Deeper CNN Architecture and Dropout Improve Loss Errors in Image Classification

Author: Thanh Bui
University of California, San Diego
t7bui@ucsd.edu

Abstract

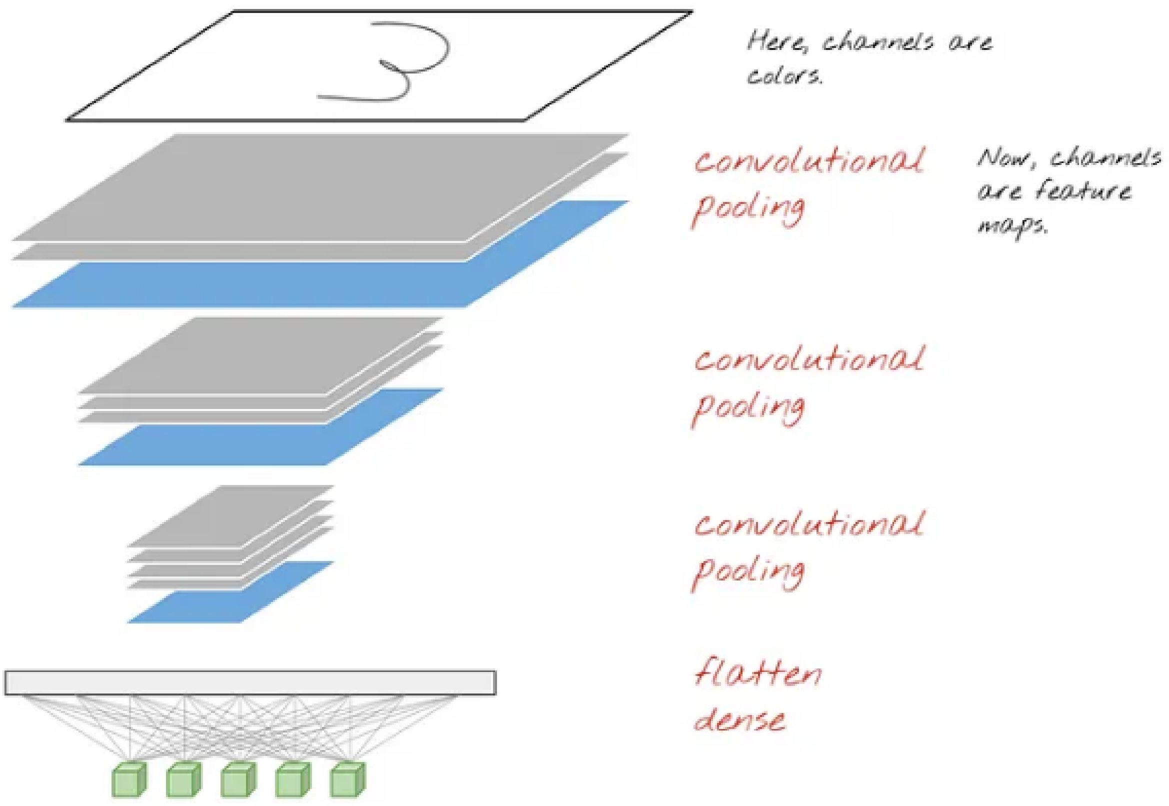
Recently, deep learning has been used to classify images, detect and recognize text, track objects, etc. Furthermore, Convolution neural networks are also commonly used in the deep learning area and also demonstrated high performance in image classification. In this paper, I built a Convolutional Neural Network (CNN) model with a three-layer convolution neural network to classify images from the CIFAR dataset. During the experiment, I try to improve my model by adding more convolution layers and dropout layers. The result shows that the final model, after iterations of the original three-layer CNN model, actually improved and learned something during the process. The accuracy and loss error also improve over time with higher epochs and callbacks.
Keywords: Image classification, image augmentation, CNN, max pooling, CIFAR, Adam optimizer

1. Introduction

1.1 Convolution Neural Networks

Deep Learning or Deep Neural Networks has been considered to be one of the most powerful tools and become very popular due to its ability to handle a large amount of data. Convolution Neural Network (CNN) has performed high results in fields related to pattern recognition, including image processing and voice recognition. CNNs are most beneficial in reducing the number of parameters in Artificial Neural Networks (ANN) [1].
CNN model is simple terms, the model takes in raw pixels from an input image, then connects the input layer to neurons/nodes in the hidden layers in the Multi-Layer perceptron to detect complex features like shapes and objects. Then the later layers are fully connected layers that are “directly connected to every node in both the previous and in the next layer” (Albawi, Mohammed) [1].

Common setup [3]



1.1.1 Padding

Disadvantage of the convolution step is the loss of information if the border of the image if the information is only captured when the filter of the hidden convolution layer slides [1]. We can use padding to solve this problem, in this case, use zero padding. In Tensorflow Keras, padding="same" implies that zero paddings will be applied evenly left/right or up/down of the input.

1.1.2 Pooling

Pooling main purpose is to downsample to reduce complexity for later layers. It is similar to reducing image resolution in image processing. It is very common to have 2x2 as the size of the pooling layer. “Max pooling is the most common type of pooling ... partitions the image to sub-region rectangles, and only returns the maximum value of the inside of that sub-region” (Albawi, Mohammed) [1]

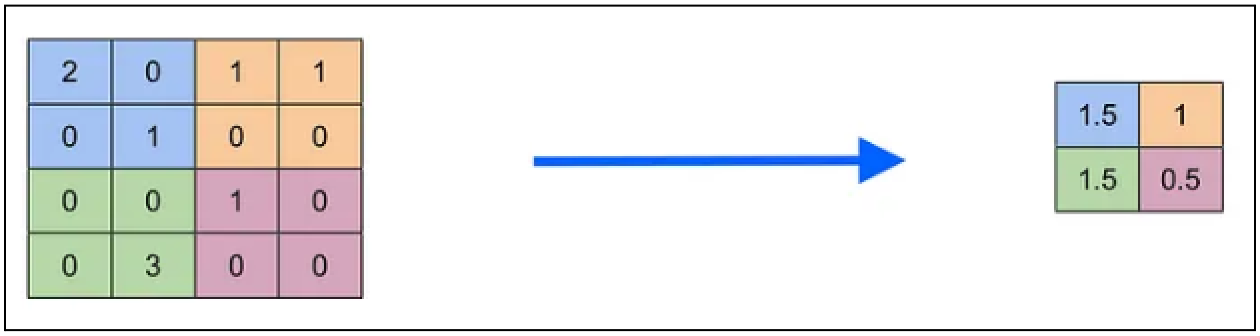


Fig 1 - Average Pooling

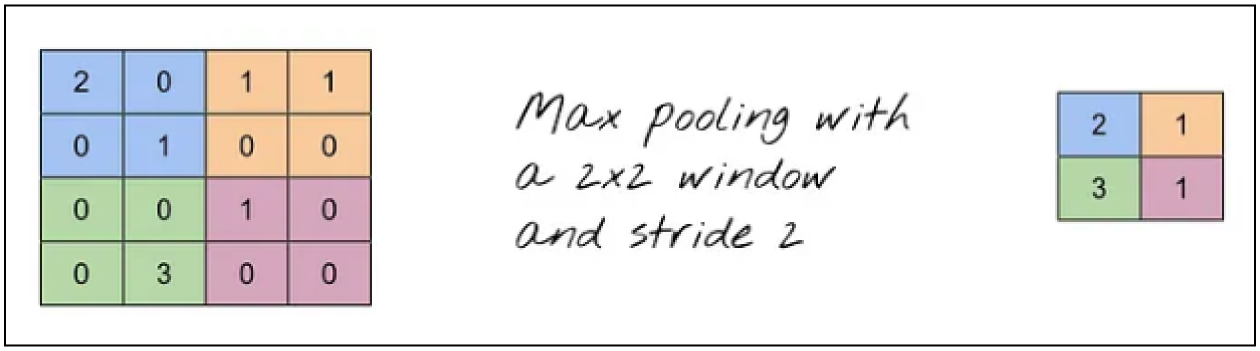


Fig 2 - Max Pooling

1.1.3 Fully Connected Layer

FOr fully-connected layer, each node in this layer is directly connected to all the nodes in the previous and latter layers. Usually, each of the nodes from the previous frames in the pooling layer are “connected as a vector to the first laer from the fully-connected layer” (Albawi, Mohammed) [1]. The softmax activation function is often used in classification in the last layer, i.e. the output later, of the neural network models [2].

1.2 Drop-Out Layer

Overfitting happens when the model learns and gives an accurate performance on the training but performs poorly on the dataset it has never seen before, i.e. test/validation dataset. Hence the generalization error increases. To avoid this, we can use “dropout to prevent complex co-adaptations on the training data” (Hinton, Srivastava, Krizhevsky, Sutskever, Salakhutdinov) [3]. In every training case presentation, a probability of 0.5 is applied to randomly exclude each hidden unit from the network. This random omission ensures that the hidden units cannot depend on the presence of others [3].

1.4 ReLU Activation Function

$ReLU(x) = max(0, x)$

$\frac{d}{dx}ReLU(x) = \{1\ if\ x > 0;\ 0\ otherwise\}$

While sigmoid and tanh were more popular for non-linearity for many years, ReLU has been used more often recently because it is simpler in both function and gradient. ReLU converts all negative values to zero, hence converting the threshold to zero. “It does not affect the volume or hyperparameters” (Ajit, Acharya, Samanta) [5].
ReLU has a constant gradient for positive input. When we have zero gradients in ReLU, this leads to obtaining a complete zero, hence creating a sparser representation [1].

1.5. Optimizer

1.5.1 Stochastic Gradient Descent (SGD)

For Stochastic gradient descent (SGD), we first initialize our parameters w and learning rate λ. For each iteration, we randomly shuffle our data to compute the output for the randomly chosen data point. Then compare the model output, $\hat{y}_i$, to the target output, y_i. If the $\hat{y}_i$ and y_i match, then we move on and choose another random datapoint.. If $\hat{y}_i$ and y_i do not match, we will update our weights as shown below, then move on and choose another random data point. You will continue to choose random data points in your dataset until you reach an approximate minimum of your parameters.
Training:

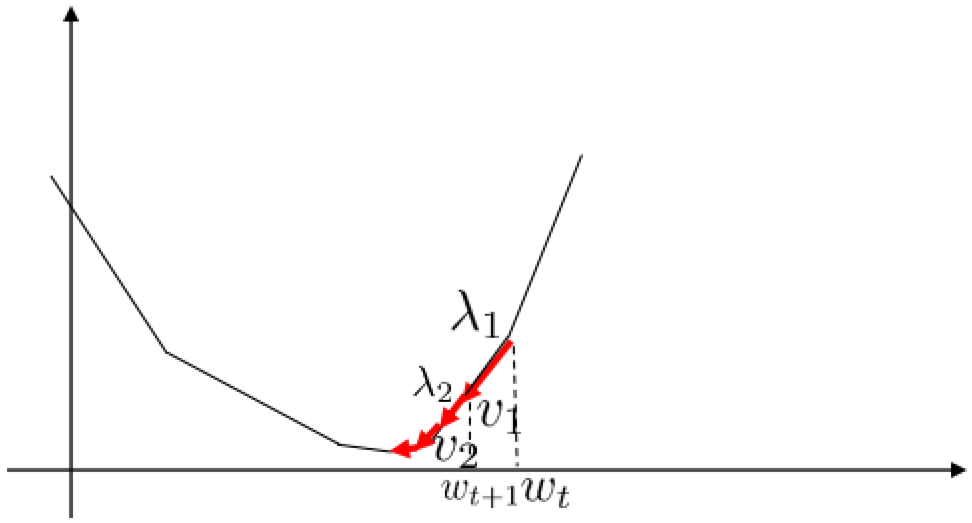
$$\mathcal{L}(w, b) = \sum_i max(0, -y_i(w^T x_i + b))$$

Update weights when $\hat{y}_i$ and y_i do not match:

$$\frac{\mathcal{L}(w, b)}{dw} = \sum_i \frac{-1}{2}(target_i - output_i)x_i$$

$$\frac{\mathcal{L}(w, b)}{db} = \sum_i \frac{-1}{2}(target_i - output_i)$$

$$(w, b)_{t+1} = (w, b)_t - \lambda_t \frac{\mathcal{L}(w, b)}{d(w,b)}$$



1.5.2 Adam

Adam is derived from adaptive moment estimation. This algorithm is an extension of SGD to update the weights parameter during training. Unlike SGD maintains the same learning rate throughout training, Adam updates the learning rate for each weight individually.

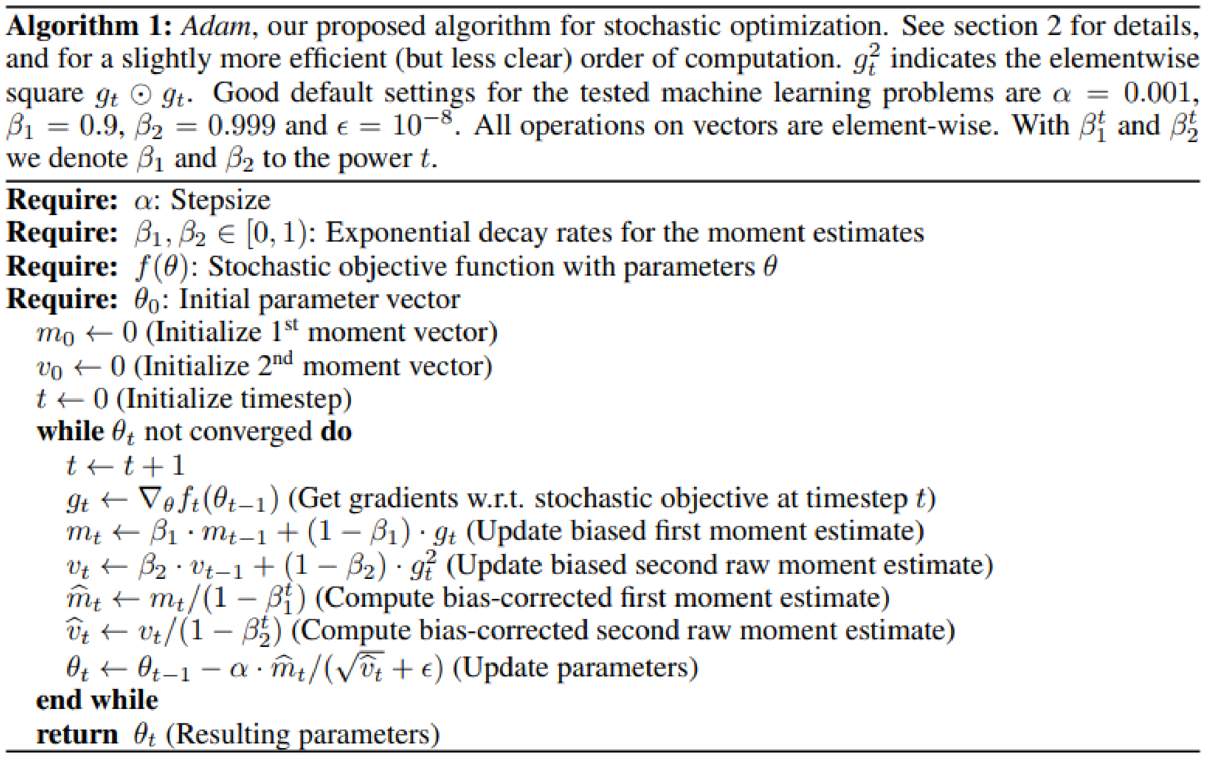
Suppose we have $f(\theta)$, a noisy Stochastic objective function with parameter θ. For t = {1, 2, ..., n}, $f_t(\theta)$ is the stochastic function at timesteps i. "The stochasticity might come from the evaluation at random subsamples (mini-batches) of data points, or arise from inherent function noise" (Kingma, Lei Ba 2015).

$g_t = \frac{df_t(\theta)}{d\theta}$ $-> vector\ of\ partial\ derivatives\ of\ f_t(\theta)\ w.r.t.\ \theta$

$m_t = moving\ averages\ of\ the\ gradient,$ a.k.a. mean

$v_t = the\ squared\ gradient,$ a.k.a. uncentered variance

$\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of m_t, v_t, respectively



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Adam algorithm pseudo code from Kingma and Lei Ba 2015 paper [6]

Related Work

1. Dropout & EarlyStopping:

1.1. CNN to predict patients' gender in FDG PET-CT:

Experiments were conducted using CNN to prevent patient misidentification in FDG-PET examination. Here, they designed a CNN architecture to predict patients' gender from FDG PET-CT images. Their CNN architecture contains 4-CNN layers, each CNN layer has a pooling layer and a ReLU activation function, local response normalization, and a softmax layer on the output layer. During the training phase, they used EarlyStopping to monitor the loss function of training, validation, and stop learning before training falls into excessive learning. They combined EarlyStopping with Dropout to prevent overfitting the data. For all of their experiments, their accuracy results in both training and testing achieve higher than 90%. [7]

1.2 Dropout to prevent Neural Networks from Overfitting

Another study used dropout to improve generalization performance on all data sets compared to when dropout was not used on neural networks. The datasets that they did experiments on were MNIST, TIMIT, CIFAR, SVHN, ImageNet, Reuters-RCV1, and Alternative Splicing dataset.

On their CIFAR-10, when training with no image augmentation and using Bayesian hyperparameters, they got an error rate of 14.98%. When adding dropout to the fully connected error, the error rate was reduced to 14.32%. They then add dropout in every layer and the error rate again improved and reduced to 12.61%. When using ReLU as activation, their error rate reduced again to 11.68%. [8]

2. Image Augmentation

2.1. Effectiveness of Image Augmentation in Image Classification

They designed an architecture they called SmallNet. See their model below.
SmallNet
1. Conv with 16 channels and 3x3 filters. Relu activations.
2. Batch normalization.
3. Max pooling with 2x2 filters and 2x2 stride.
4. Conv with 32 channels and 3x3 filters. Relu activations.
5. Conv with 32 channels and 3x3 filters. Relu activations.
6. Batch normalization.
7. Max pooling with 2x2 filters and 2x2 stride.
8. Fully connected with output dimension 1024. Dropout.
9. Fully connected layer with output dimension 2.

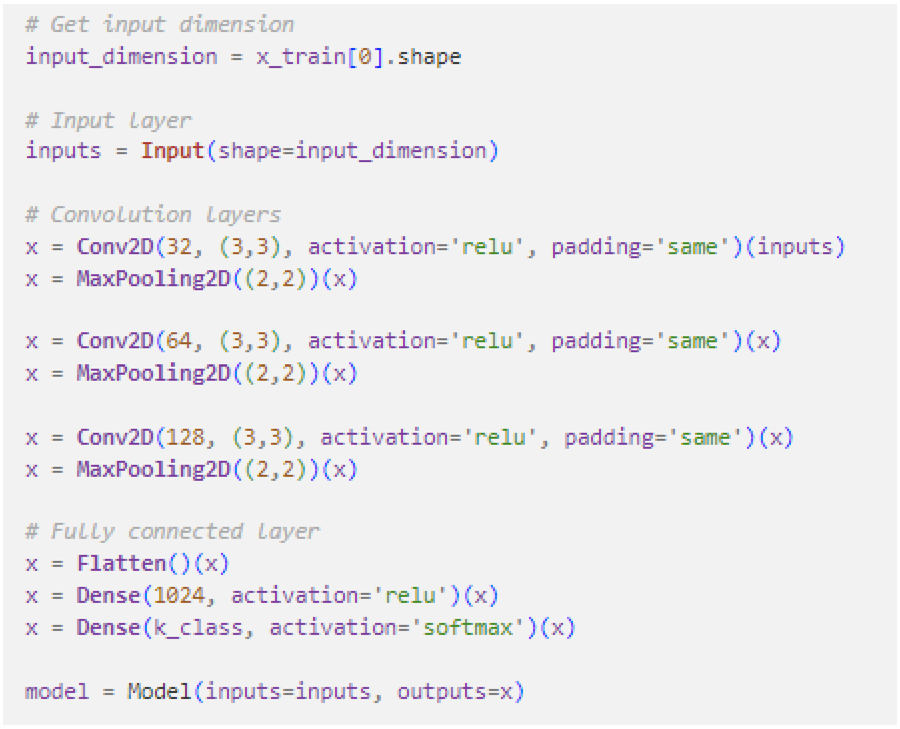
Augmentation network:
1. Conv with 16 channels and 3x3 filters. Relu activations.
2. Conv with 16 channels and 3x3 filters. Relu activations.
3. Conv with 16 channels and 3x3 filters. Relu activations.
4. Conv with 16 channels and 3x3 filters. Relu activations.
5. Conv with 3 channels and 3x3 filters.

All their experiment were run for 40 epochs at the learning rate of 0.0001 using Adam optimization. After running all experiments, they found that neural augmentation performs remarkably better than no augmentation. The neural augmentation got 91.5% compared to 85.5% with no augmentation. They had a control experiment to improve validation that used a more complicated net which prevents overfitting and allows for finer training. In this experiment, neural augmentation performs worse than no augmentation. Here they hypothesized that they might be “dealing with a larger net and are using the wrong learning rate” [9].

Models

1. Model 1

My first model without any hyperparameter tuning is a CNN with 3 convolution layers, each followed by a max pooling layer and a fully connected layer with the number of output classes. In this case, 10 classes for the CIFAR dataset.
The input layer takes the shape of the training image, (32,32,3) for the CIFAR dataset. For the three convolution layers, the first layer has 32 filters, the second layer has 64 filters, and the last layer has 128 filters. Each layer has a kernel size of (3,3) and uses ReLU as the activation function. Moreover, I also applied a max pooling layer of size (2,2) to reduce the output of each convolution layer. The flattened output of the third max pooling layer is then passed to a fully connected layer with 1024 units which uses ReLU as the activation function. The output layer has **k** number of classes and uses the softmax activation function.
The model is then compiled using the sparse categorical cross-entropy loss function, the Adam optimizer, and accuracy as the evaluation metric.
When fitting the model, the steps_per_epoch = training_size // batch_size.



```
# Get input dimension
input_dimension = x_train[0].shape

# Input layer
inputs = Input(shape=input_dimension)

# Convolution Layers
x = Conv2D(32, (3,3), activation='relu', padding='same')(inputs)
x = MaxPooling2D((2,2))(x)

x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(128, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

# Fully connected layer
x = Flatten()(x)
x = Dense(1024, activation='relu')(x)
x = Dense(k_class, activation='softmax')(x)

model = Model(inputs=inputs, outputs=x)
```

In addition to that, I also did two different test trials:

1. Only rescale image (1.1)
2. Add more image augmentation: horizontal and vertical flips, width and height shifts = 0.1 (1.2)

```
data_generator = ImageDataGenerator(horizontal_flip=True,
                                    vertical_flip=True,
                                    width_shift_range=0.1,
                                    height_shift_range=0.1)
```

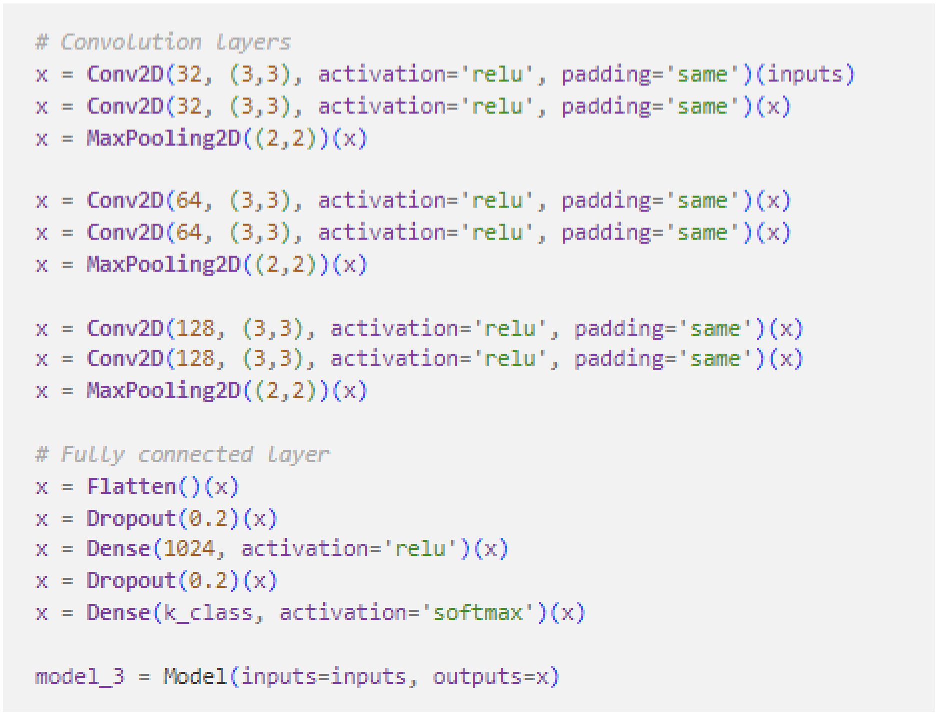
2. Model 2

Model 2 is similar to Model 1 in terms of activation function and using the max pooling layer after each layer. Though the main differences between Model 2 and Model 1 are that Model 2 has 6 convolution layers.

```
# Convolution Layers
x = Conv2D(32, (3,3), activation='relu', padding='same')(inputs)
x = Conv2D(32, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(128, (3,3), activation='relu', padding='same')(x)
x = Conv2D(128, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

# Fully connected Layer
x = Flatten()(x)
x = Dense(1024, activation='relu')(x)
x = Dense(k_class, activation='softmax')(x)

model_2 = Model(inputs=inputs, outputs=x)

model_2.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=['accuracy'])
```

3. Model 3

Model 3 is an upgrade from model 2. In Model 3, I added two Dropout layers after Flattening the Maxpool Layer. One Dropout layer after Flatten layer, and one after the Dense layer before our output layer.



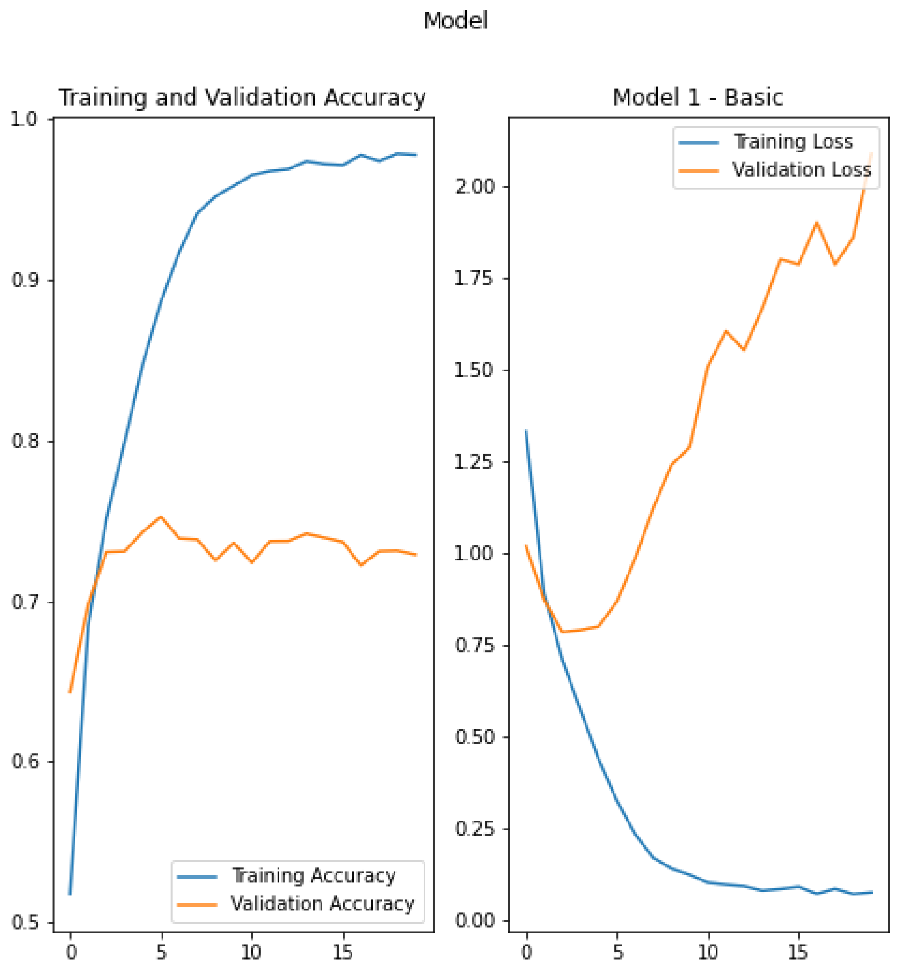
```
# Convolution layers
x = Conv2D(32, (3,3), activation='relu', padding='same')(inputs)
x = Conv2D(32, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
x = Conv2D(64, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

x = Conv2D(128, (3,3), activation='relu', padding='same')(x)
x = Conv2D(128, (3,3), activation='relu', padding='same')(x)
x = MaxPooling2D((2,2))(x)

# Fully connected layer
x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(k_class, activation='softmax')(x)

model_3 = Model(inputs=inputs, outputs=x)
```

Experimental Results

I first tried to run my model with 50 epochs, my kernel sometimes stops running half way, hence I decided to only use 20 epochs.

1. Model 1

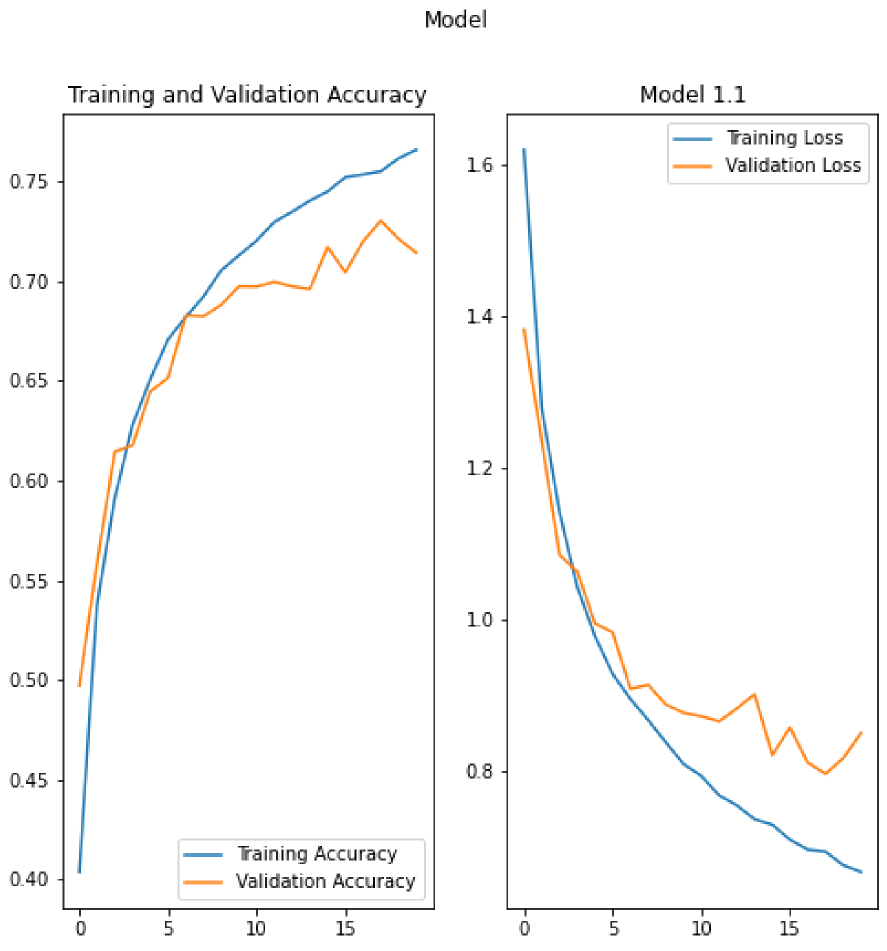
(1.1) Only rescale



```
Average Training accuracy: 89.30%
Average Validation accuracy: 72.62%
Average Loss: 0.30
Average Validation Loss: 1.37
```

While running the model on the CIFAR-10 dataset, looking at our accuracy and loss error curves, we can see that having 3 CNN layers with no Image Augmentation, we have such a huge overfitting problem here.

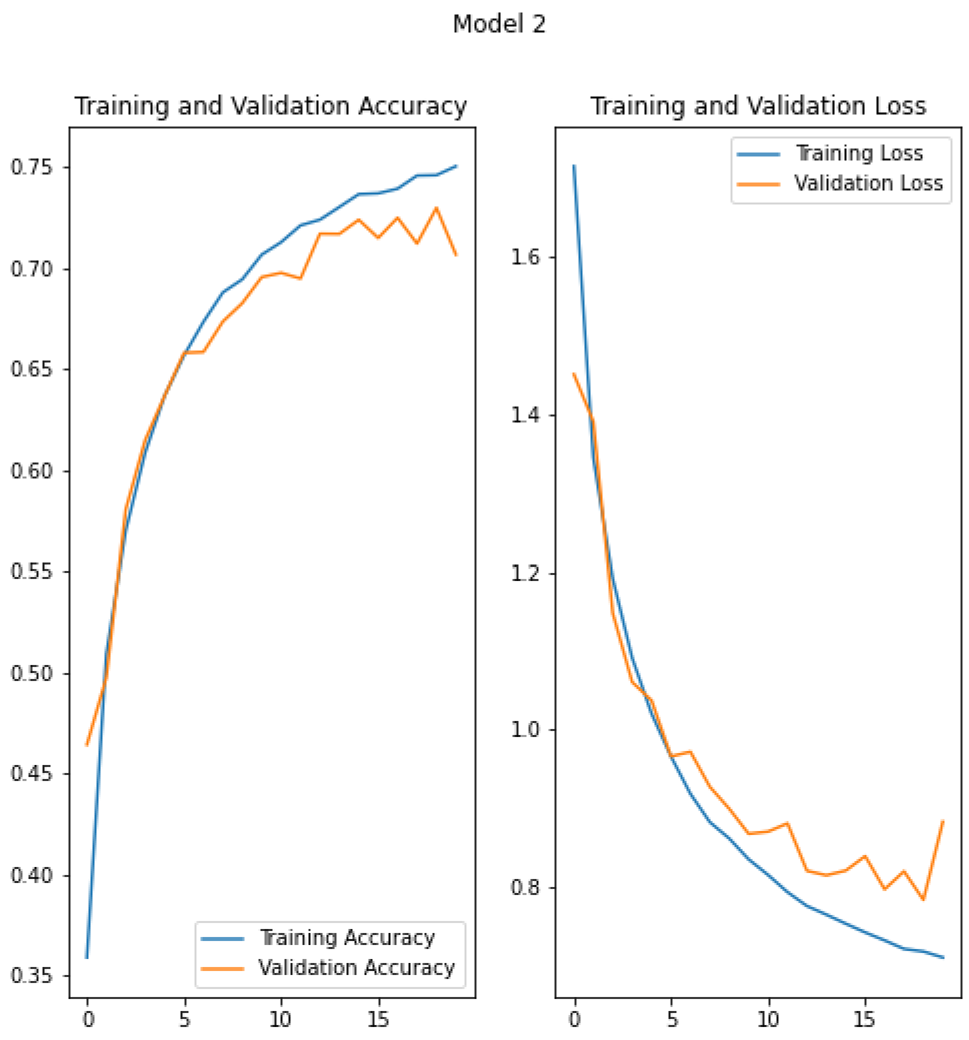
(1.2) - Image augmentation



```
Average Training accuracy: 72.56%
Average Validation accuracy: 69.57%
Average Loss: 0.79
Average Validation Loss: 0.89
```

When adding the Image Augmentation shown above, we can see that our Model was able to do a lot better than our original model. We see less overfitting, though there are still a lot of things we can still try to improve this model.

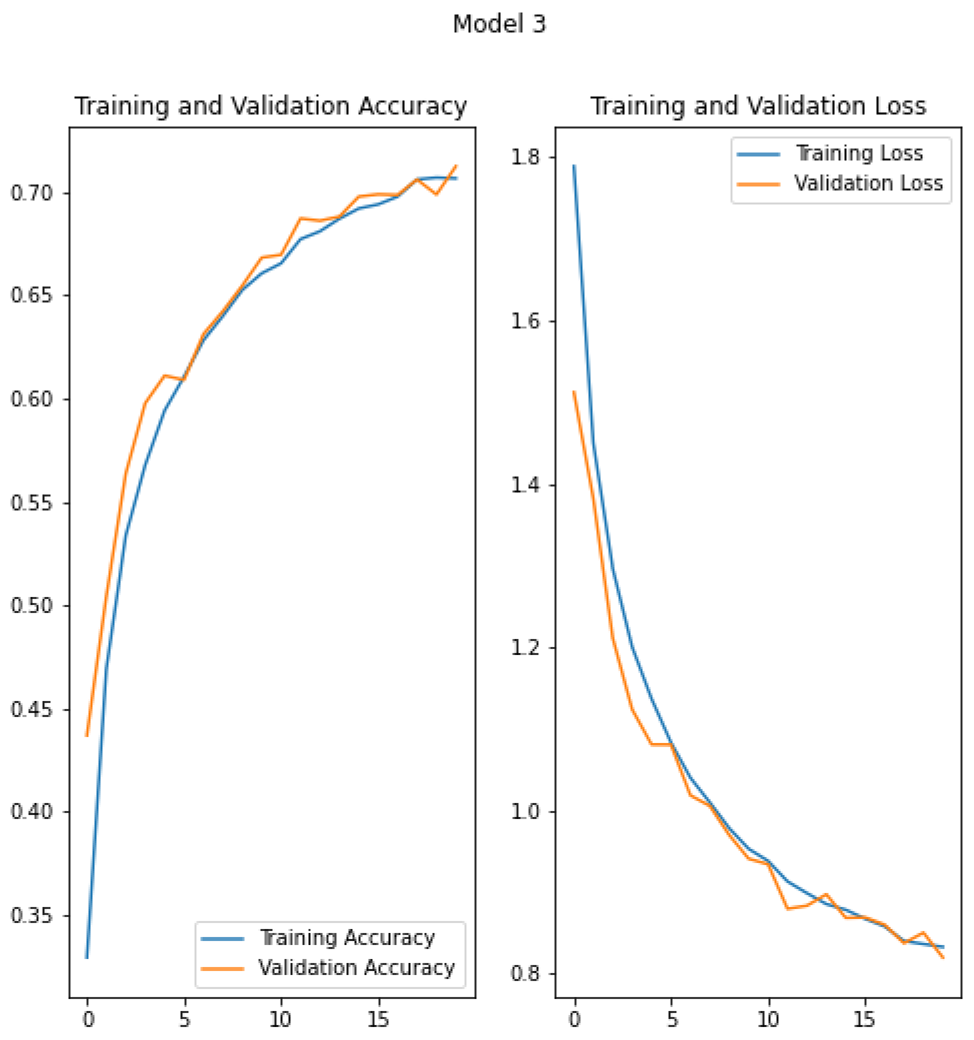
2. Model 2



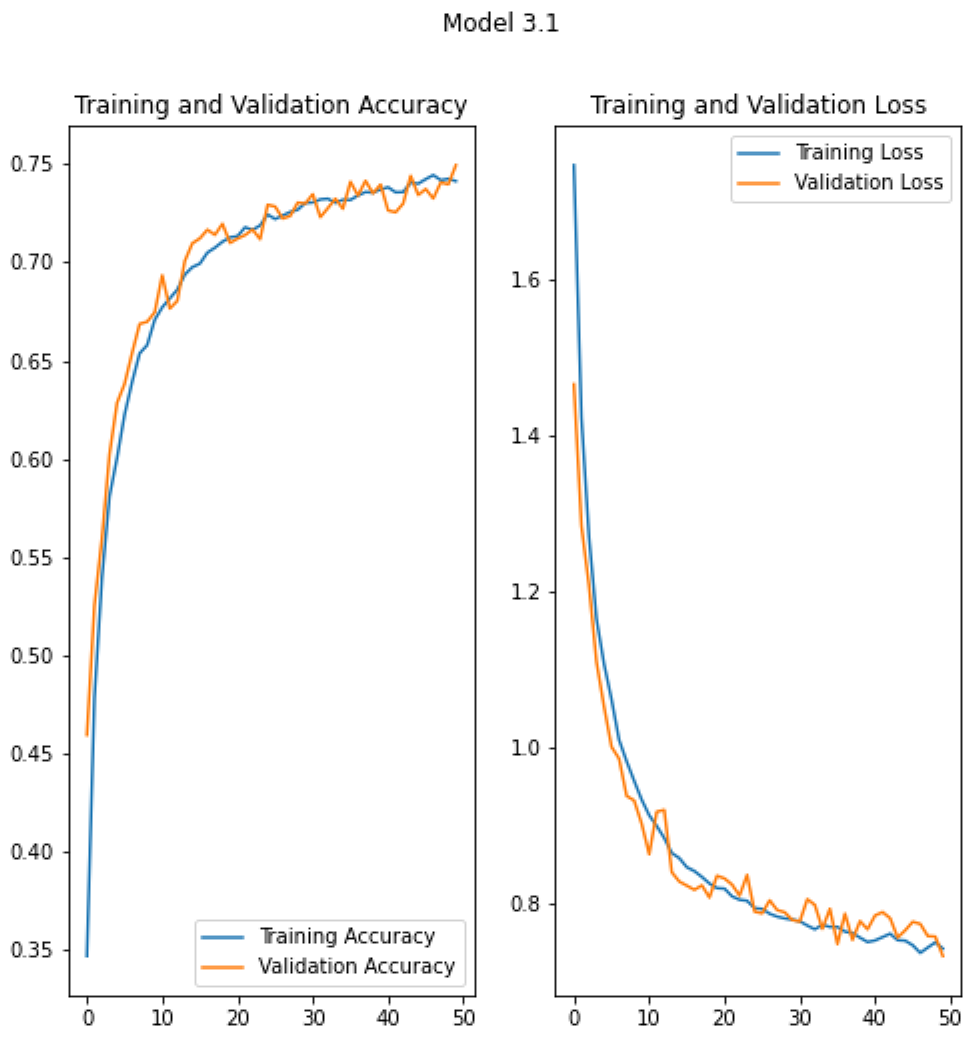
```
      Average Training accuracy: 67.22%
    Average Validation accuracy: 66.49%
             Average Loss: 0.92
       Average Validation Loss: 0.95
```

After adding 3 more layers, we can see a little improvement, but not so significant. Comparing this result to our previous results, we can see the validation accuracy is a little bit closer to training accuracy. We can see the same thing happening to our loss curves. Though again, not a significant change.

3. Model 3



```
Average Training accuracy: 63.01%
Average Validation accuracy: 64.32%
Average Loss: 1.03
Average Validation Loss: 1.00
```



```
Average Training accuracy: 69.43%
Average Validation accuracy: 70.06%
Average Loss: 0.87
Average Validation Loss: 0.86
```

Clearly looking at our Model 3 results, we can definitely see a huge improvement from our Model 2. Even though both of our graphs started with a little underfitting issue, we can see our Model was able to do better over time. Our validation accuracy and loss curves fluctuate a little but are still very similar to training accuracy and loss curves.

Looking further at 50 epochs, we can see our results a bit better. Comparing our Model 3 to its original/previous models, we can definitely conclude that by having deeper architecture and adding dropout layers definitely improve our learning and prevent overfitting.

Discussion

Looking at our experiment results above, we can safely say that incorporating image augmentation, and adding more hidden layers and dropout layers definitely improve the model and prevent overfitting.
There are still many ways to prevent overfitting. For instance, there is something called Callbacks that controls the training model. In Callbacks, we have the option to reduce the learning rate when accuracy does not improve after some number of steps. Another thing we can add to Callback to the EarlyStopping which stops the learning after some number of epochs if the validation loss error does not increase.
Another thing to discuss is that our loss error starts at a very high number. After 50 epochs, we can only get down to around 0.7-0.8 which is still very high. Maybe I can also explore more on how to improve loss errors so the accuracy might be higher.

References

[1] S. Albawi, S. Al-Zawi, T. A. Mohammed, Understanding of a convolution neural network, IEEE, 2017.

[2] J. Cao, Z. Su, L. Yu, D. Chang, X. Li, Z. Ma, Softmax Cross Entropy Loss with Unbiased Decision Boundary for Image Classification, In IEEE 2018 Chinese Automation Congress (CAC), 2018.

[3] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, Improving neural networks by preventing co-adaptation of feature detectors, Cornel University, 2012.

[4] K. Aggarwal, A guide to TensorFlow Callbacks, Paperspaceblog, 2020.

[5] A. Ajit, K. Acharya, A. Samanta, A Review of Convolutional Neural Networks, In IEEE 2020 International Conference on Emerging Trends in Information Technology and Engineering, 2020.

[6] D. P. Kingma, J. L. Ba, Adam: A Method for Stochastic Optimization, In 3rd International Conference for Learning Representations, 2015.

[7] K. Kawauchi, K. Hirata, C. Katoh, S. Ichikawa, O. Manabe, K. Kobayashi, S. Watanabe, S.Furuya, T. Shiga, A convolutional neural network-based system to prevent patient misidentification in FDG-PET examinations, National Library of Medicine, 2019.

[8] N. Srivastava, G. Hinton, A. Krizhevsky, R. Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, The Journal of Machine Learning ResearchVolume 15Issue 1pp 1929–1958, 2014.

[9] L. Perez, Ja. Wang, The Effectiveness of Data Augmentation in Image Classification using Deep Learning, Cornell University, 2017.

[10] S Kim, A Beginner's Guide to Convolutional Neural Networks (CNNs), Medium, 2019.