Nathan Loo, Dylan Tanaka, Taiki Yamashita, Zihao Xia

COMPSCI 178

March 18, 2025

Digit Recognition (SVHN)

Github: https://github.com/nathanloo87/cs178-project

Introduction

In this project, we trained a Convolutional Neural Network (CNN) to recognize street-view house numbers using the SVHN dataset. The goal was to build a deep learning model that could accurately classify digit images into one of ten categories (0-9). We used **PyTorch**, a popular machine learning framework, and performed data preprocessing, model training, and evaluation.

To compare deep learning with traditional machine learning methods, we also implemented a **Support Vector Machine (SVM)** classifier using **libSVM,** a **Naive Bayes Classifier,** and a **RandomForest Classifier,** which are encapsulated in **Scikit-Learn**. We also explored some other models like the **Gradient Boosting** Model from the XGBoost Python Library.

Convolution Neural Network (Taiki Yamashita)

Step 1: Loading and Preprocessing the Dataset

The dataset we used was **SVHN (Street View House Numbers)**, which contains digit images cropped from real-world street signs. We loaded the dataset using PyTorch's `torchvision.datasets.SVHN`. Since the data was stored in `.mat` format, we made sure to properly extract and transform it.

- We used `ToTensor()` to convert images into PyTorch tensors.
- We applied `Normalize((0.5,), (0.5,))` to scale the pixel values between -1 and 1.
- We created DataLoaders (`train_loader` and `test_loader`) to efficiently feed data into the model during training and evaluation.

Step 2: Building the CNN Model

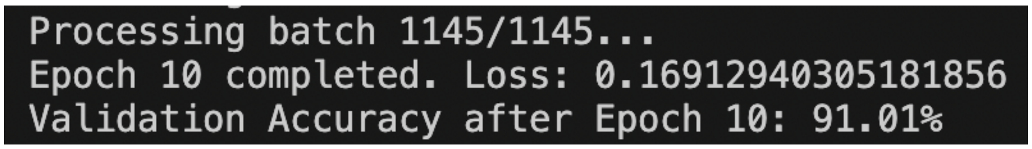
We designed a **Convolutional Neural Network (CNN)** with three convolutional layers followed by fully connected layers to process the images. The architecture was as follows:

1. **Three convolutional layers** with ReLU activation and max pooling to extract image features.
2. **Fully connected layers (FC layers)** to classify the extracted features.
3. **Dropout layer** to prevent overfitting.

Step 3: Training the Model

We trained the CNN model for **10 epochs** using **CrossEntropyLoss** and the **Adam optimizer**. During training:

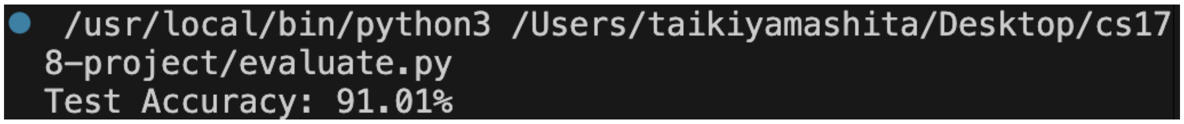
- The model learned by updating its weights after every batch.
- We kept track of the loss to measure how well the model was improving.
- After each epoch, we evaluated the model on the validation dataset.



```
Processing batch 1145/1145...
Epoch 10 completed. Loss: 0.16912940305181856
Validation Accuracy after Epoch 10: 91.01%
```

Step 4: Evaluating the Model

After training, we tested the model using the test dataset to check its accuracy. We loaded the trained model and measured the number of correct predictions:



```
● /usr/local/bin/python3 /Users/taikiyamashita/Desktop/cs17
 8-project/evaluate.py
 Test Accuracy: 91.01%
```

Results and Conclusion

After completing the training and evaluation, our CNN model achieved a **test accuracy of 91.01%**. This means the model correctly predicted the digit in **91% of test images**, showing that it effectively learned to recognize digits from street signs.

Key Takeaways:

- Using a **CNN** allowed us to efficiently extract image features and classify digits.

- The **Adam optimizer and dropout layers** helped improve learning and prevent overfitting.
- **Normalization and data transformations** were crucial in stabilizing training.

Future Improvements:

- Try using a **deeper CNN model** (e.g., ResNet) for better accuracy.
- Experiment with **data augmentation** to make the model more robust.
- Fine-tune **hyperparameters** like learning rate, batch size, and dropout rate.

Naive-Bayes Classification (Nathan Loo)

Step 1: Loading and Preprocessing the Dataset

Using the same dataset, but instead by manually parsing the data, we had to introduce gray scaling and storing those images to use for this model using numpy to modify RGB values of each picture and to store them in a numpy array.

```
train_data = loadmat('train_32x32.mat')
test_data = loadmat('test_32x32.mat')
X = np.moveaxis(train_data['X'], -1, 0)
y = train_data['y'].flatten()
```

- Manually retrieving the data from the MAT files using scipy.io

```
X_gray = np.dot(X[..., :3], [0.2989, 0.5870, 0.1140])
```

- Gray scales images manually using the 3 values on the right in place of [R, G, B] respectively

```
X_hog = np.array([hog(img, pixels_per_cell=(8,8),
cells_per_block=(2,2)) for img in X_gray])
```

- Then creating histogram of gradients (HOG) array for each image in the gray scaled array

Step 2: Model Training and Prediction

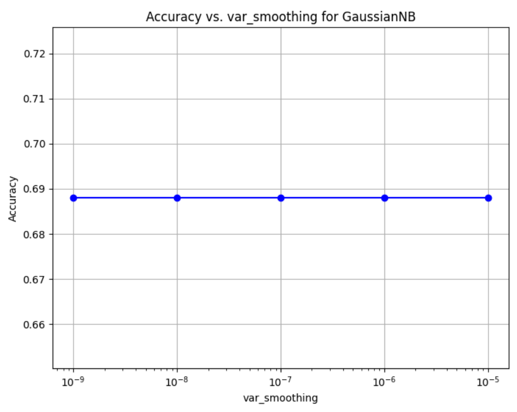
Due to the simplicity of the Gaussian-Bayes Model in the Scikit Learn library, little was done to the class to tune the results of the model as the only two parameters of the class belonged to prior values of the classes, and variance smoothing. We did tamper with the variance smoothing parameter by iterating through a list of different variance smoothing values, training the model based on that value, then representing the results on a graph

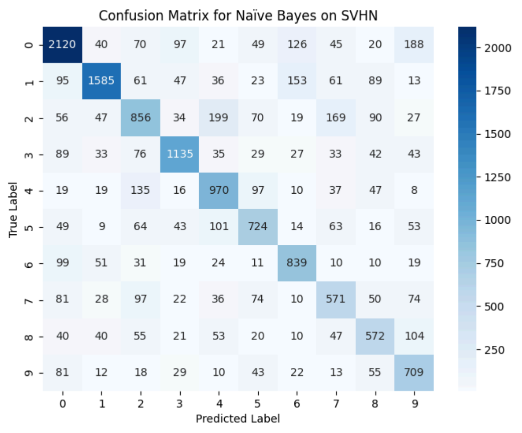
```
var_smoothing_values = [1e-9, 1e-8, 1e-7, 1e-6, 1e-5] # first one is
default
```

- The list of values. Variance smoothing pertains to largest variance of all features and can help with calculation stability

Step 3: Results and Evaluation

We utilized two types of analysis, a line graph for the different variance smoothing values and a confusion matrix for the best version of the model.





- <u>Best Naïve Bayes Accuracy on SVHN: 0.6880 using 1e-09 variance smoothing</u>
- According to the graph on the left, the values that we used for variance smoothing did not have any effect on the model's prediction accuracy at all
- For the confusion matrix, it would appear that classes 7 and 8 were the least optimized as they had the least number of correct predictions (via value along the diagonal of the matrix

Future Improvements

- Due to the inflexibility of the model, the only improvement we can have is changing what kind of variance smoothing values we use to test and use more sparsely separated values in order to get more potential variety

<u>XGB Gradient Boost (Nathan Loo)</u>

Step 1: Loading and Preprocessing the Dataset

Similarly to the Naive-Bayes Classifier that we used previously, the XGB gradient boost model also had to go through the same gray scaling and HOG array conversion. For reference:

```
train_data = loadmat('train_32x32.mat')
test_data = loadmat('test_32x32.mat')
X = np.moveaxis(train_data['X'], -1, 0)
y = train_data['y'].flatten()
```

- Manually retrieving the data from the MAT files using scipy.io

```
X_gray = np.dot(X[..., :3], [0.2989, 0.5870, 0.1140])
```

- Gray scales images manually using the 3 values on the right in place of [R, G, B] respectively

```
- X_hog = np.array([hog(img, pixels_per_cell=(8,8),
  cells_per_block=(2,2)) for img in X_gray])
```

- Then creating histogram of gradients (HOG) array for each image in the gray scaled array

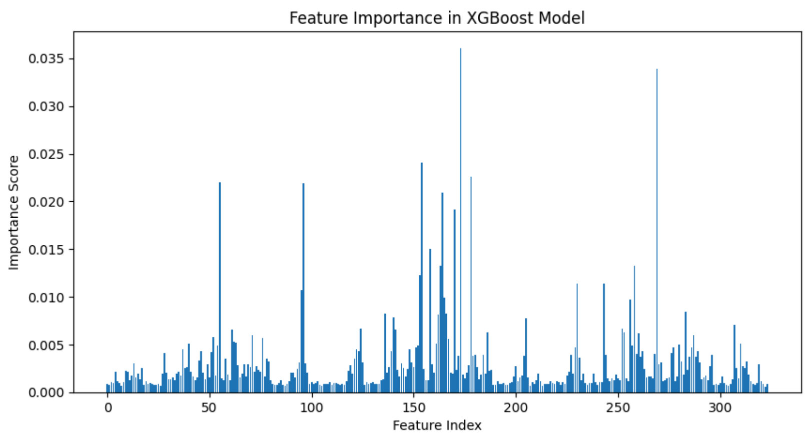
Step 2: Model Training and Prediction

This model was a little more complex than the ones we used and took us a while to configure the class to our liking. We ended up with:
`xgb.XGBClassifier(objective="multi:softmax", num_class=10, eval_metric="mlogloss", use_label_encoder=False)`

- Objective = “multi:softmax” parameter defines what the model is optimizing for. “Multi:softmax” is for a multi-class classification where the model predicts one out of many classes
- Num_class = 10 initializes the number of classes the model can predict from
- eval_metric=”mlogloss” indicates how well the model’s prediction matched the actual class. The lower mlogloss is, the better the result is

Step 3: Results and Evaluation

Aside from the accuracy score, we looked into a feature importance graph, which was a bar graph that represented which features of the model had the most impact on the predictions of the model.



- <u>From Terminal: XGBoost Accuracy on SVHN: 0.8165</u>

Future Improvements

- Using the Feature importance graph, we should be experimenting much more with the many other parameters that this class has to offer, as this graph clearly shows what is and isn’t important to this model, allowing us to potentially reduce noise and other insignificant data features that could be impacting our predictions.
- We could introduce regularization via the reg_alpha and reg_lambda parameters to remove those insignificant features and tamper with the min_child_weight in order to prevent splitting on features with less data.

Random Forest (Dylan Tanaka)

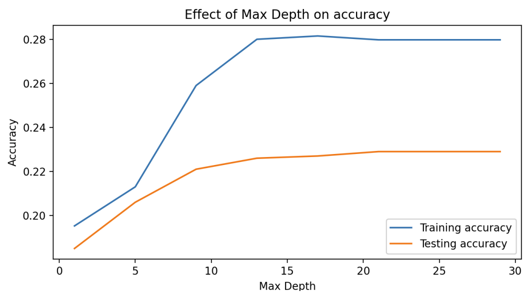
Step 1: Initializations and loading data

For our random forest, we utilized sklearn's **RandomForestClassifier**. Again, we trained on the SVHN dataset. The first step involves loading the data and modifying it's form for the purposes of this project:

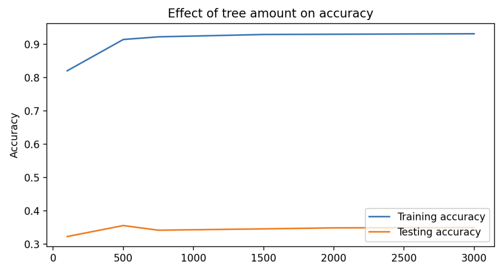
- Use **scipy** to load the mat files of the image training and testing datasets
- Use a smaller portion of the dataset to reduce training time. In this case, we used the first **4,000** samples from the training dataset, and the first **1,000** samples for the testing dataset.
- **Transposing** the training and testing images by putting the sample index (index 3) into index 0, so that the order of features in the image tuple could be processed by the classifier.
- Modifying all RGB values to a single average using numpy.mean, to **grayscale** the image.

Step 2: Tweaking hyperparameters

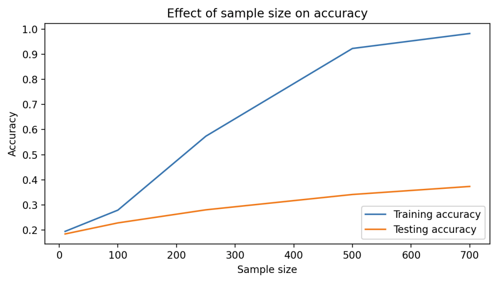
There were multiple **hyperparameters** that could be tweaked in order to make our random forest classifier more accurate. To visualize this, we trained some random forest learners with different values for each hyperparameter we tweaked, and visualized its effects by plotting the accuracy associated with each hyperparameter value.



Max Depth: Modifying the max_depth hyperparameter helps to reduce overfitting, but a max depth being too low causes underfitting.



Number of trees: Modifying n_estimators increased the amount of decision trees in the forest, thus increasing accuracy but also computation time. A high enough amount of estimators caused overfitting.



Max samples: We used bootstrapping to more accurately find correlations in the data. With this, increasing max_samples increased accuracy.

Step 3: Final random forest model

Utilizing the knowledge gained from tweaking hyperparameters, we created our final random forest classifier using these parameters:

- **2000** estimators. With higher max samples, a larger amount of trees does not reduce accuracy
- **1750** max samples, high enough to increase accuracy without overfitting
- Further reduce overfitting using **log2** for computing max features to consider
- Maximum depth of **25** to avoid underfitting

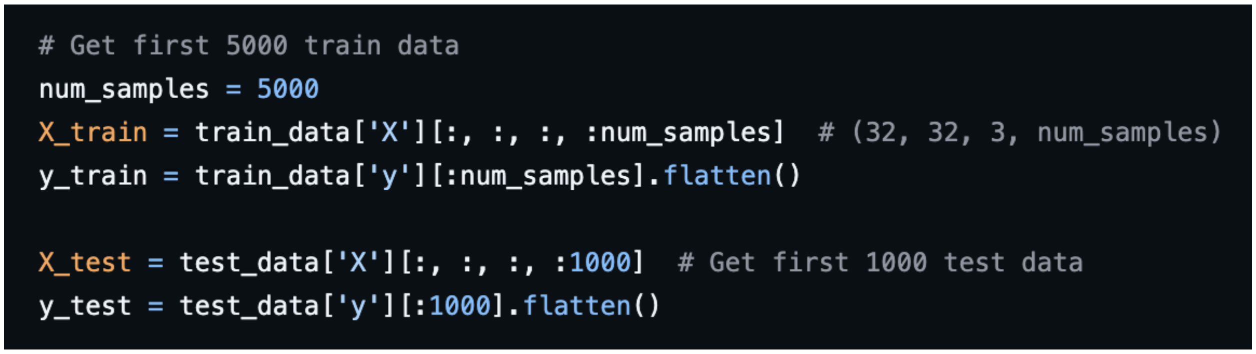
Step 4: Evaluation

With the above parameters, the error rate for our random forest was **0.0455** with 4,000 training samples, and **0.49** with 10,000 training samples. It is not the most ideal, but we must consider that a random forest is not as ideal for image classification as a convolutional neural network, despite hyperparameter tuning. Regardless, it was good to train and test a random forest and evaluate its effectiveness on image classification. In the future, increasing the sample size and using more complex hyperparameter optimization such as random or grid search could potentially help to increase the accuracy of this model.

Support Vector Machine(SVM) (ZiHao Xia)

Step 1: Loading and Processing the Dataset

Data Loading: To optimize computation time while retaining reasonable accuracy, we select the first 5,000 samples for the training dataset and 1,000 samples from the testing dataset, using **scipy.io.loadmat()** to load .mat format file.



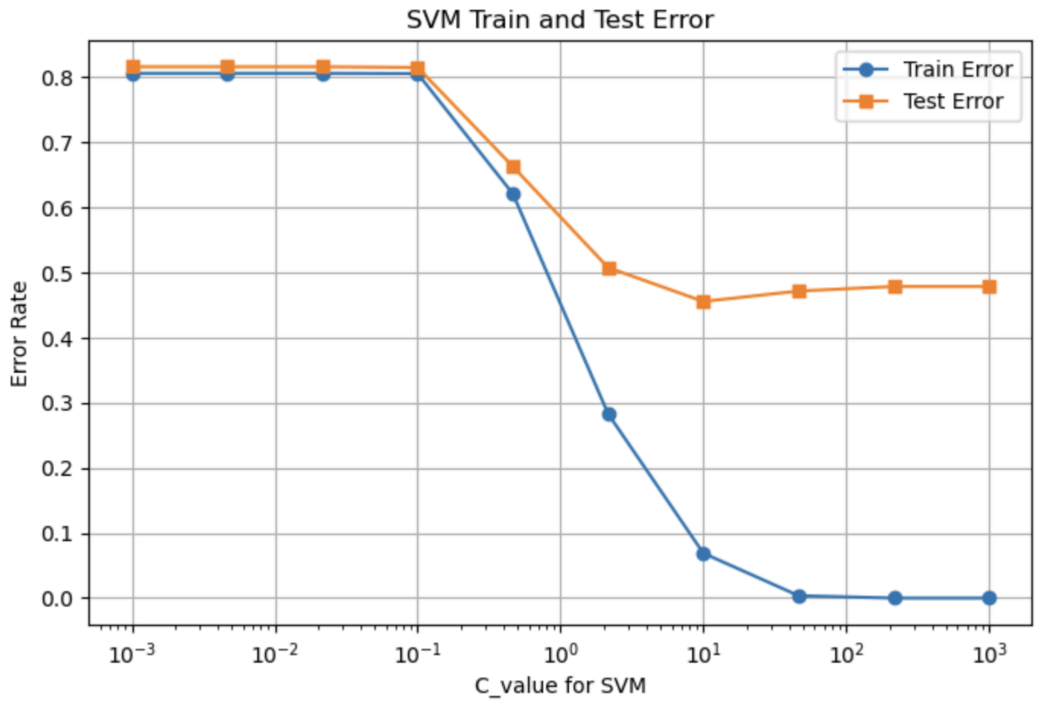
```python
# Get first 5000 train data
num_samples = 5000
X_train = train_data['X'][:, :, :, :num_samples]  # (32, 32, 3, num_samples)
y_train = train_data['y'][:num_samples].flatten()

X_test = test_data['X'][:, :, :, :1000]  # Get first 1000 test data
y_test = test_data['y'][:1000].flatten()
```

Data Transformation: The dataset assigns the label 10 to digit 0, so we mapped all 10 labels to 0. To follow the traditions with most AI training functions, we reordered the dataset from (32, 32, 3, num_samples) to (nums samples, 32, 32, 3). We then loaded photos into grayscale, and flattened data to 1D to make sure the data format is compatible with libSVM.

Step 2: Training with Different C_values

The **C_value** is one of the parameters of SVM that controls the tradeoff between maximizing the margin and minimizing classification errors. We tested 10 different C values ranging in np.logspace(-3, 3, 10).



The testing error descents to the lowest points when C_value = 10. Then the model starts overfitting as C_value goes higher. Compared to our CNN which has a test accuracy of **91.01%**, our SVM model only has a test accuracy of **53.72%.**

Since the SVHN is not a binary data set, training the SVM with a linear kernel does not make too much sense and is time consuming, so we decided to not run the SVM with a linear kernel. We instead chose the RBF kernel with gamma=”scale”, which is the most common and well-performed parameter combo for general training purposes.

Conclusion

As a whole, our group decided to train different types of models to evaluate a variety of potential solutions to the SVHN digit recognition problem. While we were able to increase the accuracy of our models through various means, ultimately, the convolutional neural network is by far the most accurate model for image recognition.