CSC2034: Biocomputing

https://colab.research.google.com/drive/19xV-s3nVfEXqpX-m1n233_5HEYAW5-b3?usp=sharing

Word count: 2602

May 11, 2021

1 Abstract

Regularizers in Convolutional Neural Network (CNN) are used to reduce generalization error. Within this assignment, I conducted systematic experiments on the frequently used regularization strategies for reducing overfitting in CNNs, after briefly investigating a disprepancy between the provided code for the assignment and the example in the Keras documentation. Overall, I was able to achieve 81% accuracy by 50th epoch with a mixture of regularization strategies, including pooling, dropout, and image augmentation. Although a far cry from the accuracies achievable by modern predefined architectures such as VGG, I was able to gradually improve the provided architecture through systematic experiments, which will be discussed in this report.

2 What Was Done and How

2.1 Custom architecture

The provided code for the assignment features a basic CNN (Convolutional Neural Network) with a custom defined architecture. The stack for the model consists of two Keras Conv2D layers, with one Max Pooling layer and two Dense layers.

```
model = models.Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

A Conv2D layer is a 2D convolutional layer. Essentially, a filter is convolved (passed over) on the feature (the image) to produce a smaller output feature map. The kernel size defines the size of the filter, in this case a 3x3 matrix. Passing this matrix over the feature highlights different aspects of the feature, such as lines in the image.[2]

```
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
```

In order to produce the output feature map from the convolution, the value at the same position is multiplied between the filter and the input feature. This process is repeated by sliding the filter over the input feature to produce a weighed combination of each individual input value. Essentially, for a input feature of size 3x3 and a filter of size 3*3, the weighted average for the element at position [2,2] would be

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} * \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = (a*1) + (b*2) + (c*3) + ... + (i*9)$$

This is also called an element-wise product or Hadamard product. [5, p. 34]

After two convolutions are completed, the output feature map is pooled using a MaxPooling2D function. A max pooling operation produces a summary of the feature map, extracting essential features. The specifics of max pooling and the experiments that I conducted are discussed in the latter chapter.

The output is then flattened from a 2D to 1D coordinate with `flatten()`, then passed into a dense layer, which is a traditional deeply connected neural network layer. Examining the stack shows that 12544 weights are fed into 128 perceptrons, resulting in a total parameters of 12544 * 128 = 1605760.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_4 (Conv2D) (None, 30, 30, 32) 896
_________________________________________________________________
conv2d_5 (Conv2D) (None, 28, 28, 64) 18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 64) 0
_________________________________________________________________
flatten_2 (Flatten) (None, 12544) 0
_________________________________________________________________
dense_4 (Dense) (None, 128) 1605760
_________________________________________________________________
dense_5 (Dense) (None, 10) 1290
=================================================================
Total params: 1,626,442
Trainable params: 1,626,442
Non−trainable params: 0
_________________________________________________________________
```

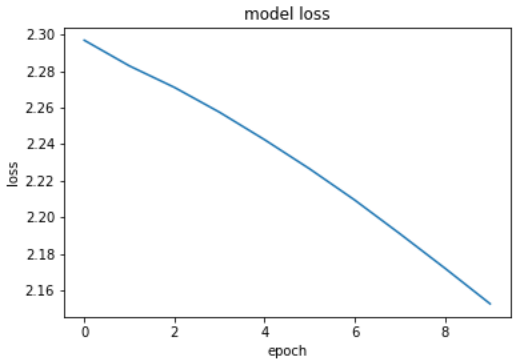
After this operation is completed, the activation function is applied, as the option has been set for ReLU (Rectified Linear Unit). This operation converts the weighted values to 0 if it is less than 0. The advantage of this is that a true 0 value can be outputted which is beneficial within the hidden layers (the dense layer). [5, p. 507]

After the probabilities are normalized with a softmax function so the probabilities of the categories add up to 1, these probabilities are finally compiled and fed into a categorical crossentropy loss function:

$$-\sum_{c=1}^{M} y_{o,c} \log\left(p_{o,c}\right)$$

where M is the number of classes, y is the binary indicator for correct classification, and p is the predicted probability. The equasion compares the output probability distribution against the desired ground truth values, and a logarithmic penalty is given based on how far the values are. Exponentially larger penalties are given for differences that are further. Hence, the loss function attempts to optimize the neural network by aiming to reduce the loss value.[5, p. 82][5, p. 179]

Training the model with above parameters yields the following loss graph:



The graph shows that loss is reducing for every epoch. Although this is indicative of the neural network improving itself, I wanted to compare it to the example given by the official Keras documentation, for the same dataset (CIFAR-10). In order to accomplish this, I added an option to evaluate the model at the end of the epoch with validation data.

```
history = model.fit(x_train, y_train,
batch_size=batch_size,
epochs=epochs,
validation_data=(x_test, y_test),
verbose=1)
```

Comparing the accuracy against the official example shows that the accuracy is significantly lower in the provided example at epoch=10.

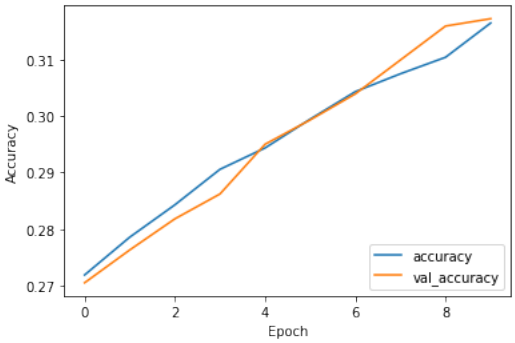


Figure 1: *Accuracy from provided example*

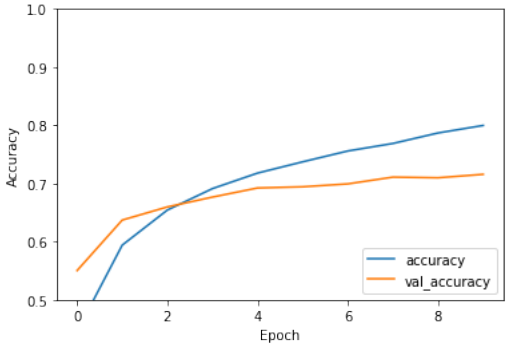


Figure 2: Accuracy from official Keras documentation[3]

2.2 Discrepancy with the official documentation

This section will be brief as it is my intention to put emphasis on the experiments on the regularization strategies that I conducted in the latter sections. This is because I spent the longest time designing and conducting experiments for regulurizers, due to the complexity of evaluating hyperparemeters and the best architecture. Instead, this section highlights the experiments that I conducted to address the discrepancy in the accuracy between the example in the Keras documentation and the provided custom architecture.

2.2.1 Disabling the conversion from dense to sparse data

I investigated the possible causes to the discrepancy, and by comparison, found that the way that the data is prepared is a potential culprit, as it did not exist in the official documentation.

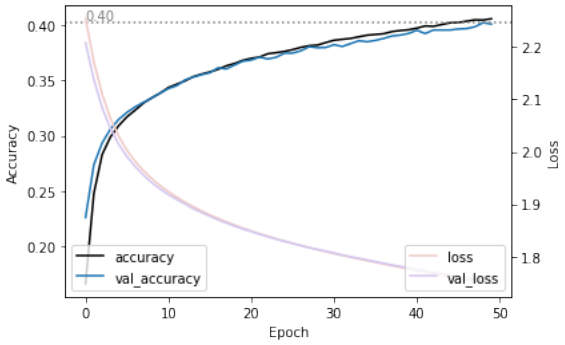
```
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 3, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 3, img_rows, img_cols)
    input_shape = (3, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 3)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 3)
    input_shape = (img_rows, img_cols, 3)
```

Another difference that did not exist in the official example is the conversion conversion from class vector to binary class matrix. Essentially, *dense* representation of vectors are converted to a *sparse* representation of one-hot encoded matrix. Hence, a label of 6 (snake) is converted to [0,0,0,0,0,1,0,0,0,0]. I also removed this, and then changed the loss function from `categorical_crossentropy`, which expects one-hot encoded matrix, to `SparseCategoricalCrossentropy`, which can be configured to use logits vectors.[6]

```
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test_orig = y_test
y_test = keras.utils.to_categorical(y_test, num_classes)
...
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

In order to more throughly test the model, I added extra information to the output graph: extra labels for plotting loss, and max accuracy attained during the epochs (see listing 6 in the appendix for the code).

Although conversion from sparse to dense representation removed overhead for the system[9, p. 24], the above modifications did not produce a significant difference in terms of the accuracy values or train time.



2.2.2 Using Adam instead of Adadelta as the optimizer

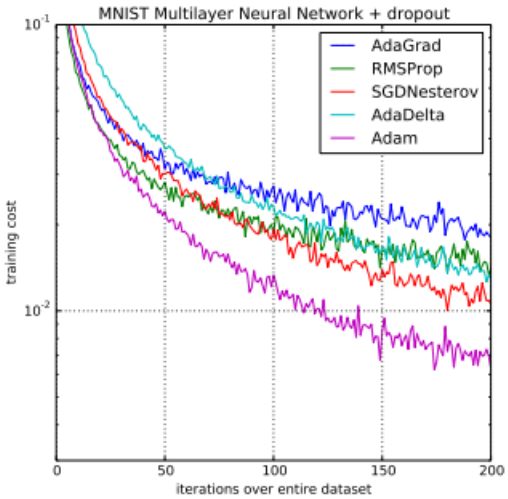
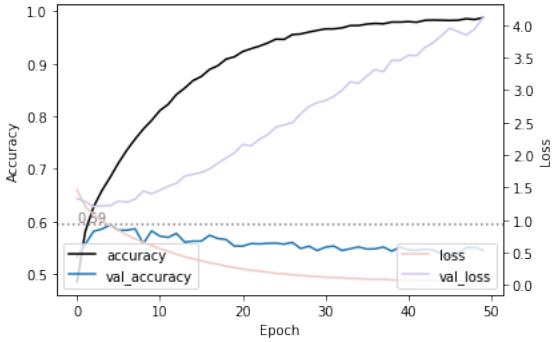


Figure 3: *Comparison of Adam with other popular optimizers, showing superior performance of Adam[4]*

Concluding that the difference is likely from a different source, I found another difference. The official example used a different optimizer called 'adams'. Changing the optimizer from `keras.optimizers.Adadelta()` to `keras.optimizer.Adams()` produced outputs that are within the range of the official example.

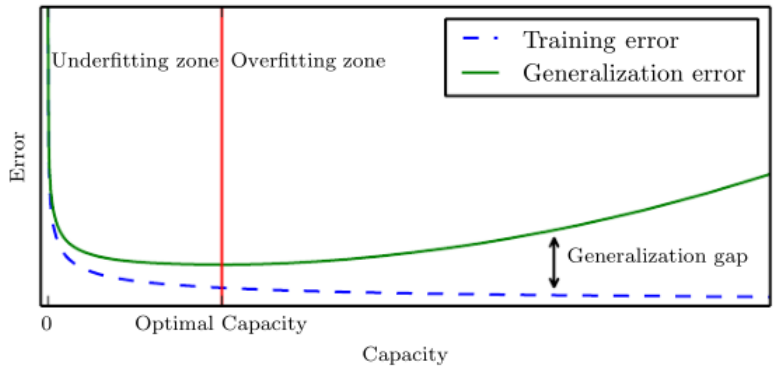


The graph produced also indicates a possible overfitting of the algorithm[1], as the acccuracy on the test data

[1]Although I wanted to explore the different types of optimizers and their effects, I opted to explore the ways to improve the overfitting that is apparant, as figure 3 suggests Adam can be the most effective optimizer for a given dataset

is much lower than for the training data. The difference between accuracy and val_accuracy (also called the *generalization gap*) suggests that the model memorizes properties of the training set that is counterproductive for the test set.

2.3 Reducing the generalization gap



Being able to reduce overfitting is an important task in creating a versatile neural network, as model with overfitting will not be able to achieve a good accuracy on real world data that is not part of the data that the model was trained with.

There are many causes of overfitting. Epoch count is one of the hyperparameters that can contribute to overfitting. One commmon method of countering this is through the use of early stopping to stop the training progress by keeping track of the out of sample error (the error for the test set). However, doing so could obscure overfitting caused by other hyper-parameters besides the epoch count.[9, p. 9] In fact, as the graph levels off at the 65% level, it likely indicates that there are other causes that lead overfitting. Instead, I decided to evaluate regularization strategies because regularization is a process that is used to reduce the generalization gap. Therefore, I decided to explore the following regularization strategies: dropout, pooling, and image augmentation. I opted for those strategies as they are regarded as basic building blocks of regularization.[5, p. 228] The following sections will discuss the methods in which I varied the architecture with regularization strategies and their respective hyperparameters.

2.3.1 Dropout

Dropout aims to reduce generalization gap by approximating *bagging*. Bagging reduces generalization error by combining several models. However, this process is computationally expensive for a CNNs due to the complexity of CNNs. Therefore, dropout approximates bagging by ranmdomly removing units from neural network by multiplying its output by 0.[5, p. 258] Within the wider neural networking context, this is a major reason that dropout is used frequently, due to the reduced computational capacity required to achieve bagging for CNNs.

I added dropout layers one after another, as it was the most basic way of systematically increasing the independent variable, and trained the model.

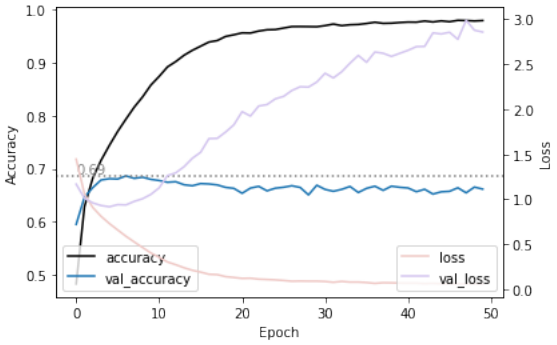


Figure 4: *Accuracy with one dropout layer*

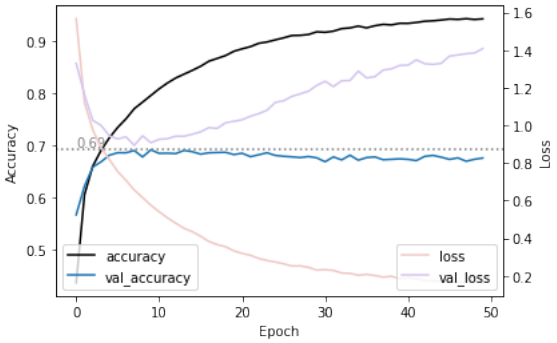


Figure 5: *Accuracy with two dropout layers*

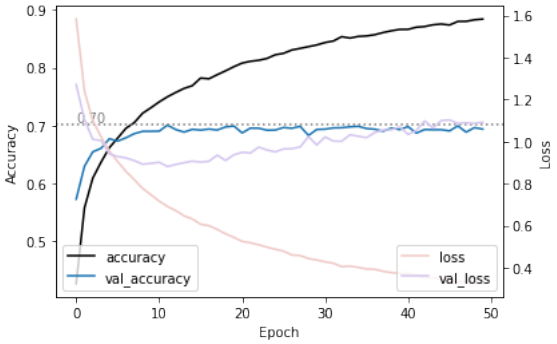


Figure 6: *Accuracy with three dropout layers*

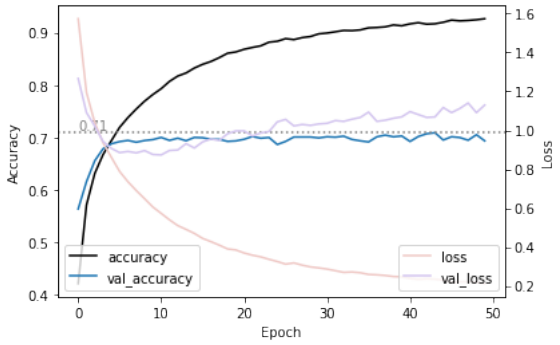


Figure 7: *Accuracy with four dropout layers*

With 4 dropout layers, the train accuracy worsens. This is a sign of *underfitting*, a condition where the neural network lacks the ability to sufficiently learn the training data.[5, p. 111] To fix the underfitting, I added more convolutional layers to increase its capacity. This resulted improvement of the test accuracy from 0.73 to 0.89, and an improvement of the test accuracy from 0.71 to 0.74.

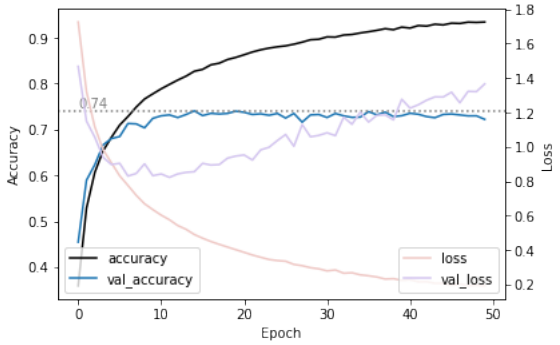


Figure 8: *Accuracy with 4 dropout and increased capacity*

With 5 dropout layers and increased capacity, the results show the underfit worsening again, suggesting that 4 dropout layers is likely the maximum amount.

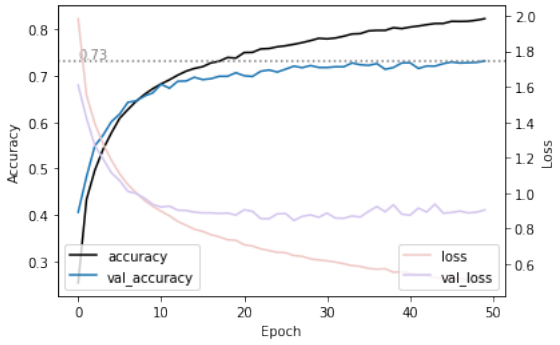


Figure 9: *Accuracy with 5 dropout layers and increased capacity*

Dropout layer has a hyperparameter that can vary the percentage of units that are retained. I also experimented with systematically changing the dropout values. I experimented with a) increasing dropout

values further down the stack, b) decreasing dropout values further down the stack, and c) constant dropout values (for exact hyperparameters, see listing 3, 4, 5 in the appendix). I found that incrementally increasing dropout values the further one goes down the stack resulted in the most accuracy for the test data.

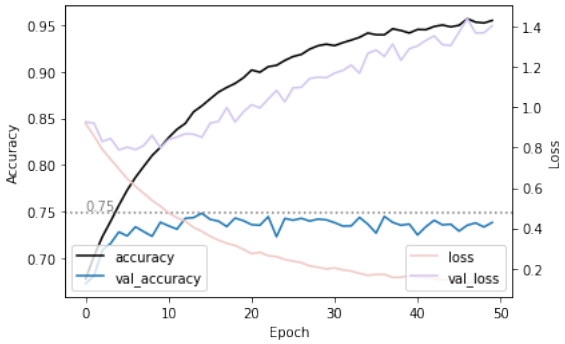


Figure 10: *Constant dropout values*

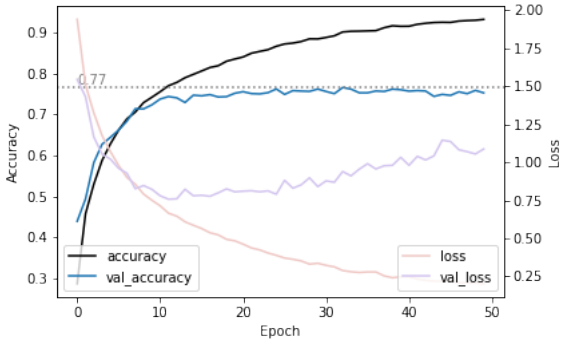


Figure 11: *Accuracy with increasing dropout values further down the stack*

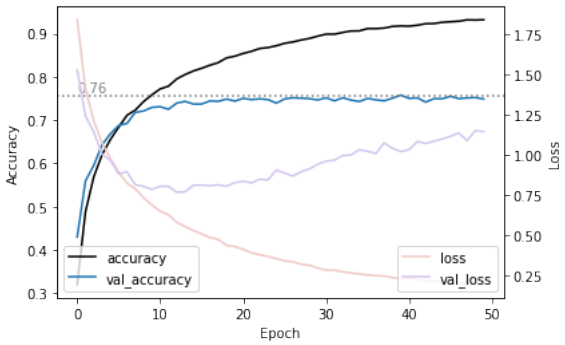


Figure 12: *Accuracy with decreasing dropout values further down the stack*

2.3.2 Pooling

Pooling is another method for regularization. During a max pooling operation, a filter is convolved around the input feature, but the highest value found within the filter area is extracted into a new feature map.

Hence, the output shape of (28,28,64) is reduced to (14,14,64). A max pooling operation with filter size of (2,2) and stride of 2 will always halve the dimension size.[5, p. 339] This is one of the reasons pooling is an important strategy for developing a CNN, as it extracts important features from the feature map, as well as reducing system overhead by downsampling.

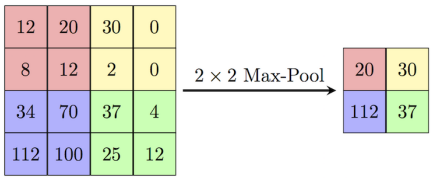


Figure 13: Example of how a 4x4 image can be max pooled into a 2x2 feature map[7]

I gradually added max pooling layers, however the max pooling layer possible was 3 layers. This is because a max pooling operation of size (2,2) will halve the feature map size, as shown above. After subsequent max pooling operations, the feature map is too small for another pooling operation.

I added max pooling layers to the previous model with the highest accuracy

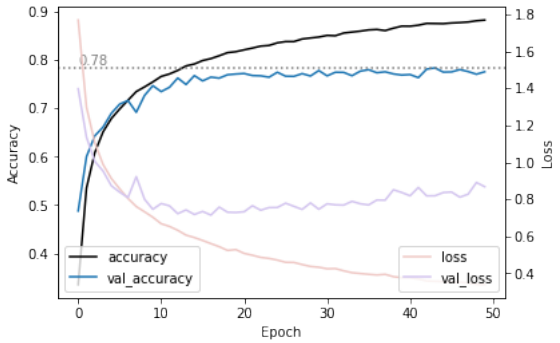


Figure 14: Accuracy with 1 max pooling layer

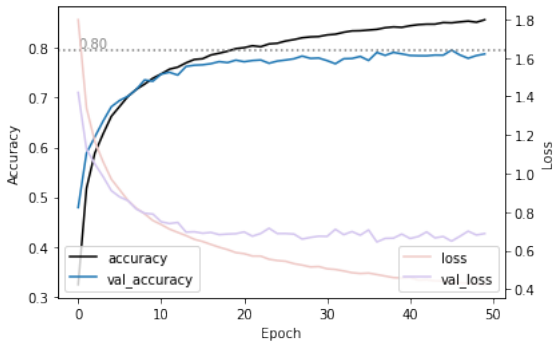


Figure 15: Accuracy with 2 max pooling layers

2.3.3 Data augmentation

With two regularization strategies explored, I wanted to explore one final regularization method and opted for *data augmentation*. Data augmentation is a process where images are modified to produce larger input data.

By making extra copies of the data with transformations, generalization error can be reduced. [5, p. 240] Data augmentation is important in the wider context of neural network, due to the fact input data in the real world can be limited in number, due to time and financial constraints. Therfore, being able to generate similar images to reduce regularization is an immense tool for creating a versatile neural network.

I added subsequent data augmentation to the previous model witht the highest accuracy (2 maxpooling layers). I used a number of different image augmentation methods. Height and width shift options will randomly select a pixel and shift its location by a given amount. Horizontal flip will mirror the image horizontally.

Highest accuracy of 0.81 by 50th epoch was achieved with horizontal flip. Height and width shift lowered the accuracy compared to not having any image augmentation. It also significantly lowered the train accuracy, which is indicative of underfitting. This could be caused by the image augmentation overtly distorting images such it becomes difficult for the model to achieve a good fit.

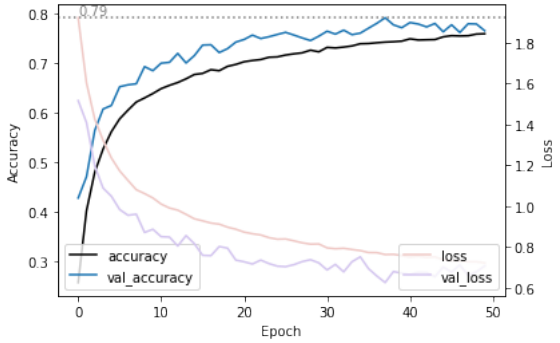


Figure 16: Accuracy with height and width shift

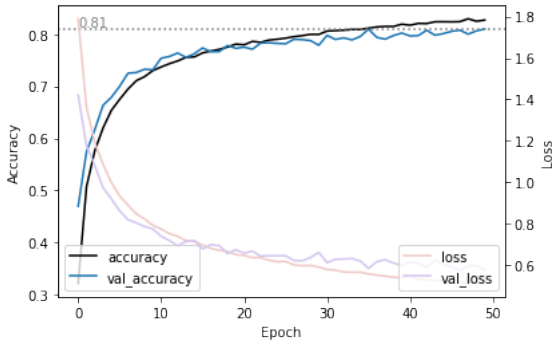


Figure 17: Accuracy with horizontal flip

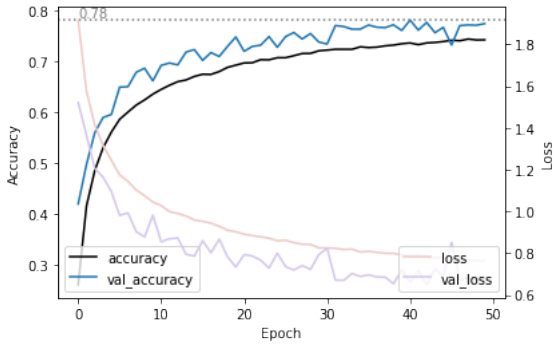


Figure 18: Accuracy with height and width shift + horizontal flip

3 Conclusions and future work

With this assignment, I was able to successfully explore a custom CNN architecture and the difficulties of improving its accuracies. I conducted systematic experiments on the various regularization strategies, with exploration of the hyperparameters. Furthermore, through experimenting with the strategies, I was able to learn the mechanisms of regularizers.

In the prior section, a single graph for each model was used to compare the outputs. However, I found that there can be significant variance between different training runs of the same model. Therefore, to derive the average accuracy of a given model, I trained each model 3 times using a helper function and outputted a box plot. (see listing 7, 8 for code).

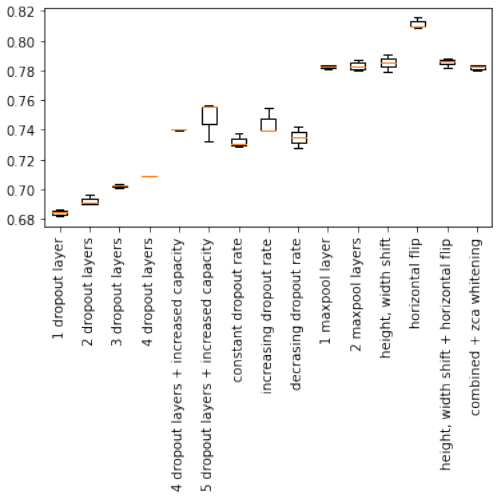


Figure 19: Box plot of the accuracies of different models

With subsequent addition of the dropout layers, the accuracy improved, but saw diminishing returns on the test set accuracy as well as a reduction in the train accuracy. Adding additional capacity in the means of convolution layers dramatically improved the test accuracy to a maximum of 0.76. I then experimented with adding even more dropout layers and capacity, and found that the optimal amount was 4 dropout layers + increased capacity.

I then experimented with adding max pooling layers, and found that having a max pooling layer leads to dramatic improvement in the accuracy, however a marginal difference between 1 and 2 max pooling layers. The location of the max pooling layer within the stack also had a major impact, with the best accuracies

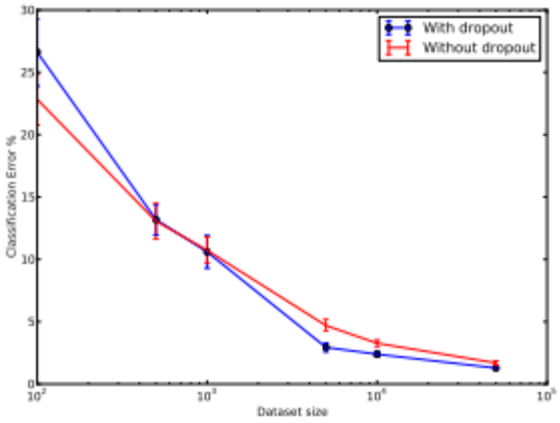


Figure 20: Effect of varying data set size with respect to dropout[8]

being achieved when the pooling layer is at the top of the stack. This is likely due to the layer having a cascading effect down the rest of the stack.

I also experimented with the hyperparamers of the image augmentation, and found that having a single horizontal flip augmentation leads to the highest accuracy of 0.82. I also found that other augmentations resulted in underfitting. This suggests that either the hyperparameters that decide the strength of the augmentation need to be reduced, or the capacity of the model needs to be increased to address the underfitting. However, image augmentation was useful in increasing the test accuracy by almost a 1%.

Overall, the latest papers being published show test accuracies ranging from 90% to almost 99.9%[1], this shows that there is a substantial gap between what I was able to achieve and what is achievable by state of the art machine learning algorithms.

In the future, I would like to conduct a thorough investigation of the hyperparameters of image augmentation, and evaluate which one will lead to the highest test accuracies. I found that image augmentation lead to worsening of the underfit (accuracy of the train data decreased), likely due to the increased diversity of the images, hence requiring extra capacity. I would like to find the optimum hyperparameters that address the overfitting without creating an extremely distorted image. I would also like to investigate the interactions between different regularization strategies, as my final model utilizes several. As figure 20 suggets, there can be a sweet spot between data set size and the use of drop out, which can also equate to image augmentation as image augmentation aims to simulate a larger data set size.

In my personal opinion, I found that this assignment to be difficult due to many reasons. For example, there were difficulties with exploring image augmentation, due to the increased load on the processing capabilities of the system. Training a model with image augmentation required almost 10 times longer due to the fact that Keras augments images by randomly applying augmentations to an existing data during each runtime. However, a more abstract difficulty I faced was due to the mathematical background required for machine learning. The formulas that underlie the algorithms such as softmax functions were personally difficult to understand, which was a major barrier to making progress with building the CNN stack, as well as understanding how to alter the hyperparameters associated with a given formula or algorithm. On the other hand, I am pleased with the Jupyter Notebook that I was able to work on, with gradual improvement of the test accuracies being visible in a chronologial order, as well as the systematic experiments that I conducted. I am also pleased to develop the automatic outputting of the box plot.

Overall, I learned a significant amount about machine learning, as it was a field that I had no prior knowledge of. It was engaging to learn about the challenges of developing a versatile neural network, as well as the the regularization strategies used to produce a neural network with a good fit.

4 Appendix

```
Model: "sequential_3"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_6 (Conv2D) (None, 30, 30, 32) 896
_________________________________________________________________
dropout_3 (Dropout) (None, 30, 30, 32) 0
_________________________________________________________________
conv2d_7 (Conv2D) (None, 28, 28, 64) 18496
_________________________________________________________________
dropout_4 (Dropout) (None, 28, 28, 64) 0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 64) 0
_________________________________________________________________
flatten_3 (Flatten) (None, 12544) 0
_________________________________________________________________
dense_6 (Dense) (None, 128) 1605760
_________________________________________________________________
dropout_5 (Dropout) (None, 128) 0
_________________________________________________________________
dense_7 (Dense) (None, 10) 1290
=================================================================
Total params: 1,626,442
Trainable params: 1,626,442
Non−trainable params: 0
_________________________________________________________________
```

Listing 1: Added dropout layers

```
Model: "sequential_7"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_26 (Conv2D) (None, 30, 30, 32) 896
_________________________________________________________________
conv2d_27 (Conv2D) (None, 28, 28, 64) 18496
_________________________________________________________________
dropout_18 (Dropout) (None, 28, 28, 64) 0
_________________________________________________________________
conv2d_28 (Conv2D) (None, 26, 26, 64) 36928
_________________________________________________________________
conv2d_29 (Conv2D) (None, 24, 24, 64) 36928
_________________________________________________________________
dropout_19 (Dropout) (None, 24, 24, 64) 0
_________________________________________________________________
conv2d_30 (Conv2D) (None, 22, 22, 64) 36928
_________________________________________________________________
conv2d_31 (Conv2D) (None, 20, 20, 64) 36928
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 10, 10, 64) 0
_________________________________________________________________
dropout_20 (Dropout) (None, 10, 10, 64) 0
_________________________________________________________________
flatten_7 (Flatten) (None, 6400) 0
_________________________________________________________________
dense_15 (Dense) (None, 128) 819328
_________________________________________________________________
dense_16 (Dense) (None, 64) 8256
```

```
-----------------------------------------------------------------
dropout_21 (Dropout) (None, 64) 0
-----------------------------------------------------------------
dense_17 (Dense) (None, 10) 650
=================================================================
Total params: 995,338
Trainable params: 995,338
Non−trainable params: 0
-----------------------------------------------------------------
```

Listing 2: Dropout but with more layers (increased capacity)

```
model = models.Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.1))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 3: Increasing dropout values further down the stack

```
model = models.Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.4))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.3))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 4: Decreasing dropout values further down the stack

```
model = models.Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.4))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.3))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 5: Constant dropout values

```
fig, ax=plt.subplots()
max_val = max(history.history['val_accuracy'])
ax.plot(history.history['accuracy'], label='accuracy', color="black")
ax.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
ax.axhline(y=max_val, color="gray", linestyle="dotted")
ax.text(0 , max_val, "{:.2f}".format(max_val), color="gray",
        ha="left", va="bottom")
plt.legend(loc='lower_left')
```

Listing 6: Additional modifications to the plot

```
def compile_then_plot(func, name='', iteration=3):
    experiment[name] = []

    for i in range(iteration):
        print(i + 1 , "/ 3")
        model = func()

        model.compile(optimizer=keras.optimizers.Adam(),
                      loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                      metrics=['accuracy'])

        history = model.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            validation_data=(x_test, y_test),
            verbose=0)

        experiment[name].append(history)
        history = []
```

Listing 7: Helper function to run experiment 3 times and save results to dictionary

```
def plot_final(experiment):
    keys = list(experiment.keys())
    index = []
    data = []

    for i in range(len(keys)):
        index.append(i + 1)

    for key in experiment:
        data.append(list(map(lambda x: max(x.history['val_accuracy']), experiment[key])))

    fig, ax = plt.subplots()
    ax.boxplot(data)
    plt.xticks(rotation=90)
```

```
    plt.xticks(index,keys)
    ax.set_aspect(50)
    plt.show()
```

Listing 8: Helper function to draw a box plot

References

[1] Papers with Code. *Image Classification on CIFAR-10*. URL: https://paperswithcode.com/sota/image-classification-on-cifar-10 (visited on 05/04/2021).

[2] *Conv2d — API Documentation — TensorFlow Core v2.4.1*. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D (visited on 04/16/2021).

[3] *Convolutional Neural Network (CNN) — TensorFlow Core*. URL: https://www.tensorflow.org/tutorials/images/cnn (visited on 04/16/2021).

[4] Jimmy Ba Diederik P. Kingma. *Adam: A Method for Stochastic Optimization*. 2014.

[5] Yoshua Bengio Ian Goodfellow and Aaron Courville. *Deep learning*. Cambridge, Massachusetts: The MIT Press, 2016.

[6] *Machine Learning Glossary — Google Developers*. URL: https://developers.google.com/machine-learning/glossary/#logits (visited on 04/17/2021).

[7] *Max-pooling / Pooling*. URL: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling (visited on 05/01/2021).

[8] Alex Krizhevsky Nitish Srivastava Geofferey Hinton. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. 2014.

[9] *Practical recommendations for gradient-based training of deep architectures*. 2012. URL: https://arxiv.org/abs/1206.5533.