Electronic Component Neural Network Image Classifier

Jasmine Khalil

1 Introduction

In this paper, I discuss the steps I took to build a multi-class image classifier that identifies which of five classes an electronic component belongs to. Starting off by organizing the dataset into train, test, and validation sets, re-scaling and manipulating these sets, completing one-hot-encoding to compute probabilities, constructing the Keras model, and training and testing.

1.1 Classifiers

An image classifier will comprehend an entire image and return if the specified object is present. Object detection will do the same but also identify where the object is located in the image. It can do this for several objects in an image.

Neural networks consist of layers made up of one or more neurons that perform mathematical operations on the data from the first input layer. Following are a series of hidden layers where the outputs of one layer become the inputs of the next. Lastly, there is a final output layer where, in a binary classifier, the final layer will be a single node outputting the probability that the image belongs to a certain class - the output is either a 0 or a 1.

Besides neural networks, there are other classifiers like support vector machines. An SVM finds the best line or plane separating two classes, increases the number of dimensions to find a clear separation of the different classes, and then, during classification, checks which side of the line/plane the input lies on. Support vector machines work well with large-dimension inputs, but their effectiveness decreases when the number of training samples is less than the number of input dimensions of each sample (width * height * number of color channels). That said, the majority of classifiers are outperformed by neural networks.

Neural networks, however, also suffer from various challenges during classification if they are not robust enough. Potential issues include lighting, deformation, and occlusion of the input image. Another issue may be having both classes in the same input image, but this can be solved with a sliding window.

1.2 Neural Network

In supervised neural network models, we create a dataset that is randomly split into training, validation, and testing sets. The training data is in the form (X, Y) where X are features, which are the pixels that form an image, and Y are the labels specifying what is in each image. The validation set is used before the testing set to measure the performance of the model and helps in adjusting the model's hyperparameters. Firstly, a weighted sum of our inputs is calculated. If this is the neuron in the first layer, the input will be the features of the image for classification. During the training of the model, the weights and bias terms make up the parameters that will be adjusted; therefore, the weights are initially randomized. The final output is then transformed through an activation function that non-linearizes the output. Activation functions like sigmoid and hyperbolic tangent constrain the output to certain values. ReLU, the most common activation function, converts all negative output values to 0 and leaves positive output values as they are.

1.3 Confusion Matrix

The confusion matrix helps assess the performance of a model by visualizing its accuracy. It does this by adding 1 to a cell when that prediction is made. The following confusion matrix for classifying a 'Dog' displays the prediction labels across the columns and the actual labels down the rows.

- True Positives: Predictions that match the actual class.
- True Negative: Predictions that match a negative class.
- False Positive: Predictions that match positive class but the actual label is not positive.
- False Negative: Predictions do not match actual class.

	Cat	Dog	Fish	Mouse
Cat	200	6	19	4
Dog	0	197	0	8
Fish	22	6	199	3
Mouse	4	7	0	203

2 Importing Dependencies

To start, I import the dependencies required to run my neural network:

- **NumPy** library for matrix operations
- **Matplotlib** library for plotting features
- **PIL** for additional image processing ability
- **Utils** for **Keras**. I use this specifically for one-hot encoding in this project
- **Sequential** model from Keras
- **Dense, Dropout, Activation** as the different layer types
- **Resize** from **scikit-image** image processing package for scaling and resizing image arrays
- **Confusion Matrix** from **scikit-learn** to build a confusion matrix

3 Loading Images

I start by setting the target resolution of each input image to `28X28` pixels and splitting my dataset into 20% for the validation set, 20% for the test set and the remaining 60% for the training set.

By looping through the directories in the dataset, I appended the name of each classification folder to an empty list called **labels**. Within this for loop, I added another loop forming a nested loop that iterates through each image in each folder, opens each image, and converts them to gray scale using the **PIL**. Once in gray scale, I converted each image into a NumPy array and appended each array to an empty list called **X_all**, a 3D array `n x width x height` ultimately comprising of all the images making up the dataset, and the length of `X_all` is the number of samples:

```
num_samples = len(X_all)
```

By also appending each label to an empty list **y_all**, which is a 1D vector of the labels corresponding to each image. At this point, y_all consists of a list of the string names that each of the images belong to. I then created another empty list **y_out**, and appended the index of each label, like 'resistors' is indexed at 4 in the 'labels' list.

4 Organizing Samples

Having the samples and their associated labels in order will result in a useless training set. To tackle this, I shuffled the samples and labels. Starting by zipping each image to its corresponding label first so they are not shuffled from each other, then using the Shuffle method in: **random.shuffle()** to complete the shuffling of the features.

Now, we split the samples into testing, training, and validation sets by:

```
num_samples_test = int(test_ratio * num_samples)
num_samples_val = int(val_ratio * num_samples)


X_test = X_all[:num_samples_test]
y_test = y_all[:num_samples_test]
```

And the same steps are repeated for the validation and training sets where for the training set, we use the remaining samples:

```
X_train = X_all[(num_samples_test + num_samples_val):]
y_train = y_all[(num_samples_test + num_samples_val):]
```

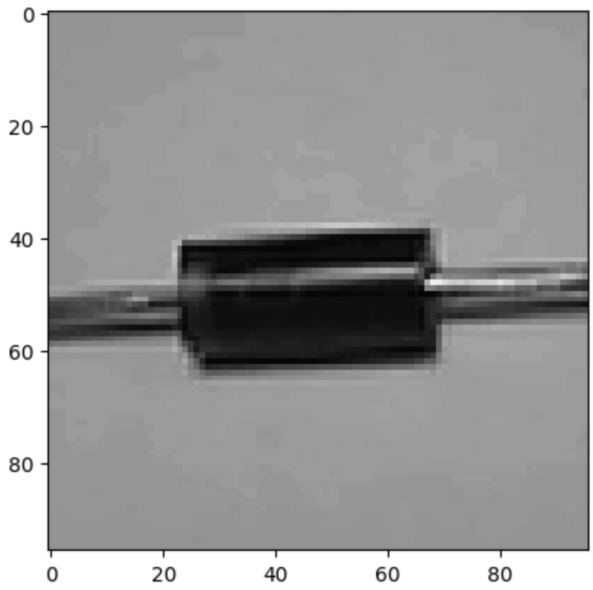
My dataset consisted of 250 samples, `num_samples = 250`, so the split is 50 test samples, 50 validation samples, and 150 training samples.

5 Re-scaling and Plotting

Using a random index, for example `idx = 0`, I then plot the image from my training set at that index using Matplotlib:

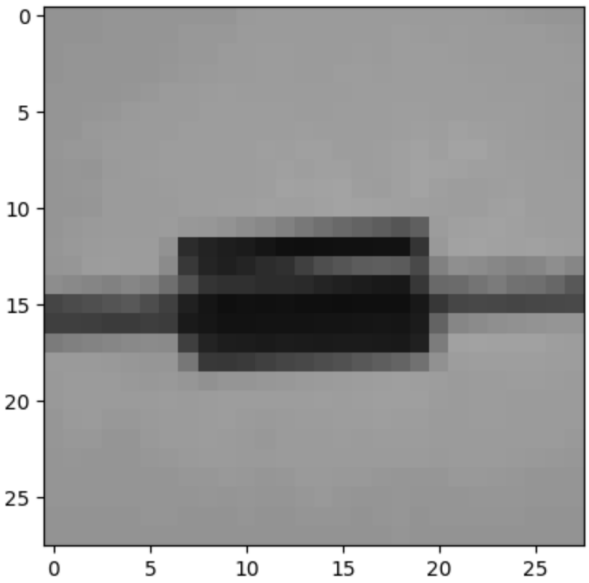
```
plt.imshow(X_train[idx], cmap = 'gray', vmin=0, vmax=255)
```

Producing the following plot of a diode from my training set at index 0:



Next, I resize all the images in each set and display the same image from the training set after the rescale:

```
X_train = resize_images(X_train, target_width, target_height)
X_val = resize_images(X_val, target_width, target_height)
X_test = resize_images(X_test, target_width, target_height)
```



6 Dataset Manipulation

In the first step of this dataset manipulation, I convert the lists of samples and labels into NumPy arrays. For example, converting the training set into a NumPy array:

```
X_train = np.asarray(X_train)
y_train = np.asarray(y_train)
```

resulting in `X_train.shape = (150, 28, 28)` and `y_train.shape = (150,)` The next step consists of flattening each image into a 1D vector of length `target_width * target_height`. So for each of the images in each set this looks like:

```
X_train = X_train.reshape(num_samples_train, len_vector)
X_val = X_val.reshape(num_samples_val, len_vector)
X_test = X_test.reshape(num_samples_test, len_vector)
```

So the new shapes of each set are:

```
X_train: (150, 784)
X_val: (50, 784)
X_test: (50, 784)
```

7 One-Hot Encoding of Labels

The one hot encoding takes our indexes for our labels: 0=background, 1=capacitors, 2=diodes, 3=leds, 4=resistors and outputs the probability that the label is part of each class. Using Keras's np_utils to create one-hot encoding:

```
Y_train = utils.to_categorical(y_train, num_classes)
Y_val = utils.to_categorical(y_val, num_classes)
Y_test = utils.to_categorical(y_test, num_classes)
```

Where the number of classes is the length of the list 'labels' from earlier. To test this, I then print out the result from 10 labels from my training set:

```
Y train: (150, 5)
Y val: (50, 5)
Y test: (50, 5)
Label: 2 | One-hot: [0. 0. 1. 0. 0.]
Label: 2 | One-hot: [0. 0. 1. 0. 0.]
Label: 1 | One-hot: [0. 1. 0. 0. 0.]
Label: 1 | One-hot: [0. 1. 0. 0. 0.]
Label: 1 | One-hot: [0. 1. 0. 0. 0.]
Label: 4 | One-hot: [0. 0. 0. 0. 1.]
Label: 2 | One-hot: [0. 0. 1. 0. 0.]
Label: 0 | One-hot: [1. 0. 0. 0. 0.]
Label: 2 | One-hot: [0. 0. 1. 0. 0.]
Label: 4 | One-hot: [0. 0. 0. 0. 1.]
```

8 Constructing the Keras Model

Using the Sequential model, which is a model consisting of a linear stack of layers, I construct a three layer neural network:

- **First Layer**: Fully connected dense layer with ReLU activation function, 64 neurons, and the input shape determined by the input image.
- **Second Layer**: Fully connected layer with ReLU activation function, 64 neurons, and input shape determined by the previous layer.
- **Third Layer**: Consists of 5 neurons, one for each class and uses the Soft max activation function

I then configure the model's training settings using **categorical cross entropy** because this is not a binary classifier, the **adam optimizer** and record the accuracy metrics to produce the following summary of the model:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                50240

 activation (Activation)     (None, 64)                0

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 64)                4160

 activation_1 (Activation)   (None, 64)                0

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 5)                 325

 activation_2 (Activation)   (None, 5)                 0

=================================================================
Total params: 54725 (213.77 KB)
Trainable params: 54725 (213.77 KB)
Non-trainable params: 0 (0.00 Byte)
```

9 Training

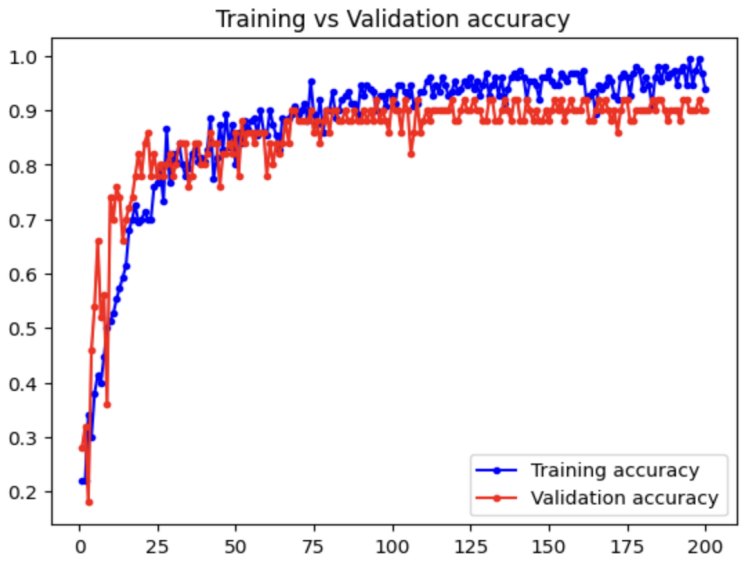
The next step is training the model. I trained the model using 200 **epochs** and a **batch size** of 32 images per batch. A batch loops through one or more samples while making predictions. These

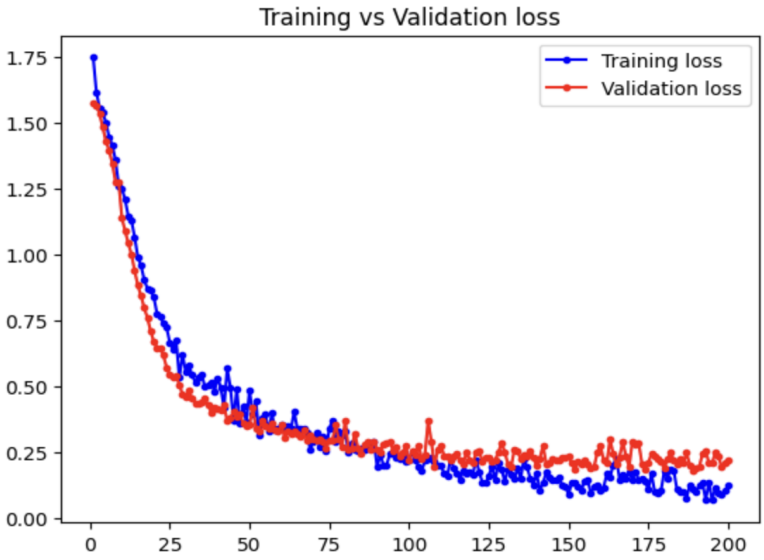
predictions are then compared to the expected outputs, and an error is calculated and used to update and improve the model. On the other hand, an epoch loops through the entire training set. Too many epochs can lead to **overfitting** but not enough could also lead to **underfitting**. As long as the validation loss continues to decrease as more epochs are completed, overfitting has not yet occurred. As the number of completed epochs increases, my validation accuracy increases, and validation loss decreases:

```
Epoch 1/200
4/4 [==============================] - 1s 74ms/step - loss: 1.6054 - acc: 0.2667 - val_loss: 1.4554 - val_acc: 0.2000
Epoch 2/200
4/4 [==============================] - 0s 20ms/step - loss: 1.5023 - acc: 0.3000 - val_loss: 1.4064 - val_acc: 0.4000
Epoch 3/200
4/4 [==============================] - 0s 16ms/step - loss: 1.4097 - acc: 0.3417 - val_loss: 1.3118 - val_acc: 0.4000
Epoch 4/200
4/4 [==============================] - 0s 19ms/step - loss: 1.3798 - acc: 0.3417 - val_loss: 1.2522 - val_acc: 0.5750
Epoch 5/200
4/4 [==============================] - 0s 20ms/step - loss: 1.3273 - acc: 0.3833 - val_loss: 1.2462 - val_acc: 0.6250
```

```
Epoch 196/200
4/4 [==============================] - 0s 14ms/step - loss: 0.0422 - acc: 1.0000 - val_loss: 0.8283 - val_acc: 0.8750
Epoch 197/200
4/4 [==============================] - 0s 13ms/step - loss: 0.0561 - acc: 0.9750 - val_loss: 0.8177 - val_acc: 0.8750
Epoch 198/200
4/4 [==============================] - 0s 14ms/step - loss: 0.0764 - acc: 0.9667 - val_loss: 0.7363 - val_acc: 0.8750
Epoch 199/200
4/4 [==============================] - 0s 19ms/step - loss: 0.0335 - acc: 1.0000 - val_loss: 0.7328 - val_acc: 0.8500
Epoch 200/200
4/4 [==============================] - 0s 15ms/step - loss: 0.0843 - acc: 0.9750 - val_loss: 0.7366 - val_acc: 0.8500
```

Then, using `plt.plot`, I plotted the training and validation losses, and the training and validation accuracies:





10 Testing

10.1 Single Test Sample

Using a sample from the test set, I created a variable, x, and convert the sample into a 2D array instead of a 1D vector by:

```
x = np.expand_dims(X_test[idx], 0)
```

Once a 2D array, I used the `.predict()` function to make a prediction:

```
y_pred = model.predict(x)
```

and completed one-hot encoding:

```
predicted_label = np.argmax(y_pred)
actual_label = np.argmax(Y_val[idx])
```

An example sample from the test set is an image of a diode from the diodes folder so **actual_label = diodes**. The model outputs the probability of the image belonging each of the 5 classes and the class with the highest probability value becomes the predicted label.

```
Model output: [[5.1832251e-07 1.8246345e-08 9.9371654e-01 8.1947485e-07 6.2820716e-03]]
Predicted label: 2 - diodes
Actual label: 2 - diodes
```

10.2 Entire Test Set

For the entire test set the prediction changes to:

```
Y_pred = model.predict(X_test)
```

Next I converted the actual and predicted testing one-hot encoding to numerical labels using the same `np.argmax()` function then printed 50 values from the actual and predicted testing sets:

```
Actual test labels:      [4 3 0 2 2 0 4 0 0 2 4 3 4 3 1 3 2 4 2 2 1 1 3 4 2 1 3 4 0 1 2 3 3 0 1 0 3
 3 4 0 2 4 3 3 1 4 2 2 4 1]
Predicted test labels:   [4 3 0 2 2 0 4 0 0 2 0 3 4 3 1 3 2 1 2 2 1 1 3 4 2 1 3 4 0 1 2 3 3 0 1 0 3
 3 4 0 2 4 3 3 1 4 2 2 4 1]
```

Next, I computed the confusion matrix using:

```
cm = confusion_matrix(y_test, y_pred)
```

and displayed the matrix as:

```
 ---> Predicted labels
|
v Actual labels
                        (0)    (1)    (2)    (3)    (4)
  background (0):  [     8      0      0      0      1]
  capacitors (1):  [     0      8      0      0      1]
      diodes (2):  [     0      0     11      0      0]
        leds (3):  [     0      0      0     12      0]
   resistors (4):  [     0      0      0      0      9]
```

The confusion matrix displays the high accuracy of my model as the values across the diagonal of the matrix are the highest with only 2 incorrect predictions made.

In the final two steps, I score the validation loss and accuracy as well as the testing set's loss and accuracy using the `.evaluate()` function to produce an accuracy and loss value for each set:

```
model.evaluate(X_val, Y_val)
model.evaluate(X_test, Y_test)
```

Where the loss is indexed at `[0]` and accuracy at `[1]`:

```
Validation loss: 0.21707944571971893
Validation accuracy: 0.8999999761581421

Test loss: 0.09772279858589172
Test accuracy: 0.9599999785423279
```

11 Conclusion

In this project, I constructed a 3-layer neural network for multi-class image classification that identifies which of five classes an electronic component belongs to. I used two fully-connected layers after the input layer each comprising of 64 nodes and used the ReLU activation function. The final layer consisted of 5 nodes, one for each of the five classes, and used the softmax activation function. I configured the model using categorical cross entropy as my labels are one-hot-encoded and the Adam optimization algorithm for loss function minimization by dynamic adjustment of the learning rate.