Compute Intelligence Lab 4 PS s7

Joris Plaščinskas

November 12, 2024

Data

I chose to use flower classification dataset. Dataset was split 80/10/10 into train, validation and test sets. DATA SET LINK. The dataset contains 5 species of flowers and in total 3540 labeled images (daisy - 607, dandelion - 872, roses - 615, sunflowers - 673, tulips - 773).

Neural Net Architecture and Data Split

```python
def split_data(data, train_size=0.8, val_size=0.1):
    if(train_size + val_size > 1 ):
        raise Exception()

    train_index = int(len(data) * train_size)
    val_index = train_index + int(len(data) * val_size)

    data_train = data[:train_index]
    data_val = data[train_index:val_index]
    data_test = data[val_index:]

    return numpy.array(data_train), numpy.array(data_val), numpy.array(data_test)
```

Figure 1: Data Split Function

Data is first indexed, then shuffled, then read from disk and finally split into train, validation and test. Figure 1 shows the function that is used to split data set, the first 80% becomes train, next 10% is split and last 10% is test. It's important to shuffle the entries before splitting the data this way.

```
model = Sequential([
    Input(shape=(128, 128, 3)),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(get_num_classes(), activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 2: Model Architecture v0

The simplest model (version 0) can be seen in Figure 2. It has an input of size (128, 128, 3) - the size of a processed image, a convolutional layer that detects certain features in a (2,2) size window throughout the whole image, a max pooling layer that simplifies the data by taking the maximum values in a (2, 2) window. The Flatten layer turns the 3D image data structure into a 1D vector and dense layer is just a classic feed forward neural network layer. I used the most popular - adam optimizer for training and categorical cross-entropy function for training.

```
model = Sequential([
    Input(shape=(128, 128, 3)),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(get_num_classes(), activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 3: Model Architecture v1

The version 1 model can be seen in Figure 3. It's almost the same as the 0th version, but this model also features two dropout layers to combat over-fitting. The dropout fights over-fitting by removing random neurons during training.

```
model = Sequential([
    Input(shape=(128, 128, 3)),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(128, kernel_size=(3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.3),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.4),
    Dense(5, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Figure 4: Model Architecture v2

The final version (2) can be seen in Figure 4. I increased the model size and layer count and introduced a batch normalization layer which helps speed up the learning process by normalizing the activations of each layer.

```
history = model.fit(X_train, Y_train, epochs=12, batch_size=32, validation_data=(X_val, Y_val))
```

Figure 5: Training Method

To train the model I simply used the built in tensorflow 'fit' method (Figure 5). Which under the hood relies on iterative optimization of the categorical cross-entropy loss function using the adam optimizer.

Data Pre-processing

I downloaded the flowers dataset into my hard-drive, then using windows ui I removed the validation and test folders, because I decided that I will use train set for everything. I put each class folder into data folder. During the execution of my python code I first scan the directories and create two lists: X - for storing image paths, Y - for storing image labels. The two lists are then shuffled in the same order and finally image paths are replaced with actual images, which are preprocessed to be 128 x 128. The string labels are also one-hot encoded. All data pre-processing functions can be seen in Figure 6.

```python
def index_data(path="data"):
    image_classes = [item for item in os.listdir(path) if os.path.isdir(os.path.join(path, item))]
    X = []
    Y = []
    for image_class in image_classes:
        class_path = os.path.join(path, image_class)
        file_list = [os.path.join(class_path, file) for file in os.listdir(class_path)]
        X += file_list
        Y += [image_class for i in range(len(file_list))]
    return X, Y


def shuffle_data(X, Y):
    data = list(zip(X, Y))
    random.shuffle(data)
    X, Y = zip(*data)
    return list(X), list(Y)


def read_and_preprocess_image(path):
    image = mpimg.imread(path)
    image = resize(image, (128, 128), anti_aliasing=True)
    image = image / 255.0 if image.max() > 1 else image
    return image


def one_hot_encode(Y, path="data"):
    image_classes = [item for item in os.listdir(path) if os.path.isdir(os.path.join(path, item))]
    Y_encoded = []
    for label in Y:
        new_entry = [0] * len(image_classes)
        new_entry[image_classes.index(label)] = 1
        Y_encoded.append(new_entry)
    return Y_encoded
```

Figure 6: Data Pre-Processing

Computational Resources

I used CPU for processing data and a lot of RAM because all images were loaded to memory at once.

Results

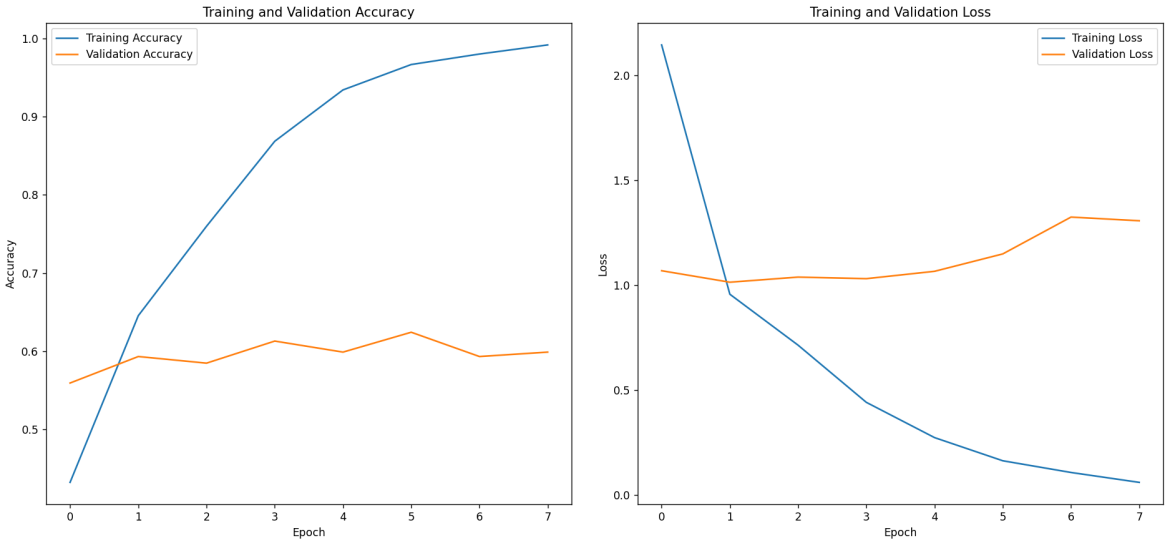

Figure 7: Architecture v0 Results

In Figure 7 are the results of training/validation sets accuracy and loss after 8 epochs of training on architecture v0. The results clearly show that the model tends to over-fit, because the training loss is significantly lower than validation loss and validation loss is even increasing.

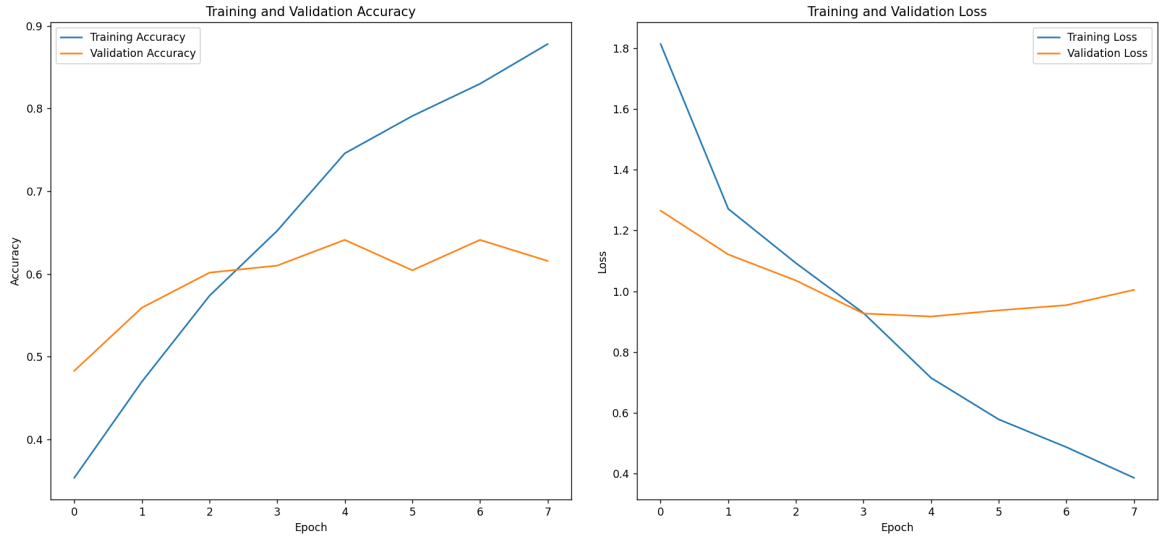

Figure 8: Architecture v1 Results

In Figure 8 are the results of model architecture v1. The results are slightly better as the validation loss does decrease a little bit (this is likely because of the two extra dropout layers), but still is over-fit.

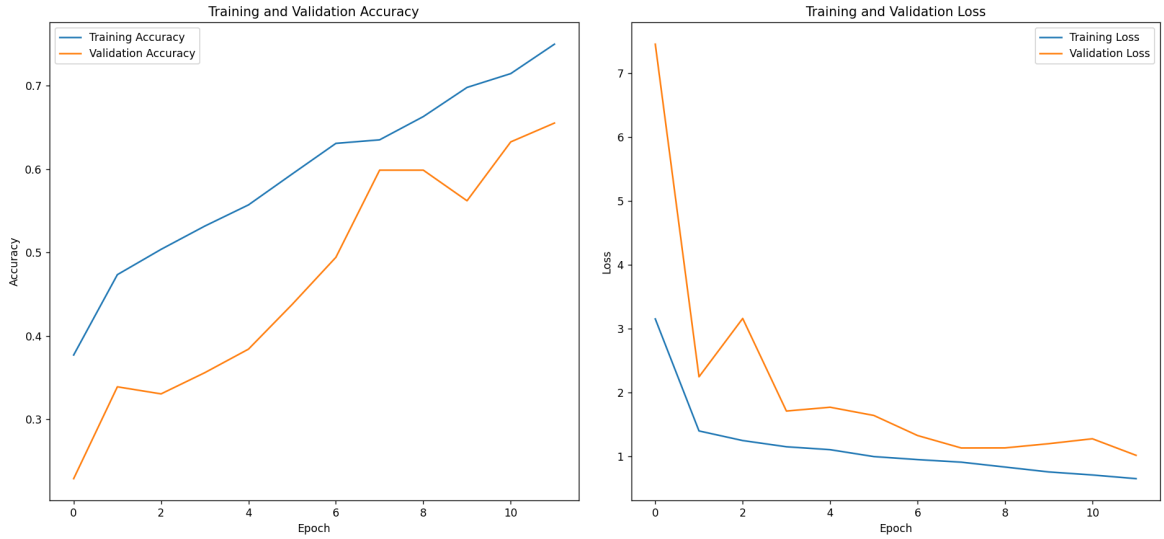

Figure 9: Architecture v2 Results

In Figure 9 are the results of model architecture v2. This version of the model showed the best results, so I decided to stick with this model for all other testing. The validation loss and accuracy functions closely follow training metrics, which is a sign that the model is not over-fitting.

```
Epoch 9/12
89/89 ━━━━━━━━━━━━━━━━━━━━ 31s 349ms/step - accuracy: 0.6696 - loss: 0.8512 - val_accuracy: 0.5593 - val_loss: 1.2994
```

Figure 10: Architecture v2 Results With SGD and 'linear'

I tried changing the optimization method to gradient descent and activation function to linear and surprisingly the results were really good (Figure 10).

Test Set Accuracy and Confusion Matrix

```
Test accuracy: 0.8136
```

Figure 11: Architecture v2 Test Set Accuracy

The accuracy (on a test set) of architecture v2 model after 12 epochs of training is 0.8136.

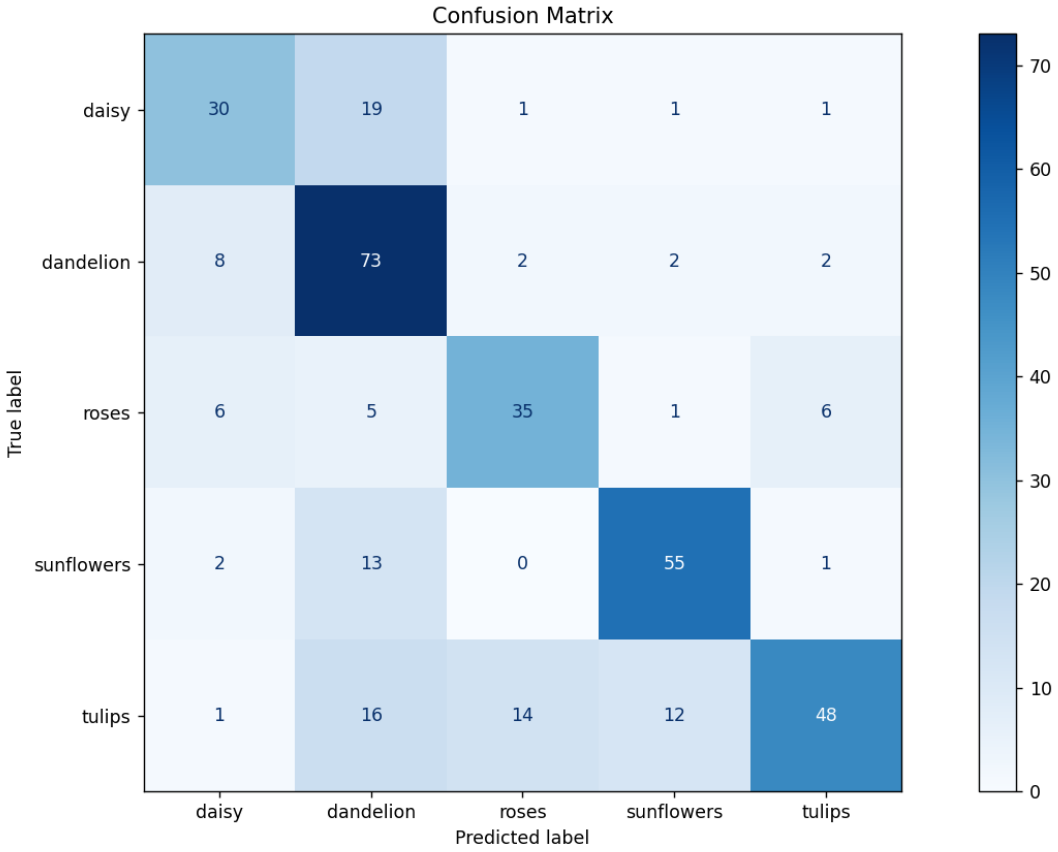

Figure 12: Architecture v2 Confusion Matrix On Test Set

Figure 12 shows the confusion matrix of test set predictions. The matrix shows good results as most the highest numbers lie on the diagonal. We can also see that the model tends to predict 'dandelion' when the true label is actually 'sunflowers' or 'tulips' or 'daisy'.

Test Set Predictions

Table 1: True Labels Compared To Model Predictions

True Label	**Predicted Label**
tulips	tulips
daisy	dandelion
dandelion	dandelion
roses	tulips
dandelion	dandelion
roses	roses
dandelion	dandelion
tulips	tulips
daisy	daisy
daisy	daisy
dandelion	dandelion
tulips	tulips
sunflowers	sunflowers
tulips	roses
roses	dandelion
dandelion	dandelion
dandelion	dandelion
roses	roses
roses	roses
dandelion	dandelion
roses	tulips
dandelion	dandelion
daisy	daisy
dandelion	dandelion
dandelion	dandelion
sunflowers	sunflowers
dandelion	dandelion
sunflowers	sunflowers
tulips	roses
dandelion	dandelion

Table 1 above compared the true labels of test set to the model predictions.

Conclusion

One of the more interesting insights to me was that the model was able to perform with linear activation function after being trained using gradient descent. I always thought that these methods were ineffective for neural networks and that a linear activation function is bad, because it does not introduce non-linearity. Also, using batch normalization and dropout proved to be super effective in combating over-fitting. Another interesting insight is that my computer memory was able to handle the whole dataset of images.