Gesture Recognition Based On Deep Learning

Runlin Hou Sifan Yuan Yuxiang Song
Haocong Wang

December 10, 2020

Abstract

Gesture recognition is a topic in computer science, mainly talking about making computers recognize human body movements. It can be captured from all parts of the human body, but it mainly comes from faces and hands. In our project, we focus on hand movements. In order to present a better result, we choose gestures of numbers as our training and testing object. Our project is based on PyTorch, an open-source machine learning library. We choose LeNet and ResNet as the network of the model and compare the performance between the two networks. Finally, we implement a product using our trained model to evaluate its feasibility.

1 Introduction

Gesture recognition is a vital approach to make the direct communication between human and computers achievable without extra mechanical devices compared to primitive text user interfaces and traditional GUI. It can be widely utilized in applications, such as virtual reality and sign language. In our daily lives, we already contact with gesture recognition in a high frequency. For example, one of the commonly used electronic devices, iPad, supports gesture operation, which requires the ability of gesture recognition. Gesture recognition has a broad range of applications in human-computer interaction. For example, in virtual reality and augmented reality, gesture recognition can be applied widely. Also, in the smart home domain, gesture recognition can be utilized to remotely control smart appliances and home robots. It can be conducted with techniques from computer vision and image processing.

2 Dataset

We load two datasets in our system which are Sign Language MNIST and Kinect Leap Dataset.

2.1 Sign Language MNIST

The original MNIST image dataset of handwritten digits which shows in the following image is a popular benchmark for image-based machine learning methods, the dataset format is patterned to match closely with the classic MNIST. Each training and test case represents a label (0-25) as a one-to-one map for each alphabetic letter A-Z (and no cases for 9=J or 25=Z because of gesture motions). The training data (27,455 cases) and test data (7172 cases) are approximately half the size of the standard MNIST but otherwise similar to a header row of the label, pixel1, pixel2 .... pixel784 which represents a single 28x28 pixel image with grayscale values between 0-255.

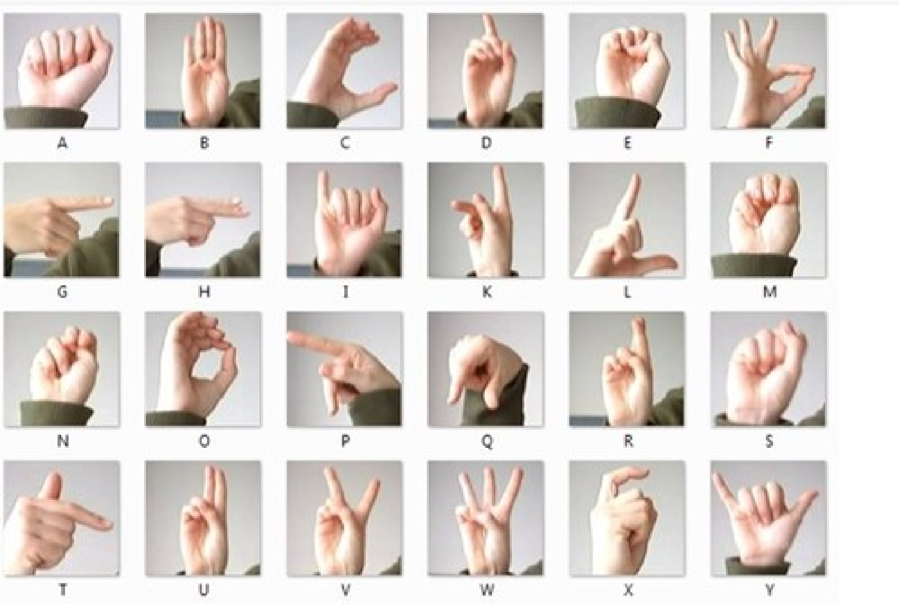


Figure 1: The dataset of MNIST

2.2 Microsoft Kinect and Leap Motion

The second one is Microsoft Kinect and leap motion. The database contains 10 different gestures performed by 14 different people and shows in the following figure. Each gesture is repeated 10 times for a total of 1400 different data samples. For each sample, Leap Motion data have been acquired together with the depth maps and color images provided by the Kinect. This dataset includes 10 gestures which represent the number 1-10.

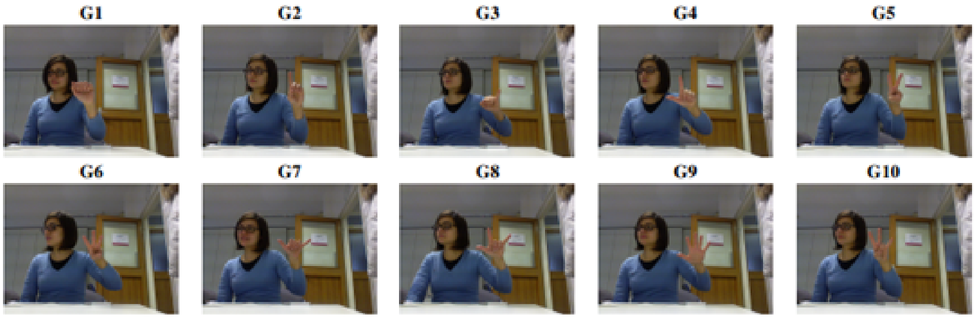
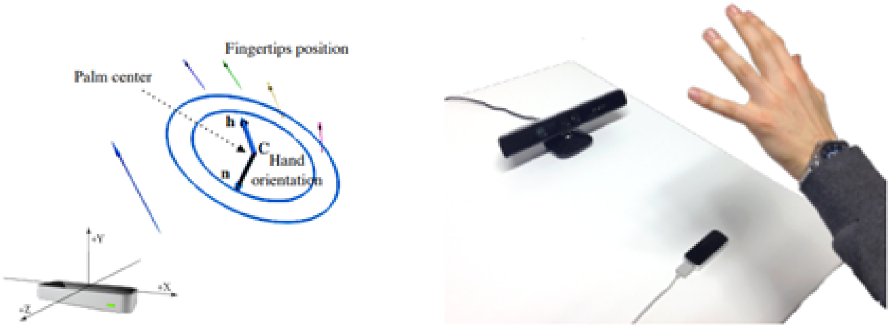


Figure 2: Picture of Kinect databset

Differently from the Kinect and other similar devices, the Leap Motion does not return a complete depth map but only a set of relevant hand points and some hand pose features. The following figure highlights the data acquired by the Leap device that will be used in the proposed gesture recognition system.



(a) Information to locate hands (b) How to locate hands information

Figure 3: Data acquired by Leap Motion

3 Model

In this section, we will have a brief introduction of the two networks we use in the project.

3.1 LeNet5

Structure of LeNet5 is: input layer→ convulational layer→pooling layer→ activation function→ convulational layer→ pooling layer→ activation function→ convulational layer→ fully connect layer→ fully connect layer→ output layer.

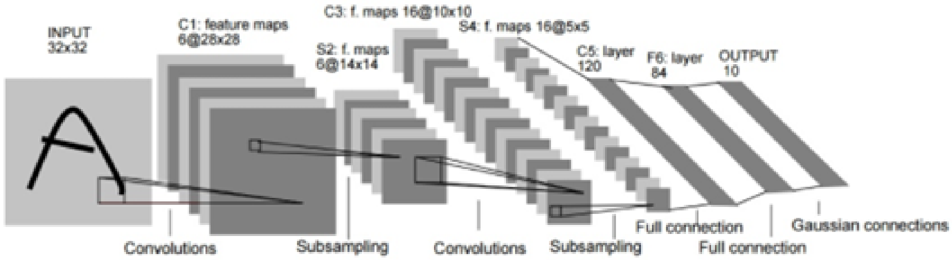


Figure 4: LeNet5

- Input layer, The input is pixel picture whose size been 28 × 28
- C1 layer, C1 layer is the first convolutional layer of LeNet. Its input is an image with a size of 32 × 32, and then after a convolution process, the output is a feature image of 6 channels with a size of 28 × 28.
- S2 layer, It is a pooling layer, which is doing down-sampling operations. The purpose is to reduce the number of parameters of the convolutional neural network while retaining the main information of the feature image. In the pooling in S2, the 2 × 2 window area of the feature image is selected for pooling, so the size of the feature image is reduced by one time, that is, the size of the feature image is reduced from 28 × 28 to 14 × 14. (The pooling process does not affect the number of feature map channels)
- C3 layer, The C3 convolutional layer uses a convolution kernel Filter size of 5 × 5 and deconvolution S2 to obtain a Feature Map size of 10 × 10. Each Feature Map on this layer is connected to all 6 Feature Maps or several Feature Maps of S2.
- S4 layer, It is also a pooling layer and composes of 16 5 × 5 Feature Maps. Each unit in the Feature Map is connected to the 2 × 2 neighborhood of the corresponding Feature Map in C3.
- C5 layer, It is a special convolutional layer. Its input is a 16-channel 5 × 5 feature image, and its output is a 120 vector. The C5 layer can be implemented in two ways, the first is to use a 5 × 5 size convolution kernel for convolution, the second is to flatten the 16-channel 5 × 5 size feature image, and then do full connection.
- F6 layer, The F6 layer is a fully connected layer with output usually been 10, to make the final prediction.
- Output layer, pick up ouput with highest probability

3.2 ResNet34

When the deeper network starts to converge, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Such degradation is not caused by overfitting or by adding more layers to a deep network leads to higher training error. The deterioration of training accuracy shows that not all systems are easy to optimize.

To overcome this problem, ResNet introduced a deep residual learning framework. Instead of hoping every few stacked layers directly fit a desired underlying mapping, they explicitly let these layers fit a residual mapping. The formulation of F(x)+x can be realized by feedforward neural networks with shortcut connections. Shortcut connections are those skipping one or more layers. The shortcut connections perform identity mapping, and their outputs are added to the outputs of the stacked layers.

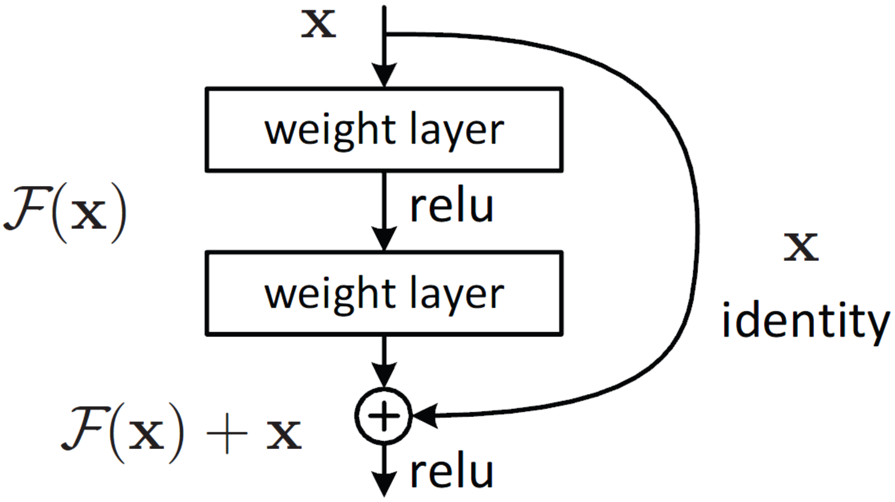

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 5: Residual Block

Plain Network: The plain baselines (Fig5, middle) are mainly inspired by the philosophy of VGG nets (Fig5, left). The convolutional layers mostly have 3×3 filters and follow two simple rules:

- For the same output feature map, the layers have the same number of filters;
- If the size of the features map is halved, the number of filters is doubled to preserve the time complexity of each layer.

Residual Network: Based on the above plain network, a shortcut connection is inserted (Fig6, right) which turn the network into its counterpart residual version. The identity shortcuts F(xW+x) can be directly used when the input and output are of the same dimensions (solid line shortcuts in Fig6). When the dimensions increase (dotted line shortcuts in Fig), it considers two options:

- The shortcut performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no additional parameter.
- The projection shortcut in F(xW+x) is used to match dimensions (done by 1×1convolutions).

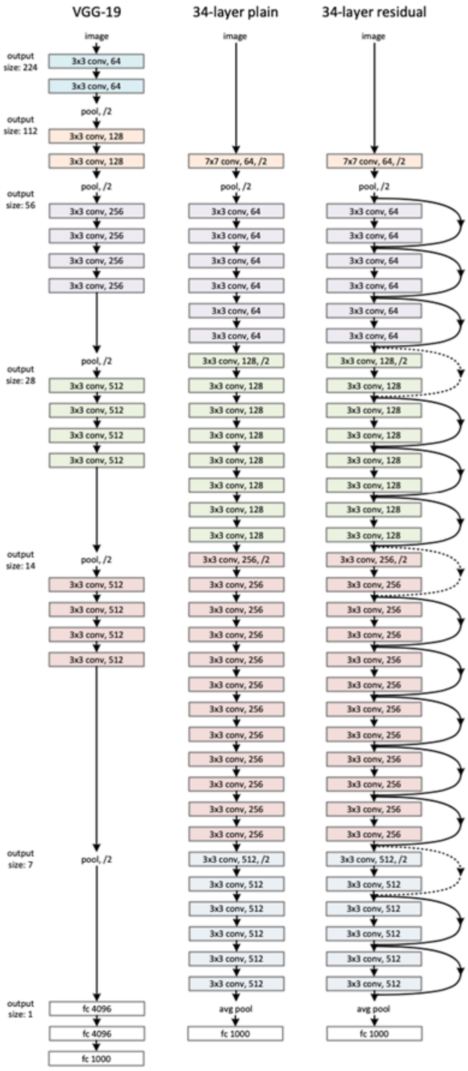


Figure 6: ResNet34

4 Implementaion

In this section, we are going to talk about how we implement the network and training process. We separate this section into three parts according to the code, which corresponding to implement the models, resizing of the dataset, and the whole training process.

4.1 Model Implementation

For our project, we implement two classic CNN models, which are LeNet5 and ResNet34. As we mentioned above, LeNet5 is a simple CNN model with 7 layers, which is relatively small compared to ResNet34. And of course, the smaller size of the model always corresponding to relatively poorer performance. With a larger scale, ResNet34 can deal with more parameters and consequently results in a better capability to a larger scale image. And we will show a more detailed test result in the next section.

LeNet5

There are only 7 layers (exclusion of the input layer) in the LeNet5, so we can easily implement layer by layer with Pytorch. In section 3 we mentioned that LeNet5 is consists of convolutional layers and fully connected layers, which all have their packaged function in Pytorch.

But for our project, something we need to pay attention to is that the input channel of the first convolutional layer. Since we have two datasets to train on and one of them is an RGB dataset, but the original LeNet5 is designed for grayscale images, we need to make the LeNet5 enable to deal with the 3-channel RGB images. Therefore, we adjust the input channel according to the input dataset. Also, the dataset does not share the same amount of classes included, which means the final output also needed to be matched with the dataset.

```
class LeNet5(nn.Module):
  def __init__(self, class_num, is_gray_scale):
      super(LeNet5, self).__init__()

      if is_gray_scale:
          input_channel = 1
      else:
          input_channel = 3

  ......

  def num_flat_features(self, x):
      size = x.size()[1:]
      num_features = 1
      for s in size:
          num_features *= s
      return num_features
```

ResNet34

Since ResNet34 got a much more complex structure compared to LeNet5 and includes a repetitive structure called a residual block, we will implement the residual block first to simplify the code. As we mentioned in section 3, a classic residual block always consists of two convolutional layers with batchnorm process between. Also, in the forward computing process, the key of the residual network is to add x to the layer output $H(x)$ to get $H(x)+x$ to compute the residual. Following the typical methods, we implement this residual block.

```
class ResBlock(nn.Module):
  def __init__(self, inchannel, outchannel, stride=1,
      shortcut=None):
      super(ResBlock, self).__init__()
      self.basic = nn.Sequential(
          nn.Conv2d(inchannel, outchannel, 3, stride, 1,
              bias=False),
          nn.BatchNorm2d(outchannel),
          nn.ReLU(inplace=True),
          nn.Conv2d(outchannel, outchannel, 3, 1, 1, bias=False),
          nn.BatchNorm2d(outchannel),
      )
      self.shortcut = shortcut

  def forward(self, x):
      out = self.basic(x)
      residual = x if self.shortcut is None else self.shortcut(x)
      out += residual
      return nn.ReLU(inplace=True)(out)
```

After we got the residual block class, we can construct the whole network by the superposition of multiple residual blocks. In our project, we choose to implement a 34-layer residual network. Before the residual learning process, we first have a convolutional layer for the normalization of the input images to make them fit the subsequent residual block layers. Then we have 4 groups of residual block layers. The first layer has 3 residual blocks with 64 input channels and 128 output channels. The second layer has 4 residual blocks with 128 input channels and 256 output channels. The third layer has 6 residual blocks with 256 input channels and 512 output channels. The fourth layer has 3 residual blocks with 512 input channels and 512 output channels. Then the network ends up with a fully connected layer for the prediction result. So the network has 1 initial convolutional layer and 32 convolutional layers in the residual block and one last fully connected layer for output prediction, which are 34 layers in total.

Similarly, with LeNet5, our project requires the network to fit the two datasets we have. Also, we want to test how the model work on the grayscale images compared to RGB images. To achieve this requirement, we have the input channel of the first convolutional layer and the output of the last fully connected layer to be adjustable.

```
class ResNet34(nn.Module):
  def __init__(self, class_num, is_gray_scale):
      super(ResNet34, self).__init__()
      if is_gray_scale:
          input_channel = 1
      else:
          input_channel = 3

      self.pre = nn.Sequential(
          nn.Conv2d(input_channel, 64, 7, 2, 3, bias=False),
          nn.BatchNorm2d(64),
          nn.ReLU(inplace=True),
          nn.MaxPool2d(3, 2, 1),)
      self.layer1 = self.__make_layer__(64, 128, 3)
      self.layer2 = self.__make_layer__(128, 256, 4, stride=2)
      self.layer3 = self.__make_layer__(256, 512, 6, stride=2)
      self.layer4 = self.__make_layer__(512, 512, 3, stride=2)
```

```
        self.fc = nn.Linear(512, class_num)
        ......

    def forward(self, x):
        ......

        return self.fc(x)
```

4.2 Datasets Adjustment

In our project, we have two datasets for training as we mentioned in section 2. The first one is the sign language MNIST dataset, the amount of this dataset is huge compared to the kinect leap dataset. Since this dataset is actually designed for the LeNet, the size of images inside the dataset is 28 × 28, which is relatively small compared to the other one. This makes us a problem that the images are too small for the ResNet34, the images are unacceptable for ResNet34 since the images are smaller than a convolutional core of the first convolutional layer. Meanwhile, LeNet5 can neither be used to learn an image with too large a scale. So the main purpose of this part is to deal with the size of the input images. Considering all the requirements mentioned above, we choose to resize the image to 224 × 224 for ResNet34 and 28 × 28 for LeNet5.

Also, another factor we want to experiment on is that he whether the grayscale of the image influenced the training result. To fulfill this purpose we add a trigger to the dataset dealing process, which allows us to choose to output the images in 1 channel or 3 channels.

```
class dataset_loader:
    ......

    def load_sign_mnist(self, img_size, isGrayScale):
        ......

    def load_kinect_leap(self, img_size, isGrayScale,
        train_size=1200):
        all_data = self.load_kinect_leap_dataset()
        shuffle(all_data)

        # set train and test set
        train_data = all_data[0:train_size - 1]
        train_set = KinectLeapDataset(train_data, img_size=img_size,
            is_gray_scale=isGrayScale)
        train_loader = DataLoader(dataset=train_set, batch_size=8,
            shuffle=False)

        test_data = all_data[train_size:len(all_data) - 1]
        test_set = KinectLeapDataset(test_data, img_size=img_size,
            is_gray_scale=isGrayScale)
        test_loader = DataLoader(dataset=test_set, batch_size=8,
            shuffle=False)
```

```
      return train_loader, test_loader

  def load_kinect_leap_dataset(self):
      ......

class KinectLeapDataset(Dataset):
  def __init__(self, data, img_size, is_gray_scale=False):
      print('loading kinect_leap_dataset')

      self.data = data
      self.labels = []
      self.is_gray_scale = is_gray_scale
      for label_tp in self.data:
          self.labels.append(label_tp[0])

      self.height = img_size
      self.width = img_size
      self.transform = transforms.Compose([transforms.ToTensor()])

  def __getitem__(self, index):
      single_image_label = self.labels[index]
      img_as_np = np.asarray(self.data[index][1])
      img_as_img = Image.fromarray(img_as_np)

      img_as_img = img_as_img.resize((self.height, self.width))

      if self.is_gray_scale:
          img_as_img = img_as_img.convert('L')

      if self.transform is not None:
          img_as_tensor = self.transform(img_as_img)

      return (img_as_tensor, single_image_label)

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

Here we take `KinectLeapDataset` as an example, resizing process of the image is embeded in the `.__getitem__()` function for each Pytorch `dataset` class. Later by `DataLoader()` class, we convert the dataset into certain size iterable batches of images in `Tensor` format.

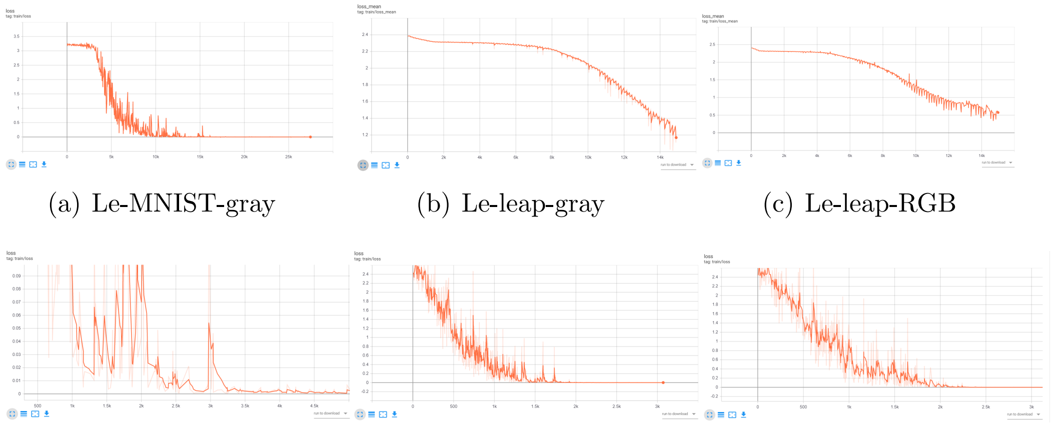
4.3 Traing Process

With all tools we needed, which are models and datasets. The training process is quite simple to build. For the criterion and optimizer we choose to use `CorssEntropyLoss` and `SGD` optimizer.

Also, we use a tool package `tensorboardX` to monitor the training process, whose result is going to be shown in the next section.

5 Test Result

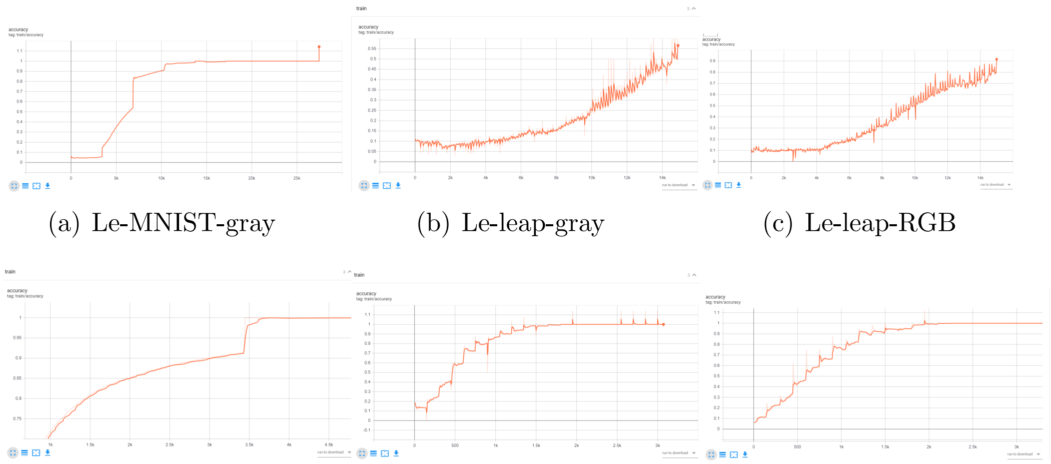
In this section, we are going to talk about the result of the training process. We totally run six different settings to find out how the performance can be influenced by the input dataset, grayscale of the images, and network itself.



(a) Le-MNIST-gray (b) Le-leap-gray (c) Le-leap-RGB

(d) Res-MNIST-gray (e) Res-leap-gray (f) Res-leap-RGB

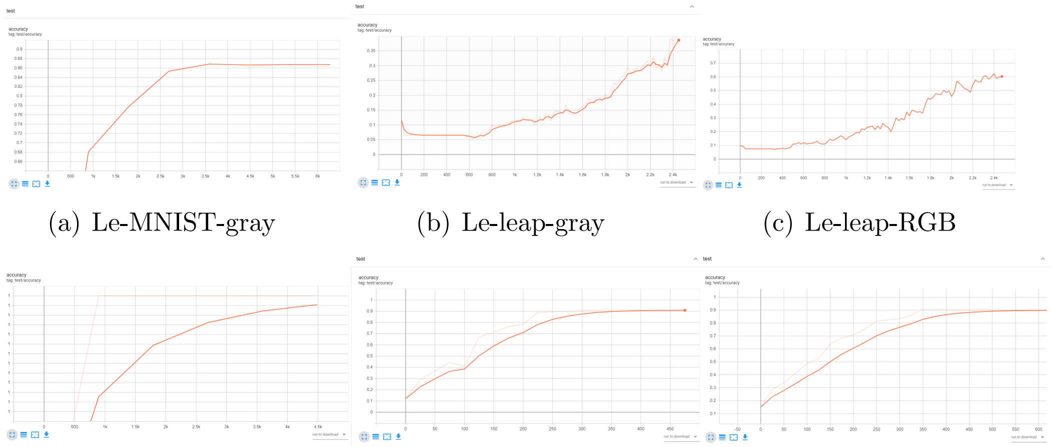
Figure 7: Variation of the loss during the training process.



(a) Le-MNIST-gray (b) Le-leap-gray (c) Le-leap-RGB

(d) Res-MNIST-gray (e) Res-leap-gray (f) Res-leap-RGB

Figure 8: Variation of the accuracy during the training process.



(a) Le-MNIST-gray (b) Le-leap-gray (c) Le-leap-RGB

(d) Res-MNIST-gray (e) Res-leap-gray (f) Res-leap-RGB

Figure 9: Variation of the accuracy during the testing process.

As we can see in the above, the first group of charts shows how the loss variate in the training process with different settings. For LeNet5, there is a clear slowdown in the speed of the loss reduction. pic(a) shows that when dealing with the MNIST dataset, LeNet5 can reduce the loss to almost zero within 15k images. But for the Kinect leap dataset with a larger scale of images, LeNet5 can not do so well. When the amount of fed images comes to 15k, the loss is only reduced to around 1.2, which is actually an unacceptable performance. The result is getting a little better when we change the images into RGB format, loss can fall around 0.5.

Respectively, the accuracy performances under each setting are highly corresponding to the loss reduction. So, imaginably, accuracy when LeNet5 dealing with the MNIST dataset is actually pretty well that can reach 100% in the training process and 87% in the testing process. But when it came to the Kinect leap dataset, even the training accuracy can only reach 55% and the testing accuracy is no more than 40%. The RGB image setting basically shares the same result.

The performance of ResNet34 shows much better performance on both datasets. On the sign language MNIST dataset, the loss can be reduced to almost zero around 3.5k. When facing the Kinect leap dataset, ResNet34 surprisingly did even better than on the MNIST dataset. Loss reaches almost 0 before 2k. For RGB images ResNet34 take 0.5k more images to reach the same level.

The result is also pretty well for accuracy. In the training process, all accuracy can reach 100%. And for the test process ResNet34 can also reach 90% accuracy which is relatively higher than the performance of LeNet5.

In conclusion, we can say that there is no doubt that ResNet34 shows better performance. And for grayscale, this attribute has a relatively low effect on the final result. Since every grayscale image is actually generated from an RGB image, we can say that each grayscale image maintains enough information from grayscale images. LeNet5 shows poor performance compared to ResNet34, especially on the large scale image. But one of its advantages is that the small scale of this network, which leads to a smaller computational resource.

Based on the analysis above, we would draw the following three conclusions: (1) ResNet34 works better on both two datasets, it can reach 100% accuracy on Kinect Leap and reduce loss to almost zero faster on MNIST (2) the grayscale of an image just has limited influence on training process that can almost be ignored (3) ResNet34 can do much better on images with a larger size

6 Application

After training our network, to make the user more convenient to use our network to get the gesture recognition result, we use python to create a product to give the answer.

This product can automatically choose the best model to do the prediction work by analyze the image itself. In the previous part, we find out that different model may have different performance when predict different image. So, after experiment, we set serval thresholds to judge the image and then choose the most suitable model and do prediction.

After choosing the model, we preprocess the image and then feed it the model to get the prediction result, compare with the ground truth to find out whether the output is correct.

7 Future Work & Conclusion

7.1 Future Work

As discussed in the previous section, our project reaches our expecting target basically. However, there are still rooms for improvement.

As for now, LeNet is more suitable for pictures whose resolution are lower than 28×28 with faster calculation and less occupation in space. ResNet works better on larger images with deeper model and longer training time. In daily applications, however, most cases require to work on larger images. In that case, we need to improve the algorithm to work faster on these images. We are aware that there are various existing methods and techniques to improve ResNet and that is exactly what we need to work on in the future.

Our project can now recognize static images, but in most cases, it is dynamic recognition that truly takes place. For this part, we still need to learn how to draw frame from a video and combine several images together to recognize gestures.

7.2 Conclusion

Our project aims at gesture recognition on numerical gestures. We choose Microsoft Kinect and Leap Motion as our training data and apply LeNet and ResNet as our network model. We successfully finish the training and testing of the two chosen networks and compare the performance between the two neural networks. We also implement a final product that can perform as an App to recognize gestures using our algorithm.

8 Improvements on review's comments

The second one is Microsoft Kinect and leap motion. The database contains 10 different gestures performed by 14 different people and shows in Figure 2. Each gesture is repeated 10 times for a total of 1400 different data samples. For each sample, Leap Motion data have been acquired together with the depth maps and color images provided by the Kinect. This dataset includes 10 gestures which represent the number 1-10.

In this section, we summarize all the peer review documents and list all the suggestions and our solutions below.

Our project lacks novelty since both LeNet and ResNet are widely applied in gesture recognition. Our project deploys and tests these two networks without coming up with new ideas. The application designed for our project could be another novelty, but the detailed description is not given in the draft.

The description for datasets is not sufficient enough. It would be better if we can add more details, such as some samples in the datasets, which can provide readers with more insights. The readers would expect a better training design. In several training experiments, the loss curves did not converge yet before the end of learning as shown in Figure 4, which would undermine the precision of trained models.

We need to cite all the references we utilized in our project. Both networks in this project are proposed long ago and we should list all the papers we referred to.

We cautiously go through all the suggestions and try our best to address all the comments. As for the novelty, we mentioned in the draft that our future works will be carried out on implementing new networks that can work better. We did not describe the application in detail since it is not a complicated project, such as Android Apps or web-based application. On the contrary, it is a simple user interface that can leverage our project. For the sections of dataset and training, we provide enough details in the final report, such as adding figures. We also include a section of references in the final report.

References

[1] Gesture recognition using recurrent neural networks https://scholar.google.com/scholar?hl=zh-CN&as_sdt=0%2C31&q=Gesture+Recognition+using+Recurrent+Neural+Networks&btnG=

[2] Hand Gesture Recognition Using Deep Convolutional Neural Networks https://link.springer.com/chapter/10.1007/978-3-319-68855-8_5

[3] Three-Dimensional Sign Language Recognition With Angular Velocity Maps and Connived Feature ResNet https://ieeexplore.ieee.org/abstract/document/8506429?casa_token=g75OyUajfIAAAAAA:Pnf9dLwBiyUU16Dj_Y_Z_ouHOS1jFz88JRqUVHb9nUwIP-20ipKqMcIapx9XR90tUpgpXhfwXY0

[4] Identity Mappings in Deep Residual Networks https://link.springer.com/chapter/10.1007/978-3-319-46493-0_38

[5] Scaling Learning Algorithms towards AI https://scholar.google.com/scholar?cluster=1777174024147096005&hl=zh-CN&as_sdt=0,31

[6] Hand posture recognition with the fuzzy glove https://www.sciencedirect.com/topics/computer-science/gesture-recognition

[7] Real-Time Hand Gesture Recognition Using Finger Segmentation https://www.hindawi.com/journals/tswj/2014/267872/

[8] G. Marin, F. Dominio, P. Zanuttigh, ”Hand gesture recognition with Leap Motion and Kinect devices”, IEEE International Conference on Image Processing (ICIP), Paris, France, 2014

[9] G. Marin, F. Dominio, P. Zanuttigh, ”Hand Gesture Recognition with Jointly Calibrated Leap Motion and Depth Sensor”, Multimedia Tools and Applications, 2015

[10] Drop-In Replacement for MNIST for Hand Gesture Recognition Tasks https://www.kaggle.com/datamunge/sign-language-mnist