Transfer Learning and Convolutional Neural Networks (TLCNN)

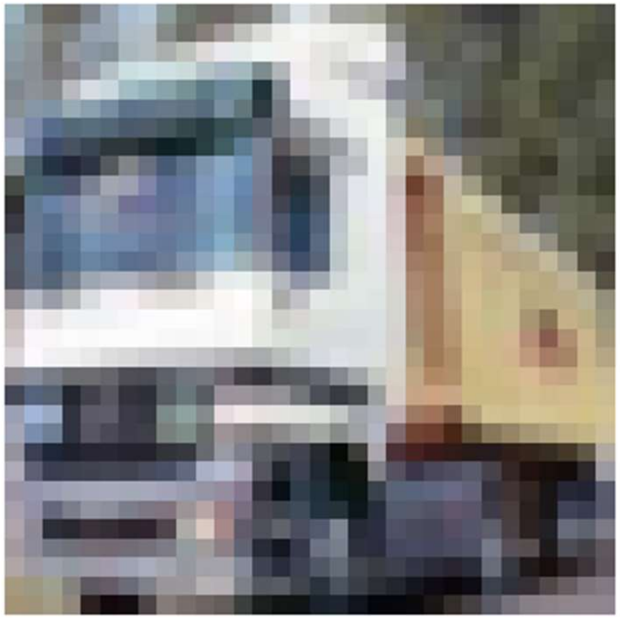
by Dan Hoogasian

Intro

- Image classification has recently boomed in popularity, thanks largely to increased computing power (due to GPUs, the cloud, etc.) and the creation of fast, accurate models
- Many pre-built deep neural network models (mainly CNNs) available to build upon for image classification
- Pre-built models are most popular option, though some users prefer building own models, especially if task is unique
- This project investigated building CNNs for two datasets and the difficulties in performing transfer learning
  - Main difficulty was altering input image size throughout models

Datasets: CIFAR-10

- From the MIT Computer Science & Artificial Intelligence Laboratory
- 60,000 labeled 32x32 color images
    - Classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck



Datasets: STL-10

- From the Stanford University Computer Science department
- Inspired by CIFAR-10
- 13,000 labeled color 96x96 images
    - Classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, and truck
- Mutually exclusive monkey class from STL-10 and frog class from CIFAR-10 were dropped to improve model transferability

Models: CIFAR-10

- A rather simple model was developed (adapted from a blog post tutorial on CNNs)
- Four convolution layers, five batch normalization layers, two pooling layers, one hidden deep layer, one dropout layer, and one dense output layer
- 590,000 parameters

```
Layer (type)                                  Output Shape          Param #
==========================================================================
conv2d (Conv2D)                               (None, 32, 32, 32)    896
batch_normalization (BatchNormalization)      (None, 32, 32, 32)    128
conv2d_1 (Conv2D)                             (None, 32, 32, 32)    9248
batch_normalization_1 (BatchNormalization)    (None, 32, 32, 32)    128
max_pooling2d (MaxPooling2D)                  (None, 16, 16, 32)    0
conv2d_2 (Conv2D)                             (None, 16, 16, 64)    18496
batch_normalization_2 (BatchNormalization)    (None, 16, 16, 64)    256
conv2d_3 (Conv2D)                             (None, 16, 16, 64)    36928
batch_normalization_3 (BatchNormalization)    (None, 16, 16, 64)    256
max_pooling2d_1 (MaxPooling2D)                (None, 8, 8, 64)      0
batch_normalization_4 (BatchNormalization)    (None, 8, 8, 64)      256
flatten (Flatten)                             (None, 4096)          0
dense (Dense)                                 (None, 128)           524416
dropout (Dropout)                             (None, 128)           0
dense_1 (Dense)                               (None, 9)             1161
==========================================================================
Total params: 592,169
Trainable params: 591,657
Non-trainable params: 512
```

Models: STL-10

- More complex approach taken, attempting to achieve a high accuracy, given that this was the more complicated dataset
- Performed hyperparameter tuning via a random search with Keras tuner:
  - Learning rate (from .1 to .001)
  - Optimizer (Nesterov, Adam, or RMSprop)
  - Number of convolution layers (from 1 to 3)
  - Number of filters per convolution layer (from 32 to 96 for input layer, increasing by 100% in each following layer)
  - Number of hidden dense layer (from 1 to 3)
  - Number neurons per hidden dense layer (from 128 to 1024 in first layer, reduced by 50% for each following layer)

Layer (type)	Output Shape	Param #
sequential (Sequential)	(None, 96, 96, 3)	0
conv2d (Conv2D)	(None, 96, 96, 64)	1792
batch_normalization (BatchNormalization)	(None, 96, 96, 64)	256
max_pooling2d (MaxPooling2D)	(None, 48, 48, 64)	0
conv2d_1 (Conv2D)	(None, 48, 48, 128)	73856
batch_normalization_1 (BatchNormalization)	(None, 48, 48, 128)	512
conv2d_2 (Conv2D)	(None, 48, 48, 256)	295168
batch_normalization_2 (BatchNormalization)	(None, 48, 48, 256)	1024
max_pooling2d_1 (MaxPooling2D)	(None, 24, 24, 256)	0
dropout (Dropout)	(None, 24, 24, 256)	0
flatten (Flatten)	(None, 147456)	0
dense (Dense)	(None, 256)	37748992
batch_normalization_3 (BatchNormalization)	(None, 256)	1024
dropout_1 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 9)	2313

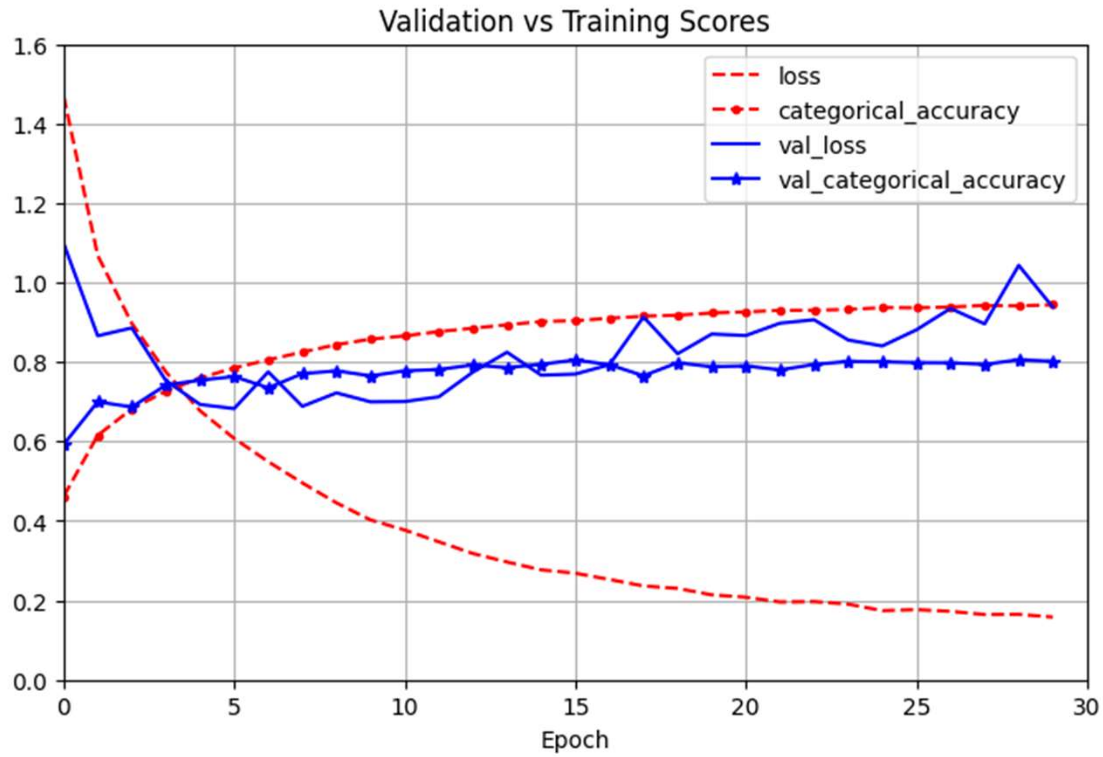
Total params: 38,124,937

Trainable params: 38,123,529

Non-trainable params: 1,408

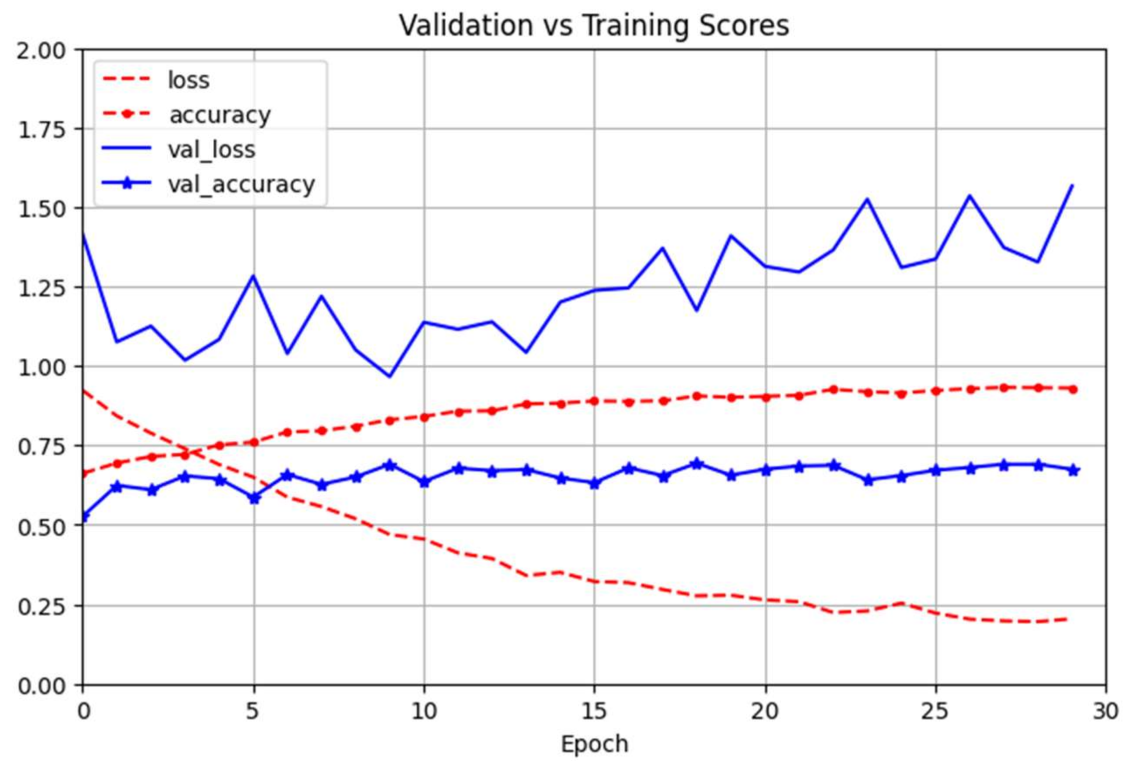
Results

- Evaluation metrics were categorical accuracy and categorical cross-entropy
- Used 30 epochs for both models
- CIFAR-10
  - Accuracy: 80%
  - Loss: 1.0
- A bit of overfitting



Results

- STL-10
  - Accuracy: 71%
  - Loss: 1.18
- Slightly worse overfitting
- Not impressive scores for either dataset, but sufficient for my primary goal



Transfer Learning

- Adjusting image sizes is big issue
- Are already many developed and pre-trained models in Keras, often have parameter for input image shape
  - tf.keras.applications.VGG16(input_shape=(224, 224, 3))
  - This likely resizes images (e.g., tf.keras.layers.Resizing)
- Resizing is simple, but I wanted to explore if I could avoid resizing, possibly just using the necessary weights, reusing weights multiple times per filter, or something of the likes
- Attempted four solutions (none were successful):
  - Changing the input layer and its input shape
  - Creating a new input layer to accommodate a differently sized input
  - Including an input shape parameter in the model itself
  - Convert the model to a JSON file, updating the model, then converting back to a Keras model
- Third method seems most plausible, with the creation of a class for the model

Discussion

- Being able to develop a model from scratch is a valuable capability that will likely become more popular with the increased applications of image classification
- Image size is very important to think about if considering transfer learning
- Image size for pretrained models when doing transfer learning would be a good topic for further research