

midastask2part1

April 11, 2021

Author: Pushkar Patel

1 Task 2 - Part 1

Imports

```
[1]: import shutil
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
%matplotlib inline

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Lambda
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.optimizers import Adam
```

Extracting the images

```
[4]: shutil.unpack_archive('trainPart1.zip', '../input/trainpart1zip')
```

1.1 Inspecting the image

Browsing through the dataset, we can see that there are a total of 62 classes - 10 numbers from 0 to 9, 26 lowercase alphabets and 26 uppercase alphabets, having 40 examples each.

Inspecting the image to view its dimensions and colour channels

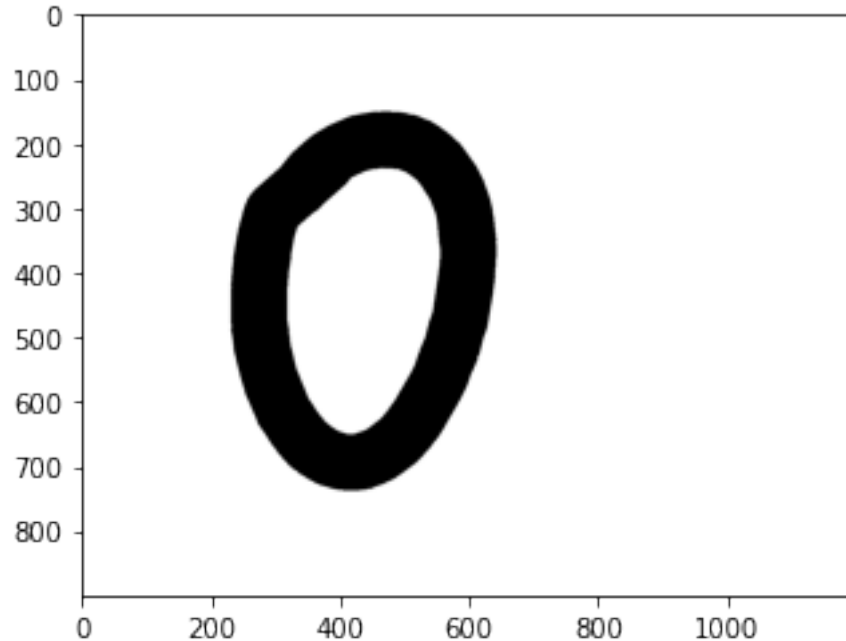
```
[ ]: image = Image.open('../input/trainpart1zip/train/Sample001/img001-001.png')
np_image = np.array(image)
print(np_image.shape)
print(image.mode)
```

```
imshow(image)
```

```
(900, 1200, 3)
```

```
RGB
```

```
[ ]: <matplotlib.image.AxesImage at 0x7efbe2070dd0>
```



The image is of dimension 900x1200 with three colour channels. Looking at the images in the directory, I found that all the images are black and white and contain only handwritten digits or alphabets. We can convert them to single grayscale colour channel to reduce computations, improve speed and make the architecture less complicated.

1.2 Preprocessing the images to convert test and validation input and labels

NOTE: Initially, I tried to build the network on the full image dimension of 900x1200, but that just overloaded the memory with too many parameters.

Since for the later tasks, I have to downsample the images to the size of MNIST i.e. 28x28, I made all other images of this dimension.

Since the images in our dataset are not square, one way to reduce the dimension of images in this set in order to have the same input shape as the MNIST dataset can be to downscale the largest dimension to 28 and then zero-pad the shorter dimension with 0 to make it 28x28.

Also, the 0 values are displayed as black and the higher values as white, which is opposite in our case. Hence, I'll invert time in our dataset to match to that of MNIST.

So, for our current image of 900x1200, 1200px would be scaled down to 28pxs and 900 would scale

down to $900 \times 28 / 1200 = 21\text{px}$. Then, I would pad it with 0s in the top and bottom to make it 28×28 . I'm padding it with zeros because: - 0 values don't add any extra information to the image
- The MNIST images have their background values as 0

Instead of padding all the images before training, I would pad them on-the-fly with a Lambda layer in the modes, before passing it to the first convolution layer.

I noticed that after resizing down, the images are still recognizable from each other so, a conv net should be able to recognize them too.

I'll use ImageDataGenerator from Keras to preprocess and split the training images into train and validation sets.

I've normalized all pixel values to be in the range of 0 to 1 for the data to have similar range.

I split the training and validation sets in 80:20 ratio.

```
[5]: train_datagen1 = ImageDataGenerator(rescale=1./255, validation_split=0.2,
    ↪ preprocessing_function=lambda x: 1-x)
```

I create the generator object which would generate the training and validation sets. It takes the input from the images folder. I'm reducing the size of the images by 20x while taking the input, and changing the colour channel to grayscale. Each set is of batch size 64. I chose this as it's a good enough batch size for this size of dataset. The class labels are categorical and are one-hot encoded for all of the 62 classes (10 numbers + 26 lowercase alphabets + 26 uppercase alphabets).

Variables that would be used globally. I set max epochs of 400.

```
[6]: BATCH_SIZE = 64
    IMAGE_SIZE = (21, 28)
    EPOCHS = 400
```

Using ImageDataGenerator, I convert the images to their one-hot encodings so that the task becomes that of classification, rather than a regression one.

```
[7]: train_generator1 = train_datagen1.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    subset='training',
    seed=42,
    shuffle=True)
```

Found 1984 images belonging to 62 classes.

```
[8]: validation_generator1 = train_datagen1.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=tf.squeeze(IMAGE_SIZE),
    batch_size=BATCH_SIZE,
```

```

class_mode='categorical',
color_mode='grayscale',
subset='validation',
seed=42,
shuffle=True)

```

Found 496 images belonging to 62 classes.

The ImageDataGenerator class has automatically detected the 62 classes.

Viewing the generated samples

```

[ ]: X_train_batch0, y_train_batch0 = train_generator1.next()
print(X_train_batch0.shape, y_train_batch0.shape)
print(y_train_batch0[0])
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_train_batch0[i]), cmap='gray')
plt.show()

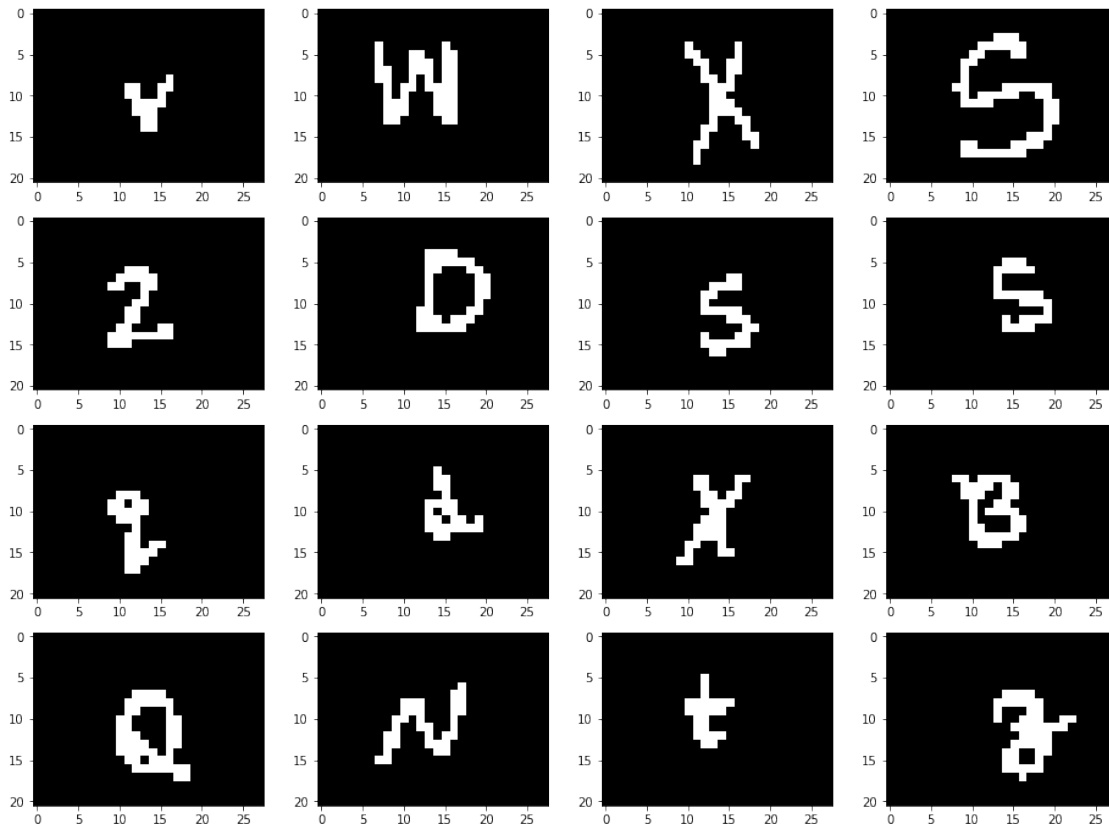
```

(64, 21, 28, 1) (64, 62)

```

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

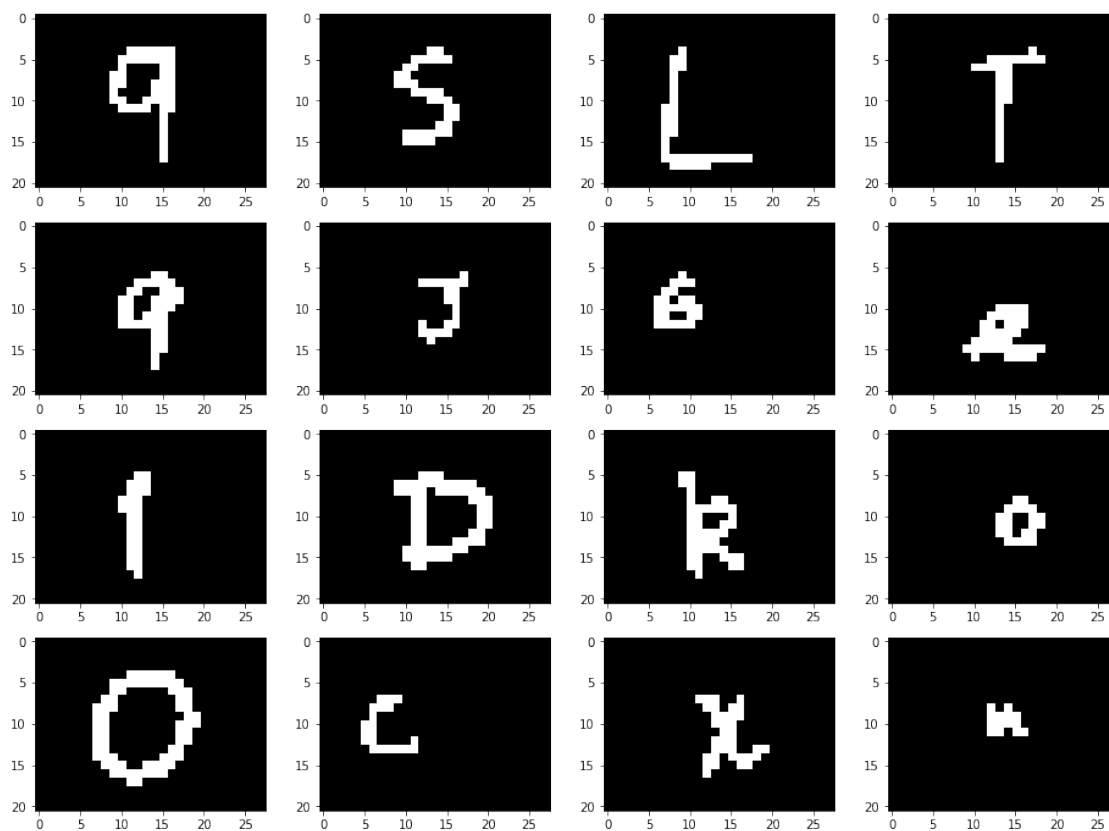
```



```
[ ]: X_validation_batch0, y_validation_batch0 = validation_generator1.next()
print(X_validation_batch0.shape, y_validation_batch0.shape)
print(y_validation_batch0[0])
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_validation_batch0[i]), cmap='gray')
plt.show()
```

(64, 21, 28, 1) (64, 62)

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```



1.3 Building the model

I train the model for 400 epochs and use early stopping with a patience level of 20 epochs in order to prevent model from overfitting and save the best weights of the mode.

Using Early Stopping to prevent overfitting, with a patience level of 20 epochs.

```
[9]: early_stopping_callback = tf.keras.callbacks.EarlyStopping(  
    monitor='val_loss',  
    mode='min',  
    patience=20,  
    restore_best_weights=True,  
    verbose=1)
```

1.3.1 Experiment 1: Building the first model inspired from LeNet

I quickly build a first model, which inspired by the original LeNet, with slight modifications, to check how it performs and will then tune the hyperparameter accordingly. I also use dropouts with a probability of 0.4 for each Fully Connected Layer.

```
[ ]: model1 = Sequential()  
  
    # Lambda Layer for adding Padding  
    model1.add(Lambda(lambda image: tf.image.resize_with_crop_or_pad(  
        image, 28, 28), input_shape=(*IMAGE_SIZE, 1)))  
  
    # 1st Convolution Layer  
    model1.add(Conv2D(6, kernel_size=(5,5), padding='same', activation='relu'))  
    model1.add(BatchNormalization())  
    model1.add(MaxPooling2D(pool_size=(2,2), strides=2))  
  
    # 2nd Convolution Layer  
    model1.add(Conv2D(16, kernel_size=(5,5), activation='relu'))  
    model1.add(BatchNormalization())  
    model1.add(MaxPooling2D(pool_size=(2,2), strides=2))  
  
    # Passing to a Fully Connected Layer  
    model1.add(Flatten())  
  
    # 1st Fully Connected Layer  
    model1.add(Dense(256, activation='relu'))  
    model1.add(BatchNormalization())  
    model1.add(Dropout(0.4))  
  
    # 2nd Fully Connected Layer  
    model1.add(Dense(128, activation='relu'))  
    model1.add(BatchNormalization())  
    model1.add(Dropout(0.4))  
  
    # Output Layer  
    model1.add(Dense(62, activation='softmax'))
```

```
[ ]: model1.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 28, 28, 1)	0
conv2d_4 (Conv2D)	(None, 28, 28, 6)	156
batch_normalization_8 (Batch Normalization)	(None, 28, 28, 6)	24
max_pooling2d_4 (MaxPooling2D)	(None, 14, 14, 6)	0
conv2d_5 (Conv2D)	(None, 10, 10, 16)	2416
batch_normalization_9 (Batch Normalization)	(None, 10, 10, 16)	64
max_pooling2d_5 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten_2 (Flatten)	(None, 400)	0
dense_6 (Dense)	(None, 256)	102656
batch_normalization_10 (Batch Normalization)	(None, 256)	1024
dropout_4 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 128)	32896
batch_normalization_11 (Batch Normalization)	(None, 128)	512
dropout_5 (Dropout)	(None, 128)	0
dense_8 (Dense)	(None, 62)	7998
Total params: 147,746		
Trainable params: 146,934		
Non-trainable params: 812		

```
[ ]: model1.compile(loss='categorical_crossentropy', optimizer=Adam(),  
    ↪metrics=['accuracy'])
```

Saving the checkpoint

```
[ ]: checkpoint_filepath1 = 'exp1/checkpoint'  
model_checkpoint_callback1 = tf.keras.callbacks.ModelCheckpoint(  
    filepath=checkpoint_filepath1,  
    save_weights_only=True,
```

```
monitor='val_loss',  
mode='min',  
save_best_only=True)
```

```
[ ]: history1 = model1.fit(  
    train_generator1,  
    epochs=EPOCHS,  
    validation_data=validation_generator1,  
    steps_per_epoch = train_generator1.samples // BATCH_SIZE,  
    validation_steps = validation_generator1.samples // BATCH_SIZE,  
    callbacks=[model_checkpoint_callback1, early_stopping_callback]  
)
```

Epoch 1/400

31/31 [=====] - 39s 1s/step - loss: 5.0043 - accuracy:
0.0213 - val_loss: 4.1208 - val_accuracy: 0.0179

Epoch 2/400

31/31 [=====] - 36s 1s/step - loss: 4.0452 - accuracy:
0.0806 - val_loss: 4.1611 - val_accuracy: 0.0357

Epoch 3/400

31/31 [=====] - 36s 1s/step - loss: 3.3932 - accuracy:
0.1817 - val_loss: 4.3968 - val_accuracy: 0.0246

Epoch 4/400

31/31 [=====] - 36s 1s/step - loss: 3.0605 - accuracy:
0.2270 - val_loss: 4.6091 - val_accuracy: 0.0246

Epoch 5/400

31/31 [=====] - 36s 1s/step - loss: 2.7359 - accuracy:
0.2881 - val_loss: 4.8350 - val_accuracy: 0.0268

Epoch 6/400

31/31 [=====] - 36s 1s/step - loss: 2.4506 - accuracy:
0.3656 - val_loss: 4.8434 - val_accuracy: 0.0469

Epoch 7/400

31/31 [=====] - 36s 1s/step - loss: 2.2959 - accuracy:
0.4072 - val_loss: 4.9224 - val_accuracy: 0.0469

Epoch 8/400

31/31 [=====] - 36s 1s/step - loss: 2.0966 - accuracy:
0.4540 - val_loss: 4.8551 - val_accuracy: 0.0513

Epoch 9/400

31/31 [=====] - 36s 1s/step - loss: 1.9068 - accuracy:
0.4856 - val_loss: 4.6808 - val_accuracy: 0.0536

Epoch 10/400

31/31 [=====] - 36s 1s/step - loss: 1.7871 - accuracy:
0.5063 - val_loss: 4.6372 - val_accuracy: 0.0714

Epoch 11/400

31/31 [=====] - 37s 1s/step - loss: 1.6535 - accuracy:
0.5538 - val_loss: 4.2520 - val_accuracy: 0.0781

Epoch 12/400

31/31 [=====] - 37s 1s/step - loss: 1.5272 - accuracy:
 0.5662 - val_loss: 3.9199 - val_accuracy: 0.1161
 Epoch 13/400
 31/31 [=====] - 37s 1s/step - loss: 1.4197 - accuracy:
 0.6169 - val_loss: 3.4918 - val_accuracy: 0.1473
 Epoch 14/400
 31/31 [=====] - 37s 1s/step - loss: 1.3951 - accuracy:
 0.6255 - val_loss: 3.1866 - val_accuracy: 0.2121
 Epoch 15/400
 31/31 [=====] - 37s 1s/step - loss: 1.2665 - accuracy:
 0.6570 - val_loss: 2.9442 - val_accuracy: 0.2679
 Epoch 16/400
 31/31 [=====] - 37s 1s/step - loss: 1.2274 - accuracy:
 0.6580 - val_loss: 2.4005 - val_accuracy: 0.3929
 Epoch 17/400
 31/31 [=====] - 37s 1s/step - loss: 1.1333 - accuracy:
 0.6832 - val_loss: 2.8734 - val_accuracy: 0.2902
 Epoch 18/400
 31/31 [=====] - 37s 1s/step - loss: 1.0525 - accuracy:
 0.7115 - val_loss: 2.5002 - val_accuracy: 0.3415
 Epoch 19/400
 31/31 [=====] - 38s 1s/step - loss: 0.9996 - accuracy:
 0.7156 - val_loss: 2.2576 - val_accuracy: 0.4018
 Epoch 20/400
 31/31 [=====] - 37s 1s/step - loss: 0.9524 - accuracy:
 0.7211 - val_loss: 2.1082 - val_accuracy: 0.4263
 Epoch 21/400
 31/31 [=====] - 38s 1s/step - loss: 0.9113 - accuracy:
 0.7535 - val_loss: 2.4135 - val_accuracy: 0.3683
 Epoch 22/400
 31/31 [=====] - 37s 1s/step - loss: 0.8432 - accuracy:
 0.7557 - val_loss: 2.1371 - val_accuracy: 0.4241
 Epoch 23/400
 31/31 [=====] - 38s 1s/step - loss: 0.7570 - accuracy:
 0.7891 - val_loss: 2.0216 - val_accuracy: 0.4464
 Epoch 24/400
 31/31 [=====] - 38s 1s/step - loss: 0.7660 - accuracy:
 0.7741 - val_loss: 1.9824 - val_accuracy: 0.4710
 Epoch 25/400
 31/31 [=====] - 37s 1s/step - loss: 0.6691 - accuracy:
 0.8188 - val_loss: 2.1950 - val_accuracy: 0.4263
 Epoch 26/400
 31/31 [=====] - 37s 1s/step - loss: 0.6867 - accuracy:
 0.8098 - val_loss: 2.2190 - val_accuracy: 0.4531
 Epoch 27/400
 31/31 [=====] - 37s 1s/step - loss: 0.6584 - accuracy:
 0.8170 - val_loss: 2.2322 - val_accuracy: 0.4487
 Epoch 28/400

31/31 [=====] - 37s 1s/step - loss: 0.6170 - accuracy:
 0.8295 - val_loss: 1.9150 - val_accuracy: 0.4866
 Epoch 29/400
 31/31 [=====] - 38s 1s/step - loss: 0.5531 - accuracy:
 0.8508 - val_loss: 2.3321 - val_accuracy: 0.4219
 Epoch 30/400
 31/31 [=====] - 37s 1s/step - loss: 0.5607 - accuracy:
 0.8538 - val_loss: 2.0796 - val_accuracy: 0.4420
 Epoch 31/400
 31/31 [=====] - 37s 1s/step - loss: 0.5285 - accuracy:
 0.8504 - val_loss: 2.0516 - val_accuracy: 0.4576
 Epoch 32/400
 31/31 [=====] - 38s 1s/step - loss: 0.4850 - accuracy:
 0.8538 - val_loss: 1.9795 - val_accuracy: 0.4821
 Epoch 33/400
 31/31 [=====] - 37s 1s/step - loss: 0.5016 - accuracy:
 0.8662 - val_loss: 2.3053 - val_accuracy: 0.4308
 Epoch 34/400
 31/31 [=====] - 37s 1s/step - loss: 0.4724 - accuracy:
 0.8681 - val_loss: 2.0423 - val_accuracy: 0.4799
 Epoch 35/400
 31/31 [=====] - 37s 1s/step - loss: 0.4426 - accuracy:
 0.8709 - val_loss: 2.0857 - val_accuracy: 0.4911
 Epoch 36/400
 31/31 [=====] - 37s 1s/step - loss: 0.4288 - accuracy:
 0.8756 - val_loss: 2.0038 - val_accuracy: 0.4777
 Epoch 37/400
 31/31 [=====] - 37s 1s/step - loss: 0.3914 - accuracy:
 0.8919 - val_loss: 2.2656 - val_accuracy: 0.4420
 Epoch 38/400
 31/31 [=====] - 36s 1s/step - loss: 0.3878 - accuracy:
 0.9013 - val_loss: 2.1795 - val_accuracy: 0.4799
 Epoch 39/400
 31/31 [=====] - 36s 1s/step - loss: 0.4015 - accuracy:
 0.8820 - val_loss: 2.6005 - val_accuracy: 0.3772
 Epoch 40/400
 31/31 [=====] - 36s 1s/step - loss: 0.3648 - accuracy:
 0.8931 - val_loss: 2.2258 - val_accuracy: 0.4397
 Epoch 41/400
 31/31 [=====] - 35s 1s/step - loss: 0.3648 - accuracy:
 0.8867 - val_loss: 2.5196 - val_accuracy: 0.4129
 Epoch 42/400
 31/31 [=====] - 36s 1s/step - loss: 0.3462 - accuracy:
 0.9028 - val_loss: 2.2333 - val_accuracy: 0.4554
 Epoch 43/400
 31/31 [=====] - 35s 1s/step - loss: 0.3170 - accuracy:
 0.9080 - val_loss: 2.1177 - val_accuracy: 0.4643
 Epoch 44/400

```

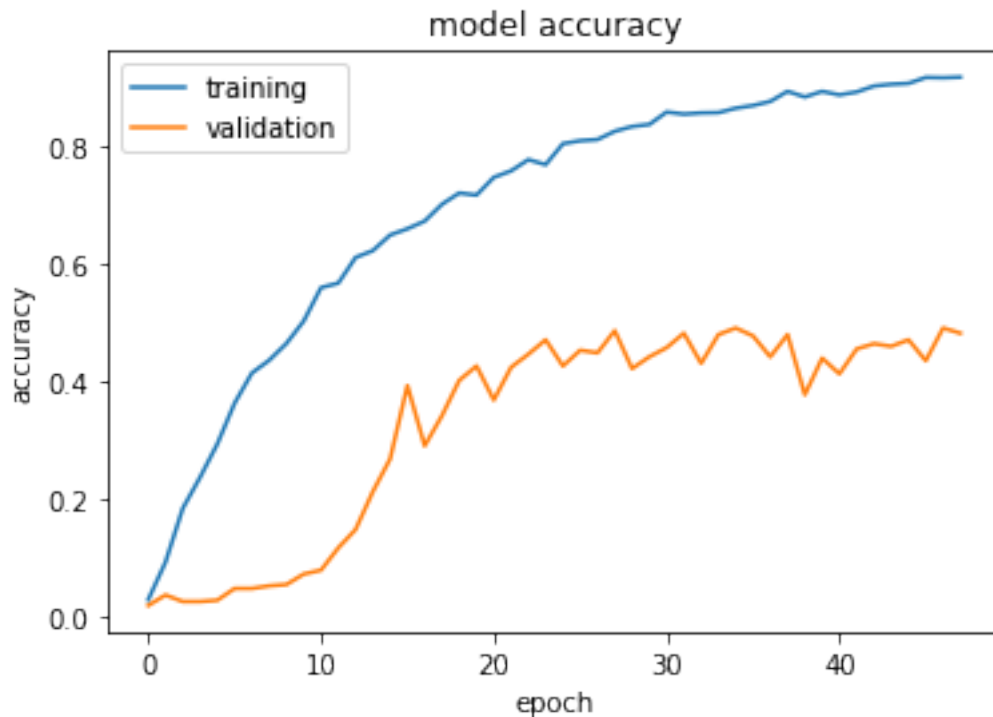
31/31 [=====] - 36s 1s/step - loss: 0.3052 - accuracy:
0.9143 - val_loss: 2.2859 - val_accuracy: 0.4598
Epoch 45/400
31/31 [=====] - 36s 1s/step - loss: 0.3035 - accuracy:
0.9181 - val_loss: 2.2188 - val_accuracy: 0.4710
Epoch 46/400
31/31 [=====] - 36s 1s/step - loss: 0.2734 - accuracy:
0.9212 - val_loss: 2.3719 - val_accuracy: 0.4353
Epoch 47/400
31/31 [=====] - 36s 1s/step - loss: 0.2909 - accuracy:
0.9129 - val_loss: 2.0599 - val_accuracy: 0.4911
Epoch 48/400
31/31 [=====] - 36s 1s/step - loss: 0.2552 - accuracy:
0.9236 - val_loss: 2.0825 - val_accuracy: 0.4821
Restoring model weights from the end of the best epoch.
Epoch 00048: early stopping

```

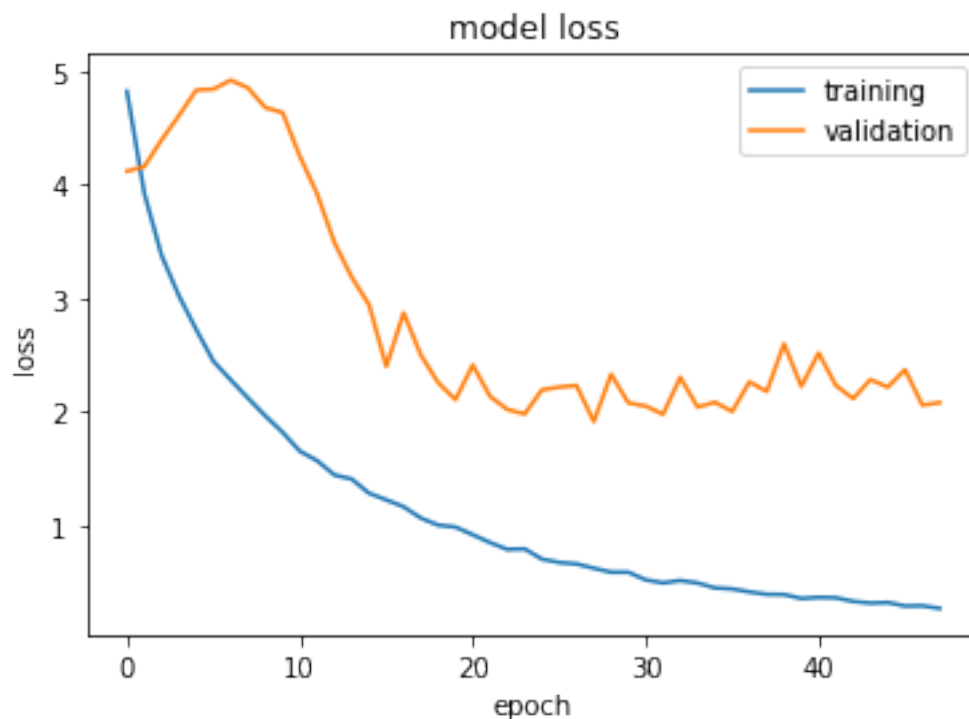
```

[ ]: plt.plot(history1.history['accuracy'])
plt.plot(history1.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()

```



```
[ ]: plt.plot(history1.history['loss'])
plt.plot(history1.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()
```



The model performs well on the training set but does not generalize well on the validation set. The model is overfitting on the training data.

One reason for the overfitting can be that there's not enough training data. For this part of the task, this cannot be improved upon. So, I'll try other ways to reduce overfitting: - Data Augmentation - Regularization - Using different activation function - Changing the model architecture

For regularization, I'm using dropout.

1.3.2 Experiment 2: Augmenting training data

I augment the data by randomly shearing it by a range of 0.1 and rotating it by a range of 0.5 degrees.

So, these augmentation methods should help generalize better on unseen images.

Sidenote: On using other data augmentation techniques I tried other parameters tool, like horizontal and vertical shifts but, they blurred the images and the training set and they didn't

really looked like the validation samples anymore (I also tested them for a small epoch and they were actually giving worse results than the first experiment). I'll show some samples to see why I did not augment much on these images, before going forward with the model building.

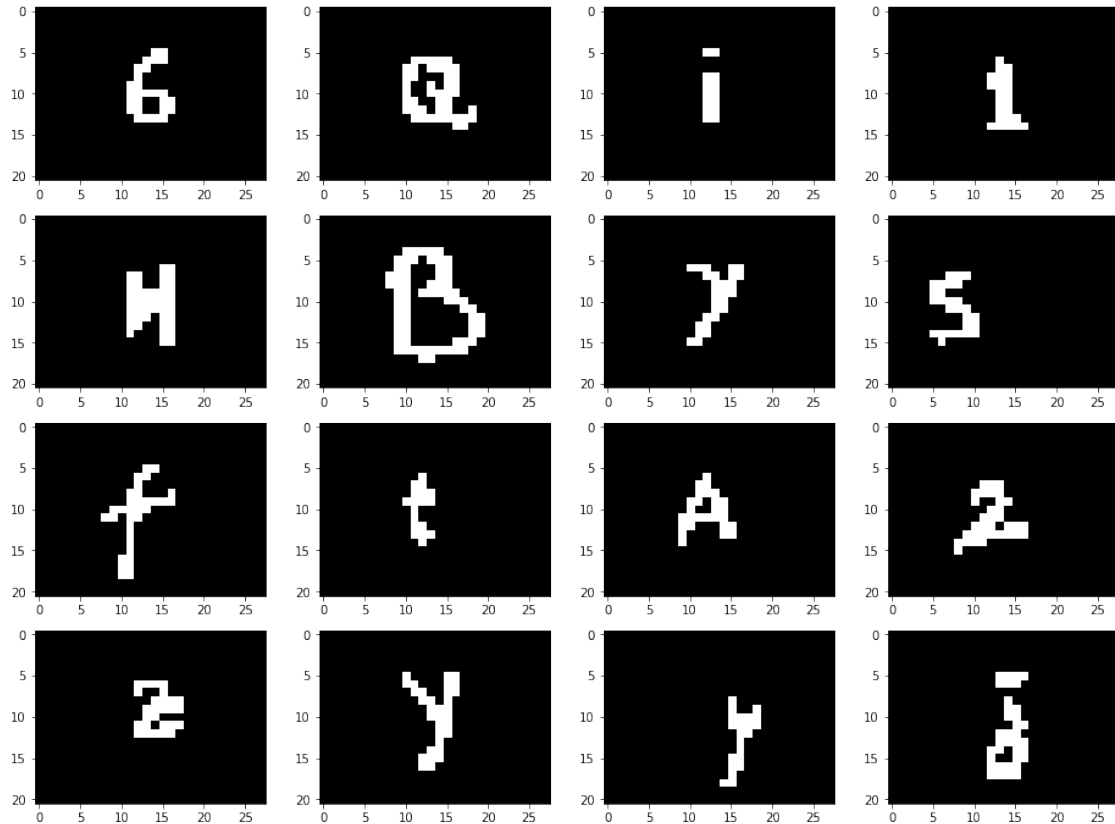
Unaugmented data

```
[ ]: no_augmentation = ImageDataGenerator(rescale=1./255,
    ↳ preprocessing_function=lambda x: 1-x)

no_augmentation_gen = no_augmentation.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    seed=42,
    shuffle=True)

X_no_aug, _ = no_augmentation_gen.next()
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_no_aug[i]), cmap='gray')
plt.show()
```

Found 2480 images belonging to 62 classes.



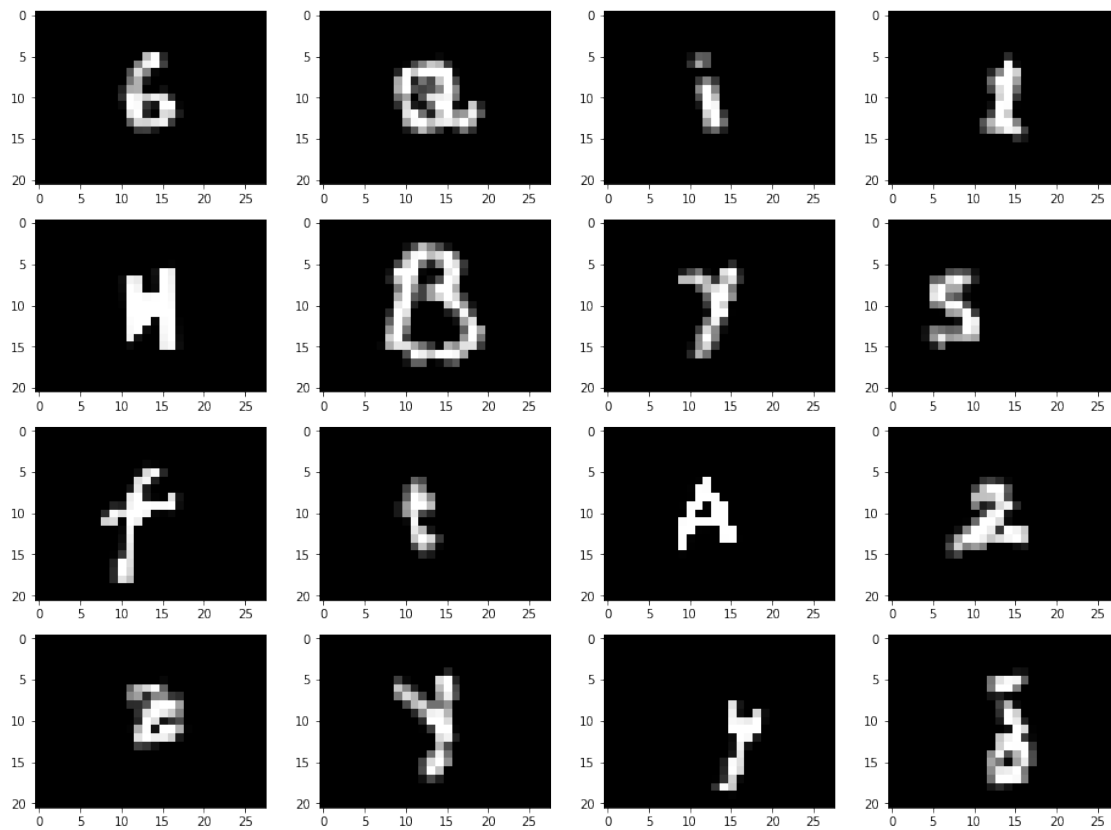
Checking rotation with max angle of 15

```
[ ]: augmentation_test_rotation = ImageDataGenerator(rescale=1./255,
↪rotation_range=15, preprocessing_function=lambda x: 1-x)

augmentation_test_rotation_gen = augmentation_test_rotation.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    seed=42,
    shuffle=True)

X_aug_rot, _ = augmentation_test_rotation_gen.next()
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_aug_rot[i]), cmap='gray')
plt.show()
```

Found 2480 images belonging to 62 classes.



Checking shear with max shear of 0.3

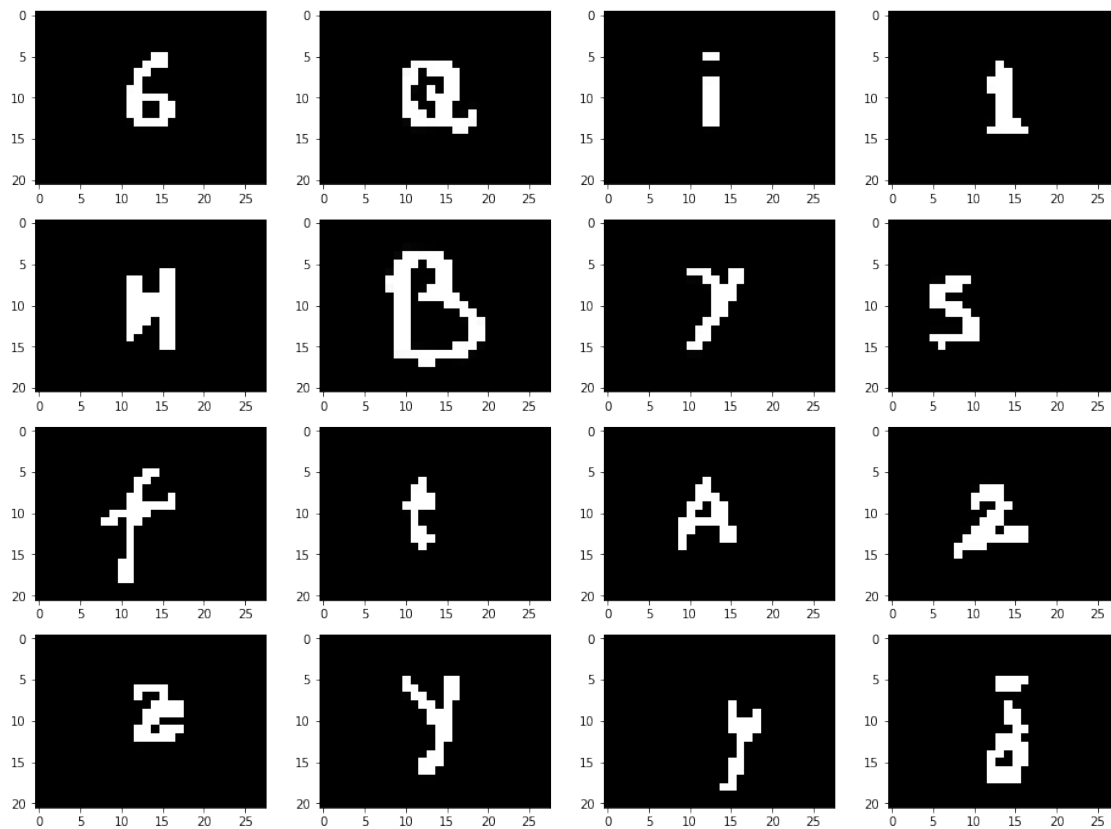
```
[ ]: augmentation_test_shear = ImageDataGenerator(rescale=1./255, shear_range=0.3,
    ↳ preprocessing_function=lambda x: 1-x)

augmentation_test_shear_gen = augmentation_test_shear.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    seed=42,
    shuffle=True)

X_aug_shear, _ = augmentation_test_shear_gen.next()
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_aug_shear[i]), cmap='gray')
```

```
plt.show()
```

Found 2480 images belonging to 62 classes.



Checking zoom with max zoom of 0.2

```
[ ]: augmentation_test_zoom = ImageDataGenerator(rescale=1./255, zoom_range=0.2,
    preprocessing_function=lambda x: 1-x)

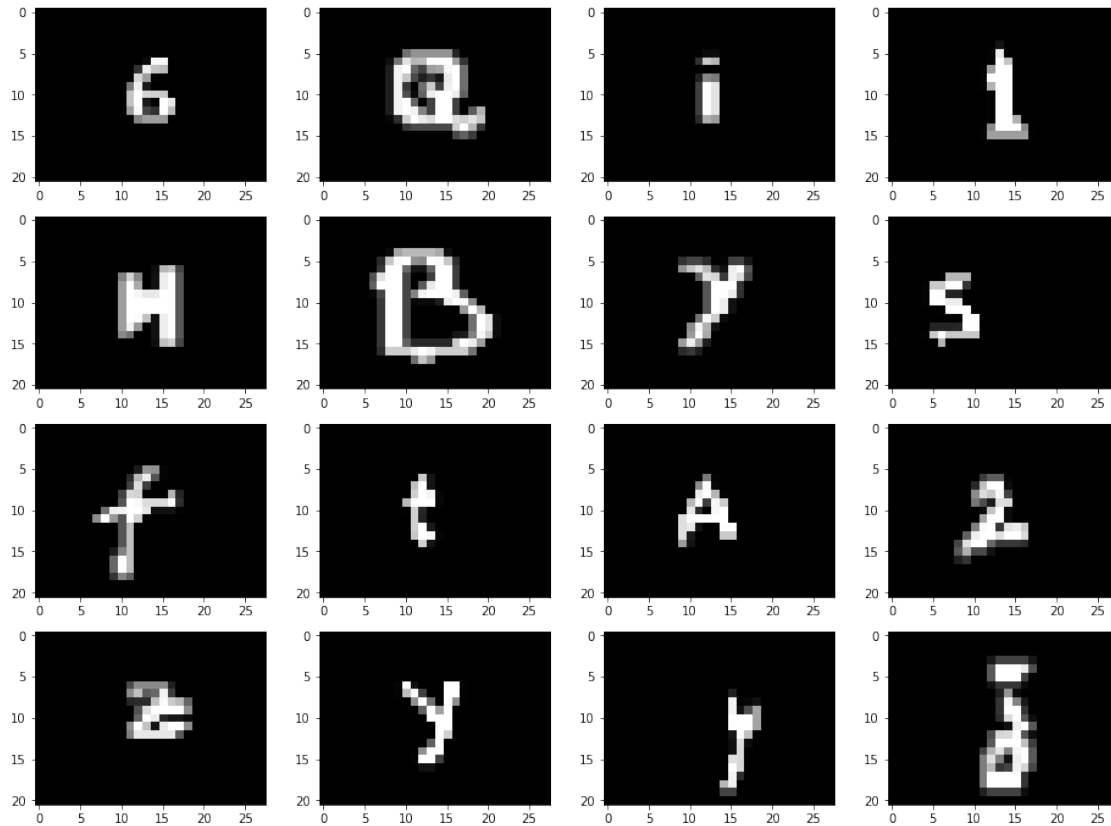
augmentation_test_zoom_gen = augmentation_test_zoom.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    seed=42,
    shuffle=True)

X_aug_zoom, _ = augmentation_test_zoom_gen.next()
plt.figure(figsize=(16,12))
for i in range(1, 17):
```



```
plt.subplot(4,4,i)
imshow(tf.squeeze(X_aug_zoom[i]), cmap='gray')
plt.show()
```

Found 2480 images belonging to 62 classes.



Checking with horizontal and vertical shift of 0.2

```
[ ]: augmentation_test_shift = ImageDataGenerator(rescale=1./255,
    ↳width_shift_range=0.2, height_shift_range=0.2, preprocessing_function=lambda
    ↳x: 1-x)

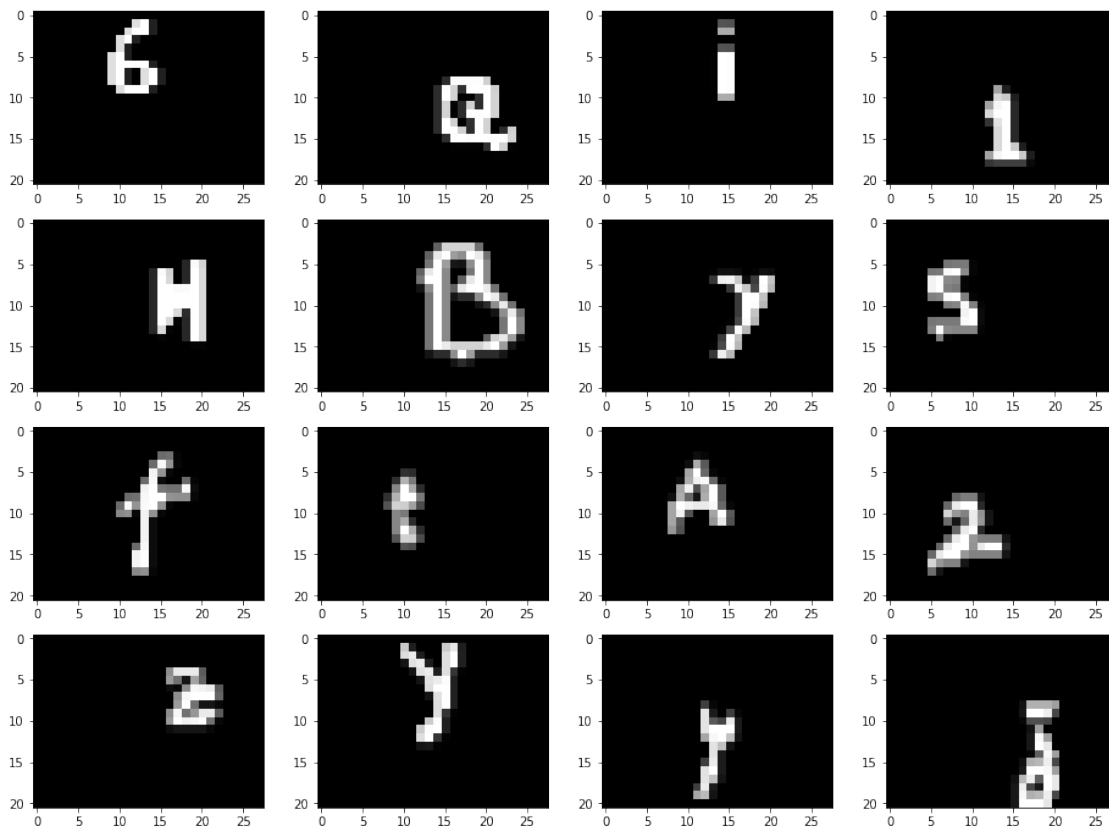
augmentation_test_shift_gen = augmentation_test_shift.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    seed=42,
    shuffle=True)
```

```

X_aug_shift, _ = augmentation_test_shift_gen.next()
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_aug_shift[i]), cmap='gray')
plt.show()

```

Found 2480 images belonging to 62 classes.



Conclusion: On what augmentation to choose These augmentations don't turn out very well. They look very different from the training data. I think the closest is the shear, but there's no noticeable effect that I can see from the naked eye. I think I'll still go on with a little of shear and a little of rotation and see how the model performs.

```

[ ]: train_datagen2 = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.2,
    shear_range=0.1,
    rotation_range=0.5,
    preprocessing_function=lambda x: 1-x)

```

```
validation_datagen2 = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.2,
    preprocessing_function=lambda x: 1-x)
```

```
[ ]: train_generator2 = train_datagen2.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    subset='training',
    seed=42,
    shuffle=True)
```

Found 1984 images belonging to 62 classes.

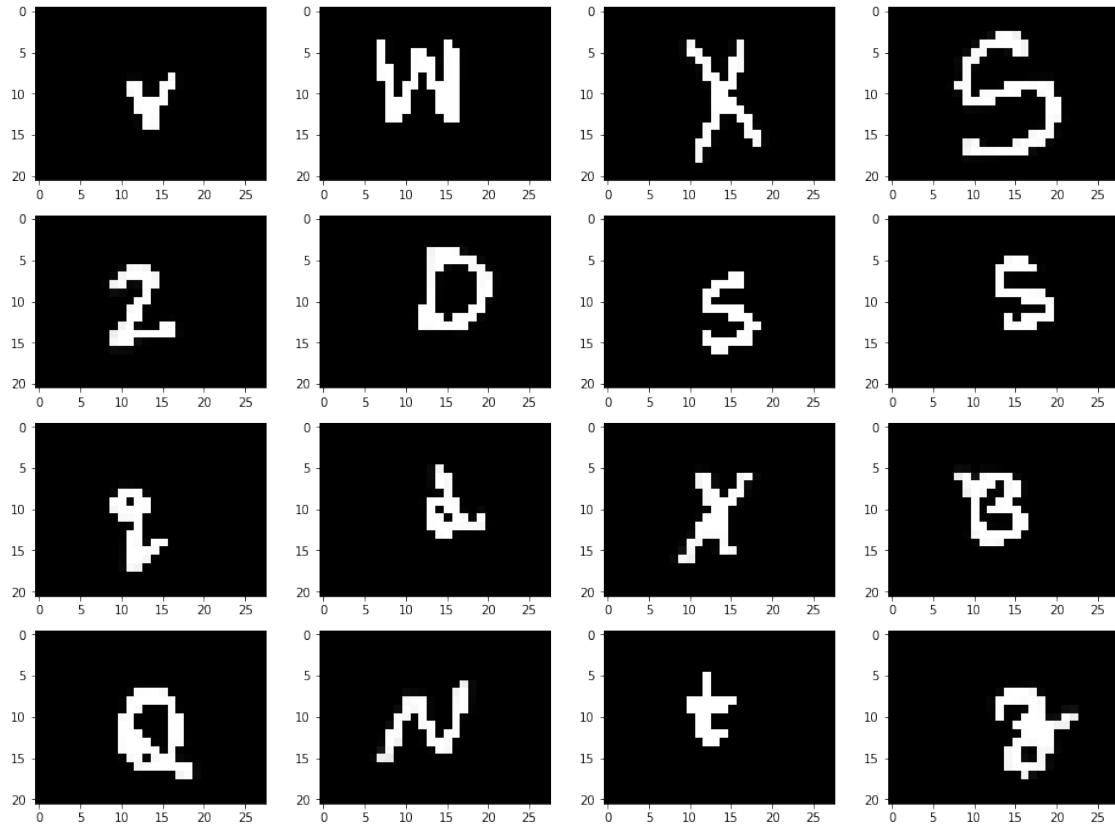
```
[ ]: validation_generator2 = validation_datagen2.flow_from_directory(
    '../input/trainpart1zip/train',
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    color_mode='grayscale',
    subset='validation',
    seed=42,
    shuffle=True)
```

Found 496 images belonging to 62 classes.

Viewing the Generated samples

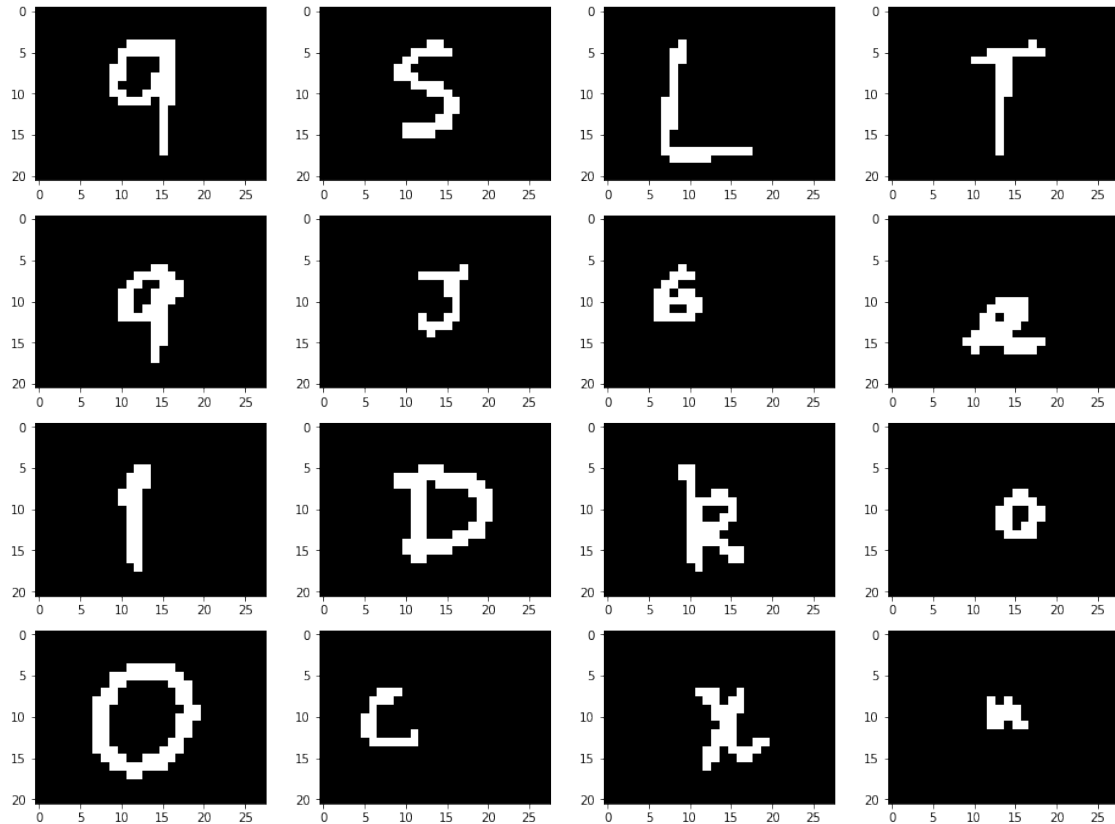
```
[ ]: X_train_batch0, y_train_batch0 = train_generator2.next()
print(X_train_batch0.shape, y_train_batch0.shape)
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_train_batch0[i]), cmap='gray')
plt.show()
```

(64, 21, 28, 1) (64, 62)



```
[ ]: X_validation_batch0, y_validation_batch0 = validation_generator2.next()
print(X_validation_batch0.shape, y_validation_batch0.shape)
plt.figure(figsize=(16,12))
for i in range(1, 17):
    plt.subplot(4,4,i)
    imshow(tf.squeeze(X_validation_batch0[i]), cmap='gray')
plt.show()
```

(64, 21, 28, 1) (64, 62)



Using the same architecture as before

```
[ ]: model2 = Sequential()

# Lambda Layer for adding Padding
model2.add(Lambda(lambda image: tf.image.resize_with_crop_or_pad(
    image, 28, 28), input_shape=(*IMAGE_SIZE, 1)))

# 1st Convolution Layer
model2.add(Conv2D(6, kernel_size=(5,5), padding='same', activation='relu'))
model2.add(BatchNormalization())
model2.add(MaxPooling2D(pool_size=(2,2), strides=2))

# 2nd Convolution Layer
model2.add(Conv2D(16, kernel_size=(5,5), activation='relu'))
model2.add(BatchNormalization())
model2.add(MaxPooling2D(pool_size=(2,2), strides=2))

# Passing to a Fully Connected Layer
model2.add(Flatten())
```

```

# 1st Fully Connected Layer
model2.add(Dense(256, activation='relu'))
model2.add(BatchNormalization())
model2.add(Dropout(0.4))

# 2nd Fully Connected Layer
model2.add(Dense(128, activation='relu'))
model2.add(BatchNormalization())
model2.add(Dropout(0.4))

# Output Layer
model2.add(Dense(62, activation='softmax'))

```

```
[ ]: model2.summary()
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
lambda_3 (Lambda)	(None, 28, 28, 1)	0
conv2d_6 (Conv2D)	(None, 28, 28, 6)	156
batch_normalization_12 (Batch Normalization)	(None, 28, 28, 6)	24
max_pooling2d_6 (MaxPooling2D)	(None, 14, 14, 6)	0
conv2d_7 (Conv2D)	(None, 10, 10, 16)	2416
batch_normalization_13 (Batch Normalization)	(None, 10, 10, 16)	64
max_pooling2d_7 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten_3 (Flatten)	(None, 400)	0
dense_9 (Dense)	(None, 256)	102656
batch_normalization_14 (Batch Normalization)	(None, 256)	1024
dropout_6 (Dropout)	(None, 256)	0
dense_10 (Dense)	(None, 128)	32896
batch_normalization_15 (Batch Normalization)	(None, 128)	512
dropout_7 (Dropout)	(None, 128)	0

```
dense_11 (Dense)                (None, 62)                7998
=====
Total params: 147,746
Trainable params: 146,934
Non-trainable params: 812
-----
```

```
[ ]: model2.compile(loss='categorical_crossentropy', optimizer=Adam(),
    ↪metrics=['accuracy'])
```

Saving the checkpoint

```
[ ]: checkpoint_filepath2 = 'exp2/checkpoint'
model_checkpoint_callback2 = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath2,
    save_weights_only=True,
    monitor='val_loss',
    mode='min',
    save_best_only=True)
```

```
[ ]: history2 = model2.fit(
    train_generator2,
    epochs=EPOCHS,
    validation_data=validation_generator2,
    steps_per_epoch = train_generator2.samples // BATCH_SIZE,
    validation_steps = validation_generator2.samples // BATCH_SIZE,
    callbacks=[model_checkpoint_callback2, early_stopping_callback]
)
```

```
Epoch 1/400
31/31 [=====] - 39s 1s/step - loss: 4.9179 - accuracy:
0.0292 - val_loss: 4.1171 - val_accuracy: 0.0134
Epoch 2/400
31/31 [=====] - 37s 1s/step - loss: 3.9544 - accuracy:
0.0845 - val_loss: 4.2481 - val_accuracy: 0.0223
Epoch 3/400
31/31 [=====] - 37s 1s/step - loss: 3.3298 - accuracy:
0.1834 - val_loss: 4.4117 - val_accuracy: 0.0179
Epoch 4/400
31/31 [=====] - 37s 1s/step - loss: 2.9691 - accuracy:
0.2378 - val_loss: 4.4836 - val_accuracy: 0.0223
Epoch 5/400
31/31 [=====] - 37s 1s/step - loss: 2.7160 - accuracy:
0.3004 - val_loss: 4.7244 - val_accuracy: 0.0246
Epoch 6/400
31/31 [=====] - 37s 1s/step - loss: 2.4925 - accuracy:
0.3474 - val_loss: 4.7267 - val_accuracy: 0.0290
Epoch 7/400
```

31/31 [=====] - 38s 1s/step - loss: 2.2964 - accuracy:
0.3897 - val_loss: 4.6166 - val_accuracy: 0.0402
Epoch 8/400
31/31 [=====] - 38s 1s/step - loss: 2.0821 - accuracy:
0.4473 - val_loss: 4.4550 - val_accuracy: 0.0513
Epoch 9/400
31/31 [=====] - 38s 1s/step - loss: 1.9095 - accuracy:
0.4767 - val_loss: 4.4226 - val_accuracy: 0.0558
Epoch 10/400
31/31 [=====] - 38s 1s/step - loss: 1.7685 - accuracy:
0.5219 - val_loss: 4.1272 - val_accuracy: 0.0804
Epoch 11/400
31/31 [=====] - 38s 1s/step - loss: 1.6479 - accuracy:
0.5503 - val_loss: 3.9994 - val_accuracy: 0.1004
Epoch 12/400
31/31 [=====] - 38s 1s/step - loss: 1.5627 - accuracy:
0.5729 - val_loss: 3.7159 - val_accuracy: 0.1406
Epoch 13/400
31/31 [=====] - 38s 1s/step - loss: 1.4425 - accuracy:
0.6084 - val_loss: 3.3262 - val_accuracy: 0.1875
Epoch 14/400
31/31 [=====] - 38s 1s/step - loss: 1.3613 - accuracy:
0.6108 - val_loss: 3.2196 - val_accuracy: 0.2098
Epoch 15/400
31/31 [=====] - 38s 1s/step - loss: 1.2495 - accuracy:
0.6619 - val_loss: 3.0782 - val_accuracy: 0.2589
Epoch 16/400
31/31 [=====] - 38s 1s/step - loss: 1.1788 - accuracy:
0.6728 - val_loss: 3.0833 - val_accuracy: 0.2388
Epoch 17/400
31/31 [=====] - 38s 1s/step - loss: 1.0816 - accuracy:
0.6973 - val_loss: 2.4277 - val_accuracy: 0.3683
Epoch 18/400
31/31 [=====] - 38s 1s/step - loss: 1.0507 - accuracy:
0.7043 - val_loss: 2.5340 - val_accuracy: 0.3728
Epoch 19/400
31/31 [=====] - 38s 1s/step - loss: 0.9735 - accuracy:
0.7281 - val_loss: 2.3432 - val_accuracy: 0.3973
Epoch 20/400
31/31 [=====] - 38s 1s/step - loss: 0.8633 - accuracy:
0.7548 - val_loss: 2.2234 - val_accuracy: 0.4152
Epoch 21/400
31/31 [=====] - 38s 1s/step - loss: 0.9129 - accuracy:
0.7495 - val_loss: 2.4204 - val_accuracy: 0.3683
Epoch 22/400
31/31 [=====] - 38s 1s/step - loss: 0.8618 - accuracy:
0.7651 - val_loss: 2.1153 - val_accuracy: 0.4219
Epoch 23/400

31/31 [=====] - 38s 1s/step - loss: 0.7984 - accuracy: 0.7648 - val_loss: 2.0327 - val_accuracy: 0.4263
Epoch 24/400
31/31 [=====] - 38s 1s/step - loss: 0.7360 - accuracy: 0.7823 - val_loss: 1.9827 - val_accuracy: 0.4397
Epoch 25/400
31/31 [=====] - 38s 1s/step - loss: 0.6709 - accuracy: 0.8152 - val_loss: 2.2615 - val_accuracy: 0.4330
Epoch 26/400
31/31 [=====] - 38s 1s/step - loss: 0.6548 - accuracy: 0.8148 - val_loss: 1.9355 - val_accuracy: 0.4576
Epoch 27/400
31/31 [=====] - 38s 1s/step - loss: 0.5771 - accuracy: 0.8309 - val_loss: 2.0500 - val_accuracy: 0.4487
Epoch 28/400
31/31 [=====] - 38s 1s/step - loss: 0.5515 - accuracy: 0.8555 - val_loss: 2.2629 - val_accuracy: 0.4062
Epoch 29/400
31/31 [=====] - 38s 1s/step - loss: 0.6068 - accuracy: 0.8217 - val_loss: 2.1148 - val_accuracy: 0.4509
Epoch 30/400
31/31 [=====] - 38s 1s/step - loss: 0.5225 - accuracy: 0.8474 - val_loss: 2.0364 - val_accuracy: 0.4688
Epoch 31/400
31/31 [=====] - 38s 1s/step - loss: 0.4543 - accuracy: 0.8794 - val_loss: 2.0007 - val_accuracy: 0.4397
Epoch 32/400
31/31 [=====] - 38s 1s/step - loss: 0.4583 - accuracy: 0.8642 - val_loss: 2.1119 - val_accuracy: 0.4330
Epoch 33/400
31/31 [=====] - 38s 1s/step - loss: 0.4582 - accuracy: 0.8802 - val_loss: 2.1717 - val_accuracy: 0.3973
Epoch 34/400
31/31 [=====] - 38s 1s/step - loss: 0.4106 - accuracy: 0.8962 - val_loss: 2.0637 - val_accuracy: 0.4688
Epoch 35/400
31/31 [=====] - 38s 1s/step - loss: 0.4293 - accuracy: 0.8870 - val_loss: 2.2267 - val_accuracy: 0.4308
Epoch 36/400
31/31 [=====] - 38s 1s/step - loss: 0.3922 - accuracy: 0.8794 - val_loss: 2.1612 - val_accuracy: 0.4442
Epoch 37/400
31/31 [=====] - 38s 1s/step - loss: 0.3878 - accuracy: 0.8886 - val_loss: 2.2871 - val_accuracy: 0.4487
Epoch 38/400
31/31 [=====] - 38s 1s/step - loss: 0.3509 - accuracy: 0.8957 - val_loss: 2.1536 - val_accuracy: 0.4353
Epoch 39/400

```

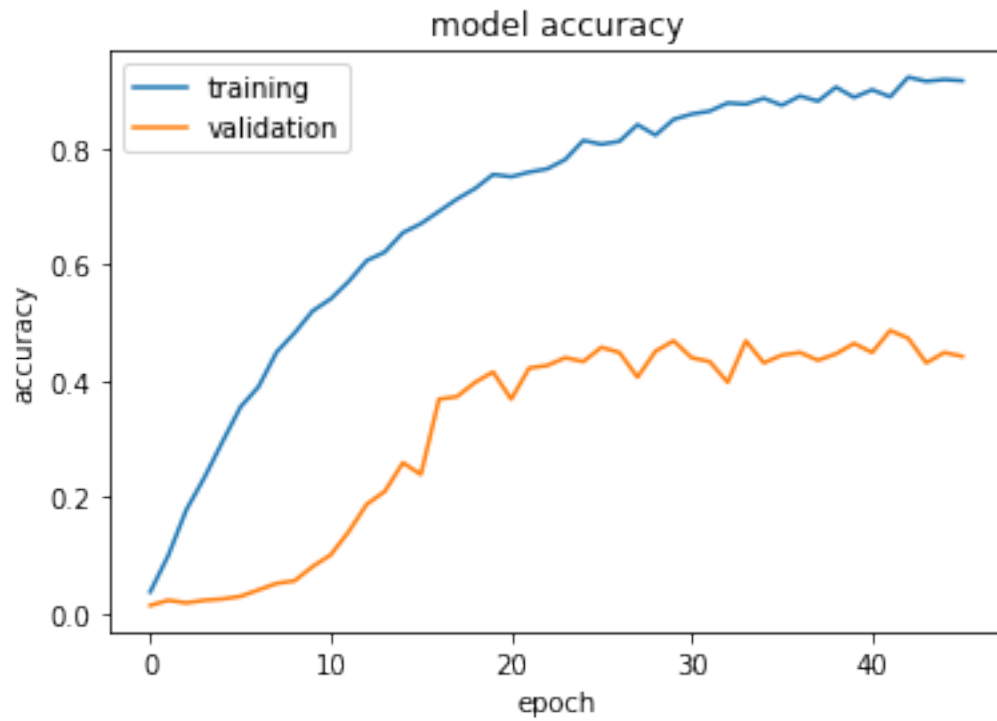
31/31 [=====] - 38s 1s/step - loss: 0.3231 - accuracy:
0.9113 - val_loss: 2.0523 - val_accuracy: 0.4464
Epoch 40/400
31/31 [=====] - 37s 1s/step - loss: 0.3275 - accuracy:
0.8947 - val_loss: 2.1564 - val_accuracy: 0.4643
Epoch 41/400
31/31 [=====] - 37s 1s/step - loss: 0.3112 - accuracy:
0.9133 - val_loss: 2.1274 - val_accuracy: 0.4487
Epoch 42/400
31/31 [=====] - 37s 1s/step - loss: 0.3607 - accuracy:
0.8859 - val_loss: 2.2628 - val_accuracy: 0.4866
Epoch 43/400
31/31 [=====] - 37s 1s/step - loss: 0.2897 - accuracy:
0.9185 - val_loss: 2.0414 - val_accuracy: 0.4732
Epoch 44/400
31/31 [=====] - 37s 1s/step - loss: 0.2757 - accuracy:
0.9142 - val_loss: 2.2787 - val_accuracy: 0.4308
Epoch 45/400
31/31 [=====] - 38s 1s/step - loss: 0.2898 - accuracy:
0.9170 - val_loss: 2.5477 - val_accuracy: 0.4487
Epoch 46/400
31/31 [=====] - 38s 1s/step - loss: 0.2611 - accuracy:
0.9306 - val_loss: 2.2970 - val_accuracy: 0.4420
Restoring model weights from the end of the best epoch.
Epoch 00046: early stopping

```

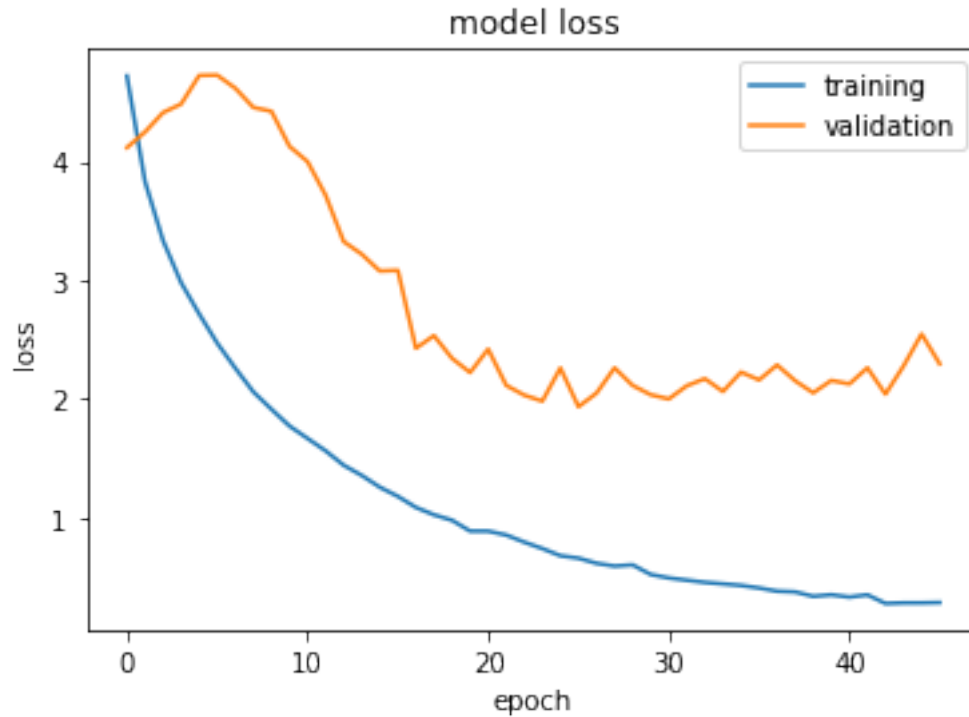
```

[ ]: plt.plot(history2.history['accuracy'])
plt.plot(history2.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()

```



```
[ ]: plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()
```



There's no visible improvements, than the first experiment. I think this is so because in this case, even slightly augmenting data leads to larger variations and since we don't have a lot of training samples, it still overfits to this data and does not generalize well on the validation set. So, I'll not use regularization in further experiments.

1.3.3 Experiment 3: Using different activation function: Mish

From refs. [4] and [5], I will use the new Mish activation over ReLU. I'll use the training samples from first experiment as they gave better results than the second experiment and will use the same activation.

```
[10]: # Mish Activation Function
def mish(x):
    return tf.keras.layers.Lambda(lambda x: x*tf.tanh(tf.math.log(1+tf.
    ↪exp(x))))(x)
```

```
[ ]: model3 = Sequential()

# Lambda Layer for adding Padding
model3.add(Lambda(lambda image: tf.image.resize_with_crop_or_pad(
    image, 28, 28), input_shape=(*IMAGE_SIZE, 1)))

# 1st Convolution Layer
model3.add(Conv2D(6, kernel_size=(5,5), padding='same', activation='relu'))
```

```

model3.add(BatchNormalization())
model3.add(MaxPooling2D(pool_size=(2,2), strides=2))

# 2nd Convolution Layer
model3.add(Conv2D(16, kernel_size=(5,5), activation=mish))
model3.add(BatchNormalization())
model3.add(MaxPooling2D(pool_size=(2,2), strides=2))

# Passing to a Fully Connected Layer
model3.add(Flatten())

# 1st Fully Connected Layer
model3.add(Dense(256, activation=mish))
model3.add(BatchNormalization())
model3.add(Dropout(0.4))

# 2nd Fully Connected Layer
model3.add(Dense(128, activation=mish))
model3.add(BatchNormalization())
model3.add(Dropout(0.4))

# Output Layer
model3.add(Dense(62, activation='softmax'))

```

```
[ ]: model3.summary()
```

Model: "sequential_9"

Layer (type)	Output Shape	Param #
lambda_7 (Lambda)	(None, 28, 28, 1)	0
conv2d_9 (Conv2D)	(None, 28, 28, 6)	156
batch_normalization_16 (Batch Normalization)	(None, 28, 28, 6)	24
max_pooling2d_8 (MaxPooling2D)	(None, 14, 14, 6)	0
conv2d_10 (Conv2D)	(None, 10, 10, 16)	2416
batch_normalization_17 (Batch Normalization)	(None, 10, 10, 16)	64
max_pooling2d_9 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten_4 (Flatten)	(None, 400)	0
dense_12 (Dense)	(None, 256)	102656

```

-----
batch_normalization_18 (Batch Normalization) (None, 256) 1024
-----
dropout_8 (Dropout) (None, 256) 0
-----
dense_13 (Dense) (None, 128) 32896
-----
batch_normalization_19 (Batch Normalization) (None, 128) 512
-----
dropout_9 (Dropout) (None, 128) 0
-----
dense_14 (Dense) (None, 62) 7998
=====
Total params: 147,746
Trainable params: 146,934
Non-trainable params: 812
-----

```

```
[ ]: model3.compile(loss='categorical_crossentropy', optimizer=Adam(),
    ↪metrics=['accuracy'])
```

Saving the checkpoint

```
[ ]: checkpoint_filepath3 = 'exp3/checkpoint'
model_checkpoint_callback3 = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath3,
    save_weights_only=True,
    monitor='val_loss',
    mode='min',
    save_best_only=True)
```

```
[ ]: history3 = model3.fit(
    train_generator1,
    epochs=EPOCHS,
    validation_data=validation_generator1,
    steps_per_epoch = train_generator1.samples // BATCH_SIZE,
    validation_steps = validation_generator1.samples // BATCH_SIZE,
    callbacks=[model_checkpoint_callback3, early_stopping_callback]
)
```

Epoch 1/400

```
31/31 [=====] - 40s 1s/step - loss: 4.9238 - accuracy:
0.0266 - val_loss: 4.1515 - val_accuracy: 0.0134
```

Epoch 2/400

```
31/31 [=====] - 37s 1s/step - loss: 3.8077 - accuracy:
0.0993 - val_loss: 4.2567 - val_accuracy: 0.0156
```

Epoch 3/400

```
31/31 [=====] - 37s 1s/step - loss: 3.2195 - accuracy:
```

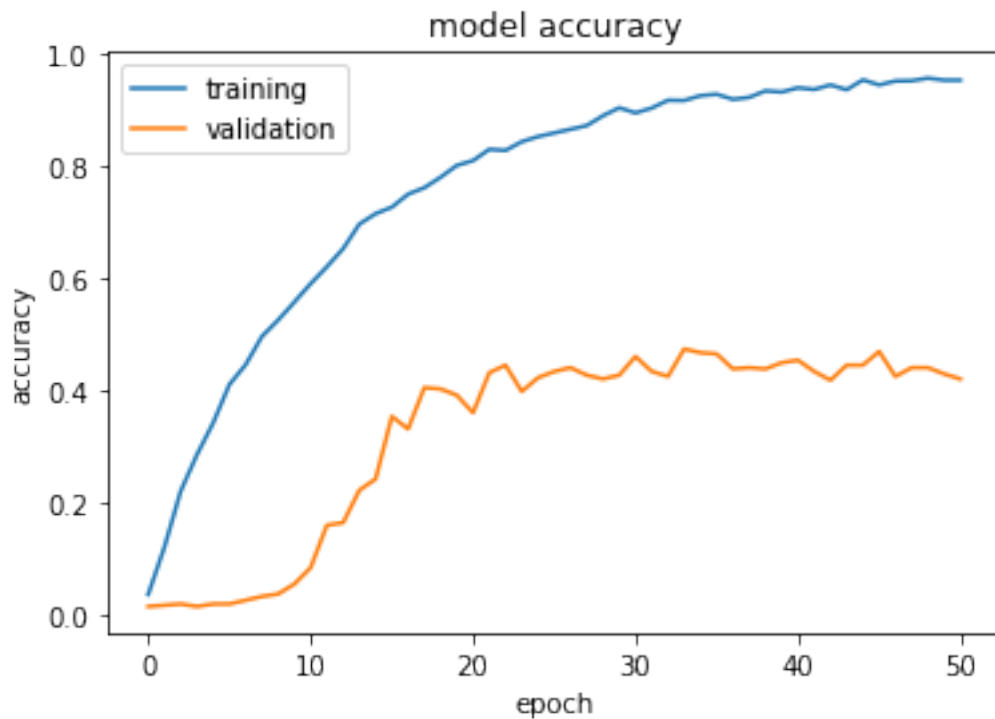
0.1945 - val_loss: 4.3816 - val_accuracy: 0.0179
 Epoch 4/400
 31/31 [=====] - 37s 1s/step - loss: 2.8448 - accuracy:
 0.2681 - val_loss: 4.4611 - val_accuracy: 0.0134
 Epoch 5/400
 31/31 [=====] - 37s 1s/step - loss: 2.5315 - accuracy:
 0.3386 - val_loss: 4.4283 - val_accuracy: 0.0179
 Epoch 6/400
 31/31 [=====] - 37s 1s/step - loss: 2.2639 - accuracy:
 0.4173 - val_loss: 4.4882 - val_accuracy: 0.0179
 Epoch 7/400
 31/31 [=====] - 37s 1s/step - loss: 2.1102 - accuracy:
 0.4271 - val_loss: 4.4642 - val_accuracy: 0.0246
 Epoch 8/400
 31/31 [=====] - 37s 1s/step - loss: 1.9080 - accuracy:
 0.4965 - val_loss: 4.4612 - val_accuracy: 0.0312
 Epoch 9/400
 31/31 [=====] - 37s 1s/step - loss: 1.7929 - accuracy:
 0.5288 - val_loss: 4.3792 - val_accuracy: 0.0357
 Epoch 10/400
 31/31 [=====] - 37s 1s/step - loss: 1.6198 - accuracy:
 0.5701 - val_loss: 4.1353 - val_accuracy: 0.0536
 Epoch 11/400
 31/31 [=====] - 37s 1s/step - loss: 1.4427 - accuracy:
 0.6180 - val_loss: 3.9084 - val_accuracy: 0.0826
 Epoch 12/400
 31/31 [=====] - 37s 1s/step - loss: 1.3872 - accuracy:
 0.6358 - val_loss: 3.5125 - val_accuracy: 0.1585
 Epoch 13/400
 31/31 [=====] - 37s 1s/step - loss: 1.2686 - accuracy:
 0.6511 - val_loss: 3.4489 - val_accuracy: 0.1629
 Epoch 14/400
 31/31 [=====] - 37s 1s/step - loss: 1.1243 - accuracy:
 0.7029 - val_loss: 3.1760 - val_accuracy: 0.2210
 Epoch 15/400
 31/31 [=====] - 37s 1s/step - loss: 1.0083 - accuracy:
 0.7348 - val_loss: 3.0905 - val_accuracy: 0.2411
 Epoch 16/400
 31/31 [=====] - 38s 1s/step - loss: 1.0226 - accuracy:
 0.7115 - val_loss: 2.5326 - val_accuracy: 0.3527
 Epoch 17/400
 31/31 [=====] - 37s 1s/step - loss: 0.9417 - accuracy:
 0.7337 - val_loss: 2.6130 - val_accuracy: 0.3304
 Epoch 18/400
 31/31 [=====] - 37s 1s/step - loss: 0.8189 - accuracy:
 0.7852 - val_loss: 2.3660 - val_accuracy: 0.4040
 Epoch 19/400
 31/31 [=====] - 37s 1s/step - loss: 0.7793 - accuracy:

0.7889 - val_loss: 2.2848 - val_accuracy: 0.4018
Epoch 20/400
31/31 [=====] - 37s 1s/step - loss: 0.7237 - accuracy:
0.8143 - val_loss: 2.3281 - val_accuracy: 0.3906
Epoch 21/400
31/31 [=====] - 37s 1s/step - loss: 0.7046 - accuracy:
0.8139 - val_loss: 2.6179 - val_accuracy: 0.3594
Epoch 22/400
31/31 [=====] - 37s 1s/step - loss: 0.6245 - accuracy:
0.8436 - val_loss: 2.1490 - val_accuracy: 0.4308
Epoch 23/400
31/31 [=====] - 37s 1s/step - loss: 0.5827 - accuracy:
0.8399 - val_loss: 2.1528 - val_accuracy: 0.4442
Epoch 24/400
31/31 [=====] - 37s 1s/step - loss: 0.5779 - accuracy:
0.8506 - val_loss: 2.3637 - val_accuracy: 0.3973
Epoch 25/400
31/31 [=====] - 37s 1s/step - loss: 0.5390 - accuracy:
0.8463 - val_loss: 2.3321 - val_accuracy: 0.4219
Epoch 26/400
31/31 [=====] - 37s 1s/step - loss: 0.5005 - accuracy:
0.8700 - val_loss: 2.1540 - val_accuracy: 0.4330
Epoch 27/400
31/31 [=====] - 37s 1s/step - loss: 0.5079 - accuracy:
0.8603 - val_loss: 2.1453 - val_accuracy: 0.4397
Epoch 28/400
31/31 [=====] - 36s 1s/step - loss: 0.4628 - accuracy:
0.8690 - val_loss: 2.2682 - val_accuracy: 0.4263
Epoch 29/400
31/31 [=====] - 36s 1s/step - loss: 0.4201 - accuracy:
0.8810 - val_loss: 2.3403 - val_accuracy: 0.4196
Epoch 30/400
31/31 [=====] - 36s 1s/step - loss: 0.3814 - accuracy:
0.9025 - val_loss: 2.2534 - val_accuracy: 0.4263
Epoch 31/400
31/31 [=====] - 36s 1s/step - loss: 0.3789 - accuracy:
0.9030 - val_loss: 2.1394 - val_accuracy: 0.4598
Epoch 32/400
31/31 [=====] - 37s 1s/step - loss: 0.3672 - accuracy:
0.9033 - val_loss: 2.4519 - val_accuracy: 0.4330
Epoch 33/400
31/31 [=====] - 37s 1s/step - loss: 0.3402 - accuracy:
0.9130 - val_loss: 2.4075 - val_accuracy: 0.4241
Epoch 34/400
31/31 [=====] - 37s 1s/step - loss: 0.3090 - accuracy:
0.9154 - val_loss: 2.2416 - val_accuracy: 0.4732
Epoch 35/400
31/31 [=====] - 37s 1s/step - loss: 0.2895 - accuracy:

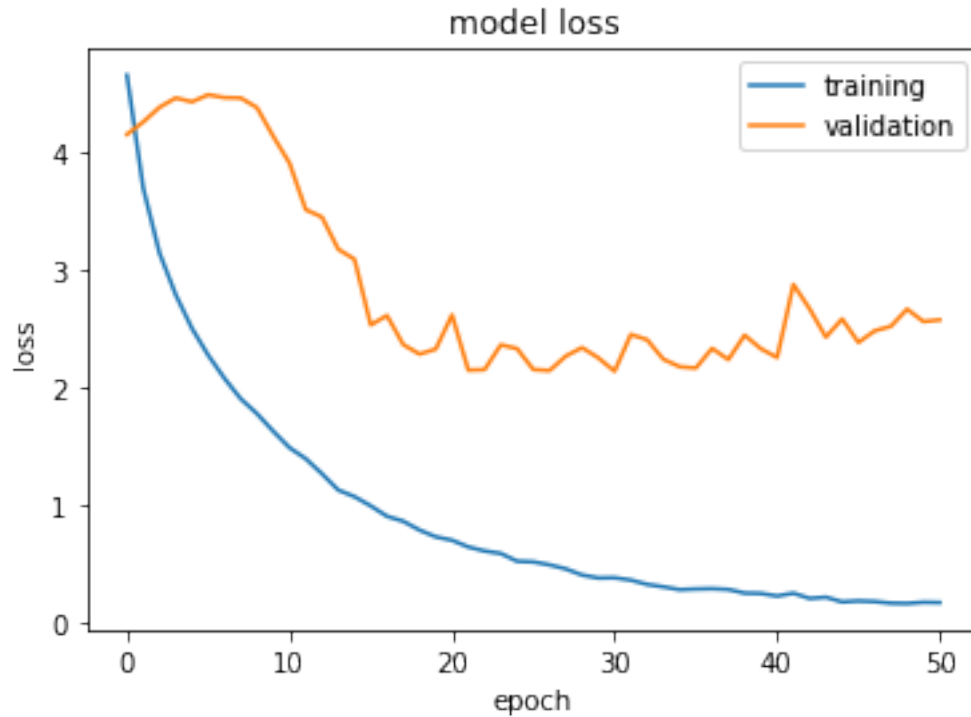
0.9247 - val_loss: 2.1780 - val_accuracy: 0.4665
 Epoch 36/400
 31/31 [=====] - 37s 1s/step - loss: 0.3024 - accuracy:
 0.9267 - val_loss: 2.1670 - val_accuracy: 0.4643
 Epoch 37/400
 31/31 [=====] - 37s 1s/step - loss: 0.2943 - accuracy:
 0.9190 - val_loss: 2.3336 - val_accuracy: 0.4375
 Epoch 38/400
 31/31 [=====] - 37s 1s/step - loss: 0.2721 - accuracy:
 0.9259 - val_loss: 2.2386 - val_accuracy: 0.4397
 Epoch 39/400
 31/31 [=====] - 37s 1s/step - loss: 0.2492 - accuracy:
 0.9399 - val_loss: 2.4461 - val_accuracy: 0.4375
 Epoch 40/400
 31/31 [=====] - 37s 1s/step - loss: 0.2537 - accuracy:
 0.9271 - val_loss: 2.3309 - val_accuracy: 0.4487
 Epoch 41/400
 31/31 [=====] - 37s 1s/step - loss: 0.2149 - accuracy:
 0.9474 - val_loss: 2.2564 - val_accuracy: 0.4531
 Epoch 42/400
 31/31 [=====] - 37s 1s/step - loss: 0.2330 - accuracy:
 0.9441 - val_loss: 2.8769 - val_accuracy: 0.4330
 Epoch 43/400
 31/31 [=====] - 37s 1s/step - loss: 0.1878 - accuracy:
 0.9553 - val_loss: 2.6698 - val_accuracy: 0.4174
 Epoch 44/400
 31/31 [=====] - 37s 1s/step - loss: 0.2190 - accuracy:
 0.9330 - val_loss: 2.4294 - val_accuracy: 0.4442
 Epoch 45/400
 31/31 [=====] - 37s 1s/step - loss: 0.1724 - accuracy:
 0.9581 - val_loss: 2.5843 - val_accuracy: 0.4442
 Epoch 46/400
 31/31 [=====] - 36s 1s/step - loss: 0.1778 - accuracy:
 0.9464 - val_loss: 2.3843 - val_accuracy: 0.4688
 Epoch 47/400
 31/31 [=====] - 36s 1s/step - loss: 0.1820 - accuracy:
 0.9528 - val_loss: 2.4827 - val_accuracy: 0.4241
 Epoch 48/400
 31/31 [=====] - 36s 1s/step - loss: 0.1540 - accuracy:
 0.9590 - val_loss: 2.5212 - val_accuracy: 0.4397
 Epoch 49/400
 31/31 [=====] - 36s 1s/step - loss: 0.1612 - accuracy:
 0.9595 - val_loss: 2.6670 - val_accuracy: 0.4397
 Epoch 50/400
 31/31 [=====] - 36s 1s/step - loss: 0.1595 - accuracy:
 0.9595 - val_loss: 2.5618 - val_accuracy: 0.4286
 Epoch 51/400
 31/31 [=====] - 36s 1s/step - loss: 0.1643 - accuracy:

0.9547 - val_loss: 2.5758 - val_accuracy: 0.4196
Restoring model weights from the end of the best epoch.
Epoch 00051: early stopping

```
[ ]: plt.plot(history3.history['accuracy'])  
plt.plot(history3.history['val_accuracy'])  
plt.title('model accuracy')  
plt.ylabel('accuracy')  
plt.xlabel('epoch')  
plt.legend(['training', 'validation'], loc='best')  
plt.show()
```



```
[ ]: plt.plot(history3.history['loss'])  
plt.plot(history3.history['val_loss'])  
plt.title('model loss')  
plt.ylabel('loss')  
plt.xlabel('epoch')  
plt.legend(['training', 'validation'], loc='best')  
plt.show()
```



This model performs “slightly” better than our model from Experiment 1 and also does so in less epochs. However, there is some variance as the model overfits. We used early stopping with patience level 20, so our model gives best results at around epoch 31.

1.3.4 Experiment 4: Using higher temperature in Softmax

Higher temperature in softmax helps get ‘softer’ weights. This can help improve accuracy further as the ‘probability’ is now more evenly distributed across the classes.

```
[ ]: model14 = Sequential()

# Lambda Layer for adding Padding
model14.add(Lambda(lambda image: tf.image.resize_with_crop_or_pad(
    image, 28, 28), input_shape=(*IMAGE_SIZE, 1)))

# 1st Convolution Layer
model14.add(Conv2D(6, kernel_size=(5,5), padding='same', activation='relu'))
model14.add(BatchNormalization())
model14.add(MaxPooling2D(pool_size=(2,2), strides=2))

# 2nd Convolution Layer
model14.add(Conv2D(16, kernel_size=(5,5), activation='mish'))
model14.add(BatchNormalization())
model14.add(MaxPooling2D(pool_size=(2,2), strides=2))
```

```

# Passing to a Fully Connected Layer
model4.add(Flatten())

# 1st Fully Connected Layer
model4.add(Dense(256, activation=mish))
model4.add(BatchNormalization())
model4.add(Dropout(0.4))

# 2nd Fully Connected Layer
model4.add(Dense(128, activation=mish))
model4.add(BatchNormalization())
model4.add(Dropout(0.4))

# Output Layer
# Increasing the softmax temperature
temp = 5
model4.add(Lambda(lambda x: x / temp))
model4.add(Dense(62, activation='softmax'))

```

```
[ ]: model4.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 28, 28, 6)	156
batch_normalization_1 (Batch Normalization)	(None, 28, 28, 6)	24
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 6)	0
conv2d_2 (Conv2D)	(None, 10, 10, 16)	2416
batch_normalization_2 (Batch Normalization)	(None, 10, 10, 16)	64
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 256)	102656
batch_normalization_3 (Batch Normalization)	(None, 256)	1024
dropout (Dropout)	(None, 256)	0

```

-----
dense_1 (Dense)                (None, 128)                32896
-----
batch_normalization_4 (Batch Normalization) (None, 128)                512
-----
dropout_1 (Dropout)            (None, 128)                0
-----
lambda_2 (Lambda)              (None, 128)                0
-----
dense_2 (Dense)                (None, 62)                 7998
=====
Total params: 147,746
Trainable params: 146,934
Non-trainable params: 812
-----

```

```
[ ]: model4.compile(loss='categorical_crossentropy', optimizer=Adam(),
    ↪metrics=['accuracy'])
```

```
[ ]: checkpoint_filepath4 = 'exp4/checkpoint'
model_checkpoint_callback4 = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath4,
    save_weights_only=True,
    monitor='val_loss',
    mode='min',
    save_best_only=True)
```

```
[ ]: history4 = model4.fit(
    train_generator1,
    epochs=EPOCHS,
    validation_data=validation_generator1,
    steps_per_epoch = train_generator1.samples // BATCH_SIZE,
    validation_steps = validation_generator1.samples // BATCH_SIZE,
    callbacks=[model_checkpoint_callback4, early_stopping_callback]
)
```

Epoch 1/400

31/31 [=====] - 63s 991ms/step - loss: 4.1323 - accuracy: 0.0277 - val_loss: 4.1238 - val_accuracy: 0.0134

Epoch 2/400

31/31 [=====] - 30s 980ms/step - loss: 3.8507 - accuracy: 0.1154 - val_loss: 4.1265 - val_accuracy: 0.0179

Epoch 3/400

31/31 [=====] - 30s 974ms/step - loss: 3.6529 - accuracy: 0.2144 - val_loss: 4.1399 - val_accuracy: 0.0179

Epoch 4/400

31/31 [=====] - 30s 977ms/step - loss: 3.4515 - accuracy: 0.2855 - val_loss: 4.1655 - val_accuracy: 0.0179

Epoch 5/400
31/31 [=====] - 30s 973ms/step - loss: 3.2286 - accuracy: 0.3662 - val_loss: 4.1651 - val_accuracy: 0.0179

Epoch 6/400
31/31 [=====] - 30s 973ms/step - loss: 3.0389 - accuracy: 0.4318 - val_loss: 4.1768 - val_accuracy: 0.0156

Epoch 7/400
31/31 [=====] - 30s 976ms/step - loss: 2.8271 - accuracy: 0.4688 - val_loss: 4.1357 - val_accuracy: 0.0134

Epoch 8/400
31/31 [=====] - 30s 971ms/step - loss: 2.6800 - accuracy: 0.5179 - val_loss: 4.0965 - val_accuracy: 0.0156

Epoch 9/400
31/31 [=====] - 30s 985ms/step - loss: 2.4400 - accuracy: 0.5749 - val_loss: 4.0263 - val_accuracy: 0.0201

Epoch 10/400
31/31 [=====] - 30s 973ms/step - loss: 2.2798 - accuracy: 0.6201 - val_loss: 3.9638 - val_accuracy: 0.0491

Epoch 11/400
31/31 [=====] - 30s 975ms/step - loss: 2.0604 - accuracy: 0.6669 - val_loss: 3.7348 - val_accuracy: 0.0670

Epoch 12/400
31/31 [=====] - 30s 981ms/step - loss: 1.9174 - accuracy: 0.6509 - val_loss: 3.5989 - val_accuracy: 0.1049

Epoch 13/400
31/31 [=====] - 30s 973ms/step - loss: 1.7477 - accuracy: 0.7129 - val_loss: 3.4710 - val_accuracy: 0.1406

Epoch 14/400
31/31 [=====] - 30s 972ms/step - loss: 1.6214 - accuracy: 0.7249 - val_loss: 3.1647 - val_accuracy: 0.2188

Epoch 15/400
31/31 [=====] - 30s 974ms/step - loss: 1.4487 - accuracy: 0.7668 - val_loss: 3.0287 - val_accuracy: 0.2321

Epoch 16/400
31/31 [=====] - 30s 973ms/step - loss: 1.2949 - accuracy: 0.7949 - val_loss: 2.5981 - val_accuracy: 0.3281

Epoch 17/400
31/31 [=====] - 30s 975ms/step - loss: 1.2116 - accuracy: 0.8257 - val_loss: 2.5388 - val_accuracy: 0.3438

Epoch 18/400
31/31 [=====] - 30s 980ms/step - loss: 1.0757 - accuracy: 0.8483 - val_loss: 2.3304 - val_accuracy: 0.4107

Epoch 19/400
31/31 [=====] - 30s 982ms/step - loss: 0.9624 - accuracy: 0.8597 - val_loss: 2.2412 - val_accuracy: 0.4286

Epoch 20/400
31/31 [=====] - 30s 977ms/step - loss: 0.8800 - accuracy: 0.8718 - val_loss: 2.1417 - val_accuracy: 0.4308

Epoch 21/400
31/31 [=====] - 30s 974ms/step - loss: 0.8075 - accuracy: 0.8760 - val_loss: 2.1626 - val_accuracy: 0.4308
Epoch 22/400
31/31 [=====] - 30s 976ms/step - loss: 0.7716 - accuracy: 0.8862 - val_loss: 2.2033 - val_accuracy: 0.4129
Epoch 23/400
31/31 [=====] - 30s 976ms/step - loss: 0.6725 - accuracy: 0.9092 - val_loss: 1.9373 - val_accuracy: 0.4754
Epoch 24/400
31/31 [=====] - 30s 970ms/step - loss: 0.5959 - accuracy: 0.9146 - val_loss: 1.9298 - val_accuracy: 0.4821
Epoch 25/400
31/31 [=====] - 30s 974ms/step - loss: 0.5605 - accuracy: 0.9423 - val_loss: 1.9122 - val_accuracy: 0.4888
Epoch 26/400
31/31 [=====] - 30s 973ms/step - loss: 0.4715 - accuracy: 0.9533 - val_loss: 1.9479 - val_accuracy: 0.4598
Epoch 27/400
31/31 [=====] - 30s 970ms/step - loss: 0.4661 - accuracy: 0.9359 - val_loss: 1.9799 - val_accuracy: 0.4531
Epoch 28/400
31/31 [=====] - 30s 973ms/step - loss: 0.4278 - accuracy: 0.9543 - val_loss: 1.8585 - val_accuracy: 0.5022
Epoch 29/400
31/31 [=====] - 30s 978ms/step - loss: 0.3650 - accuracy: 0.9591 - val_loss: 1.9026 - val_accuracy: 0.4688
Epoch 30/400
31/31 [=====] - 30s 976ms/step - loss: 0.3498 - accuracy: 0.9648 - val_loss: 1.9449 - val_accuracy: 0.4487
Epoch 31/400
31/31 [=====] - 30s 973ms/step - loss: 0.3252 - accuracy: 0.9624 - val_loss: 2.0093 - val_accuracy: 0.4464
Epoch 32/400
31/31 [=====] - 30s 978ms/step - loss: 0.2969 - accuracy: 0.9650 - val_loss: 1.9374 - val_accuracy: 0.4487
Epoch 33/400
31/31 [=====] - 30s 972ms/step - loss: 0.2831 - accuracy: 0.9660 - val_loss: 1.8723 - val_accuracy: 0.4777
Epoch 34/400
31/31 [=====] - 30s 976ms/step - loss: 0.2665 - accuracy: 0.9678 - val_loss: 1.9376 - val_accuracy: 0.4598
Epoch 35/400
31/31 [=====] - 30s 972ms/step - loss: 0.2329 - accuracy: 0.9817 - val_loss: 1.9904 - val_accuracy: 0.4688
Epoch 36/400
31/31 [=====] - 30s 977ms/step - loss: 0.2290 - accuracy: 0.9727 - val_loss: 2.0521 - val_accuracy: 0.4107

```

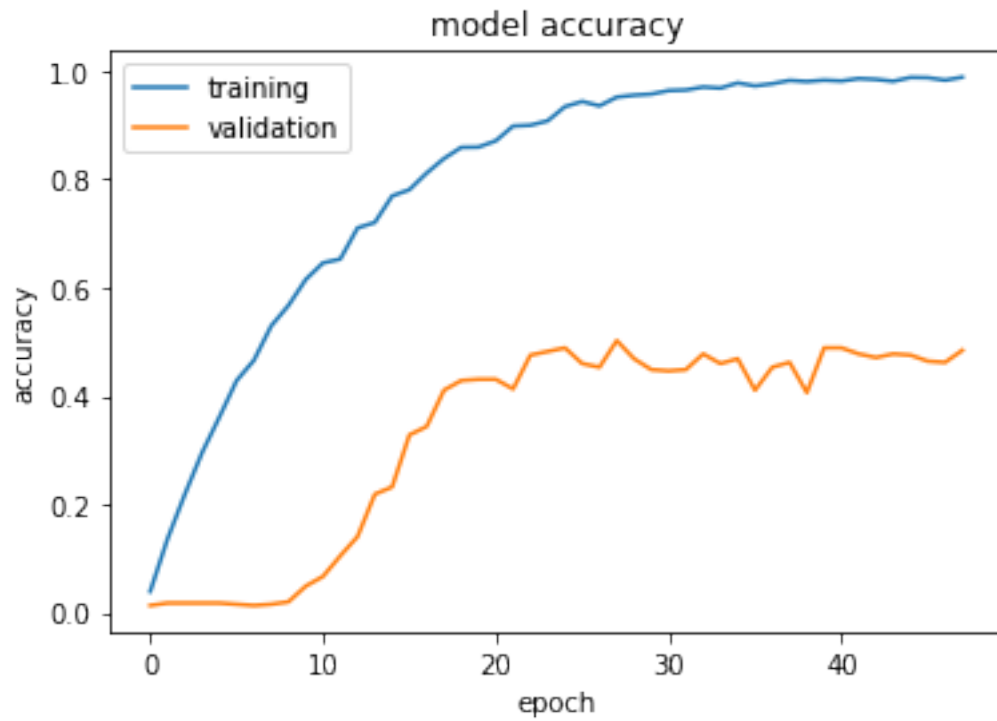
Epoch 37/400
31/31 [=====] - 30s 980ms/step - loss: 0.2256 -
accuracy: 0.9739 - val_loss: 1.9637 - val_accuracy: 0.4531
Epoch 38/400
31/31 [=====] - 30s 979ms/step - loss: 0.1992 -
accuracy: 0.9830 - val_loss: 1.9705 - val_accuracy: 0.4621
Epoch 39/400
31/31 [=====] - 30s 983ms/step - loss: 0.1922 -
accuracy: 0.9827 - val_loss: 2.1982 - val_accuracy: 0.4062
Epoch 40/400
31/31 [=====] - 30s 983ms/step - loss: 0.1709 -
accuracy: 0.9852 - val_loss: 1.9766 - val_accuracy: 0.4888
Epoch 41/400
31/31 [=====] - 30s 975ms/step - loss: 0.1803 -
accuracy: 0.9800 - val_loss: 1.9562 - val_accuracy: 0.4888
Epoch 42/400
31/31 [=====] - 30s 982ms/step - loss: 0.1542 -
accuracy: 0.9809 - val_loss: 2.0347 - val_accuracy: 0.4777
Epoch 43/400
31/31 [=====] - 30s 978ms/step - loss: 0.1449 -
accuracy: 0.9886 - val_loss: 2.0074 - val_accuracy: 0.4710
Epoch 44/400
31/31 [=====] - 30s 979ms/step - loss: 0.1430 -
accuracy: 0.9827 - val_loss: 1.9388 - val_accuracy: 0.4777
Epoch 45/400
31/31 [=====] - 30s 978ms/step - loss: 0.1281 -
accuracy: 0.9916 - val_loss: 1.9889 - val_accuracy: 0.4754
Epoch 46/400
31/31 [=====] - 30s 980ms/step - loss: 0.1262 -
accuracy: 0.9885 - val_loss: 1.9810 - val_accuracy: 0.4643
Epoch 47/400
31/31 [=====] - 30s 983ms/step - loss: 0.1284 -
accuracy: 0.9813 - val_loss: 2.0325 - val_accuracy: 0.4621
Epoch 48/400
31/31 [=====] - 30s 978ms/step - loss: 0.1186 -
accuracy: 0.9898 - val_loss: 2.0685 - val_accuracy: 0.4844
Restoring model weights from the end of the best epoch.
Epoch 00048: early stopping

```

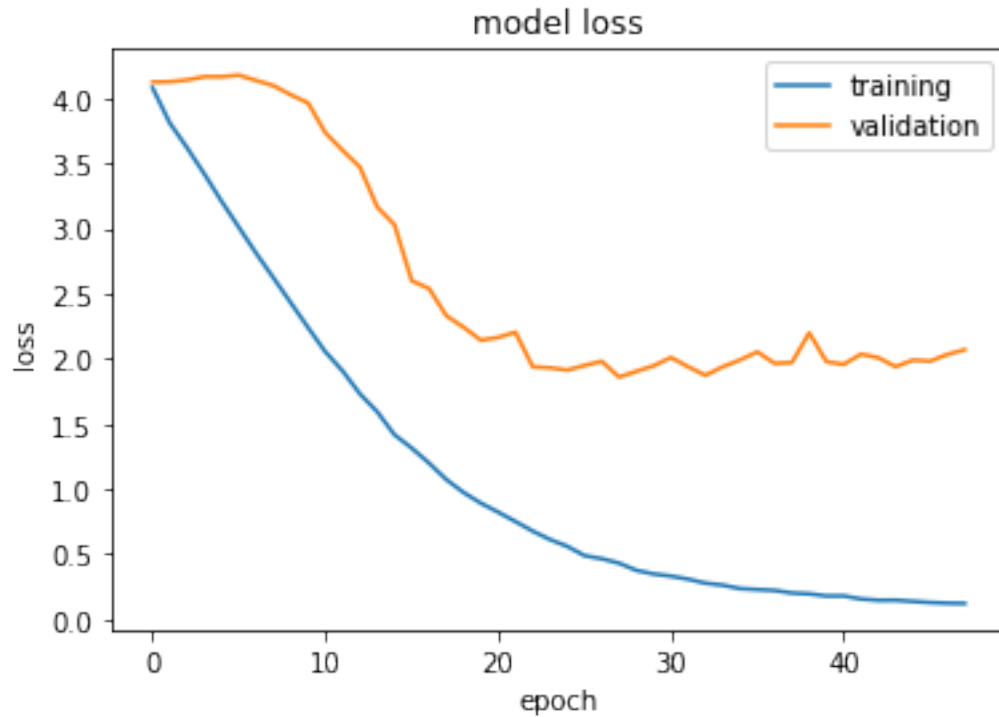
```

[ ]: plt.plot(history4.history['accuracy'])
plt.plot(history4.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()

```

```
[ ]: plt.plot(history4.history['loss'])
plt.plot(history4.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()
```



This is a slight improvement over the last experiment. This can be due to the case that the ‘probabilities’ are now more spread out.

1.3.5 Experiment 5: Changing the Model Architecture

From ref. [1] and [2], I modify the architecture to see if it gives better results. This model is actually shown to give some very good results on the MNIST competition in Kaggle.

```
[12]: model15 = Sequential()

# Lambda Layer for adding Padding
model15.add(Lambda(lambda image: tf.image.resize_with_crop_or_pad(
    image, 28, 28), input_shape=(*IMAGE_SIZE, 1)))

# 1st Convolution Layer
model15.add(Conv2D(32, kernel_size=3, activation=mish))
model15.add(BatchNormalization())
model15.add(Conv2D(32, kernel_size=3, activation=mish))
model15.add(BatchNormalization())
model15.add(Conv2D(32, kernel_size=5, strides=2, padding='same',
    ↪activation=mish))
model15.add(BatchNormalization())
model15.add(Dropout(0.4))
```

```

# 2nd Convolution Layer
model5.add(Conv2D(64, kernel_size=3, activation=mish))
model5.add(BatchNormalization())
model5.add(Conv2D(64, kernel_size=3, activation=mish))
model5.add(BatchNormalization())
model5.add(Conv2D(64, kernel_size=5, strides=2, padding='same',
    ↪activation=mish))
model5.add(BatchNormalization())
model5.add(Dropout(0.4))

# 3rd Convolution Layer
model5.add(Conv2D(128, kernel_size = 4, activation=mish))
model5.add(BatchNormalization())

# Passing to a Fully Connected Layer
model5.add(Flatten())
model5.add(Dropout(0.4))

# Output Layer
# Increasing the softmax temperature
temp = 5
model5.add(Lambda(lambda x: x / temp))

model5.add(Dense(62, activation='softmax'))

```

```
[13]: model5.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 28, 28, 1)	0
conv2d_7 (Conv2D)	(None, 26, 26, 32)	320
batch_normalization_7 (Batch Normalization)	(None, 26, 26, 32)	128
conv2d_8 (Conv2D)	(None, 24, 24, 32)	9248
batch_normalization_8 (Batch Normalization)	(None, 24, 24, 32)	128
conv2d_9 (Conv2D)	(None, 12, 12, 32)	25632
batch_normalization_9 (Batch Normalization)	(None, 12, 12, 32)	128
dropout_3 (Dropout)	(None, 12, 12, 32)	0

conv2d_10 (Conv2D)	(None, 10, 10, 64)	18496

batch_normalization_10 (Batch Normalization)	(None, 10, 10, 64)	256

conv2d_11 (Conv2D)	(None, 8, 8, 64)	36928

batch_normalization_11 (Batch Normalization)	(None, 8, 8, 64)	256

conv2d_12 (Conv2D)	(None, 4, 4, 64)	102464

batch_normalization_12 (Batch Normalization)	(None, 4, 4, 64)	256

dropout_4 (Dropout)	(None, 4, 4, 64)	0

conv2d_13 (Conv2D)	(None, 1, 1, 128)	131200

batch_normalization_13 (Batch Normalization)	(None, 1, 1, 128)	512

flatten_1 (Flatten)	(None, 128)	0

dropout_5 (Dropout)	(None, 128)	0

lambda_2 (Lambda)	(None, 128)	0

dense_1 (Dense)	(None, 62)	7998
=====		
Total params: 333,950		
Trainable params: 333,118		
Non-trainable params: 832		

```
[18]: model5.compile(loss='categorical_crossentropy', optimizer=Adam(),
    ↪metrics=['accuracy'])
```

Saving the Checkpoint

```
[19]: checkpoint_filepath5 = 'exp5/checkpoint'
model_checkpoint_callback5 = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath5,
    save_weights_only=True,
    monitor='val_loss',
    mode='min',
    save_best_only=True)
```

```
[20]: history5 = model5.fit(
    train_generator1,
    epochs=EPOCHS,
    validation_data=validation_generator1,
```

```
steps_per_epoch = train_generator1.samples // BATCH_SIZE,  
validation_steps = validation_generator1.samples // BATCH_SIZE,  
callbacks=[model_checkpoint_callback5, early_stopping_callback]  
)
```

```
Epoch 1/400  
31/31 [=====] - 33s 1s/step - loss: 3.1841 - accuracy:  
0.3234 - val_loss: 4.0727 - val_accuracy: 0.0246  
Epoch 2/400  
31/31 [=====] - 31s 1s/step - loss: 2.9242 - accuracy:  
0.4321 - val_loss: 4.1239 - val_accuracy: 0.0290  
Epoch 3/400  
31/31 [=====] - 31s 1000ms/step - loss: 2.6343 -  
accuracy: 0.5060 - val_loss: 4.0812 - val_accuracy: 0.0446  
Epoch 4/400  
31/31 [=====] - 31s 1s/step - loss: 2.4394 - accuracy:  
0.5597 - val_loss: 4.0960 - val_accuracy: 0.0424  
Epoch 5/400  
31/31 [=====] - 31s 1s/step - loss: 2.1861 - accuracy:  
0.6308 - val_loss: 4.0556 - val_accuracy: 0.0536  
Epoch 6/400  
31/31 [=====] - 31s 1s/step - loss: 1.9959 - accuracy:  
0.6641 - val_loss: 3.9447 - val_accuracy: 0.0915  
Epoch 7/400  
31/31 [=====] - 31s 1s/step - loss: 1.7984 - accuracy:  
0.6955 - val_loss: 3.7015 - val_accuracy: 0.1161  
Epoch 8/400  
31/31 [=====] - 31s 1s/step - loss: 1.5956 - accuracy:  
0.7308 - val_loss: 2.8872 - val_accuracy: 0.3080  
Epoch 9/400  
31/31 [=====] - 31s 1s/step - loss: 1.4371 - accuracy:  
0.7673 - val_loss: 2.9189 - val_accuracy: 0.2701  
Epoch 10/400  
31/31 [=====] - 31s 1s/step - loss: 1.2780 - accuracy:  
0.7878 - val_loss: 2.5545 - val_accuracy: 0.3728  
Epoch 11/400  
31/31 [=====] - 31s 1s/step - loss: 1.1382 - accuracy:  
0.8218 - val_loss: 2.2063 - val_accuracy: 0.4643  
Epoch 12/400  
31/31 [=====] - 31s 1s/step - loss: 0.9887 - accuracy:  
0.8514 - val_loss: 2.4770 - val_accuracy: 0.3817  
Epoch 13/400  
31/31 [=====] - 31s 999ms/step - loss: 0.8759 -  
accuracy: 0.8842 - val_loss: 1.6656 - val_accuracy: 0.6049  
Epoch 14/400  
31/31 [=====] - 31s 1s/step - loss: 0.7964 - accuracy:  
0.8834 - val_loss: 1.5947 - val_accuracy: 0.6071
```

Epoch 15/400
31/31 [=====] - 31s 1s/step - loss: 0.7168 - accuracy: 0.8920 - val_loss: 1.3526 - val_accuracy: 0.6629
Epoch 16/400
31/31 [=====] - 31s 1s/step - loss: 0.6451 - accuracy: 0.9087 - val_loss: 1.3519 - val_accuracy: 0.6339
Epoch 17/400
31/31 [=====] - 31s 1s/step - loss: 0.5804 - accuracy: 0.9138 - val_loss: 1.2571 - val_accuracy: 0.6518
Epoch 18/400
31/31 [=====] - 31s 1s/step - loss: 0.5334 - accuracy: 0.9205 - val_loss: 1.1376 - val_accuracy: 0.6830
Epoch 19/400
31/31 [=====] - 31s 1s/step - loss: 0.4586 - accuracy: 0.9406 - val_loss: 1.1997 - val_accuracy: 0.6964
Epoch 20/400
31/31 [=====] - 31s 1s/step - loss: 0.4115 - accuracy: 0.9452 - val_loss: 1.3390 - val_accuracy: 0.6384
Epoch 21/400
31/31 [=====] - 32s 1s/step - loss: 0.3837 - accuracy: 0.9473 - val_loss: 1.0999 - val_accuracy: 0.6942
Epoch 22/400
31/31 [=====] - 31s 1s/step - loss: 0.3531 - accuracy: 0.9535 - val_loss: 1.0842 - val_accuracy: 0.6875
Epoch 23/400
31/31 [=====] - 31s 1s/step - loss: 0.3024 - accuracy: 0.9622 - val_loss: 1.1422 - val_accuracy: 0.6763
Epoch 24/400
31/31 [=====] - 31s 1s/step - loss: 0.2903 - accuracy: 0.9608 - val_loss: 1.1675 - val_accuracy: 0.6719
Epoch 25/400
31/31 [=====] - 31s 1s/step - loss: 0.2511 - accuracy: 0.9785 - val_loss: 1.1351 - val_accuracy: 0.6830
Epoch 26/400
31/31 [=====] - 31s 1s/step - loss: 0.2368 - accuracy: 0.9721 - val_loss: 1.0954 - val_accuracy: 0.6897
Epoch 27/400
31/31 [=====] - 31s 1s/step - loss: 0.2059 - accuracy: 0.9838 - val_loss: 1.2542 - val_accuracy: 0.6585
Epoch 28/400
31/31 [=====] - 31s 1s/step - loss: 0.1972 - accuracy: 0.9802 - val_loss: 1.0532 - val_accuracy: 0.6964
Epoch 29/400
31/31 [=====] - 31s 1s/step - loss: 0.1714 - accuracy: 0.9878 - val_loss: 1.0767 - val_accuracy: 0.6964
Epoch 30/400
31/31 [=====] - 31s 1s/step - loss: 0.1723 - accuracy: 0.9813 - val_loss: 1.0640 - val_accuracy: 0.7098

Epoch 31/400
31/31 [=====] - 31s 1s/step - loss: 0.1543 - accuracy: 0.9878 - val_loss: 1.0837 - val_accuracy: 0.7009

Epoch 32/400
31/31 [=====] - 31s 999ms/step - loss: 0.1501 - accuracy: 0.9857 - val_loss: 1.1099 - val_accuracy: 0.6942

Epoch 33/400
31/31 [=====] - 31s 1s/step - loss: 0.1503 - accuracy: 0.9784 - val_loss: 1.8992 - val_accuracy: 0.5156

Epoch 34/400
31/31 [=====] - 31s 1000ms/step - loss: 0.1311 - accuracy: 0.9885 - val_loss: 1.5575 - val_accuracy: 0.5848

Epoch 35/400
31/31 [=====] - 31s 997ms/step - loss: 0.1309 - accuracy: 0.9828 - val_loss: 1.0923 - val_accuracy: 0.6808

Epoch 36/400
31/31 [=====] - 31s 1s/step - loss: 0.1215 - accuracy: 0.9868 - val_loss: 1.0968 - val_accuracy: 0.6897

Epoch 37/400
31/31 [=====] - 31s 1000ms/step - loss: 0.1171 - accuracy: 0.9904 - val_loss: 1.1810 - val_accuracy: 0.6830

Epoch 38/400
31/31 [=====] - 32s 1s/step - loss: 0.1018 - accuracy: 0.9915 - val_loss: 1.0821 - val_accuracy: 0.6897

Epoch 39/400
31/31 [=====] - 34s 1s/step - loss: 0.1058 - accuracy: 0.9864 - val_loss: 1.0334 - val_accuracy: 0.7009

Epoch 40/400
31/31 [=====] - 33s 1s/step - loss: 0.0967 - accuracy: 0.9879 - val_loss: 1.1909 - val_accuracy: 0.6897

Epoch 41/400
31/31 [=====] - 32s 1s/step - loss: 0.0905 - accuracy: 0.9896 - val_loss: 1.1311 - val_accuracy: 0.6920

Epoch 42/400
31/31 [=====] - 32s 1s/step - loss: 0.0891 - accuracy: 0.9928 - val_loss: 1.0773 - val_accuracy: 0.7143

Epoch 43/400
31/31 [=====] - 32s 1s/step - loss: 0.0850 - accuracy: 0.9908 - val_loss: 1.1309 - val_accuracy: 0.6942

Epoch 44/400
31/31 [=====] - 32s 1s/step - loss: 0.0850 - accuracy: 0.9880 - val_loss: 1.4652 - val_accuracy: 0.6429

Epoch 45/400
31/31 [=====] - 32s 1s/step - loss: 0.0786 - accuracy: 0.9925 - val_loss: 1.0613 - val_accuracy: 0.7054

Epoch 46/400
31/31 [=====] - 32s 1s/step - loss: 0.0687 - accuracy: 0.9924 - val_loss: 1.1210 - val_accuracy: 0.6830

```

Epoch 47/400
31/31 [=====] - 31s 1s/step - loss: 0.0765 - accuracy:
0.9910 - val_loss: 1.2553 - val_accuracy: 0.6540
Epoch 48/400
31/31 [=====] - 31s 1s/step - loss: 0.0688 - accuracy:
0.9947 - val_loss: 1.0441 - val_accuracy: 0.7076
Epoch 49/400
31/31 [=====] - 31s 1s/step - loss: 0.0616 - accuracy:
0.9929 - val_loss: 1.1608 - val_accuracy: 0.6942
Epoch 50/400
31/31 [=====] - 32s 1s/step - loss: 0.0683 - accuracy:
0.9918 - val_loss: 1.9271 - val_accuracy: 0.5357
Epoch 51/400
31/31 [=====] - 32s 1s/step - loss: 0.0758 - accuracy:
0.9872 - val_loss: 1.2352 - val_accuracy: 0.6786
Epoch 52/400
31/31 [=====] - 31s 1s/step - loss: 0.0678 - accuracy:
0.9896 - val_loss: 1.1573 - val_accuracy: 0.6853
Epoch 53/400
31/31 [=====] - 31s 1s/step - loss: 0.0777 - accuracy:
0.9883 - val_loss: 1.1964 - val_accuracy: 0.6830
Epoch 54/400
31/31 [=====] - 31s 1s/step - loss: 0.0666 - accuracy:
0.9878 - val_loss: 1.2779 - val_accuracy: 0.6518
Epoch 55/400
31/31 [=====] - 31s 1s/step - loss: 0.0598 - accuracy:
0.9932 - val_loss: 1.2255 - val_accuracy: 0.6920
Epoch 56/400
31/31 [=====] - 31s 1s/step - loss: 0.0656 - accuracy:
0.9905 - val_loss: 1.1773 - val_accuracy: 0.6875
Epoch 57/400
31/31 [=====] - 31s 1s/step - loss: 0.0590 - accuracy:
0.9905 - val_loss: 1.1688 - val_accuracy: 0.6875
Epoch 58/400
31/31 [=====] - 31s 1s/step - loss: 0.0446 - accuracy:
0.9944 - val_loss: 1.1803 - val_accuracy: 0.6652
Epoch 59/400
31/31 [=====] - 31s 1s/step - loss: 0.0505 - accuracy:
0.9906 - val_loss: 1.5860 - val_accuracy: 0.5982
Restoring model weights from the end of the best epoch.
Epoch 00059: early stopping

```

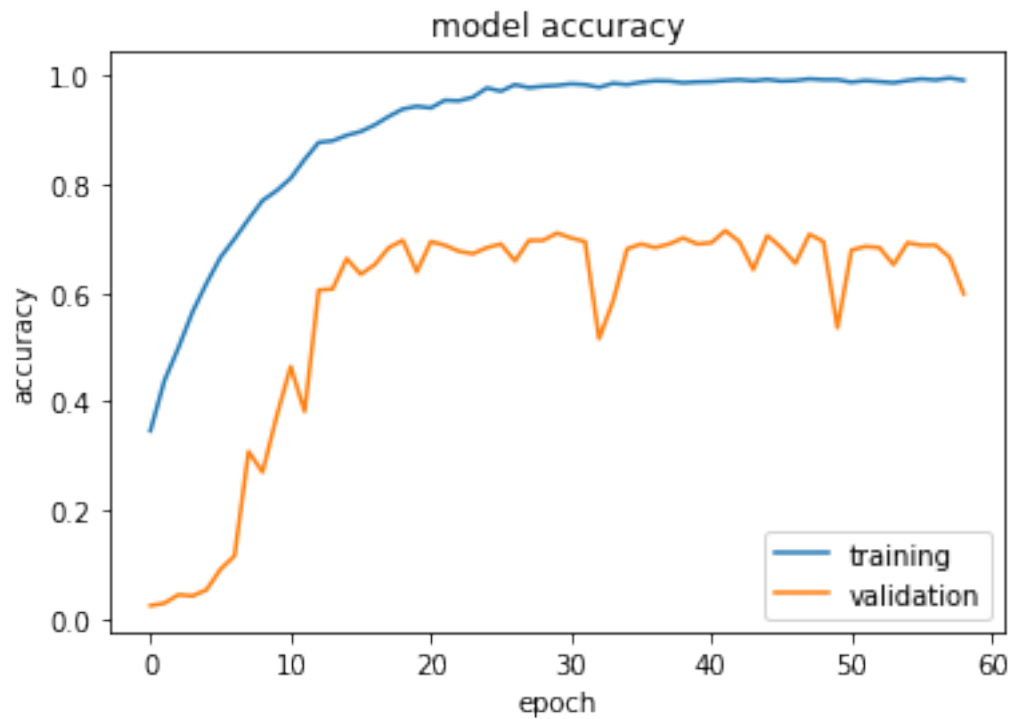
```

[21]: plt.plot(history5.history['accuracy'])
      plt.plot(history5.history['val_accuracy'])
      plt.title('model accuracy')
      plt.ylabel('accuracy')
      plt.xlabel('epoch')

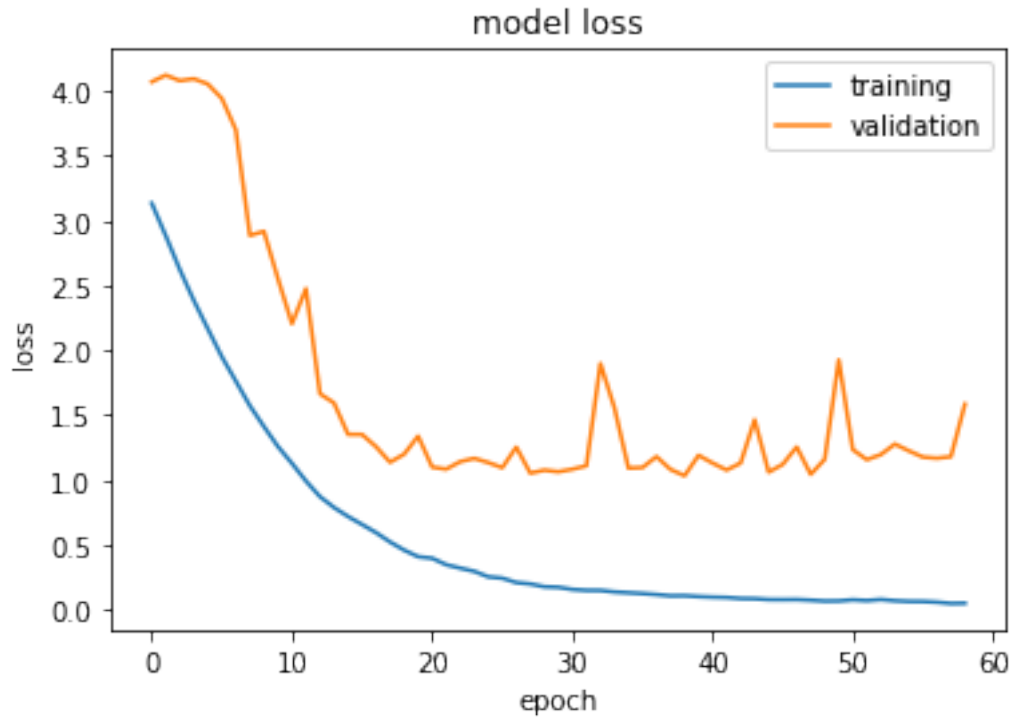
```



```
plt.legend(['training', 'validation'], loc='best')  
plt.show()
```



```
[22]: plt.plot(history5.history['loss'])  
plt.plot(history5.history['val_loss'])  
plt.title('model loss')  
plt.ylabel('loss')  
plt.xlabel('epoch')  
plt.legend(['training', 'validation'], loc='best')  
plt.show()
```



This gives the best accuracy among all of the previous examples. Works great!

1.3.6 Experiment 6: Changing the Model Architecture even further: Using Efficient-net

Efficient Net is being used a lot nowadays so, I'll see how this architecture works for this dataset. I don't have very high hopes for this because: 1. We are not using pre-trained weights 2. We don't have a large trainign data

But I'll still give it a try to see how it does.

For this EfficientNet Model, I would use Early Stopping with larger patience value. This is so because the Keras Website says

“Note: the accuracy will increase very slowly and may overfit.”

on Training a model with EfficientNet from scratch.

Saving the Checkpoint

```
[ ]: checkpoint_filepath6 = 'exp6/checkpoint'
model_checkpoint_callback6 = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath6,
    save_weights_only=True,
    monitor='val_loss',
    mode='min',
    save_best_only=True)
```

Early Stopping Callback

```
[ ]: early_stopping_callback2 = tf.keras.callbacks.EarlyStopping(  
    monitor='val_loss',  
    mode='min',  
    patience=100,  
    restore_best_weights=True,  
    verbose=1)
```

```
[ ]: from tensorflow.keras.applications import EfficientNetB0  
  
# Initializer taken from the source code  
DENSE_KERNEL_INITIALIZER = {  
    'class_name': 'VarianceScaling',  
    'config': {  
        'scale': 1. / 3.,  
        'mode': 'fan_out',  
        'distribution': 'uniform'  
    }  
}  
  
# Input Layer  
inputs = tf.keras.layers.Input(shape=(*IMAGE_SIZE, 1))  
  
# Lambda Layer for adding Padding  
x = tf.keras.layers.Lambda(lambda image: tf.image.resize_with_crop_or_pad(  
    image, 28, 28), input_shape=(*IMAGE_SIZE, 1))(inputs)  
  
# Efficient layer except the top layer  
x = EfficientNetB0(include_top=False, weights=None,  
    input_shape=(48, 48, 1))(x)  
  
# Top  
  
# Global Average Pooling Layer  
x = tf.keras.layers.GlobalAveragePooling2D(name='avg_pool')(x)  
x = tf.keras.layers.Dropout(0.4, name='top_dropout')(x)  
  
# Output Layer  
outputs = tf.keras.layers.Dense(62,  
    activation='softmax',  
    kernel_initializer=DENSE_KERNEL_INITIALIZER,  
    name='predictions')(x)  
  
model6 = tf.keras.Model(inputs, outputs)  
  
model6.compile(  

```

```

optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"]
)

model6.summary()

```

Model: "model_7"

Layer (type)	Output Shape	Param #
input_19 (InputLayer)	[(None, 45, 60, 1)]	0
efficientnetb0 (Functional)	(None, 2, 2, 1280)	4048991
avg_pool (GlobalAveragePooli	(None, 1280)	0
top_dropout (Dropout)	(None, 1280)	0
predictions (Dense)	(None, 62)	79422

Total params: 4,128,413
 Trainable params: 4,086,394
 Non-trainable params: 42,019

```

[ ]: history6 = model6.fit(
    train_generator1,
    epochs=EPOCHS,
    validation_data=validation_generator1,
    steps_per_epoch = train_generator1.samples // BATCH_SIZE,
    validation_steps = validation_generator1.samples // BATCH_SIZE,
    callbacks=[model_checkpoint_callback6, early_stopping_callback2]
)

```

Epoch 1/400

31/31 [=====] - 42s 1s/step - loss: 4.9151 - accuracy: 0.0201 - val_loss: 4.1315 - val_accuracy: 0.0179

Epoch 2/400

31/31 [=====] - 32s 1s/step - loss: 4.4224 - accuracy: 0.0468 - val_loss: 4.1495 - val_accuracy: 0.0156

Epoch 3/400

31/31 [=====] - 32s 1s/step - loss: 4.2469 - accuracy: 0.0709 - val_loss: 4.1952 - val_accuracy: 0.0179

Epoch 4/400

31/31 [=====] - 32s 1s/step - loss: 3.8510 - accuracy: 0.1198 - val_loss: 4.2058 - val_accuracy: 0.0134

Epoch 5/400

31/31 [=====] - 32s 1s/step - loss: 3.4325 - accuracy: 0.2088 - val_loss: 4.2741 - val_accuracy: 0.0156

Epoch 6/400
31/31 [=====] - 32s 1s/step - loss: 2.8368 - accuracy:
0.3269 - val_loss: 4.3717 - val_accuracy: 0.0179
Epoch 7/400
31/31 [=====] - 32s 1s/step - loss: 2.2589 - accuracy:
0.4527 - val_loss: 4.7584 - val_accuracy: 0.0156
Epoch 8/400
31/31 [=====] - 32s 1s/step - loss: 1.6094 - accuracy:
0.5704 - val_loss: 5.0353 - val_accuracy: 0.0134
Epoch 9/400
31/31 [=====] - 32s 1s/step - loss: 1.2984 - accuracy:
0.6688 - val_loss: 5.7188 - val_accuracy: 0.0179
Epoch 10/400
31/31 [=====] - 32s 1s/step - loss: 0.9605 - accuracy:
0.7401 - val_loss: 6.1509 - val_accuracy: 0.0156
Epoch 11/400
31/31 [=====] - 32s 1s/step - loss: 0.7479 - accuracy:
0.7929 - val_loss: 6.6623 - val_accuracy: 0.0156
Epoch 12/400
31/31 [=====] - 32s 1s/step - loss: 0.6724 - accuracy:
0.7995 - val_loss: 7.3675 - val_accuracy: 0.0156
Epoch 13/400
31/31 [=====] - 32s 1s/step - loss: 0.6412 - accuracy:
0.8292 - val_loss: 8.9104 - val_accuracy: 0.0156
Epoch 14/400
31/31 [=====] - 32s 1s/step - loss: 0.4667 - accuracy:
0.8784 - val_loss: 8.4325 - val_accuracy: 0.0179
Epoch 15/400
31/31 [=====] - 32s 1s/step - loss: 0.4531 - accuracy:
0.8752 - val_loss: 8.1772 - val_accuracy: 0.0156
Epoch 16/400
31/31 [=====] - 32s 1s/step - loss: 0.4186 - accuracy:
0.8964 - val_loss: 8.1100 - val_accuracy: 0.0179
Epoch 17/400
31/31 [=====] - 32s 1s/step - loss: 0.4693 - accuracy:
0.8748 - val_loss: 6.7461 - val_accuracy: 0.0268
Epoch 18/400
31/31 [=====] - 32s 1s/step - loss: 0.4318 - accuracy:
0.8759 - val_loss: 6.5141 - val_accuracy: 0.0179
Epoch 19/400
31/31 [=====] - 32s 1s/step - loss: 0.4297 - accuracy:
0.8865 - val_loss: 8.9487 - val_accuracy: 0.0223
Epoch 20/400
31/31 [=====] - 32s 1s/step - loss: 0.3713 - accuracy:
0.8911 - val_loss: 9.2894 - val_accuracy: 0.0179
Epoch 21/400
31/31 [=====] - 32s 1s/step - loss: 0.4135 - accuracy:
0.8883 - val_loss: 9.7851 - val_accuracy: 0.0179

Epoch 22/400
31/31 [=====] - 32s 1s/step - loss: 0.3804 - accuracy: 0.8952 - val_loss: 8.2647 - val_accuracy: 0.0246
Epoch 23/400
31/31 [=====] - 32s 1s/step - loss: 0.2838 - accuracy: 0.9314 - val_loss: 6.5974 - val_accuracy: 0.0491
Epoch 24/400
31/31 [=====] - 32s 1s/step - loss: 0.2819 - accuracy: 0.9231 - val_loss: 7.7193 - val_accuracy: 0.0424
Epoch 25/400
31/31 [=====] - 32s 1s/step - loss: 0.2607 - accuracy: 0.9325 - val_loss: 6.6000 - val_accuracy: 0.1183
Epoch 26/400
31/31 [=====] - 32s 1s/step - loss: 0.2471 - accuracy: 0.9353 - val_loss: 5.7006 - val_accuracy: 0.1518
Epoch 27/400
31/31 [=====] - 32s 1s/step - loss: 0.2334 - accuracy: 0.9373 - val_loss: 5.6679 - val_accuracy: 0.1830
Epoch 28/400
31/31 [=====] - 32s 1s/step - loss: 0.2431 - accuracy: 0.9280 - val_loss: 5.2927 - val_accuracy: 0.1920
Epoch 29/400
31/31 [=====] - 32s 1s/step - loss: 0.2262 - accuracy: 0.9401 - val_loss: 5.1008 - val_accuracy: 0.2366
Epoch 30/400
31/31 [=====] - 32s 1s/step - loss: 0.2060 - accuracy: 0.9410 - val_loss: 5.3950 - val_accuracy: 0.2143
Epoch 31/400
31/31 [=====] - 32s 1s/step - loss: 0.1982 - accuracy: 0.9356 - val_loss: 6.5190 - val_accuracy: 0.1585
Epoch 32/400
31/31 [=====] - 32s 1s/step - loss: 0.2223 - accuracy: 0.9415 - val_loss: 5.8759 - val_accuracy: 0.2009
Epoch 33/400
31/31 [=====] - 32s 1s/step - loss: 0.1850 - accuracy: 0.9443 - val_loss: 5.5675 - val_accuracy: 0.2076
Epoch 34/400
31/31 [=====] - 32s 1s/step - loss: 0.1874 - accuracy: 0.9465 - val_loss: 5.6272 - val_accuracy: 0.2188
Epoch 35/400
31/31 [=====] - 32s 1s/step - loss: 0.2202 - accuracy: 0.9338 - val_loss: 5.9353 - val_accuracy: 0.1987
Epoch 36/400
31/31 [=====] - 32s 1s/step - loss: 0.2147 - accuracy: 0.9442 - val_loss: 6.0334 - val_accuracy: 0.1830
Epoch 37/400
31/31 [=====] - 32s 1s/step - loss: 0.2092 - accuracy: 0.9382 - val_loss: 6.3359 - val_accuracy: 0.2188

Epoch 38/400
31/31 [=====] - 32s 1s/step - loss: 0.2211 - accuracy:
0.9304 - val_loss: 5.9970 - val_accuracy: 0.2098
Epoch 39/400
31/31 [=====] - 32s 1s/step - loss: 0.1915 - accuracy:
0.9412 - val_loss: 6.3312 - val_accuracy: 0.2143
Epoch 40/400
31/31 [=====] - 32s 1s/step - loss: 0.2200 - accuracy:
0.9326 - val_loss: 6.4238 - val_accuracy: 0.2143
Epoch 41/400
31/31 [=====] - 32s 1s/step - loss: 0.2278 - accuracy:
0.9385 - val_loss: 6.2502 - val_accuracy: 0.2031
Epoch 42/400
31/31 [=====] - 32s 1s/step - loss: 0.2487 - accuracy:
0.9292 - val_loss: 5.7950 - val_accuracy: 0.2522
Epoch 43/400
31/31 [=====] - 32s 1s/step - loss: 0.2104 - accuracy:
0.9415 - val_loss: 5.5785 - val_accuracy: 0.2545
Epoch 44/400
31/31 [=====] - 32s 1s/step - loss: 0.1827 - accuracy:
0.9417 - val_loss: 6.5161 - val_accuracy: 0.2455
Epoch 45/400
31/31 [=====] - 32s 1s/step - loss: 0.1858 - accuracy:
0.9436 - val_loss: 6.1442 - val_accuracy: 0.2031
Epoch 46/400
31/31 [=====] - 32s 1s/step - loss: 0.1728 - accuracy:
0.9513 - val_loss: 6.6339 - val_accuracy: 0.1786
Epoch 47/400
31/31 [=====] - 32s 1s/step - loss: 0.1473 - accuracy:
0.9556 - val_loss: 5.2963 - val_accuracy: 0.2723
Epoch 48/400
31/31 [=====] - 32s 1s/step - loss: 0.1677 - accuracy:
0.9508 - val_loss: 6.0020 - val_accuracy: 0.2433
Epoch 49/400
31/31 [=====] - 32s 1s/step - loss: 0.2041 - accuracy:
0.9449 - val_loss: 5.8678 - val_accuracy: 0.2500
Epoch 50/400
31/31 [=====] - 32s 1s/step - loss: 0.1601 - accuracy:
0.9616 - val_loss: 5.5817 - val_accuracy: 0.2522
Epoch 51/400
31/31 [=====] - 32s 1s/step - loss: 0.1253 - accuracy:
0.9651 - val_loss: 5.5550 - val_accuracy: 0.2656
Epoch 52/400
31/31 [=====] - 32s 1s/step - loss: 0.1480 - accuracy:
0.9604 - val_loss: 5.1319 - val_accuracy: 0.2589
Epoch 53/400
31/31 [=====] - 32s 1s/step - loss: 0.1346 - accuracy:
0.9591 - val_loss: 5.7757 - val_accuracy: 0.2455

Epoch 54/400
31/31 [=====] - 32s 1s/step - loss: 0.1710 - accuracy: 0.9507 - val_loss: 5.8126 - val_accuracy: 0.2612

Epoch 55/400
31/31 [=====] - 32s 1s/step - loss: 0.1060 - accuracy: 0.9709 - val_loss: 5.4918 - val_accuracy: 0.2478

Epoch 56/400
31/31 [=====] - 32s 1s/step - loss: 0.0894 - accuracy: 0.9723 - val_loss: 5.1332 - val_accuracy: 0.2589

Epoch 57/400
31/31 [=====] - 32s 1s/step - loss: 0.0752 - accuracy: 0.9770 - val_loss: 5.1077 - val_accuracy: 0.2679

Epoch 58/400
31/31 [=====] - 32s 1s/step - loss: 0.1364 - accuracy: 0.9625 - val_loss: 5.5382 - val_accuracy: 0.2321

Epoch 59/400
31/31 [=====] - 32s 1s/step - loss: 0.0782 - accuracy: 0.9789 - val_loss: 5.3676 - val_accuracy: 0.2522

Epoch 60/400
31/31 [=====] - 32s 1s/step - loss: 0.0827 - accuracy: 0.9771 - val_loss: 5.4748 - val_accuracy: 0.2835

Epoch 61/400
31/31 [=====] - 32s 1s/step - loss: 0.0941 - accuracy: 0.9738 - val_loss: 5.1669 - val_accuracy: 0.2188

Epoch 62/400
31/31 [=====] - 32s 1s/step - loss: 0.1117 - accuracy: 0.9675 - val_loss: 5.1975 - val_accuracy: 0.2656

Epoch 63/400
31/31 [=====] - 31s 1s/step - loss: 0.0793 - accuracy: 0.9790 - val_loss: 5.4748 - val_accuracy: 0.2768

Epoch 64/400
31/31 [=====] - 32s 1s/step - loss: 0.1675 - accuracy: 0.9548 - val_loss: 6.8398 - val_accuracy: 0.1875

Epoch 65/400
31/31 [=====] - 32s 1s/step - loss: 0.1590 - accuracy: 0.9543 - val_loss: 6.5336 - val_accuracy: 0.2321

Epoch 66/400
31/31 [=====] - 32s 1s/step - loss: 0.1743 - accuracy: 0.9508 - val_loss: 6.0277 - val_accuracy: 0.2522

Epoch 67/400
31/31 [=====] - 33s 1s/step - loss: 0.1331 - accuracy: 0.9639 - val_loss: 5.5973 - val_accuracy: 0.2567

Epoch 68/400
31/31 [=====] - 32s 1s/step - loss: 0.1496 - accuracy: 0.9551 - val_loss: 5.4199 - val_accuracy: 0.2522

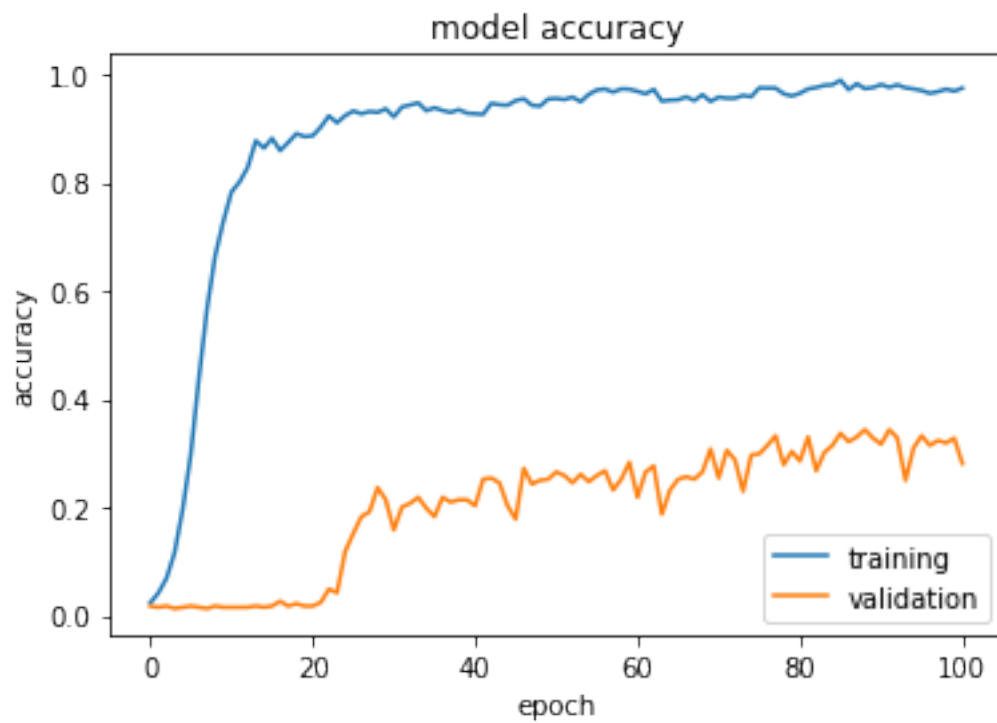
Epoch 69/400
31/31 [=====] - 32s 1s/step - loss: 0.1073 - accuracy: 0.9657 - val_loss: 5.3094 - val_accuracy: 0.2656

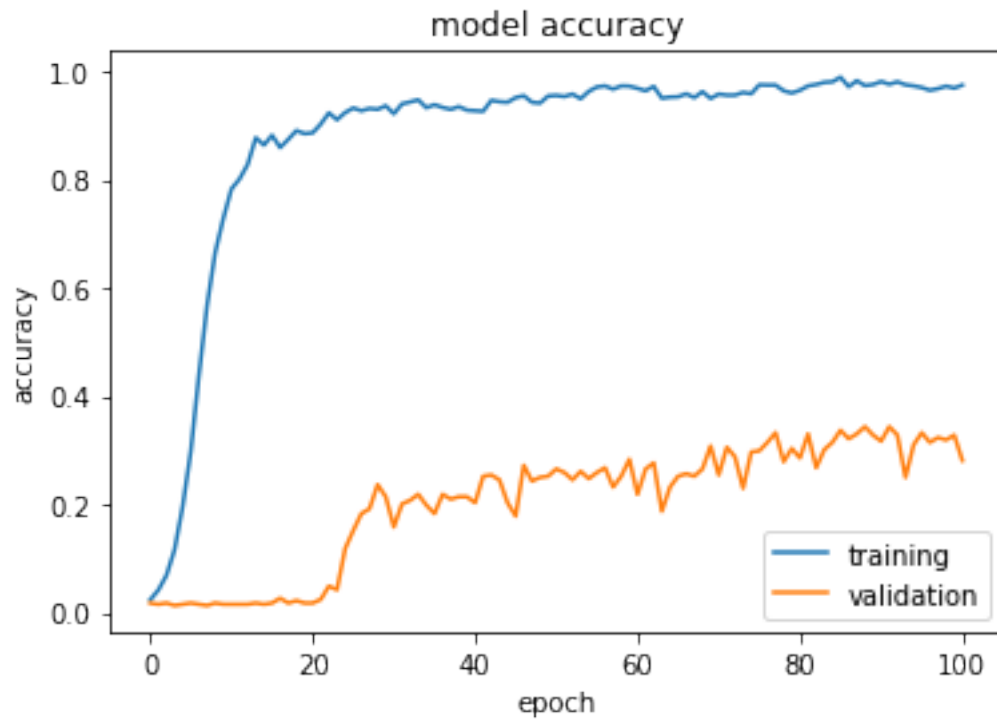
Epoch 70/400
31/31 [=====] - 32s 1s/step - loss: 0.1745 - accuracy: 0.9510 - val_loss: 5.9611 - val_accuracy: 0.3080
Epoch 71/400
31/31 [=====] - 32s 1s/step - loss: 0.1436 - accuracy: 0.9619 - val_loss: 6.7030 - val_accuracy: 0.2545
Epoch 72/400
31/31 [=====] - 32s 1s/step - loss: 0.1382 - accuracy: 0.9589 - val_loss: 5.7207 - val_accuracy: 0.3058
Epoch 73/400
31/31 [=====] - 32s 1s/step - loss: 0.1525 - accuracy: 0.9569 - val_loss: 5.5011 - val_accuracy: 0.2879
Epoch 74/400
31/31 [=====] - 32s 1s/step - loss: 0.1293 - accuracy: 0.9704 - val_loss: 6.1242 - val_accuracy: 0.2299
Epoch 75/400
31/31 [=====] - 32s 1s/step - loss: 0.1456 - accuracy: 0.9605 - val_loss: 5.5948 - val_accuracy: 0.2969
Epoch 76/400
31/31 [=====] - 33s 1s/step - loss: 0.0789 - accuracy: 0.9792 - val_loss: 5.0845 - val_accuracy: 0.2991
Epoch 77/400
31/31 [=====] - 33s 1s/step - loss: 0.1044 - accuracy: 0.9736 - val_loss: 5.4756 - val_accuracy: 0.3147
Epoch 78/400
31/31 [=====] - 32s 1s/step - loss: 0.0819 - accuracy: 0.9782 - val_loss: 5.4632 - val_accuracy: 0.3326
Epoch 79/400
31/31 [=====] - 32s 1s/step - loss: 0.1429 - accuracy: 0.9582 - val_loss: 5.0367 - val_accuracy: 0.2790
Epoch 80/400
31/31 [=====] - 32s 1s/step - loss: 0.1364 - accuracy: 0.9656 - val_loss: 5.1341 - val_accuracy: 0.3036
Epoch 81/400
31/31 [=====] - 32s 1s/step - loss: 0.0854 - accuracy: 0.9695 - val_loss: 5.0010 - val_accuracy: 0.2857
Epoch 82/400
31/31 [=====] - 32s 1s/step - loss: 0.0764 - accuracy: 0.9776 - val_loss: 4.7405 - val_accuracy: 0.3304
Epoch 83/400
31/31 [=====] - 32s 1s/step - loss: 0.0855 - accuracy: 0.9755 - val_loss: 5.2778 - val_accuracy: 0.2679
Epoch 84/400
31/31 [=====] - 32s 1s/step - loss: 0.0981 - accuracy: 0.9753 - val_loss: 5.1260 - val_accuracy: 0.3013
Epoch 85/400
31/31 [=====] - 32s 1s/step - loss: 0.0479 - accuracy: 0.9835 - val_loss: 4.8906 - val_accuracy: 0.3147

Epoch 86/400
31/31 [=====] - 31s 1s/step - loss: 0.0572 - accuracy: 0.9886 - val_loss: 4.5963 - val_accuracy: 0.3371
Epoch 87/400
31/31 [=====] - 31s 1s/step - loss: 0.0698 - accuracy: 0.9780 - val_loss: 4.7077 - val_accuracy: 0.3214
Epoch 88/400
31/31 [=====] - 31s 1s/step - loss: 0.0604 - accuracy: 0.9848 - val_loss: 4.8737 - val_accuracy: 0.3304
Epoch 89/400
31/31 [=====] - 31s 1s/step - loss: 0.0778 - accuracy: 0.9751 - val_loss: 4.8525 - val_accuracy: 0.3438
Epoch 90/400
31/31 [=====] - 32s 1s/step - loss: 0.0640 - accuracy: 0.9795 - val_loss: 4.9865 - val_accuracy: 0.3281
Epoch 91/400
31/31 [=====] - 32s 1s/step - loss: 0.0600 - accuracy: 0.9865 - val_loss: 4.9824 - val_accuracy: 0.3170
Epoch 92/400
31/31 [=====] - 32s 1s/step - loss: 0.0794 - accuracy: 0.9811 - val_loss: 5.0791 - val_accuracy: 0.3438
Epoch 93/400
31/31 [=====] - 31s 1s/step - loss: 0.0570 - accuracy: 0.9834 - val_loss: 5.2380 - val_accuracy: 0.3281
Epoch 94/400
31/31 [=====] - 31s 1s/step - loss: 0.1120 - accuracy: 0.9740 - val_loss: 6.1203 - val_accuracy: 0.2500
Epoch 95/400
31/31 [=====] - 32s 1s/step - loss: 0.0696 - accuracy: 0.9751 - val_loss: 5.0995 - val_accuracy: 0.3103
Epoch 96/400
31/31 [=====] - 32s 1s/step - loss: 0.0898 - accuracy: 0.9723 - val_loss: 5.0952 - val_accuracy: 0.3326
Epoch 97/400
31/31 [=====] - 32s 1s/step - loss: 0.1033 - accuracy: 0.9724 - val_loss: 5.6316 - val_accuracy: 0.3147
Epoch 98/400
31/31 [=====] - 32s 1s/step - loss: 0.0979 - accuracy: 0.9698 - val_loss: 5.4321 - val_accuracy: 0.3237
Epoch 99/400
31/31 [=====] - 32s 1s/step - loss: 0.0772 - accuracy: 0.9730 - val_loss: 5.2021 - val_accuracy: 0.3192
Epoch 100/400
31/31 [=====] - 32s 1s/step - loss: 0.0845 - accuracy: 0.9761 - val_loss: 5.2520 - val_accuracy: 0.3281
Epoch 101/400
31/31 [=====] - 32s 1s/step - loss: 0.0814 - accuracy: 0.9800 - val_loss: 5.7342 - val_accuracy: 0.2812

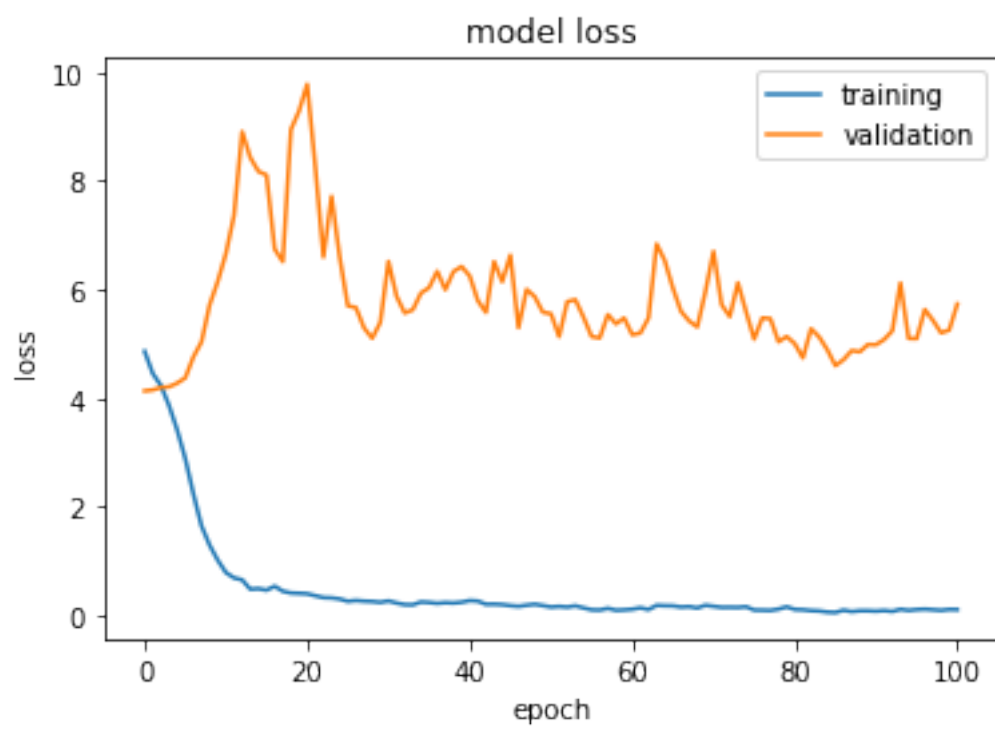
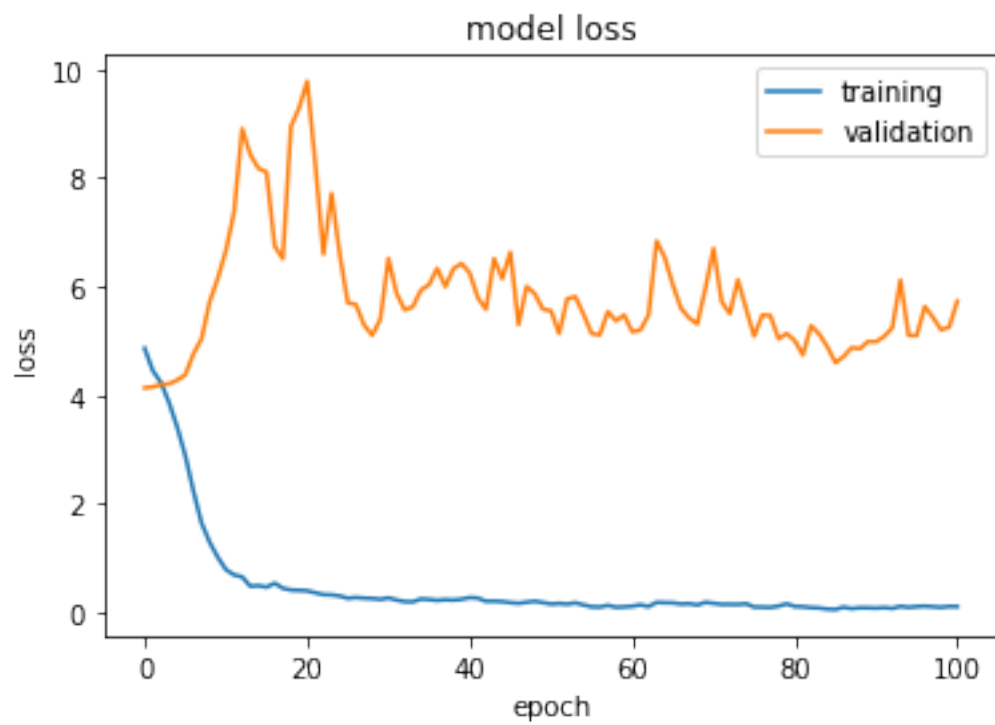
Restoring model weights from the end of the best epoch.
Epoch 00101: early stopping

```
[ ]: plt.plot(history6.history['accuracy'])  
plt.plot(history6.history['val_accuracy'])  
plt.title('model accuracy')  
plt.ylabel('accuracy')  
plt.xlabel('epoch')  
plt.legend(['training', 'validation'], loc='best')  
plt.show()
```





```
[ ]: plt.plot(history6.history['loss'])
plt.plot(history6.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()
```



As expected, this does not converge. Infact, it's only half as good. This accuracy result is similar to the one shown in ref. [6] [here](#).

1.4 Observations form these experiments

- I downscaled the images to the dimensions of that of the MNIST Images while making sure that the images are still recognizable from each other. This would reduce the number of parameters in our network. I also padded the images with 0 to maintain the aspect ratio and inverted the images so that the background now has 0 as the pixel value.
- I started with a modified version of the original LeNet, which was used to classify on the MNIST dataset, as our current dataset closely resembles it, I got an accuracy of around 50%.
- Data Augmentation did not help for this dataset as the augmented data was very different from the original data and it actually performed worse. I did not use Data augmentation for later experiments.
- I tried the Mish Activation function instead of ReLU and it reached the the same accuracy in lesser epochs, however it had some variance issues for the training accuracy in the later epochs.
- Using Softmax with greater temperature, which spreads the probabilities and it actually gave similar results in lesser epochs.
- I tried different architectures: One which is known to give pretty good accuracy on the MNIST dataset in Kaggle Competitions and one Modified EfficientNet Architecture.
 - The first of them gave very good results compared to the original LeNet and in lesser epoch. **It achieved an accuracy of 71.43%!** An improvement of more than 20% on the original experiment.
 - The EfficientNet architecture actually gave the worst results in all of the experiments, but I think this was due to the scarcity of training data and not using a pretrained model.

1.5 Conclusions and Future Improvements

- Overall, I think the Experiment 5 gave very good result, with around 71.43% accuracy, followed by LeNet with Mish and higher temperature Softmax. Clearly, the models were overfitting the data and I think that if trained with more data, they would perform better. For the future parts of this task, I would be using these two architectures.
- EfficientNet can give better results than from Experiment 6 but again, it needs more data and some pre-trained weights. EfficientNetV2, released recently can also be a good model, but I can't find an Open Source Implementation of it and it's a little complex to build it in this duration, so I didn't try it, but I'm sure that it would give better results than what I got from EfficientNet here.
- I have tuned the hyperparameters from what I could read online. However, I think using [Weights & Biases Sweeps](#) would give better insights. I got to know about this relatively late and there was not much time left to set it up. But it would be sweet in the future to improve the model using this.

2 References

- [1] [EMNIST handwritten character recognition with Deep Learning](#)
- [2] [How to choose CNN Architecture MNIST](#)

- [3] [Swish Vs Mish: Latest Activation Functions](#)
- [4] [Mish Class Definition in Keras](#)
- [5] [Mnist_EfficientNet Kaggle Notebook](#)
- [6] [Image classification via fine-tuning with EfficientNet](#)
- [7] [Keras EfficientNet Implementation Source Code](#)
- [8] [Keras ModelCheckpoint Documentaion](#)
- [9] [Keras EarlyStopping Documentaion](#)
- [10] [Keras ImageDataGenerator Documentation](#)
- [11] [Keras EfficientNetB0 Documentation](#)
- [12] [Effect of Batch Size on Neural Net Training](#)
- [13] [How does temperature affect softmax in machine learning?](#)
- [14] [Temperature softmax · Issue #3092 · keras-team/keras](#)
- [15] [EfficientNetV2: Smaller Models and Faster Training](#)
- [16] [Notes from Deeplearning.ai's Deep Learning Specialization](#)

[]: