Convolutional Models for classification and segmentation

Anirudh Singh Sisodia

F215611

A.S.Sisodia-22@student.lboro.ac.uk

Contents

Abstract

Convolutional neural networks(CNN) provide mechanism to learn features of visual data for example pictures, videos etc. These features may include edges, shapes, facial features etc. In this report I am presenting 2 CNNs that I created where one is to **classify** and the other is to **segment** images with the help of training datasets provided by Prof. Baihua Li. These models will be a combination of CNN layers with Normalisation, MaxPool and dropouts etc. I have tried multiple architectures and hyper parameter combinations to get the best accuracy and the least overfitting I could. I have also tried a way to start designing network architecture to build intuitions of how they behave with change in hyperparameters. The semantic

segmentation CNN is inspired by U-NET architecture that explains reusing existing architecture is a good idea for better results .The main focus is on building intuition of building a model and playing with the hyperparameters so I have tried to plot accuracy, val_accuracy and loss, val_loss on a graph to see the trends of how these are behaving while a model is being trained.

Introduction

In CNN architectures there a few decisions that we have to take to build a good performing model. These are given below.

- No. of layers
- No. of hidden units in each layer
- Learning rate
- Optimiser
- Call-backs
- ……..

And so on.

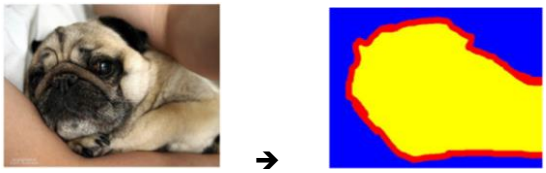
We are going to test and try different such decisions in our problem statements that we have chosen for this project. The first one is to segment 24 categories of dogs images using semantic segmentation techniques and the other is to classifying images among 8 classes.

The whole process of designing a Neural Network can be divided in subtasks

- Data Visualization
- Data Pre-Processing
- Model Selection/Configuration
- Training, Evaluation
- Result Visualization and Comparison

CNN Model for Segmentation

The dataset provides images in 25 categories of dogs, each category has ~200 images, and image size varies e.g. ~ 400x400 pixels. We have to do a semantic segmentation of an image like this



Data Visualisation

In this section I wrote a piece of code to extract files from the zip file provided as a dataset(dog_dataset) for segmentation. Then I loaded the images and their masks in different variables names train_imgs and train_masks and resized everything to 96 X 96 to match the model input dimensions.

Then I randomly picked images from train_imgs and train_masks and plot them next to each other.

Data Pre-Processing

In this section I normalise the pixel information by dividing train_imgs(which had pixel values from 0 to 255) by 255 to get values ranging from 0 to 1 and For train_masks(which had values of 1,2,3) I subtracted 1 to get values 0,1 and 2.

Model Selection/Configuration

I took to the internet to see various successful architectures in segmentation and arrived on U-Net. I chose U-net and thought of implement my own version of U-Net for dog_dataset. The changes that I made to this architecture was reduce the size of input from 570 X 570 to 96 X 96 and output was having 2 channels which was able to do a grayscale segmentation but I made it do a 3 channel segmentation like the training data. In my architecture I used 5 blocks of (conv1,dropout,batchNormalisation,conv2,maxpool) in contracting path and 3 blocks of (convtranspose,skip,conv1,dropout,conv2) in expanding path with output layer as one conv with 3 filters and activation as SoftMax. I decided to train the model for 10 epochs with a batch size of 100 and with 2 call backs(save best model and adaptive learning rate for accuracy). Final(Model 3) Pipeline given below. Getting to the final model was an iterative process where I tried many different variations to get good results. Three(Model 1,2 and 3) of those variations are explained below.

FINAL Model 3

Layer (type)	Output Shape	Param #	Connected to
input_45 (InputLayer)	[(None, 96, 96, 3)]	0	[]
conv2d_956 (Conv2D)	(None, 96, 96, 16)	448	['input_45[0][0]']
dropout_404 (Dropout)	(None, 96, 96, 16)	0	['conv2d_956[0][0]']
batch_normalization_200	(None, 96, 96, 16)	64	['dropout_404[0][0]']
conv2d_957 (Conv2D)	(None, 96, 96, 16)	2320	['batch_normalization_200[0][0]']
max_pooling2d_185	(None, 48, 48, 16)	0	['conv2d_957[0][0]']
conv2d_958 (Conv2D)	(None, 48, 48, 32)	4640	['max_pooling2d_185[0][0]']
dropout_405 (Dropout)	(None, 48, 48, 32)	0	['conv2d_958[0][0]']
batch_normalization_201	(None, 48, 48, 32)	128	['dropout_405[0][0]']
conv2d_959 (Conv2D)	(None, 48, 48, 32)	9248	['batch_normalization_201[0][0]']
max_pooling2d_186	(None, 24, 24, 32)	0	['conv2d_959[0][0]']
conv2d_960 (Conv2D)	(None, 24, 24, 64)	18496	['max_pooling2d_186[0][0]']
dropout_406 (Dropout)	(None, 24, 24, 64)	0	['conv2d_960[0][0]']

```
batch_normalization_202        (None, 24, 24, 64)   256        ['dropout_406[0][0]']
conv2d_961 (Conv2D)            (None, 24, 24, 64)   36928      ['batch_normalization_202[0][0]']
max_pooling2d_187              (None, 12, 12, 64)   0          ['conv2d_961[0][0]']
conv2d_962 (Conv2D)            (None, 12, 12, 128)  73856      ['max_pooling2d_187[0][0]']
dropout_407 (Dropout)          (None, 12, 12, 128)  0          ['conv2d_962[0][0]']
batch_normalization_203        (None, 12, 12, 128)  512        ['dropout_407[0][0]']
conv2d_963 (Conv2D)            (None, 12, 12, 128)  147584     ['batch_normalization_203[0][0]']
max_pooling2d_188              (None, 6, 6, 128)    0          ['conv2d_963[0][0]']
conv2d_964 (Conv2D)            (None, 6, 6, 256)    295168     ['max_pooling2d_188[0][0]']
dropout_408 (Dropout)          (None, 6, 6, 256)    0          ['conv2d_964[0][0]']
batch_normalization_204        (None, 6, 6, 256)    1024       ['dropout_408[0][0]']
conv2d_965 (Conv2D)            (None, 6, 6, 256)    590080     ['batch_normalization_204[0][0]']
conv2d_transpose_214           (None, 12, 12, 128)  131200     ['conv2d_965[0][0]']
concatenate_176 (Concatenate)  (None, 12, 12, 256)  0          ['conv2d_transpose_214[0][0]'
                                                                 'conv2d_963[0][0]']
conv2d_966 (Conv2D)            (None, 12, 12, 128)  295040     ['concatenate_176[0][0]']
dropout_409 (Dropout)          (None, 12, 12, 128)  0          ['conv2d_966[0][0]']
conv2d_967 (Conv2D)            (None, 12, 12, 128)  147584     ['dropout_409[0][0]']
conv2d_transpose_215           (None, 24, 24, 64)   32832       ['conv2d_967[0][0]']
concatenate_177 (Concatenate)  (None, 24, 24, 128)  0          ['conv2d_transpose_215[0][0]'
                                                                 ,'conv2d_961[0][0]']
conv2d_968 (Conv2D)            (None, 24, 24, 64)   73792      ['concatenate_177[0][0]']
dropout_410 (Dropout)          (None, 24, 24, 64)   0          ['conv2d_968[0][0]']
conv2d_969 (Conv2D)            (None, 24, 24, 64)   36928      ['dropout_410[0][0]']
conv2d_transpose_216           (None, 48, 48, 32)   8224       ['conv2d_969[0][0]']
concatenate_178 (Concatenate)  (None, 48, 48, 64)   0          ['conv2d_transpose_216[0][0]'
                                                                 ,'conv2d_959[0][0]']
conv2d_970 (Conv2D)            (None, 48, 48, 32)   18464      ['concatenate_178[0][0]']
dropout_411 (Dropout)          (None, 48, 48, 32)   0          ['conv2d_970[0][0]']
conv2d_971 (Conv2D)            (None, 48, 48, 32)   9248       ['dropout_411[0][0]']
conv2d_transpose_217(Conv2DTr  (None, 96, 96, 16)   2064       ['conv2d_971[0][0]']
concatenate_179 (Concatenate)  (None, 96, 96, 32)   0          ['conv2d_transpose_217[0][0]'
                                                                 ,'conv2d_957[0][0]']
conv2d_972 (Conv2D)            (None, 96, 96, 16)   4624       ['concatenate_179[0][0]']
dropout_412 (Dropout)          (None, 96, 96, 16)   0          ['conv2d_972[0][0]']
conv2d_973 (Conv2D)            (None, 96, 96, 16)   2320       ['dropout_412[0][0]']
conv2d_974 (Conv2D)            (None, 96, 96, 3)    51         ['conv2d_973[0][0]']
```

Training, Evaluation

I trained the many models by adding/Changing layers like dropouts, batch Normalisation, Max Pool etc. The best Model that I saved was having ~ 85% accuracy on test data.

Compile Model

I compiled the model by using Adam as an **optimiser** with a **learning rate** of 0.001, a **loss** of sparse_categorical_crossentropy and a **metric** of accuracy. This combination of hyperparameters I got when I researched on the internet.

batch_size=100,epochs=10,validation_split=0.1

Call backs

I added 2 call backs for this model.

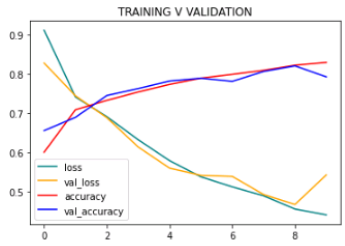
1. **Model Checkpoint**: To save the best model in a .h5 file in terms of accuracy
2. **ReduceLROnPlateau**: To reduce the learning rate if the accuracy stays the same for patience of 3.

Model 1:

From the traditional U-NET I just added a dropout layer in each block of contracting and expanding layer

Training accuracy of ~ 82% and test accuracy of ~ 78%

```
model.evaluate(test_imgs,test_maps)
41/41 [==============================] - 3s 73ms/step - loss: 0.5595 - accuracy: 0.7885
[0.5595049262046814, 0.7885467410087585]
```

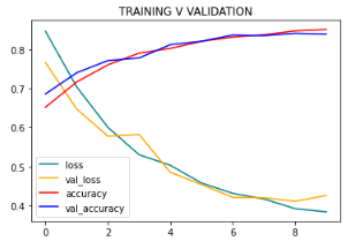


Model 2:

Then I reduced the batch size to 50 keeping rest of components same to see if the batch size makes any difference. This was the result which shows some reduction in overfitting but there is still some overfitting left when I tested the model on test data as it was low as compared to training accuracy.

Training accuracy of ~ 85% and test accuracy of ~ 83%

```
model.evaluate(test_imgs,test_maps)
41/41 [==============================] - 4s 87ms/step - loss: 0.4391 - accuracy: 0.8364
[0.43914517760276794, 0.8363869190216064]
```



Overfitting

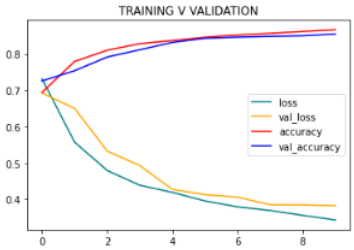
For the first 2 Models I was facing the problem of overfitting even after introducing a dropout layer in each block so in the Model 3 I added another layer of batch Normalisation

Model 3:

Then added batch normalisation layer after the first conv layer in the contracting path. This was the best model so far and gave good results for both accuracy and loss as we can see. The overfitting is also quite low now as the accuracy on test data was almost equal to training accuracy. This was the best model I got.

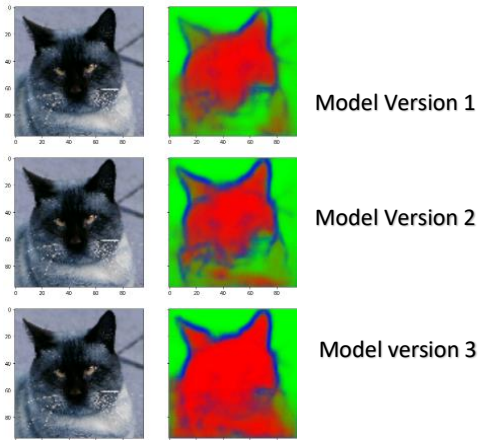
Training accuracy of ~ 86% and test accuracy of ~ 85%

```
model.evaluate(test_imgs,test_maps)
41/41 [==============================] - 4s 91ms/step - loss: 0.3920 - accuracy: 0.8521
[0.3919985890388489, 0.8520660996437073]
```

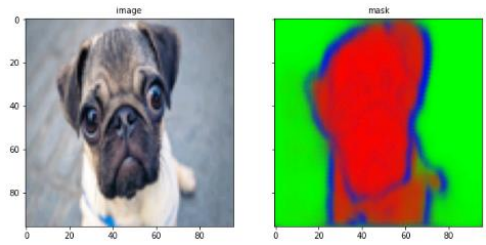


Result Visualization and Comparison

I plotted some images next to their predicted masks on the loaded model that was saved during training and got the below results.



I also tested the model on a new image downloaded from the internet and got these results.



CNN Model for Classification

Data Visualisation

In this step I loaded the data and split it to train and test variables using train_test_split function in sklearn module and try to visualise the data in the form of tables(in case of numerical data) or images (in case of visual data). Dataset provided to us for image classification is in the form of a file named object_recognition_dataset.pth I download this file from the cloud and extracted the folders containing training and testing images along with labels provided to those images to train_imgs, train_labels and test_imgs, test_labels variables

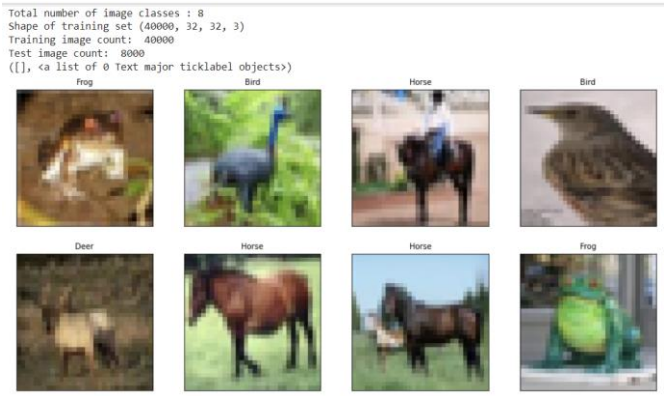
We often split the data between train set and test set, one for training and one for test the model.

Before training I scooped out a small chunk of data from the training set for validation(set used in training to evaluate in terms of accuracy if our model is training well). It is also called a validation set which is used while and after training for pre-testing before the real testing of the model.

I used 10% of our training data as a validation set.

I then plotted the first few images in our training dataset to visualise the data I have. In this case I have 40000 colour images of size 32 X 32 X 3 in our training set and 8000 images in our testing set split among **8** classes.

```
["Airplane","Automobile","Bird","Cat","Deer","Dog","Frog","Horse"]
```

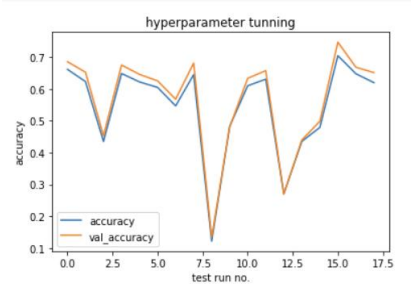


Data Pre-Processing

In This step I tried to normalise the data so that pixel values in the images range from -1 to 1 to get better results and after the multiple iteration of calculations. I did this by dividing the whole set by 255.

Model Selection/Configuration

I decided to first think of the number of layers and number of neurons/filter in each layer. For this I ran a **piece of code**(in the notebook) to train(for 5 epochs) multiple permutations and combinations of these 2 hyperparameters and plot the accuracies to see which combinations worked the best. First, I took a range of 1 to 5 CNN layers and 2 dense layers with a range of 1 to 64 filters in each CNN layer and 8 neurons in each dense layer(adding 4 filters after each epoch in the CNN layer) it performed well for layers 3 to 5 and for number of filters with a range of filters 45 to 64. Then I ran another test with just 3-5 layers and 45-64 filters and below is the result for that.



I found that test run no. 15 worked the best which used 5 layers with 58 neurons in each layers and accuracy, val_accuracy came out to be around 75% and quite close to each other. So I decided to go with 5 layers. After some test runs on 5 layers and changing some hyperparameters I arrived on the best performing architecture which can be seen below.

Final Model

Layer (type)	Output Shape	Param #

```
conv2d_980 (Conv2D)          (None, 32, 32, 58)        1624

conv2d_981 (Conv2D)          (None, 32, 32, 58)        30334

conv2d_982 (Conv2D)          (None, 32, 32, 58)        30334

conv2d_983 (Conv2D)          (None, 32, 32, 58)        30334

conv2d_984 (Conv2D)          (None, 32, 32, 58)        30334

max_pooling2d_190            (None, 16, 16, 58)        0

dropout_413 (Dropout)        (None, 16, 16, 58)        0

flatten_7 (Flatten)          (None, 14848)             0

dense_14 (Dense)             (None, 32)                475168

dense_15 (Dense)             (None, 8)                 264

=================================================================
Total params: 598,392
Trainable params: 598,392
Non-trainable params: 0
_________________________________________________________________
```

Training, Evaluation

In this step I added some callbacks for the model compiled the model after deciding some more hyperparameters and then trained it on the training set and also gave the validation split for training. Model gave an accuracy of 75% on training data.

Callbacks

I added 2 call backs for this model.

3. **ModelCheckpoint**: To save the best model in a .h5 file in terms of accuracy
4. **ReduceLROnPlateau**: To reduce the learning rate if the accuracy stays the same for patience of 3.

Compile Model

I compiled the model by using Adam as an **optimiser** with a **learning rate** of 0.001, a **loss** of sparse_categorical_crossentropy and a **metric** of accuracy. This combination of hyperparameters I got when I researched on the internet.

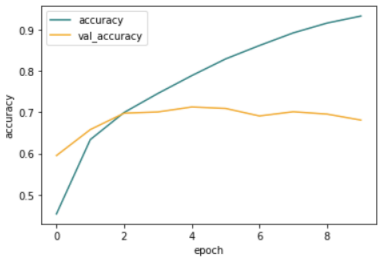
Training

I trained many variations of this model by changing the hyperparameters and found these 2 the best ones to showcase in the report with the best accuracy of ~ 72%

Model 1:

batch_size=50,epochs=10,validation_split=0.1

You can see below the epochs, loss varying as the training progressed. This was showing overfitting as training and validation accuracy were behaving quite differently.

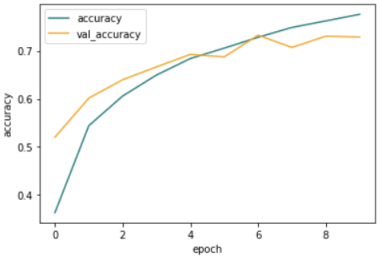


Overfitting

First model showed overfitting as we can see the accuracy after each epoch was steadily increasing but the validation accuracy was fluctuation and there was a big difference between validation and training accuracy so evidently our model is overfitting.

Model 2:

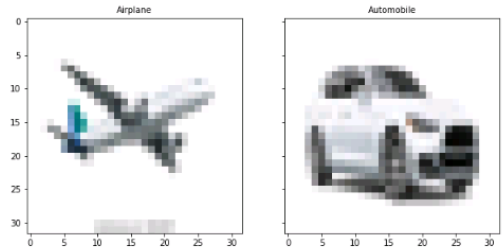
For this we can regularise the model a bit so that it does not train the weights too much to fit the training data. For this we have dropout layers in NN. Let's add one and see. After adding dropout with a rate of 0.5 the performance improved a bit.



Result Visualization and Comparison

It gave an accuracy of ~ 72% on test data.

```
model.evaluate(test_imgs,test_labels)
250/250 [==============================] - 4s 16ms/step - loss: 0.8228 - accuracy: 0.7181
[0.8228397965431213, 0.7181249856948853]
```



References

- https://www.youtube.com/watch?v=fTw3K8D5xDs&ab_channel=LeoIsikdogan
- https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594
- https://www.youtube.com/watch?v=C1N_PDHuJ6Q&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc&index=4&ab_channel=DeepLearningAI
- https://www.youtube.com/watch?v=RaswBvMnFxk&list=PLZsOBAyNTZwbR08R959iCvYT3qzhxvGOE&index=6&ab_channel=DigitalSreeni
- https://www.youtube.com/watch?v=1waHlpKiNyY&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc
- https://github.com/bnsreenu/python_for_microscopists/blob/master/231_234_BraTa2020_Unet_segmentation
- https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6