Introduction to Deep Learning Assignment 2

Group 53

Chenyu Shi (s3500063); Shupei Li (s3430863); Shuang Fan (s3505847)

November 24, 2022

Task 1

1.1 Official Repository Examples

We reproduce the results of `mnist_mlp.py` and `mnist_cnn.py` for the MNIST dataset. Without adjusting model architectures, Table 1 reports our experimental results.

Table 1: MNIST: Official Repository Examples

Model	#Epoch	Test Loss	Test Accuracy
MLP	20	0.1206	0.9837
CNN	12	0.4189	0.8891

1.2 Exploratory Experiments

Based on the experience we learned from Keras official documentation, we set the model architectures provided in the textbook as benchmarks and try different hyperparameter settings on Fashion MNIST and CIFAR-10 datasets. It is worth noting that we set the number of epoches to 50 and use early stopping technique during the hyperparameter tuning process. We also add model checkpoints to save the weights of the best model and load the weights when evaluating the performance. The combination of early stopping and model checkpoints dynamically balances between training costs and model performance. The following describes our experiments in detail.

1.2.1 Fashion MNIST

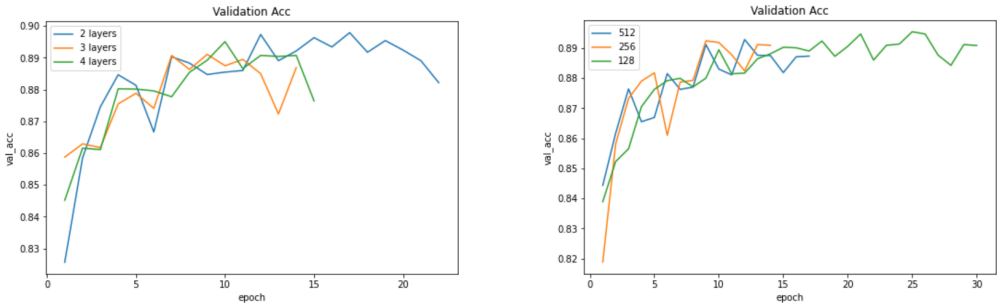
MLP. Firstly, we adjust the number of layers as well as hidden layer units to determine the best model architecture. Table 2 summarizes the results of different model settings. Figure 1 illustrates the changing trends of validation accuracy during the training.

Table 2: MLP Model Architectures

Parameter	Value	Test Loss	Test Accuracy
#layers	2	0.3679	**0.8911**
	3	0.3622	0.8843
	4	0.3583	0.8842
#units	128	0.3659	**0.8876**
	256	0.3302	0.8847
	512	0.3539	0.8858
Baseline	-	0.6186	0.8863

According to the results, MLP model with two layers and 128 units is the best model. Therefore, we adopt this architecture when exploring other hyperparameters. We follow the hints in the assignment description and experiment with initializations, activations, optimizers, and regulariztions. Table 3 summarizes the options we have tested.

We begin with models without Dropout layers, considering the relatively large parameter space. Grid search method is used to optimize hyperparameters, which leads to 81 different settings. The detailed results can be



(a) #layers **(b)** #units

Figure 1: MLP: Validation Accuracy

Table 3: MLP Hyperparameter Tuning

Category	Parameter	Values
Initialization	kernel initializer	RandomNormal, GlorotNormal, GlorotUniform
Activation	layer activation	relu, sigmoid, tanh
Optimizer	optimizer	RMSprop, Adam (learning_rate=0.001), Adam (learning_rate=0.01)
Regularization	kernel regularizer	None, L1, L2
	Dropout layer	With / Without one dropout layer after the hidden layer.
	dropout rate	0.1, 0.3, 0.5 (The dropout rate when adding the Dropout layer)

found in the jupyter notebook. We omit results here due to the limited space. The best model without dropout layers has the setting: {kernel initializer: GlorotUniform, layer activation: sigmoid, optimizer: RMSprop, kernel regularizer: None}. Based on that, we add dropout layers to the model with different dropout rates. Table 4 lists the top three MLP model settings. The layer activation, kernel initializer, and optimizer of these settings are relu, GlorotUniform, RMSprop respectively. All settings don't use the kernel regularizer.

Table 4: Top Three MLP Model Settings

Model	#layers	#units	Dropout Layer	Test Accuracy
Top 1	2	(512, 10)	Yes. rate=0.3.	0.8919
Top 2	2	(512, 10)	No.	0.8911
Top 3	2	(128, 10)	Yes. rate=0.1.	0.8891

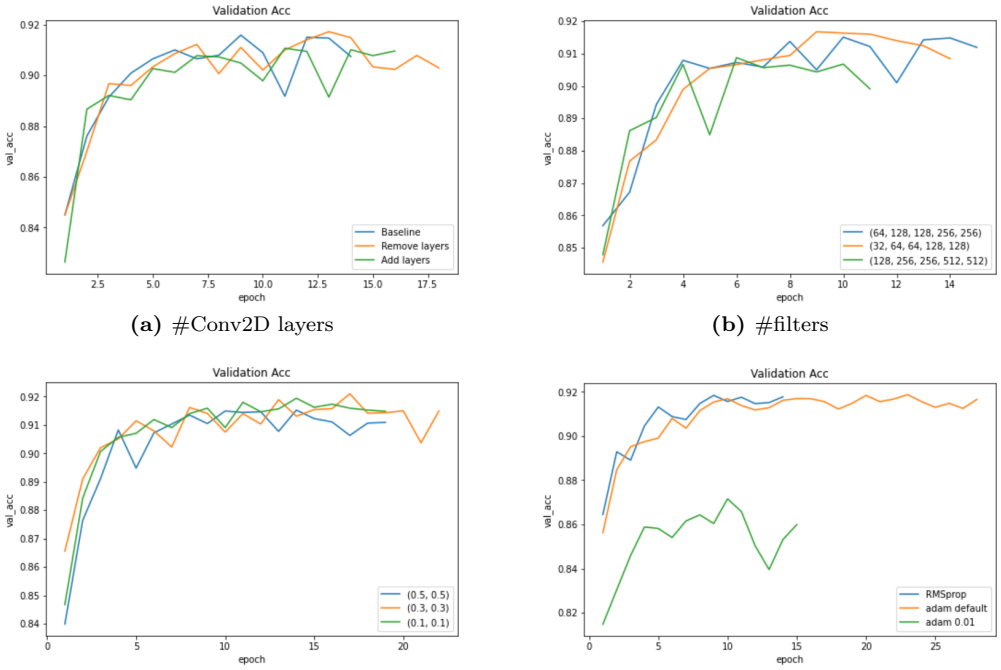
CNN. We follow the similar workflow as MLP experiments to explore the best CNN model. First of all, we try to change the number of Conv2D layers and convolutional filters. Table 5 and Figure 2 (a)(b) report experimental results of model architecture modification.

Table 5: CNN Model Architectures

Parameter	Value	Test Loss	Test Accuracy
#Conv2D layers	5	0.2642	**0.9109**
	6 (Add layers)	0.3105	0.9060
	3 (Remove layers)	0.3360	0.9101
#filters	(64, 128, 128, 256, 256)	0.2767	**0.9094**
	(32, 64, 64, 128, 128)	0.2916	**0.9094**
	(128, 256, 256, 512, 512)	0.2907	0.9042
Baseline	-	0.3330	0.8909

The best model architecture is adding five Conv2D layers with (32, 64, 64, 128, 128) filter setting. Due to the high training cost of CNN, we reduce the parameter space compared to MLP. To be specific, we experiment with different dropout rates and optimizers. Table 6 and Figure 2 (c)(d) show the results.

Table 7 lists the top three CNN model settings. All settings use the architecture with five Conv2D layers and (32, 64, 64, 128, 128) filter setting.



(a) #Conv2D layers **(b)** #filters

(c) dropout rate **(d)** optimizer

Figure 2: CNN: Validation Accuracy

Table 6: CNN Hyperparameter Tuning

Parameter	Value	Test Loss	Test Accuracy
dropout rate	0.1	0.3279	**0.9183**
	0.3	0.3613	0.9175
	0.5	0.3713	0.9105
optimizer	RMSprop	0.2768	0.9111
	Adam (learning rate=0.001)	0.4071	**0.9116**
	Adam (learning rate=0.01)	0.3658	0.8732

Table 7: Top Three CNN Model Settings

Model	dropout rate	optimizer	Test Accuracy
Top 1	0.1	RMSprop	0.9183
Top 2	0.1	Adam (learning rate=0.001)	0.9116
Top 3	0.5	RMSprop	0.9109

1.2.2 CIFAR-10

We set the models provided in the textbook as baselines for CIFAR-10 data. To test whether performance gains can be translated, we directly use MLP settings in Table 4 and CNN settings in Table 7 on CIFAR-10. Besides, we repeat the hyperparameter optimization process to find other possibly better hyperparameter settings. The best MLP new settings are: {#layers: 4, #units: (512, 128, 32, 10), layer activation: relu, kernel initializer: GlorotUniform, kernel regularizer: None, Dropout layers: Yes. rates=(0.1, 0.1, 0.1)}. And the best CNN new settings are: {#Conv2D layers: 5, #filters: (64, 128, 128, 256, 256), dropout rate: (0.1, 0.1), optimizer: Adam (learning rate=0.001) }. Table 8 summarizes the results of our experiments on CIFAR-10.

According to Table 8, the performance gains don't translate to a different dataset.

1.3 Discussion

Hyperparameter tuning plays an important role in improving model performance. Deep learning mdoels usually have many hyperparameters. Therefore, we need to use a systematic approach to optimizing hyperparameters. For example, we can apply grid search technique on the parameter space. If the model architecture is very

Table 8: CIFAR-10: Performance of Different Settings

Category	Model	Test Loss	Test Accuracy
MLP	Baseline	1.4616	0.4977
	Top 1 in Fashion MNIST	1.5321	0.4559
	Top 2 in Fashion MNIST	1.4960	0.4735
	Top 3 in Fashion MNIST	1.5921	0.4304
	MLP Best New Settings	1.4300	**0.4985**
CNN	Baseline	1.5516	0.6478
	Top 1 in Fashion MNIST	0.9998	0.6969
	Top 2 in Fashion MNIST	1.0246	0.7004
	Top 3 in Fashion MNIST	0.9959	0.6802
	CNN Best New Settings	1.1895	**0.7024**

complicated, we may need to use more advanced tuning frameworks, such as Ray Tune and Hyperopt. When exploring the best settings, it is helpful to set callbacks in Keras models to control the training process dynamically. It is worth noting that the best hyperparameter setting on one dataset generally doesn't obtain the best performance on another dataset. That means we should perform hyperparameter tuning on a new dataset even the model structure remains the same.

Task 2

2.1 Regression

The regression model structure is shown in Figure 3. And its corresponding results are shown in Table 9. Considering this is a regression task, we choose MSE as loss function and Adam as optimizer. This model includes 2,207,593 trainable parameters, which is relatively large compared to other models in Task 2. However, the final common sense loss is still slightly high, which is 0.7053 hours (about 42.3 minutes). This is mainly caused by the inconsistency between label representation and common sense definition. In other words, representing time in this form disobeys common sense. For example, 11:55 is represented as 11.917, while 0:05 is represented as 0.0833. Even though the difference between 11:55 and 0:05 is only 10 minutes in common sense, their corresponding labels (11.917 and 0.0833) lead to a very high MSE. Therefore, the regression model has a limited final performance.

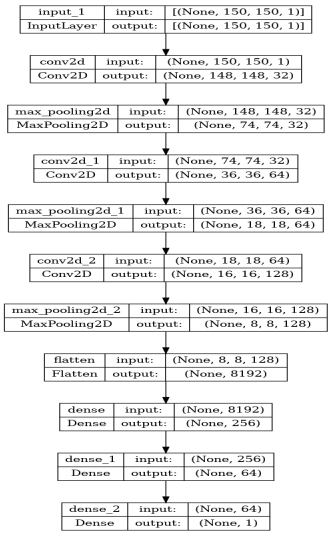


Figure 3: Regression Model

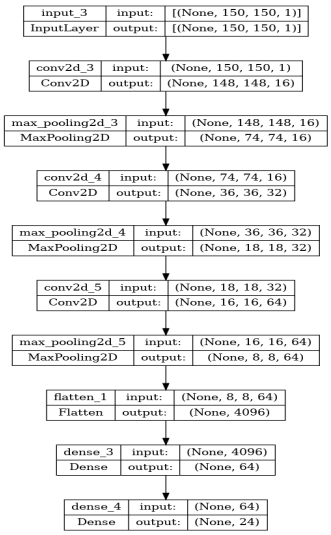


Figure 4: Classification Model

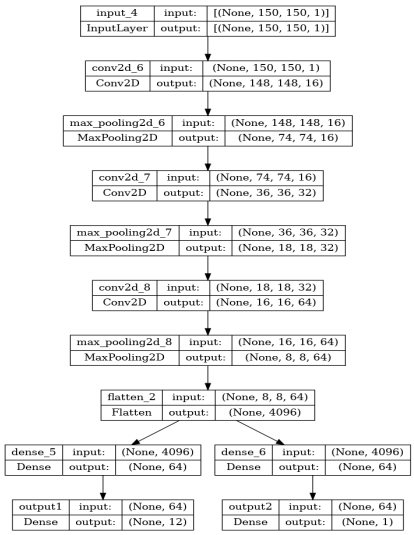


Figure 5: Multi-head Model

2.2 Classification

Figure 4 illustrates the architecture of classification model. Table 9 reports the corresponding results. We experiment with 24, 72, and 720 classes in the classification task, where models are only different in the last

Table 9: Results and Algorithm Comparison

Model	Common Sense Loss(hours)
Regression	0.7053
Classification(24 classes)	0.6206
Classification(72 classes)	0.9517
Classification(720 classes)	3.0309
Multi-head	0.6161
Label transformation	0.3447
Label transformation(fine-tuned)	**0.1921**

output layer. We use the cross entropy loss function and the Adam optimizer, which are the common choices for multi-class classification task. Although the classification model has much fewer trainable parameters(287,064 trainable parameters for 24 classes), the performance of 24 classes model achieves 0.6206 hours (about 37.2 minutes), which is better than the regression model. However, the increase of the categories impairs model performance. In the case of 720 classes, the model even doesn't converge and has a very high common sense loss. The reason is that using time intervals as class labels has many drawbacks. The first problem is that this kind of label representation can't measure the degree of difference between two categories. For example, 0:01 and 0:31 belong to different categories in 24 classes model. Meanwhile, 0:01 and 6:00 are also in different categories. Although the common sense loss between 0:01 and 6:00 is much larger than that between 0:01 and 0:31, the cross entropy loss function assigns the same loss values to these two cases. The second problem is that the trade-off between the length of sampling interval and the number of samples is hard to be determined. If we group the samples into fewer classes, each class will cover a longer time interval and therefore have more samples. Then, the model will be well trained because there are sufficient samples in each class. Nonetheless, the common sense loss of each class will become rather high. We still use the model with 24 classes as an example. In this model, 0:01 and 0:29 are classified into the same class even they are almost half an hour apart, which causes the major deviation from the common sense. On the contrary, if we classify the samples into more classes whose corresponding time interval is shorter, the common sense loss within each class will decrease. However, the model itself will be underfitting due to the deficiency of samples for each class. Considering the case of constructing model for 720 classes, the average number of samples for each class is only 25, which is generally insufficient for training CNN.

2.3 Multi-head

We display the model structure and corresponding results of multi-head model in Figure 5 and Table 9. In this model, we build the model with two heads, one for predicting hours and another for predicting minutes, and we consider the two-head outputs as multi-class classification task and regression task respectively. For classification task of predicting hours, we consider 12 hours as 12 different categories and use cross entropy as loss function. For regression task of predicting minutes, we use MSE as loss function. And this model uses 548,557 trainable parameters and finally achieves 0.6161 hours(about 37.0 minutes) on common sense loss, which is the best performance up to now compared to regression and classification models. The performance improvement is due to that using multi-head model can take advantage of different features of the target labels. For example, for target of predicting hours, it's a good way to consider it as a classification task because there are only 12 classes and each class can have adequate training samples. On the contrary, for target of predicting minutes target, it's a good way to consider it as a regression task because the target values are in a rather large interval(0-59). It's flexible to use multi-head model to combine these features using different losses and targets, so the final performance can be improved. Besides, there's an important thing needed to pay attention to. That is for regression task, we should scale and normalize predicting target from interval(0-59). For example, divide 60 for each minute target. Otherwise, the cross entropy loss for hours and the MSE loss for minutes will have great difference in quantity, which has an negative impact on training neural networks. However, although multi-head model has gained performance improvement, the similar problems of labels as in classification and regression tasks are still there. Therefore, in the next subsection, we will use Label transformation to improve model and get much better results.

2.4 Label transformation

According to the performance of previous model, we can know that directly using hours and minutes as target fails to capture common sense loss well, which then is inconsistent to commonse sense and has a negative influence on the model performance. Therefore, we use sine and cosine functions to represent the angles on the unit circle to improve the model performance.

It's easy to think that the angle of hour hand θ contains entire information about the time on the clock. Then we apply sin function on that angle θ. Now let's see the difference of previous example, 0:05 and 11:55. It's simple to compute the corresponding angles of 0:05 and 11:55, which are 0.0436 rads and 6.240 rads. And we could see, sin(0.0436) and cos(6.240) equals to 0.0436 and -0.0436 respectively, which are very close to each other due to the property of periodic function. It's exactly what we want for our labels.

But there is another problem. For sin function, there are many hour hand angles are projected to a same value even if they represent different time. For example, 1:00($\frac{\pi}{6}$) and 5:00($\frac{5\pi}{6}$) are both projected to 0.5. Luckily, we could apply cos function to distinguish these angles. Therefore, we need to transform the original label [hour,minute] to [sin(θ),cos(θ)]. So the final output nodes should be 2, one for sin, and another for cos. We know sin and cos function both have a range of [-1,1]. In addition, we know tanh activation function has the same range. Therefore, applying a tanh activation function in the output layer would be perfect. And we consider it as a regression problem as well.

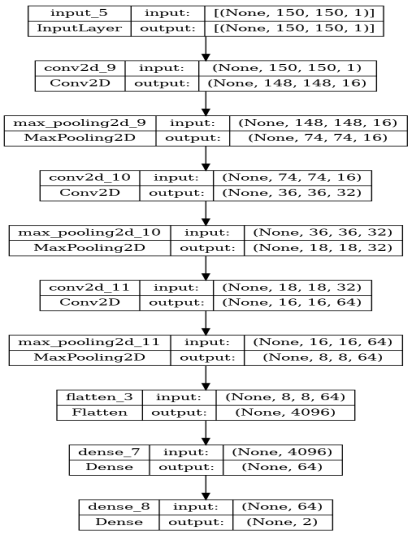


Figure 6: Label Transformation Model

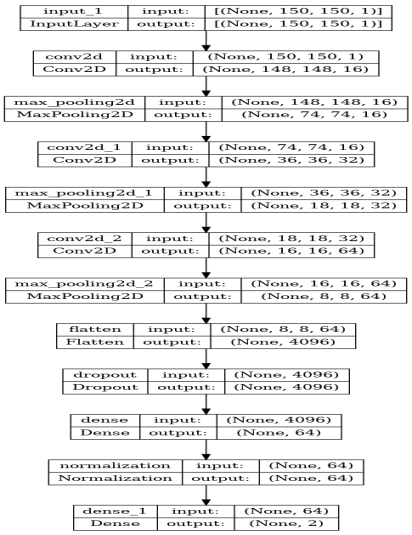


Figure 7: Final Model

This label transformation regression model works well. The specific model structure and model performance are shown in figure 6 and table 9 respectively. It only takes 285634 trainable parameters and finally obtains 0.3447 hours(about 20.7 minutes) common sense loss, which is the best up to now. In the next section, we will fine-tune this model using the knowledge gained by previous part and improve its performance as the final model.

2.5 Final model

The final model structure and its performance are shown in figure 7 and 9 respectively. Comparing to the model in section 2.4, we made several adjustments. Firstly, we use leakyrelu instead of relu as activation function in all hidden layers to avoid vanishing gradient problem. In addition, we apply L2 regularization for each layer that contains trainable parameters to avoid overfitting. Besides, a dropout layer is applied after flatten and a normalization layer is applied after the first dense layer. Finally, we set checkpoints to store the model with best performance on validation set, and use a smaller learning rate and larger total training epochs.

The final performance obtained by this model achieves 0.1921 hours(about 11.5 minutes) common sense loss, which is the best one among all models, especially compared to the first three models. That's mainly because periodic functions sin and cos can exactly capture the common sense loss of the clock time. To conclude, from this experiments we can see, it's of much importance to design a good target label for neural networks to train and obtain a rather good final performance. Also, we could see a fine-tune can improve model's performance as well.

Task 3

3.1 Datasets

We use two datasets in Task 3. Firstly, we explore the performance of different model architectures and the effects of different parameters with MNIST data. After that, we leverage the power of generative models on Butterfly & Moth data.

We directly call Tensorflow API to download the MNIST dataset. However, the original dataset is also available on https://deepai.org/dataset/mnist. MNIST dataset contains 70,000 grayscale images (28 × 28 × 1), whose content is handwritten numbers.

Butterfly & Moth is an open source dataset on Kaggle. There are 13,639 RGB images (224 × 224 × 3) composed of 100 butterfly or moth species. Link of the dataset is https://www.kaggle.com/datasets/gpiosenka/butterfly-images40-species?resource=download.

3.2 Experimental Set-up

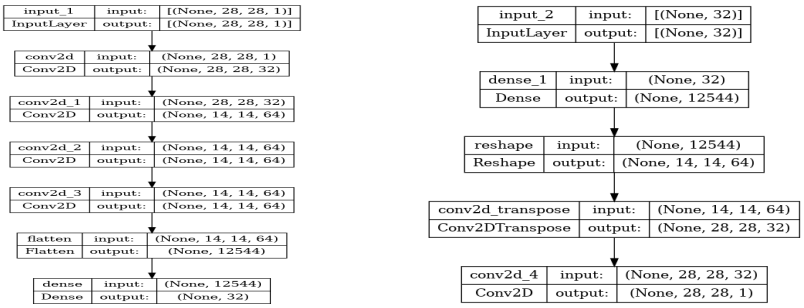
All experiments are deployed on two servers. Server 1 has an Intel(R) Xeon(R) Platinum 8358P CPU and a RTX A5000 GPU, while server 2 has an Intel(R) Xeon(R) E5-2680 v4 CPU and a TITAN Xp GPU. Table 10 summarizes the hyperparameter values in the experiments. The following describes the process of our experiments.

Table 10: Hyperparameter Settings

Parameter	Value	Meaning
cae_latent_dim	32	Dimensions of the latent space in CAEs.
cae_epoch	10	The number of training epochs in CAEs.
vae_latent_dim	32	Dimensions of the latent space in VAEs.
vae_epoch	20 (MNIST) / 100 (Butterfly & Moth)	The number of training epochs in VAEs.
gan_latent_dim	256	Dimensions of the latent space in GANs.
gan_epoch	20 (MNIST) / 250 (Butterfly & Moth)	The number of training epochs in GANs.

3.2.1 MNIST

We modify the model architecture to decrease the model complexity and match the data better. Specially, we build the basic convolutional network with four Conv2D layers and construct the basic deconvolutional network with one Conv2DTranspose layer. Figure 8 illustrates our model settings. There is no need to resize the images due to the modification of the model.



(a) Convolutional network **(b)** Deconvolutional network

Figure 8: Model Structure

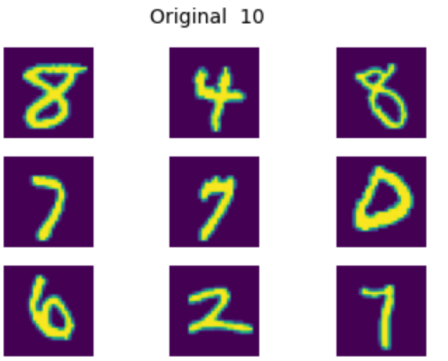
3.2.2 Butterfly & Moth

We rescale the images to 64 × 64 × 3 and directly apply the model architecture provided in the notebook when working with Butterfly & Moth data.

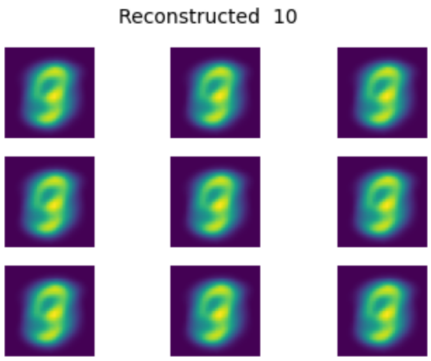
3.3 Results

CAEs. Figure 9 shows the reconstructed images from CAEs. It is easy to see that CAEs have captured the main features in the original images.

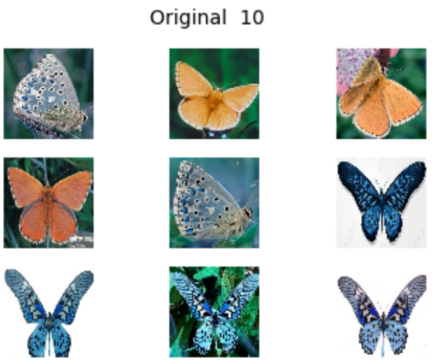
VAEs. We explore the learned latent space with linear interpolation technique. Firstly, we sample a point from the latenty space by generating its coordinates from a standard normal distribution. Then, we change one or two coordinates along the straight line in the latent space, while keep other coordinates unchanged. For MNIST dataset, we apply linear interpolation on the 10th and 27th coordinates simultaneously, which are related to



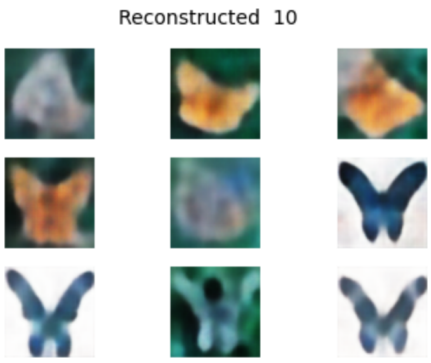
(a) MNIST: Original Images



(b) MNIST: Reconstructed Images



(c) Butterfly & Moth: Original Images



(d) Butterfly & Moth: Reconstructed Images

Figure 9: Results of CAEs

the shape of the number. Figure 10 shows the visualization results. As for Butterfly & Moth data, we linearly

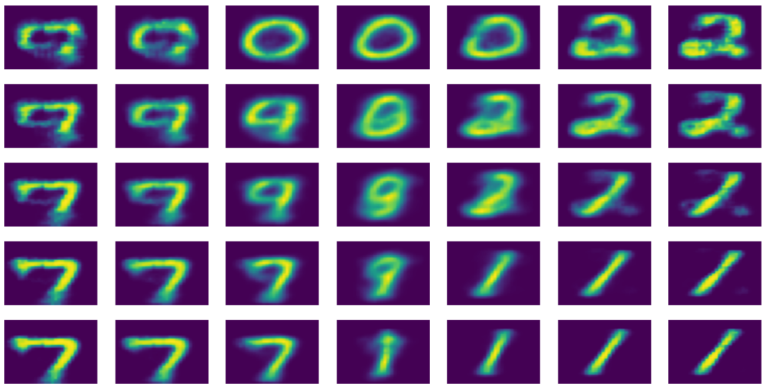


Figure 10: VAEs: MNIST Data

interpolate the 6th and 29th coordinates, whose outputs are presented in Figure 11. According to Figure 11, the 6th coordinate is related to the color of wings, while the 29th coordinate is concerned with the width of wings.

GANs. We visualize the outputs of linear interpolation in the same way as VAEs. We change the 1st, 120th, and 126th coordinates for MNIST data to obtain different numbers. Figure 12 shows the generated images. For Butterfly & Moth data, we change the 40th coordinate, which is related to the color and the posture of the butterfly. Figure 13 shows the generated butterfly.

3.4 Discussion: Model Comparison

CAEs are autoencoders that apply the CNN architecture to compress the input images into a lower dimensional latent space and reconstruct the images from the learned latent space. Using an encoder/decoder structure enables CAEs to capture as much information about data as possible, even without labels. However, CAEs are deterministic. That means there is a one-to-one relationship between the input and output in CAEs. Therefore,

Figure 11: VAEs: Butterfly & Moth Data

Figure 12: GANs: MNIST Data

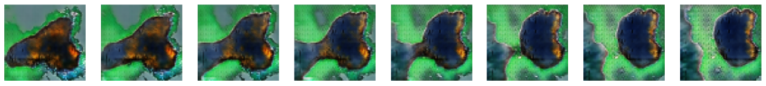


Figure 13: GANs: Butterfly & Moth Data

CAEs can't generate new samples. Based on the architecture of the traditional autoencoder, VAEs introduce randomness into the model by assuming a prior distribution of latent space and infering the posterior distribution during the training process. In most cases, we choose the standard Gaussian distribution as prior distribution, which helps the latent space to be complete and continuous. The probabilistic nature allows VAEs to generate new images from random noise.

GANs are designed for generating new samples. Instead of inferring the distribution of latent space, GANs sample from random noise and learn a transformation to imitate the real distribution of data. Gans improve the quality of imitations by training a discriminator to distinguish between generated samples and real samples.

In summary, VAEs sample from a prior distribution and infer the real distribution of latent space, while GANs sample from random noise and learn the data transformation by encouraging the competition between generator and discriminator.

Contributions

Name	**Contribution**
Chenyu Shi	Task 1 code, Task 2 code, Task 2 report.
Shupei Li	Task 1 code, Task 1 report, Task2 code, Task 3 code, Task 3 report.
Shuang Fan	Task 1 code, Task 1 report.