CS577: Assignment 4

Jane Downer
Department of Computer Science
Illinois Institute of Technology

November 11, 2022 (using 1 late day)

Abstract
Building and training basic convolutional neural networks using the Tensorflow framework in Python. Modifying networks by incorporating pre-trained conv base, data augmentation, inception blocks, and residual blocks.

<u>Task 1: Binary Classification</u>

1 Problem Statement

- Build and train convolutional neural nets for binary image classification. Observe results from different CNN architectures.

2 Proposed Solution

- Write functions that will be reused to build basic CNN models, perform grid search, and plot results.
- Build models according to guidelines provided in lecture notes.
- Use grid search to tune hyperparameters.
- Evaluate model and plot performance.
- Architectures and techniques explored:
    - Basic model with several conv-maxpool-batchnorm blocks
    - Transfer learning with VGG-16 conv base
    - Transfer learning with VGG-16 conv base plus data augmentation

3 Implementation Details

Before running code, set 'ROOT_PATH' at top of file to your preferred local directory, in addition to the locations of any models and data you download from the shared folder. At each step in the notebook, you can evaluate the model by simply loading it from your local files rather than training it.

- <u>Data</u>
    - Kaggle's "Cats and Dogs" dataset: 25000 images, 2 categories
    - Reduce each class to subset of 2000 images
    - Split into train, test, and validation folders with Python's `splitfolders` package
    - Remove files that are either (a) not images or (b) unable to be opened by the `PIL.Image`
    - Use ImageDataGenerator to create image generators and data iterators for each data directory
    - Set target size to (150,150,3), batch size to 32, and class mode to 'binary'.
- <u>Basic CNN</u>:
    - Define model structure:
        - 1. 2D Convolution (ReLU) + 2x2 Maxpooling + BatchNormalization
        - 2. 2D Convolution (ReLU) + 2x2 Maxpooling + BatchNormalization
        - 3. 2D Convolution (ReLU) + 2x2 Maxpooling + BatchNormalization
        - 4. Flatten + Dense (ReLU, 512 outputs) + Dense (Sigmoid, 1 output)
    - Grid search to find best learning rate and hidden units
        - Using custom grid search function, defined near the beginning of the notebook
        - Learning rate options: 1e-04, 1e-03
        - Hidden unit options (for 3 convolution layers): (64,64,64), (32,32,32)
        - Compile and train model using Adam as optimizer and batch size of 32.
    - Identify best model from grid search and continue training if needed.
    - Plot train accuracy and validation accuracy against epochs, and evaluate on test data.
- <u>Visualize intermediate activations</u>
    - Define a new model with `tf.keras.models.Model(inputs, outputs)`, where `inputs` are the same as the inputs to the original model and `outputs` are the outputs from each layer.
    - Get activations by calling `.predict(image)` on the new model, where `image` is a sample image from the dataset.
    - Plot the activations that correspond to the convolution layers.
- <u>Visualize filters for a given layer</u>
    - For each filter in the layer output:
        - Randomly instantiate an "image" matrix that we will continue to update

        - Obtain the loss and gradients associated with that layer and filter. Use these values to perform gradient descent on the randomly-generated image for a fixed number of iterations
        - Normalize, scale, shift, and clip image values to between 0 and 255. Plot result.
- <u>CNN with VGG-16 conv base</u>:
    - 1. Replace the convolutional layers from my model with frozen VGG conv base.
        - Model structure:
            - 1. VGG conv base (when frozen, represented as a single layer)
            - 2. Flatten + Dense (ReLU, 512 outputs) + Dense (sigmoid, 1 output)
        - Compile (optimizer: Adam, lr and hidden units: best from grid search, batch size: 32), train for 10 epochs
        - Plot train / validation accuracy against epochs, evaluate on test data
    - 2. Unfreeze conv base and continue training (layers increase from 4 to 30)
        - Compile model (optimizer: Adam, lr: 1e-06 (explained later)), train for 5 epochs
        - Plot train / validation accuracy against epochs, evaluate on test data
    - 3. Re-train frozen conv base with augmented data
        - Define new data generators with built-in transformations to augment data
            - Transformations (in addition to rescaling by 1/255, which we do regardless):

```
1. rotation_range=40
2. width_shift_range=0.2, height_shift_range=0.2
3. shear_range=0.2
4. zoom_range=0.2
5. horizontal_flip=0.2
6. fill_mode='nearest'
```

        - Compile model (optimizer: Adam, lr: best from step 1), train for 10 epochs
        - Plot train/validation accuracy against epochs, evaluate on test data

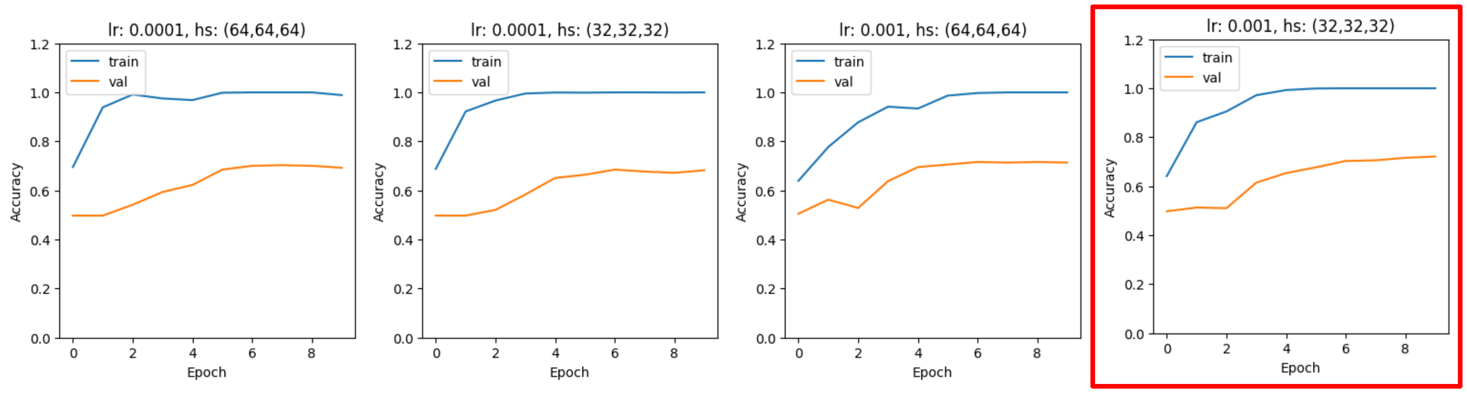
4 Results and Discussion

Note about optimizer and batch size:

For each cat/dog model, I used the Adam optimizer because, as we have discussed in lecture and previous assignment, it combines desirable features of other optimizers like momentum and adaptive learning rates. I also used a batch size of 32 because we are dealing with only 2000 images per class. When tuning hyperparameters, I kept these constant and focused on others.

<u>Basic CNN</u>

Grid search results

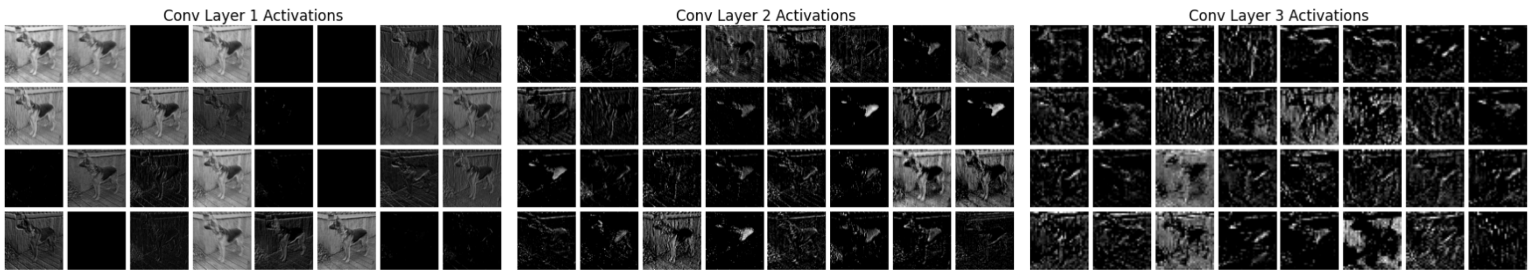


The four models resulting from grid search performed similarly over 10 epoch. The model that performed best (by a tiny amount – outlined in **red**) used learning rate 1e-03 and 32 units for each convolution layer.

```
Final training accuracy: 1.0            Test loss: 1.116
Final validation accuracy: 0.721        Test acc:  0.709
```

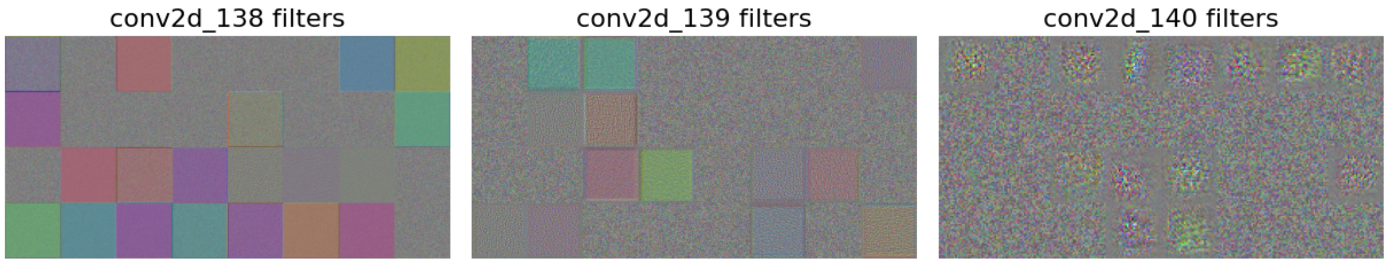
While the models generally performed well on the training set, I was unable to build a model that achieved much over 70% accuracy on unseen data. A possible explanation is the small size of our data subset (4000 images).

<u>Intermediate Layer Activations</u>



The images above show that the activations of early layers tend to represent big-picture objects, whereas the activations of later layers tend to represent more abstract features.

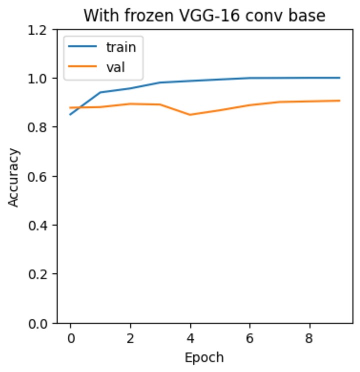
<u>Visualizing Filters</u>



Generally, filters in early layers of a convolutional neural network tend to show more specific features whereas filters in deeper layers tend to show larger patterns and objects. That trend is beginning to occur in the filters shown above, although it is happening slowly – the filters from the first convolutional layer mostly show colors, whereas the filters in the third convolutional layer are beginning to show patterns with multiple colors and features. I suspect the trend would be more obvious if the network had more layers.

<u>Training with frozen VGG-16 base</u>

Continuing to use lr=0.001 and batch size=32, since I previously achieved the best performance with those values.

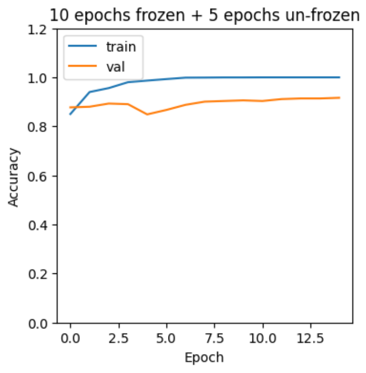


```
Final training accuracy: 1.0
Final validation accuracy: 0.906

Test loss: 0.352
Test acc:  0.907
```

When I exchanged my own convolution layers for the existing conv base from VGG-16, the model achieved its best performance on the validation set (about 90%) almost immediately – it achieved almost perfect accuracy on the training set after 4 epochs. This is particularly remarkable because VGG-16 was not trained to specifically identify cats and dogs. This suggests that when building a model from scratch, the bulk of training time is spent training the convolution layers to extract features – and that using an already-trained conv base will reduce training time to very few epochs. It also suggests the task of actually classifying the images is accomplished by the dense layers, which are the only layers actually being trained on the cat/dog data in this example.

<u>Continued training with un-frozen VGG-16 base</u>



```
Final training accuracy: 1.00
Final validation accuracy: 0.917

Test loss: 0.394
Test acc:  0.919
```

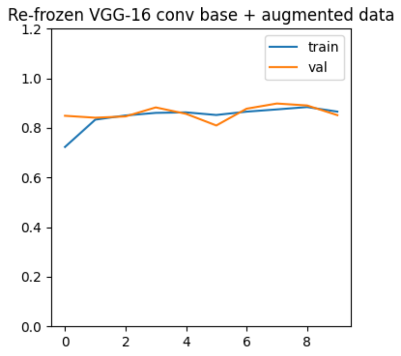
When fine-tuning the model, I reduce the learning rate significantly – from 1e-03 to 1e-06. This is because un-freezing the conv base results in a much larger model – 30 layers rather than 4 – but the dataset is still small (4000 images total). Tensorflow documentation explains that reducing the learning rate for fine-tuning helps prevent overfitting, which can easily happen in this case. [1]

Fine-tuning resulted in slighty better performance. We expect it to be better because it's a closer fit to our specific task as a result of training all layers of the network. The improvement is not large in this case, but to be fair, the model was already performing very well, so there wasn't much room for improvement.

Training with frozen VGG-16 base + data augmentation

Sample transformed images:

Results of training:



```
Final training accuracy: 0.866
Final validation accuracy: 0.852

Test loss: 0.342
Test acc:  0.846
```

Incorporating augmented data increased the size of the dataset. This helped reduce overfitting – we can see this in the above plot, which shows that validation accuracy is no longer lower than test accuracy. (It performs similarly well on the test set.) One thing to note: while this model reduces overfitting, performance on training, validation, and test sets are all somewhat lower than they were in the previous model – about 85% for all three measures (rather than ~100% for training and ~90% for validation and testing) . However, as a result of data augmentation, we are working with a fundamentally different dataset, so it might not make sense to have the same standard of success when compared to the previous model. It is also worth noting that we have trained for 15 or fewer epochs in this case and the one before it – we may need to train longer to get an accurate comparison of model performance.

5 Conclusion

I produced four types of models in this task:

1. a basic CNN,
2. a CNN with a frozen VGG conv-base,
3. the previous model with additional fine-tuning
4. a CNN with a frozen VGG conv-base with data-augmenting generators

The first model was the weakest, which makes sense because it has few layers and trained from scratch. The second model was a huge improvement, because the convolution layers were already trained, and the third model was a *slight* improvement on that, because it was fine-tuned to more closely match our particular task. The fourth model was better than the first, but not as good as 2 and 3 – but as I discuss above, since the dataset changes as a result of data augmentation, this may be an "apples and oranges" comparison and it may make more sense to judge this model individually on its own merits rather than compared to the others. If given more time, I would continue experimenting with hyperparameters for models 2, 3, and 4, and would train for more epochs.

6 References

[1] https://keras.io/guides/transfer_learning/

Task 2: Multi-class Classification (CIFAR-10)

- Data
  - CIFAR-10 Dataset
    - 60000 images, 10 categories, shape (32,32,3)
    - I wanted to focus on a small subset of the data, as we did with the first task, but did not, because the instructions did not specify that this was ok to do. The size of the dataset made it difficult to experiment with many hyperparameter settings.
  - Load pickled data and split into train, test, and validation sets. Rescale by dividing images by 1/255 and converting to float. Use `np_utils.to_categorical()` to perform one-hot encoding on labels.
  - Use ImageDataGenerator to create image generators and data iterators. Set batch size to 1024 (explained below).
- Basic CNN:
  - Model structure:
    - 1. 2D Convolution (ReLU) + 2x2 Maxpooling + BatchNormalization
    - 2. 2D Convolution (ReLU) + 2x2 Maxpooling + BatchNormalization

        - 3. 2D Convolution (ReLU) + 2x2 Maxpooling + BatchNormalization
        - 4. Flatten + Dense (Softmax, 10 outputs)
    - Grid search to find best number of hidden units
        - Hidden unit options (for 3 convolution layers): (256, 256, 256), (128, 128, 128)
        - Note: unlike previous task, did not include learning rate options in grid search. I explain my reasons below.
        - Compile and train for 60 epochs using Adam as optimizer and batch size of 1024 (explained below).
    - Identify best model from grid search. Plot train / validation accuracy and evaluate on test data.
- <u>CNN with inception blocks</u>
    - Model structure same as before, except the second block of conv-maxpool-batchnorm is followed by an inception block with the following structure:
        - Four parallel components, each of which takes the output of the batch normalization layer as input:
            - 1x1 convolution (ReLU)
            - 1x1 convolution (ReLU) followed by 3x3 convolution (ReLU, 'same' padding)
            - 1x1 convolution (ReLU) followed by 5x5 convolution (ReLU, 'same' padding)
            - 2x2 max pooling followed by 1x1 convolution (ReLU)
        - The output of the inception block is the concatenation of the output of these four components.
    - Compile model (optimizer: Adam, hidden layer sizes: best from grid search, batch size: 1024). Train for 60 epochs.
    - Plot train / validation accuracy against epochs, evaluate on test data
- <u>CNN with residual blocks</u>
    - Model structure same as the basic CNN, except used skip connections to feed residuals into later layers:

```
input____1 = Input(shape=cifar_shape)
output___1 = Conv2D(h1,(3,3),activation='relu',padding='same')(input____1)
output___1 = MaxPool2D((2,2))(output___1)
output___1 = BatchNormalization()(output___1)
residual_1 = Conv2D(h1,1,strides=2,padding='same')(input____1)
input____2 = tf.keras.layers.add([residual_1, output___1])
output___2 = Conv2D(h2,(3,3),activation='relu',padding='same')(input____2)
output___2 = MaxPool2D((2,2))(output___1)
output___2 = BatchNormalization()(output___1)
residual_2 = input____2
input____3 = tf.keras.layers.add([residual_2, output___2])
output___3 = Conv2D(h3,(3,3),activation='relu',padding='same')(input____3)
output___3 = MaxPool2D((2,2))(output___3)
output___3 = BatchNormalization()(output___3)
flat = Flatten()(output___3)
out  = Dense(10,activation='softmax')(flat)
```

    - Compile (optimizer: Adam, hidden layer size: best from grid search, batch size: 1024). Train for 60 epochs.
    - Plot train / validation accuracy against epochs, evaluate on test data

4 Results and Discussion

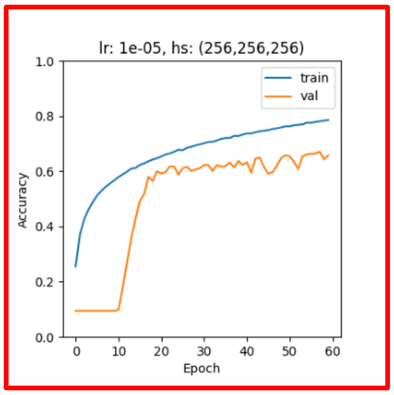
Note about optimizer and batch size:

For this model, I used the Adam optimizer and a batch size of 1024. I used the Adam optimizer for reasons mentioned above. I used a batch size of 1024 because we are dealing with a lot of images -- 60000. I kept these values constant and modified other hyperparameters if needed to improve model performance.

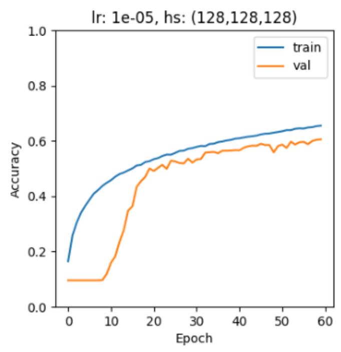
<u>Basic CNN</u>

Note: the grid search only explored variations in hiden layer sizes, unlike the first task, where I also experimented with learning rates. This is because training time increased dramatically – each of the modelsin this grid search took several hours to train. I decided to choose a safe option for learning rate (a low value of 1e-05, which would be slow but likely wouldn't produce bad/noisy results) to make sure the time spent training wouldn't be wasted by a learning rate that was too high. I also trained for more epochs (60 instead of 10) to make sure the model had enough time to train.

<u>Grid search results:</u>



```
Final train accuracy:
0.785
Final val accuracy:
0.657
```



```
Final train accuracy:
0.655
Final val accuracy:
0.605
```

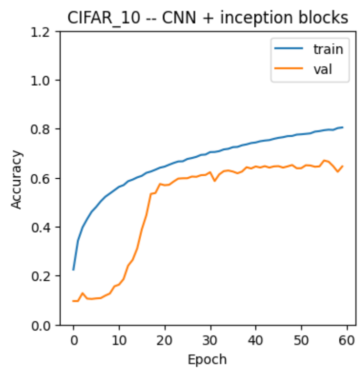
Of the two models trained in grid search, the best (in red) was with a learning rate of 1e-05 and a size of 256 for each of the three convolution layers. Test performance was similar to validation performance:

```
Test loss: 1.014          Test acc:  0.651
```

It makes sense that 256 hidden units are better than 128, because it allows the model to see more of the (large) dataset. However, training and validation accuracy plateau at about 78.5% and 65.7%, respectively. It's possible that this only *seems* like a plateau because the learning rate is so low – however, as mentioned before, training took to long to test this idea.

Another note: in both grid search models, validation performance doesn't change until about 10 epochs. This suggests that it takes a while for the model to learn patterns that are useful outside of the particularities of the training data.

Basic CNN with an inception block

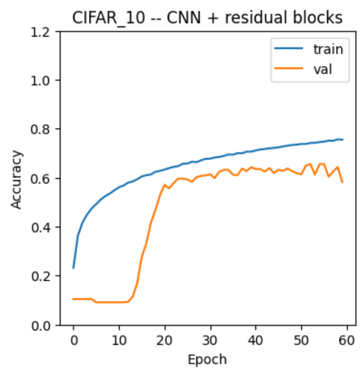


```
Final training accuracy: 0.805
Final validation accuracy: 0.646

Test loss: 1.037
Test acc:  0.64
```

The an inception block led to similar final results, with one difference: while validation accuracy still has a huge spike around 10 epochs, it isn't entirely stagnant before then. This suggests the inception block helped the model generalize earlier.

Basic CNN with residual blocks



```
Final training accuracy: 0.755
Final validation accuracy: 0.583

Test loss: 1.171
Test acc:  0.592
```

Incorporating two residual blocks into the basic CNN architecture produced similar, but slightly worse, performance on the train, validation, and test sets. This is disappointing, since the original CNN wasn't performing very well to begin with. One thing that occurred to me is that skip connections tend to be useful in deep networks, where the increased number of multiplied gradients in backpropagation can lead to vanishing or exploding gradients. My network is pretty shallow – just three groups of conv-maxpool-batchnorm layers. Something that occurred to me is that the model is possibly to shallow for residual blocks to yield any benefits.

5 Conclusion

I built and trained three models for the CIFAR-10 dataset: (1) a basic CNN, (2) CNN + inception block, (3) CNN + residual blocks. Training accuracy reached between 75 and 80 % in each of these models, and validation and test accuracy reached about 60 to 65%. This is significantly better than randomly choosing 1 of 10 classes, but still not as good as I would have hoped for.

If given more time (and a faster computer), I would have liked to experiment with different hyperparameters. Specifically, I would have liked to increase the number of units in the hidden layers, since the grid search suggested this could help. I would also like to increase the batch size, since it takes the model a while to improve its performance on unseen data.

6 References

n/a