Classification of Street View House Numbers Using PyTorch

A Simple Study On The Impact of Different Hyper-Parameters On The Performance Of A VGG Network

Student Huang Yanzhen DC126732
Groupmate Mai Jiajun DC127853
Source Code https://github.com/hyz-courses/CISC3024-Pattern-Recognition.git

1 Introduction

Visual Geometry Group, also known as VGG or VGGNet, is a classical type of convolutional neural network (CNN) structure that enhances the performance of the model by increasing its depth, using only 3×3 convolution filters in every convolutional layer [1]. In this project, we are going to construct a VGG to perform a classification task on the Street View House Numbers (SVHN) [2] dataset and perform a simple evaluation on how multiple hyper-parameters will affect the performance of the model by performing sets of experiments.

2 Methodology

In this part, this report will introduce how the dataset class and its useful features are defined, as well as how the training and validation dataset is split and settled. Then, it will introduce the method of training a single VGG model called SmallVGG, along with the implementation of an early-stop mechanism. After that, it will describe how each experiment in a set is performed, and how each set of experiments are loaded for evaluation. Lastly, the structure of the VGG model will be introduced.

2.1 SVHNDataset Class

A Dataset class is required to work with the DataLoader object to better manage different types of data. Along with the constructor `__init__`, more useful functions are defined to better manipulate the experiments.

2.1.1 Constructor

The class's constructor function takes the file's location as an input and stores the images into the class's parameter. It also transposes the data part of the input dataset into a nested list, with each layer of the list representing multiple images, multiple image rows in an image, multiple image pixels in a row, and multiple image channels in a pixel, respectively. It also flattens the label part of the dataset to match the transposed data part.

```
class SVHNDataset(Dataset):
    def __init__(self, mat_file, transform=None):
        data = sio.loadmat(mat_file)

        self.images = np.transpose(data['X'], (3, 0, 1, 2))
        self.labels = data['y'].flatten()
        self.labels[self.labels == 10] = 0
        self.transform = transform

    def __len__(self):
        return len(self.labels)
```

Figure 1: The partial definition of `SVHNDataset` class.

As defined, the label of 10 will be given to the number of 0. Therefore, the conversion is also performed during the construction process of an SVHNDataset class. A `transform` defines a sequence of processing steps of normalizing an image and converting it into a tensor, only after which can the image be passed through the neural network. The basic transform is defined to only perform a simple mean-normalization on all the images with

the mean and standard derived from any required set of images. There may also be some other transformations assisted with `albumentations`, like `RandomResizedCrop`, whose effect on the performance of model training and application will be soon investigated. It is allowed not to specify the transform during construction.

2.1.2 Get Item

The `__getitem__` realization of the abstract function with the same name in PyTorch's `Dataset` class defines how each data sample is retrieved from the class. However, as we can not let an image that's un-normalized or not a tensor be passed into the neural network, the function will raise an error if the transform is not defined. That is, the caller can indeed postpone the definition of transform, but he or she shall inject one before passing the `Dataset` object to the data loader. This design is settled because one shall calculate the mean and standard deviation in order to retrieve the transform object while calculating the two values requires retrieving the original data first.

```
def __getitem__(self, idx):
    image = self.images[idx]
    label = self.labels[idx]

    if self.transform is None:
        raise ValueError(f"CISC3024 Custom Error: The transform should not be None"
                         "when this object is passed into a DataLoader.")

    image = self.transform(image=image)['image']
    return image, label
```

Figure 2: The `__getitem__` function.

2.1.3 Caluclate Mean and Standard Deviation

The dataset class also provides the function to calculate the mean and standard deviation of the data within itself. The mean and standard deviation of a dataset object are two vectors, each having the length of 3, which represent the mean and standard deviations on the 3 channels respectively. Moreover, the invoker could also choose to add a contrast factor to the images onto the mean.

2.1.4 Overwrite

During the train-valid split, it is necessary to overwrite an original dataset, filtering out only a partial of its original data. The `overwrite` function first gives a deep copy of the dataset class instance itself. It then modifies the deep copy using the input indices, which store the information of which data samples we are preserving, given by the two `Subset` class instances returned from the `random_split` function of PyTorch, which we used in the train-validation split process.

```
def overwrite(self, indices:Union[list, np.ndarray]):
    if any(index < 0 or index >= len(self.labels) for index in indices):
        raise IndexError("CISC3024 Custom Error: One or more indices are out of bounds.")

    new_dataset = copy.deepcopy(self)
    new_dataset.images = self.images[indices]
    new_dataset.labels = self.labels[indices]
    return new_dataset
```

Figure 3: The overwrite function for the train-validation split.

2.2 Train-Validation Split

The SVHN dataset classifies its data into three categories: "train", "test", and "extra". We are splitting the "train" dataset from SVHN into our own training and validation dataset. The ratio of train data over validation is $8:2$. In other words, we are using 80% percent of the original "train" data as our own training dataset, and use the rest par as the validation dataset. The "test" dataset is used to inspect the application performance.

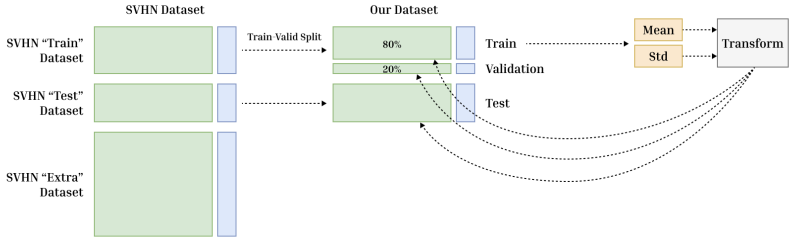

Figure 4: Train-validation split and the mean and standard deviations for image normalization and augmentation.

We will be calculating the mean and standard deviation using only our split train dataset for image normalization to prevent data leakage. The calculated mean and standard deviation will be used in all the datasets.

2.3 Train and Evaluate

For each epoch, the function `train_and_evaluate` will train the model with the entire training and validation datasets. It performs training and evaluation of the model at the same time, splitting each epoch into a training cycle and an evaluating cycle. The length of each cycle is controlled by the size of the dataset. After training, the function will return a list of training losses and a list of validation losses of each epoch for inspection.

2.3.1 Training and Validation Cycles

For each training cycle, all data and samples in the `train_loader` will be passed into the model correspondingly. Getting the loss, the optimizer will step forward, and the loss will be backward propagated to update the weights. The loss at this cycle will be accumulated to an epoch training loss.

For each validation cycle, all data and samples in the `valid_loader` will be passed into the model correspondingly, getting the validation loss. Similarly, the validation loss at this cycle will be accumulated to the epoch validation loss. The validation loss is used to perform early stoppings to prevent over-fitting.

```
# Train
model.train()
running_loss = 0.0
for images, labels in tqdm(train_loader):
    images, labels = images.to(device),
                     labels.to(device)
    optimizer.zero_grad()
    outputs = model(images)

    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    running_loss += loss.item() * len(images)
train_losses.append(running_loss/len(train_loader))
```

```
# Evaluate
model.eval()
valid_loss = 0.0
with torch.no_grad():
    for images, labels in valid_loader:
        images, labels = images.to(device),
                         labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        valid_loss += loss.item() * len(images)

valid_losses.append(valid_loss/len(valid_loader))
```

Figure 5: Training and evaluation of a single epoch.

2.3.2 Early Stopping Mechanism

A model that's not over-fitted performs well on unseen data. This function implements an early stopping mechanism using an over-fitting accumulator to prevent over-fitting. The principle is that we record the smallest validation loss we have ever seen along with its current model state, and compare it to the current validation loss of each epoch. If the current validation loss is higher than the minimum, we consider it as a tendency to over-fit, thus adding the over-fitting accumulator by 1. Similarly, if we find a new minimum validation loss that's smaller than the current one, we overwrite the minimum loss and the optimum model state, decreasing the accumulator by 1.

```
for epoch in range(num_epochs):
    ... # Train and evaluation logic

    # Early Stop?
    if stop_early_params is None:
        continue

    if current_min_valid_loss - stop_early_params["min_delta"] > valid_losses[-1]:
        # Validation loss decreases
        current_min_valid_loss = valid_losses[-1]
        current_optimized_model = copy.deepcopy(model)
        num_overfit_epochs = (num_overfit_epochs - 1) if num_overfit_epochs > 0 else 0
    else:
        # Validation loss increases
        num_overfit_epochs += 1

    if num_overfit_epochs > stop_early_params["patience"]:
        print(f"Early stopping at epoch {epoch+1}.")
        model = current_optimized_model
        break
```

Figure 6: The early stopping mechanism.

Once the over-fitting accumulator exceeds a patience factor, the model will stop training and the function will overwrite the model to the stored optimal state. This simple early stopping mechanism allows the invoker to keep the running optimal model as the function foresees a rise in validation loss, preventing over-fitting. However, for certain experimental purposes, the invoker of the function can also choose not to provide any early-stopping parameters and simply run through the entire epoch, potentially creating an over-fit situation with intention.

The advantage of this mechanism is that if the algorithm finds the validation loss continuously decreasing, it will decrease the over-fitting accumulator to 0 gradually, allowing the training process to continue. If the algorithm witnesses an occasional rise in validation loss, the mechanism will tolerate it if the accumulator does not reach the patience factor. This mechanism demonstrates the balance of finding a potential future optimum and preventing future over-fitting.

2.4 Run and Load Experiments

2.4.1 Run and Store Experiments

The experiments will be conducted in sets. Each experiment set will have at most two lists of candidate hyper-parameters to be tested. If there are two lists, the two candidate lists will form a Cartesian Product using the `itertools.product` function to produce the list of parameter tuples of all the possible combinations of the candidate hyper-parameters, representing all the possible experiment sets. Namely, if S_1 and S_2 are the two lists of the candidate hyper-parameters, the experiment set will be represented as follows.

$$S_{\text{Experiment}} = S_1 \times S_2 \tag{1}$$

Each experiment set is represented as a list of experiment objects. An experiment object is a data structure that stores the two candidate hyper-parameters of the current experiment, the lists of training and validation losses during model training of this experiment, and the model state itself. In each experiment set with 2 tested hyper-parameters, a grid search will be conducted over the produced tuples of candidate hyper-parameters for a result set that's more intuitive and trustworthy.

```
{
    "HYPER_PARAM_1": combo[0],
    "HYPER_PARAM_2": combo[1],
    "train_losses": train_losses,
    "valid_losses": valid_losses,
    "model_state_dict": this_model.state_dict()
}
```

Figure 7: Structure of an experiment object.

The function for running an experiment is roughly as follows. The first two parameters are the two lists of candidate hyper-parameters. The parameter `hyper_params` are the two names of the hyper-parameters, which will be used as the keys of the corresponding candidate hyper-parameter within the experiment object.

```
def run_exp3_1(angles, crops, hyper_params, train_dataset, valid_loader):
    combinations = list(itertools.product(angles, crops))
    experiments = []

    for i, combo in combinations:
        angle, crop = combo
        ... # Model training using the parameters.

    return experiments
```

Figure 8: Code for running an experiment that's partially omitted.

Storing the experiments involves no extra work but storing the list of experiment objects into a `.pth` file in the `models/` directory. However, to prevent accidental overwriting, a timestamp string is added to the model name.

```
exp3_2 = run_exp3_2(candidate_ratios,
                    candidate_contrast_factors,
                    exp3_2_hyperparams, exp3_train_dataset, exp3_valid_dataset)
time_str = str(time.time()).replace(".", "")
torch.save(exp3_2, f"./models/exp3-2_{time_str}.pth")
```

Figure 9: Code for storing the experiment set.

2.4.2 Load Experiments and Get Predictions

One could use the model state in the experiment object to perform classification on the test data and evaluate the actual classification performance by using multiple metrics, including accuracy, as well as the class-wise precisions, recalls and F_1 scores. The function `get_predictions` takes an input of a model and a data loader, performs the classification, and outputs a tuple that contains the list of prediction scores, the list of true labels as well as the list of the predicted labels. The prediction scores list is the one-of-K metric of the output, each of whose elements is a list of length-10 vectors of 10 scores from a single classification of all the possible samples.

```
exp3_1_loaded = torch.load("./models/exp3-1_17307349208257582.pth")
exp3_1_results = get_experiment_results(exp3_1_loaded,
                                        test_hyperparam_names=["angle", "crop"],
                                        extra_loader=exp3_test_loader)
```

Figure 10: Code for loading and running an experiment set.

Corresponding to an experiment set, an experiment result set is also a list of objects, called the "experiment result object". The function `get_experiment_results` invokes the function `get_predictions` for all the experiment objects in the loaded experiment list, storing all the required evaluation metrics in the corresponding experiment result object of this experiment object. The structure of an experiment result object is given as follows.

```
{
    "HYPER_PARAM_1": combo[0],
    "HYPER_PARAM_2": combo[1],
    "true_labels": true_labels,
    "pred_labels": pred_labels,
    "pred_scores": pred_scores
}
```

Figure 11: Structure of an experiment object.

Namely, an experiment result object is simply a converted version of its corresponding experiment object, whose model state is replaced by the evaluation metrics of the application of that model over the test dataset.

2.5 Neural Network Architecture

2.5.1 Convolution

As the name suggests, convolution is the key of CNNs, including VGGNet. The reason why we need convolution to classify images with neural networks is that the simple structure of fully connected layers cannot effectively handle the large input of a regular 3-channeled image, while convolution layers convolve the input into a smaller feature map, with better abstraction and a set of refined and useful data.

In a narrow sense, convolution is the process of iterating an image with a weighted kernel, generating a new image with more abstraction by generalizing multiple neighbor pixels around a center pixel and summarizing them into one pixel. The smaller the kernel, the fewer neighbors are considered and the generalized image is less abstract. In VGGNet, the convolutional kernel is limited to 3×3, therefore we need more layers for a certain level of abstraction. The mathematical description of a convolution with one of multiple kernels is listed below.

$$Y_i = \sigma\Big(\sum_{j=1}^{n} X_j * W_{ij} + b_i\Big) \tag{2}$$

Here, n is the input channel number, while i is the index of the weighted kernels. σ is the activation function, usually being ReLU. In the above formula, W_{ij} is the i-th slice of the $3 \times 3 \times n$ kernel W_j. After being biased by b_i, the intermediate convolution result will be then activated by the ReLU function, getting the corresponding feature map Y_i of convolution kernel W_i.

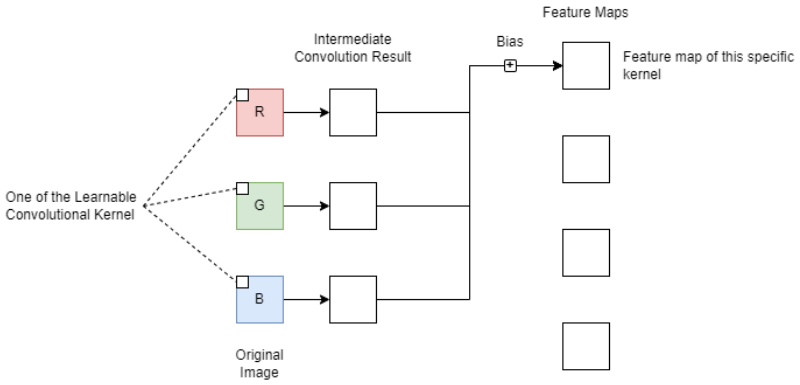

Figure 12: Convolution applied by one of the multiple kernels.

We can apply multiple learnable weighted kernels on all 3 channels of an original image and generate original-sized multiple-channeled feature maps. However, as we increase the number of channels, we need to merge local pixels to reduce the size of the input with pooling. Since more abstraction is given, the fewer threats of losing useful information the pooling process creates.

In this project, the convolution process takes a colored image, which is a 3-channeled 32×32 matrix, as an input, and outputs a multi-channeled matrix with a smaller size, which will be further taken as an input to a traditional neural network. The figure below demonstrates the convolution part of the VGG.

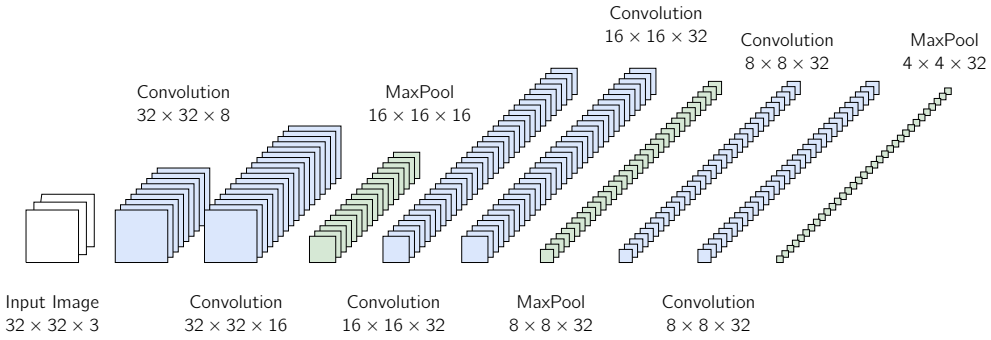

Figure 13: The output of each convolutional layers.

In this custom VGG, there are three convolution processes, each containing two convolution operations, and a max-pooling operation. ReLU activation is applied after every convolution. The first step uses two convolution layers to convolute the original normalized 32×32-sized image with three channels into a 32×32-sized matrix with 8 channels. During this process, 8 learnable 3×3 kernel is applied on all the 3 original channels, each generating its own feature map, sharing the exact same principle of the entire following process. Then, preserving the image size, the matrix is then convoluted further into 16 channels, before it is max-pooled into size 16×16, preserving the number of channels. The second convolution step convolutes the 16×16 matrix into a 32-channeled one with 2 layers, between which size and the number of channels is preserved before it max-pools the matrix into an 8×8 32-channel one. The last convolution process takes a similar step as the one before, convoluting the 8×8 32-channeled matrix twice and finally max-pooling it into a 4×4 32-channeled matrix.

By now, the original number of inputs per image is now refined from $32 \times 32 \times 3 = 3072$ into $4 \times 4 \times 32 = 512$ inputs per image, reducing 83% of the data amount. During forward propagation, the final production of the convolutional layer will be further flattened to feed into the fully connected layers.

2.5.2 Fully Connected Layers

At the end of the convolution, we obtain a small image matrix of shape 4×4 with a total of 32 channels. By flattening them into inputs pixel by pixel, we need an input size of $32 \times 4 \times 4 = 512$. There are a total of 2 hidden layers before a 10-node output layer. Except for the input layer, the output of every hidden layer will be activated by a ReLU function.

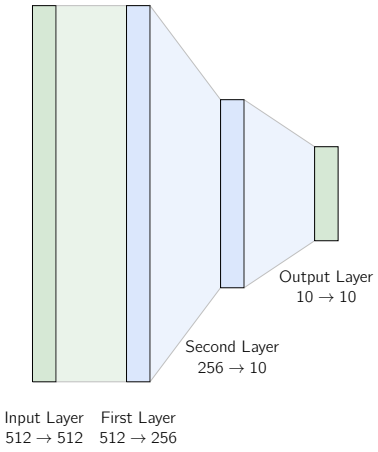

Figure 14: Fully connected layers of the VGGNet.

The first layer takes an input of size 512, which is the convolution result of the image, and outputs the logits of size 256. The second layer takes the biased ReLU-activated size-256 output of the first layer as an input, and output a logit of 10, which will be the one-of-K classification result.

3 Experiment Results and Discussion

We designed four experiment sets, some of which have their own subset of experiments. The first experiment set only aims at finding the best optimizer. The second experiment set aims at finding the best combination of epoch number and learning rate, additionally exploring different batch sizes. The third experiment set aims at searching for the best augmentation parameters. The fourth experiment set aims to discover the effect of different model architectures and activation functions on training and applications.

3.1 Experiment Set 1: Optimizers

The standard process of gradient descent has the problems of oscillations and slow convergence. Namely, if the learning rate is too high, the model will jump over the local minimum. On the contrary, if the learning rate is too low, the model will spend an exponential amount of epochs to converge to a single local minimum.

$$\theta_t = \theta_{t-1} - \eta \cdot \nabla \mathcal{J}(\theta) \tag{3}$$

Optimizers are introduced to solve this problem by introducing the concept of "momentum" [3]. Momentum accumulates past gradients in the direction of the consistent descent. It gives a weighted sum of the previous gradients so that the direction of the update of parameters is not dependent only on the current gradient but also on the previous ones. The updating of the momentum is assisted by a decay coefficient β.

$$m_t = \beta \cdot m_t + (1 - \beta) \cdot \nabla \mathcal{J}(\theta) \tag{4}$$

Thus, having the new momentum, the updated gradient descent formula is thus

$$\theta_t = \theta_{t-1} - \eta \cdot m_t \tag{5}$$

Experiment set 1 promotes six candidate optimizers, each possessing its own variant of momentum updating measure. The candidate optimizers are:

1. Adaptive Moment Estimation (Adam) [4]
2. Stochastic Gradient Descent (SGD) [5]
3. Root Mean Square Propagation (RMSprop)
4. Adam with Weight Decay (AdamW, using default weight decay of 0.01)
5. Adaptive Gradient Algorithm (Adagrad) [6]
6. SGD with Momentum and Nesterove Accelerated Gradient

All the hyper-parameters except for the optimizers are fixed. All the experiments are conducted under the epoch number of 25 and the initial learning rate of 1.0×10^{-3}. Since the optimizers will adjust the learning rate during the training process of all the experiments, the initial learning rate will be referenced as simply "learning rate" on certain occasions for the following part. The training performances of all the experiments are displayed as follows.

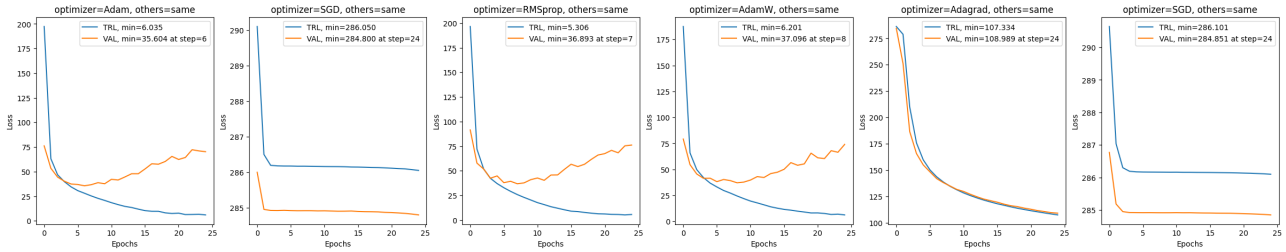

Figure 15: Experiment Set 1: Epoch-loss curves.

Adagrad (the 5th plot) cumulatively updates the learning rate. It gives a decay to both weight and learning rate: λ_1 and λ_2.

$$g_t = \nabla \cdot \mathcal{J}(\theta_{t-1}) + \lambda_1 \cdot \theta_{t-1} \tag{6}$$

$$\widehat{\eta} = \frac{\eta}{1 + (t-1)\lambda_2} \tag{7}$$

$$\tau_t = \tau_{t-1} + g_t^2 \tag{8}$$

$$\theta_t = \theta_{t-1} - \widehat{\eta} \frac{g_t}{\sqrt{\tau_t} + \epsilon} \tag{9}$$

Obviously, in Adagrad, the learning rate is monotonically decayed through the training process. τ, as the accumulator, accumulates the squared gradient without weights, which results in the difference of weights in each step $\Delta\theta_t = -\widehat{\eta}\frac{g_t}{\sqrt{\tau_t}+\epsilon}$ also decreases monotonically. Therefore, it is reasonable to find that, for Adagrad, the learning process is indeed smoother, but the speed of convergence is significantly slower.

On the contrary, the performance of Adam, RMSprop, and AdamW (the 1st, 3rd, and 4th plot respectively) all leads to an over-fit of the model since around the 7th epoch, serving a similar characteristic. The validation loss curve of them fluctuates more than Adagrad, meaning that the decay of the learning rate are not as strong as the others.

In fact, expanding the ideas in 4, Adam, AdamW, and RMSprop all adapt different variants of RMSProp algorithm that apply two momentum items. The first momentum m_t and the second momentum v_t, each having

their own decay coefficient β_1 and β_2 respectively. The update processes are as follows.

$$g_t = \nabla \mathcal{J}(\theta) + \lambda \cdot \theta_{t-1} \tag{10}$$

$$\widehat{m_t} = \frac{\beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t}{1-\beta_1^t} \tag{11}$$

$$\widehat{v_t} = \frac{\beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2}{1-\beta_2^t} \tag{12}$$

$$\theta_t = \theta_{t-1} - \eta \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \tag{13}$$

Similar to τ in Adagrad, v_t is introduced to adapt the learning rate to record the squared gradients, thus improving stability. When the gradient fluctuates, v_t becomes larger, reducing the learning step η into $\frac{\eta}{\sqrt{\widehat{v_t}}}$. Instead of simply accumulating the squared gradient as it would be in Adagrad, v_t is the exponentially weighted average of the squared gradient. With decay factor $\beta_2 \in [0, 1]$, v_t will "forget" the historically added values "a little bit", therefore being extremely unlikely to increase monotonically through the training process. Therefore, although the validation loss curve of Adam (with weight decay $\lambda = 0$) and AdamW (with weight decay $\lambda = 0.01$) still displays some fluctuations, the training process is still robust and is able to find further local minimum points within the same number of epochs.

The SGD family, however, performs poorly compared to others, leaving a huge drop of learning steps before the 5th epoch and an almost non-changing training and validation loss at the end. The problem of SGD is that the momentum μ and dampening τ (higher value results in a higher contribution of the momentum) need to be manually initialized and do not change at all.

$$g_t = \nabla \mathcal{J}(\theta_{t-1}) + \lambda \theta_{t-1} \tag{14}$$

$$\mathbf{b}_t = \begin{cases} \mu \mathbf{b}_{t-1} + (1-\tau) g_t & \text{if } t > 1 \\ g_t & \text{otherwise} \end{cases} \tag{15}$$

$$\theta_t = \begin{cases} \theta_{t-1} - \eta(g_t + \mu \mathbf{b}_t) & \text{if use Nesterov} \\ \theta_{t-1} - \eta \mathbf{b}_t & \text{otherwise} \end{cases} \tag{16}$$

As shown in the epoch-loss plots, poor initialization leads to poor results. Moreover, we can see the contribution of Nesterov in the graph that, the SGD that applies Nesterov (the 6th plot) has a smoother transition before the 5th epoch compared to the one that didn't (the 2nd plot).

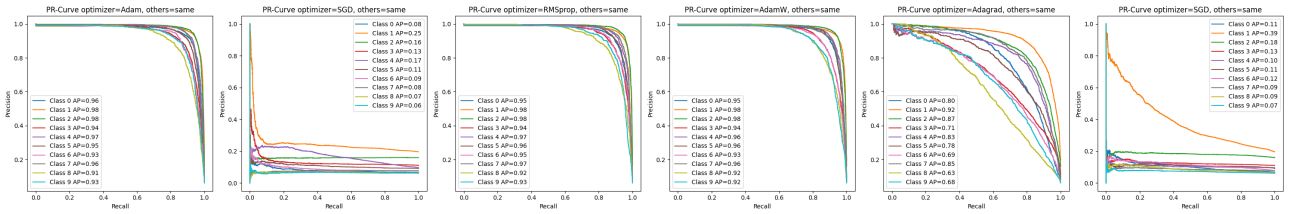

Figure 16: Experiment Set 1: Precision-recall curves.

```
Accuracies:
0.907 0.196 0.906 0.901 0.735 0.196

F1 Score Lists:
0.912 0.935 0.938 0.874 0.921 0.905 0.881 0.906 0.847 0.867 | Avg F1=0.899, Std F1=0.028664446684648247
0.000 0.328 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 | Avg F1=0.033, Std F1=0.09827503131926378
0.888 0.933 0.939 0.876 0.913 0.903 0.878 0.923 0.850 0.865 | Avg F1=0.897, Std F1=0.028470034992916605
0.887 0.935 0.936 0.871 0.914 0.907 0.871 0.908 0.830 0.843 | Avg F1=0.890, Std F1=0.03434799781092271
0.723 0.840 0.802 0.638 0.776 0.700 0.647 0.789 0.515 0.628 | Avg F1=0.706, Std F1=0.09492988307421246
0.000 0.328 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 | Avg F1=0.033, Std F1=0.09827503131926378
Best: 1-th
```

Figure 17: Experiment Set 1: Accuracies and F_1 scores.

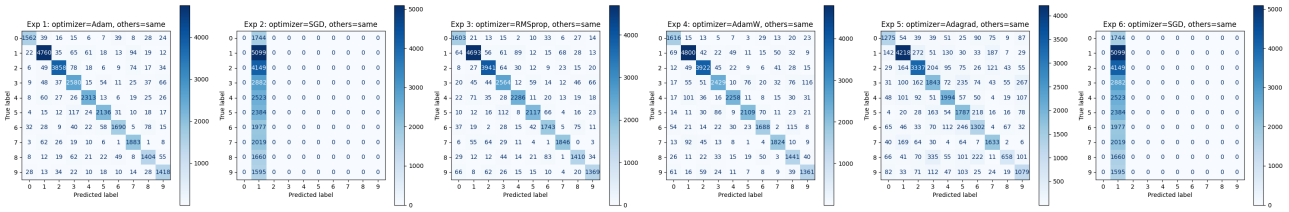

Figure 18: Experiment Set 1: Confusion matrices.

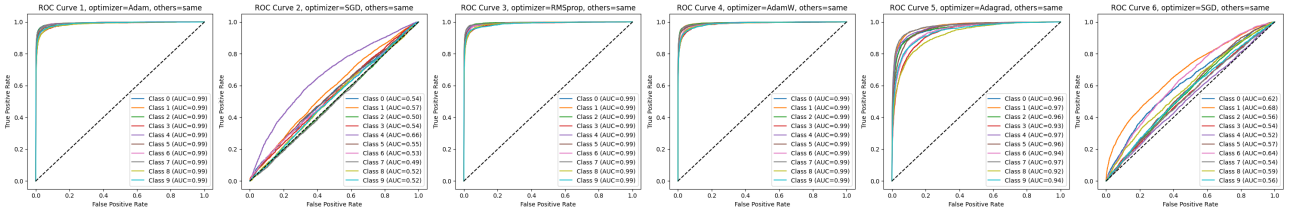

Figure 19: Experiment Set 1: ROC curves of all classes.

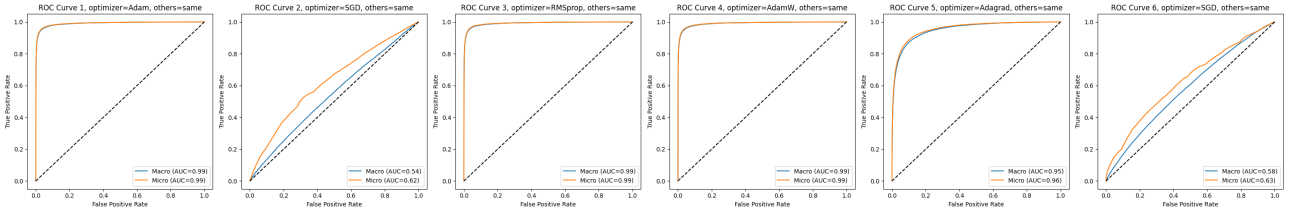

Figure 20: Experiment Set 1: Macro and micro ROC curves.

The family of SGD optimizers, when poorly initialized, consistently yields low performance, with common average precisions (APs) under 0.2, and a typical poor accuracy of around 0.196. While Nesterov momentum offers some improvement over standard SGD (as seen in the sixth plot), particularly in classifications involving label 1 (the most frequent label), its overall performance remains inferior to other optimizers. It could be observed from the confusion matrices that the SGD-based models tend to classify every instance into label 1, leading to a skewed distribution. Moreover, the ROC curves indicate a near-random classifier, with ROC-AUC scores close to 0.5. The disparity between the macro and micro ROC curves also proved that inconsistency in performance exists among different categories.

Adagrad, though less effective than Adam, AdamW, and RMSprop, achieves more reasonable results, with class-wise APs around 0.80 (ranging from 0.63 to 0.92) and a respectable overall accuracy of 0.735. The precision-recall curves indicate room for improvement, particularly in comparison to the RMSprop family. Adagrad provides a steep PR curve for label 1, having the highest AP of 0.92; While it provides a relatively flatter curve for label 8, having the lowest AP of 0.63. The confusion matrix of Adagrad shows some noise around the counter-diagonal line, reflecting slightly lower performance than RMSprop, but the ROC curves still indicate a fairly good performance, with even the lowest ROC-AUC remaining above 0.90. Despite label 8 being the weakest, its ROC curve still trends toward the top-left corner.

Concerning the average F_1 score, the best optimizer is Adam, with an average class-wise F_1 score of 0.899. The accuracy also tells the same story, with the accuracy of Adams being the highest at 0.907. However, the difference in performance among Adam, AdamW, and RMSprop are not immense, each having an accuracy of around 0.9 and an average F_1 score of 0.89.

For the following experiments, for simplicity and performance, Adam is chosen to be the constant optimizer.

3.2 Experiment Set 2: Training Parameters

In this experiment set, training parameters of epoch number, learning rate, and batch size will be tested.

Epoch number and learning rate are the two important hyper-parameters that affect the training performance of the model. An epoch refers to the process of feeding the entire batched training and validation data into the model, performing forward and backward propagation for each batch. The more epoch is defined, the more time the entire training and validation data is used to update the weights in the model, thus the further the gradient descent advances. If the epoch number is too small, the model will stop training before it meets a local minimum; while if the epoch number is too large, the model may pass the local minimum and get over-fitted.

A learning rate defines the step size of gradient descent, telling how fast each gradient descent proceeds to the next one. If the learning rate is too small, the model will converge in an extremely slow manner such that it may not meet a local minimum within abundant epochs; while if the learning rate is too large, a gradient explosion will be passed through each layer of the network, leading the model to jump back and forth around the local minimum without converging, or even explode the loss value to infinity.

In this experiment, two sets of epoch numbers and learning rates will be given, and a grid search will be performed on them.

3.2.1 Experiment Subset 2-1: (Rough Search) Epoch Numbers and Learning Rates

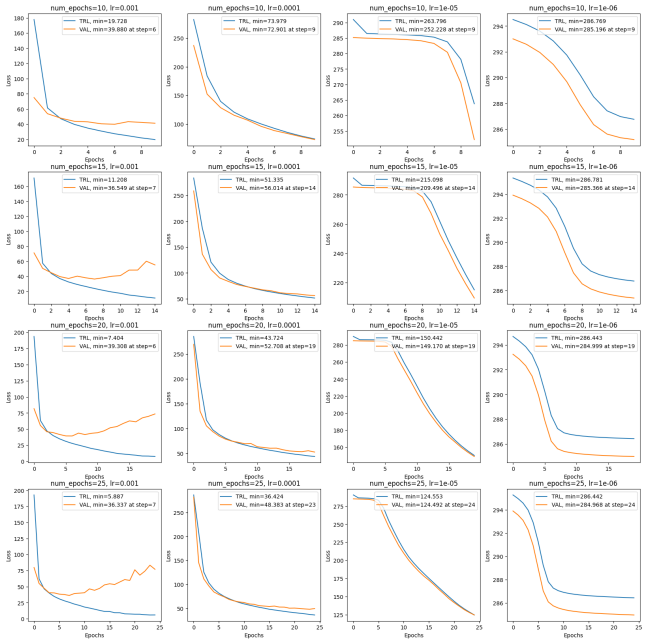

Figure 21: Experiment Subset 2-1: Epoch-loss curves.

The epoch-loss curves of the grid search are shown above. In this experiment subset, we need to make the step sizes of both parameters larger to find the rough range around the possible optimum. The designed candidate epoch numbers and learning rates are:

Epoch Numbers	**Learning Rates**
1. 10	1. 1.0×10^{-3}
2. 15	2. 1.0×10^{-4}
3. 20	3. 1.0×10^{-5}
4. 25	4. 1.0×10^{-6}

Figure 22: Experiment Subset 2-1: Candidate epoch numbers and learning rates.

The epoch-loss curves tell abundant information about the training performance.

From the left-most column to the right-most one, the curve gets smoother, which is a result of the gradually decreasing learning rate. The leftmost column of the grid comes from the model training data with the learning rate of 1.0×10^{-3}. This learning rate is fairly small that all the models converge at around 6 to 7 epochs, and the model starts to over-fit after their convergence.

The learning rate of 1.0×10^{-4} converges slower than its predecessor, not reaching a confident local minimum within all the given candidate epochs. Their ending validation loss is in the range of $[50, 100]$, higher than the over-fitted ending range of $[35, 40]$ from its predecessor. The initial validation loss is also higher, indicating that the learning process gets slower as the learning rate decreases.

For the learning rate of 1.0×10^{-5}, the absolute slope of the model increased after a flat region. Due to the accumulated gradient by Adams, the learning steps are adjusted to become larger after around the 5th epoch. However, the learning rate of 1.0×10^{-5} is too small that Adam can not counter the effect, therefore we can see a larger absolute slope compared to the one in the learning rate of 1.0×10^{-4}, an even higher ending validation loss in the range of $[124, 252]$, as well as an even higher initial validation loss.

The powerlessness of the optimizer is more evident in the last column with the learning rate of 1.0×10^{-6}. This learning rate is in fact too low that it is only less than 100 times the square of the machine epsilon of 64-bit floating point number, which is $\epsilon_{\text{float64}} = 2.22 \times 10^{-16}$, therefore potential gradient vanishing may occur.

From the top-most row to the bottom-most one, the curves appear increasingly "zoomed out." This suggests that while the number of epochs may influence the model's training performance, its effect is not substantial enough to alter the final results. However, the epoch count does control the pace of training, with fewer epochs leading to an earlier stopping point in the training process.

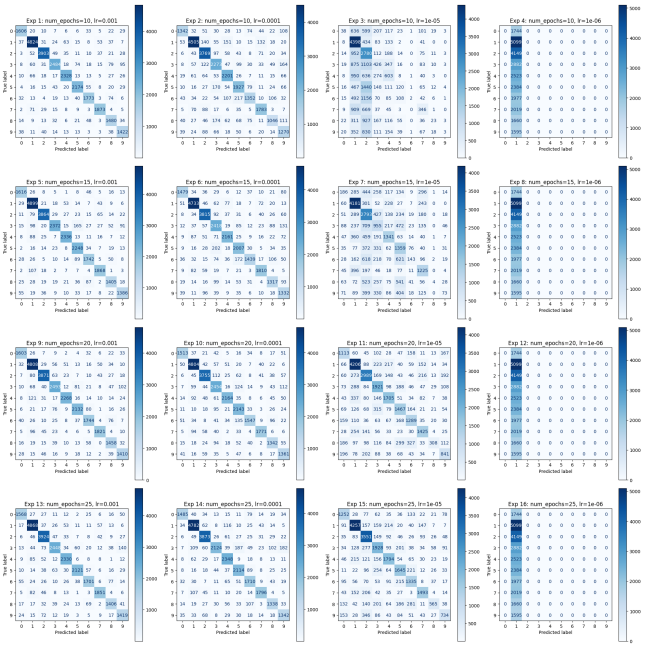

Figure 23: Experiment Subset 2-1: Confusion matrices.

The confusion matrices stand on the same side as the training performance. The columns on the right give confusion matrices whose counter-diagonal is more blurred compared to the columns on the left, while the last column gives a stroke on the 1st label. This indicates that as the learning rate decreases exponentially from a reasonable initial point, the training performance gets worse within the same epochs since the pace is slowed down. For the rows, from to top bottom, the confusion matrices get a clearer counter-diagonal except for the last one, meaning that the more epoch is given, the better the model is trained.

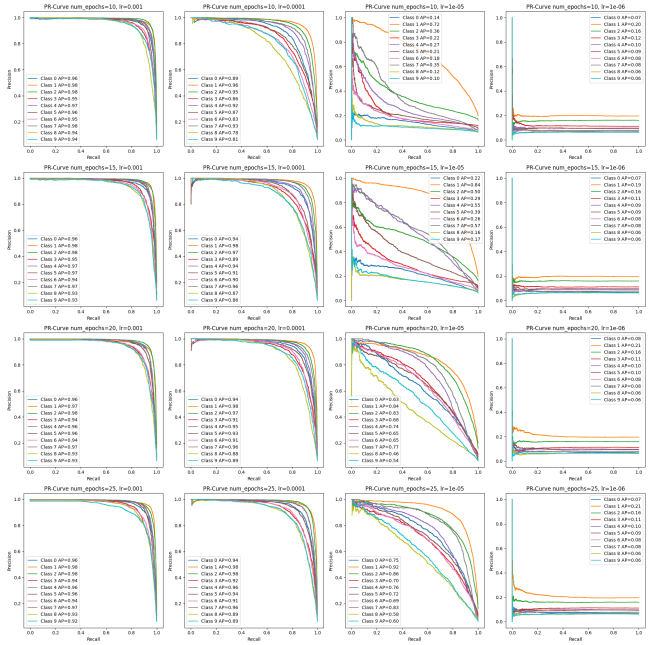

Figure 24: Exeriment Subset 2-1: Precision-recall curves.

The testing process is conducted over the test dataset. The precision-recall curves and the evaluation metrics indicate that the first model is the best at predicting such unseen data, even though it is overfitted. It has the highest accuracy of 0.917 and the best average F_1 score of 0.909.

In the first two columns, the precision-recall curves on each row don't give too much difference, with roughly the same AP scores for each class. However, for the third column with the learning rate of 1.0×10^{-5}, an increase in the epoch number gives an evident increase in the APs. This is because the learning rate is small enough that the gradient is still increasing, making the final results sensitive to the number of epochs. The last column, however, also didn't show much sensitivity over the number of epochs because the learning rate is too low and the model doesn't make any progress within a finite number of epochs.

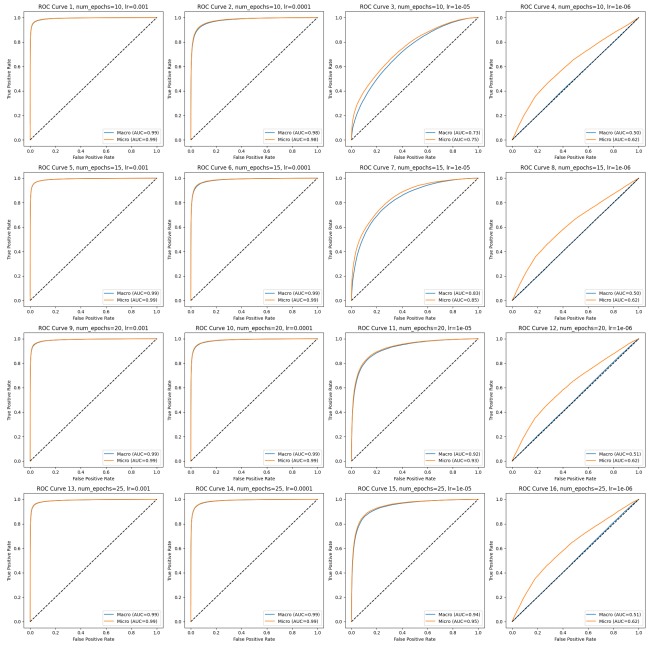

Figure 25: Experiment Subset 2-1: Macro and micro ROC curves.

The macro and micro ROC curves displayed a feature similar to the one of the precision-recall curves.

From the column to the left to the column to the right, the macro ROC curve gradually collapses to the random-splitting diagonal line. For the first two columns, the macro and micro ROC-AUC are all above 0.99, and for the third column, the two values gradually grow from around 0.73 to around 0.95 as the epoch number increases from 10 to 25, emphasizing the sensitivity of the training results over epochs. The last column, being the worst, has a bad macro ROC-AUC of 0.51, and a micro ROC-AUC of 0.62, telling that the models are classified in an extremely class-wise unbalanced manner.

3.2.2 Experiment Subset 2-2: (Detailed Search) Epoch Numbers and Learning Rates

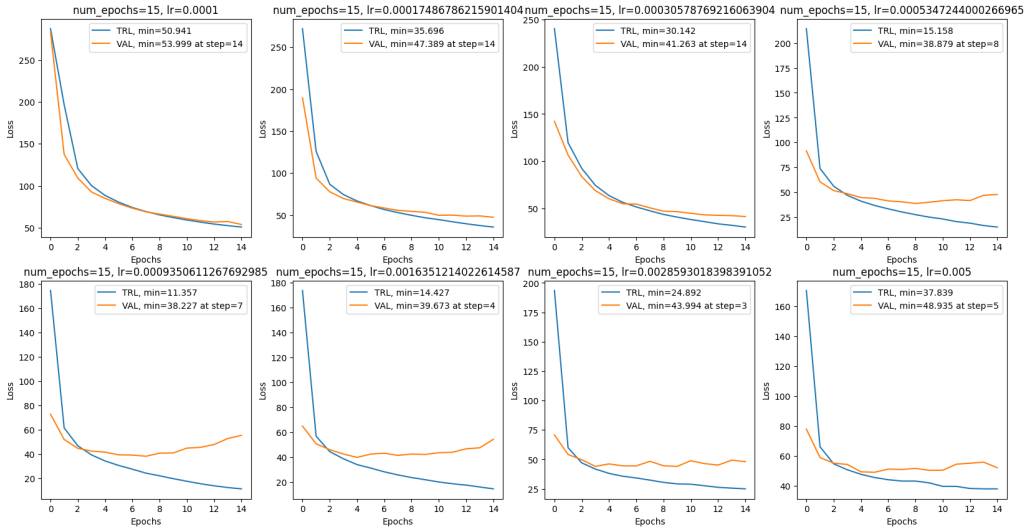

Figure 26: Exeriment Subset 2-2: Epoch-loss curves.

According to the results from the experiment set 2-1, there is a potential local optimum learning rate around the learning rate of 0.01. Therefore, a second search is performed under the fixed epoch number of 15, with a smaller difference between the candidate learning rates.

In this experiment subset, we fix the number of epochs to 15 and conduct a search over the learning rate with the candidates being the geometric series with the starting point of 1.0×10^{-4} and an ending point of 5.0×10^{-3}. According to the epoch-loss given above, as the learning rate gets larger, the over-fitting point becomes smaller due to faster convergence, and the curve becomes less smoother. The optimal learning rate in this experiment is the 5th one, whose value is 9.35×10^{-4}, producing a local minimum validation loss of 38.227, which is inferior to the one produced by the learning rate 1.0×10^{-3} discovered in the previous experiment subset. Since no potential local minimum was discovered in this experiment, it is proposed that 1.0×10^{-3} is the optimum learning rate and will be used in the following experiments.

3.2.3 Experiment Subset 2-3: Batch Sizes

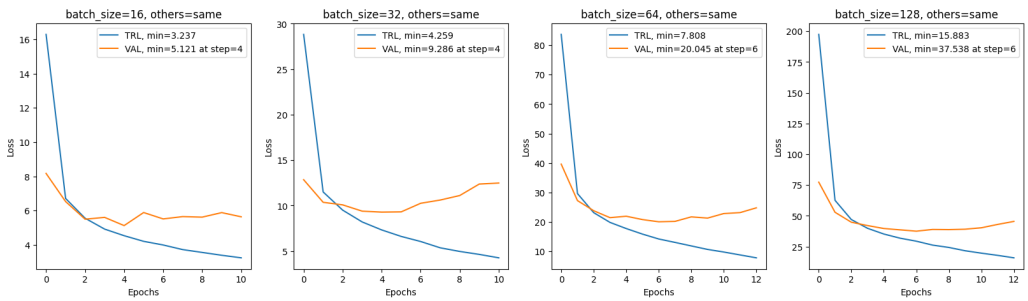

Figure 27: Experiment Subset 2-3: Epoch-loss curves.

The batch size is the size of a chunk of the set data-label pairs in the dataset.

In this experiment, we proposed four batch sizes of 16, 32, 64, and 128. We applied the early-stop mechanism to find a local minimum validation loss, not limiting the epoch number.

As a result, the local minimum points discovered are respectively around 5, 9, 20, and 38. Being divided by their corresponding batch size to get a normalized validation loss value, all four batch sizes produce a minimum loss of around 0.3, being roughly identical.

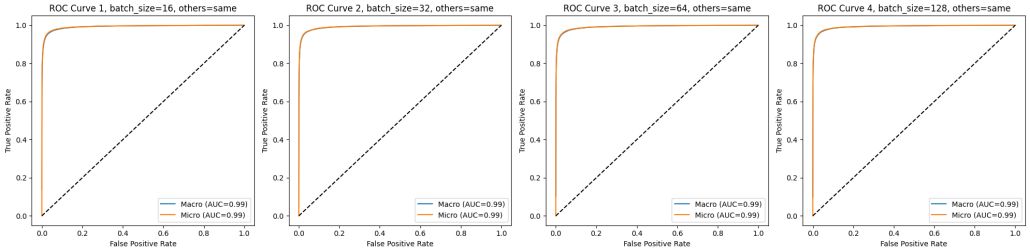

Figure 28: Experiment Subset 2-3: Macro and micro ROC curves.

Lying on the foundation of previous experiments, the current ones produce an identically ideal model whose application performance could be visualized by the ROC curves. All the models produced great macro and micro ROC-AUC scores of around 0.99. Therefore, from both the training and the application's perspective, the batch size does not affect the model's performance significantly enough.

3.3 Experiment Set 3: Augmentation Parameters

Image augmentations are techniques to be used in the training dataset to mimic different lighting and capturing conditions, thus increasing the variety of data to produce a model that's robust enough on unseen data. This process involves the usage of `albumentations` python package. In this experiment set, there will be four augmentation parameters tested: angle, crop, ratio and contrast factor.

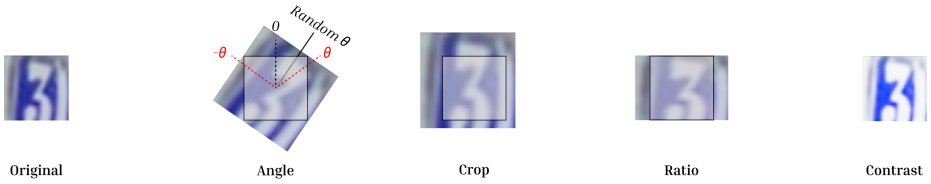

Figure 29: Image augmentation operations.

The angle parameter specifies the limit of the rotation of the image in the directions of both clockwise and counterclockwise. For an angle specified as $\theta \in [0, 180]$, the range of the random rotation of the image is $[-\theta, \theta]$.

Crop defines the minimum boundary of the percentages of the original image's random crop to the original image with respect to area. A crop of $\delta \in [0, 1]$ is given to produce a range of random percentages of $[\delta, 1]$.

The ratio parameter defines the left and right boundary of the stretched aspect ratio of the original image. Once the ratio $\gamma \in [0, 1]$ is given, the range of the random aspect ratio for stretching will be manually defined as $[\gamma, \frac{1}{\gamma}]$.

The contrast factor induces the images' contrast to mimic different lighting conditions. A contrast factor is a positive constant that controls the level of contrast of an image. Given a pixel $\mathbf{p} = \begin{bmatrix} p_R, p_G, p_B \end{bmatrix}^\top$ and a contrast factor $\sigma \in [0, \infty]$, the contrasted pixel would be $\tilde{\mathbf{p}}$ as follows.

$$\tilde{\mathbf{p}} = \begin{bmatrix} min(\sigma \cdot p_R, 255) \\ min(\sigma \cdot p_G, 255) \\ min(\sigma \cdot p_B, 255) \end{bmatrix}, \quad \tilde{\mathbf{p}} \in (\mathbb{N} \cap [0, 255])^3 \tag{17}$$

If the contrast factor is given as σ, a range of the contrast factor will be specified as $[\frac{1}{\sigma}, \sigma]$. Therefore, if the contrast factor given here is large, more lighting conditions could be mimicked, and vice versa.

3.3.1 Experiment Subset 3-1: Angle Limits and Minimum Crops

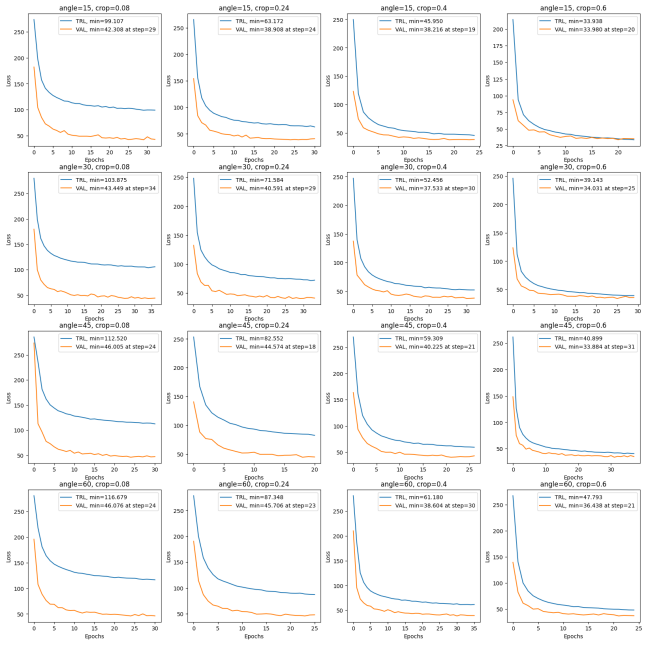

Figure 30: Experiment Subset 3-1: Epoch-loss curves.

This experiment subset conducted a grid search on the angle limits and minimum crops over the following candidate values.

Angles	**Crops**
1. 15	1. 0.08
2. 30	2. 0.24
3. 45	3. 0.4
4. 60	4. 0.6

Figure 31: Experiment Subset 3-1: Candidate angle limits and minimum crops.

From the left-most column to the right-most one, the crop value increases. Correspondingly, the initial and the minimum validation loss decreases. It is reasonable since the crop value defines the minimum boundary of the area percentage so that a larger crop value means more parts of the original image will remain in most of the training dataset, suppressing less challenge to the model.

From the top-most row to the bottom-most one, the angle value increases. If the crop value is small enough, the model training performance will decrease as the angle value increases. Since there is already abundant lost information after cropping, the rotation will further produce noise on training data, making it more randomized and thus bad for training. However, if the crop value is large enough so that more information is preserved, a slight rotation may produce a better training result, since it increases the variety of data and makes the model more regularized, helping it perform better on unseen data. It could be witnessed that, under the crop values of 0.08 and 0.24, the initial and minimum validation loss increases along with the angle, while under the crop values of 0.4 and 0.6, the minimum validation loss in the angles of 30 and 45 could be smaller than the ones under the angles of 15 and 60.

According to the minimum validation values, the best model is the 12th, with a validation loss of 33.884 under the angle value of 45 and the crop value of 0.6, better than the optimum of 36.337 in experiment subset 2-2, which does not include any image augmentation.

Moreover, it is important to address a threat produced by a high rotation angle. As the angle limit gets larger, some data may be mixed and confused. For instance, if an image of the number "6" is rotated 60 degrees, it may get mixed together with the number "9", since they have a similar pattern and one of their key differences is exactly the angle of perspective.

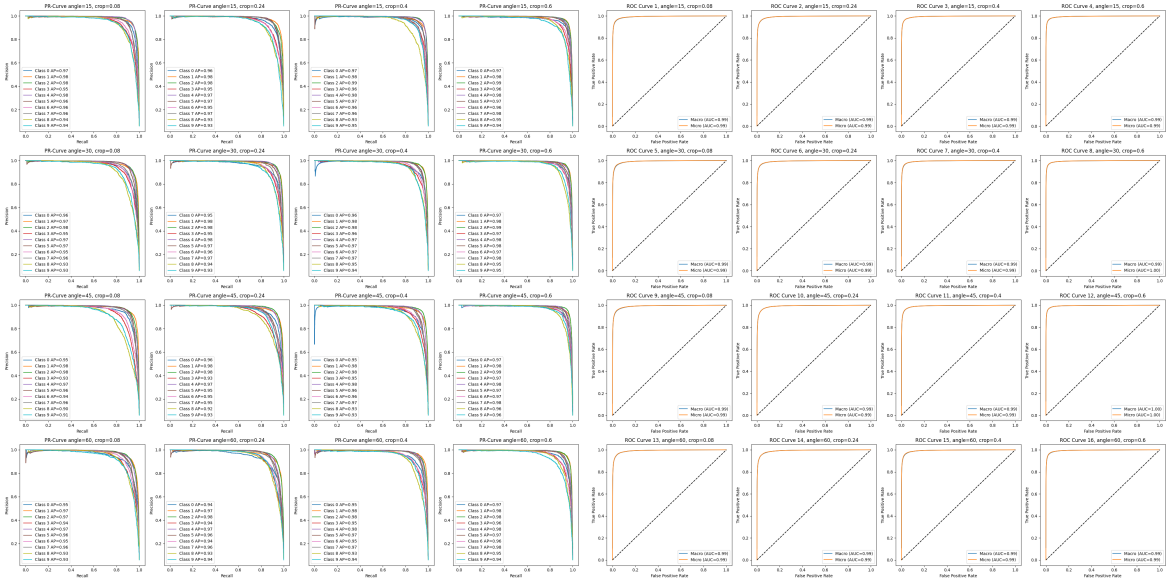

Figure 32: Experiment Subset 3-1: Precision-recall curves and macro-micro ROC curves.

As for the evaluation of the application performance, there is no radical visual difference in either the precision-recall curves or the ROC curves. All models produced an average precision score of around 0.90 to 0.95 and an ROC-AUC score of 0.99 in both macro and micro perspectives. However, it could be witnessed that the precision-recall curves under the crop value of 0.6 experienced a sudden increase in deficiency at the angle value of 60, produced by the threat of a high angle value.

3.3.2 Experiment Subset 3-2: Aspect Ratios and Contrast Factors

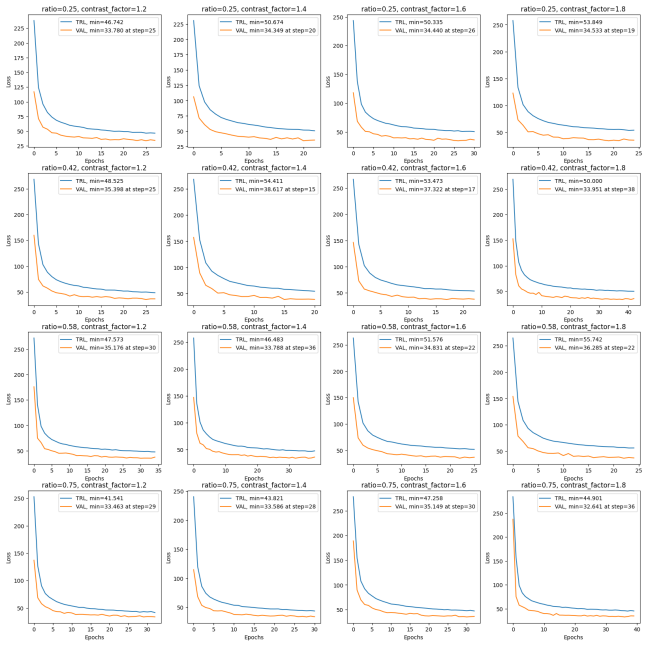

Figure 33: Experiment Subset 3-2: Epoch-loss curves.

In this grid search, the performance of the model training is more stable and tends not to change in an evident manner compared to the last search. All models received a minimum validation loss of around 32 to 35, with the 16th being the optimum of 32.641, under the ratio of 0.75 and the contrast factor of 1.8.

From the top-most row to the bottom-most one, the ratio increases and more pixels are stretched or suppressed, thus the initial validation loss increases.

From the left-most column to the right-most one, the contrast factor increases. However, there seemed to be no obvious tendency of monotonicity in either initial or minimal validation loss with the increase. This may be caused by the dual randomness of image pixels and the ranged random contrast factor. In theory, a larger contrast range should mimic more lighting conditions, producing data variety, but the results could be different for various image quality. For clear images, larger contrast may separate texts and their backgrounds, giving the model a stronger feature to better learn the pattern of the text. For blurred images, raising contrast may emphasize noise pixels and thus bring inference to the model training.

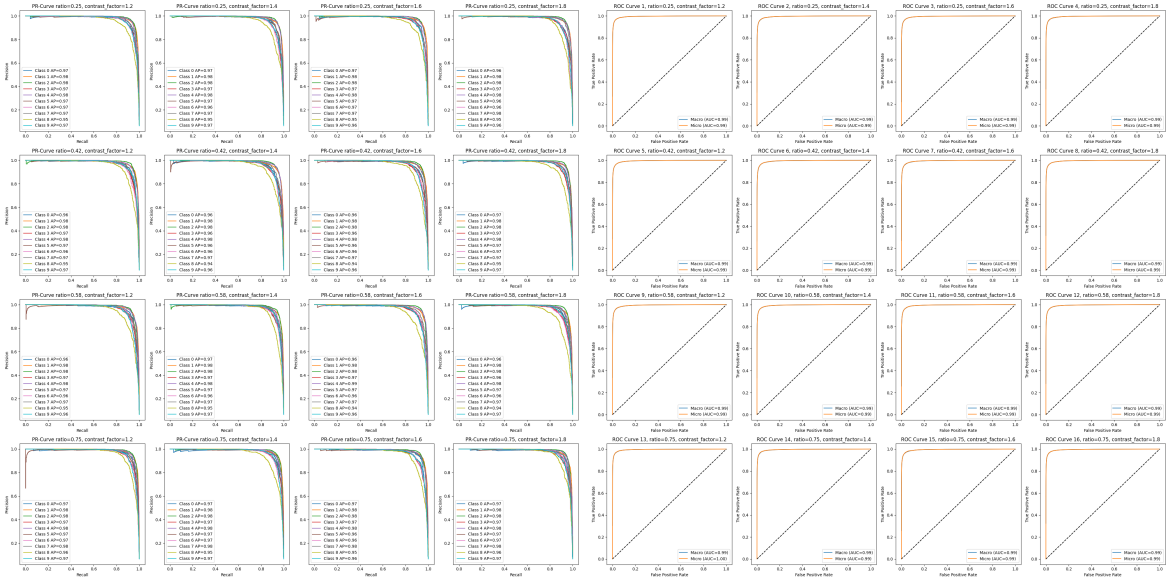

Figure 34: Experiment Subset 3-2: Precision-recall curves and macro-micro ROC curves.

In application, all the models performed equally well, giving an AP score of around 0.99 for all the classes and the ROC-AUC scores of around 0.99 in both macro and micro perspectives. Given that the precision-recall curves of all the classes tend to the top-right corner, and the macro and micro ROC curves overlap, it could be concluded that the classification behavior is balanced in each class.

3.4 Experiment 4: Neural Network Structures and Activation Functions

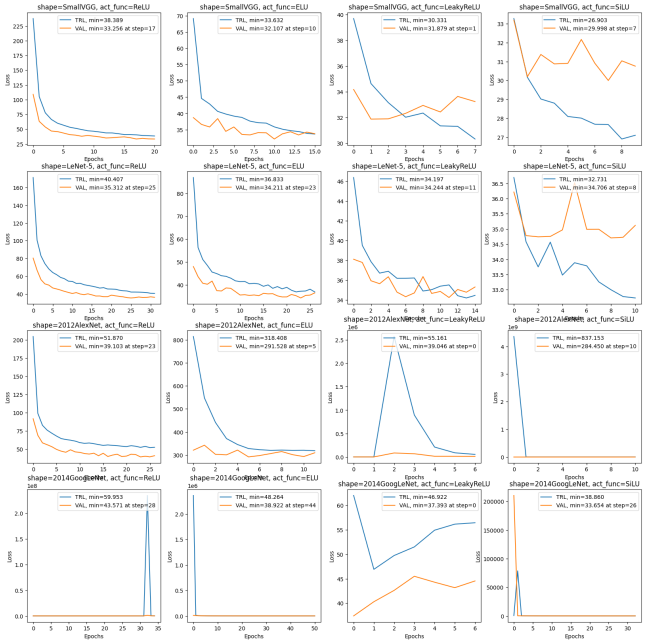

Figure 35: Experiment Set 4: Neural network structures and activation functions.

We would like to further explore how different modern convolutional neural network structures train and perform under the same training and augmentation hyper-parameters. We conducted an extra experiment with a series of candidate model structures of SmallVGG, LeNet-5 [7], AlexNet [8], as well as GoogLeNet [9]. We also applied a series of activation functions to conduct the grid search, being ReLU, ELU, LeakyReLU, and SiLU.

The AlexNet and GoogLeNet seem not to be optimized under the discovered parameters for SmallVGG. For instance, GoogLeNet does not use the activation function parameter, but its training performance still varies exaggeratedly along the changes of activation functions. This means that given the adjusted hyper-parameters discovered in previous experiments, GoogLeNet itself is unstable in the training process. For the first, second, and fourth plots, there was a sudden and dominating huge increase in either the training or the validation loss and worked out a minimum validation loss at the epoch number of around 30 to 40; while in the third plot, the experiment stopped at the oddly early stage of 7 epochs and produced the minimum validation loss of 37.393 at the first epoch.

For SmallVGG and LeNet-5, the effect of different activation functions on the model's training performance is relatively more evident. The images are normalized using mean normalization, which may produce negative output values in the feature maps. Correspondingly, the candidate activation functions differ only in their activations on the negative values. When a bunch of weight scalars to be activated is normalized to be negative, ReLU suppresses all the values to 0; ELU uses a biased exponential function with the base of the natural number e; LeakyReLU uses a direct proportional function; and SiLU only activates values that are close to 0.

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{18}$$

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha \cdot (e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{19}$$

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha \cdot x & \text{if } x \leq 0 \end{cases} \tag{20}$$

$$\text{SiLU}(x) = \frac{x \cdot e^x}{1 + e^x} \tag{21}$$

Interestingly, from all the candidate activation functions, LeakyReLU considers the negative values the most, resulting in more information being preserved. Fortunately, the noise in the dataset seemed not abundant enough to mess up with the training, therefore the model could learn slightly more information in an epoch, leveraging an earlier stop and a better minimum validation loss.

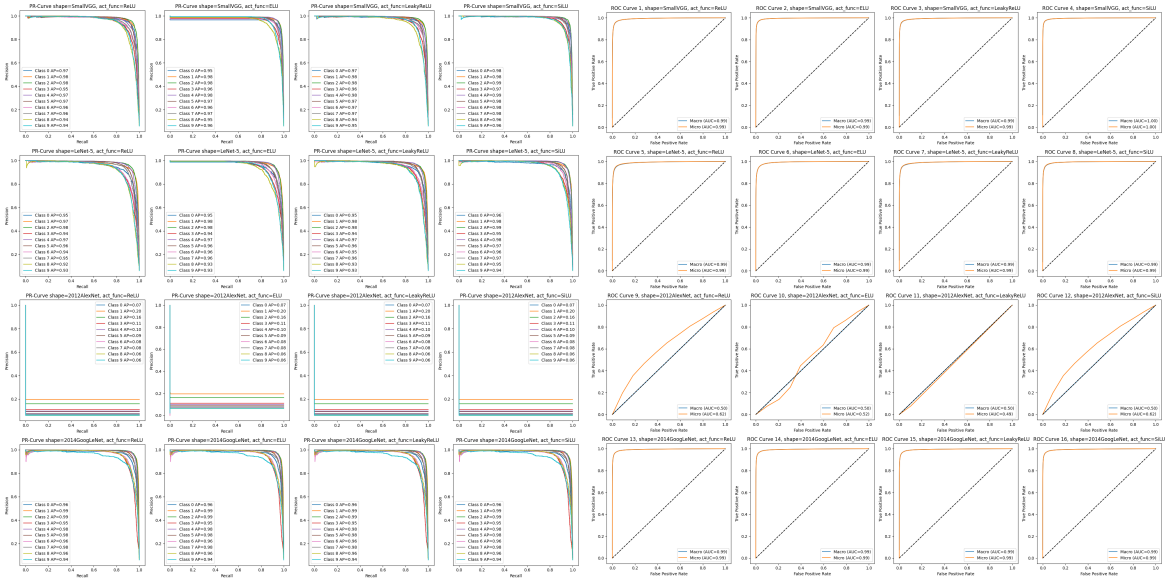

Figure 36: Experiment Set 4: Precision-recall curves and macro-micro ROC curves.

Despite the training performance being randomized, the application result is fairly good for all the models except for AlexNet. For other models, the AP scores are all roughly around 0.95 and the ROC-AUC scores are leveraged around 0.99; while for AlexNet, the AP scores for almost all classes are around 0.01, and the macro and micro ROC-AUC scores are around 0.5 and 0.6, indicating an imbalance in classification performance among classes.

4 Conclusion

4.1 Summary of Methodologies and Discoveries

In this project's preparation process, we first created the `SVHNDataset` class with useful features to better manipulate different branches of data, including train-valid splits of the SVHN train dataset, as well as adding normalization and augmentations. Then, we built a small VGGNet that contains 3 convolutional processes that map a 32×32 colored image into a feature vector of 512 dimensions. Eventually, we made use of Python dictionaries to store the model states along with related information during the training process and load the information during the testing process in order to perform grid searches effectively.

We conducted 4 sets of experiments. In the first experiment, we explored different optimizers, concluding that the RMSprop family, especially for the optimizer of Adam, gives the best performance when initialized in the simplest manner. In the second experiment set, we explored the effects of epoch numbers and learning rate on the model's training performance and found an optimal learning rate using a rough search and a detailed search. Moreover, we explored different batch sizes and discovered it having little effect on the overall training performance. In the third experiment, we explored augmentation parameters of random angles, crop percentages, aspect ratios, and contrast factors. We discovered that crop percentages could affect the model's training performance and high crop percentages with more information preserved could display sensitivity in the training performance on angle change. Contrast factors and aspect ratios, however, do not display an obvious effect on model training due to the abundance of randomness. Lastly, we conducted an extra experiment to explore the training and application performance of differently structured models on the discovered optimum hyper-parameters for SmallVGG, along with a set of candidate activation functions. While the training performance of all the models has a different pattern compared to SmallVGG, most models performed identically well in the application except for AlexNet. From all the activation functions, LeakyReLU produces the earliest stopping point for training.

4.2 Future Improvements

There is a triple randomness of contrast factor, image pixels, and train-valid split in the experiment on augmentation. The contrast factor applied on each image that is used to calculate the mean and standard deviation values is not exactly the same as the one that's used during training due to the apply-on-get mechanism of the `DataLoader` class and the randomness of the distribution of contrast factors. Although the contrast factor is set to be uniformly distributed, the image pixels are not, and thus the mean and standard deviation of the training dataset may be biased, In further works, normalization factors may be introduced to address this issue to generate results with better credibility.

There is no cross-grid search. Namely, each parameter is only applied in exactly one grid search, which may bring inaccuracies if there is a 3-factor correlation on hyper-parameters. In future works, each hyper-parameter should join at least 2 grid searches in an attempt to find potential correlations of such.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large scale image recognition," *arXiv preprint arXiv:1409.1556v6*, 2014.

[2] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[5] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, JMLR: W&CP, 2013.

[6] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 07 2011.

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, p. 84–90, May 2017.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.