CS 577 - F22 - Assignment 4 Programming Portion

Due date: 11/10/2022

Anas Puthawala - A20416308
Professor Gady Agam

Binary Classification

Data Description

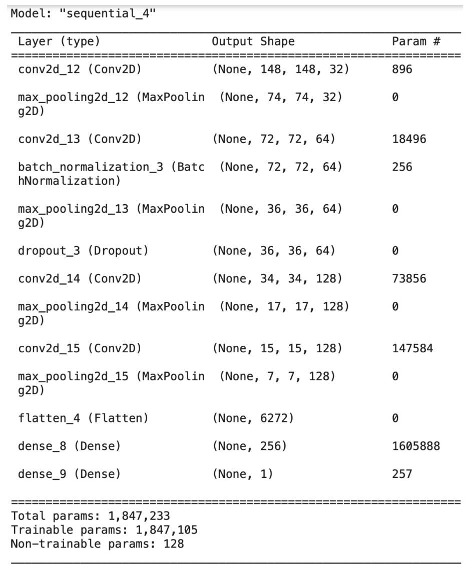
We're using Kaggle's Cats and Dogs dataset, and we're only using 2000 random cats and 2000 random dogs as our entire data set from which we must split and get training, validation and test data. The data is downloaded into my google colab enviroment, unzipped, and split into the respective folders for train, test, and validation (i.e. each folder will have two subfolders, cats and dogs, containing the respective amounts of data). I utilized a 70, 15, 15 split for train, val, and test sets.

Problem Statement & Proposed Solution

For binary classification the problem statement is that we're aiming to utilize train, test, and validation generators, build out a CNN from scratch, visualize activations of said CNNs' layers and filters learned in training, and lastly familiarize ourselves with the process of using pre-trained models. Specifically, we use the base of VGG-16 and fine-tune the model with our 2000 samples from each class, and evaluate the performance and compare it to our naive CNN model. Lastly, we perform data augmentation and re-train with the frozen convolution base from VGG-16 and evaluate the performance.
Our solution consists of using the convolution base from VGG-16 in order to improve our performance. Our performance is initially not as optimal since we only have around 2000 samples from each class to learn from.

Implementation Details

The main problem we quickly faced when training with the CNN is that the models performance was pretty poor. It was around 60% accuracy with reasonably high loss which made it an unfit model for the task at hand. These problems were resolved when I used pre-trained models to help boost performance. The model used for the naive CNN implementation had the following architecture:



```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_12 (Conv2D)          (None, 148, 148, 32)      896

 max_pooling2d_12 (MaxPoolin  (None, 74, 74, 32)       0
 g2D)

 conv2d_13 (Conv2D)          (None, 72, 72, 64)        18496

 batch_normalization_3 (Batc  (None, 72, 72, 64)       256
 hNormalization)

 max_pooling2d_13 (MaxPoolin  (None, 36, 36, 64)       0
 g2D)

 dropout_3 (Dropout)         (None, 36, 36, 64)        0

 conv2d_14 (Conv2D)          (None, 34, 34, 128)       73856

 max_pooling2d_14 (MaxPoolin  (None, 17, 17, 128)      0
 g2D)

 conv2d_15 (Conv2D)          (None, 15, 15, 128)       147584

 max_pooling2d_15 (MaxPoolin  (None, 7, 7, 128)        0
 g2D)

 flatten_4 (Flatten)         (None, 6272)              0

 dense_8 (Dense)             (None, 256)               1605888

 dense_9 (Dense)             (None, 1)                 257

=================================================================
Total params: 1,847,233
Trainable params: 1,847,105
Non-trainable params: 128
_________________________________________________________________
```

Fig. 1 - Model architecture | Naive CNN

Visualizing activations & filters

When visualizing the filters and activations, some problems faced were that the activations didn't really show anything that important in the initial layers. For example, the entire image was activated, that doesn't really mean much so I dug a bit deeper and went into the deeper layers and that began to show some interesting features such as activation around the eyes of a cat & ears of a cat. Although, it was also activating around the edges of a wall, this could be interesting because it might suggest something like in the dataset most pictures of cats had walls around them (i.e. the pictures of cats were mostly taken indoors as opposed to dogs).
The process of visualizing activations & filters is as such:

1. Select outputs of top *n* layers
2. Create a new model, the inputs of the model are the same as the previous model's inputs, the outputs however are the outputs from the top *n* layers
3. Run the new model on the image (predict) and save the activations
4. And you can examine the activations from each layer by indexing and plotting using plt.matshow

The activations show information from specific layers and what specifically those layers got 'activated' on for a given image.

As for the filters, I was unable to get it to work but here is the implementation details:
I ran into an issue here so I am unable to move forward with the task of visualizing trained filters to see what will maximize their response. But here is how I WOULD continue if I didn't get this error:

1. Get the gradients, compute the loss (from above)
2. Instantiate a Keras function w/ inputs (model.input) and outputs (loss, grads)
3. Run the computation graph from inputs to outputs with an input image of zeros
4. Maximize the response measure wrt input using gradient ascent. So we start with a gray image with some noise
Can instantiate the gray image as such: np.random.random((1, img_dim, img_dim, 3)) * 20 + 128) have a step size and then run gradient ascent for 50 - 100 iterations where-by I evaluate the sub-graph loss and gradients for the current input and update the image using the gradient and step size. The image would get updated by adding the gradient value and multiplying by the step.

5. And then you can 'deprocess' the image by scaling to 0 mean, unit std dev., clipping to 0,1 and converting vals between 0,255
6. lastly just plot it to visualize using plt.imshow()

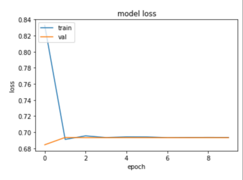
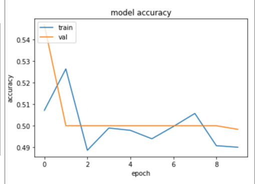
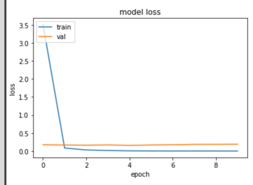
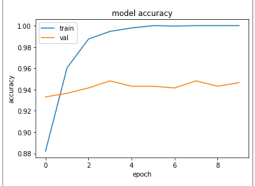
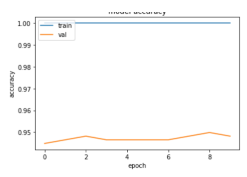
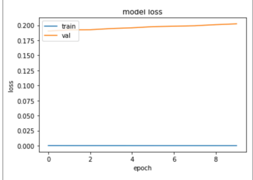
The image shows what maximizes the output for a specific given layer (i.e. what excites the neurons the most) it may be something as benign as a bunch of random lines or something interesting like the shape of a cats ears.

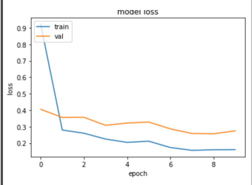
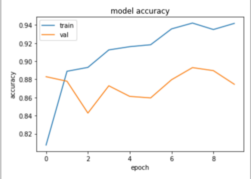
Using pre-trained models

As for using pre-trained models, there are two avenues you can take:
1. Just use the pre-trained base, stick a classification head that is fit for the task (i.e. in this case it was a dense unit with sigmoid activation), **freeze the pre-trained base**, and train the classification head.
2. After you do that, you can decide to fine-tune the pre-trained base by unfreezing the base and re-training on the training set so the gradients flow through the entire network and update the weights accordingly.

Results & Discussion

Note that in all cases, an Adam optimizer was used with BCE loss. The epochs, LR, and Batch size were modified as part of hyper-parameter tuning process. The model was evaluated with 'Accuracy' as metric.

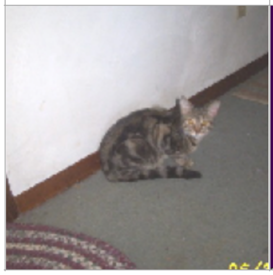
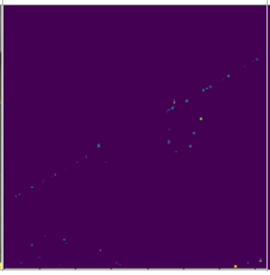
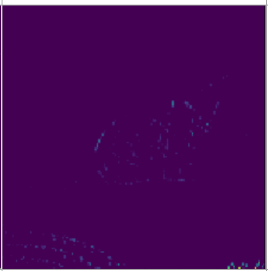
Model Type	Loss Curve	Activation Curve	Lowest Loss	Best Accuracy & Params
Naive CNN			Loss = .694	Acc = 0.503, epochs=10, LR = 0.00085,
VGG base CNN (frozen)			0.258	Acc = .955 epochs = 10, LR = .0008,
VGG base CNN (fine-tuned)			0.318	Acc = .933, epochs = 10, LR = .0008

Model Type	**Loss Curve**	**Activation Curve**	**Lowest Loss**	**Best Accuracy & Params**
VGG base CNN (fine-tuned + data augmentation)			0.685	Acc = .926, epochs = 10, LR = .0008

The best results were from the case where we just use the frozen VGG base and tune only the classifier head. We still get great performance when we use pre-trained models (we got an improvement from like .503 accuracy to nearly .95 and .93) which is big.

Visualizing activations

I selected the top 8 layers and found sufficient results (visualizations) from the first conv layer, 3rd and 7th activation layers:

Initial Image	Activations from 3rd layer	Activation from 7th layer
		

If you zoom in, you can be discern how the third layer isn't really picking up on anything that interesting except the wall. That's interesting because it may signify some characteristic such as most cats are indoors and have a wall near them. The activation from the 7th layer is interesting because it is very faintly beginning to show signs of activating around the actual figure of the cat, like the roundness of the animal and the eyes and the ears.

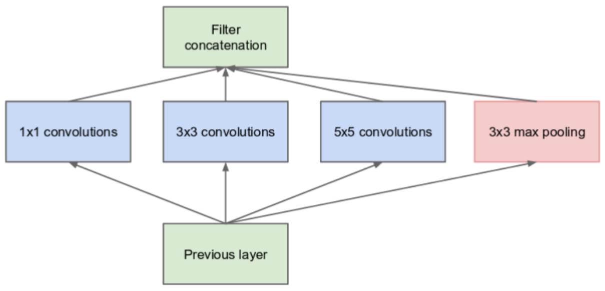
Multi-class Classification

Data Description

The data we are dealing with is the CIFAR-10 data. CIFAR-10 data consists of 60,000 images of size 32x32 in 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). I obtain this data in my colab environment, and set up the train, test, val amounts to be 45,000, 7,500 and 7,500 respectively. Additionally, reshape the image vectors and divide to standardize by 255. The training, test, val split is 75, 15, 15. We're not using a generator from directories in this method so no need to make and store all the data in the different directories. Additional care was taken to convert the integer labels (y_train, y_test, and y_val) to a OHE class matrix.

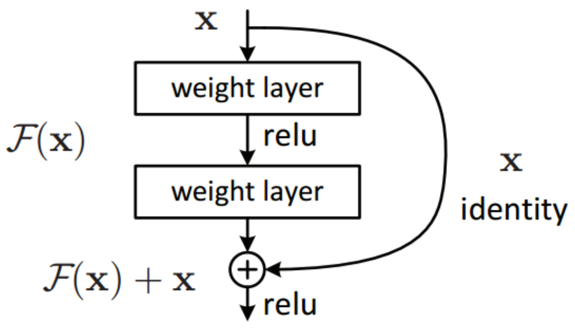
Problem Statement & Proposed Solution

The problem statement is that we want to observe the effects of adding in inception blocks and residual blocks to an initial naive CNN network for multi-class classification. There are many different types of inception blocks. For this assignment, I manually built the 'naive' inception block as seen in figure (a) on the right. The idea is that the inception block allows for the usage of multiple types of filter sizes, instead of being restricted to a single



(a) Inception module, naïve version

filter size (conventional CNN), in a single image block, which we then are able to concatenate and pass onto the next layer [1]. As for the residual block, it features what is called skip connections which were pivotal in the sense that they allowed information to be transitioned throughout the architecture and allowed for architectures to get pretty large and deeper [2]. The figure on the right shows an implementation of a simple residual block containing a skip connection.

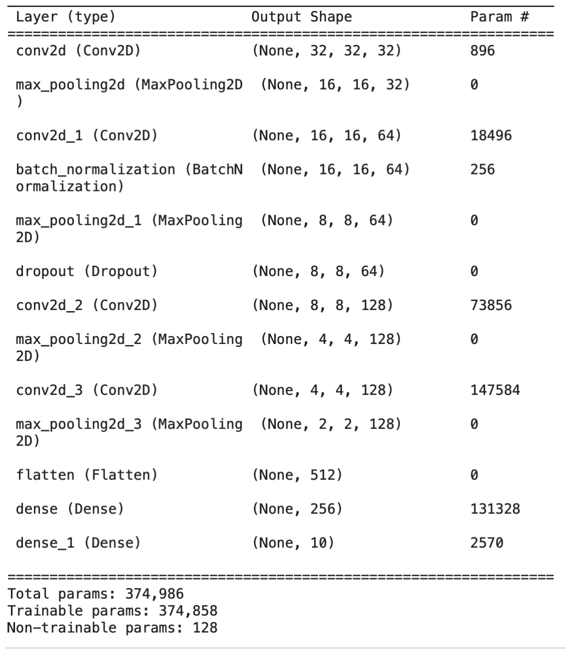


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

(b) residual block w/ skip connection

Implementation Details

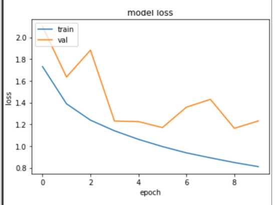
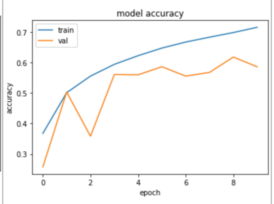
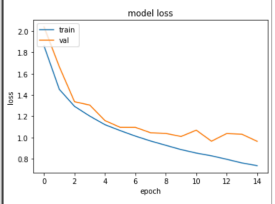
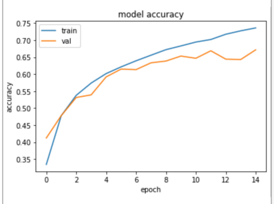
We start off by developing a naive CNN architecture and evaluating the performance of the model on the CIFAR-10 dataset. Next we add a single inception block in the mix and evaluate the performance of the model on the CIFAR-10 dataset. Lastly, we remove the inception block and add a single inception block and again, evaluate the performance of the model on the CIFAR-10 dataset. The model parameters that were held constant are the optimizer (Adam) and the loss of categorical crossentropy. The models parameters that were adjusted (hyper-parameter tuning) were epochs, learning rate, and the batch size. Note that when we were dealing with skip connections for the residual block, we are summing a previous part of the network to a later part and if that part is a matrix then the dimensions must match. For this reason, additional care was taken to ensure the dimensions match up before adding the two points. The categorical accuracy was considered as the metric to track for performance as-well as the loss. The naive CNN network built is the shown on the right side:

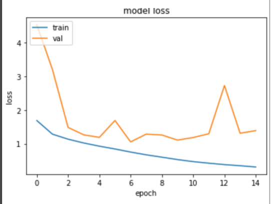
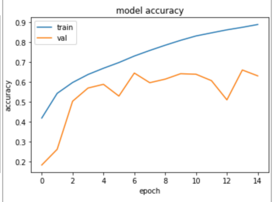


```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 16, 16, 64)        18496

 batch_normalization (BatchN  (None, 16, 16, 64)       256
 ormalization)

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 dropout (Dropout)           (None, 8, 8, 64)          0

 conv2d_2 (Conv2D)           (None, 8, 8, 128)         73856

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 conv2d_3 (Conv2D)           (None, 4, 4, 128)         147584

 max_pooling2d_3 (MaxPooling  (None, 2, 2, 128)        0
 2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 256)               131328

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 374,986
Trainable params: 374,858
Non-trainable params: 128
```

Fig. 2 - Naive CNN architecture used

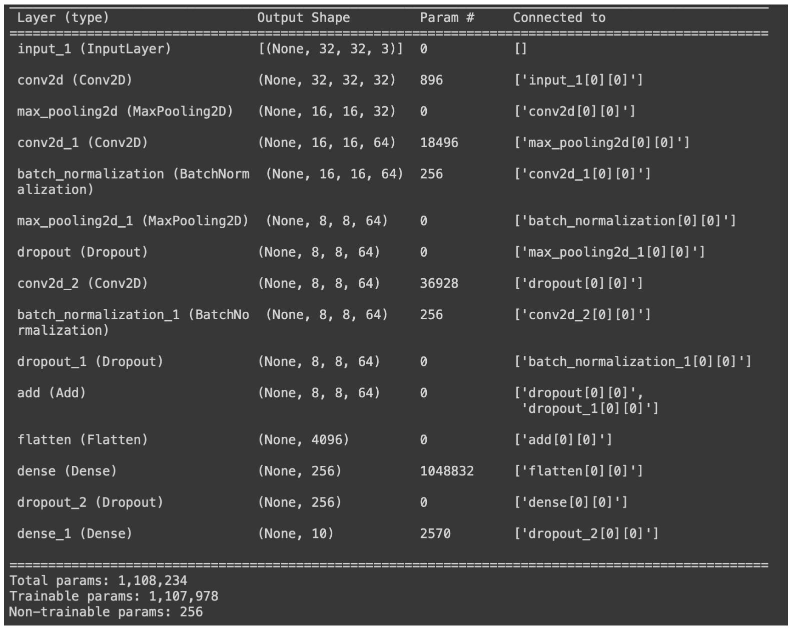
Results & Discussion

Model Type	Loss Curve	Accuracy Curve	Lowest Loss	Best Accuracy & Params
Naive CNN			1.137	Acc = .601, 7 epochs, .00085 LR
CNN w/ Inception block			1.18	Acc = .672, 12 epochs, .001 LR

Model Type	Loss Curve	Accuracy Curve	Lowest Loss	Best Accuracy & Params
CNN w/ residual block			1.44	Acc = .628, 15 epochs, .001 LR

The best performance came from my naive CNN with the implementation of the naive inception block (as seen in figure A). The naive inception block was added in between 'maxpooling2d_2' and 'conv2d_3' from the implementation shown in Figure 2 above. The architecture was modified slightly to have the residual block which is represented in Figure 3 to the right. Note that the pooling layer had to be removed in order to maintain proper dimensions for the adding together.

One interesting note is that the CNN w/ inception block had much nicer looking loss curves. As expected, adding a residual block and a inception block both improved performance of the naive CNN model. However, interestingly while the accuracy did increase when comparing to the naive CNN model, in both cases the loss also increased. Some interesting future directions may be to test different inceptions modules (not just the naive approach) and also to test residual connections within different parts of the model architecture.



```
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 32, 32, 3)]  0           []

 conv2d (Conv2D)                (None, 32, 32, 32)   896         ['input_1[0][0]']

 max_pooling2d (MaxPooling2D)   (None, 16, 16, 32)   0           ['conv2d[0][0]']

 conv2d_1 (Conv2D)              (None, 16, 16, 64)   18496       ['max_pooling2d[0][0]']

 batch_normalization (BatchNorm  (None, 16, 16, 64)  256         ['conv2d_1[0][0]']
 alization)

 max_pooling2d_1 (MaxPooling2D)  (None, 8, 8, 64)    0           ['batch_normalization[0][0]']

 dropout (Dropout)              (None, 8, 8, 64)     0           ['max_pooling2d_1[0][0]']

 conv2d_2 (Conv2D)              (None, 8, 8, 64)     36928       ['dropout[0][0]']

 batch_normalization_1 (BatchNo  (None, 8, 8, 64)    256         ['conv2d_2[0][0]']
 rmalization)

 dropout_1 (Dropout)            (None, 8, 8, 64)     0           ['batch_normalization_1[0][0]']

 add (Add)                      (None, 8, 8, 64)     0           ['dropout[0][0]',
                                                                  'dropout_1[0][0]']

 flatten (Flatten)              (None, 4096)         0           ['add[0][0]']

 dense (Dense)                  (None, 256)          1048832     ['flatten[0][0]']

 dropout_2 (Dropout)            (None, 256)          0           ['dense[0][0]']

 dense_1 (Dense)                (None, 10)           2570        ['dropout_2[0][0]']

==================================================================================================
Total params: 1,108,234
Trainable params: 1,107,978
Non-trainable params: 256
```

Fig. 3 - Model with residual block

References

[1] Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[2] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.