Term Project: Deep Learning Image Recognition Robustness

Tyler Ward

December 8, 2023

1 Introduction

The use of multi-layered artificial neural networks (also known as deep neural networks) is a field of increasing interest since 1943 when Walter Pitts and Warren McCulloch created a computer model based on the neural networks of the human brain [6] [7]. In recent years, the use of deep neural networks (DNNs) and their extensions (grouped under the term "deep learning") have been used to perform image recognition in even everyday tasks. This is highlighted significantly by Face recognition technologies used for password access on most smart phones, along with image search applications, where a database can both be searched by applying an image and images can be found easily by keyword search. These methods also have important applications for government and healthcare agencies in the ability to perform identity recognition and help medical professionals in the detection of tumors and other ailments [5].

When considering the use of deep learning for these applications, several questions of interest surface: 1. How does the performance of a standalone deep neural network (DNN) compare to a network that includes convolutional structure (i.e., a convolutional neural network (CNN))? 2. What role does learning color play in accuracy of image recognition? (I.e., what changes in model performance are there when images in the training dataset are gray scale as opposed to in color?) 3. How does random noise affect model performance, and is this affect parallel for both a DNN and a CNN, and for color vs. gray-scale images? This project presents answers to these questions based on a simulation study using a subset of the Caltech 256 dataset [9] comparing the DNN and the CNN when applying different levels of random noise and gray scale filters to images in the training data. The abilities of CNNs are then explored further, including the use of a pretrained CNN with 16 layers called VGG16 [16].

2 Methodology

Many label machine learning methods such as deep neural networks as "a black box", owing to the myriad number of parameters to be estimated and optimized. However, any artificial neural network (ANN), whether it be deep (with multiple hidden layers) or not, can really be simplified into the five pieces: inputs, linear combinations of inputs, weight matrices and bias vectors, activation functions, differentiation, and optimization. These methods are used repeatedly information is repeatedly passed through the layers of the neural network and then updated depending on the in-sample error. I describe this structure in section 2.1 below.

2.1 Artificial Neural Network Structure

Inputs For a 256/256 pixel image, there are 65,536 inputs for each pixel in the image, with the intensity of red, green, and blue for each image encoded into separate nodes to be passed through to the hidden layers the neural network.

Linear combinations For each hidden layer in the network, a set of weights are generated associated with each row of inputs $\mathbf{X}$ as it feeds forward to an each node in the layer along with a bias value. For the first iteration, these weights and bias values are randomly assigned initial values. The input values are then matrix multiplied by their associated weights and then summed with the bias vector in preparation to have the activation function applied before these values are passed to the next layer.

The number of neurons in each hidden layer is specified by the user, with more neurons often leading to greater ability to catch novel relationships. This weighted sum is shown in mathematical notation in Equation 1, as described in Aurelien Geron's book on Hand's on Machine Learning [7]. For a given hidden layer $\mathbf{h}$, with inputs from the previous layer $\mathbf{X}$, will be weights matrix $\mathbf{W}$ with a row for each input and a column for each hidden layer node times that input matrix $\mathbf{X}$ the values for that hidden layer plus the bias vector. This value is also called the logit score. The values for the next hidden layer are computed as an activation function f is applied to the logit scores, as defined in Equation 1 below.

$$\mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = f(\mathbf{X}\mathbf{W} + \mathbf{b}) \tag{1}$$

. This process is repeated for all hidden layers until the output layer, where an activation function is applied to give desired outputs from the model, as shown in Figure 1. These hidden layers are called dense layers when every neuron in a previous layer is connected to every one of the neuron in the current layer.

Activation Functions After each hidden layers, an activation function is used to introduce non-linearities into neural network. This allows for the neural network to capture more complex relationships in the input data during training[17]. Activation functions are also important as they align values into differentiable function that allow for iteratively updating the weights of the neural network using vectors of partial derivatives (i.e., the gradient). This process is known as back propagation and is discussed further below. I chose to use the Rectified Linear Unit (ReLU) activation function, as it has been shown to have faster training time than other functions, especially after it's use in the revolutionary AlexNet model which beat state of the art performance on the ImageNet dataset by over 10% and arguably kick-started the deep learning image recognition boom [4]. This function constrains all negative inputs to be 0 and then holds all other values constant (see Equation 2), and is therefore fast to compute and avoids the problem of vanishing gradients as it is simple to compute and is unbounded, allowing values from 0 to infinity [3].

$$ReLU(x) = max(0, x) = \mathbf{I}_{x>0}(x) \tag{2}$$

One other activation function I apply in my image recognition methods is the softmax activation function. This function converts the logit score of the last hidden layer ($z_c = \mathbf{W}^T\mathbf{h} + \mathbf{b}$) for a given class c into a probability that the input image belongs to a class. This conversion is made by dividing the exponentiated logit for a given class divided by the sum of all the exponentiated z_j logits for all K of the classes, as shown in Equation 3. This will therefore produce a value bounded between 0 and 1, or a probability that the input image belongs to class c.

$$P(y = c) = \frac{e^{z_c}}{(\sum_{j=1}^{K} e^{z_j})} \tag{3}$$

Differentiation The weights and bias values are then iteratively updated via a process using partial derivatives called back propagation, which computes the gradient of the network's error with regard to every single parameters in the model [7]. The specific back propagation algorithm is as follows: After a forward pass through the layers of the neural network is complete, the networks output error is computed, which for categorical multi-level classification problems is commonly measured by categorical cross entropy (defined in 4). Then, the contribution of each output connection to the error is quantified using the chain rule. This is followed by quantification of error contributions from each connection in the hidden layers (still using the chain rule) below in the network, continuing *back* until reaching the input layer. The end product is therefore an error gradient a cross all the weights in the network, which can then be tweaked using a Gradient Descent algorithm until the network error is uniquely minimized (or the number of specified epochs is reached). See Figure 2 for a visual representation of this process.

Optimization As the feed-forward and back propagation process is repeated for the specified number of epochs, the parameters are tweaked in order to minimize a cost function. As mentioned before, the cost function commonly used for our categorical classification problems such as image recognition is called categorical cross entropy, which is defined in Equation 4 (see [8]). With s_c as the softmax score

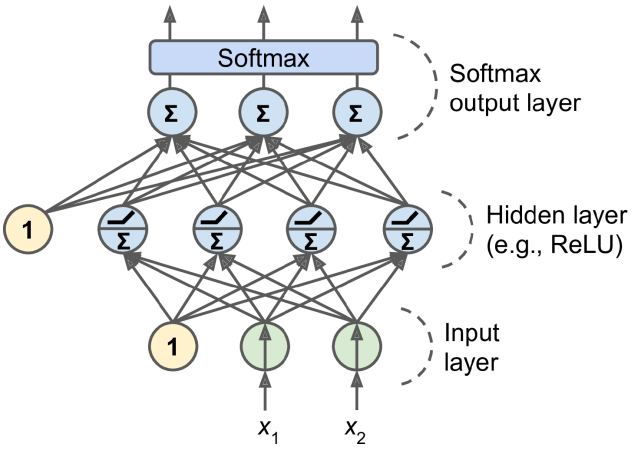

Figure 1: Basic ANN framework, including inputs, linear combinations, activation functions. Note the bias values in yellow used to help the model be more generalizable to potential deviations in future data

of a given class c, this metric quantified the ability for the network to predict a class, with lower values indicating greater ease of prediction.

$$CE = -log(\frac{e^{s_c}}{\sum_j^K e^{s_j}}) \tag{4}$$

Gradient Descent The method used for minimizing categorical cross entropy with respect the weight parameters is called Gradient Descent, which is a common optimization method in machine learning. In basic terms, this method uses the vector of partial derivatives (i.e., the gradient) to find the direction of greatest increase in an objective function of parameters of interest $J(\theta)$, and then goes in the opposite direction at a specified "learning rate" α.

$$\theta_{n+1} = \theta_n - \alpha \nabla J(\theta_n) \tag{5}$$

Therefore, the value of the parameter of interest θ "descends" to the minimum of the function, shown in Equation 5 for the nth iteration of the gradient calculation. This process is repeated ideally until convergence, or until the number of epochs has been reached.

Additional considerations To decrease overfitting I randomly dropped neurons in the network. Additionally, to reduce computational expense, I used an early stopping method, with training stopping if loss was not reduced for 2 consecutive epochs. Also, I chose to use the Adam optimization method in model fitting, which is an adaptive form of gradient descent that uses the average of the second moments of the gradients, and reaches lower training loss with fewer epochs (preferred for me as I only had time and resources to use 10-15 epochs in my model training).

For even more detail on neural network methodology, see Michael Nielsen's online book on Deep Learning [15] and Chapter 10 of Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow by Aurélien Géron [7].

2.2 Convolutional Neural Networks

Convolutional neural networks harness an architecture that takes advantage of the spatial structure in an image, which makes them particularly effective at classifying images [15]. They do this with the addition of two types of layers to the traditional ANN structure: convolutional layers and pooling layers, as visualized in Figure 4. The purpose of these layers is to extract features from the inputs, which is particularly effective for images as the grid of pixel intensity values is searched spatially for unique combinations. The resulting extracted features from these layers are then flattened, and then input into the connected ANN structure

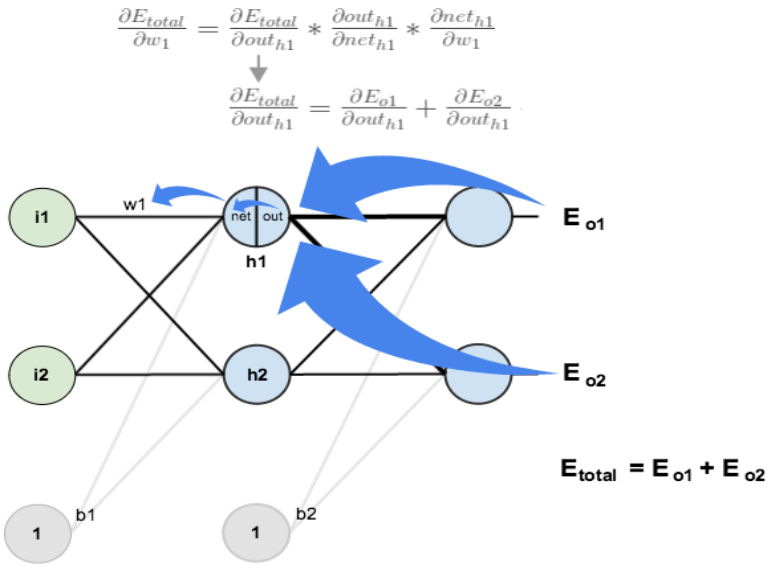

Figure 2: A basic visualization of the back propagation process [13]

described in section 2.1, which then performs the classification task using these learned features [1]. The specifics of these layers are described below

Convolutional layers Convolutional layers help find maps of features in an image that can help identify the image and future images. These feature maps are chosen by the model, first initialized randomly and then altered by the model as different patterns are found across all images [1]. Activation functions are then applied to these patterns to standardized the output and introduce nonlinearity into the network, just as in the forward pass through the layers of a full connected neural network.

Pooling layers Pooling layers are applied to feature maps to summarize pixels in a specified rectangular neighborhood. This helps to reduce the spatial size of the feature map, while still allowing for identification of similar patterns in other images. The most common pooling methods is called max pooling, which takes the max value of a given box of pixels. This pooling is applied in "strides", meaning the specified box is moved from left to right either column by column (i.e., a stride = 1) or by another specified amount. A visual example of max pooling with a 2x2 box filter and a stride of 2 can be found in Figure 5 in Appendix A.

3 Simulation Study

The task of classifying images with neural networks is computational expensive and can be quite difficult unless large amounts of training data that cover many possible displays of a certain class of object are accessible. Some have argued that augmenting images, such as adding random noise over the images, can help improve overall generalizability of deep learning models to new images [2]. Others also advocate for the possible use of gray scale images in image recognition task, as this should reduce computational requirements and simplify feature learning [11]. Therefore, I performed a simulation study studying my own variation of these types of questions. Specifically, can noisy image, gray scale images, or even a combination of the two be used as training data for object recognition tasks without significant effect on performance? If there is an affect, are these effects different for a simple DNN with only dense layers versus a CNN? I studied these questions via 20 simulations of all 8 combinations of three two-level factors: which type of neural network was used (simple DNN vs. CNN), noise vs, no noise, and original color vs. gray scale. The low number of simulations was due to the high computational cost of this task, with one simulation of these 8 combinations of factors taking approximately an hour (or more) to complete. The 4 types of images which each model

Table 1: Simulation study results (two methods, 2x2 factor set)

Method	Mean Accuracy	SD
DNN BASE	**0.068353**	0.015568
DNN NOISY	0.065173	0.015315
DNN GRAY	0.062909	0.015348
DNN NOISY+GRAY	0.057033	0.009087
CNN BASE	**0.249855**	0.009302
CNN NOISY	0.216908	0.011855
CNN GRAY	0.191089	0.009587
CNN NOISY+GRAY	0.177168	0.008236

was tested on are shown in Figure 3. For application of noise, random Gaussian noise with a mean of zero and a standard deviation of 0.1 was applied as images were input into model training.

These applications were applied to a subset of images from 46 classes from the Caltech 256 dataset. This dataset includes at least 80 images of various sizes for each class that were downloaded from Google images and then manually screened to make sure they fit their class. The image included for the simulation include images from 44 different types of animals (such as the adorable penguin shown in Figure 3), along with images of people and images of superman. This data was then split into a training and test set, with The DNN had three hidden layers with 512,256, and 128 neurons in each layer, respectively. The CNN had 3 convolutional layers with 32, 64, and 128 3x3 feature maps, respectively, each followed by 2x2 max pooling 2x2 with a stride of 1. 2 fully connected dense layers (i.e., a DNN) were then applied, with 512 and 256 neurons in the respective layers, and random dropout applied between the layers. Each method was fit using 12 epochs. Aggregated results from the simulation study are shown in Table 1.

3.1 Results

The results of this simulation study show several interesting trends. First, it is clear that using a convolutional structure adds significant capability beyond what is provided by the standalone 3 layer DNN. This is shown clearly by base average accuracy levels almost 20% higher by the convolutional method than the simple DNN. However, it is interesting to note the steep drop off in average performance that occurs for CNNs as noise is added and color is taken away. It is likely that the addition of random noise makes it much harder for the convolutional and pooling layers to identify unique features needed to correctly classify an image. Transforming the images to gray scale also appeared to contributed to significant declines in performance of the CNN. Though the performance of the basic DNN was quite low, it is interesting to see how the application of the noise and gray scale factors did not seem to contribute to as steep of a drop in performance for simple DNNs as for the CNNs. This highlights further the importance of a deep study of image architecture in order to effectively extract the features, and then apply classification techniques with the flexibility of a deep neural network. From this study, I can infer that it would not be wise to train a model with gray scale or noisy images when clear, color images will be the target for image recognition. However, these training methods may have greater benefit if new incoming images are likely to be gray scale or noisy. Studying performance of noisy and gray scale images on out of sample images of the same type would be very intriguing topics for a future simulation study.

4 Real Data Example: Deeper CNN to classify 46 classes from the Caltech 256 dataset

After observing the results of my simulation study, I chose to apply a slightly deeper CNN to the 46 classes of animals and people (plus superman) to see if I could improve upon the out of sample accuracy. I did this by adding a fourth convolutional layer with 256 features maps along with a max pooling layer of the same form as described in my simulation study. I also added another dense layer with 128 neurons to aid in classification. The results of this example are shown by the plot of the learning curves in Figure

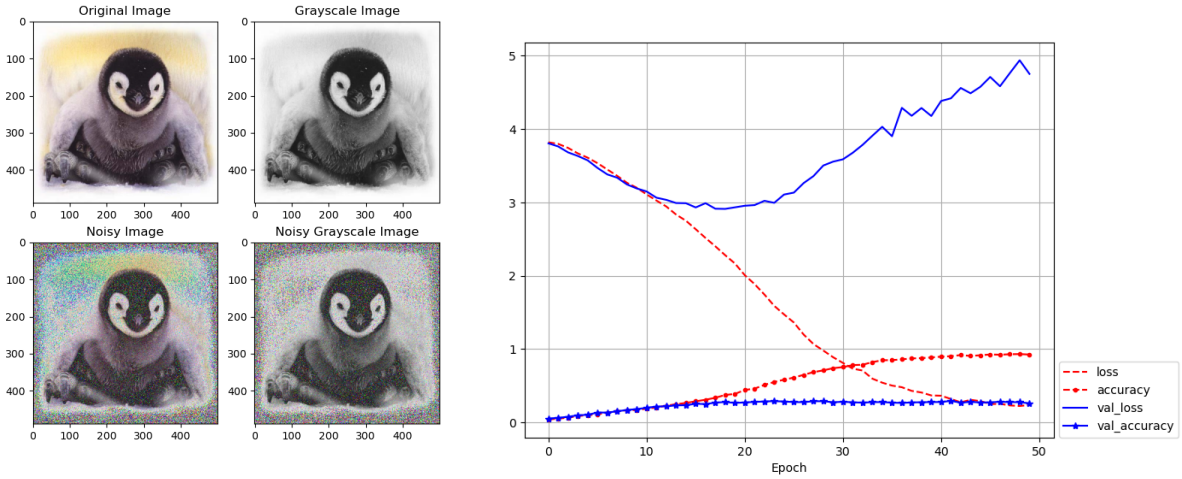

Figure 3: left: Application of factors to images in simulation study; right: learning curve results from deeper CNN trained with 50 epochs

3. Interestingly, even after applying more epochs, the CNN model was unable to learn much more than approximately 30% of the images. More precisely, it was only able to accurately predict 30% of the images correctly out of sample after around 20 rounds of forward passes and back propagation, even without any noise added to the images, and this performance did not improve with more iterations. An even larger pool of images is likely needed for performance to be improved, along with more convolutional layers to more fully learn the features needed to accurately classify the 46 classes. Additionally, using a pretrained extremely deep CNN model, such as AlexNet [12] or VGG16 [16] which are trained on millions of images and much more layers than the networks I fit, followed by a few layers of training with this data could be a way to use features learned from thousands of other images to classify the 46 classes of interest here. Using VGG16 pretrained model plus two fully connected layers trained on the Caltech 256 data showed improvements in validation accuracy for the 46 classes of interest by over 20%, with the best model after 12 epochs predicting with over 55% accuracy. This performance is shown in Figure 6 (see Appendix A).

5 Conclusion

In this study I reviewed the methodology behind deep artificial neural networks and their image recognition friendly extensions, convolutional neural networks. A simulation study found that convolutional neural networks significantly increase model performance when compared to a standalone neural network. However, this method has significantly diminishing returns when random noise is applied and when image colors are augmented to gray scale. Furthermore, the effect of random noise appeared to be much less significant on the DNN. Applying a slightly deeper CNN to classify 46 of the 257 classes in the Caltech 256 dataset yielded some improvements in out of sample performance (up to $\approx 30\%$ from an average of 25% in the simulation studies). However, this performance pales in comparison to the use of pretrained networks such as VGG16, which are trained on millions of images and have many more layers in their convolutional framework. Future examination can be performed to assess the performance of deep learning methods when out of sample images are also noisy or gray scale, along with further use of pretrained models to optimize performance and minimize computational cost when engaging in deep learning image recognition. Future efforts can also look into which classes are most often misclassified, including if different animals or other classes of image are commonly confused, and look into changes in training data to help address these confusions.

References

[1] J. Brownlee. How do convolutional layers work in deep learning neural networks?, Apr 2020.

[2] J. Brownlee. How to improve deep learning model robustness by adding noise, Aug 2020.

[3] J. Brownlee. How to choose an activation function for deep learning, Jan 2021.

[4] R. Chow. Imagenet: A pioneering vision for computers, Mar 2022.

[5] Comidor. 6 use cases of image recognition in our daily lives: Comidor, Nov 2022.

[6] K. D. Foote. A brief history of deep learning, Feb 2022.

[7] A. Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2nd edition, 2019.

[8] R. Gomez. Understanding categorical cross-entropy loss, binary cross-entropy loss, softmax loss, logistic loss, focal loss and all those confusing namesr, May 2018.

[9] G. Griffin, A. Holub, and P. Perona. Caltech 256, Apr 2022.

[10] S. N. Gupta. Deep convolutional neural networks (dcnns) explained in layman's terms, Feb 2022.

[11] C. Kanan and G. Cottrell. Color-to-grayscale: Does the method matter in image recognition? *PloS one*, 7:e29740, 01 2012.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[13] M. Mazur. A step by step backpropagation example, Aug 2022.

[14] D. Nielsen. Deep learning cage match: Max pooling vs convolutions, Sep 2018.

[15] M. A. Nielsen, Dec 2019.

[16] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[17] T. Wood. Activation function, Sep 2020.

A Additional Figures

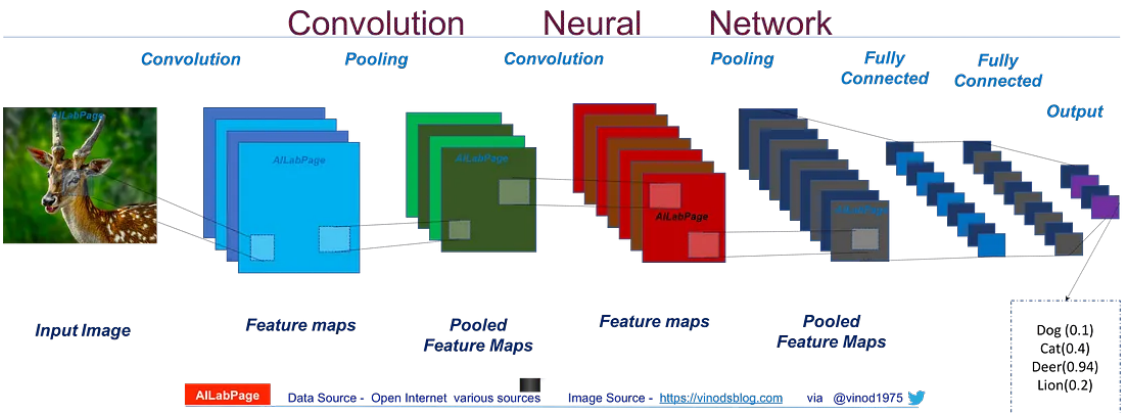

Figure 4: Detailed visualization of convolutional neural network structure (from Medium post [10])

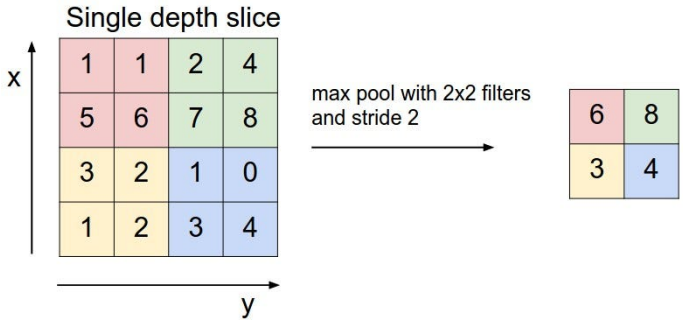

Figure 5: A visual example of max pooling with a stride of 2, from a blog post by by Duane Nielsen [14]

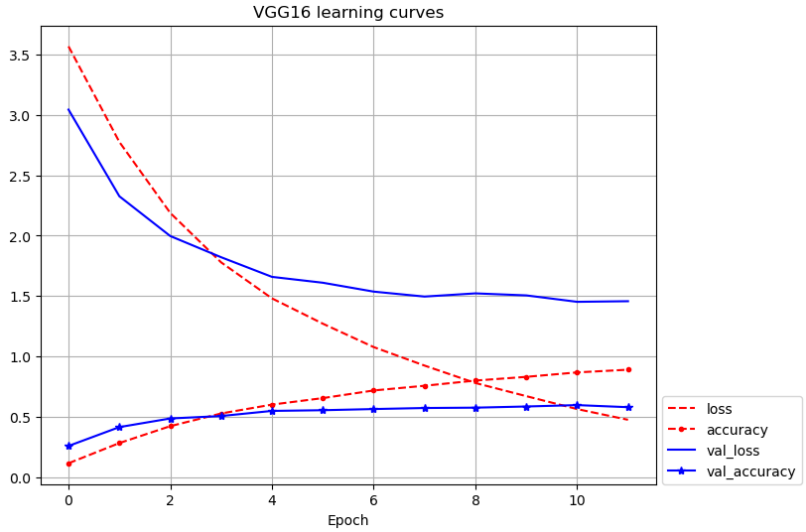

Figure 6: Performance of model with two hidden layers transferred from pretrained VGG16 model

B Python code

```
# imports
import numpy as np
import pandas as pd
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import pickle

# %%
import tensorflow as tf
# tensor flow functions
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, InputLayer, Dropout,GaussianNoise
from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from tensorflow.keras.applications import VGG16, ResNet50
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam


# %%
# define path to image folders
img_folders_path = '256_ObjectCategories/sim_images/'

# %%
# from tensorflow.python.ops.numpy_ops import np_config
# np_config.enable_numpy_behavior()

# %%
# load data
datagen = ImageDataGenerator(rescale=1./255,
# converts pixels in range 0,255 to between 0 and 1
                             # this will make every image contribute more evenly
                             #to the total loss
                             validation_split=0.2)

# if want to implement gray scale:
def to_grayscale(image):
    image = tf.image.rgb_to_grayscale(image)
    return image

# then use preprocessing_function=to_grayscale in ImageDataGenerator
traingen_g = ImageDataGenerator(rescale=1./255,
                                validation_split = 0.2,
                                preprocessing_function=to_grayscale)

# if split into train and test dirs, could apply brightness range to
# only test set using "brightness_range" option in ImageDataGenerator
traingen_b = ImageDataGenerator(rescale=1./255,
                                validation_split = 0.2,
                                brightness_range = [0.5,1.5])
```

```
train_generator = datagen.flow_from_directory(
    img_folders_path,
    target_size=(150, 150),
    batch_size=32,
    shuffle = True,
    class_mode='categorical',
    subset='training'
)

# in case I want to experiment with brightness as an augmentation
# train_generator_b = traingen_b.flow_from_directory(
#     img_folders_path,
#     target_size=(150, 150),
#     batch_size=32,
#     shuffle = True,
#     class_mode='categorical',
#     subset='training'
# )

# or with gray scale as augmentation
# train_generator_g = traingen_g.flow_from_directory(
#     img_folders_path,
#     target_size=(150, 150),
#     batch_size=32,
#     shuffle = True,
#     class_mode='categorical',
#     subset='training'
# )


# will use this validation set as a test set, as val metrics can be used
# for early stopping and ability for tuning number of epochs
# but do not affect training (according to search I made)
test_generator = datagen.flow_from_directory(
    img_folders_path,
    target_size=(150, 150),
    batch_size=32,
    shuffle = True,
    class_mode='categorical',
    subset='validation',
)

# %%
# define important parameters
num_classes = len(np.unique(train_generator.classes))

# %%
# create early stopping
es = EarlyStopping(monitor='loss',patience=3,mode='min')
# could also add "start_from_epoch" specification in this es object
# in .fit() add the following to implement early stopping:
# callbacks=[es]

# %%
```

```
model_ffn = Sequential([
    RandomContrast(0, input_shape=(150, 150, 3)),
    GaussianNoise(0),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.4),
    # GaussianNoise(noise_sd),
    Dense(256, activation='relu'),
    Dropout(0.2),
    # GaussianNoise(noise_sd),
    Dense(128, activation='relu'),
    Dropout(0.1),
    Dense(num_classes, activation='softmax')
])
# there may also be some overfitting, so using some dropout could be helpful

model_ffn.compile(optimizer=Adam(learning_rate=0.0001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

model_ffn.fit(train_generator, epochs=10,
              #   steps_per_epoch=10,
                  callbacks=[es])
model_ffn.summary()


# %%
model_cnn = Sequential([
    # syntax: Conv2D(num_filters, kernel_size (shape1,shape2), activation,
    # input_shape for first layer)
    # RandomContrast(0.2, input_shape=(150, 150, 3)),
    # GaussianNoise(noise_sd),
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),
    # GaussianNoise(noise_sd),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    # GaussianNoise(noise_sd),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    # after convolutional layers, flatten the output, then use 1-2(/3?) dense layers
    Flatten(),
    # GaussianNoise(noise_sd),
    Dense(512, activation='relu'),
    Dropout(0.4),
    # GaussianNoise(noise_sd),
    Dense(256, activation='tanh'),
    Dropout(0.2),
    # GaussianNoise(noise_sd),
    # Dense(128, activation='relu'),
    # Dropout(0.1),
    Dense(num_classes, activation='softmax')
])
```

```
model_cnn.compile(optimizer=Adam(learning_rate=0.0001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

model_cnn.fit(train_generator, epochs=10,
              callbacks = [es])
# default is to take len(train_generator steps per epoch)
model_cnn.summary()

# %%
# # save and restore models
# model_ffn.save("ffn_model")
# model_ffn_nc.save("ffn_model_nc")
# model_ffn_c.save("ffn_model_c")
# model_ffn_n.save("ffn_model_n")

# model_cnn.save("cnn_model")
# model_cnn_nc.save("cnn_model_nc")
# model_cnn_c.save("cnn_model_c")
# model_cnn_n.save("cnn_model_n")

# # reload the model
# model_ffn_reload = tf.keras.models.load_model('ffn_model')
# model_ffn_nc_reload = tf.keras.models.load_model('ffn_model_nc')
# model_cnn_reload = tf.keras.models.load_model('cnn_model')
# saver = tf.train.Saver(max_to_keep=1)
# with tf.Session() as sess:
#     # train your model, then:
#     savePath = saver.save(sess, 'my_ffn.ckpt')
# # To restore:

# with tf.Session() as sess:
#     saver = tf.train.import_meta_graph('my_ffn.ckpt.meta')
#     saver.restore(sess, pathModel + 'someDir/my_model.ckpt')
#     # access a variable from the saved Graph, and so on:
#     someVar = sess.run('varName:0')

# %%
# results = model_ffn_reload.predict(test_generator)


# %%
# test out functionality to
n_sim = 2
test_accs = [np.nan]*n_sim

# %%
# evaluate the model
# _, train_acc = model_ffn.evaluate(train_generator, verbose=0)
_, test_accs[j] = model_ffn_reload.evaluate(test_generator)
_, test_accs[1] = model_ffn_nc_reload.evaluate(test_generator)
print(test_accs)

# %%
```

```
pred_digits = np.argmax(results, axis=1)
test_labels = test_generator.classes
len(test_labels)
len(pred_digits)

# %%
# combine model fits and accuracy assessments into one function
def network_sim(train_generator, test_generator, n_epochs, noise_sd=0, contrast_factor=0):
    """Fit Feed Forward and Convolutional Neural Networks with 2 factors:
    # Random Noise and Random Contrast (changed to gray scale)
    parameters:
        train_generator
        test_generator
        noise_sd: float [0.0,1.0], default = 0
            sd of Gaussian(0,sd) random noise to be added during training
        contrast_factor: float [0.0,1.0], defauly = 0
            factor to scale random contrast adjustment of images;
            factor applied using following formula:
            layer computes the mean of the image pixels in the channel and then adjusts
            each component x
            of each pixel to (x - mean) * contrast_factor + mean
    returns: *dictionary* {FFN_accuracy, CNN_accuracy}
    """
    # get number of classes in training data
    num_classes = len(np.unique(train_generator.classes))
    # define early stopping criteria
    es = EarlyStopping(monitor='loss',patience=3,mode='min')

    # feed forward nn fit
    model_ffn_nc = Sequential([
        RandomContrast(factor=contrast_factor, input_shape=(150, 150, 3)),
        GaussianNoise(noise_sd),
        Flatten(),
        Dense(512, activation='relu'),
        Dropout(0.2),
        # GaussianNoise(noise_sd),
        Dense(256, activation='relu'),
        Dropout(0.1),
        # GaussianNoise(noise_sd),
        Dense(128, activation='relu'),
        Dropout(0.1),
        Dense(num_classes, activation='softmax')
    ])
    # there may also be some overfitting, so using some dropout could be helpful

    model_ffn_nc.compile(optimizer=Adam(learning_rate=0.0001),
                         loss='categorical_crossentropy',
                         metrics=['accuracy'])
    print(f'Fitting feed forward network with Gaussian(0,{noise_sd})
----noise and {contrast_factor} random contrast')
    model_ffn_nc.fit(train_generator, epochs=n_epochs,
                     #   steps_per_epoch=10,
                         callbacks=[es])
```

```
    # cnn fit
    model_cnn_nc = Sequential([
        # syntax: Conv2D(num_filters, kernel_size (shape1,shape2), activation,
        # input_shape for first layer)
        # recall: adding dropout can be helpful to remedy overfitting
        RandomContrast(factor=contrast_factor, input_shape=(150, 150, 3)),
        GaussianNoise(noise_sd),
        Conv2D(32, (3, 3), activation='relu'), #input_shape=(150, 150, 3)),
        MaxPooling2D(2, 2),
        # Dropout(0.1),
        # GaussianNoise(noise_sd),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D(2, 2),
        # Dropout(0.1),
        # GaussianNoise(noise_sd),
        Conv2D(128, (3, 3), activation='relu'),
        MaxPooling2D(2, 2),
        # Dropout(0.1),
        Conv2D(256, (3, 3), activation='relu'),
        MaxPooling2D(2, 2),
        # Dropout(0.1),
        # after convolutional layers, flatten the output, then use 1-2(/3?) dense layers
        Flatten(),
        # GaussianNoise(noise_sd),
        Dense(512, activation='relu'),
        Dropout(0.2),
        # GaussianNoise(noise_sd),
        Dense(256, activation='tanh'),
        Dropout(0.2),
        # GaussianNoise(noise_sd),
        # Dense(128, activation='relu'),
        # Dropout(0.1),
        Dense(num_classes, activation='softmax')
    ])

    model_cnn_nc.compile(optimizer=Adam(learning_rate=0.0001),
                         loss='categorical_crossentropy',
                         metrics=['accuracy'])

    print(f'Fitting CNN with Gaussian(0,{noise_sd}) noise and {contrast_factor}
----random contrast')
    model_cnn_nc.fit(train_generator, epochs=n_epochs,
                         callbacks = [es])
    # default is to take len(train_generator steps per epoch)

    # accuracy evaluations
    _, test_acc_ffn = model_ffn_nc.evaluate(test_generator)
    _, test_acc_cnn = model_cnn_nc.evaluate(test_generator)

    return {"FFN_accuracy": test_acc_ffn, "CNN_accuracy": test_acc_cnn}

# %%
# test network_sim
```

```
baseline_results = network_sim(train_generator,test_generator,n_epochs=1)
nc_results = network_sim(train_generator,test_generator,n_epochs=1,noise_sd=0.2,
contrast_factor=0.1)
n_results = network_sim(train_generator,test_generator,n_epochs=1,noise_sd=0.2,
contrast_factor=0)
c_results = network_sim(train_generator,test_generator,n_epochs=1,noise_sd=0,
contrast_factor=0.1)


# %%
# test list structure before aplying in loop
n_sim = 3
ffn_base_accs = [np.nan]*n_sim
ffn_nc_accs = [np.nan]*n_sim
ffn_n_accs = [np.nan]*n_sim
ffn_c_accs = [np.nan]*n_sim
cnn_base_accs = [np.nan]*n_sim
cnn_nc_accs = [np.nan]*n_sim
cnn_n_accs = [np.nan]*n_sim
cnn_c_accs = [np.nan]*n_sim

ffn_base_accs[0] = baseline_results["FFN_accuracy"]
ffn_nc_accs[0] = nc_results["FFN_accuracy"]
ffn_n_accs[0] = n_results["FFN_accuracy"]
ffn_c_accs[0] = c_results["FFN_accuracy"]

cnn_base_accs[0] = baseline_results["CNN_accuracy"]
cnn_nc_accs[0] = nc_results["CNN_accuracy"]
cnn_n_accs[0] = n_results["CNN_accuracy"]
cnn_c_accs[0] = c_results["CNN_accuracy"]

# %%
# repeat process for simulation study
# then can also decide if we want to up noise/contrast or change filter

n_sims = 20
# initialize result lists
ffn_base_accs = [np.nan]*n_sim
ffn_nc_accs = [np.nan]*n_sim
ffn_n_accs = [np.nan]*n_sim
ffn_c_accs = [np.nan]*n_sim
cnn_base_accs = [np.nan]*n_sim
cnn_nc_accs = [np.nan]*n_sim
cnn_n_accs = [np.nan]*n_sim
cnn_c_accs = [np.nan]*n_sim
for j in range(n_sims):
    print(f"run {j+1}")
    # train networks
    baseline_results = network_sim(train_generator,test_generator,n_epochs=1)
    nc_results = network_sim(train_generator,test_generator,n_epochs=1,noise_sd=0.2,
    contrast_factor=0.1)
    n_results = network_sim(train_generator,test_generator,n_epochs=1,noise_sd=0.2,
    contrast_factor=0)
    c_results = network_sim(train_generator,test_generator,n_epochs=1,noise_sd=0,c
```

```
    ontrast_factor=0.1)

    # export results
    ffn_base_accs[j] = baseline_results["FFN_accuracy"]
    ffn_nc_accs[j] = nc_results["FFN_accuracy"]
    ffn_n_accs[j] = n_results["FFN_accuracy"]
    ffn_c_accs[j] = c_results["FFN_accuracy"]

    cnn_base_accs[j] = baseline_results["CNN_accuracy"]
    cnn_nc_accs[j] = nc_results["CNN_accuracy"]
    cnn_n_accs[j] = n_results["CNN_accuracy"]
    cnn_c_accs[j] = c_results["CNN_accuracy"]

# %%
# save results as pickles
# ('wb': 'write byte/binary' => write using byte data: in binary mode)
# ffn
fileObj = open('ffn_base_accs.obj', 'wb')
pickle.dump(ffn_base_accs, fileObj)
fileObj.close()

fileObj = open('ffn_nc_accs.obj', 'wb')
pickle.dump(ffn_nc_accs, fileObj)
fileObj.close()

fileObj = open('ffn_n_accs.obj', 'wb')
pickle.dump(ffn_n_accs, fileObj)
fileObj.close()

fileObj = open('ffn_c_accs.obj', 'wb')
pickle.dump(ffn_c_accs, fileObj)
fileObj.close()

# cnn
fileObj = open('cnn_base_accs.obj', 'wb')
pickle.dump(ffn_base_accs, fileObj)
fileObj.close()

fileObj = open('cnn_nc_accs.obj', 'wb')
pickle.dump(ffn_nc_accs, fileObj)
fileObj.close()

fileObj = open('cnn_n_accs.obj', 'wb')
pickle.dump(ffn_n_accs, fileObj)
fileObj.close()

fileObj = open('cnn_c_accs.obj', 'wb')
pickle.dump(ffn_c_accs, fileObj)
fileObj.close()

# %%
# deserialize (reload) objects
fileObj = open('./shortsim2/fn_base_accs.obj', 'rb')
ffn_base_accs_reload2 = pickle.load(fileObj)
```

```
fileObj.close()
print(ffn_base_accs_reload2)

fileObj = open('./shortsim2/ffn_c_accs.obj', 'rb')
ffn_c_accs_reload2 = pickle.load(fileObj)
fileObj.close()

fileObj = open('./shortsim2/fn_n_accs.obj', 'rb')
ffn_n_accs_reload2 = pickle.load(fileObj)
fileObj.close()

fileObj = open('./shortsim2/ffn_nc_accs.obj', 'rb')
ffn_nc_accs_reload2 = pickle.load(fileObj)
fileObj.close()

fileObj = open('./shortsim2/cnn_base_accs.obj', 'rb')
cnn_base_accs_reload2 = pickle.load(fileObj)
fileObj.close()
print(cnn_base_accs_reload2)

fileObj = open('./shortsim2/cnn_c_accs.obj', 'rb')
cnn_c_accs_reload2 = pickle.load(fileObj)
fileObj.close()

fileObj = open('./shortsim2/cnn_n_accs.obj', 'rb')
cnn_n_accs_reload2 = pickle.load(fileObj)
fileObj.close()

fileObj = open('./shortsim2/cnn_nc_accs.obj', 'rb')
cnn_nc_accs_reload2 = pickle.load(fileObj)
fileObj.close()


print("ANN-BASE-accuracy:", np.mean(ffn_base_accs_reload2),
"with-sd:", np.std(ffn_base_accs_reload2))
print("CNN-BASE-accuracy:", np.mean(cnn_base_accs_reload2),
"with-sd:", np.std(cnn_base_accs_reload2))
print("ANN-NOISY-accuracy:", np.mean(ffn_n_accs_reload2),
"with-sd:", np.std(ffn_n_accs_reload2))
print("CNN-NOISY-accuracy:", np.mean(cnn_n_accs_reload2),
"with-sd:", np.std(cnn_n_accs_reload2))
print("ANN-GRAY-accuracy:", np.mean(ffn_c_accs_reload2),
"with-sd:", np.std(ffn_c_accs_reload2))
print("CNN-GRAY-accuracy:", np.mean(cnn_c_accs_reload2),
"with-sd:", np.std(cnn_c_accs_reload2))
print("ANN-NOISY-GRAY-accuracy:", np.mean(ffn_nc_accs_reload2),
"with-sd:", np.std(ffn_nc_accs_reload2))
print("CNN-NOISY-GRAY-accuracy:", np.mean(cnn_nc_accs_reload2),
"with-sd:", np.std(cnn_nc_accs_reload2))

# %%
# deserialize (reload) objects
# original sim: 12 epochs, 20 sims, to gray scale as in to 2d, rather than b&w
fileObj = open('./pickles_shortsim/ffn_base_accs.obj', 'rb')
```

```
ffn_base_accs_reload = pickle.load(fileObj)
fileObj.close()
print(ffn_base_accs_reload)

fileObj = open('./pickles_shortsim/ffn_c_accs.obj', 'rb')
ffn_c_accs_reload = pickle.load(fileObj)
fileObj.close()

fileObj = open('./pickles_shortsim/ffn_n_accs.obj', 'rb')
ffn_n_accs_reload = pickle.load(fileObj)
fileObj.close()

fileObj = open('./pickles_shortsim/ffn_nc_accs.obj', 'rb')
ffn_nc_accs_reload = pickle.load(fileObj)
fileObj.close()

fileObj = open('./pickles_shortsim/cnn_base_accs.obj', 'rb')
cnn_base_accs_reload = pickle.load(fileObj)
fileObj.close()
print(cnn_base_accs_reload)

fileObj = open('./pickles_shortsim/cnn_c_accs.obj', 'rb')
cnn_c_accs_reload = pickle.load(fileObj)
fileObj.close()

fileObj = open('./pickles_shortsim/cnn_n_accs.obj', 'rb')
cnn_n_accs_reload = pickle.load(fileObj)
fileObj.close()

fileObj = open('./pickles_shortsim/cnn_nc_accs.obj', 'rb')
cnn_nc_accs_reload = pickle.load(fileObj)
fileObj.close()


print("ANN-BASE-accuracy:", np.mean(ffn_base_accs_reload),
"with-sd:", np.std(ffn_base_accs_reload))
print("CNN-BASE-accuracy:", np.mean(cnn_base_accs_reload),
"with-sd:", np.std(cnn_base_accs_reload))
print("ANN-NOISY-accuracy:", np.mean(ffn_n_accs_reload),
"with-sd:", np.std(ffn_n_accs_reload))
print("CNN-NOISY-accuracy:", np.mean(cnn_n_accs_reload),
"with-sd:", np.std(cnn_n_accs_reload))
print("ANN-GRAY-accuracy:", np.mean(ffn_c_accs_reload),
"with-sd:", np.std(ffn_c_accs_reload))
print("CNN-GRAY-accuracy:", np.mean(cnn_c_accs_reload),
"with-sd:", np.std(cnn_c_accs_reload))
print("ANN-NOISY-GRAY-accuracy:", np.mean(ffn_nc_accs_reload),
"with-sd:", np.std(ffn_nc_accs_reload))
print("CNN-NOISY-GRAY-accuracy:", np.mean(cnn_nc_accs_reload),
"with-sd:", np.std(cnn_nc_accs_reload))

# %%
sim_results_dict = {"Method": ["ANN_BASE","ANN_NOISY","ANN_GRAY", "ANN_NOISY_GRAY",
"CNN_BASE","CNN_NOISY","CNN_GRAY", "CNN_NOISY_GRAY"],
```

```
"Mean Accuracy":[np.mean(ffn_base_accs_reload),np.mean(ffn_n_accs_reload),
np.mean(ffn_c_accs_reload),np.mean(ffn_nc_accs_reload),
                                        np.mean(cnn_base_accs_reload),
                                        np.mean(cnn_n_accs_reload),np.mean(cnn_c_accs_reload)
                                        np.mean(cnn_nc_accs_reload)],
"SD Accuracy": [np.std(ffn_base_accs_reload),np.std(ffn_n_accs_reload),
np.std(ffn_c_accs_reload),np.std(ffn_nc_accs_reload),
                                       np.std(cnn_base_accs_reload),np.std(cnn_n_accs_reload)
                                       np.std(cnn_c_accs_reload),np.std(cnn_nc_accs_reload)]}
sim_res_df = pd.DataFrame(sim_results_dict)
sim_res_df

# %%
# output table to latex
sim_res_df.to_latex

# %%
# deserialize (reload) objects
# corrected sim: 12 epochs, 20 sims, fully to gray scale, back as 3d rgb image
fileObj = open('./sim_gray_corrected/ffn_base_accs.obj', 'rb')
ffn_base_accs_reloadc = pickle.load(fileObj)
fileObj.close()
print(ffn_base_accs_reloadc)

fileObj = open('./sim_gray_corrected/ffn_c_accs.obj', 'rb')
ffn_c_accs_reloadc = pickle.load(fileObj)
fileObj.close()

fileObj = open('./sim_gray_corrected/ffn_n_accs.obj', 'rb')
ffn_n_accs_reloadc = pickle.load(fileObj)
fileObj.close()

fileObj = open('./sim_gray_corrected/ffn_nc_accs.obj', 'rb')
ffn_nc_accs_reloadc = pickle.load(fileObj)
fileObj.close()

fileObj = open('./sim_gray_corrected/cnn_base_accs.obj', 'rb')
cnn_base_accs_reloadc = pickle.load(fileObj)
fileObj.close()
print(cnn_base_accs_reloadc)

fileObj = open('./sim_gray_corrected/cnn_c_accs.obj', 'rb')
cnn_c_accs_reloadc = pickle.load(fileObj)
fileObj.close()

fileObj = open('./sim_gray_corrected/cnn_n_accs.obj', 'rb')
cnn_n_accs_reloadc = pickle.load(fileObj)
fileObj.close()

fileObj = open('./sim_gray_corrected/cnn_nc_accs.obj', 'rb')
cnn_nc_accs_reloadc = pickle.load(fileObj)
fileObj.close()
```

```
print("ANN-BASE accuracy:", np.mean(ffn_base_accs_reloadc),
"with sd:", np.std(ffn_base_accs_reloadc))
print("CNN-BASE accuracy:", np.mean(cnn_base_accs_reloadc),
"with sd:", np.std(cnn_base_accs_reloadc))
print("ANN-NOISY accuracy:", np.mean(ffn_n_accs_reloadc),
"with sd:", np.std(ffn_n_accs_reloadc))
print("CNN-NOISY accuracy:", np.mean(cnn_n_accs_reloadc),
"with sd:", np.std(cnn_n_accs_reloadc))
print("ANN-GRAY accuracy:", np.mean(ffn_c_accs_reloadc),
"with sd:", np.std(ffn_c_accs_reloadc))
print("CNN-GRAY accuracy:", np.mean(cnn_c_accs_reloadc),
"with sd:", np.std(cnn_c_accs_reloadc))
print("ANN-NOISY-GRAY accuracy:", np.mean(ffn_nc_accs_reloadc),
"with sd:", np.std(ffn_nc_accs_reloadc))
print("CNN-NOISY-GRAY accuracy:", np.mean(cnn_nc_accs_reloadc),
"with sd:", np.std(cnn_nc_accs_reloadc))

# %%
sim_results_dict = {"Method": ["DNN-BASE","DNN-NOISY","DNN-GRAY", "DNN-NOISY+GRAY",
"CNN-BASE","CNN-NOISY","CNN-GRAY", "CNN-NOISY+GRAY"],
"Mean Accuracy":[np.mean(ffn_base_accs_reloadc),np.mean(ffn_n_accs_reloadc),
np.mean(ffn_c_accs_reloadc),np.mean(ffn_nc_accs_reloadc),
np.mean(cnn_base_accs_reloadc),np.mean(cnn_n_accs_reloadc),np.mean(cnn_c_accs_reloadc),
np.mean(cnn_nc_accs_reloadc)],
"SD": [np.std(ffn_base_accs_reloadc),np.std(ffn_n_accs_reloadc),
np.std(ffn_c_accs_reloadc),np.std(ffn_nc_accs_reloadc),
np.std(cnn_base_accs_reloadc),np.std(cnn_n_accs_reloadc),
np.std(cnn_c_accs_reloadc),np.std(cnn_nc_accs_reloadc)]}
sim_res_dfc = pd.DataFrame(sim_results_dict)
sim_res_dfc

# output table to latex
print(sim_res_dfc.style.to_latex(position = 't'))

# %%
# attempt to look at confusion matrix/classification report
# predictions appear to be in shuffled order,
# so performance not matching what is shown when model.evaluate() is run

# test_labels = test_generator.classes_
# pred_digits = np.argmax(model_cnn.predict(test_generator),axis=1)
# from sklearn.metrics import classification_report, confusion_matrix
# confusion_matrix(y_true=test_labels,y_pred=pred_digits)
# print(classification_report(test_labels,pred_digits))

# real data example
img_folders_path = '256_ObjectCategories/sim_images'

# %%
# format data for training
# load data
datagen = ImageDataGenerator(rescale=1./255,
# converts pixels in range 0,255 to between 0 and 1
                             # this will make every image contribute more evenly
```

```
                                    # to the total loss
                                    validation_split=0.2)

train_generator = datagen.flow_from_directory(
    img_folders_path,
    target_size=(150, 150),
    batch_size=32,
    # shuffle = True,
    class_mode='categorical',
    subset='training'
)


# will use this validation set as a test set, as val metrics can be used for
# early stopping and ability for tuning,
# but do not affect training (according to search I made)
val_generator = datagen.flow_from_directory(
    img_folders_path,
    target_size=(150, 150),
    batch_size=32,
    # shuffle = True,
    class_mode='categorical',
    subset='validation',
)

# cnn fit
model_cnn = Sequential([
    # syntax: Conv2D(num_filters, kernel_size (shape1,shape2), activation,
    # input_shape for first layer)
    # recall: adding dropout can be helpful to remedy overfitting
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D(2, 2),
    #Dropout(0.1),
    # GaussianNoise(noise_sd),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    # Dropout(0.1),
    # GaussianNoise(noise_sd),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    # Dropout(0.1),
    Conv2D(256, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    # Dropout(0.1),
    # after convolutional layers, flatten the output, then use 1-2(/3?) dense layers
    Flatten(),
    # GaussianNoise(noise_sd),
    Dense(512, activation='relu'),
    Dropout(0.4),
    # GaussianNoise(noise_sd),
    Dense(256, activation='relu'),
    Dropout(0.2),
    # GaussianNoise(noise_sd),
    Dense(128, activation='relu'),
```

```
    Dropout(0.1),
    Dense(num_classes, activation='softmax')
])

# model_cnn_nc.compile(optimizer='SGD',
#                      loss='categorical_crossentropy',
#                      metrics=['accuracy'])

model_cnn.compile(optimizer=Adam(learning_rate = 0.0001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
n_epochs = 50
model_cnn.fit(train_generator, validation_data=val_generator, epochs=n_epochs,
                   callbacks = [es])


# save model history
losses = model_cnn.history.history

fileObj = open('cnn_losses_real.obj', 'wb')
pickle.dump(losses, fileObj)
fileObj.close()

n_epochs = 40
model_cnn.fit(train_generator, validation_data=val_generator, epochs=n_epochs,
                   callbacks = [es])

# save model history
losses = model_cnn.history.history

fileObj = open('losses_cnn.obj', 'wb')
pickle.dump(losses, fileObj)
fileObj.close()


# fit model from pretraining VGG16
from keras.applications import VGG16

# Load the VGG16 base model without the top classification layer
base_model_pt = VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))
# download link finishes with "_notop"

# Freeze all layers in the base model to retain their learned weights
for layer in base_model_pt.layers:
    layer.trainable = False

# Optionally, unfreeze the top N layers
# N = 5
# for layer in base_model_pt.layers[-N:]:
#     layer.trainable = True

# Add custom layers on top
x = base_model_pt.output
x = Flatten()(x) # try both Flatten and GlobalAveragePooling2D
```

```
# x = GlobalAveragePooling2D(2,2)
x = Dense(512, activation='relu')(x)
x= Dropout(0.4)(x)
x = Dense(512, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)

model_transfer = Model(inputs=base_model_pt.input, outputs=predictions)

# Compile with a smaller learning rate
model_transfer.compile(optimizer=Adam(learning_rate=0.0001),
                       loss='categorical_crossentropy',
                       metrics=['accuracy'])

model_transfer.fit(train_generator, validation_data=val_generator, epochs=12,
                   )

# save model object
model_transfer.save("vgg16_transfer")

# save model history
t_losses = model_transfer.history.history

fileObj = open('model_t_losses.obj', 'wb')
pickle.dump(t_losses, fileObj)
fileObj.close()

# get preds and save pred digits
# for val data
labels = val_generator.classes
# print(labels)
res = model_transfer.predict(val_generator)
# print(res)
pred_digits = np.argmax(res,axis=1)

# save pred digits
fileObj = open('model_t_digits.obj', 'wb')
pickle.dump(pred_digits, fileObj)
fileObj.close()

# reload model object
cnnObj = open('./real_data_eg/model_cnn.obj','rb')
model_cnn_reload = pickle.load(cnnObj)
cnnObj.close()

# %%
# reload model performance (losses and accuracies from 50 epoch run on server)
lossesObj = open('./real_data_eg/cnn_losses_real.obj','rb')
model_cnn_losses_reload = pickle.load(lossesObj)
lossesObj.close()

# 40 epoch run
lossesObj = open('./real_data_eg/losses_cnn.obj','rb')
model_cnn_losses_reload2 = pickle.load(lossesObj)
lossesObj.close()
```

```
# reload losses for VGG16 run
lossesObj = open('./real_data_eg/model_t_losses.obj','rb')
model_cnn_losses_reloadt = pickle.load(lossesObj)
lossesObj.close()


# %%
print(model_cnn_losses_reload)
print(model_cnn_losses_reload2)
print(model_cnn_losses_reloadt)

# %%
# plot model performance
# evaluation
# example code
losses_long = model_cnn_losses_reload
# print(losses_long)
df_loss_l = pd.DataFrame(losses_long)
display(df_loss_l.sort_values(by='val_accuracy',ascending=False).head())

df_loss_l.plot(
    figsize=(8, 6), grid=True, xlabel="Epoch",
    style=["r-", "r--.", "b-", "b-*"])
plt.legend(loc=(1.01,0))  # extra code
 # extra code
plt.show()
# plt.savefig("cnn_learning_curves_plot.png")

# %%
# plot model performance
# evaluation
# example code
losses_long2 = model_cnn_losses_reload2
# print(losses_long)
df_loss_l2 = pd.DataFrame(losses_long2)
display(df_loss_l2.sort_values(by='val_accuracy',ascending=False).head())

df_loss_l2.plot(
    figsize=(8, 6), grid=True, xlabel="Epoch",
    style=["r-", "r--.", "b-", "b-*"])
plt.legend(loc=(1.01,0))  # extra code
 # extra code
plt.show()
# plt.savefig("cnn_learning_curves_plot.png")

# %%
# plot model performance
# evaluation
# example code
losses_t = model_cnn_losses_reloadt
# print(losses_long)
df_loss_t = pd.DataFrame(losses_t)
display(df_loss_t.sort_values(by='val_accuracy',ascending=False).head())
```

```
df_loss_t.plot(
    figsize=(8, 6), grid=True, xlabel="Epoch",
    title='VGG16 learning curves',
    style=["r-", "r--.", "b-", "b-*"])
plt.legend(loc=(1.01,0))  # extra code
plt.show()


# %%
## reload model to get predictions and classification report
## this still had issues, appears predictions in shuffled order
## from my cnn
# fileObj = open('real_data_eg/cnn_val_digits.obj','rb')
# mycnn_digits = pickle.load(fileObj)
# fileObj.close()

## from VGG16 model transfer
# fileObj = open('real_data_eg/model_t_digits.obj','rb')
# t_digits = pickle.load(fileObj)
# fileObj.close()

## %%
## get classification report
## test_labels = val_generator.classes
## pred_digits = np.argmax(model_cnn_nc.predict(val_generator),axis=1)
## for all data
# labels =val_generator.classes
## print(labels)
# res = model_ffn_nc.predict(all_dat_generator)
## print(res)
# pred_digits = np.argmax(res,axis=1)
## print(pred_digits)
# print(confusion_matrix(y_true=labels,y_pred=pred_digits))
# print(classification_report(labels,pred_digits))

## %%
## cnn
# labels = val_generator.classes
## print(labels)
## res = model_cnn_nc.predict(all_dat_generator)
## print(res)
## pred_digits = np.argmax(res,axis=1)
## print(pred_digits)
# print(confusion_matrix(y_true=labels,y_pred=t_digits))
# print(classification_report(labels,t_digits,))
```