loss plot of 3 finetuned ResNet18

Figure 17: Epoch vs Training loss and Validation loss (Red[1e-5], orange[1e-3], blue[1e-4])

4.3 Transfer learning from MNIST

MNIST dataset is a dataset with similar in flavour to the SVHN dataset, instead of google images of house number digits, MNIST consist of handwritten digits. Although the two dataset are from two distribution, the two classification task are similar in terms of having 10 classes of digits ranging from 0 to 9 and each training data has only one label.

We applied the domain adaptation technique from transfer learning to transfer the learned knowledge from the MNIST dataset to detect digits in the SVHN dataset. We applied the Domain Adaptation Neural network (DANN) [4] to this problem.

The `domain_train` and `domain_label` is used to train the domain discriminator module in the DANN model while the `task_train` is used to train the emotion classifier. Last but not least, `task_test` is the testing subject for that particular model. The algorithm for splitting the classification data and domain discriminator data is shown in Algorithm 1.

4.3.1 Preprocessing

Transfer learning require the data to be as possible, so we need to resize the 32 x 32 SVHN datasets to the size of 28 x 28 pixel to be same size with MNIST dataset. The number of SVHN training data is also reduced to 500, 200, 100 and 10 to mimic the lack of data in target domain.

The other preprocessing steps are same as stated in section 2.1. We also need to label the training data according to their source, i.e. source domain is labelled as 0 and target domain is labelled as 1. Next, the dataset is then distributed according to table 7.

Table 7: training and testing data distribution for DANN Dataset

DANN Dataset	MNIST (train)	SVHN (train)	SVHN (test)	Total
Training	60000	1000	-	60100
Test	-	-	26032	26032

Algorithm 1 *GetData*(sub_i)

```
1: $\text{train}_{domain}$ = list()
2: $\text{label}_{domain}$ = list()
3: $\text{train}_{task}$ = list()
4: $\text{label}_{task}$ = list()
5: $\text{test}_{task}$ = list()
6: **for** $i$ in sub **do**
7:     append(sub[i]["data"]) to $\text{train}_{domain}$
8:     **if** $i = sub_i$ **then**
9:         append(sub[i]["data"]) to $\text{task}_{test}$
10:        append [1] to $\text{label}_{domain}$
11:    **else**
12:        append sub[i]["data"] to $\text{task}_{train}$
13:        append [0] to $\text{label}_{domain}$
14: return $\text{domain}_{train}$, $\text{domain}_{label}$,
15: $\text{task}_{train}$,$\text{task}_{test}$
```

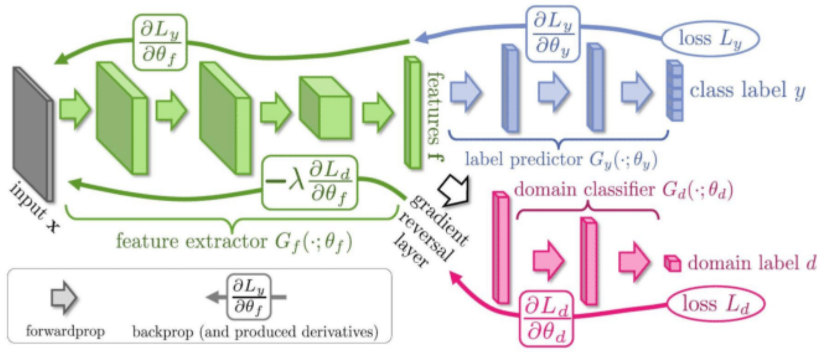
4.3.2 DANN network

DANN model is a type of inductive transfer learning model which is normally used in problems where source domain has much labelled data and target domain has lesser labelled data, where both data comes from similar but different distribution. The model architecture is illustrated in Figure 18.

The idea of this algorithm composed of two sub network modules:

1. A feature extractor network to extract the important features of the digits from the source domain (main task).

2. A domain discriminator network to differentiate between the the subjects from source and target domain.



$\frac{\partial L_y}{\partial \theta_f}$
$\frac{\partial L_y}{\partial \theta_y}$
loss L_y
class label y
features $\mathbf{f}$
label predictor $G_y(\cdot;\theta_y)$
$-\lambda\frac{\partial L_d}{\partial \theta_f}$
domain classifier $G_d(\cdot;\theta_d)$
input $\mathbf{x}$
feature extractor $G_f(\cdot;\theta_f)$
gradient reversal layer
domain label d
loss L_d
$\frac{\partial L_d}{\partial \theta_d}$
forwardprop
backprop (and produced derivatives)

Figure 18: DANN architecture. Img taken from paper[4]

This approach is inspired by the concept from domain adaptation, if the domain discriminator cannot differentiate whether the data distribution are from source or target, then the effective domain transfer has been achieved. We wish to have a feature representation which the domain discriminator cannot distinguish between the source and target domains. So the main trick here is to introduce a reverse gradient w.r.t to the loss function in the back propagation algorithm of the domain discriminator. Ideally, the domain discriminator converges when the domain accuracy equals 50% which means it can no longer distinguish the distribution of feature learnt.

In the other hand, the features extracted by the feature extractor is feed into the label predictor network, which is used to predict the digits from the features learnt.

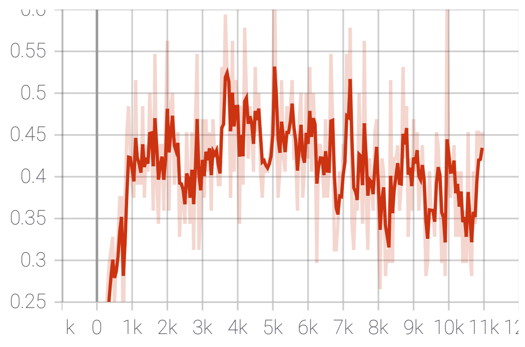
4.3.3 Experiment

The learning rate λ is chosen from the range between $1e-2$ to $1e-6$, we find that $\lambda = 1e-3$ works the best. For the domain discriminator, the learning rate is chosen to be $1e-6$ whereas the learning rate of feature extractor fixed at $1e-3$. As for the control subject, the same DANN network where the domain discriminator is discarded, is trained using the same "small" SVHN training dataset to compare the performance of both DANN and control network.

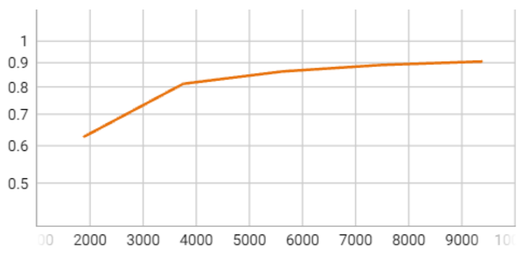
For the DANN network, we used all the MNIST dataset (60000 samples). We run a series of experiment to determine the best hyperparameter such as ratio of 'small SVHN' dataset and number of epochs.

4.3.4 Results

However, this method did not work as well as expected although both the domain discriminator network and label predictor are working as intended. The domain classifier accuracy oscillate around the 0.45 value and validation accuracy increases as shown in Figure 19.



(a) Domain classifier accuracy



(b) Label predictor validation acc

Figure 19: DANN training graphs

Table 8: DANN results

Model	SVHN Data Size	CNN Accuracy (%)	DANN Accuracy (%)
A	50	28.91	26.41
B	100	41.12	33.51
C	200	52.66	41.69
D	500	64.18	55.97

The results in Table 8 showed the DANN network not only has no improvements over the vanilla CNN. This maybe reason of this findings could be the gap between the domain distributions of the MNIST and SVHN dataset are quite similar, so the introduction of domain discriminator caused the network to face negative transfer problem. We also attempted to reduce the network layer of the feature extractor but this show similar results. The experiments could not be continued due to limitation of GPU resources.

5 Conclusion

In this project, we implemented the traditional machine learning techniques like Logistic Regression and Support Vector Machine to classify the images in SVHN dataset. The best accuracy score we got is from the RBF SVM which achieve the accuracy of 55%. Then, we introduced deep neural network to the same problem with three different type of deep learning methods, namely vanilla CNN, Transfer Learning and fine-tuning a pretrained Alexnet model. We managed to achieve a better results with 94.41% accuracy using the vanilla CNN.

Our findings show deep learning methods have relatively better performance especially with availability of large datasets and powerful computing hardware.

References

[1] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[2] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[3] William Falcon and Kyunghyun Cho. A framework for contrastive self-supervised learning and designing a new approach. *arXiv preprint arXiv:2009.00104*, 2020.

[4] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

[5] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification, 2003.

[6] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[7] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[8] Pavlo M Radiuk. Impact of training set batch size on the performance of convolutional neural networks for diverse datasets. *Information Technology and Management Science*, 20(1):20–24, 2017.