Image Classification using modified Fashion MNIST dataset

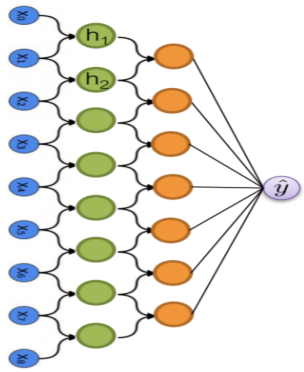
Abstract

This mini project involves classification of images using machine learning models. The dataset used for this project is a modification of the benchmark Fashion MNIST dataset. Deep Neural Network architectures involving Convolutional Neural Networks are implemented for this task. The best model, a modified Residual Network, is able to achieve an accuracy of 95.7% over 30% of the training data split and a 96.5% accuracy over a subset of the final test dataset (on Kaggle).

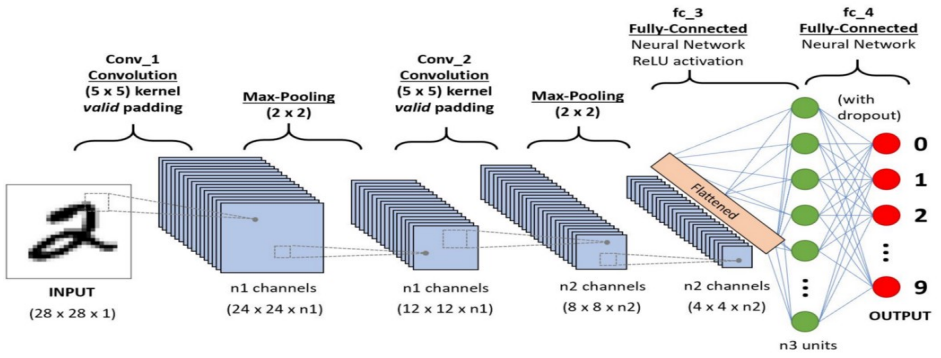
1 Introduction

Computer vision tasks like image classification, object detection and image recognition are gaining a lot of attention from researchers in the field of Machine Learning and Artificial Intelligence. Convolutional Neural Networks (CNNs) have broken the mold and ascended to become the state-of-the-art computer vision technique. Among the different types of neural networks, CNNs are easily the most popular.

Image Classification involves comprehending an entire image and assigning it to a class/label or outputting the probability of it belonging to certain predefined classes. For such a task, the best-suited approach would be to deploy a CNN where each neuron in the network is connected to only a patch of neurons (as shown in Fig. 1a) in the previous layer with the entire layer sharing matrix of parameters, weights (W) and bias term (b), in turn excelling at analyzing images using much lesser parameters in comparison with the Multi-layer Perceptron (MLP) of the same depth (hidden layers).



(a) connections in a typical CNN



(b) Architecture of a typical CNN

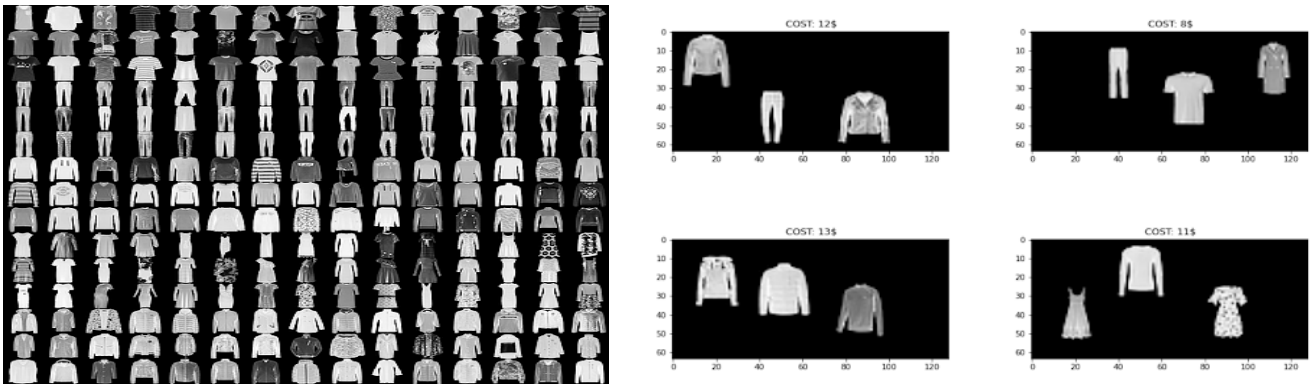
A CNN applies convolution to the data using 2D or 3D convolution layers which makes that this type of network ideal for processing of images. CNNs do not require a lot of data preprocessing as the model eventually learns the filters they need to accomplish the given task during training. In comparison, a fully connected neural network (or MLP) is prone to overfitting to the data and have to deal with a larger number of parameters thus impacting run-time performance.

Fig. 4b is a pictorial representation of a CNN model used for identification of handwritten digits, built over the famous ImageNet Digits dataset.

2 Dataset

The Fashion MNIST dataset, as shown in Fig. 2a, is a drop-in replacement for the very well known machine learning MNIST dataset, Digits. Each image in this dataset belongs to one of the ten clothing item categories. The dataset provided for this project comprises of 60000, 64 x 128 pixel gray-scale (channel = 1) images, as shown in Fig. 2b, with each image comprising of

multiple items from the Fashion MNIST dataset. Interestingly, only 5 items are used from this Fashion MNIST dataset to form the images for our dataset and each item has a cost associated with it, namely: T-shirt/top ($1), Trouser ($2), Pullover($3), Dress ($4), Coat($5). The images are modified in a way that the total price of the items in them can range from $5 to $13 and the class/label associated with each image is the total cost of the items present in the image.



(a) Examples of images in Fashion MNIST dataset

(b) Examples of images in modified Fashion MNIST dataset

Figure 2: A sneak peak in the benchmark and modified dataset

Studying the dataset shows that it has an even distribution of data with respect to the possible target labels as shown in Fig. 6 in Appendix B.

2.1 Image Preprocessing

2.1.1 Image Flipping & Random Rotation

To include randomness within the training set and avoid possible overfitting on the training set, horizontal flipping and random rotations of images were tested on ImageNet ResNet models. However, when both were introduced together in the model, it caused the model to under-fit. When either of them were used in the models, it was observed that it did not produce major variations in the result. Therefore, in our final model, both the image processing techniques were omitted.

2.1.2 Image Resizing and Cropping

Majority of the proposed CNN models in literature are optimised for square images which makes it difficult for us to directly apply them on our dataset, given that our images are rectangular in shape. An attempt to resize the images was made to fit the requirements of input image sizes for the classical CNN models suggested in literature like AlexNet, VGG-16, LeNet, etc. However, on re-scaling the dimensions of the image, the clarity of the image was compromised given that the items are already much smaller in size in comparison to the entire image. On increasing the dimension to 224 x 224 x 1, as required by the classical ResNet50/152 and VGG-16 networks, the GPU on the Google Colab would end up running out of memory and crash. Additionally, since the location of the clothing items in the images does not follow a particular format, it becomes difficult to apply image cropping. Hence, image resizing did not prove to be a strong preprocessing step and we ended up not using it in the final CNN model.

3 Proposed Approach

For this image classification problem, testing began with with various ImageNet models such as ResNet 50/152 and VGG16 [1]. After acquiring a fair intuition of the performance of different models, the final model was built. We recorded the best-fitting test models in Table 1. Please note that none of the networks were pre-trained Keras Application networks, but were replicated from scratch.

To begin with, 70% of dataset was tested on LeNeT-7 [2]. However, the accuracy (11%) was abysmal, which could be due to the simple 7 sequential convolution layers. Therefore, it was decided to increase number of layers gradually and then observe accuracy, as noted in Table 1. In addition to LeNet-7, we tested VGG-16 (16 layers), ResNet-34 (34 layers), ResNet-50 (50 layers) and finally ResNet-152 (152 layers).

Accuracy was not the only criteria for ranking different models; other considerations were the total number of trainable parameters and the time taken to train per epoch. Based on these considerations, VGG-16 was immediately ruled, because of its large

number of parameters and high computation time per epoch. Additionally, we were able to record only 88.32% accuracy on it. In addition to this, with the increasing accuracy achieved by ResNet-34 in just 5 epochs, with 27M+ trainable parameters; it was decided to pursue the ResNet architecture further.

Although VGG-16 was not chosen due to its size, it is noted that it did give favourable results on the validation set. The smallest ResNet model tested here has 34 layers and the highest ResNet model has 152 layers. These models have a difference of about 26M+ parameters between them, while the accuracy of both these models, as observed, did not vary greatly. Therefore, bias shifted towards lower number of ResNet layers, favouring lesser trainable parameters and faster training. After building two random ResNet models, each of 15 and 27 layers, we noticed that performance of both were comparable. Therefore, we opted towards towards a lower layer model, i.e. 15 layers for our final model.

3.1 Modified ResNet - The Final Model

As claimed by Veit et al., Residual Networks (or ResNet) essentially works as an ensemble of relatively shallow networks [3]. ResNets learn residual functions with reference to the layer inputs, instead of learning unreferenced functions. Instead of hoping each few stacked layers directly fit a desired underlying mapping, residual nets let these layers fit a residual mapping. They stack residual blocks on top of each other to form network: e.g. a ResNet-50 has fifty layers using these blocks. Formally, denoting the desired underlying mapping as $H(x)$, the stacked nonlinear layers fit another mapping $f(x) := H(x) - x$. The original mapping is recast into $f(x) + x$. There is empirical evidence that these types of network are easier to optimize, and can gain accuracy from considerably increased depth [4]. Fig. 7 (in Appendix B) shows the overview of a generic residual block used in ResNet architecture. These residual blocks are the spotlight element of ResNets. Fig. 3 gives a top-level view of the modified ResNet that we picked for the final Kaggle model. The output of each residual block is added to the output of the next block and fed as input to the next block. The CONV1 and CONV2 layers forms the basic block of the structure while CONV 3, 4, 5 and 6 are the residual blocks followed by the CONV7 layer. Global Average Pooling is then applied to the output of CONV7 layer and then flattened to support a fully connected layers of 256 neurons followed by the output layer having nine classes/labels.

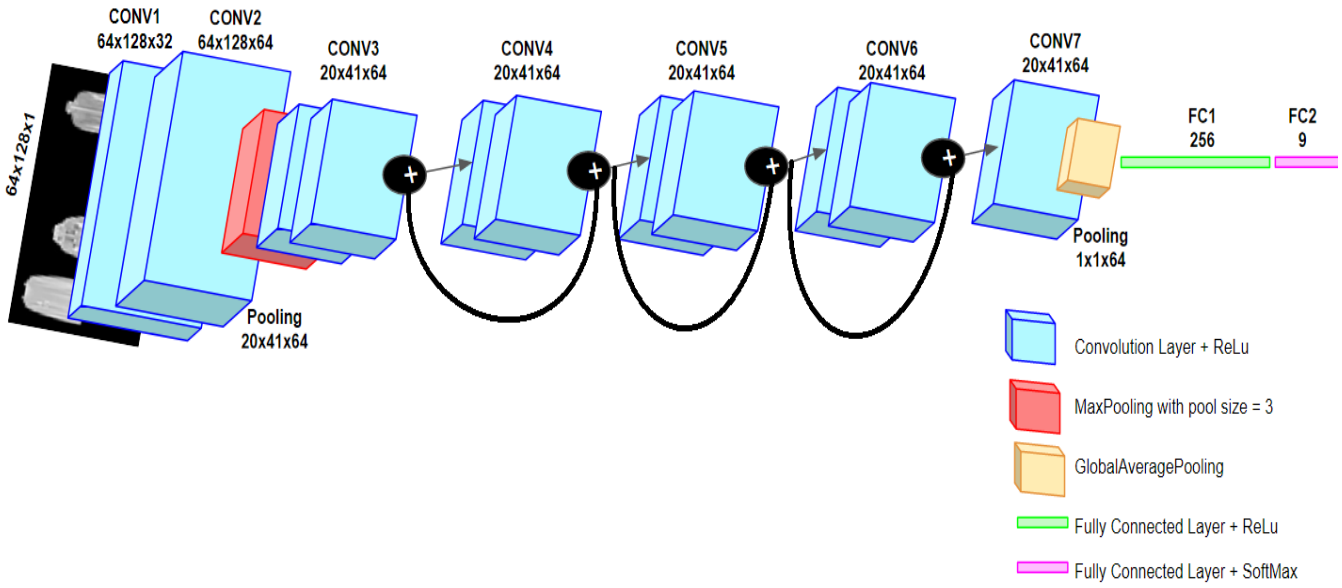


Figure 3: Modified ResNet Architecture

4 Results

Table 1 shows the performance of various architectures on our problem statement. Our final Modified Resnet model provides a good balance between runtime performance and accuracy. Three types of optimizers were used for training of the final network: SGD, Adam and RMSProp. Based on their accuracy as listed in Table 2, RMSProp was selected.

Table 1: Comparison of Number of parameters and run time per epoch on different models

Model	**# Parameters**	**Run-time per epoch (GPU enabled)**	**Accuracy on Validation set (30% Dataset)**	**Accuracy on KaggleBoard**
LeNet	0.44M+	∼1 min (Batch size = 512)	35.33% (10 epochs only)	Untested
VGG-16	137M+	∼45-60 min (Batch size = 64) (learning rate = 1e-2)	88.32% (5 epochs only)	Untested
ResNet-34	21M+	∼5-7 min (Batch size = 128) (learning rate = 1e-4)	83.72% (5 epochs only)	Untested
ResNet-50	23M+	∼7-10 min (Batch size = 100) (learning rate = 1e-4)	81.21% (8 epochs)	80.16%
ResNet-152	58M+	∼15-30 min (Batch size = 100) (learning rate = 1e-3)	81.26% (10 epochs only)	81.866%
Modified ResNet (15 layers)	0.37M+	∼ 26 sec (Batch size = 64) (learning rate = 1e-3)	95.7% (30 epochs)	96.50%

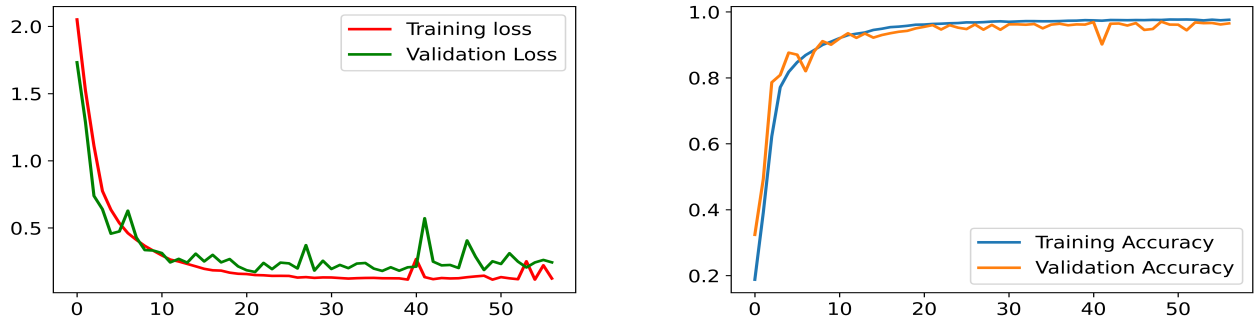
Table 2: Comparison of different optimizers on Modified ResNet (15 Layers)

Model	**Optimizer**	**Accuracy on validation set (Epoch 20)**
Modified ResNet	SGD	12.52%
Modified ResNet	Adam	92.01%
Modified ResNet	RMSProp	94.81%

After selecting RMSProp as our optimizer, we tested different learning rates to find an optimal one. Best results were produced with learning rate of $1e-3$ as it gave higher accuracy on validation set with lesser number of epochs (in comparison with the learning rate of 1e-4)

Table 3: Comparison of different learning rates on Modified ResNet (15 Layers) with RMSProp

Optimizer	**Learning rate**	**Number of Epochs**	**Accuracy on validation set**
RMSProp	1e-2	13 (Early stopping applied after epoch 13)	11.19%
RMSProp	1e-3	30	95.71%
RMSProp	1e-4	59	93.12%



(a) Loss vs Epoch

(b) Accuracy vs Epoch

Figure 4: Relation between Loss and Accuracy with respect to number of epochs

5 Discussion and Conclusion

While carrying out rigorous testing on the given dataset, it was observed that the number of layers used in Convolutional Neural Network have an approximate direct relation with the accuracy on validation set *(*30% *dataset)*. However, while the probability of observing higher accuracy increases with an increase in number of layers, the total number of trainable parameters also increase which creates a higher demand for RAM. While training our model with *(higher)* number of layers on Google colab, RAM shortage was often faced and eventually resulting in crashes, especially in VGG-16 which has $137M+$ trainable parameters. Since all models were built in Google Colab which provides about 13 GB of RAM, reduced batch sizes were used to handle limited RAM. For example, as observed in Table 1, LeNeT which has significantly lower number of layers could handle a batch size of 512 as opposed to batch size of 64 in VGG-16.

An increase in number of parameters resulted in two bottlenecks, higher RAM requirements and increased training times per epoch. Any model with number of parameters less than 1 million can be trained in about a minute, while models with 20M+ parameters take more than 5 minutes per epoch to train. It was observed that the number of neurons in the fully connected layer have to be at least equal to or greater than the size of the image after flattening. Even though it did not have any significant impact on the performance, it was a good point to be kept in mind for future references.

Overfitting, a common problem in neural networks, was also observed in the models we tested. The models would give higher performance while training, but fail considerably on the test set. Overfitting was observed when we allowed the model to run for higher number of epochs. An example is given in table 3. To combat this issue, dropout layers were added and introduced early stopping at *(lesser number of epoch)* when the performance over validation set did not improve over time.

Table 4: An example of observed case of Over-fitting

Model	**Accuracy on Train Set**	**Accuracy on Kaggleboard**
ResNet-50	97.21% (20 epoch)	80.16%
ResNet-152	98.32% (15 epochs)	81.86%

Additionally, two loss functions were tried: Cross Entropy (CE) and Negative Log-Likelihood (NLL). A Sigmoid output layer combined with CE and a softmax output layer with NLL behave in a similar fashion. However, since softmax output layer has K outputs/classes whose sum equals to one, there exists a constraint in flexibility in the network in comparison to sigmoid output. Sigmoid output has K different sigmoid outputs whose summation does not have to be equal to 1. As observed slightly better results when using CE loss, it was selected as the final loss function.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

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

[3] A. Veit, M. J. Wilber, and S. J. Belongie, "Residual networks are exponential ensembles of relatively shallow networks," *CoRR*, vol. abs/1605.06431, 2016. [Online]. Available: http://arxiv.org/abs/1605.06431

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

A Appendix: Modified ResNet Code

```
# -*- coding: utf-8 -*-
"""KaggleModel.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1sCQOn3ixACqBdrI5AQ3e5Zgr1b7R7zKF
"""

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

import pickle
import numpy as np
from skimage import transform
import tensorflow as tf
from tensorflow import keras
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, Dropout, MaxPooling2D
from tensorflow.keras.layers import experimental, Activation, InputLayer
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import KFold
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/drive')

# Read a pickle file and disply its samples
# Note that image data are stored as unit8 so each element is an int from 0-255
data = pickle.load( open( '/content/drive/MyDrive/Colab Notebooks/MP3/Train.pkl',
                          'rb' ), encoding='bytes')

test_data=pickle.load(open( '/content/drive/MyDrive/Colab Notebooks/MP3/Test.pkl',
                            'rb' ), encoding='bytes')
targets=np.genfromtxt('/content/drive/MyDrive/Colab Notebooks/MP3/TrainLabels.csv',
                      delimiter=',', skip_header=1)[:,1:]
#plt.imshow(data[1234,:,:],cmap='gray', vmin=0, vmax=256)
print(data.shape, targets.shape)

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
le =LabelEncoder()
labels=le.fit_transform(targets.flatten())
labels=to_categorical(labels,num_classes=9)
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.3,
                                                    random_state=42)

x_test = x_test.reshape((x_test.shape[0], 64, 128, 1)).astype("float32")/255
test_data=test_data.reshape((test_data.shape[0],64,128,1)).astype("float32")/255
x_train= x_train.reshape((x_train.shape[0], 64, 128, 1)).astype("float32")/255
```

```
"""**Run the following code only when training set is NOT to be split**"""

x_train = data
y_train = labels

from sklearn.utils import shuffle
x_train, y_train = shuffle(x_train, y_train)

x_train= x_train.reshape((x_train.shape[0], 64, 128, 1)).astype("float32")/255

print(x_train.shape, y_train.shape)
print(test_data.shape)

"""**Modified ResNet**"""

inputs = keras.Input(shape=(64,128, 1), name="img")
x = layers.Conv2D(32, 3, activation="relu")(inputs)
x = layers.Conv2D(64, 3, activation="relu")(x)
block_1_output = layers.MaxPooling2D(3)(x)

x = layers.Conv2D(64, 3, activation="relu", padding="same")(block_1_output)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
block_2_output = layers.add([x, block_1_output])

x = layers.Conv2D(64, 3, activation="relu", padding="same")(block_2_output)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
block_3_output = layers.add([x, block_2_output])

x = layers.Conv2D(64, 3, activation="relu", padding="same")(block_3_output)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
block_4_output = layers.add([x, block_3_output])

x = layers.Conv2D(64, 3, activation="relu", padding="same")(block_4_output)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
block_5_output = layers.add([x, block_4_output])

x = layers.Conv2D(64, 3, activation="relu")(block_5_output)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(9)(x)

model = keras.Model(inputs, outputs, name="modified_resnet")
model.summary()

model.compile(
    optimizer=keras.optimizers.RMSprop(1e-3),
    loss=keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=["acc"],
)
early=EarlyStopping(monitor='val_acc',min_delta=0,patience=10,verbose=1,
                    mode='auto')
history = model.fit(x_train, y_train, batch_size=64, epochs=200,
                    validation_split=0.2,callbacks=[early])

model_loss = pd.DataFrame(history.history)
```

```
plt.rcParams.update({'font.size': 13})
plt.plot(model_loss['loss'][:-2],'r',linewidth=2.0)
plt.plot(model_loss['val_loss'][:-2],'g',linewidth=2.0)
plt.legend(['Training loss','Validation Loss'])
plt.savefig('Loss.png',dpi=600)

plt.plot(model_loss['acc'][:-2],linewidth=2.0)
plt.plot(model_loss['val_acc'][:-2],linewidth=2.0)
plt.legend(['Training Accuracy','Validation Accuracy'])
plt.savefig('accuracy.png',dpi=600)

test_scores = model.evaluate(x_test, y_test, verbose=2)
print("Test loss:", test_scores[0])
print("Test accuracy:", test_scores[1])

test_data_pred = model.predict(test_data)

import numpy as np

test_prediction = []

for y_i in test_data_pred:
  test_prediction.append(np.argmax(y_i) + 5)

import pandas as pd

df = pd.DataFrame(columns=['class'])

df['class'] = test_prediction

df.to_csv('ResNet.csv', index=True)

from google.colab import files
files.download('ResNet.csv')

keras.utils.plot_model(model, "mini_resnet.png", show_shapes=True)
```

B Appendix: Miscellaneous

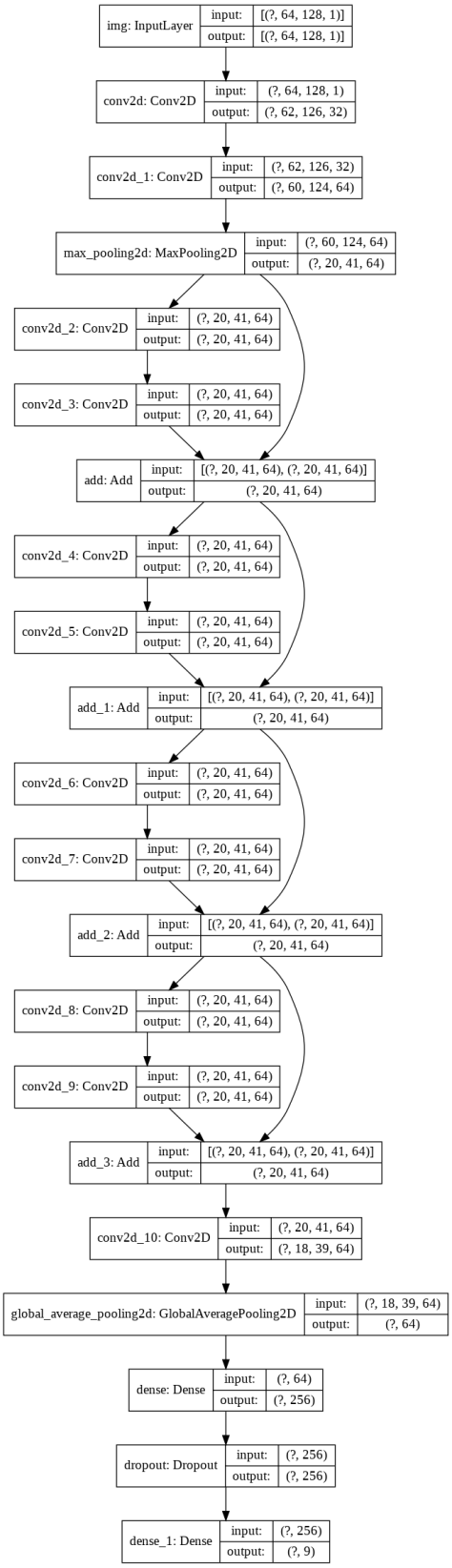


Figure 5: Final model architecture

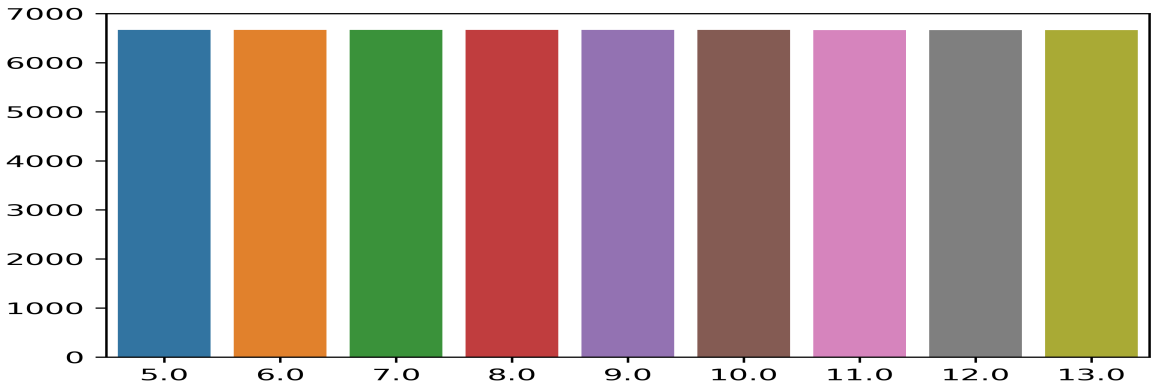


Figure 6: Distribution of target labels

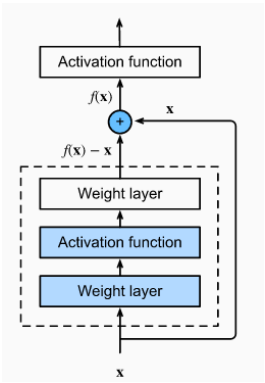


Figure 7: A generic residual block in ResNet