Image Classification Transfer Learning

Definition

Project Overview

Computer vision is one of the fields where machine learning has made great strides in this decade. It involves gaining high-level understanding from digital images or videos. It can be thought of as an application of artificial intelligence to automate the tasks that the human visual system can do. Computer vision itself has many sub-domains depending on the objective one is trying to achieve. Some of these are scene reconstruction, event detection, object recognition, motion estimation, image restoration[1] etc. Object recognition can also be called object or image classification and the one we shall focus on here is with classify multiple types of images with labels and hence is a case of multi-label classification.

Image classification is important as it helps in other areas of computer vision like object detection and localization used in applications like self driving cars. Large amounts of effort have gone into research of this area and huge datasets have been collected for benchmark to measure the progress. The breakthrough happened with the introduction of Convolution Neural Networks(CNNs) for image classification task and LeNet architecture[2] was applied to classify MNIST dataset. More bigger datasets like ImageNet[3] came after that and the architectures applied on this benchmark dataset changed the course of AI. Some state of the art CNN architectures used on ImageNet are AlexNet[4], ZFNet[5], VGGNet[6], GoogLeNet[7], ResNet[8] etc. The history shows the importance and efforts put in the task of image classification. We shall use CIFAR-100 dataset as our input data and is available for download here [9].

Problem Statement

Given a dataset of images with multiple categories, train a model to classify the images into their respective categories. The problem is best resolved by applying CNN but finding a good architecture is the challenge. In our particular case, our goal is to correctly classify as many as CIFAR-100 images as possible.

Many state of the art CNN architectures have been designed and used to get high classification performance on CIFAR-100. The top performing benchmarks available are listed in [12]. Deep neural networks like DenseNet and Wide ResNet have reported over 80% accuracy on the test set. But these architectures have high number of parameters and require high computing power like GPUs and also long duration to train. So, we would explore other options like transfer

learning with bottleneck features extraction and fine tuning to train a model in shorter period of time and also achieve a decent performance.

Metrics

Different metrics exist for comparing performance of deep learning models and their suitability depends on the distribution of data to be examined. The CIFAR-100 dataset has equal number of images for each class and superclass defined. So, the data is uniformly distributed among the categories and we do not have to consider metrics like precision, recall, F-score, etc. that are essential for skewed distribution. We shall use accuracy as our metric for comparing the performance of models. Accuracy is defined as

$$accuracy = correct\ predictions / total\ predictions$$

We shall calculate accuracies on both the sets of training examples and test examples. We shall mention training accuracy as accuracy and test accuracy as validation accuracy in our code and notebooks. The performance shall be compared based on the validation accuracy as the models can very easily overfit on the training data. This distinction shall be same for both the coarse label and fine label predictions and the accuracies shall be calculated separately.

Analysis

Data Exploration

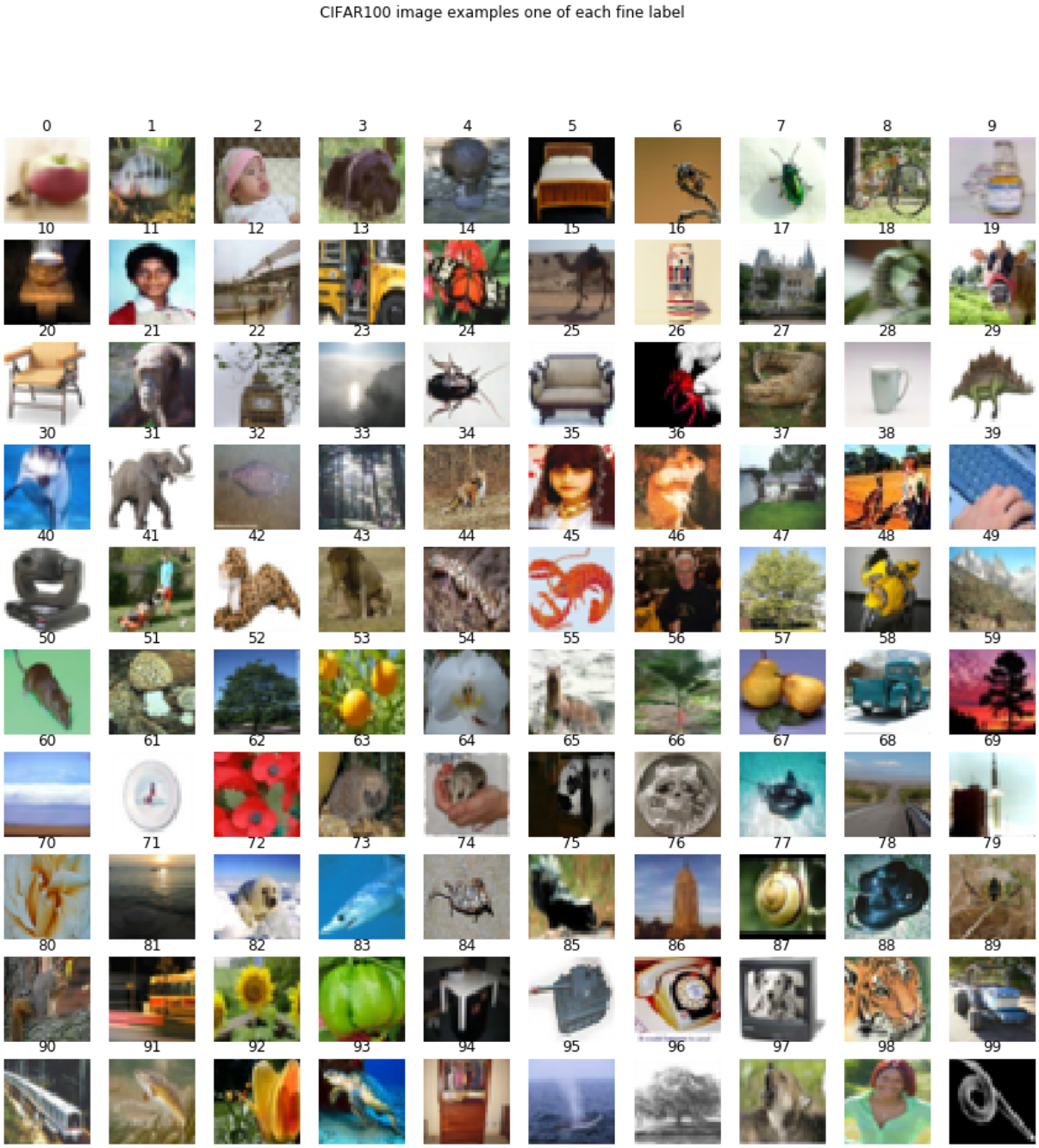
CIFAR-100 is a labeled subset of the 80 million tiny images dataset[10] and was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton[11]. The dataset consists of 60000 32x32 colour images of which 50000 are training images and 10000 are test images. CIFAR-100 has 100 classes with 600 images per class and 20 superclasses.

The classes are stored as fine labels and superclasses as coarse labels. So, each CIFAR-100 image has two labels associated with it. We have done classification for both the labels in our code. The list of fine and coarse label categories can be viewed in the data_exploration.ipynb notebook at our code repository[13] in github. We also have listed samples belonging to each one of the fine labels there.

Exploratory Visualisation

There is not much statistical data of significance to visualize in the CIFAR-100 dataset, as there are 100 classes with 600 images belonging to each class. However, each superclass has different number of classes and the superclass to class mapping is shown in [9]. The following image consists of sample images belonging to each one of the 100 classes with the class index

to provide us a glimpse of the data. The index to class name mappings are available at data_exploration.ipynb in [13].



Algorithms and Techniques

We shall train a basic CNN and also state of the art Wide ResNet from scratch for comparison purposes. But our prime focus here is to explore the techniques of transfer learning using bottleneck feature extraction and fine tuning.

Training SoTA CNN models from scratch on very large datasets would surely give better performance than the techniques discussed. But they most of the times require high computational power setups with GPUs and also a long duration to train. They may even give poor performance if the dataset is not large enough to learn the parameters. So, the techniques discussed may help us to leverage pre-trained models parameters to our advantage. We may use the parameters of these pre-trained models to learn features of our dataset in smaller duration of time and less computational power, and even with smaller datasets. Various possible scenarios along with suitable techniques are discussed in [14]. We shall discuss the model architectures for the fine labels here. Architectures for coarse labels is same except that the output layer has 20 units instead of 100.

Basic CNN

Figure 1 below shows the basic CNN model with four convolutional layers we are going to train.

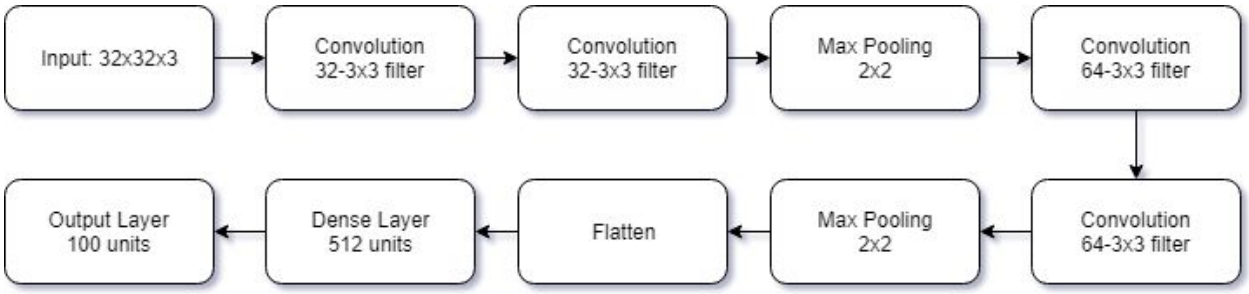


Figure 1: Basic CNN architecture for fine labels

Input is an CIFAR-100 rgb image and output is a softmax layer predicting the probability of the image belonging to each of the 100 class labels.

The next two techniques uses transfer learning on a pre-trained model. We shall use ResNet50 model pre-trained on ImageNet. Figure 2 below shows the ResNet50 architecture with the layers for ImageNet classification.

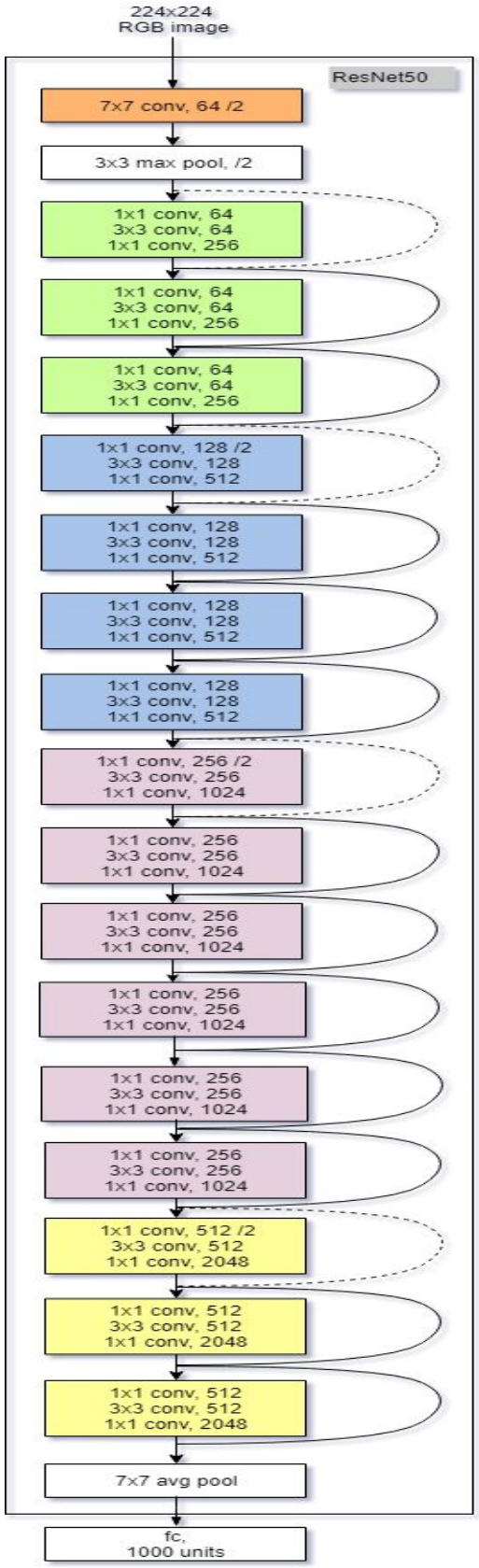


Figure 2: ResNet50 architecture pre-trained on ImageNet[28]

Bottleneck Features Extraction

This approach is suitable for datasets similar in characteristics to the pre-trained model dataset but with much smaller number of data. In this approach, we leverage a network pre-trained on a large dataset. We shall use the ResNet50 architecture shown above in Figure 2 pre-trained on ImageNet. We add a fully-connected classifier to the model to classify our input data.

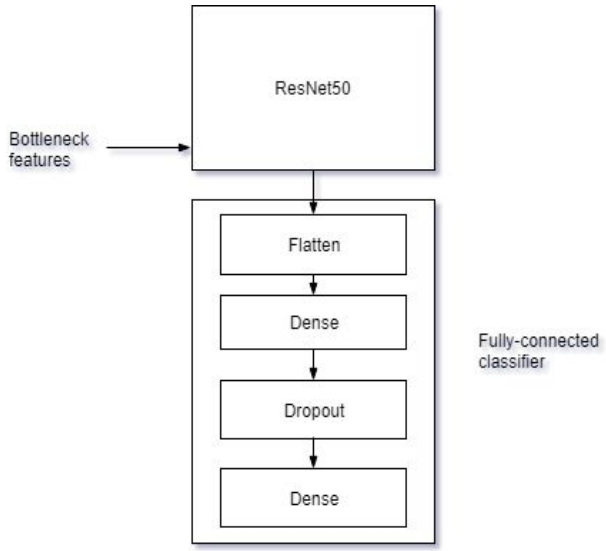


Figure 3: Bottleneck feature extraction model

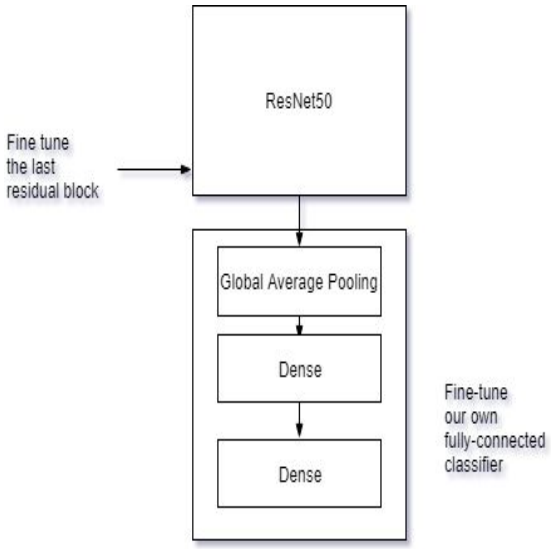


Figure 4: Fine-tuning model

We only instantiate the convolutional, i.e. ResNet50, part of the model everything up to the fully-connected layers. We then run this model on our training and validation data once, recording the output, called bottleneck features or CNN codes, in two numpy arrays. Then we train the fully-connected classifier on top of the recorded features.

Fine Tuning

This is suitable for similar datasets but with large amounts of data. Apart from removing the last classification layer and adding a new dense layer, here we also train all the previous convolutional layers or some of the last layers. Training these layers is what we refer as fine tuning. So, time required may be more than the previous technique and is also prone to overfitting in case of small datasets.

In our case, we add a fully-connected classifier as shown in Figure 4. Fine-tuning consist in starting from a trained network, then re-training it on a new dataset using very small weight updates. The steps involved are:

1. Instantiate the pre-trained ResNet50 base model and load its weights
2. Add fully-connected classifier on top and train for few epochs keeping the ResNet50 block frozen.
3. Make only the last residual block of ResNet50 trainable and fine tune the block along with the fully-connected classifier.

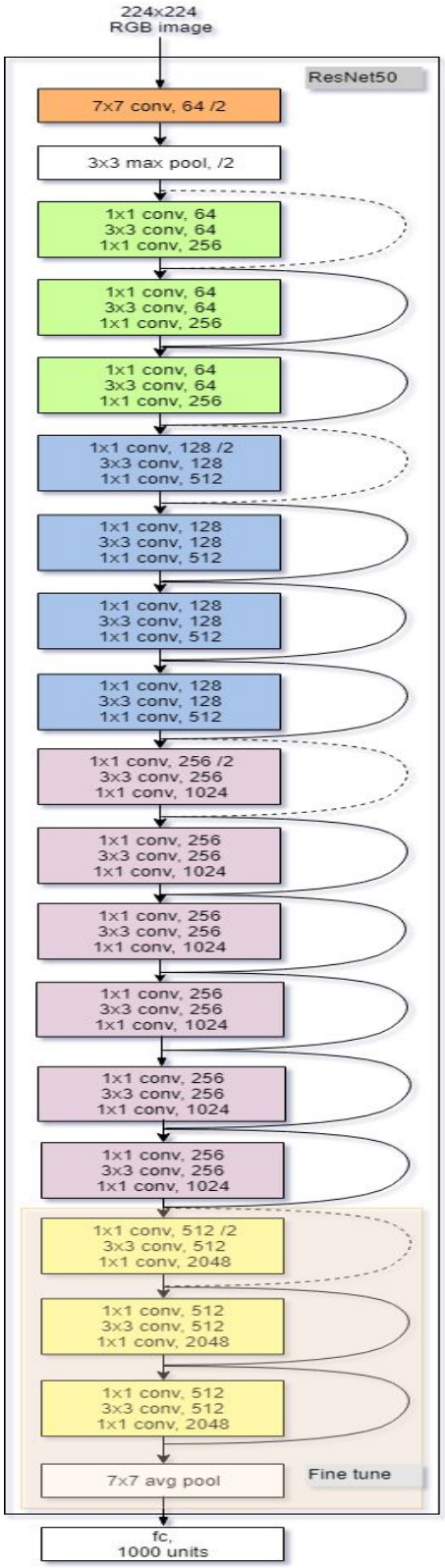


Figure 5: Last residual block of ResNet50 to fine-tune

Wide Residual Network(28-8)

Wide Residual Networks[20] are state of the art CNN architectures used for achieving high accuracy on the image classification task. They have reported validation accuracy of over 82% on the CIFAR-100 dataset. In our case, we choose WRN of depth 28 and width factor 10 to train from scratch. It is very computationally intensive and so we shall run it only for 60 epochs with a batch size of 128. We use Adam optimizer instead of SGD+Momentum for faster convergence.

Benchmark

We use the basic CNN shown in algorithms and techniques section with four convolutional layers as our benchmark model. We have taken the architecture from Keras examples available in [15]. Although the performance would be low compared to SoTA CNN models for the simple benchmark model we have chosen, it would still be better than a random guess accuracy of 1% for fine labels and 5% for coarse labels as we have 100 classes and 20 superclasses.

Methodology

Data Preprocessing

Data Preprocessing is very crucial for converting input into suitable form expected by the framework chosen and also for faster convergence.

A. Input dataset converted to (n_examples, 32, 32, 3) form as tensorflow expects ‘channels last’ dimension ordering for rgb images.[16]
B. Zero mean center and std normalization has been applied to input images as a preprocessing step in all the models.
C. Data augmentation using shifting, horizontal flip and rotation is applied for the benchmark CNN and Wide ResNet model.
D. Resizing of input images of size 32 x 32 to 224 x 224 has been done for bottleneck features extraction and fine tuning on the pretrained ResNet50 model.

Implementation

Basic CNN, bottleneck feature extraction and fine-tuning model have been trained for classifying both the coarse and fine labels. However, our primary focus shall be only on fine label classification. WRN 28-8 is trained for fine label classification and not for coarse labels. We have used Keras meta framework with Tensorflow backend and all documents and source code are available at github repo[13].

Basic CNN

Basic CNN model with architecture from Keras examples[15]. Source *cifar100_cnn.py* run for fine and coarse labels and output logs and plot available at [17] and [18]. *benchmark_kfold_fine.ipynb* is the k-fold cross validation implementation.

Basic CNN as shown in Figure 1 uses few convolutional layers with few filters per layer, alongside data augmentation and dropout. Code snippet below is the basic CNN model, a simple stack of 4 convolutional layers with a ReLU activation and followed by max-pooling layers. Architecture very similar to LeNet.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras import optimizers

# Create model
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

We stack two fully-connected layers on top and end the model with a softmax function. We use *categorical_crossentropy* loss to train our model.

```
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation('softmax'))

# Optimizer
opt = optimizers.rmsprop(lr=0.0001, decay=1e-6)

# Compile model
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

We use ImageDataGenerator for data augmentation and *.flow()* to generate batches of image data and their labels. We use these generators to train our model for 100 epochs. Each epoch takes 20-30s on GPU.

```
# Data augmentation
datagen = ImageDataGenerator(
    rescale = 1./255,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True)

datagen.fit(X_train)

X_test = X_test/255.

# Train
model.fit_generator(datagen.flow(X_train, Y_train, batch_size=batch_size),
                    steps_per_epoch=len(X_train)//batch_size + 1,
                    epochs=n_epochs,
                    validation_data=(X_test, Y_test),
                    verbose=2)
```

Bottleneck features with ResNet50

bottleneck_features_100.ipynb and *bottleneck_features_20.ipynb* are the fine and coarse label notebook implementations. *transfer_learning_100_kfold.ipynb* is the k-fold implementation.
Pretrained ResNet50 model on ImageNet from Keras applications[19] is used as base model without including the top or softmax classification layer.

```
from keras.applications.resnet50 import ResNet50
pretrained_model = ResNet50(weights='imagenet', include_top=False)
```

The *pretrained_model* expects input image of shape 224x224 and so CIFAR-100 images had to be resized.
Complication: The resizing using Keras ImageDataGenerator causes memory overflow issues as it converts image dtype to *float* from *uint8* leading to machine crash or out of memory error. So, we did not use data augmentation and created a custom generator to resize using *opencv* library and do preprocessing steps like zero centering and scaling on each image. Used small batch size of 16 in custom generator to avoid memory issues.

```
from keras.applications.resnet50 import preprocess_input
import cv2

def image_generator(images, labels):
    def generator():
        start = 0
        end = batch_size
        n = len(images)
        while True:
            img_batch_resized = np.array([cv2.resize(img, (224, 224), interpolation=cv2.INTER_CUBIC) for i
mg in images[start:end]])
            img_batch_resized = img_batch_resized.astype('float32')
            img_batch_preprocessed = preprocess_input(img_batch_resized)
            label_batch = labels[start:end]

            start += batch_size
            end += batch_size
            if start >= n:
                start = 0
                end = batch_size
            yield (img_batch_preprocessed, label_batch)
    return generator
```

We then use these generators to extract bottleneck features of our input images using *predict_generator()*.

```
train_datagen = image_generator(x_train, Y_train)
bottleneck_features_training = pretrained_model.predict_generator(train_datagen(), n_training_examples//ba
tch_size)

test_datagen = image_generator(x_test, Y_test)
bottleneck_features_testing = pretrained_model.predict_generator(test_datagen(), n_testing_examples//batch
_size)
```

We use the bottleneck features to train a small fully-connected model.

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Dropout

model = Sequential()
model.add(Flatten(input_shape=bottleneck_features_training.shape[1:]))
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes, activation='softmax'))
```

```
from keras import optimizers
model.compile(optimizer=optimizers.RMSprop(lr=2e-5), loss='categorical_crossentropy', metrics=['accuracy'
])
```

```
train_datagen = image_generator(x_train, Y_train)
test_datagen = image_generator(x_test, Y_test)

history = model.fit(bottleneck_features_training, Y_train,
          epochs=n_epochs,
          batch_size=batch_size,
          validation_data=(bottleneck_features_testing, Y_test))
```

We train the model for 30 epochs and takes only 10-20s for each epoch on GPUs.

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/30
50000/50000 [==============================] - 15s 306us/step - loss: 2.9066 - acc: 0.3134 - val_loss: 1.5
385 - val_acc: 0.6051
Epoch 2/30
50000/50000 [==============================] - 14s 290us/step - loss: 1.6061 - acc: 0.5574 - val_loss: 1.1
942 - val_acc: 0.6630
Epoch 3/30
50000/50000 [==============================] - 14s 290us/step - loss: 1.3232 - acc: 0.6199 - val_loss: 1.0
757 - val_acc: 0.6904
Epoch 4/30
50000/50000 [==============================] - 15s 299us/step - loss: 1.1798 - acc: 0.6604 - val_loss: 1.0
106 - val_acc: 0.7044
Epoch 5/30
50000/50000 [==============================] - 15s 292us/step - loss: 1.0836 - acc: 0.6825 - val_loss: 0.9
```

Fine tuning with ResNet50

fine_tuning_100.ipynb and *fine_tuning_20.ipynb* are the fine and coarse label notebook implementations.

Load pre-trained ResNet50 model as shown for bottleneck features above.

Complication: Memory issues same as for bottleneck features. Create custom datagenerator for resolution as in previous model.

Add fully-connected classifier on top of the base model.

```
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(n_classes, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)
```

Train the fully-connected classifier only after freezing all layers of base model.

```
for layer in base_model.layers:
    layer.trainable = False

model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
train_datagen = image_generator(x_train, Y_train)
```

```
test_datagen = image_generator(x_test, Y_test)
```

```
model.fit_generator(train_datagen(),
                    steps_per_epoch=n_training_examples//batch_size,
                    epochs=2,
                    validation_data=test_datagen(),
                    validation_steps=n_testing_examples//batch_size,
                    verbose=2)
```

Make the last residual block of the base model trainable and fine tune the model with very low learning rate.

```
for layer in model.layers[:142]:
    layer.trainable = False
for layer in model.layers[142:]:
    layer.trainable = True
```

```
from keras.optimizers import SGD
model.compile(optimizer=SGD(lr=0.0001, momentum=0.9), loss='categorical_crossentropy', metrics=['accuracy'
])
```

```
history = model.fit_generator(train_datagen(),
                    steps_per_epoch=n_training_examples//batch_size,
                    epochs=n_epochs,
                    validation_data=test_datagen(),
                    validation_steps=n_testing_examples//batch_size,
                    verbose=2)
```

Fine tuning the new classifier with the residual block takes over 5 minutes per epoch in GPUs and so, we train it only for 20 epochs.

Wide ResNet 28-8

wide_resnet_100.ipynb is the notebook implementation.
WRN model with 28 depth and 8 width factor is trained using using Keras contrib[21]. Keras contrib do not come with Keras and has to be installed separately.

```
sudo pip install git+https://www.github.com/keras-team/keras-contrib.git
```

We create a WRN by calling the keras contrib applications library.

```
from keras_contrib.applications import wide_resnet as wrn

model = wrn.WideResidualNetwork(depth=28, width=8, dropout_rate=0., weights=None, classes=n_classes, activ
ation='softmax')
```

Use ImageDataGenerator for data augmentation

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rotation_range=10,
                                   width_shift_range=5./32,
                                   height_shift_range=5./32,
                                   horizontal_flip=True)

train_datagen.fit(X_train, seed=0, augment=True)
```

We only train the model for 60 epochs as each epoch takes over 5 minutes on GPUs. Use Adam optimizer for faster convergence.

```
import os

wt_dir_name = 'weights'
wt_file_name = 'wrn_fine_wts.h5'
wt_file_path = os.path.join(wt_dir_name, wt_file_name)
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
from keras.callbacks import ModelCheckpoint
import time

if os.path.exists(wt_file_path):
    model.load_weights(wt_file_path)

checkpointer = ModelCheckpoint(wt_file_name, monitor="val_acc", save_best_only=True)

t_s = time.time()
history = model.fit_generator(train_datagen.flow(X_train, Y_train, batch_size=batch_size),
                    steps_per_epoch=n_training_examples//batch_size + 1,
                    epochs=n_epochs,
                    validation_data=(X_test, Y_test),
                    validation_steps=n_testing_examples//batch_size,
                    callbacks=[checkpointer],
                    verbose=2)

t_f = time.time()

print('Training time: {}s'.format(t_f - t_s))
```

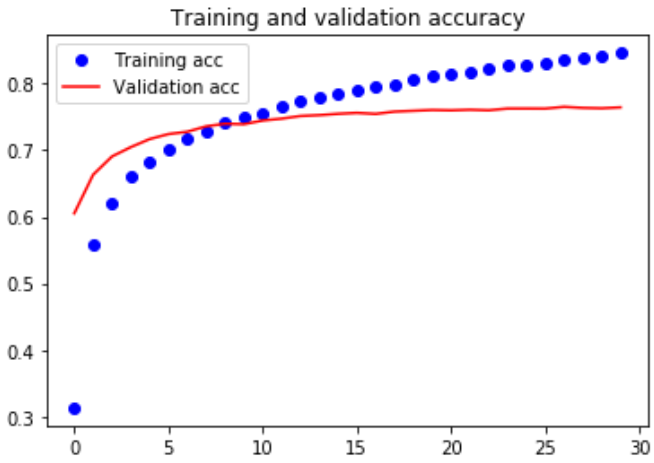
Model Evaluation and Validation

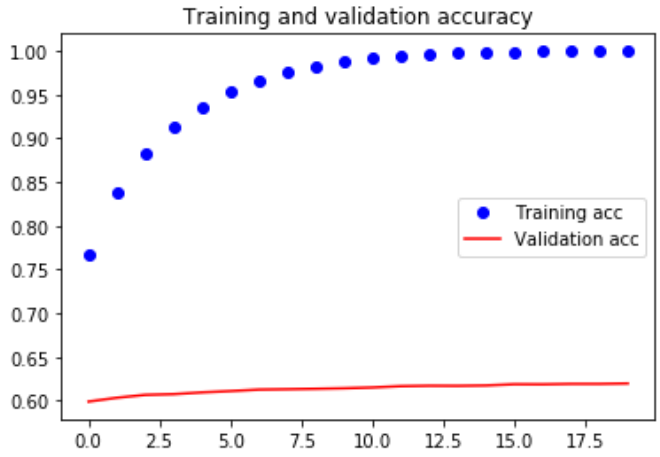
Table below shows the validation accuracy in percentage obtained by the models for coarse and fine labels.

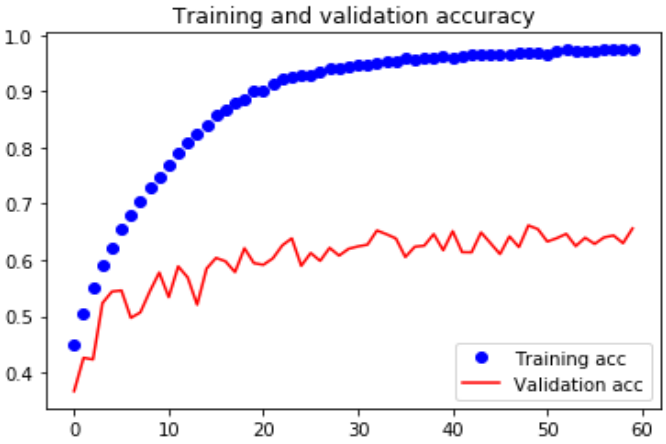
	Benchmark CNN	Bottleneck Features	Fine tuning	WRN 28-8
Fine	44.30	76.41	61.93	65.63
Coarse	56.40	84.87	75.24	-

Epochs	100	30	20	60

Benchmark models mainly report and focus on accuracies of fine labels and we shall do the same here. All future references of performance comparison shall be in the context of fine labels only. Validation accuracy and training accuracy plots for bottleneck, fine tuning and WRN are shown in respective order below.







The models have been trained on FloydHub Tesla K80 free GPU version and Paperspace P4000 paid GPU machine. The approximate training time in seconds of the models are listed in table below.

	Benchmark CNN	Bottleneck	Fine Tuning	WRN 28-8
Tesla K80	23	-	-	-
P4000	-	15	397	315

All the three models result in higher validation accuracies in less number of epochs than the naive benchmark CNN model. Training time is high for fine tuning and wrn model but bottleneck has comparable training time per epoch to the benchmark. Fine tuning starts to overfit with very less number of epochs.

Both benchmark CNN model and the best model of bottleneck feature extraction were trained using k-fold cross validation with k = 6 for fine labels and the validation accuracy in percentage are tabulated below:

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	mean	std
Benchmark CNN	40.42	42.47	40.54	41.40	41.56	41.11	41.25	0.69
Bottleneck feature extraction	77.09	76.36	76.40	76.36	76.32	77.54	76.68	0.47

Justification

The validation accuracies along with the plots show that bottleneck feature transfer learning achieves the highest accuracy without overfitting in the least amount of epochs. WRN has less performance because we only trained for 60 epochs. [23] has reported over 82% validation accuracy for fine labels using learning rate decay and trained for about 100-200 epochs. We trained it to compare its training time with other techniques.

Comparison table between benchmark CNN and bottleneck feature extraction show clearly that the transfer learning technique using bottleneck features outperform the basic CNN benchmark model both in accuracy and computational cost as well as in the number of epochs required.

	6-fold validation accuracy in percentage	Training time in seconds per epoch	Number of epochs
Benchmark CNN	41.25 +/- 0.69	15-20	100
Bottleneck feature extraction	76.68 +/- 0.47	10-15	30

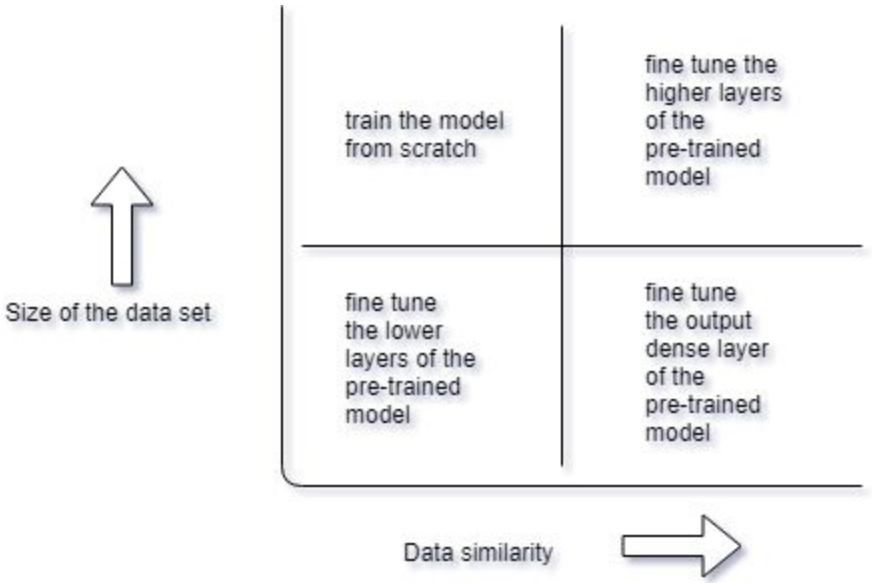
The results show that transfer learning with bottleneck features is the best method to use in our case of CIFAR-100 dataset as it achieves performance comparable to many good benchmark models[12] with few epochs and less time. The computation could even be performed with less computational resources like a CPU as is evident from the training times. Fine tuning and WRN would require computational machines like GPUs and take over 5 mins per epoch even in

GPUs. The results match the concepts of bottleneck features and fine tuning as explained in references [14]. Keras implementation details referred from [24] - [27].

Conclusion

Visualisation

The use case scenarios of transfer learning can be summarised by the following diagram.



In this project, we have explored the two techniques on the right quadrants. Our input dataset CIFAR-100 using pre-trained model on ImageNet resembles the lower right quadrant of the diagram. We have tried both transfer learning using bottleneck features(lower right quadrant technique) and fine tuning the higher layers(upper right quadrant technique) of the model. The results obtained justify the appropriate technique presented above and gives us a taste of practical scenario of the theory. We may apply other techniques enumerated depending on the size of pre-trained data set and our input data set.

Reflection

[14] presents the theory and concepts of training models on new datasets and how to approach them. It enumerates the possible scenarios and possible techniques to apply depending on the size and characteristics of dataset. It advises how to take advantage of models and weights trained by other people and leverage that to our use cases. We have implemented the first two techniques of bottleneck features or CNN codes and fine tuning to evaluate how they fare in practice. The CIFAR-100 dataset has much smaller number of images than the ImageNet but

similar goal and characteristics. So, it falls under the category of CNN codes. Our results show the applicability of the techniques and matches the theory. It also validates the second point of fine tuning overfitting very quickly on the small CIFAR-100 dataset.

Improvement

For further improvement in validation accuracy of the best technique, transfer learning with bottleneck features, we could try other classifiers like SVM. Also we could train for more epochs and use other regularisation techniques or different values of dropout to prevent overfitting. We could experiment with other optimizers and different values of hyperparameters like learning rate. There are many hyperparameters which could be tuned to come up with higher performance.

References:

1. https://en.wikipedia.org/wiki/Computer_vision
2. LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998d). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324.
3. Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009.
4. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.
5. Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." European Conference on Computer Vision. Springer International Publishing, 2014.
6. Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556(2014).
7. Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015.
8. He, Kaiming, et al. "Deep residual learning for image recognition." arXiv preprint arXiv:1512.03385 (2015).
9. https://www.cs.toronto.edu/~kriz/cifar.html
10. http://groups.csail.mit.edu/vision/TinyImages
11. Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.
12. CIFAR-100 benchmarks https://benchmarks.ai/cifar-100
13. https://github.com/lapjarn/capstone_udactiy
14. http://cs231n.github.io/transfer-learning/
15. Keras examples https://github.com/keras-team/keras/tree/master/examples
16. https://github.com/guillaume-chevalier/caffe-cifar-10-and-cifar-100-datasets-preprocessed-to-HDF5
17. https://www.floydhub.com/lapjarn/projects/cifar100_capstone/5
18. https://www.floydhub.com/lapjarn/projects/cifar100_capstone/6

19. https://keras.io/applications/
20. Wide Residual Networks (BMVC 2016) http://arxiv.org/abs/1605.07146 by Sergey Zagoruyko and Nikos Komodakis.
21. https://github.com/keras-team/keras-contrib
22. https://github.com/titu1994/Wide-Residual-Networks
23. Torch implementation of WRN https://github.com/szagoruyko/wide-residual-networks
24. Keras blog at https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
25. CIFAR-10 experiments https://github.com/rnoxy/cifar10-cnn
26. https://www.learnopencv.com/keras-tutorial-transfer-learning-using-pre-trained-models/
27. https://www.learnopencv.com/keras-tutorial-fine-tuning-using-pre-trained-models/
28. Rezende, Edmar & Ruppert, Guilherme & Carvalho, Tiago & Ramos, Fabio & De Geus, Paulo. (2017). Malicious Software Classification Using Transfer Learning of ResNet-50 Deep Neural Network. 10.1109/ICMLA.2017.00-19.