Group 3
Final Report
Nhung Nguyen & Tina Nguyen
12/09/2019
Machine Learning II, Fall 2019

I. Introduction

1) Overview of the project

This project aims to classify images of characters in sign language. There have been multiple developments to incorporate machine learning with technology to assist people with disability. The goal of this project is to create an effective algorithm that can translate sign language images into letters.

2) Description of the data set.

We use the Sign Language MNIST dataset from Kaggle for this project. The dataset includes training and testing files and has around 30,000 images. The images' size are 28x28 in grayscale. The data is categorized into 24 classes based on the alphabet characters. There are no labels for J or Z in the dataset because those letters requires motion.

3) Outline

This report will include the below sections:

- The frameworks used to develop deep learning network
- The network architecture in each framework
- Results
- Summary and Conclusion
- References
- Appendix

II. Frameworks

We experiment with 2 frameworks: Keras and Tensorflow to compare the performance of the two. We use the Sequential API to create model structure using Keras and model subclassing using Tensorflow 2.0.

III. Network architecture

1) Keras network

First, we use numpy package to reshape the data to load into the model. While keeping the test data aside, we use train_test_split from sklearn to split the training data into training and validation data with the following ratio: 7:3.

```
test_imgs = test.iloc[:,1:].values.reshape((test.iloc[:,1:].shape[0],28,28))
test_imgs = np.expand_dims(test_imgs, axis=-1)

x_train, x_test2, y_train, y_test2 = train_test_split(train_imgs, train_labels, test_size=0.30)
```

We use the sequential API Keras to create the model network architecture. With images of multiple hand patterns, we decided to use conv2D since it works well in detecting particular pattern in images. We first experimented with 2 layers and continue to increase. After 4 layers, the result did not significantly improve. As a result, we use 4 layers of conv2D.

```
# initilize a model
model = Sequential()

# add the first layer
model.add(Conv2D(32, (2, 2), padding= 'same', strides =  (1, 1), act
kernel_initializer=weight_init))
```

Within each layer, we experiment with multiple parameters inside layer conv2D. For the first layer, we initialized the weight using kernel_initlizer. For ‘padding’ parameter, we use ‘same’ instead of ‘valid’ to maintain that the input and output length are the same. For ‘stride’ parameter, we use (1,1). For ‘activation’ parameter, we use ‘relu’ for all layers as ‘relu’ has been discussed as a good activation layer to adjust to nonlinearity. Furthermore, ‘relu’ allows layers to train faster (reference). We also use ‘maxpool2d’ in the layers to reduce output size and allow the model to focus on specific features within features. We also use dropout in layers to avoid overfitting. We also use ImageDataGenerator to ensure the model is more flexible with distorted data. To transform the data, we used rotation, shift and flip.

```
# construct the image generator for data augmentation
img_aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
                             height_shift_range=0.1, horizontal_flip=True, shear_range=0.2,
                             vertical_flip=True)
```

To compile the model, we used Adam optimizer as it is fast and requires low memory. The chosen loss criterion was ‘categorical_crossentropy’ since the labels was formatted using ‘to_categorical’.

```
# compile model
model.compile(optimizer=Adam(learning_rate), loss=categorical_crossentropy, metrics=[categorical_accuracy])
```

To fit the model, we used fit_generator. The batch size was 200 to reduce memory and allow the model to train faster.

```
# using fit_generator for model
model.fit_generator(img_aug.flow(x_train, y_train, batch_size=200),
    validation_data=(x_test2, y_test2), steps_per_epoch=len(x_train) // 200, epochs=n_epochs,
     verbose=1, shuffle=True, callbacks=[model_c])
```

2) Tensorflow network

Our data contains two csv files for training and testing sets. We split the training data in training and validation sets and hold out the testing set to test the model. We extracted the label from the dataset, normalized the pixel values, and split the data into training and validation sets with a ratio of 7:3. Then, we reshape the data and create a training dataset including features and labels and a validation dataset using Dataset.from_tensor_slices().

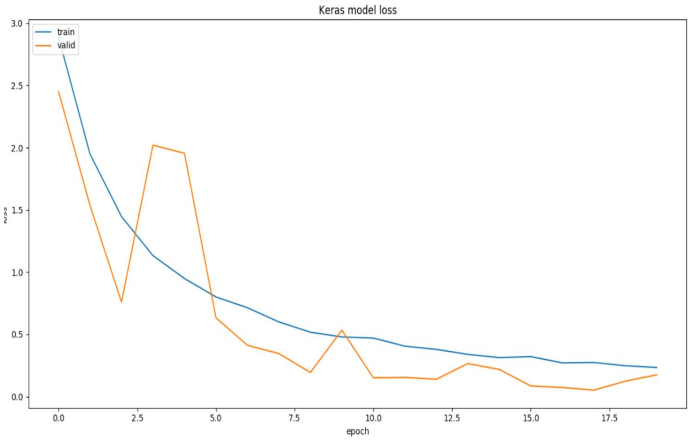
```
images = data.values/255 # normalize the data

# split data into train and validation tests
x_train, x_val, y_train, y_val = train_test_split(images, labels, test_size=0.3, strati

# reshape data and convert labels to tensor
x_train, x_val = tf.reshape(x_train, (-1, 28, 28, 1), ), tf.reshape(x_val, (-1, 28, 28,
x_train, x_val = tf.dtypes.cast(x_train, tf.float32), tf.dtypes.cast(x_val, tf.float32)
y_train, y_val = tf.convert_to_tensor(y_train), tf.convert_to_tensor(y_val)

train = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(10000).batch(32)
val = tf.data.Dataset.from_tensor_slices((x_val, y_val)).batch(32)
```

We use Adam as the optimizer and Sparse Categorical Cross Entropy as the loss function. Training on a one-layered network produced a 99% accuracy on the train and validation data but only 72% on test data. We suspect that the model is not complex enough, and we also need to control for overfitting. Our next model has two layers; each contains Conv2D, Batchnorm, and MaxPool2d with ReLu and softmax as activation functions. We also added dropout of 0.5 to control for overfitting. This network shows better performance than the previous network.

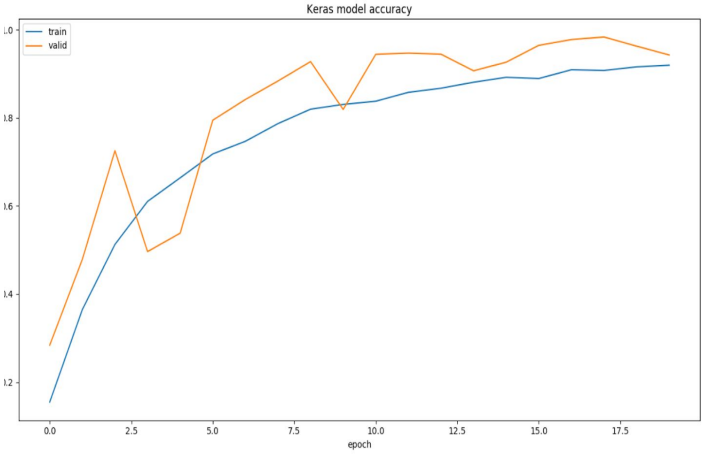
IV. Results

1) Keras result

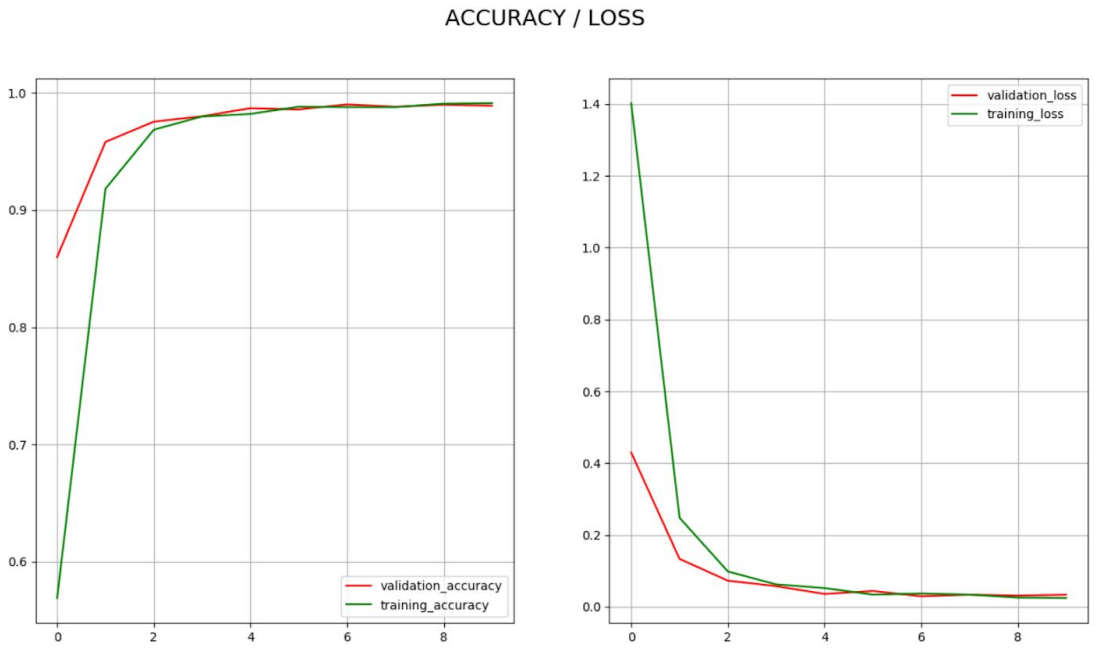
The loss for the valid and train set over the epochs gradually decrease. Although the loss was spiked up for test set in first few epochs, the overall decrease pattern shows that the learning rate was probably chosen.



The final accuracy on test set, which we set aside, is 92%. This accuracy is around 3% lower than the average accuracy for the validation and train dataset (figure below). As a result, the model is not likely to be overfitted.



2) Tensorflow result



Using a two-layered network, the accuracy and loss significantly increased/ decreased after the first epoch. After that, they fluctuate a little bit but still remain consistently around 98% accuracy and 0.01 loss on both training and validation data. Using the same trained parameters on the hold out data results in the accuracy of 92.39%.

V. Summary and conclusions

Overall, we are satisfied with the performance of both networks using Keras and Tensorflow 2.0. We noticed that the Keras model contained image augmentation, was trained over 20 epochs with 4 layers and achieve similar results to the network using Tensorflow 2.0 which was only trained over 10 epochs, 2 layers, and without image augmentation. Future project could generate input by taking pictures of multiple characters that make up a word in Sign Language, use these networks to see if they can classify the real images correctly, and adjust the network accordingly.

VI. References

Dataset Source:
https://www.kaggle.com/datamunge/sign-language-mnist
Code reference
https://stackoverflow.com/questions/55588201/pytorch-transforms-on-tensordataset
https://github.com/amir-jafari/Deep-Learning/blob/master/Pytorch_/CNN/1_ImageClassification/example_MNIST.py
https://www.kaggle.com/ranjeetjain3/deep-learning-using-sign-langugage

https://www.kaggle.com/avinashlalith/cnn-with-tensorflow-2-0
https://github.com/amir-jafari/Deep-Learning/blob/master/Tenflow_2/CNN/1_ImageClassification/example_MNIST.py
Article reference:
https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

Appendix

The complete code document is available at *Code* folder in the following link:
https://github.com/nhungnoon/Final-Project-Group3

For Keras code:
The file is called **keras_train_predict_combine.py**. To access, click here.

For Tensorflow code:
The train file is called **01-train_tensorflow_2.0.py**. To access, click here.
The test file is called **02-predict_test_tensorflow_2.0.py**. To access, click here.